Peer review - Bigtable: A Distributed Storage System for Structured Data

María Fernanda Mora-Alba - Master in Computer Science, ITAM

I. Summary

Big Table is a scalable, highly available and high-performing distributed storage system for managing structured data. It is used by many Google products (+50).

II. Problem definition

Google developed *Big Table* (*BT*) for its particular needs. The technological environment was the following:

- The Boxwood project provided infrastructure for higher-level services (file systems/datasets), while *BT* wanted to directly support client apps aiming to store data.
- CAN, Chord, etc., offered distributed storage over WANs but addressed concerns not relevant for *BT* (variable bandwidth and configuration, untrusted participants, etc).
- Key-value pair model of B-trees/hash tables is limiting. *BT* wanted a richer but efficient and tunable structure that supported sparsed semi-structured data.
- Oracle's and IBM's databases stored lots of data but with a relational model; *BT's* model is NoSQL.
- C-store is only read-optimized while *BT* is aimed to be also write-optimized. C-store suffers from load and memory balancing problems, *BT* does not.

III. Contribution

Sparse, distributed, persistent multidimensional sorted map.

A. Data Model, API and Building Blocks

Sorted, multidimensional and sparse map. *BT's* data type is similar to a dictionary in Python. As so, it is indexed by an associated *key*, *column key* (and a *time-stamp* discussed later).

The *row keys* are *sorted* arbitrary small strings (for example URLs). Reads or writes under a key are atomic. Consecutive rows are organized in *tablets*, the distribution and load balancing unit. This allows efficient transactions for low range rows and encourages users to take advantage of the locality their sorted data (storing pages with same domain near each other).

The *column keys* are grouped into a small number of *column families*, the unit of access control, disk and memory accounting and created once before storing data. The data in the *column family* usually is of the same type. By contrast, new and many *columns* can be created anytime allowing *BT* to be sparse. The additional level makes the map *multidimensional*.

Timestamps allow data versioning, can be assigned by *BT* or by the clients' apps. To reduce the overhead a two per-column-family settings tells *BT* to garbage-collect versions automatically, keeping only the most recent versions.

The **API** allows creation, deletion and modification of tables and column families. Also the usual db-operations: write, look for or delete values, as well as iterate over column families and limit the rows, columns and timestamps. They can also make complex queries such as atomic read-modify-write sequences of a *single* row key and execution of scripts. *BT* can be integrated to MapReduce to run large-parallel computations.

Distributed and persistent. BT is built over distributed *GFS* to store log and data files. SSTable is *BT's* storage data structure and provides a persistent, *sorted* and immutable map from keys to values. *Chubby* provides a distributed lock service, it stores the root table, schema info and access control list and also synchronizes and detects tablet servers.

B. Implementation and refinements

The implementation has 3 components: *client library* (linked with the user's code), *master server* (coordinates activity) and many *tablet servers* (added or removed dynamically). The *master* assigns tablets to *tablet servers* and balances tablet server load. It also collects garbage of files in GFS and creates table and column family. Each *tablet server* manages a set of 10-1K tablets, handles read/write requests to the tablets and splits large tablets. *Clients* communicate directly with tablet servers for reads/writes to reduce overhead with master. Some refinements to improve *BT*: locality groups, compression, caching, bloom filters and commit-log implementation.

C. Performance evaluation and real applications

Experiments showed that random reads are the slowest because they transfer a SSTable block from GFS; they also have the poorest scaling. Writes and scans are fast (commit log and many values returned). Scaling is not linear but not bad though. *BT* is used in Google's Analytics, Earth, Finance, each with different data size, throughput and latency requirements.

IV. Critic

BT doesn't support byzantine failures, the most general and difficult. [2] proposes protection for Cassandra against them.

BT's reproducibility is questionable: it makes intense use of other tailored-made Google systems.

Each machine in the experiments had enough memory to run the workload: how common is this?

Compactions are expensive: no discussion on the overhead.

References

[1] Chang, Fay, et al. "Bigtable: A distributed storage system for structured data." ACM Transactions on Computer Systems (TOCS) 26.2 (2008): 4.

[2] Friedman, Roy, and Roni Licher. "Hardening Cassandra Against Byzantine Failures." arXiv preprint arXiv:1610.02885 (2016).