Bigtable: A Distributed Storage System for Structured Data

Charul Daudkhane
charul.daudkhane@campus.tu-berlin.de
BDASEM 2018
DIMA – TU BERLIN Student

Report on the Paper
“Bigtable: A Distributed Storage System for Structured data” [1]

ABSTRACT

Needless to say, Google is the most powerful web search engine which provides access to billions of documents over 30 different languages.
Apart from being the web search engine, it also provides a number to other services like Gmail, Google Maps, Google Analytics, Google Finance etc.. One of the key goals of google is to organize the world’s information in order to make it universally accessible and useful.
In order to provide such a service with minimum access latency, they needed a flexible, high-performance distributed storage system.
Google, in 2005, released Bigtable, as a structured, distributed, storage system for managing the structures/semi-structured data for its various applications. The design and structure of Bigtable kickstarted the NO-SQL industry and lead to the development of HBase, Casandra, MongoDB etc. This paper summarizes the data model, components, implementation, recent works and the positives and negatives of the original Bigtable paper published by Google.

1. Introduction

Google’s broad spectrum of applications enforces different data requirements on the underlying system. For example, the web indexing applications are required to store the URL content of a page like anchors, contents, links, page rank etc.... whereas the geographical applications are required to store the information about physical entities like shops, roads, distance between two points etc. Clearly some of this data is structured and some semi-structured. Moreover, in mid 2000’s Googles web indexes behind the search engines took a long time to build, thus impacting latency. In order to overcome the above shortcomings, Google came up with one common solution: Bigtable.
Bigtable is a multi-dimensional, distributed, highly available and scalable database storage system. Bigtable clusters span over thousands of servers with different configuration setting per application. It doesn't support a full relational data model but it provides control over the data layout and format via sparse, distributed sorted map. Data in Bigtable is stored in the form of rows and columns and indexed by the row and column keys. Initially Bigtable was only for Google’s inhouse application. In 2015, Bigtable was released as database as a service via Cloud Bigtable.

2. Data Model

The paper describes the data model as a sparse, distributed, persistent multi-dimensional sorted map. This map is indexed by a row key, column key and timestamp.

Row-
Rows of the Bigtable are indexed by row keys. The data in the Bigtable is sorted lexicographically on the basis of row keys. All the operations on a single row are atomic. Some example of rows keys could be – reversed domain names, multiple values separated by a delimiter, etc.
Rows are dynamically partitioned into *tablets* which is a unit of distribution and load balancing. Tablets are managed by tablet servers.

Column-
Columns of a Bigtable are indexed by column key of the form *family :optional_qualifier*. Data in the same column family is of the same time and compressed together. Access control is performed at the column-family level.
The Webtable in Figure 1, shows *anchor:cnnsi.com* storing the link text addressed by the anchor tag in *cnnsi.com*, the content’s column family storing the content of the web page addressed by the row key *com.cnn.www*.

Timestamp-
Cells in Bigtable can contain multiple versions of data. These versions are indexed by timestamps which can be either real time or assigned by client. Bigtable cells store data in decreasing timestamp order. The Webtable in Figure 1, shows value stored in the contents column at three different timestamps (t3,t5,t6)

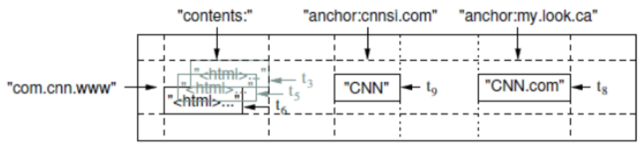


Figure 1 : A Bigtable storing Web page information with the row keys as reversed URL names. The column keys store different attributes of a web page like contents, pagerank, anchor tags etc. The contents column stores information at 3 different timestamps.

3. Building Blocks

The building blocks of Bigtable include *Google File System* for storing logs and data files. These logs are stored in a special file format called the *Sorted String Table Format(SSTable).*

An SSTable stores the Bigtable key-value pair in a sequence of 64KB blocks and has a block index file for locating these blocks. In order to ensure consistency, Bigtable uses a highly available and distributed lock service called *Chubby*. It consists of directories and files which are used as locks. Chubby is used for electing the master, helping master discover live tablet servers, storing the bootstrap location of the Bigtable data etc... If chubby becomes unavailable, Bigtable becomes unavailable.

4. API

Bigtable is accessible via the client API which allows to perform metadata operations like creating, deleting tables, column families. It allows to perform writes in a Bigtable cell via *set(), deleterows(), deletecells()*. It also allows to read the Bigtable cells via *scanner()*.

5. Implementation

The Bigtable architecture consists of a master, multiple tablet servers and a client library at its backbone. Initially the entire Bigtable is one tablet, as rows are added to this table, it is split into multiple tablets. The typical tablet size is 100-200MB.
The *master* is responsible for assigning tablets to tablet servers, detecting live or dead tablet servers, performing load balancing (moving the tablets between servers to balance the load), performing garbage collection and managing schema changes.
Rows are dynamically partitioned into tablets which are served by the *tablet servers*. These servers are responsible for handling the read write request to the tablets and splitting the tablets when they gown in size. According to the paper [1], one tablet server can handle between ten to thousand tablets.
The clients interact directly with the tablet servers through the client library. In order to perform a read/write request to the Bigtable, the client first locates the tablet server serving the rows it wants to read/write. This is done via the chubby lock service [2] explained in Section 4.1
Upon identification, the request is directed to the correct tablet server which in turn contacts the Google File System to process the data.

5.1 Tablet Location

The location information of all the tablets managed by the Bigtable Cluster is stored in a three-level hierarchy similar to a B+ tree.
The client first contacts chubby and gets the tablet location via the three-level hierarchy. At the first level is a Chubby file which stores the information of the root tablet. This tablet never splits and contains the location of the Metadata tablets.

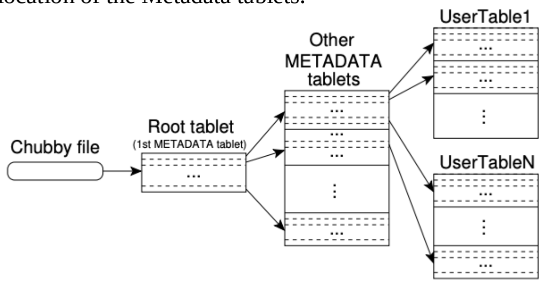


Figure 2: The three-level tablet location hierarchy

The row keys of the metadata tablets are made up of the table identifier and end rows which then points to the user table. In order to avoid this three-level look up for every request, the client library caches the tablet location. However, when the tablets have moved and the cache points to the old location, it needs six network round trips to reach to the new location.

5.2 Tablet Assignment

Bigtable uses Chubby to keep track of the tablet servers. Upon initialization of the tablet server, it acquires a lock on a uniquely named file in the chubby directory. The master monitors this chubby directory to keep track of live tablet servers. Upon loss of this lock, the tablet server stops serving the tablets. When the tablet server loses its lock due to network partitioning, it tries to reacquire the lock but on failure, kills itself as it no longer has access to the uniquely named file in the directory.

When the tablet server terminates (due to cluster management releasing the tablet server machine), it releases its lock for the master to reassign the tablets it was serving. In order for the master to know about any tablet's termination, it regularly queries the tablet server for the status of the lock.

The chubby lock service maintains five active replicas, one of which is elected as the master. So, when a master server is started by the cluster management system, it grabs a unique master lock in the chubby directory. When this lock is lost, the new master is elected form one of the four remaining replicas.

5.3 Tablet Serving

In order to service the read/write request from the client, the tablet server looks for the correct tablet which maintains – *a commit log*: to store recent updates on disk, a *memtable buffer*: to store the recent updates memory, *a sequence of SSTable*: to store older updates on disk.

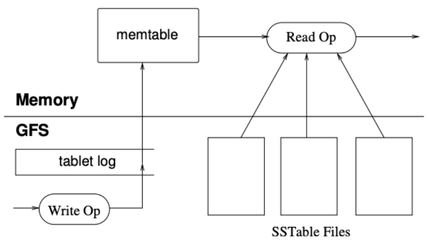


Figure 3: Tablet Representation

When a read/write request arrives, the tablet server checks if the sender is authorized and the request is well formed. The writes are first written to the commit log followed by the memtable. Writing to the commit log helps during recovery. For the reads, the results are returned from the merged view of the memtable and SSTable. This merging is needed as the SSTable hold the old updates and memtable the recent updates.

5.4 Compaction

When the size of the memtable reaches a threshold, a *minor compaction* is performed where a new memtable is created and the contents of the old memtable are converted to a new SSTable and written to GFS. Doing so reduces the memory usage of the tablet server and the amount of data to be read by the commit log during recovery. Minor compaction, leads to creation of multiple SSTables which are merged together into one SSTable. This operation is called *major compaction*, which is done to remove the deleted data from the older SSTables.

6 Refinements

In order to achieve high performance and reliability, a number of refinements were made while implementing Bigtable.

Locality Groups-
The concept of locality group enables efficient reads. It allows clients to group multiple column families into one locality group. A separate SSTable can be generated for each of these locality groups. For example, if a webpage application wants to access only the page rank of a webpage, it can be grouped into a separate locality groups than with the contents so that reads through all the contents can be avoided.

Compression-
Clients can specify if the SSTables for the locality groups have to compressed and the compression format. SSTables can be read without decompressing them completely thus balancing for the space reduction during compression. A typical compression scheme used by clients is called a two-pass scheme where the first pass is based on Bentley and Mcllroy's scheme [3] and the second pass uses a fast compression algorithm.

Bloom Filters-
When a read request arrives for a tablet, the result is returned after reading all the SSTables which form the tablet. Since these SSTables may not be in memory, additional I/O is needed to load them and check for the requested data. In order to reduce the number of missed reads, clients can allow the creation of Bloom filters for a locality group. Bloom filters directly specifies if the requested row/column pair is present in a particular SSTable thus reducing the number of reads to the accesses.

Commit Log Implementation-
As mentioned in section 5.3, a sperate commit log is maintained for each tablet. As the number of tablets can increase depending on the load, maintaining too many commit logs can become a bottleneck. Thus, one such refinement is to maintain one commit log file per tablet server and save the entries in the form <table, row name, log sequence numbers> for quick identification.

7. Performance Evaluation

The performance of Bigtable was tested on N tablet servers with N being varied from 1 to 500, on six different benchmarks: scans, random reads from memory, random writes, sequential reads, sequential writes and random reads from disk. All these benchmarks were executed such that the tablet servers, masters, test clients and GFS servers all ran on the same set of machines. The results of the performance of the benchmarks when reading and writing 1000 bytes values to Bigtable were as shown in Figure 4.

For the performance with one tablet server, random reads performed the worst. The main reason is because in random reads, 64KB of SSTable blocks are transferred over the network from which only 1000 bytes are read, thus saturating the CPU bandwidth. However, in case of sequential reads, the next 64kb blocks to be read were already cached and thus not saturating the network.

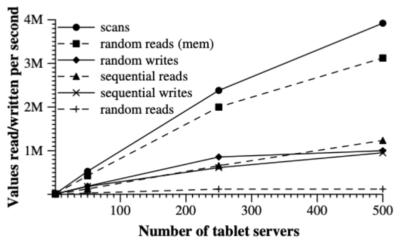
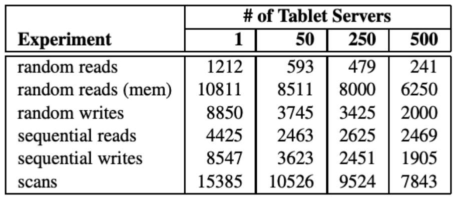


Figure 4: Performance comparison of six benchmarks while reading and writing 1000 bytes to Bigtable



Experiment	# of Tablet Servers 1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

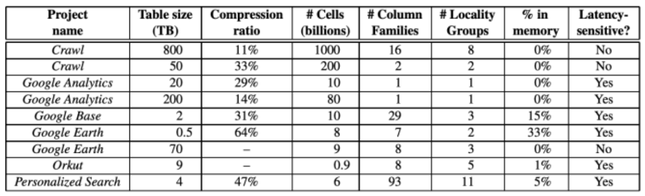
Figure 5: Performance comparison of six benchmarks while reading and writing 1000 bytes to Bigtable with variation in the number of tablet servers

As the number of tablet servers increase to 500, the performance of random read from memory increases by a factor of 300. However, the performance of the other benchmarks decreases with increase in the servers due to load imbalancing. The performance of random reads is the lowest due to the transfer of 64KB blocks over the network as above

8. Real Applications

Bigtable is currently used by more than 60 google products as the underlining distributed storage system. The paper [1] mentions, Google Earth, Personalized Search, Google Analytics as some of its application out of which Google Analytics is described below.

Google Analytics Application-
It provides services which helps the web master analyze web traffic patterns like, number of visitors/days, number of purchases made, number of page views in a day etc.
Google Analytics uses two Bigtables: *Raw Click table* (approx. 200 TB) which records the click information and maintains a row for every end-user session with the row key of the form *website_name+session_creation_time* and *Summary table* (approx. 20TB) which is generated by running MapReduce[6] jobs on the raw clicks table. It maintains the summary of each website.
Few other working Bigtables in production at Google are summarized in Figure 6



Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

Figure 6: Bigtables in production use at Google

9. Strengths and Limitations

The paper introduces strong concepts about distributed storage system so much so that a lot of NO-SQL databases were created based on this paper.
In my opinion, the most interesting aspect of the paper is Bigtable's data model and the way in which the data is partitioned into tablets for distributed processing. The data model created easily fits to various types of applications. The fine line of separating the storage (GFS) and the logic makes it easy to maintain. The paper describes how the use of Chubby lock service ensures consistent state of all the tablet servers to the master.

Since the clients, directly interact with the tablet servers without going through the master, there is no single point of failure.
The paper does not explain how Bigtable deals with multiple read request to a small number of rows. In the case of multiple writes, rows are divided into tablets but for increasing reads requests, nothing has been mentioned. The paper does not talk about the way of identify the authenticity of the request during the tablet read write operations. Though the use of one commit log file per tablet server saves a lot of disk space, however during recovery a lot of log analysis has to be done which the paper does not account for.

11. Recent Developments

In 2015 Google released Bigtable, as database-as-a-service to public. Until then it was only used for Google's internal use. Since its release,
A lot of languages have been added to its client library. It currently supports Nodejs, C++, Python.
Cloud Bigtable integrates easily with popular big data tools like Hadoop, Cloud Dataflow, and Cloud Dataproc. Some of the main applications of cloud Bigtable includes, financial analysis, IOT and AdTech.

12. Conclusion

Bigtable is a massively scalable, high performance, cost effective distributed system aimed at storing structured data at Google. Bigtable features robust scalability to petabytes of data and thousands of machines. A number of database implementation strategies of conventional database are utilized in Bigtable, such as parallel databases [5] and main-memory databases [4].Bigtable is also available for external users. Data in Bigtable is indexed by a time stamp along with arbitrary strings of row and column keys. Bigtable's simple data model allows user to control the layout and format of data dynamically. The schema of Bigtable also allows end user to fully control the locality of their data.

13. References

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," 7th Symp. Oper. Syst. Des. Implement. (OSDI '06), Novemb. 6-8, Seattle, WA, USA, pp. 205– 218, 2006.

[2] M. Burrows, "The Chubby lock service for loosely coupled distributed systems," OSDI '06 Proc. 7th Symp. Oper. Syst. Des. Implement. SE - OSDI '06, pp. 335–350, 2006.

[3] BENTLEY, J. L., AND MCILROY, M. D. Data compression using long common strings. In Data Compression Conference (1999), pp. 287–295

[4] DEWITT, D., KATZ, R., OLKEN, F., SHAPIRO, L., STONEBRAKER, M., AND WOOD, D. Implementation techniques for main memory database systems. In Proc. of SIGMOD (June 1984), pp. 1–8.

[5] DEWITT, D. J., AND GRAY, J. Parallel database systems: The future of high performance database systems. CACM 35, 6 (June 1992), 85–98.

[6] J. Dean and S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters," Proc. 6th Symp. Oper. Syst. Des. Implement., pp. 137–149, 2004.