Bigtable: A Distributed Storage System for Structured Data

Data Model

A Bigtable is a sparse, distributed, persistent multidimensional sorted map. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. We settled on this data model after examining a variety of potential uses of a Bigtable-like system.

Column Families

Column keys are grouped into sets called column families, which form the basic unit of access control. All data stored in a column family is usually of the same type . Column family names must be printable, but qualifiers may be arbitrary strings. An example column family for the Webtable is language, which stores the language in which a web page was written. We use only one column key in the language family, and it stores each web page’s language ID.

Timestamps

Bigtable timestamps are 64-bit integers. Different versions of a cell are stored in decreasing timestamp order, so that the most recent versions can be read first.

To make the management of versioned data less onerous, we support two per-column-family settings that tell

Bigtable to garbage-collect cell versions automatically. The client can specify either that only the last n versions of a cell be kept, or that only new-enough versions be kept .

Building Blocks

System to store log and data files. A Bigtable cluster typically 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 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 internally to store Bigtable data.

An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Internally, each SSTable contains a sequence of blocks . Optionally, an SSTable can be completely mapped into memory, which allows us to perform lookups and scans without touching disk. Bigtable relies on a highly-available and persistent distributed lock service called Chubby .

A Chubby service consists of five active replicas, one of which is elected to be the master and actively serve requests. Chubby provides a namespace that consists of directories and small files. If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable. We recently measured this effect in 14 Bigtable clusters spanning 11 Chubby instances.

Tablet Serving

The persistent state of a tablet is stored in GFS, as illustrated in Figure 5. Updates are committed to a commit log that stores redo records. To recover a tablet, a tablet server tablet log. This metadata contains the list of SSTables that comprise a tablet and a set of a redo points, which are pointers into any commit logs that may contain data for the tablet. The server reads the indices of the SSTables into memory and reconstructs the memtable by applying all of the updates that have committed since the redo points. 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 similarly checked for well-formedness and proper authorization.

Compactions

- SSTables. A merging compaction reads the contents of a few SSTables and the memtable, and writes out a new SSTable. The input SSTables and memtable can be discarded as soon as the compaction has finished. A merging compaction that rewrites all SSTables into exactly one SSTable is called a major compaction.
- SSTables produced by non-major compactions can contain special deletion entries that suppress deleted data in older SSTables that are still live. A major compaction, on the other hand, produces an SSTable that contains no deletion information or deleted data. Bigtable cycles through all of its tablets and regularly applies major compactions to them. Bigtable to reclaim resources used by deleted data, and also allow it to ensure that deleted data disappears from the system in a timely fashion, which is important for services that store sensitive data.

Locality groups

- A separate SSTable is generated for each locality group in each tablet. Segregating column families that are not typically accessed together into separate locality groups enables more efficient reads. In addition, some useful tuning parameters can be specified on a per-locality group basis. For example, a locality group can be declared to be in-memory.
- SSTables for in-memory locality groups are loaded lazily into the memory of the tablet server. Once loaded, column families that belong to such locality groups can be read without accessing the disk.

Compression

Although we lose some space by compressing each block separately, we benefit in that small portions of an SSTable can be read without decompressing the entire file. Many clients use a two-pass custom compression scheme. The second pass uses a fast compression algorithm that looks for repetitions in a small 16 KB window of the data. Even though we emphasized speed instead of space reduction when choosing our compression algorithms, this two-pass compression scheme does surprisingly well.

Bloom filters

A read operation has to read from all SSTables that make up the state of a tablet. A Bloom filter allows us to ask whether an SSTable might contain any data for a specified row/column pair. For certain applications, a small amount of tablet server memory used for storing Bloom filters drastically reduces the number of disk seeks required for read operations.

Commit-log implementation

If we kept the commit log for each tablet in a separate log file, a very large number of files would be written concurrently in GFS. Depending on the underlying file system implementation on each GFS server, these writes could cause a large number of disk seeks to write to the different physical log files. In addition, having separate log files per tablet also reduces the effectiveness of the group commit optimization, since groups would tend to be smaller. To fix these issues, we append mutations to a single commit log per tablet server, co-mingling mutations for different tablets in the same physical log file .

Using one log provides significant performance benefits during normal operation, but it complicates recovery. To recover the state for a tablet, the new tablet server needs to reapply the mutations for that tablet from the commit log written by the original tablet server. One approach would be for each new tablet server to read this full commit log file and apply just the entries needed for the tablets it needs to recover. However, under such a scheme, if 100 machines were each assigned a single tablet from a failed tablet server, then the log file would be read 100 times .

Each machine had two dual-core Opteron 2 GHz chips, enough physical memory to hold the working set of all running processes, and a single gigabit Ethernet link. Every machine ran a GFS server. Some of the machines also ran either a tablet server, or a client process, or processes from other jobs that were using the pool at the same time as these experiments. R is the distinct number of Bigtable row keys involved in the test.

Single tablet-server performance

Random reads are slower than all other operations by an order of magnitude or more. Each random read involves the transfer of a 64 KB SSTable block over the network from GFS to a tablet server, out of which only a single 1000-byte value is used.

Scaling

A drop is caused by imbalance in load in multiple server configurations, often due to other processes contending for CPU and network. This transfer saturates various shared 1 Gigabit links in our network and as a result, the per-server throughput drops significantly as we increase the number of machines.

8 Real Applications

As of August 2006, there are 388 non-test Bigtable clusters running in various Google machine clusters, with a combined total of about 24,500 tablet servers. Table 1 shows a rough distribution of tablet servers per cluster. One group of 14 busy clusters with 8069 total tablet servers saw an aggregate volume of more than 1.2 million requests per second, with incoming RPC traffic of about 741 MB/s and outgoing RPC traffic of about 16 GB/s.

- Compression ratio is not given for tables that have compression disabled. The table contains a column family to keep track of the sources of data for each segment. The overall system processes over 1 MB/sec of data per tablet server during some of these MapReduce jobs. The serving system uses one table to index data stored in GFS.
- Personalized Search is an opt-in service that records user queries and clicks across a variety of Google properties such as web search, images, and news.

9 Lessons

In the process of designing, implementing, maintaining, and supporting Bigtable, we gained useful experience and learned several interesting lessons. One lesson we learned is that large distributed systems are vulnerable to many types of failures, not just the standard network partitions and fail-stop failures assumed in many distributed protocols. As we have gained more experience with these problems, we have addressed them by changing various protocols. Now that we have many real applications running on Bigtable, we have been able to examine their actual needs, and have discovered that most applications require only single-row transactions.