Bigtable : A Distributed Storage System For Structured Data

F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber

Gartheeban Ganeshapillai, MIT (6.897 Spring 2011)

Google handles tremendous amount of data, and provides diverse set of services. Their data-storage requirements are diverse and unique; so is Bigtable : the home-brew solution for the distributed storage of structured data.

Bigtable was built with the goals of providing highly and widely available, scalable, high performance data storage solution to support broad set of Google services. Currently, more than 60 Google applications, including Google Earth, Google Analytics, and Web indexing, use Bigtable. Bigtable, published in ACM-TCM 2008, provides a detailed description of the system.

In Bigtable, a table is a sparse, distributed, persistent multidimensional-sorted map. It is indexed in three dimensions: (row, column, timestamp) → string. Rows are keyed by arbitrary strings that are ordered lexicographically. For instance, in Webtable, the reverse URL keys the rows. Consecutive rows are grouped into tablets, which form the unit of distribution and load balancing. Hence, accessing a short row range, such as scanning is more efficient compared to random reads. Column keys are grouped into column families that form the unit of access control. Column family must be created explicitly before any data can be stored, and they change rarely during the operation. However, any column key within the family can be added later. A column key is named the following syntax: *family:qualifier*. Cells in a table can contain multiple versions of the same data, and the versions are indexed by the timestamp. To manage the versioned data, the older versions are automatically garbage-collected.

Bigtable is built on several pieces of Google infrastructure. It relies on Google Cluster Management for scheduling jobs, managing resources with shared machines, monitoring machine status, and dealing with machine failures. Bigtable uses GFS to store log and data files. It also depends on highly available persistent distributed lock service called *Chubby*. The Chubby service provides a namespace that consists of directories and small files. The file can be used as a lock, because the read and writes are atomic. The Chubby client provides consistent caching of files by maintaining a session with the Chubby service.

The Bigtable implementation comprises of a client library, one master server and many tablet servers. The master server assigns tablets to tablet servers, balances the load, garbage collects the files in GFS, and handles the schema changes. Each tablet server manages about 10 to 1000 tablets, and handles the read and write requests. It also splits the tablets that have grown too large. Chubby is used to make sure that there is at most one active master at any time; to store the root-tablets location to bootstrap the Bigtable data; to discover tablet servers; to store Bigtable schemas.

The implementation uses a three level hierarchy to store the tablet location information. A file in Chubby contains the root tablets location. The root tablet contains the locations of all of the tablets of a special Metadata table. The Metadata table contains the user tablets location: (table_id, end_row_id) → (ip, port). To locate a tablet, the client library traverses through the hierarchy. In addition to caching the locations it finds, it also prefetches the tablet locations when it reads the Metadata table. If the cache is empty, it takes three network round-trips to obtain the location of the tablet; whereas, if the clients cache is empty, it takes six round-trips for the same.

The master keeps track of the set of tablet servers using Chubby and assigns each tablet to at most one tablet server. When a tablet server starts, it creates and acquires an exclusive lock on a file in Chubby, and serves as long as the lock exists. If it loses the lock, it stops serving, and attempts to reacquire the lock as long as the file exists. If the master deletes the file, the tablet server terminates. The master kills itself if its Chubby session expires.

The Google *SSTable* immutable-file format is used internally to store Bigtable data files in GFS. An SSTable provides a persistent ordered immutable map from keys to values. Each SSTable contains a sequence of blocks, and a block index. The block index is used to locate the blocks. A lookup is performed by finding the appropriate block through binary search in the in-memory index, and read the appropriate block from disk; the value is returned for the lookup key. The tablets are loaded by reading the SSTables from GFS and applying the logs after the redo point that is stored in the Metadata of the tablet. The reads are served by merging the SSTable reads with Memtable, an in-memory sorted buffer. Bigtable commits are stored in a log, and entered to Memtable. When the Memtable grows large, it is frozen and converted an SSTable and written to GFS; this is called minor compaction. In contrast, a merging compaction bounds a number of such files periodically in the background. A major compaction rewrites all such SSTables into exactly one SSTable. The major compaction also removes all the deleted data, where as minor compaction keeps them.

The authors present the results of the micro benchmark they carried out. They scaled the number of machines from 1 to 500, and tested for the number of 1000-byte values read/written per second for the following operations: random read, random reads from memory, random writes, sequential reads, sequential writes, and scans. On a single tablet server, each random read required the transfer of a 64 KB SSTable block over the network from GFS out of which only 1000-byte value was used. This resulted in 75 MB/s of data reads from GFS and saturated the tablet server CPU and the network. Further, this also resulted in the worst performance for random reads for scaling. For example, when the system was scaled up by 500 times, random reads achieved only 100 fold increase compared to 300 fold increase achieved by random reads from memory.

Finally, authors also discuss the experience they gained and interesting lessons learned. They have experienced many unexpected failures in implementing the Bigtable. Hence, authors warn against assuming only the standard failures. Further, they suggest delaying the implementation of the new features until it is clear how the new features are used. They cite the use of single-row transactions by most of the applications to support this. They also emphasize the importance of proper system level monitoring and simple designs.