



# DISTRIBUTED STORAGE SYSTEMS – GOOGLE BIGTABLE

Mohamed Faadil Shaikh

BOSTON UNIVERSITY METROPOLITON COLLEGE CS-777



## Table of Contents

<b>1. Introduction.....</b>	<b>2</b>
<b>2. Data Model .....</b>	<b>3</b>
a) Rows .....	4
b) Columns .....	4
c) Timestamps .....	5
<b>3. Bigtable architecture .....</b>	<b>6</b>
<b>4. Load Balancing .....</b>	<b>7</b>
<b>5. API's .....</b>	<b>8</b>
<b>6. Building Blocks .....</b>	<b>8</b>
<b>7. Implementation .....</b>	<b>10</b>
<b>8. Startup and Growth.....</b>	<b>11</b>
<b>9. Code Samples.....</b>	<b>12</b>

## 1. Introduction

A new era started at the beginning of the 21st century – the Digital Era. Most things now become digital or heavily dependent on technology – starting with things like radio and TV, going through healthcare, even most of our memories. Between 1986 and 2007 the amount of data per person has been growing with 23% per year. As a result, there is a huge amount of digital data which is created daily and accumulates to unseen amounts.

Storing data has evolved during the years to accommodate the rising needs of companies and individuals. We are now reaching a tipping point at which the traditional approach to storage – the use of a stand-alone, specialized storage box – no longer works, for both technical and economic reasons. We need not just faster drives and networks, we need a new approach, a new concept of doing data storage. At present, the best approach to satisfying current demands for storing data seems to be distributed storage.

We can summarize distributed storage systems as.

“Storing data on a multitude of standard servers, which behave as one storage system although data is distributed between these servers.”

In this paper we will be focusing on Google BigTable. It is a fully managed distributed storage system, scalable NoSQL database service for large analytical and operational workloads with up to 99.9999% availability. It was designed to support applications requiring massive scalability, the technology was intended to be used with petabytes of data.









Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

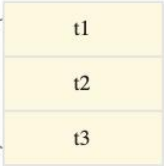
## 2. Data Model

Bigtable treats all data as raw byte strings for most purposes. The only time Bigtable tries to determine the type is for increment operations, where the target must be a 64-bit integer encoded as an 8-byte big-endian value.

Bigtable stores data in massively scalable tables, each of which is a sorted key/value map. The table is composed of rows, each of which typically describes a single entity, and columns, which contain individual values for each row. Each row is indexed by a single row key, and columns that are related to one another are typically grouped into a column family. Each column is identified by a combination of the column family and a column qualifier, which is a unique name within the column family.

Each row/column intersection can contain multiple cells. Each cell contains a unique timestamped version of the data for that row and column. Storing multiple cells in a column provides a record of how the stored data for that row and column has changed over time. Bigtable tables are sparse; if a column is not used in a particular row, it does not take up any space.

	Column family 1		Column family 2	
	<i>Column 1</i>	<i>Column 2</i>	<i>Column 1</i>	<i>Column 2</i>
Row key 1				  
Row key 2				



A few things to notice in this illustration:

- Columns can be unused in a row.
- Each cell in each row and column has a unique timestamp (t).

## a) Rows

The row keys in a table are arbitrary strings (currently up to 64KB in size, although 10-100 bytes is a typical size for most of our users). Every read or write of data under a single row key is atomic (regardless of the number of different columns being read or written in the row).

Bigtable maintains data in lexicographic order by row key. Lexicographical order is nothing but the dictionary order or preferably the order in which words appear in the dictionary. The row range for a table is dynamically partitioned. Each row range is called a tablet, which is the unit of distribution and load balancing. As a result, reads of short row ranges are efficient and typically require communication with only a small number of machines.

## b) Columns

Column keys are grouped into sets called column families, which form the basic unit of access control. All data stored in a column family is usually of the same type (we compress data in the same column family together). A column family must be created before data can be stored under any column key in that family; after a family has been created, any column key within the family can be used.

A column key is named using the following syntax: family:qualifier. Column family names must be printable, but qualifiers may be arbitrary strings. Access control and both disk and memory accounting are performed at the column-family level.

The diagram illustrates the Bigtable data structure. It shows a table with row keys and column families. The row keys are: com.aaa, com.cnn.www, com.cnn.www/TECH, and com.weather. The column families are: "language:", "contents:", anchor:cnnsi.com, and anchor:mylook.ca. The data is organized into a table with 4 rows and 4 columns. The first column contains the row keys. The second column contains the "language:" family data. The third column contains the "contents:" family data. The fourth column contains the "anchor:cnnsi.com" family data. The fifth column contains the "anchor:mylook.ca" family data. The row keys are sorted in lexicographic order, as indicated by the "Sorted rows" label and arrow.

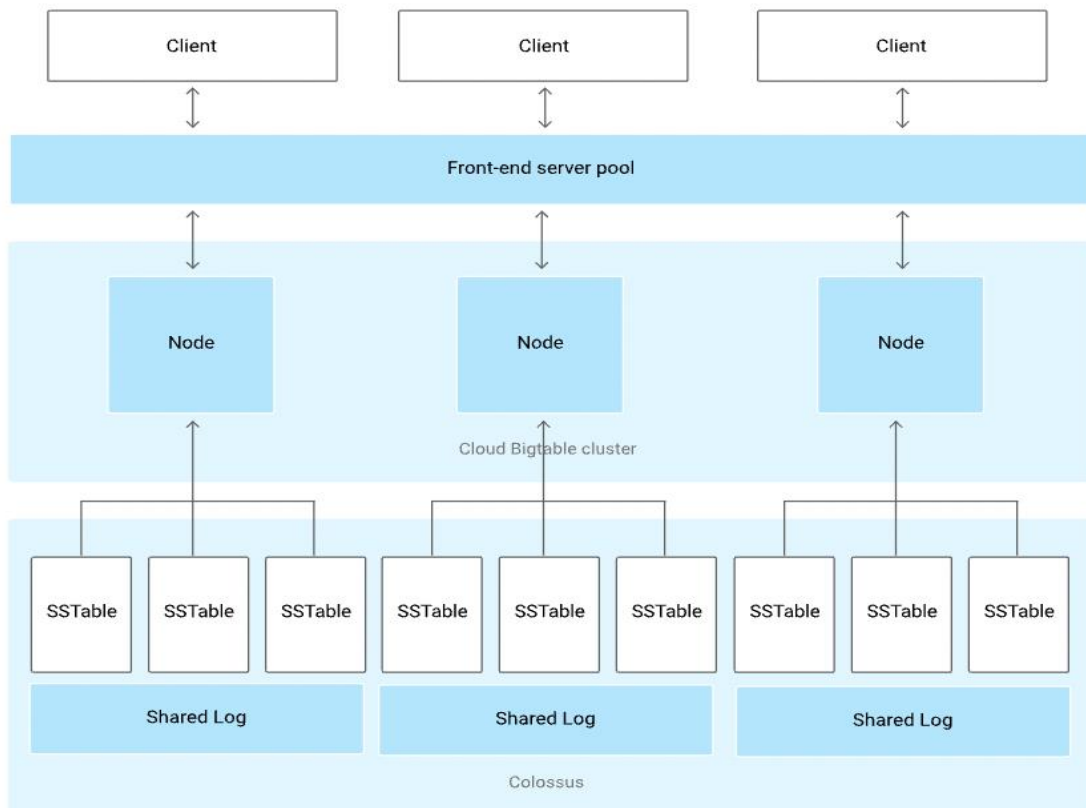
row keys	column family "language:"	column family "contents:"	column family anchor:cnnsi.com	column family anchor:mylook.ca
com.aaa	EN	<!DOCTYPE html PUBLIC...		
com.cnn.www	EN	<!DOCTYPE HTML PUBLIC...	"CNN"	"CNN.com"
com.cnn.www/TECH	EN	<!DOCTYPE HTML>...		
com.weather	EN	<!DOCTYPE HTML>...		

### c) Timestamps

Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by timestamp. Bigtable timestamps are 64-bit integers. They can be assigned by Bigtable, in which case they represent “real time” in microseconds or be explicitly assigned by client applications.

Applications that need to avoid collisions must generate unique timestamps themselves. Different versions of a cell are stored in decreasing timestamp order, so that the most recent versions can be read first.

### 3. Bigtable architecture



As the diagram illustrates, all client requests go through a frontend server before they are sent to a Bigtable node. The nodes are organized into a Bigtable cluster, which belongs to a Bigtable instance, a container for the cluster.

Each node in the cluster handles a subset of the requests to the cluster. By adding nodes to a cluster, you can increase the number of simultaneous requests that the cluster can handle. Adding nodes also increases the maximum throughput for the cluster. If you enable replication by adding additional clusters, you can also send different types of traffic to different clusters. Then if one cluster becomes unavailable, you can fail over to another cluster.

A Bigtable table is sharded into blocks of contiguous rows, called tablets, to help balance the workload of queries. (Tablets are like HBase regions.) Tablets are stored on Colossus, Google's file system, in SSTable format. An

SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Each tablet is associated with a specific Bigtable node. 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.

Importantly, data is never stored in Bigtable nodes themselves; each node has pointers to a set of tablets that are stored on Colossus. As a result:

- Rebalancing tablets from one node to another happens quickly, because the actual data is not copied. Bigtable simply updates the pointers for each node.
- Recovery from the failure of a Bigtable node is fast, because only metadata must be migrated to the replacement node.
- When a Bigtable node fails, no data is lost.

#### 4. Load Balancing

Each Bigtable zone is managed by a primary process, which balances workload and data volume within clusters. This process splits busier/larger tablets in half and merges less accessed/smaller tablets together, redistributing them between nodes as needed. If a certain tablet gets a spike of traffic, Bigtable splits the tablet in two, then moves one of the new tablets to another node. Bigtable manages the splitting, merging, and rebalancing automatically, saving you the effort of manually administering your tablets

To get the best write performance from Bigtable, it's important to distribute writes as evenly as possible across nodes. One way to achieve this goal is by using row keys that do not follow a predictable order. For example, usernames tend to be distributed evenly throughout the alphabet, so including a username at the start of the row key will tend to distribute writes evenly.

At the same time, it's useful to group related rows so they are next to one another, which makes it much more efficient to read several rows at the same time.



For example, if you're storing different types of weather data over time, your row key might be the location where the data was collected, followed by a timestamp (for example, WashingtonDC#201803061617). This type of row key would group all of the data from one location into a contiguous range of rows. For other locations, the row would start with a different identifier; with many locations collecting data at the same rate, writes would still be spread evenly across tablets.

## 5. API's

The Bigtable API provides functions for creating and deleting tables and column families. It also provides functions for changing cluster, table, and column family metadata, such as access control rights.

Bigtable supports several other features that allow the user to manipulate data in more complex ways.

1. Bigtable supports single-row transactions, which can be used to perform atomic read-modify-write sequences on data stored under a single row key. Bigtable does not currently support general transactions across row keys, although it provides an interface for batching writes across row keys at the clients.
2. Bigtable allows cells to be used as integer counters.
3. Bigtable supports the execution of client-supplied scripts in the address spaces of the servers. The scripts are written in a language developed at Google for processing data called Sawzall.

Bigtable can be used with MapReduce, a framework for running large-scale parallel computations developed at Google.

## 6. Building Blocks

Bigtable is built on several other pieces of Google infrastructure. Bigtable uses the distributed Google File System (GFS) to store log and data files. A Bigtable cluster typically operates in a shared pool of machines that run a wide variety of other distributed applications, and Bigtable processes often share the same machines with processes from other applications. Bigtable depends on a cluster management system for scheduling jobs, managing

resources on shared machines, dealing with machine failures, and monitoring machine status.

Bigtable relies on a highly available and persistent distributed lock service called Chubby. A Chubby service consists of five active replicas, one of which is elected to be the master and actively serve requests. The service is live when most of the replicas are running and can communicate with each other. Chubby uses the Paxos algorithm to keep its replicas consistent in the face of failure. Chubby 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. The Chubby client library provides consistent caching of Chubby files. Each Chubby client maintains a session with a Chubby service. A client's session expires if it is unable to renew its session lease within the lease expiration time. When a client's session expires, it loses any locks and open handles. Chubby clients can also register callbacks on Chubby files and directories for notification of changes or session expiration.

To summarize in Bigtable Chubby is used for;

- ensure there is only one active master
- store the bootstrap location of Bigtable data
- discover tablet servers
- store Bigtable schema information
- store access control lists

## 7. Implementation

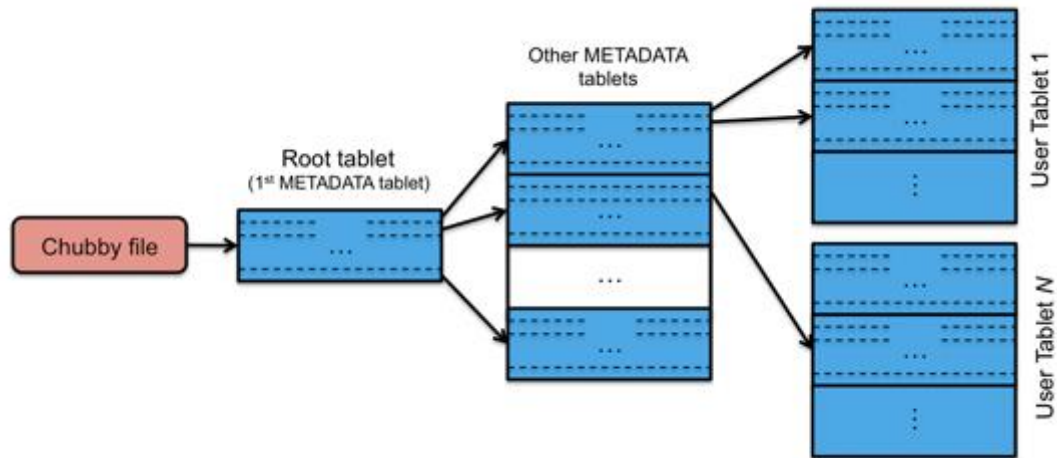
Bigtable comprises a client library (linked with the user's code), a master server that coordinates activity, and many tablet servers. Tablet servers can be added or removed dynamically. If the master dies, another master can take over.

The master assigns tablets to tablet servers and balances tablet server load. It is also responsible for garbage collection of files in GFS and managing schema changes (table and column family creation). As such, a tablet server is responsible for tablets, but the tablets do not necessarily live on that node since any node can access any data within GFS.

Each tablet server manages a set of tablets (typically 10–1,000 tablets per server). It handles read/write requests to the tablets it manages and splits tablets when a tablet gets too large. Client data does not move through the master; clients communicate directly with tablet servers for reads/writes. The internal file format for storing data is Google's SSTable, which is a persistent, ordered, immutable map from keys to values. Rows of data are always kept sorted by the row key.

Bigtable uses the Google File System (GFS) for storing both data files and logs. A cluster management system contains software for scheduling jobs, monitoring health, and dealing with failures.

## 8. Startup and Growth



A table starts off with just one tablet. As the table grows, it is split into multiple tablets. By default, a table is split at around 100 to 200 MB.

Locating rows within a Bigtable is managed in a three-level hierarchy. The root (top-level) tablet stores the location of all **Metadata tablets** in a special **Metadata table**. This root tablet is simply the first tablet in the set of tablets that comprise the Metadata table. Each Metadata table contains the location of user data tablets. This table is keyed by node IDs and each row identifies a tablet's table ID and end row. For efficiency, the client library caches tablet locations.

A tablet is assigned to one tablet server at a time. Chubby keeps track of tablet servers. When a tablet server starts, it creates and acquires an exclusive lock on a uniquely named file in a Chubby *server's* directory. The master monitors this directory to discover new tablet servers.

When the master starts, it:

- Grabs a unique master lock in Chubby (to prevent multiple masters from starting)
- Scans the server's directory in Chubby to find live tablet servers
- Communicates with each tablet server to discover what tablets are assigned to each server. This is important because the master might be recovering for a failed master and tablets have already been allocated to tablet servers.
- Scans the Metadata table to learn the full set of tablets
- Builds a set of unassigned tablet servers. These are eligible for tablet assignment and the master will choose a tablet server and send it a tablet load request.

## 9. Code Samples

I am using python to in the below examples. Many other languages likes scala, Java, Go etc.

This example uses the Cloud Bigtable package of the Google Cloud Client Library for Python to communicate with Bigtable.

```
"""Demonstrates how to connect to Cloud Bigtable and run some basic operations.
```

Prerequisites:

- Create a Cloud Bigtable cluster.  
<https://cloud.google.com/bigtable/docs/creating-cluster>
  - Set your Google Application Default Credentials.  
<https://developers.google.com/identity/protocols/application-default-credentials>
- ```
"""
```

```
## Importing Modules
```

```
import datetime
from google.cloud import bigtable
from google.cloud.bigtable import column_family
from google.cloud.bigtable import row_filters
```

```
## Connect to Bigtable using a bigtable.Client.
```

```
# The client must be created with admin=True because it will create a table.
```

```
client = bigtable.Client(project=project_id, admin=True)
instance = client.instance(instance_id)
```

## Instantiate a table object using Instance.table(). Create a column family and set its garbage collection policy, then pass the column family to Table.create() to create the table.

```
print("Creating the {} table.".format(table_id))
table = instance.table(table_id)

print("Creating column family cf1 with Max Version GC rule...")
# Create a column family with GC policy : most recent N versions
# Define the GC policy to retain only the most recent 2 versions

max_versions_rule = column_family.MaxVersionsGCRule(2)

column_family_id = "cf1"

column_families = {column_family_id: max_versions_rule}

if not table.exists():
    table.create(column_families=column_families)
else:
    print("Table {} already exists.".format(table_id))
```

## Writing rows to a table

```
greetings = ["Hello World!", "Hello Cloud Bigtable!", "Hello Python!"]
rows = []
column = "greeting".encode()
for i, value in enumerate(greetings):
    row_key = "greeting{}".format(i).encode()
    row = table.direct_row(row_key)
    row.set_cell(
        column_family_id, column, value, timestamp=datetime.datetime.utcnow()
    )
    rows.append(row)
table.mutate_rows(rows)
```