Paper Trail

Computer Systems, Distributed Algorithms and Databases

BigTable: Google's Distributed Data Store

Although GFS provides Google with reliable, scalable distributed file storage, it does not provide any facility for structuring the data contained in the files beyond a hierarchical directory structure and meaningful file names. It's well known that more expressive solutions are required for large data sets. Google's terabytes upon terabytes of data that they retrieve from web crawlers, amongst many other sources, need organising, so that client applications can quickly perform lookups and updates at a finer granularity than the file level.

So they built BigTable, wrote it up, and published it in OSDI 2006. The paper is here, and my walkthrough follows.

Data Model

The very first thing you need to know about BigTable is that it isn't a relational database. This should come as no surprise: one persistent theme through all of these large scale distributed data store papers is that RDBMSs are hard to do with good performance. There is no hard, fixed schema in a BigTable, no referential integrity between tables (so no foreign keys) and therefore little support for optimised joins.

Instead, a BigTable (note that there is some term overloading here: BigTable is the system, *a BigTable* is a single table) is described as a *sparse, distributed, persistent, multi-dimensional, sorted map*. That's quite a mouthful. To parse that, let's start with the noun *map*. This tells us that a BigTable provides a mapping from some index – perhaps a row number and a column description – to some data, which may be completely unstructured. The other adjectives modify this definition:

- *Sparse* – it's not expected that the map will be defined at very many places compared to the total index space. For example, if the map is from timestamps to a copy of a webpage (like archive.org), there won't be an entry for most timestamps.
- *Distributed* – we know what this means. The entries in the map may exist on many computers. This gives us a shot at reliable fault-tolerance and availability.
- *Persistent* – once we've created the map, it hangs around effectively for ever, and doesn't get either deliberately or inadvertantly deleted by the system.
- *Multi-dimensional* – this – I think – refers to the fact that the index used to lookup data may be comprised of several dimensions. A simple key->value lookup is one-dimensional. (row,column)->value is two-dimensional. BigTables may be multi-dimensional as the index maps from (row, column, time, column qualifier) onto a value.

So there are some new concepts here. In particular the idea that you can index values by time is novel – so you can retrieve the previous versions of a value by timestamp if you like. This implies that no data are ever deleted from a BigTable, and this is indeed the case. Once a value is written, it's written for life, uniquely identified by its timestamp. Well – almost. Applications may specify to BigTable that only the most recent n or only versions that

were written recently should be kept, and BigTable is free to garbage collect those that are not.

I also skimmed over the idea of a *column qualifier*. Columns are grouped into sets called *column families*, and a qualifier is used to distinguish between two members of a column family. A column is identified by a `family:qualifier` pair. So, to take the example from the paper, consider a table that stores the incoming links to a web page. The table has a column family *anchor* that stores the address linked to, and the site from which it comes. Each referring site has its own column qualifier, for example `anchor:bbc.co.uk`, and the contents of that row and column is the text of the anchor on the web page itself, so perhaps "cnn.com" or "this story at CNN" etc. There is therefore only one column family, but a potentially unbounded number of columns. The design of BigTable is such that the number of column families is expected to be kept small, but the number of columns may become very large. Columns with the same family typically contain the same kind of data, and so may be stored together and compressed together efficiently. Columns may be added at run-time, but the intention again is that column families should be fairly fixed.

So there is a schema of sorts in a BigTable, describing the set of column families. However, no row is required to have data in any column family, and the ability to define columns on the fly means that extra structure may be added to the data at run-time if required, as long as it lies within the loose structure mandated by the column families.

Column families also provide a unit suitable for access control as well as storage.

Programming Model

BigTable provides an API to application developers that allows the typical operations you might expect; creation and deletion of tables and column families, writing data and deleting columns from a row. Transactions are supported at the row level, but not across several row keys (just like PNUTS). However, write batching is implemented to improve throughput.

The paper makes a point of mentioning that BigTable is compatible with Sawzall (the Google data processing language) and MapReduce (the parallel computation framework), the latter uses BigTable as an input and output source for MapReduce jobs.

Implementation

BigTable is built on GFS, which it uses as a backing store both log and data files. GFS provides reliable storage for *SSTables*, which are the Google-proprietary file format used to persist table data. SSTables store data as a simple key->value map which can be looked up with a single disk access by searching an index (which is stored at the end of the SSTable) and then doing a read from the indexed location. Alternatively, SSTables can be completely memory-mapped which avoids the disk read.

The other service that BigTable makes heavy use of is Chubby, the highly-available, reliable distributed lock service written by Mike Burrows (legend has it that he wrote within a couple of days of arriving at Google). Chubby allows clients to take a lock, possibly associating it with some metadata, which it can renew by sending keep alive messages back to Chubby. If the client's session should expire, Chubby will revoke all the locks that it has. Chubby locks are stored in a filesystem-like hierarchical naming structure. (I'll be writing up Chubby before long as an example usage of the Paxos protocol).

Tablet Servers

Just like in PNUTS, BigTables are split into lexicographically ordered (by row key) chunks called *tablets*. BigTable tablet servers each maintain responsibility for a number of tablets (the paper says typically between 10 and 1000). There's no need to replicate tablet servers, because every update to a tablet is reflected in GFS (and therefore replicated) immediately either through a normal log file or being written to an SSTable.

Tablet servers keep the most recent updates to a tablet in an in-memory representation called a *memtable*. Memtables are essentially an in-memory commit log for a tablet, sorted in order (I think) by row key. As updates are made to a tablet, the memtable grows until it is too unwieldy for main memory, at which point it is written to an SSTable on GFS. These SSTables act as snapshots for a tablet server, such that if the server needs to recover from failure it can load the most recent SSTable and apply only the entries in the persistent commit log since the SSTable was stored.

Reads from a tablet server need to possibly go via the saved sequence of SSTables to recover the right instance of the row. The paper says that this is done by querying a merged view of the memtable and the most recent SSTables, which 'can be formed efficiently' – quite how this is done is not specified. Perhaps the view is computed for every read that comes in (and therefore only for small portions of the tablet), or maybe each tablet server always maintains a merged view.

One optimisation that the paper does mention is that, since there may be an unbounded number of SSTables, periodically the SSTables for a given tablet are merged into a single SSTable, with the combined set of updates and a new index. This prevents a read operation from having to trawl back through many SSTables, hitting GFS each time.

Another optimisation mentioned later in the paper is using Bloom filters kept in memory to give a probabilistic indication of whether an SSTable contains data for a given row / column. Bloom filters give only false positives, so there's no danger of missing an update. Some reads from SSTables are also cached, again to save if possible going to disk.

SSTables may be compressed. To make this compression efficient, SSTables may be dedicated to *locality groups*, which are groups of column families. Locality groups may also be tuned on a per-group basis – they may be permanently kept in memory, for example – and by grouping commonly accessed columns together can improve read efficiency.

Finding Tablets

Tablets are located via a three-tier hierarchical lookup mechanism. The location of a root tablet, which contains metadata for a BigTable instance, is located in Chubby and can be read from the filesystem there. The root tablet contains the locations of other metadata tablets, which themselves point to the location of data tablets. Together the root and metadata tablets form the METADATA table, which is itself a BigTable table. Each row in the METADATA table maps the pair (table name, last row) to a location for a tablet whose last row is as given. What isn't completely clear is how a client knows what the last row value for a tablet should be, since tablets change size. This information could be stored in Chubby (most likely), or it might be possible to search the keys of the METADATA table rows (less likely).

Since this three-level lookup requires a lot of network round trips, clients cache the location of tablets. If a cached location is no longer valid, the client can go back to the network. To save yet further on round-trips, locations are speculatively pre-fetched by piggy-backing them on real location queries.

The Master Server

Controlling all of this apparatus is a master server, which is responsible for tracking the set of live servers and assigning tablets to them. It does this by leaning heavily on Chubby. Firstly, the master takes a lock out from Chubby that declares it to be the master, so that no other server can come along and start acting as the master at the same time. Tablet servers also create locks in Chubby, in a directory that is monitored by the master to keep track of the live servers. Periodically, the master asks each tablet server about the state of its lock. If the server responds that it has lost its lock, the master tries to take the lock on its behalf. If the master can acquire the lock, Chubby is clearly alive and it is the tablet server that is having difficulties. The master then removes that server from the list of live servers (by permanently deleting its lock file), and reassigns its tablets.

The master maintains a list of unassigned tablets, and assigns them to live tablet servers as it sees fit. Unassigned tablets are found by comparing the list of tablets reported as managed by tablet servers to the list of tablets in METADATA. It is also responsible for making sure that the METADATA tablets are kept on live servers, and to avoid a chicken-and-egg situation (when it can't read the METADATA tablets to find out that they're not managed) assigns the METADATA tablets as a matter of priority if the root tablet is not reported as managed. That is, the root tablet is the only tablet that the master knows should exist and be managed, and if it is not managed by a server the master is able to bootstrap the system by assigning the root tablet, which contains details of the rest of the METADATA tablets. The details of the root tablet, including I presume its location in GFS, are stored in Chubby.

If the master dies, a backup can bootstrap itself quickly once asked. It's not clear who is responsible for detecting the death of the master – someone needs to monitor the master lock in Chubby and become the master when it gets reclaimed due to session timeout.

Conclusions

A number of real applications are mentioned as running on BigTable. Notable names include Google Analytics and Google Earth. Analytics has a 200TB table size, with 80 billion data 'cells' – a cell is an (row, column) entry. Clearly, BigTable scales to large data sets. Performance also seems impressive: with 250 tablet servers the aggregate number of random reads from memory per second was 2 million, but the aggregate number of reads when all hit the disk was only about 100,000. Clearly the effort taken to keep SSTables cached in memory and reduce the number of disk reads is worthwhile. There aren't any obvious latency figures which is a shame, but it seems clear that most of BigTable's clients will be concerned with throughput (e.g. MapReduce jobs).

There's not a lot of difference, necessarily, between PNUTS and BigTable. PNUTS treats the data lookup and replication problems together, by doing replication of tablets itself rather than relying on a custom replicated storage backend. The cleanliness of Google's approach appeals to me. However, there's a possibility that this separation of concerns has lead to some duplicated effort. In particular the slow path involves two hierarchical lookups: one to find the tablet server and one to lookup the GFS node on which the SSTable is stored (and possibly even more if there are several backing SSTables that have not been compacted). The routing costs in PNUTS aren't clear because they are a function of the implementation of the Yahoo Message Broker, but once

the record master has been found, the tablet server can be looked up with only a couple of network round trips, since it stores the tablet on disk itself.

PNUTS offers strong consistency on a per-record level, the same, except in pathological failure conditions, that BigTable offers (the failure would involve a write to the GFS-held commit log not making it to all GFS replicas, one of which is read due to a cached chunk location by a recovering tablet server).

Both systems have the same sort of requirements and take similar approaches to meeting them. If asked to extract a set of design principles for designing large distributed data stores, these are some of the ones I would come up with:

1. Keep a single master that you can bring up quickly for managing metadata, but keep it out of the critical path for the common operations.
2. Avoid going to disk wherever possible, disk latencies are orders of magnitude more costly than network round trips.
3. Replicate for storage.
4. Don't replicate for functionality: keeping server state in sync is more expensive than replacing a failed server. (Chubby is a counter-example: it has to be seriously highly available).
5. Another benefit of not replicating functional components: serialisation is automatic as all accesses go through the same bottleneck.
6. Treat the row as your fundamental data item. Multiple row transactions are too expensive.

Any others you'd care to add?

Published: October 29, 2008
Filed Under: computer science, Distributed systems, Paper Walkthrough
Tags: bigtable : Distributed systems : google : paper review

2 Responses to "BigTable: Google's Distributed Data Store"

1. *links for 2008-10-30 « Bloggitation* says:
   October 30, 2008 at 10:26 pm

   [...] BigTable: Google's Distributed Data Store at Paper Trail (tags: google database cluster) Possibly related posts: (automatically generated)My LuxUbuntu Laptops!Ubuntu – The Human being Linux [...]

2.  *Neil Devadasan* says:
   December 16, 2008 at 7:00 pm

   Excellent re description to the paper.

Leave a Comment

Name: *Required* Email: *Required, not published*

Homepage: