Bigtable and Boxwood

Robert Burgess
March 24, 2009

What is Bigtable?

"...a sparse, distributed, persistent multi-dimensional sorted map."

- A database system...
  - ...but does not implement a full relational model.
  - Instead, more of an indexed find/iterate system.
  - Column-oriented.
- Aimed at "petabytes" of data in a cluster of "thousands" of commodity servers, within Google
  - Throughput-oriented batch-processing jobs
  - Real-time, interactive data serving

How Does Bigtable Fit In?

- GFS stores *unstructured* data.
- Bigtable stores *structured* data.
- MapReduce handles batch processing over both

Outline

- Data model
- Physical model
- Master and server details
- Implementation details
- Evaluation

The Bigtable Data Model

$$(row, column, time) \rightarrow string$$

Each value in a Bigtable is of this form.

Webtable, the example, stores web indexing data, including contents and links, and was one of the use-case motivating examples used by the designers.

The Bigtable Data Model

$$(row, column, time) \rightarrow \underline{string}$$

In Bigtable, values are always uninterpreted arrays of bytes. Applications may store whatever data they like in each column, including text, serialized structured objects, etc. There is no size limit on each value.

The Bigtable Data Model

$$(\underline{row}, column, time) \rightarrow string$$

- Row keys are also uninterpreted arrays of bytes.
- Implementation supports 64 KiB
  - 10–100 B is more typical.
- Data is stored sorted by row key. Thus,
  - Splitting the table into smaller *tablets* for storage groups similar row keys.
  - Iteration proceeds predictably in order.
- Changes to multiple columns with the same row key are performed atomically
- No other transaction support

The Bigtable Data Model

$$(\underline{row}, column, time) \rightarrow string$$

In the Webtable, row keys are the URL indexed by each row, with DNS names reversed to put the higher-level components first, e.g. `com.google.maps/index.html`.

This is an example of Google changing applications to suit infrastructure; the DNS reversal keeps similar pages together and lets whole domains be explored without ranging across too many tablets.

The Bigtable Data Model

$$(row, \underline{column}, time) \rightarrow string$$

- Columns are grouped into *column families*
- Even a lone column is in a trivial family
- Column families are
  - The basic unit of access control.
  - Compressed together.
  - Limited to a few hundred, although the actual number of columns is unbounded.
- Family names are limited to printing characters
- Column names are uninterpreted bytes

The Bigtable Data Model

$$(row, \underline{column}, time) \rightarrow string$$

Examples from the Webtable:

`language` is a family containing only one column that stores the web page's language ID. Thus, some web page might have the column `language:` with the value `en`.

`anchor` is a more complex family. The column names are the targets of links in the document, and the value stored in each column is the text of the associated link(s). Thus, some web page might have the column `anchor:cnnsi.com` with the value `CNN`, and another column `anchor:my.look.ca` with the value `CNN.com`.

The Bigtable Data Model

$$(row, column, \underline{time}) \rightarrow string$$

- Timestamps are 64-bit integers
- By default, the "current" time in milliseconds
- Applications can explicitly specify timestamps
- Timestamps exist to provide version-limited queries:
  - Return only most recent, or latest n versions.
  - Return only versions within a particular range (e.g. the last 10 days).
- Individual columns are time-stamped
  - Not families or rows.

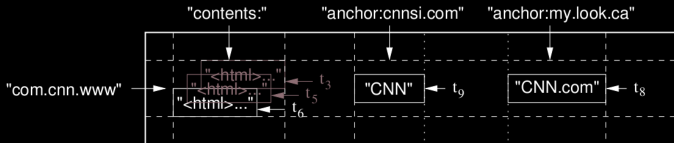
The Bigtable Data Model

$$(row, column, \underline{time}) \rightarrow string$$

The timestamps of data in Webtable are all the time the page was actually crawled. A garbage collection routine periodocally removes all but the most recent three versions of each URL.

The Bigtable Data Model

A logical view of one row of the Webtable:



Storage

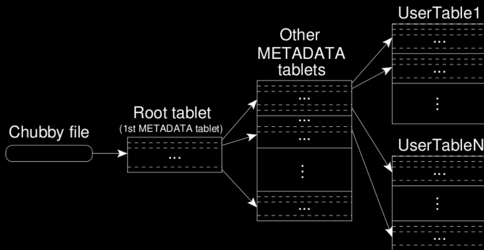
Recall that the logical table is a sorted map of $(row, column, timestamp) \rightarrow string$ values.

- First, group by column family and consider each separately (column-oriented).
- Split into *tablets* of roughly equal size.
- Assign tablets to tablet servers.
    - Dynamically adding and removing tablet servers gives horizontal scalability.
    - Each tablet server stores roughly ten to one thousand tablets.
- As tablets grow and shrink, dynamically split and re-combine them to maintiain size.

Storage

- To locate tablets, Bigtable introduces a special `METADATA` table with one row per tablet.
  - Row keys are an encoding of the tablet's table identifier and its end row.
- The first `METADATA` tablet is special:
  - Called the *root tablet*.
  - Stores the locations of the `METADATA` tablets.
  - Never split.

Storage



Building blocks

Chubby is a highly-available persistent lock service.

- A simple filesystem with directories and small files.
- Any directory or file can be used as a lock.
- Reads and writes to files are always atomic.
- When sessions end, clients lose all locks.

Building blocks

Bigtable uses Chubby to:

- Store the root tablet.
- Store schema (column family) information.
- Store access control lists.
- Synchronize and detect tablet servers (more on this coming up).

Bigtable uses GFS to store all of the actual table data.

Managing Tablets and Tablet Servers

- Tablet servers handle actual tablets
- Master handles `METADATA` and assignment of tablets to servers

When a tablet server starts up, it

- Creates a uniquely-named file in a special *servers directory* in Chubby.
- Attempts to hold this lock permanently.
  - Thus if the lock fails or the file no longer exists, a network partition or other error has broken the Chubby session and the server kills itself.

Master finds running servers by examining the directory.

Managing Tablets and Tablet Servers

If a server reports to the master that it has lost its lock, or the master can not communicate with it after several attempts,

- The master attempts to lock the server's file.
  - If this fails, then Chubby is down, and the master kills itself.
  - Otherwise, the server is just down, and the master
    - Deletes the file, preventing the server from coming back up unexpectedly.
    - Re-assigns the tablets for that server (recall the tablet data itself is actually stored in GFS).

Master failure

Masters in Bigtable are much easier to re-start on failure than in, say, MapReduce. At startup, the master

1. acquires a unique *master* lock to ensure only one master runs at a time,
2. scans the servers directory to find live tablet servers,
3. communicates with each to learn the current assignment of tablets, and
4. scans the `METADATA` table to discover un-assigned tablets.

Thus, the only loss when a master goes down is that during the period of having no master, no new tablets can be assigned or combined.

Serving tablets

A tablet's persistent data is stored in a sequence of *SSTables*. SSTables are

- Ordered
- Immutable
- Mappings from keys to values, both arbitrary byte arrays

SSTables are optimized for storage in GFS and can optionally be mapped into memory.

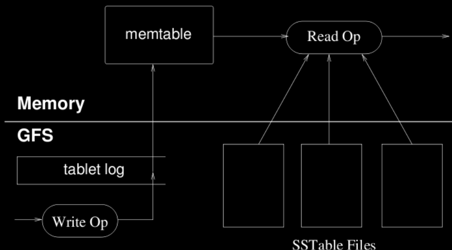
Serving tablets

To serve a tablet assigned to it, a tablet server

- Finds relevant SSTables using the `METADATA` table.
- The SSTables are delta logs with a sequence of checkpoints.
- Recent updates are held in a sorted *memtable* buffer and committed to a tablet log in GFS.

Serving tablets

Thus each tablet server maintains an architecture like this for each tablet assigned to it:



When a server dies, only the memtable is lost, and can be recovered from the SSTables and the tablet log.

Serving tablets

minor compaction When the memtable grows beyond a threshold, it is written to an SSTable in GFS and a new memtable is created. This reduces memory consumption and the amount of the log required for recovery if this server dies.

merging compaction To bound the number of SSTables created by minor compactions, the current memtable and a few SSTables are periodically merged into a single SSTable; the old tables are then deleted.

major compaction Rarely, SSTables containing deletion events are cleaned up to actually perform the delete and forget about the deletion event, purging old data from the system to reclaim resources.

Implementation details

- In addition to grouping columns together in families for access control and locality, column families can be grouped into *locality groups*. This is actually the level at which tables are split into parts and separate SSTables are generated for each. Because of this split, a client access to link information in Webtable, say, need not touch the SSTables for the page content information at all.
- Clients can control if compression is used, and what kind is used, at the granularity of locality groups; Google applications often use a two-pass compression scheme for high-level and low-level repetition, taking advantage of the assumption that similar row keys will contain similar information and be sorted nearby.

Implementation details

- Tablet servers cache key-value pairs returned by the SSTable interface (Scan Cache) to improve performance to applications that read the same data repeatedly.
- Tablet servers also have a lower-leve cache of the SSTable blocks read from GFS (Block Cache) to improve performance to applications that read nearby data repeatedly.
- The immutability of SSTables was deliberate and is exploited by not requiring synchronization and allowing sharing when tablets are split. The memtable does require synchronization, which is implemented by making each row copy-on-write and allowing parallel reads and writes.

Implementation details

- Bloom filters can be optionally used to reduce the SSTables that must be fetched and searched for read operations.
- To avoid keeping many small files in GFS and to improve disk locality, commit logs are kep per table rather than per-tablet; for recovery, this log must be sorted to group by tablet and then read and divided among tablet servers in one pass to avoid hammering the whole file once for each tablet.
- Minor compaction is always applied before re-assigning tablets so that the new server need not consult the log to reconstruct a memtable.

Evaluation

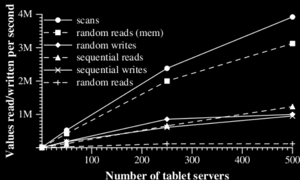
The authors evaluate Bigtable under several workloads:

- Writing to random rows.
- Reading from random rows.
- Reading from random rows of a locality group marked to be served from memory.
- Writing to sequential rows.
- Reading from sequential rows.
- Reading from sequential rows using the iterator interface.

Evalution

Benchmarks were evaluated on increasing cluster sizes to test scalability:

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843



Evaluation

Additionally, Google provides hard numbers for the properties of some Bigtable tables actually in use by several Google applications:

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

Related Work

- Boxwood
- Distributed Hash Tables
- Industry parallel databases from Oracle and IBM
- Column-oriented databases
  - C-Store
- Log-Structured Merge Tree
- Sawzall for scripting
  - Pig
- HBase

Conclusions

- There was a design trend to discover new and unexpected failures and remove assumptions.
- Adding features only when use cases arise may allow eliding them completely, e.g. Transactions.
- Monitoring proves crucial.
- Adapting data model to applications provides huge flexibility gains.
- Adapting applications to data model provides performance and homogeneity.

Boxwood

Overview

Provide abstractions that commonly underly storage applications

- File systems
- Databases

Overview

Get tricky distributed abstractions correct once

- B-tree
- Chunk store
- Infrastructure
  - Paxos service
  - Lock service
  - Transaction service
- Failure detection
- Replicated Logical Devices (RLDevs)
- Transactions and logging

System Model

- Set of "server nodes"
  - Rack-mounted servers
  - Disk units
  - Disk controllers
- High-speed (Gigabit ethernet) network
  - Synchronous replication
- Trusted network, fail-stop nodes
  - RPC without privacy or authentication

Paxos

- General-purpose
  - Client-defined states and "decrees"
- Runs on separate machines (typically 3)
- No dedicated Paxos servers
- Sequential decrees
- Failure detection for liveness

Failure detection

- Clocks need not be synchronized; assume
  - Clocks go forwards
  - Clock drift is bounded
  - UDP message delay is non-zero
- Each node sends keepalive beacons to set of observers
- When a majority fail to receive beacons, the node is announced dead

Lock Service

- Related work to Chubby, but lower-level
- Locks identified with byte arrays
- No timeouts on locks, but failure detector enforces liveness
    - Clients can register recovery function to be finished before lock is released on failure
- Master + 1 or more slaves
    - Master, selected by Paxos, handles all locks
    - If master dies, a slave takes over
    - More complicated (scalable) system not needed

Replicated Logical Devices

- Logical block device
- Replication at this low-level to avoid reimplementation at higher levels
  - Primary/secondary with synchronous replication

Chunk Manager

- Fundamental storage unit is the chunk
  - Sector-aligned consecutive bytes on an RLDev
  - Identified by globally-unique opaque handle
- For simplicity, 1 chunk manager per RLDev
  - Mapping from chunks to RLDev offsets persistent in RLDev
  - Primary/secondary with syncronous replication

B-tree

- B-link tree
  - B*-tree
  - Extra next pointer for efficient concurrent operations
- Built on top of Chunk Manager

BoxFS

- Distributed NFSv2 service based on B-trees
  - Directories, files, symbolic links stored as B-trees
- Directories
  - Keyed by child name
  - Data is wire representation of NFS file handle
- Files
  - Keyed by block number
  - Data is uninterpreted block
- Symbolic links
  - Single special key
  - Data is contents of the link

Conclusions

- Built abstractions for building storage systems rather than building a particular storage system
- Built proof of concept file service
- Made many simplifying assumptions enabling simple designs