This is a summary of the paper “Bigtable: A Distributed Storage System for Structured Data”. References are shorthanded as (x.y) where x is the page number and y is the paragraph on that page.

Background

Google’s Bigtable is a datastructure similar to, but not to be confused with a relational database (1.3). It is meant to be general enough to handle a wide variety of uses, but the primary drivers are extremely large data sets and high performance for the client (1.2). Currently more than 60 products use Bigtable and several hundred terabytes in practice (although petabyte size is theoretically possible) (1.2).

Data Structure

The basic unit of the table is the untyped string which at the lowest level are stored in sorted string tables or SSTables which are small enough to fit in RAM (3.9, 4.1). The SSTable is an immutable map of type (string key) → string data. Bigtable is exposed to the user as a three-dimensional map of type (string row, string col, int64 time)→string data (1.3, 1.4). As a result of the underlying SSTable structure Bigtable is sorted lexicographically along row, col and time (2.3). The row and col keys can be 64KB in size, although that’s not usually necessary (2.2).

The multi-dimensional map is divided into row ranges (i.e. A-D, E-F,G-P,Q-Z) referred to as a tablet (2.3, 4.8). Each tablet can be between 100MB and 200MB (4.8). As data is added to the tablet range the tablet expands and eventually splits into two distinct tablets (i.e. A-D → A-B and C-D) (4.8).

The multi-dimensional map is divided into column families based on user defined groups (2.4). The column family is created first as a data category (2.4). Each column is added to a column family and cannot exist on its own (2.4). Typically there are relatively few column families, but many more columns (2.4). The column keys are in the form Family_name:Column_name (2.5). Granularity at the column family level is used to control access, disk/memory accounting and garbage handling (2.6, 3.2).

The multi-dimensional map automatically assigns “real” timestamps based on when data was entered (2.7). If a real timestamp is used, the possibility of collision is guaranteed not to occur (3.1). The client application can explicitly assign timestamps based on some internal versioning system (2.7). Version numbers are used in garbage collection either to maintain a maximum number of versions or maintain versions “younger” than an arbitrary age (3.2).

Bigtable API

The API has several means of controlling data access to improve performance and simplify coding. Read-modify-write operations can be performed atomically on a single row regardless of the number columns updated (2.2, 3.6, Figure 2). Reads can be performed in sections based on a row range (3.5). Typically similar or nearly adjacent rows are on the same tablet and therefore can be accessed with one network call (2.3). Records can be fetched by specific rows, columns or versions based on various limiting factors such as matching regular expressions or a range of versions (3.5, Figure 3). New data can be written and old data deleted (3.5). Direct iteration over the fetched data can be used just as on any other collection library (3.5, 4.1). Client applications can also use the scripts written in Sawzall to instruct server-side data processing prior to the network fetch but specifically excluding any write operation (3.6). The API also allows map reduce function to work on Bigtable and produce Bigtable in a closed algebra (3.7).

Client applications can improve performance in various ways beyond those imposed by the API. As noted above they can define the access control rights, disk/memory storage, versioning schemes and garbage handling functions for a column family (2.6, 3.2). The client application can also use their own

"cleverness" in serializing the data and creating the row and column names (1.3). The client application can add/delete tables and column families as necessary (3.4). By analyzing the application domain, these properties may be used in conjunction with the abilities of the API to improve performance beyond that of a generic Bigtable.

Server Address Discovery

The Bigtable servers are discovered through a B-Tree with three levels (4.9). A bootstrap pointer is used to identify the location of the root tablet (4.9). The root tablet is unique in that it is never split (4.9). The root tablet points to a set of tablets which point to all the other tablets in the Bigtable (4.9). The root tablet and the set it points to are known collectively as the metadata table (4.9). The metadata table stores table name and end-row key values along with the address (4.10). Even with a moderate tablet size the resulting address space can be very large (5.1).

The system requires 3 network reads to discover any address (5.2). Although this system works for discovery, it is not necessary to always perform so many as addresses are aggressively cached and kept in larger-than necessary groups (5.2). Occasionally a tablet will have been moved so a search at the next highest level of the B-tree will be necessary (5.2). In the worst case scenario (all tables in the branch have been moved) six network reads are necessary (5.2).

Server Failover and Recovery

The table exist on a variety of servers, necessitating a management scheme encompassing failover and recovery (3.8). Availability statistics show an average of .0047% down time (4.3). A mainstay of the system's reliability is the Chubby file system which is used throughout (4.3). Chubby creates 5 replicas of each file to avoid loss during failover (4.2). One copy is the master and the rest are kept current using the Paxos algorithm (4.2). Chubby's use of files and folders with individual locks that must be maintained within a session (4.2). If a server, including the master, loses connectivity for too long its session will automatically expire and the lock will be lost (4.2, 5.6). If the master server loses its lock it will automatically kill itself.

The underlying Chubby file system is used in many ways to coordinate and increase availability. One server is uses as the master for the system (4.3). The Chubby system stores the bootstrap pointer to the table's B-Tree (4.3). It maintains a tablet directory with individual files for each tablet and particularly the unique master file (4.3, 5.7). This directory is used for tablet discovery, to determine the death of a server and to coordinate tablet splits/mergers (4.3).

The master server contains no tablets and is not used to locating the tablet servers (4.4, 4.7, 5.2). It's sole purpose is to assign tablets to the various tablet servers and balance the load between them (4.5). This also leads to related duties such as adding/removing tablet servers, maintaining a list of unassigned tablets, garbage collection, creation of new tables and merging of tablets (4.5, 5.8). Only one server has a "live" copy of a tablet at a time (5.4). The live tablet status is determined by ownership of a Chubby file lock corresponding to the tablet (5.5).

The master server periodically attempts to acquire the existing locks (5.7). If the master identifies an unused lock the tablet is seen as unassigned and the master will reassign it to another tablet (5.5, 5.6). The original file is destroyed to prevent the former tablet server from reacquiring it (5.6). If the master itself fails a new master is assigned and it will start by checking the locks again (5.7). The polled locks are used to identify live servers, which are then polled to determine the set of live tablets (5.7). If the root tablet is not identified in this step it is immediately listed as unassigned and assigned to another server

(5.8). Finally a search through the metadata tablets will indicate the complimentary set of unassigned tablets (5.7).

There are many tablet servers, each keeping 10 to 1000 tablets (4.4, 4.6). The server handles all read/write requests on its own tablets (4.6, 4.7). When any one tablet gets too large the server will split the tablet (4.6, 5.8). Upon a split, the tablet server creates a new file in the Chubby system and notifies the master (5.8). Even if the message is lost or if the master server fails, the new file itself will prompt the same division (6.1).

Read, Write and the Google File System

Changes to the data are initially committed to the a log on the GFS (6.2). The original data is maintained in SSTables also on the GFS (6.2). The live tablet server will maintain a "memtable" in RAM which contains a record of all the logged commitments (6.2). The union of the memtable and SSTables is the true state of memory (6.2). Write operations result in updates in both the commit log and memtable, including deletions (6.3, 6.7). Read operations read simultaneously and sequentially from memtable and SSTable with an automatic merge as they are both sorted (6.2, 6.4). If the tablet must be recovered the memtable and SSTables are read and merged (6.2). Deletions in any subset will result in suppressing other data from the whole (6.7).

Periodically the whole set of data is collected. This comes in three flavors. Minor compactions merge the memtable and SSTable into a new SSTable (6.5). This is prompted with the memtable is too large and the result is free server RAM in trade for more GFS space (6.5). Eventually this behavior will overwhelm the system (6.6). An unnamed middle compaction merges memtable and more than one SSTable to free up GFS space (6.6). Major compactions merge the whole set into one SSTable (6.7). The resulting SSTable contains no deletions as they have been used to suppress prior data (6.7). Major compactions have additional benefits of permanently removing old data, reducing the total amount of memory and permanently finalizing the data into GFS (6.7).