Bigtable: A Distributed Storage System for Structured Data

Data Model

A Bigtable is a sparse, distributed, persistent multi- dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

Rows

The row keys in a table are arbitrary strings (currently up to 64KB in size, although 10-100 bytes is a typical size for most of our users).

Bigtable maintains data in lexicographic order by row key. The row range for a table is dynamically partitioned. Each row range is called a tablet, which is the unit of distribution and load balancing.

Column Families

Column keys are grouped into sets called column fami- lies, which form the basic unit of access control. All data stored in a column family is usually of the same type (we compress data in the same column family together).

Timestamps

Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by timestamp. Bigtable timestamps are 64-bit integers. They can be as- signed by Bigtable, in which case they represent “real time” in microseconds, or be explicitly assigned by client applications.

API

The Bigtable API provides functions for creating and deleting tables and column families. It also provides functions for changing cluster, table, and column family metadata, such as access control rights.

Building Blocks

Bigtable is built on several other pieces of Google in- frastructure. Bigtable uses the distributed Google File System (GFS) 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.

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.

Implementation

The Bigtable implementation has three major compo- nents: a library that is linked into every client, one mas- ter server, and many tablet servers. Tablet servers can be dynamically added (or removed) from a cluster to acco- modate changes in workloads.

Tablet Location

The first level is a file stored in Chubby that contains the location of the root tablet. The root tablet contains the location of all tablets in a special METADATA table. Each METADATA tablet contains the location of a set of user tablets. The root tablet is just the first tablet in the METADATA table, but is treated specially—it is never split—to ensure that the tablet location hierarchy has no more than three levels.

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, including which tablets are unassigned. 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. Of these updates, the re- cently committed ones are stored in memory in a sorted buffer called a memtable; the older updates are stored in a sequence of SSTables. To recover a tablet, a tablet server reads its metadata from the METADATA table. This meta- data 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.

Compactions

As write operations execute, the size of the memtable in- creases. When the memtable size 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 process has two goals: it shrinks the memory usage of the tablet server, and it reduces the amount of data that has to be read from the commit log during recovery if this server dies. Incom- ing read and write operations can continue while com- pactions occur.

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 typically accessed together into sep- arate locality groups enables more efficient reads.

Compression

Clients can control whether or not the SSTables for a locality group are compressed, and if so, which com- pression format is used. The user-specified compres- sion format is applied to each SSTable block (whose size is controllable via a locality group specific tuning pa- rameter).

Caching for read performance

To improve read performance, tablet servers use two lev- els 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.

Bloom filters

A read operation has to read from all SSTables that make up the state of a tablet. If these SSTables are not in memory, we may end up doing many disk accesses. We reduce the number of accesses by allowing clients to specify that Bloom fil- ters should be created for SSTables in a particu- lar locality group.

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

Speeding up tablet recovery

If the master moves a tablet from one tablet server to another, the source tablet server first does a minor com- paction on that tablet. This compaction reduces recov- ery time by reducing the amount of uncompacted state in the tablet server's commit log.

Exploiting immutability

Besides the SSTable caches, various other parts of the Bigtable system have been simplified by the fact that all
of the SSTables that we generate are immutable.

Performance Evaluation

We set up a Bigtable cluster with N tablet servers to measure the performance and scalability of Bigtable as N is varied. The tablet servers were configured to use 1 GB of memory and to write to a GFS cell consisting of 1786 machines with two 400 GB IDE hard drives each. N client machines generated the Bigtable load used for these tests.

- The sequential read benchmark generated row keys in exactly the same way as the sequential write benchmark, but instead of writing under the row key, it read the string stored under the row key (which was written by an earlier invocation of the sequential write benchmark).
- The scan benchmark is similar to the sequential read benchmark, but uses support provided by the Bigtable API for scanning over all values in a row range.
- The random reads (mem) benchmark is similar to the random read benchmark, but the locality group that con- tains the benchmark data is marked as in-memory, and therefore the reads are satisfied from the tablet server's memory instead of requiring a GFS read.

Scalling

Aggregate throughput increases dramatically, by over a factor of a hundred, as we increase the number of tablet servers in the system from 1 to 500.

Real Applications

- Google Analytics
- Google Earth
- Personalized Search

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 sys- tems are vulnerable to many types of failures, not just the standard network partitions and fail-stop failures assumed in many distributed protocols.

Related Work

The Boxwood project has components that overlap in some ways with Chubby, GFS, and Bigtable, since it provides for distributed agreement, locking, distributed chunk storage, and distributed B-tree storage. In each case where there is overlap, it appears that the Box- wood's component is targeted at a somewhat lower level than the corresponding Google service. The Boxwood project's goal is to provide infrastructure for building higher-level services such as file systems or databases,
while the goal of Bigtable is to directly support client applications that wish to store data