> *Bases de Datos 2, Resumen 2 y 3: Bigtable: A Distributed Storage System for Structured Data*
> *Miguel Ku Liang - 2019061913*

Introduction

Bigtable is designed to reliably scale to petabytes of data and thousands of machines, achieving wide applicability, scalability, high performance and high availability. Bigtable resembles a database. It does not support a full relational data model, but provides clients with a simple data model that supports dynamic control over data layout and format and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names. It treats data as uninterpreted strings.

Data Model

A Bigtable is a sparse, distributed, persistent multidimensional sorted map. It is indexed by a row key, column key and a timestamp. Each value is an uninterpreted array of bytes. A **Webtable** is a Bigtable-like system to keep large collections of web pages and related information that could be used in many projects.

Rows

The row keys are arbitrary strings. Every read or write of data under a single row key is atomic. Bigtable maintains data in lexicographic order by row key. The row range is dynamically partitioned. Each row range is called a **tablet**. It is the unit of distribution and load balancing, so reads of short row range are efficient and only needs to communicate with a small amount of machines.

Column Families

Column keys are grouped in sets called **column families**. They form the basic unit of access control. All data stored in a column family is usually of the same type. A column family must be created before storing data. A column key is named in the form of ***family:qualifier***. Family names must be printable and qualifiers may be arbitrary strings. Access control and disk and memory accounting are performed at the column-family level.

Timestamps

Each cell in a Bigtable can contain multiple versions of the same data indexed by timestamps. They are 64-bit integers. They can be assigned by Bigtable, representing real time in microseconds or explicitly assigned by client applications. Applications that need to avoid collisions must generate unique timestamps themselves. Different versions of a cell are sorted in decreasing timestamp order so that the most recent versions can be read first. To manage versioned data, it has two per-column-family settings that tell Bigtable to collect cell versions automatically.

API

The Bigtable API provides functions for creating and deleting tables and column families. It provides functions for changing cluster, table, and column family

metadata. Client applications can write or delete values in Bigtable, look up values from individual rows or iterate over a subset of the data in a table.

Bigtable supports other features that allow the user to manipulate data in more complex ways. It supports single-row transactions that are used to perform atomic read-modify-write sequenceson data stored under a single row key. Bigtable does not support general transactions across row keys, but it provides an interfacce for batching writes across row keys at the clients. It allows cells to be used as integer counters. 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.

Building Blocks

Bigtable uses the distributed **Google File System** to store log and data files. A Bigtable cluster 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. It depends on a cluster management system for scheduling jobs, managing resources on shared machines, dealing with machine failures and monitoring machine status.

The Google **SSTable** file format is used to store Bigtable data. It provides a persistent, ordered immutable map from keys to values, where both are arbitrary byte strings. Operations are provided to look up the value associated with a specified key. Each SSTable contains a sequence of blocks. A **block index** is used to locate blocks. A lookup can be performed with a single disk seek. It finds the corresponding block by performing a binary search in the in-memory index and reads the block from disk. The SSTable can be mapped into memory, allowing it to perform lookups and scans without touching the disk.

Bigtable relies on a highly-available and persistent distributed lock service called **Chubby**. It consists of five active replicas, one elected to be the master. Chubby uses the **Paxos algorithm** to keep its replicas consistent if they fail. Bigtable uses Chubby to ensure that there is at most one active master everytime, to store the bootstrap location of Bigtable data, to discover tablet servers and finalize tablet server deaaths, to store Bigtable schema information and to store access control lists. If Chubby is unavailable for an extended period of time, Bigtable becomes unavailable.

Implementation

The Bigtable implementation has three components: a library that is linked into every client, one master server and many tablet servers. Tablet servers can be dynamically added from a cluster to accomodate changes in workloads. The master is responsible for assigning tablets to tablet servers, detecting the addition and expiration of tablet servers, balancing tablet-server load, garbage collection of files in GFS and handles schema changes. Each tablet server manages a set of tablets. They handle read and write requests to the tablets that it has loaded and splits tablets that have grown too large. Clients communicate directly with tablet servers for reads and writes so the master is lightly loaded in practice.

A Bigtable cluster stores a number of tables. Each table consists of a set of tablets and each tablet contains all data associated with a row range. As a table grows, it is split into multiple tablets, each 100-200 MB in size by default.

Tablet Location

It uses a three-level hierarchy to store tablet location information. The first level is a file stored in Chubby that contains the location of the **root tablet**. It contains the location of all tablets in a special **Metadata table**. Each metadata table contains the location of a set of **user tablets**.

Tablet Assignment

Each tablet is assigned to one tablet server at a time. The master keeps track of the set of live tablet servers and the current assignment of tablets to tablet servers. When a tablet is unassigned and a tablet server with sufficient room for the tablet is available, the master assigns the tablet by sending a tablet load request to the tablet server.

Tablet Serving

The persistent state of a tablet is stored in GFS. Updates are committed to a commit log that stores redo records. The recently committed ones are stored in memmory in a sorted buffer called **memtable**. The older updates are stored in a sequence of SSTable. To recover a tablet, a tablet server reads its metadata from the metadata table. When a write operation arrives at a tablet server, the server checks that it is well-formed and that the sender is authorized to perform the mutation. When a read operation arrives at a tablet server, it is checked for well-formedness and proper authorization.

Compactions

As write operations execute, the size of the memtable increases. When it reaches a threshold, the memtable is frozen, a new memtable is created and the frozen memtable is converted to an SSTable and written to GFS. This **minor compaction** shrinks the memory usage of the tablet server and reduces the amount of data that has to be read from the commit log during recovery if this server dies. If this continues unchecked, read operations might need to merge updates from an arbitrary number of SSTables. Instead, we execute a **merging compaction** in the background. It reads the contents of a few SSTables and the memtable and writes out a new SSTable. A merging compaction that rewrites all SSTables into one SSTable is called a **major compaction**.

Refinements

Locality groups

Clients can group multiple column families together into a **locality group**. A separate SSTAble is generated for each locality group in each tablet. Segregating column families that are not accessed together into separate locality groups enables more efficient reads.

Compression

Clients can control whether or not the SSTables for a locality group are compressed, and if they are compressed, which compression format is used. The user-specified compression format is applied to each SSTable block. Many clients use a two-pass custom compression scheme. The first pass uses **Bentley and McIlroy's scheme** which

compresses long common strings across a large window. The second pass uses a fast compression algorithm that looks for repetitions in a small 16 KB window of the data.

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 were read from GFS. The **Scan Cache** is most useful for applicactions that tend to read the same data repreatedly. The **Block Cache** is useful for applications that tend to read data that is close to the data they recently read.

Bloom filters

A **Bloom filter** allows us to ask whether an SSTable might contain any data for a specified row or column pair. For some applications, a small amount of tablet server memory used for storing Bloom filters reduces the number if disk seeks for read operations.

Commit-log implementation

Depending on the file system implementation on each GFS server, the amount of written files in GFS could cause a large number of disk seeks to write to the different log files. To fix this, we append mutations to a single commit log per tablet server, co-mingling mutations for different tablets in the same log file.

Speeding up tablet recovery

If the master moves a tablet from one tablet server to another, the source tablet server does a minor compaction on that tabler. This reduces recovery time by reducing the amount of uncompacted state in the tablet server's commit log. The tablet server stops serving the tablet. Before this, the tablet server does another minor compaction to eliminate any uncompacted state in the tablet server's log. Then, the tablet can be loaded on another tablet server without requiring any recovery of log entries.

Exploiting immutability

Various parts of the Bigtable system have been simplified because all of the SSTables that we generate are immutable. Since SSTables are immutable, the problem of permanently removing deleted data is transformed to garbage collecting obsolete SSTables. It also enables us to split tablets quickly.

Performance Evaluation

Single tablet-server performance

Random reads are slower than all other operations by an order of magnitude or more. Random reads from memory are much faster.

Random and sequential writes perform better than random reads since each tablet server appends all incoming writes to a single commit log and uses group commit to stream these writes efficiently to GFS. There is no significant difference between the performance of random writes and sequential writes.

Sequential reads perform better than random reads. Scans are even faster since the tablet server can return a large number of values in response to a single client RPC.

Scaling

Aggregate throughput increases dramatically as we increase the number of tablet servers in the system from 1 to 500.

Real Applications

Google Analytics

Google Analytics is a service that helps webmasters analyze traffic patterns at their web sites. The **raw click table** maintains a row for each end-user session. The row name is a tuple containing the website's name and the time the session was created. The **sumary table** contains various predefined summaries for each website. It is generated from the raw click table by periodically scheduled MapReduce jobs. Each job extracts recent session data from the raw click table.

Google Earth

Provides users with access to high-resolution satellite imagery of the world's surface. This system uses one table to preprocess data and a different set of tables for serving client data.

Personalized Search

Personalized Search is an opt-in service that records user queries and clicks across a variety of Google properties. It stores each user's data in Bigtable. Each user has a unique user id and is assigned a row named by that user id. All user actions are stored in a table.

Lessons

- Large distributed systems are vulnerable to many types of failures.
- Delay adding new features until it is clear how the new features will be used.
- Proper system-level monitoring
- Value of simple designs

Related Work

The **Boxwood project** has components that overlap with Chubby, GFS and Bigtable. Its goal is to provide infrastructure for building higher-level services such as databases while Bigtable's goal is to support client applications that wish to store data. Several database vendors have developed parallel databases that can store large volumes of data like **Oracle's Real Application Cluster database**. It uses shared disks to store data and a distributed lock manager. IBM's **DB2 Parallel Edition** is based on a shared-nothing architecture similar to Bigtable.

Conclusions

The control over Bigtable's implementation and other Google infrastructure that Bigtable depends, means that it can remove bottlenecks and inefficiencies as they arise.