Bigtable: A Distributed Storage System for Structured Data

Alvanos Michalis

April 6, 2009

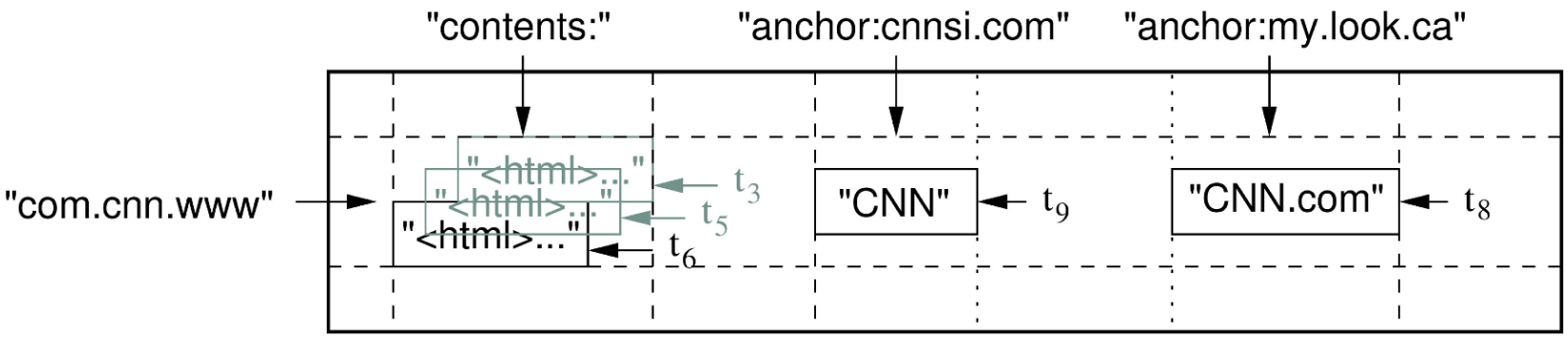
Google!

- Lots of Different kinds of data!
  - Crawling system URLs, contents, links, anchors, page-rank etc
  - Per-user data: preferences, recent queries/ search history
  - Geographic data, images etc ...
- Many incoming requests
- No commercial system is big enough
  - Scale is too large for commercial databases
  - May not run on their commodity hardware
  - No dependence on other vendors
  - Optimizations
  - Better Price/Performance
  - Building internally means the system can be applied across many projects for low incremental cost

Google goals

- Fault-tolerant, persistent
- Scalable
  - 1000s of servers
  - Millions of reads/writes, efficient scans
- Self-managing
- Simple!

Bigtable



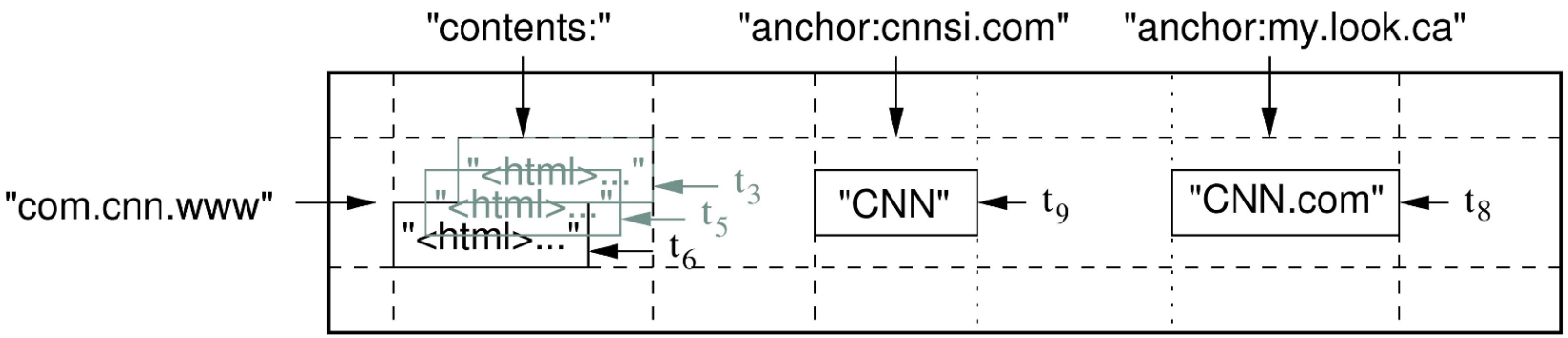
Definition

A Bigtable is a sparse, distributed, persistent multidimensional 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.

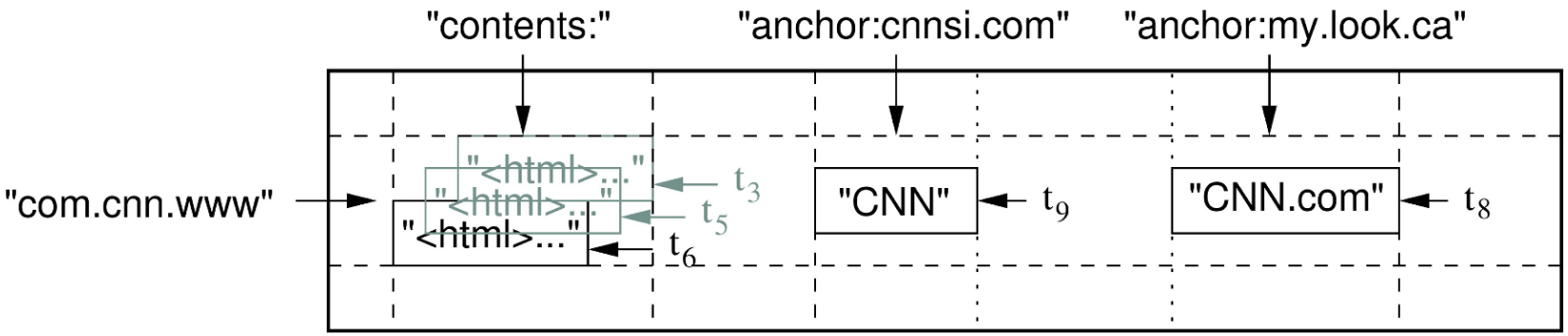
```
(row:string, column:string, time:int64) -> string
```

Rows



- The *row* keys in a table are arbitrary strings
- Every read or write of data under a single row key is atomic
- maintains data in lexicographic order by row key

Column Families



- Grouped into sets called *column families*
- All data stored in a column family is usually of the same type
- A column family must be created before data can be stored under any column key in that family
- A column key is named using the following syntax: *family:qualifier*

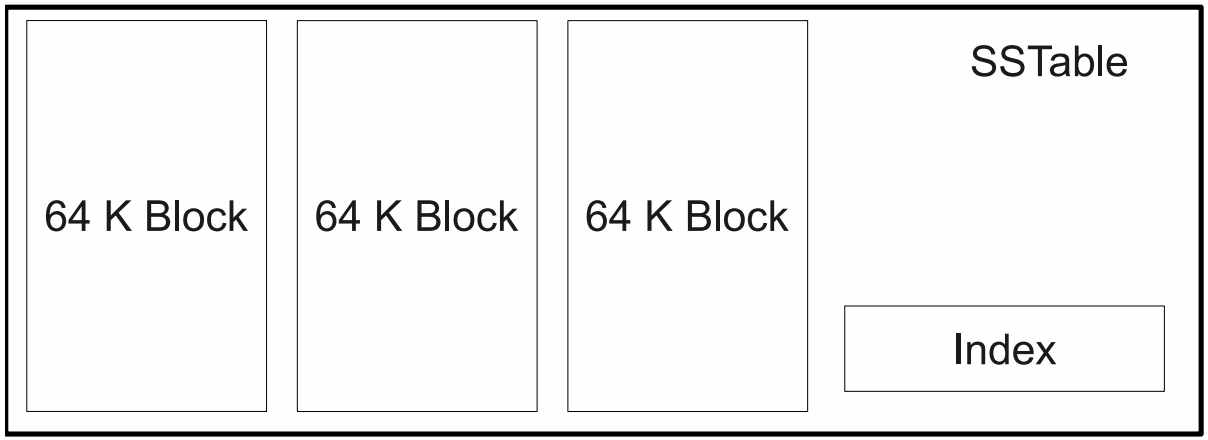
Timestamps

- Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by *timestamp* (64-bit integers).
- Applications that need to avoid collisions must generate unique timestamps themselves.
- To make the management of versioned data less onerous, they support two per-column-family settings that tell Bigtable to garbage-collect cell versions automatically.

Infrastructure

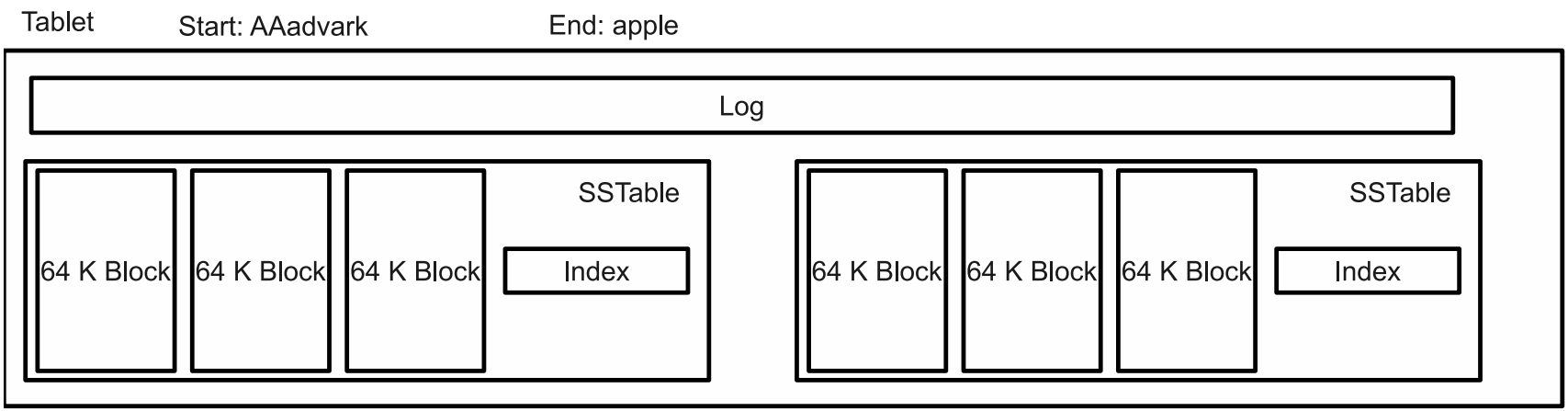
- Google WorkQueue (scheduler)
- GFS: large-scale distributed file system
  - Master: responsible for metadata
  - Chunk servers: responsible for r/w large chunks of data
  - Chunks replicated on 3 machines; master responsible
- Chubby: lock/file/name service
  - Coarse-grained locks; can store small amount of data in a lock
  - 5 replicas; need a majority vote to be active

SSTable



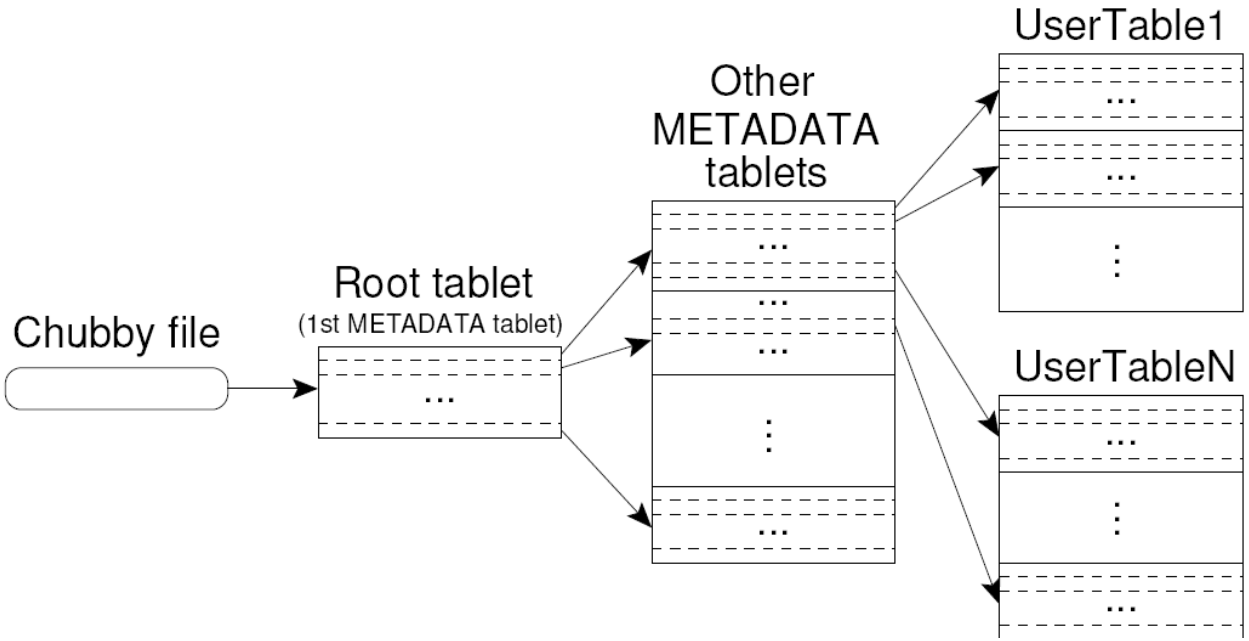
- Lives in GFS
- Immutable, sorted file of key-value pairs
- Chunks of data plus an index
- Index is of block ranges, not values

Tablet Design



- Large tables broken into tablets at row boundaries
  - Tablets hold contiguous rows
  - Approx 100 200 MB of data per tablet
- Approx 100 tablets per machine
  - Fast recovery
  - Load-balancing
- Built out of multiple SSTables

Tablet Location

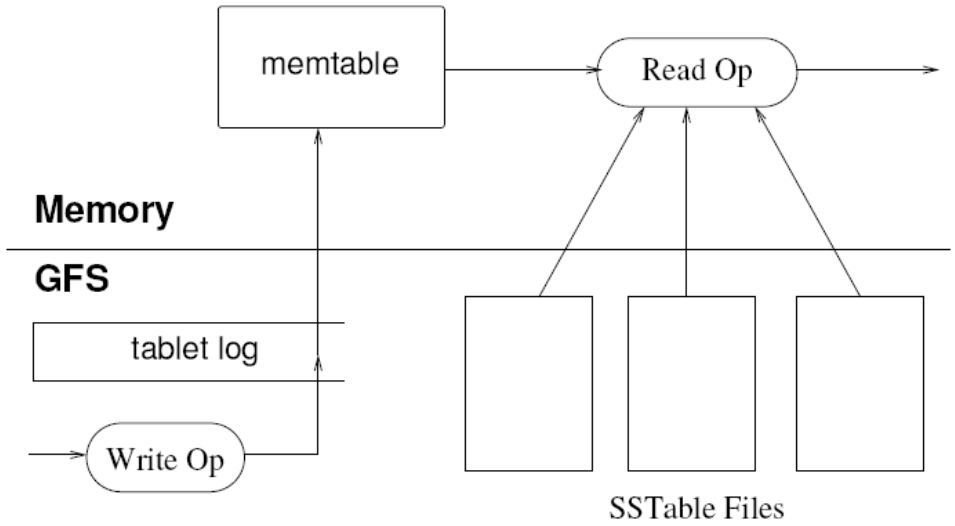


- Like a B+-tree, but fixed at 3 levels
- How can we avoid creating a bottleneck at the root?
  - Aggressively cache tablet locations
  - Lookup starts from leaf (bet on it being correct); reverse on miss

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.
- Bigtable uses Chubby to keep track of tablet servers. When a tablet server starts, it creates, and acquires an exclusive lock on, a uniquely-named file in a specific Chubby directory.
- Tablet server stops serving its tablets if loses its exclusive lock
- The master is responsible for detecting when a tablet server is no longer serving its tablets, and for reassigning those tablets as soon as possible.
- When a master is started by the cluster management system, it needs to discover the current tablet assignments before it can change them.

Serving a Tablet



- Updates are logged
- Each SSTable corresponds to a batch of updates or a snapshot of the tablet taken at some earlier time
- Memtable (sorted by key) caches recent updates
- Reads consult both memtable and SSTables

Compactions

As write operations execute, the size of the memtable increases.

- Minor compaction convert the memtable into an SSTable
  - Reduce memory usage
  - Reduce log traffic on restart
- Merging compaction
  - Periodically executed in the background
  - Reduce number of SSTables
  - Good place to apply policy keep only N versions
- Major compaction
  - Merging compaction that results in only one SSTable
  - No deletion records, only live data
  - Reclaim resources.

Refinements (1/2)

- Group column families together into an SSTable. Segregating column families that are not typically accessed together into separate locality groups enables more efficient reads.
- Can compress locality groups, using Bentley and McIlroy's scheme and a fast compression algorithm that looks for repetitions.
- Bloom Filters on locality groups allows to ask whether an SSTable might contain any data for a specified row/column pair. Drastically reduces the number of disk seeks required - for non-existent rows or columns do not need to touch disk.

Refinements (2/2)

- Caching for read performance (two levels of caching)
  - Scan Cache: higher-level cache that caches the key-value pairs returned by the SSTable interface to the tablet server code.
  - Block Cache: lower-level cache that caches SSTables blocks that were read from GFS.
- Commit-log implementation
- Speeding up tablet recovery (log entries)
- Exploiting immutability

Hardware Environment

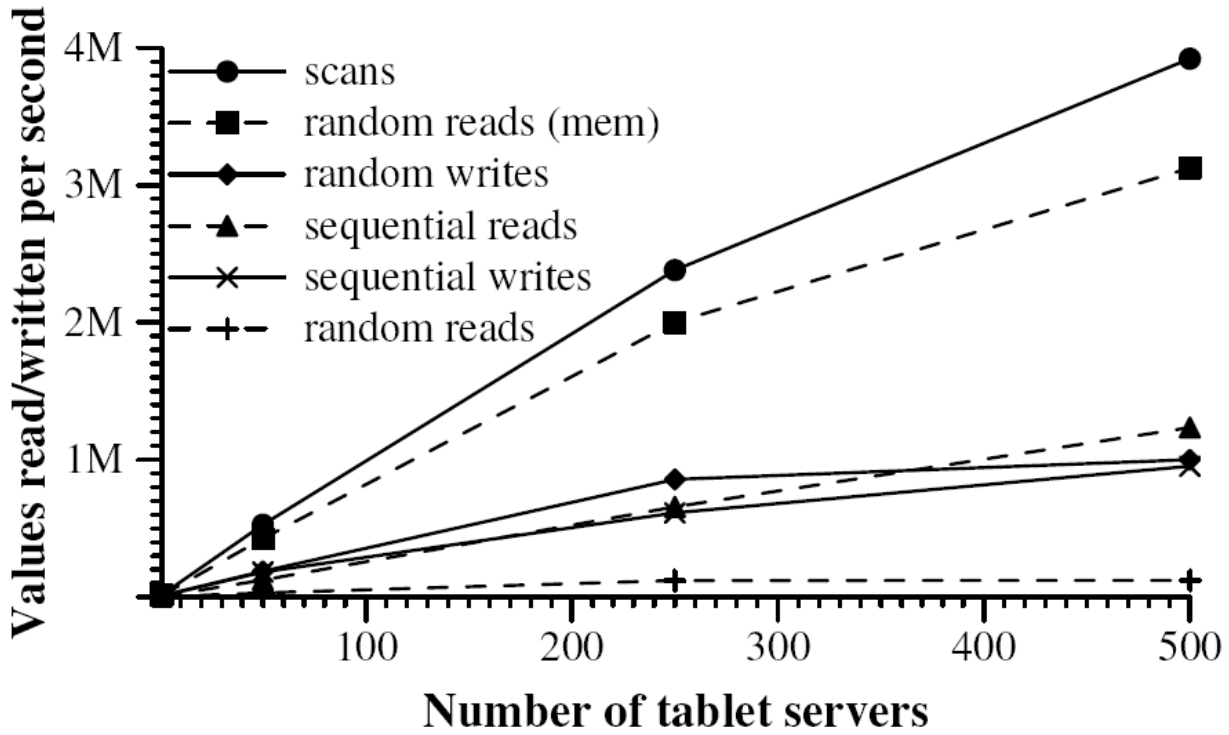
- Tablet servers were configured to use 1 GB of memory and to write to a GFS cell consisting of 1786 machines with two 400 GB IDE hard drives each.
- Each machine had two dual-core Opteron 2 GHz chips
- Enough physical memory to hold the working set of all running processes
- Single gigabit Ethernet link
- Two-level tree-shaped switched network with 100-200 Gbps aggregate bandwidth at the root.

Results Per Tablet Server

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

Number of 1000-byte values read/written per second.

Results Aggregate Rate



Number of 1000-byte values read/written per second.

Single tablet-server performance

- The tablet server executes 1200 reads per second (75 MB/s), enough to saturate the tablet server CPUs because of overheads in networking stack
- Random and sequential writes perform better than random reads (commit log and uses group commit)
- No significant difference between random writes and sequential writes (same commit log)
- Sequential reads perform better than random reads (block cache)

Scaling

- Aggregate throughput increases dramatically performance of random reads from memory increases
- However, performance does not increase linearly
- Drop in per-server throughput
  - Imbalance in load: Re-balancing is throttled to reduce the number of tablet movement and the load generated by benchmarks shifts around as the benchmark progresses
  - The random read benchmark: transfer one 64KB block over the network for every 1000-byte read and saturates shared 1 Gigabit links

Timestamps

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

- Google Analytics
- Google Earth
- Personalized Search

Lessons learned

- Large distributed systems are vulnerable to many types of failures, not just the standard network partitions and fail-stop failures
  - Memory and network corruption
  - Large clock skew
  - Extended and asymmetric network partitions
  - Bugs in other systems (Chubby !)
  - ...
- Delay adding new features until it is clear how the new features will be used
- A practical lesson: the importance of proper system-level monitoring
- Keep It Simple!

END!

QUESTIONS ?