**Isaac Araya Solano | Carnet: 2018151703**

**Resumen #8**

# Data Model

Bigtable is a distributed map indexed by row key, column key, and timestamp. It stores arrays of bytes as values. It is designed for various applications, such as storing web pages in a table called Webtable using URLs as row keys and different aspects of web pages as column names.

## Rows

In Bigtable, row keys are arbitrary strings and operations on a single row key are atomic. Data is stored in lexicographic order by row key, and the table is divided into tablets for distribution and load balancing. Reading short row ranges is efficient, and clients can optimize data access by selecting row keys for locality. For example, in Webtable, pages from the same domain are grouped together by reversing hostname components in the row keys.

## Column Families

In Bigtable, column keys are grouped into column families for access control. Data within a column family is of the same type and can be compressed together. Column families must be created before storing data, and a table can have a large number of columns. Column keys follow the format "family:qualifier." Access control and accounting are done at the column-family level, enabling different types of applications to manage and view data with specific restrictions.

## Timestamps

In Bigtable, cells can have multiple versions indexed by timestamps. Timestamps can be assigned by Bigtable or the client. The latest versions are stored first. Bigtable provides settings for automatic garbage collection, allowing the retention of a specific number of versions or only recent versions. In the Webtable example, only the three most recent versions of crawled pages are kept.

# API

The Bigtable API allows managing tables, columns, and metadata. Clients can read, write, and perform atomic updates. Bigtable supports transactions, counters, and data processing with Sawzall. It integrates with MapReduce and offers input/output wrappers for parallel computations.

# Building Blocks

Bigtable uses Google's infrastructure components like GFS, cluster management, and Chubby for storage, scheduling, and lock service. It employs the SSTable file format for efficient data storage. Chubby ensures consistency and access control through Paxos. Chubby's unavailability has minimal impact on Bigtable, affecting only a small fraction of server hours across clusters.

# Implementation

Bigtable has three main components: a client library, a master server, and multiple tablet servers. The master server handles tablet assignments, load balancing, and schema changes. Tablet servers manage tablets for read and write operations, including automatic splitting of growing tables. Clients directly communicate with tablet servers, reducing load on the master. Tables are divided into tablets that dynamically adjust in size.

### Tablet Location

Bigtable uses a hierarchical location scheme with tablets. The root tablet in Chubby stores location information for all tablets in the METADATA table. Caching and prefetching techniques optimize tablet location retrieval. The METADATA table also holds additional data like event logs for debugging and performance analysis.

### Tablet Assignment

In Bigtable, tablets are assigned to tablet servers. The master tracks live servers and their assignments. Chubby is used to monitor tablet servers. If a server loses its lock, it stops serving and tries to reacquire it. The master periodically checks server status and reassigns tablets as needed. At startup, the master scans servers and METADATA table to learn about tablets. Tablet assignments change with table operations. Splits are initiated by servers and confirmed with the master.

### Tablet Serving

In Bigtable, a tablet's state is stored in GFS. Updates are recorded in a commit log and stored in memory as a memtable or in SSTables. To recover a tablet, the server reads metadata from the METADATA table and reconstructs the memtable using the commit log and SSTables. Write operations are authorized through Chubby and stored in the commit log and memtable. Read operations are executed on a merged view of SSTables and the memtable. Tablets can be split or merged while allowing incoming read and write operations to continue.

### Compactions

Bigtable uses a memtable to store write operations, which is periodically converted into SSTables through minor compactions. Merging compactions combine SSTables and the memtable to reduce the number of files. Major compactions consolidate all SSTables into one, removing deletion information and deleted data. These compactions help manage memory, ensure data integrity, and reclaim resources.

## Refinements

### Locality groups

Bigtable allows clients to group column families into locality groups, improving read efficiency by separating data and enabling selective access. Locality groups support tuning parameters like in-memory storage for faster access to frequently used data, such as the location column family in the METADATA table.

### Compression

Clients can choose whether or not to compress SSTables in a locality group, with options for compression formats. A two-pass compression scheme is commonly used, achieving significant space reduction. For example, experiments with Web page contents achieved a 10-to-1 reduction in space. Storing multiple versions of the same value in Bigtable further improves compression ratios.

### Caching for read performance

To improve read performance, tablet servers use two levels of caching. The Scan Cache is a higher-level cache that caches the key-value pairs returned by the SSTable interface to the tablet server code. The Block Cache is a lower-level cache that caches SSTables blocks that wereb read from GFS.

### Bloom Filters

Clients in Bigtable can improve read performance by using Bloom filters, which are created for SSTables in a locality group. Bloom filters help determine if an SSTable might contain data for a specific row/column pair, reducing the need for disk accesses and improving efficiency. They also eliminate the need to access disk for non-existent rows or columns.

### Commit-log implementation

In Bigtable, mutations are appended to a single commit log per tablet server to avoid managing multiple log files. During recovery, the log entries are sorted by keys to efficiently reapply mutations. The log writing process is parallelized and protected against performance issues.

### Speeding up tablet recovery

When a tablet is moved between tablet servers in Bigtable, the source server performs compactions to reduce recovery time. After completing these compactions, the tablet can be loaded onto the new server without log entry recovery.

### Exploiting immutability

The immutability of SSTables in Bigtable simplifies various aspects of the system. It eliminates the need for synchronization when reading from SSTables, allowing efficient concurrency control. The memtable, the only mutable data structure, is designed to minimize contention during reads and writes. Instead of permanently removing deleted data, obsolete SSTables are garbage collected. SSTables' immutability also enables fast tablet splitting by allowing child tablets to share the SSTables of the parent tablet.

## Performance Evaluation

A Bigtable cluster was tested with varying numbers of tablet servers (N). The tests measured performance and scalability using benchmarks for sequential and random reads/writes, scans, and in-memory reads. The cluster consisted of machines with sufficient resources and high-speed network connections. The results showed the number of operations per second per tablet server and the overall aggregate performance.

### Single tablet-server performance

In Bigtable, when using a single tablet server, random reads are slower than other operations due to network transfers. Random and sequential writes perform similarly and are faster than random reads. Sequential reads are faster than random reads due to block caching. Scans are the fastest because they return multiple values in a single request, reducing RPC overhead.

### Scaling

Increasing the number of tablet servers in Bigtable improves aggregate throughput significantly, but not linearly. There is a drop in per-server throughput when going from 1 to 50 servers due to load imbalances and contention with other processes. The random read benchmark shows the weakest scaling, as network transfers saturate shared links, reducing per-server throughput as the number of machines increases.

# Real Applications

## Google Analytics

Google Analytics is a web service that helps webmasters analyze website traffic. It uses a JavaScript program to collect data about user visits and stores it in two tables: the raw click table (compresses to 14% of its size) and the summary table (compresses to 29% of its size). The summary table is generated from the raw click table through MapReduce jobs.

## Google Earth

Google's satellite imagery services use a preprocessing pipeline with a 70 terabyte table for storing cleaned and consolidated imagery. The serving system utilizes a smaller table for indexing data stored in GFS, distributed across multiple servers for efficient query response.

## Personalized Search

Personalized Search is a service by Google that stores user data in Bigtable, allowing personalized search results based on user history. User actions are recorded in separate column families, and user profiles are generated using MapReduce. The data is replicated across clusters for availability, and a quota mechanism limits storage consumption for shared tables.

# Lessons

Lessons learned from designing and supporting Bigtable include the vulnerability of distributed systems to various failures, the importance of delaying new feature implementation until their use is clear, the value of proper system-level monitoring, and the benefits of simple designs. Various problems were addressed by changing protocols and removing assumptions. Monitoring helped detect and fix issues like lock contention and slow writes. Code and design clarity aided in maintenance and debugging. A simpler tablet-server membership protocol was adopted after encountering complexity and dependency issues with the initial protocol.

# Related Work

Boxwood project provides infrastructure for building higher-level services, while Bigtable focuses on direct client data storage. Distributed hash tables address different concerns than Bigtable. Bigtable's data storage model supports semi-structured data efficiently. Bigtable has similarities to parallel databases like Oracle's Real Application Cluster and IBM's DB2 Parallel Edition. Bigtable's locality groups offer compression and read performance benefits. Memtables and SSTables in Bigtable resemble the Log-Structured Merge Tree approach. Bigtable and C-Store share characteristics but differ in API and application focus. Bigtable's load balancer handles load balancing with simpler considerations than shared-nothing databases.