> Bases de Datos 2, Resumen 8
> 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. It does not support a full relational data model, but provides clients with a simple data model that supports dynamic control over data layout.

**Data Model**

A Bigtable is a sparse, distributed, persistent multidimensional sorted map. It is indexed by a row key, column key and a timestamp.

- 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.
- Column Families: column keys are grouped in sets called column families. All data stored in a column family is usually of the same type. A column key is named in the form of family:qualifier. 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 can be assigned by Bigtable, representing real time in microseconds or explicitly assigned by client applications.Different versions of a cell are sorted in decreasing timestamp order so that the most recent versions can be read first.

**API**

The Bigtable API provides functions for creating and deleting tables and column families. 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 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.

Bigtable relies on a highly-available and persistent distributed lock service called Chubby. 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.

**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. A Bigtable cluster stores a number of tables.

- Tablet Location: it uses a three-level hierarchy to store tablet location information.
- 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.
- Tablet Serving: the persistent state of a tablet is stored in GFS. To recover a tablet, a tablet server reads its metadata from the metadata table.
- Compactions: minor compaction process

**Refinements**

- Locality groups: clients can group multiple column families together into a locality group. 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.
- 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.
- Bloom filters: a Bloom filter allows us to ask whether an SSTable might contain any data for a specified row or column pair.
- 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.
- Exploiting immutability: various parts of the Bigtable system have been simplified because all of the SSTables that we generate are immutable.

**Performance Evaluation**

- Single tablet-server performance: random reads are slower than all other operations by an order of magnitude or more. 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. 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.