BigTable:
A System for Distributed Structured Storage

OSDI 2006

Fay Chang, Jeffrey Dean, Sanjay Ghemawatet.al.

BigTable: Motivation

- Lots of (semi-)structured data at Google –
  - URLs: Contents, crawl metadata(when, response code), links, anchors
  - Per-user Data: User preferences settings, recent queries, search results
  - Geographical locations: Physical entities – shops, restaurants, roads
- Scale is large
  - Billions of URLs, many versions/page - 20KB/page
  - Hundreds of millions of users, thousands of q/sec – Latency requirement
  - 100+TB of satellite image data

Why Not Commercial Database ?

- Scale too large for most commercial databases
- Even if it weren't, cost would be too high
- Building internally means low incremental cost
  - System can be applied across many projects used as building blocks.
- Much harder to do low-level storage/network transfer optimizations to help performance significantly.
  - When running on top of a database layer.

Goals

- Want asynchronous processes to be continuously updating different pieces of data
  - Want access to most current data at any time
- Need to support:
  - Very high read/write rates (millions of ops per second)
  - Efficient scans over all or interesting subsets of data
  - Efficient joins of large one-to-one and one-to-many datasets
- Often want to examine data changes over time
  - E.g. Contents of a web page over multiple crawls

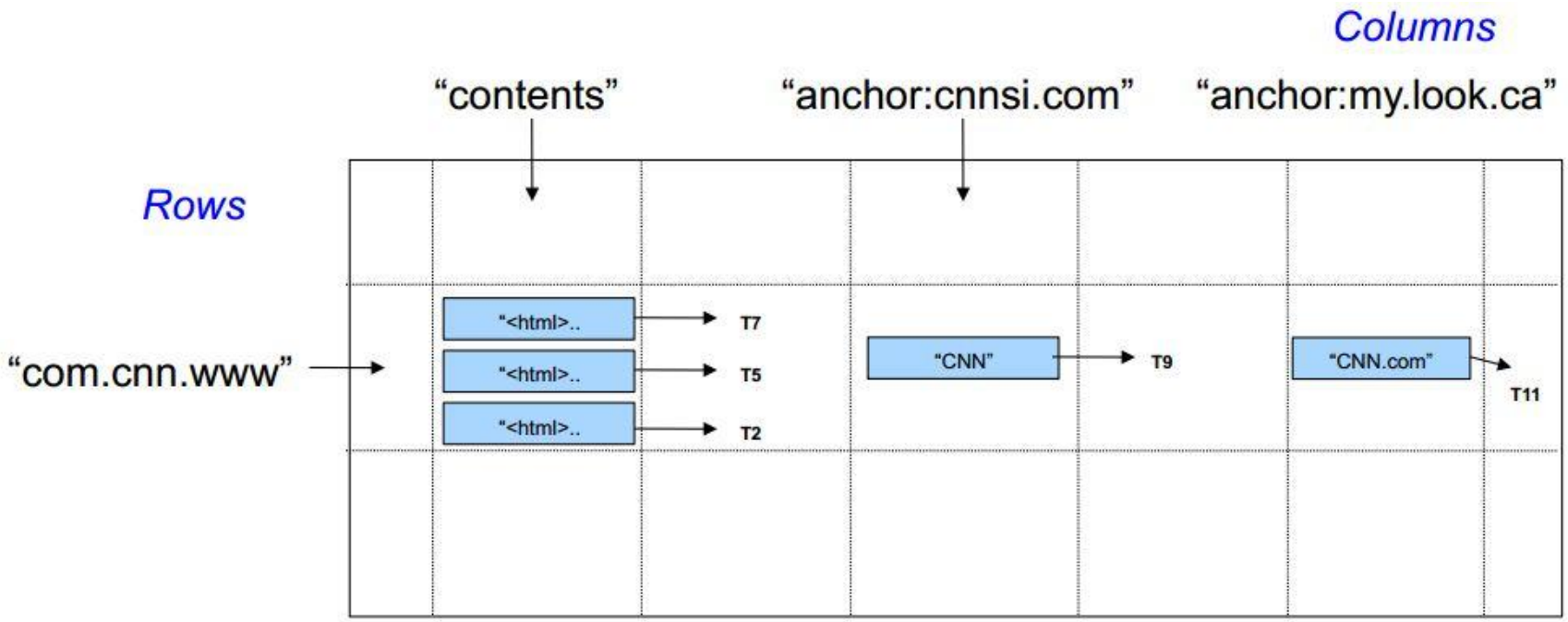
BigTable

- Distributed Multilevel Map
- Fault-tolerant, persistent
- Scalable
  - 1000s of servers
  - TB of in-memory data
  - Petabyte of disk based data
  - Millions of read/writes per second, efficient scans
- Self-managing
  - Servers can be added/removed dynamically
  - Servers adjust to the load imbalance

BigTable Overview

- Data Model
- API
- Building Blocks
- Implementation Structure
  - Tablets, compactions, locality groups, …
- Details
  - Shared logs, compression, replication, …

Basic Data Model



- Distributed multi-dimensional sparse map (row, column, timestamp) -> cell contents
- Good match for most of Googles applications

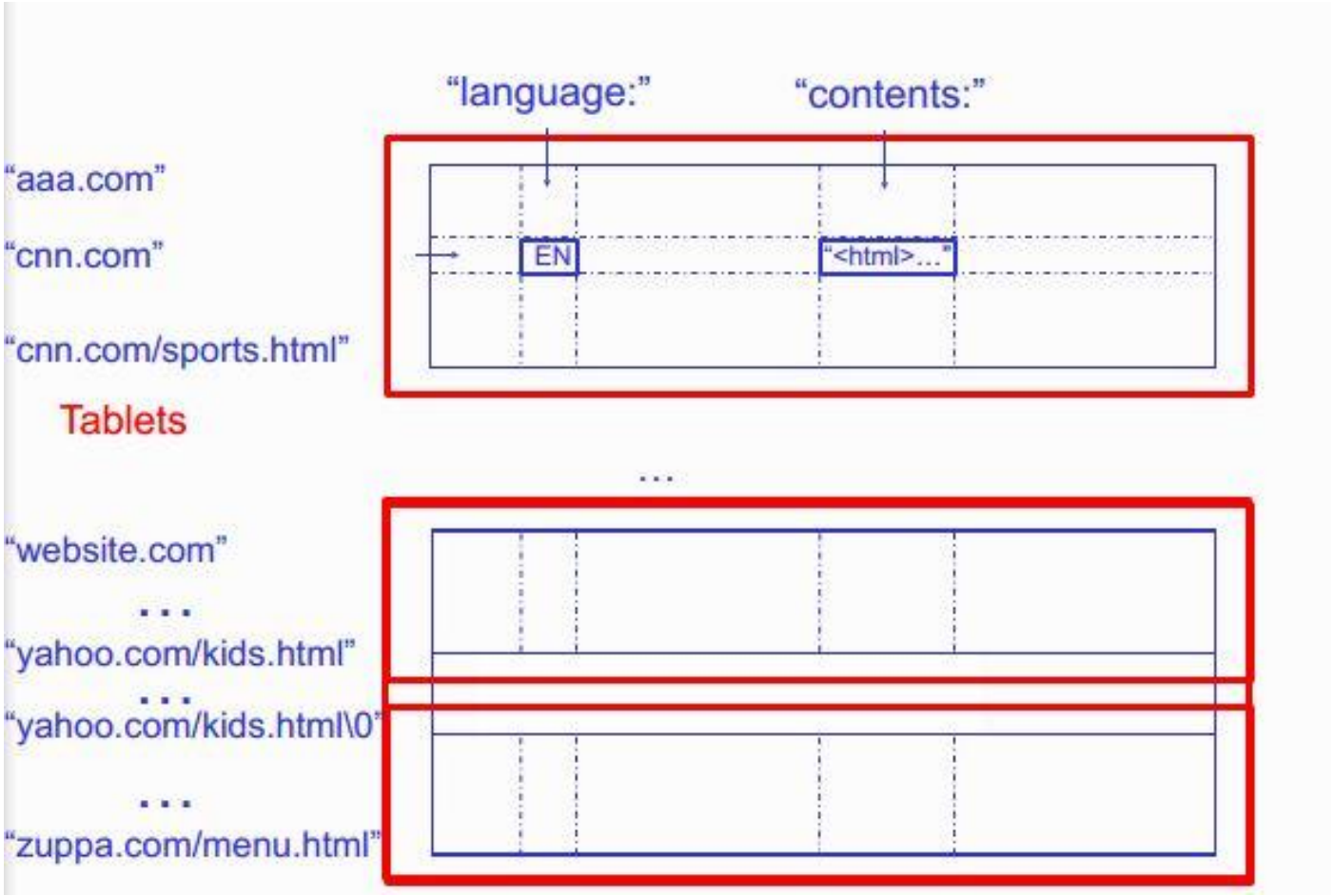
Rows

- Name is an arbitrary string.
  - Access to data in a row is atomic.
  - Row creation is implicit upon storing data.
  - Transactions within a row
- Rows ordered lexicographically
  - Rows close together lexicographically usually on one or a small number of machines.
- Does not support relational model
  - No table wide integrity constants
  - No multi row transactions

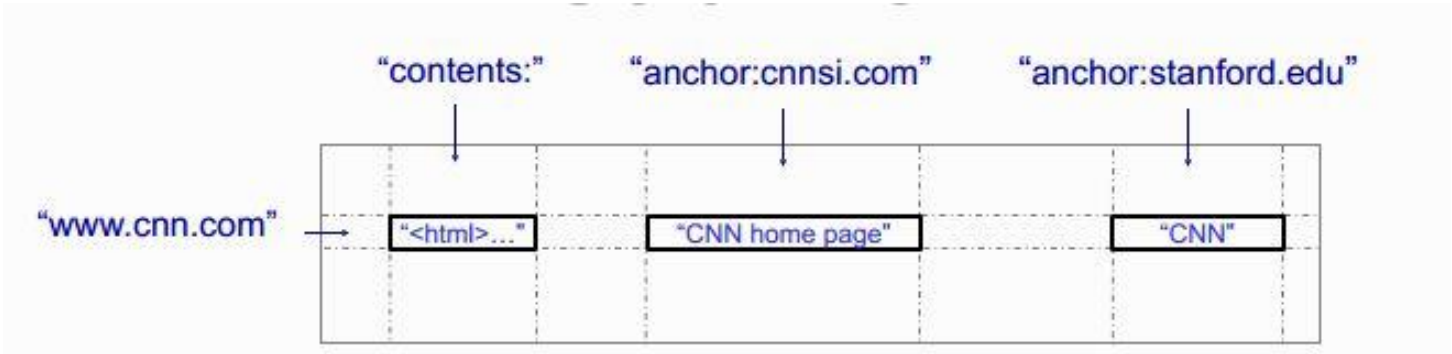
Tablets

- Large tables broken into tablets at row boundaries
  - Tablet holds contiguous range of rows
    - Clients can often choose row keys to achieve locality
  - Aim for ~100MB to 200MB of data per tablet
- Serving machine responsible for ~100 tablets
  - Fast recovery:
    - 100 machines each pick up 1 tablet from failed machine
  - Fine-grained load balancing:
    - Migrate tablets away from overloaded machine
    - Master makes load-balancing decisions

Tablets & Splitting



Columns



- Columns has two-level name structure:
  - family:optional_qualifier
- Column family
  - Unit of access control
  - Has associated type information
- Qualifier gives unbounded columns
  - Additional level of indexing, if desired

Timestamps

- Used to store different versions of data in a cell
  - New writes default to current time, but timestamps for writes can also be set explicitly by clients
- Look up options:
  - “Return most recent K values”
  - “Return all values in timestamp range(on all values)”
- Column families can be marked with attributes
  - “Only retain most recent K values in a cell”
  - “Keep values until they are older than K seconds”

API

- Metadata operations
  - Create/delete tables, column families, change metadata
- Writes (atomic)
  - Set():write cells in a row
  - DeleteCells():delete cells in a row
  - DeleteRow():delete all cells in a row
- Reads
  - Scanner:read arbitrary cells in a bigtable
    - Each row read is atomic
    - Can restrict returned rows to a particular range
    - Can ask for just data from 1 row, all rows, etc.
    - Can ask for all columns, just certain column families, or specific columns

API

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

Fig. 2. Writing to Bigtable.

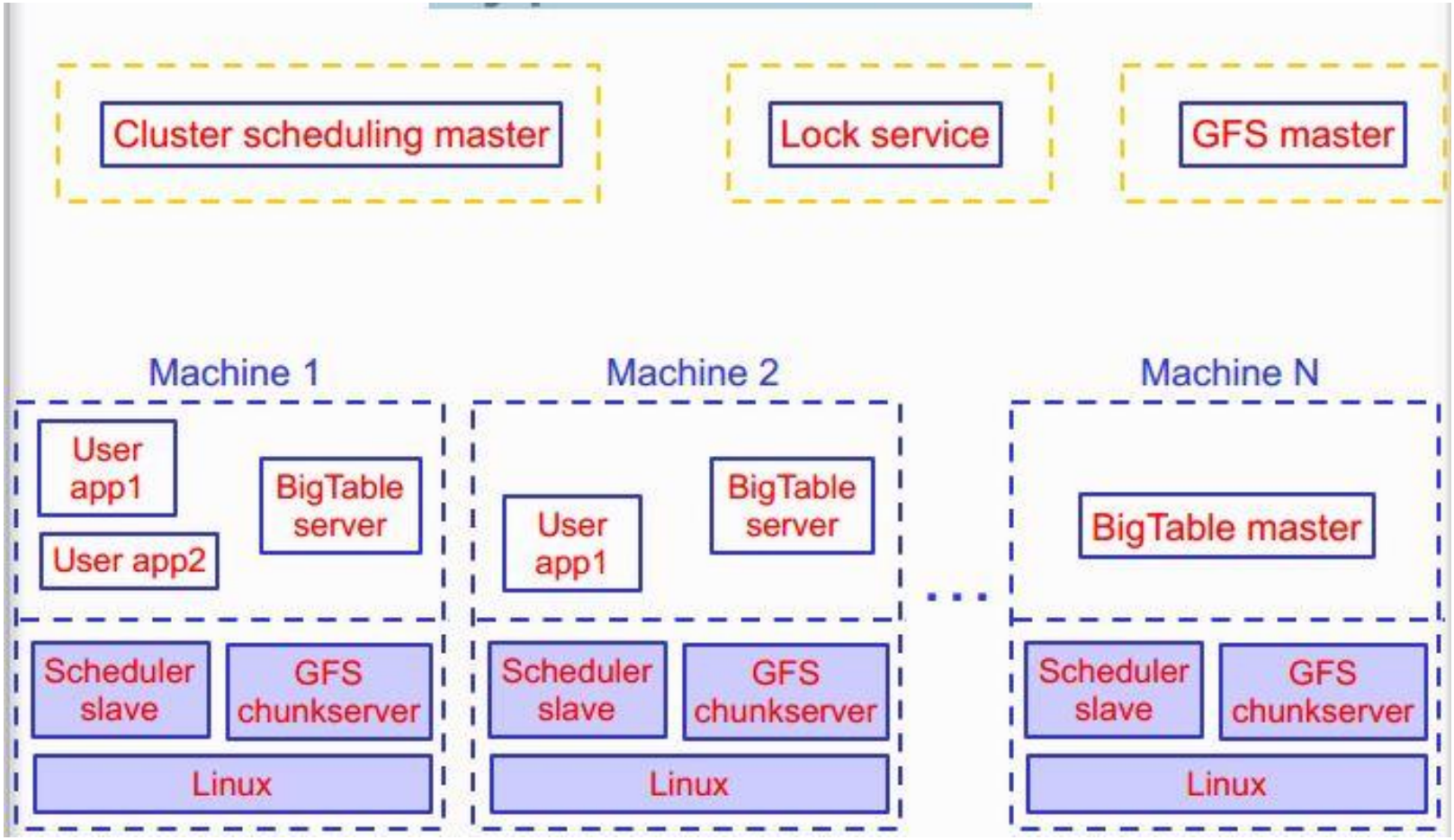
```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

Fig. 3. Reading from Bigtable.

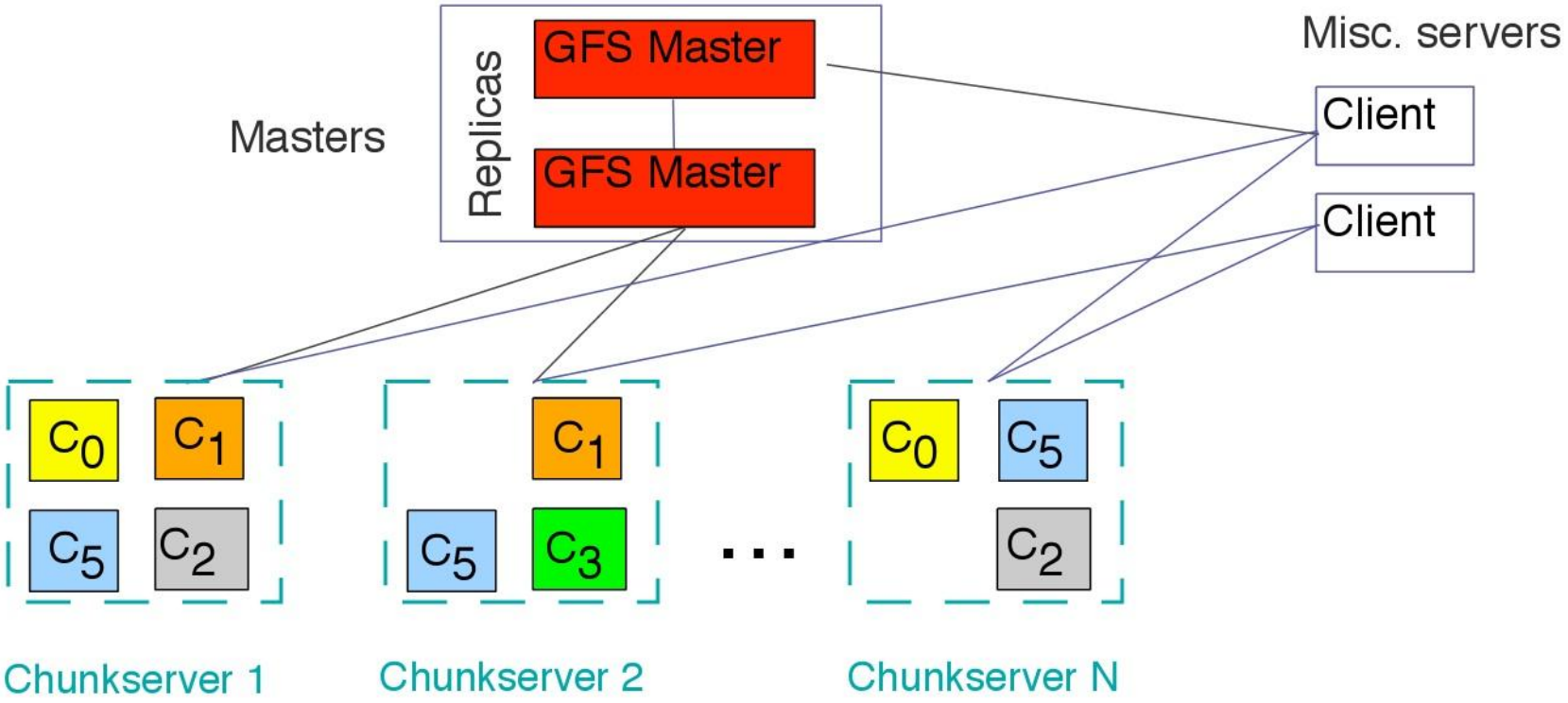
Building Blocks

- Building blocks:
  - Google File System (GFS): Raw storage
  - Scheduler: schedules jobs onto machines
  - Lock service: distributed lock manager
    also can reliably hold tiny files (100s of bytes) w/ high availability
  - MapReduce: simplified large-scale data processing
- BigTable uses of building blocks:
  - GFS: stores persistent state
  - Scheduler: schedules jobs involved in BigTable serving
  - Lock service: master election, location bootstrapping
  - MapReduce: often used to read/write BigTable data

Typical Cluster



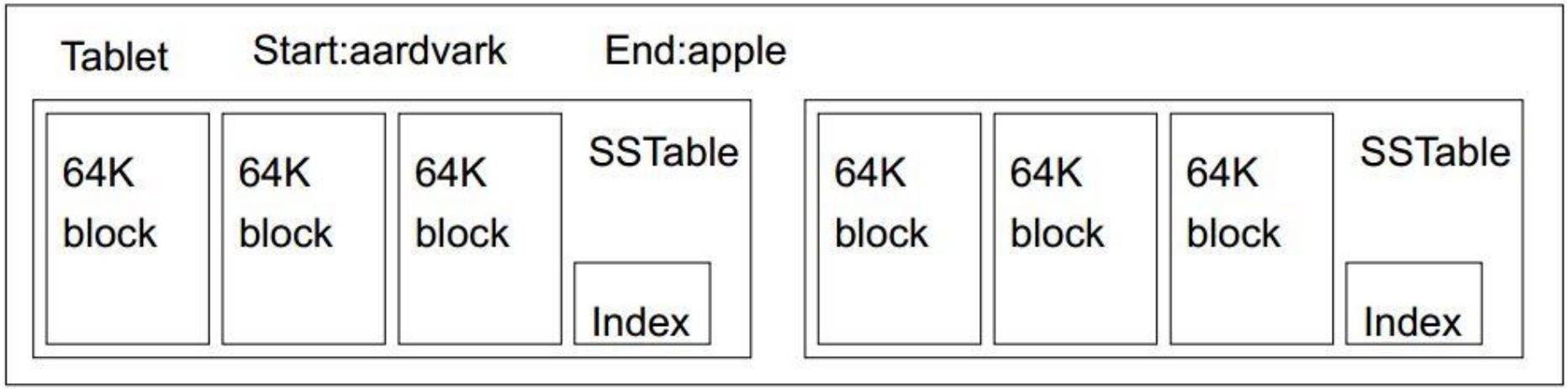
Google File System (GFS)



- Master manages metadata
- Data transfers happen directly between clients/chunkservers
- Files broken into chunks (typically 64 MB)
- Chunks triplicated across three machines for safety
- See SOSP'03 paper at http://labs.google.com/papers/gfs.html

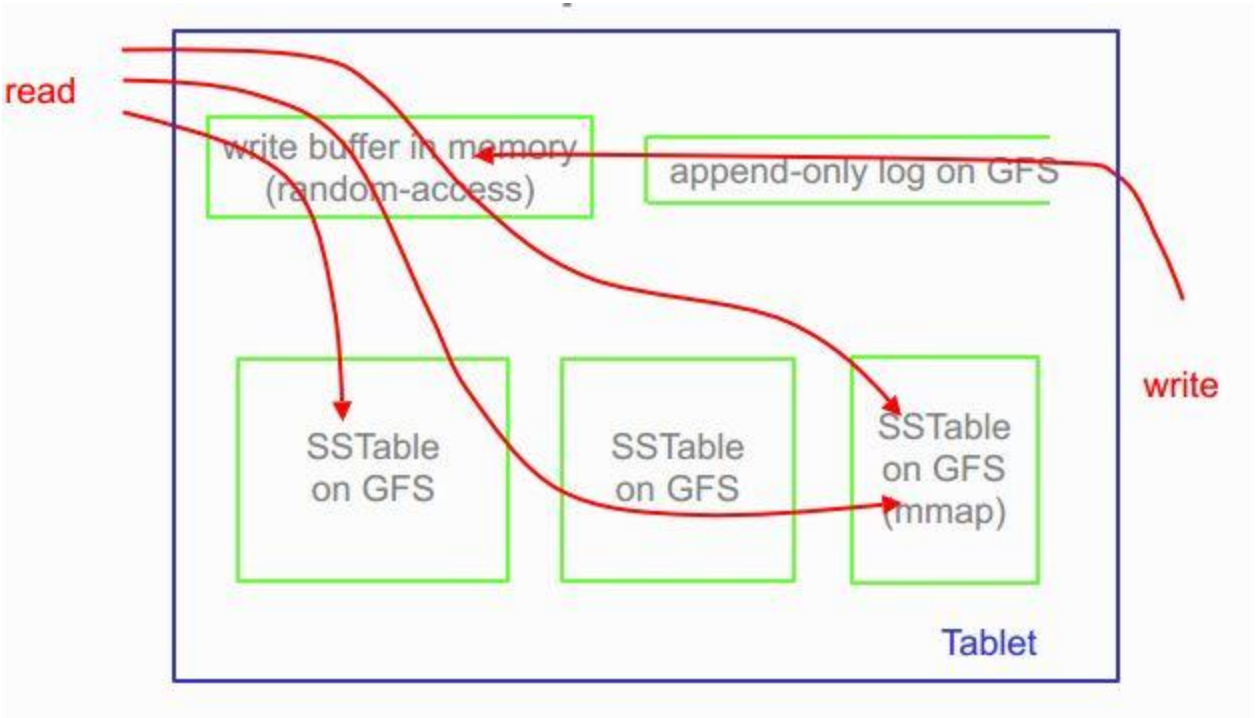
SS Table

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



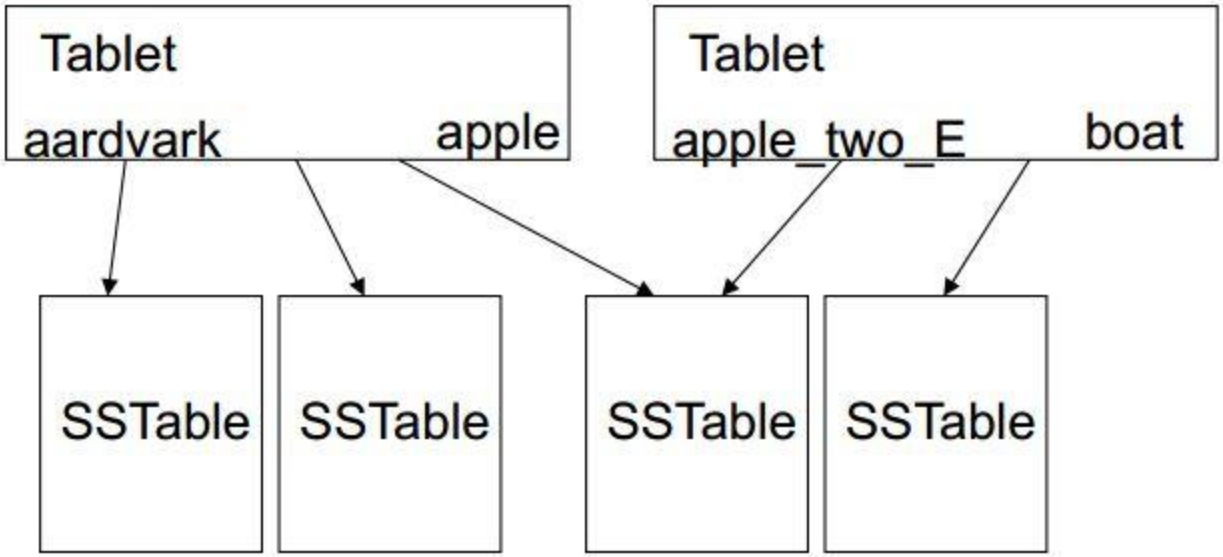
SSTable

- Immutable on-disk ordered map from string->string
- string keys: <row, column, timestamp>triples



SSTable

- Multiple tablets make up the table
- SSTablescan be shared
- Tablets do not overlap, SSTablescan overlap



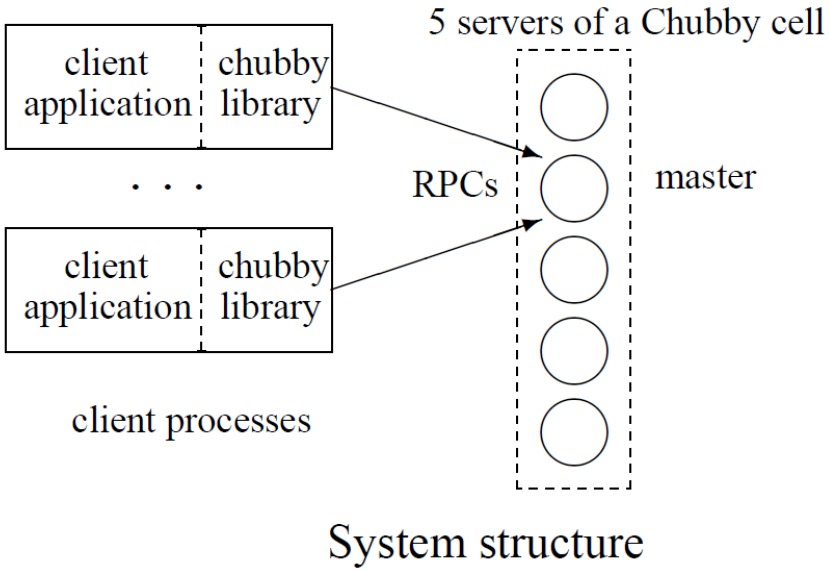
Chubby: Locking Service

- Highly available and persistent distributed lock service
- Five active replicas

  - one is elected master and actively serve requests

  - service is live when majority of the replicas are active and communicate with each other
- Provides a namespace consisting of directories and small files.

  - Each directory or file is used as a lock

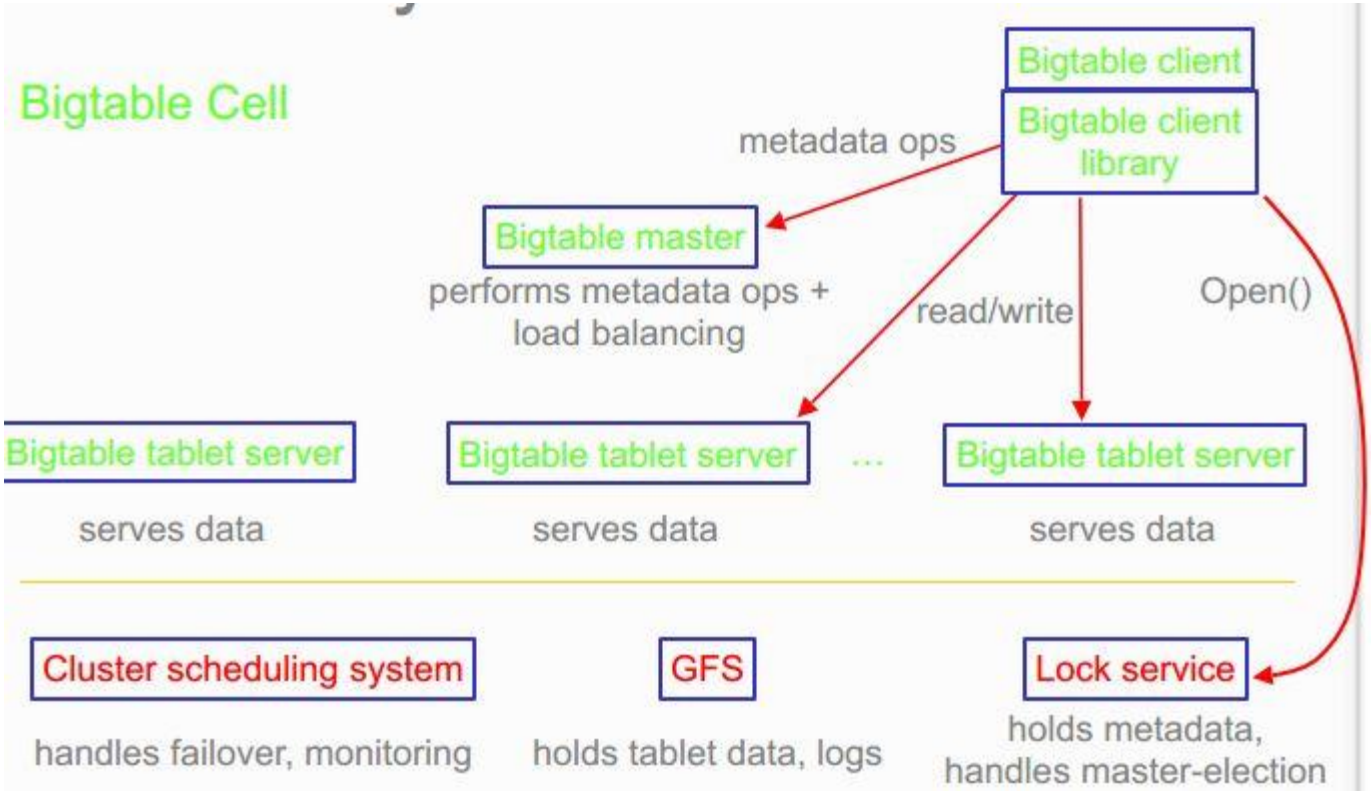


System structure

Bigtable and Chubby

- Bigtable uses Chubby to:
  - Ensure there is at most one active master at a time,
  - Store the bootstrap location of Bigtable data (Root tablet),
  - Discover tablet servers and finalize tablet server deaths,
  - Store Bigtable schema information (column family information),
  - Store access control list.
  - If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable

System Structure

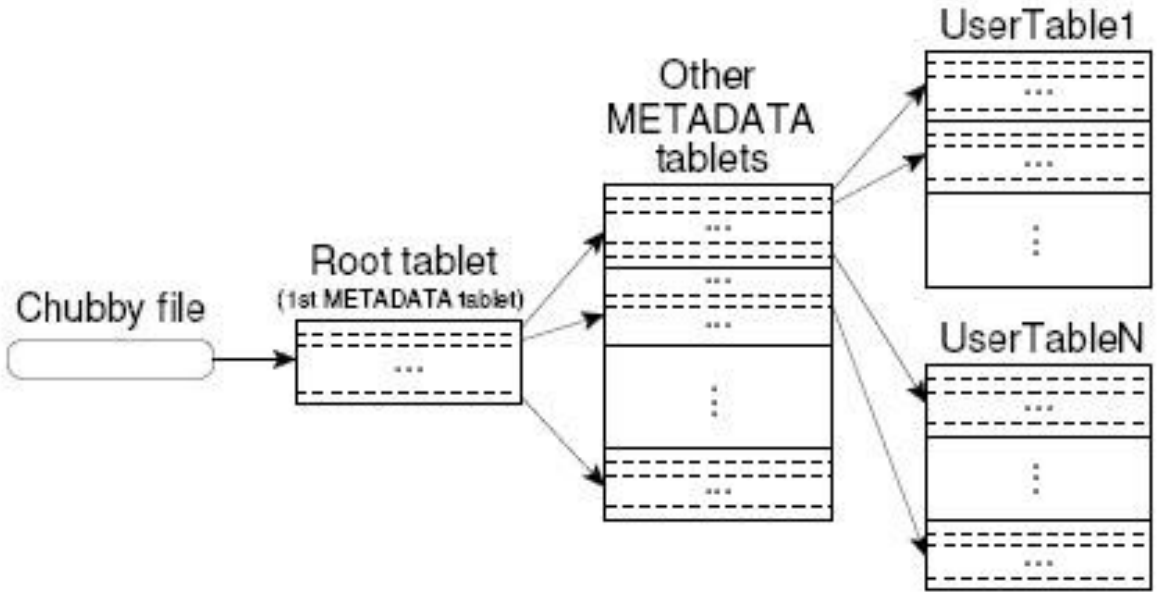


Implementation: Three major components

- A library that is linked into every client
- One master server
  - Assigning tablets to tablet servers
  - Detecting the addition and deletion of tablet servers
  - Balancing tablet-server load
  - Garbage collection of files in GFS
- Many tablet servers
  - Tablet servers manage tablets
  - Tablet server splits tablets that get too big
- Client communicates directly with tablet server for reads/writes.

Tablet Location

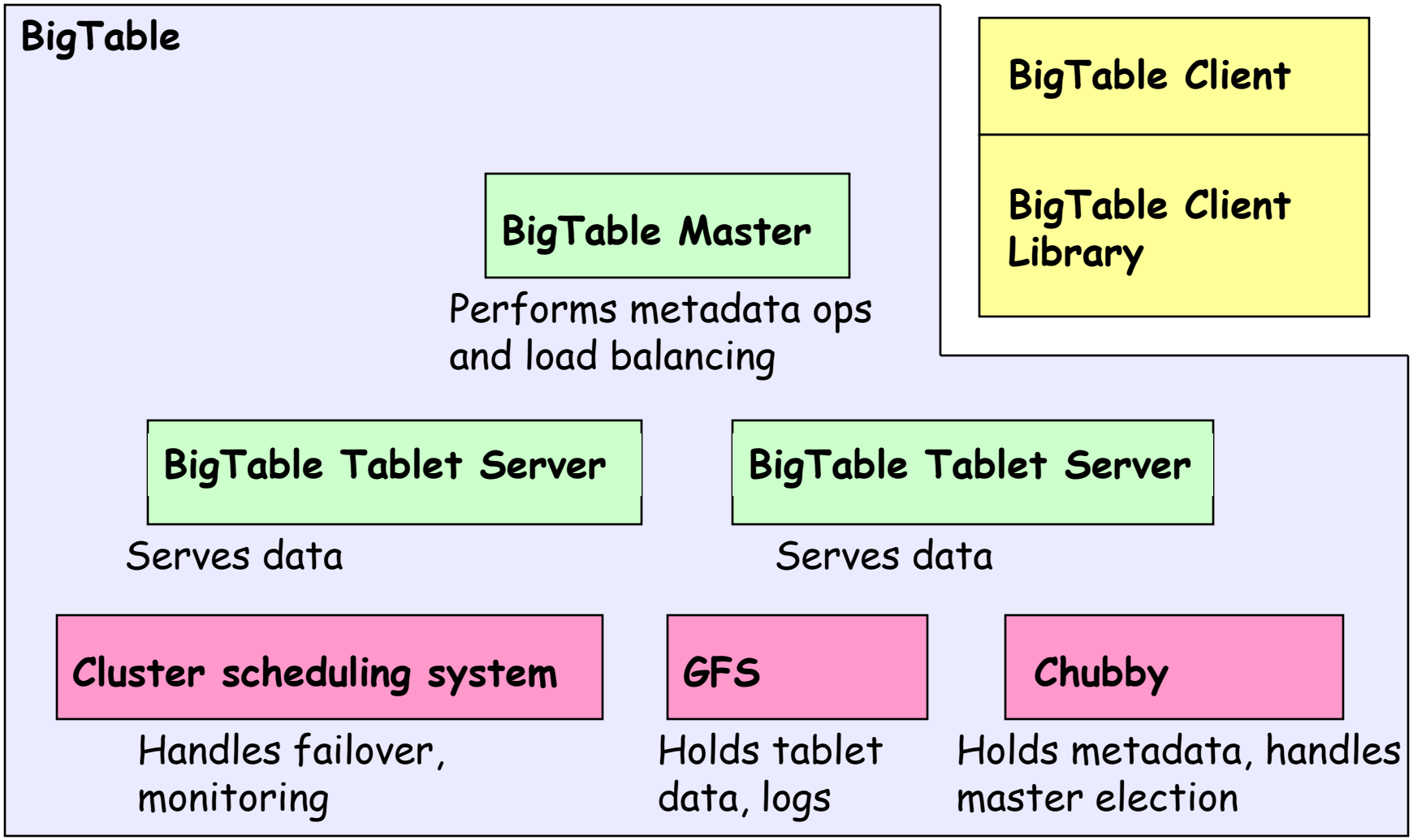
- Since tablets move around from server to server, given a row, how do clients find the right machine?
  - Need to find a tablet whose row range covers the target row



Tablet Location

- A 3-level hierarchy analogous to that of a B+-tree to store tablet location information :
  - A file stored in chubby contains location of the root tablet
  - Root tablet contains location of *Metadata tablets*
    - The root tablet never splits
  - Each meta-data tablet contains the locations of a set of user tablets
- Client reads the Chubby file that points to the root tablet
  - This starts the location process
- Client library caches tablet locations
  - Moves up the hierarchy if location N/A

Tablet Assignment



Tablet Server

- When a tablet server starts, it creates and acquires exclusive lock on, a uniquely-named file in a specific Chubby directory
  - Call this servers directory
- A tablet server stops serving its tablets if it loses its exclusive lock
  - This may happen if there is a network connection failure that causes the tablet server to lose its Chubby session

Tablet Server

- A tablet server will attempt to reacquire an exclusive lock on its file as long as the file still exists
- If the file no longer exists then the tablet server will never be able to serve again
  - Kills itself
  - At some point it can restart; it goes to a pool of unassigned tablet servers

Master Startup Operation

- Upon start up the master needs to discover the current tablet assignment.
  - Grabs unique master lock in Chubby
    - Prevents concurrent master instantiations
  - Scans servers directory in Chubby for live servers
  - Communicates with every live tablet server
    - Discover all tablets
  - Scans METADATA table to learn the set of tablets
    - Unassigned tablets are marked for assignment

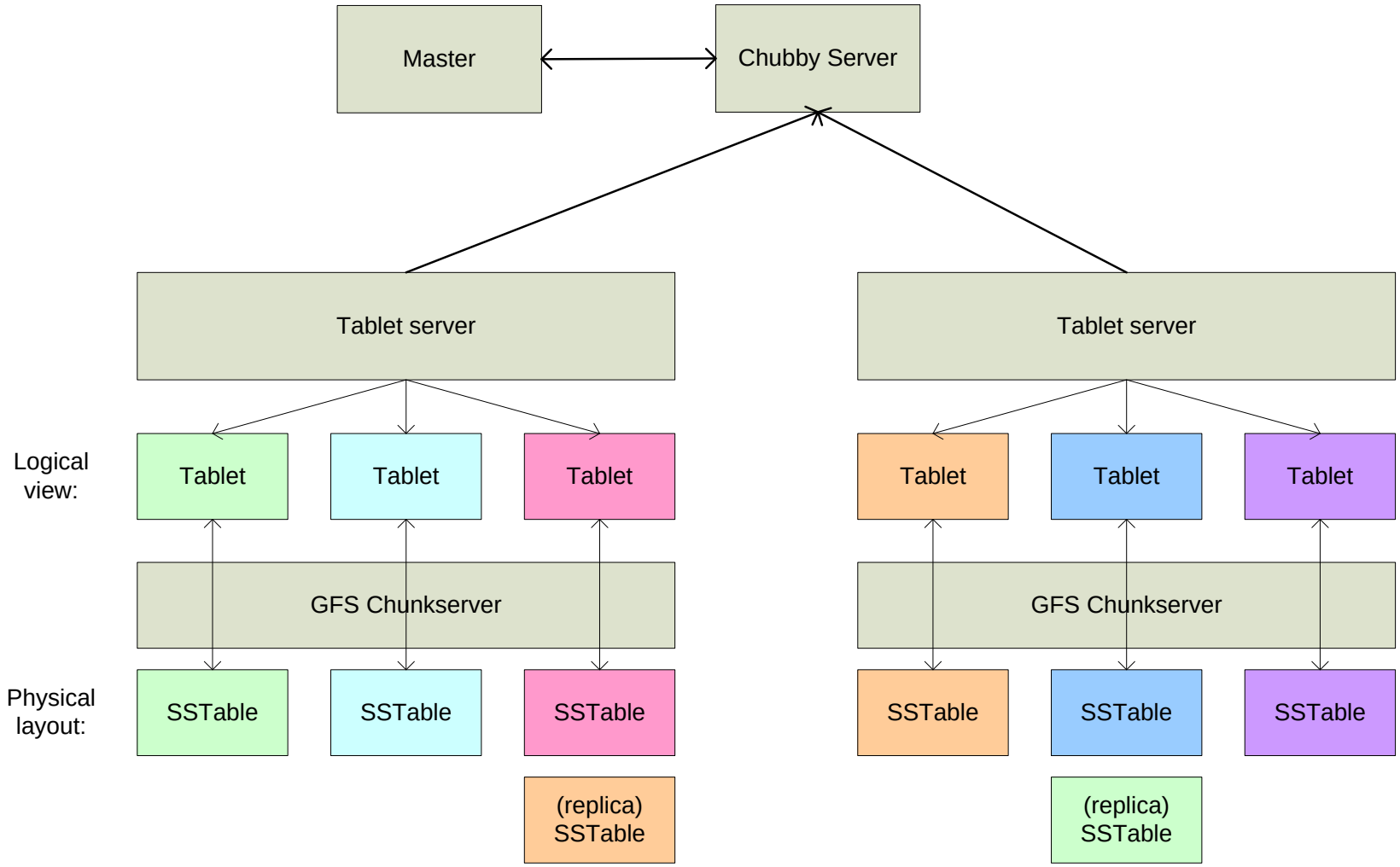
Master Operation

- Detect tablet server failures/resumption
- Master periodically asks each tablet server for the status of its lock

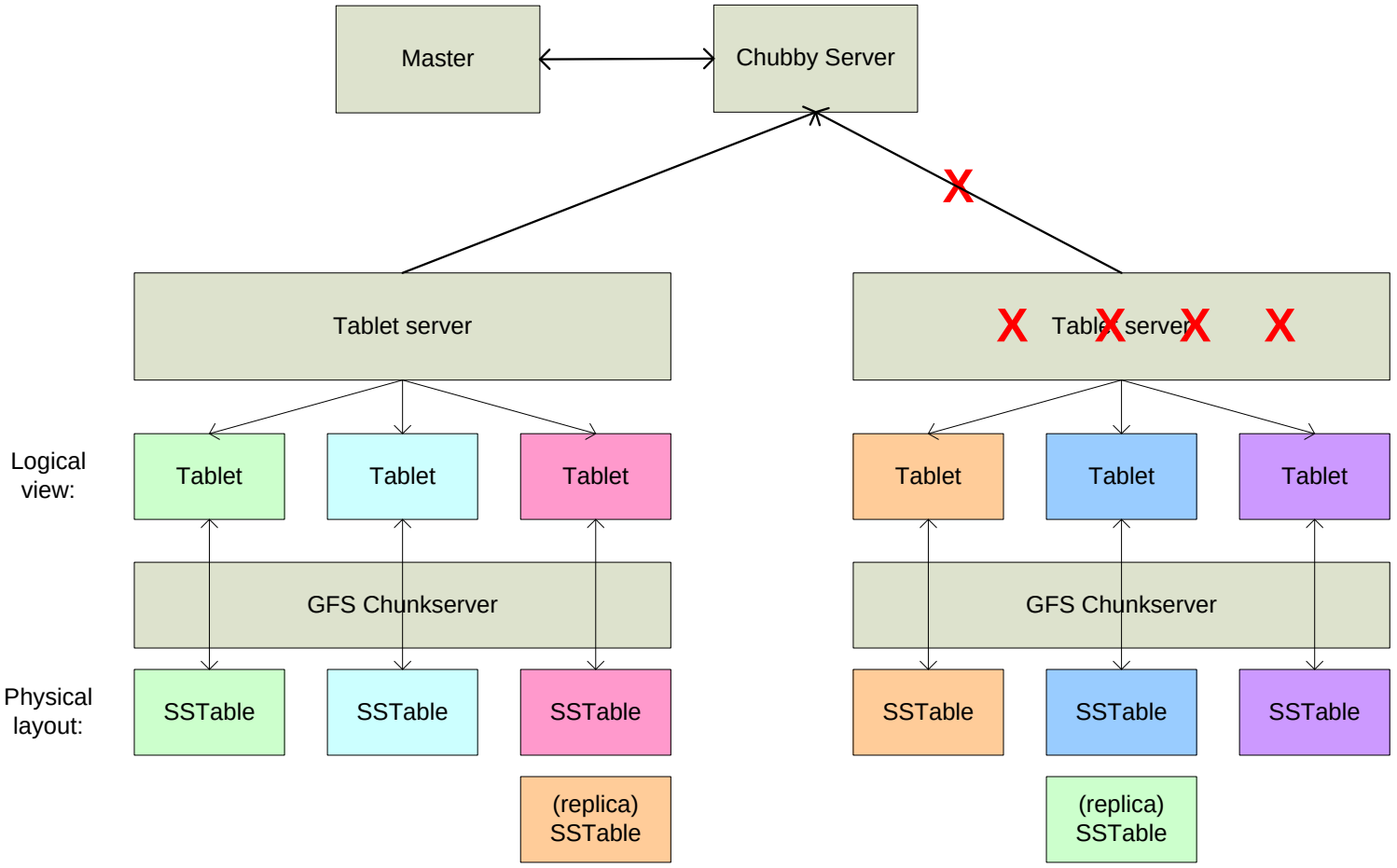
Master Operation

- Tablet server lost its lock or master cannot contact tablet server:
  - Master attempts to acquire exclusive lock on the server's file in the servers directory
  - If master acquires the lock then the tablets assigned to the tablet server are assigned to others
    - Master deletes the server's file in the servers directory
  - Assignment of tablets should be balanced
- If master loses its Chubby session then it kills itself
  - An election can take place to find a new master

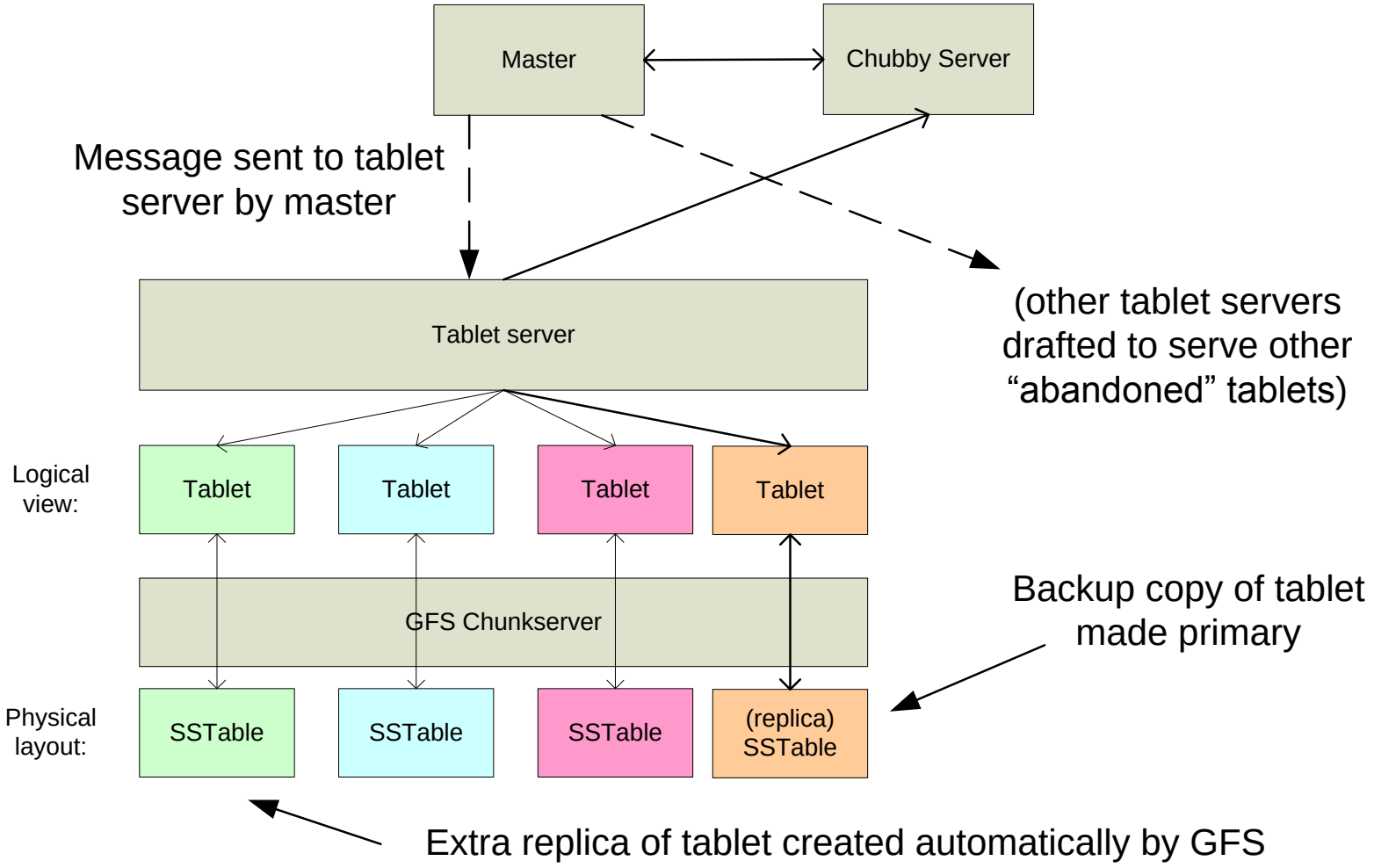
Tablet Server Failure



Tablet Server Failure

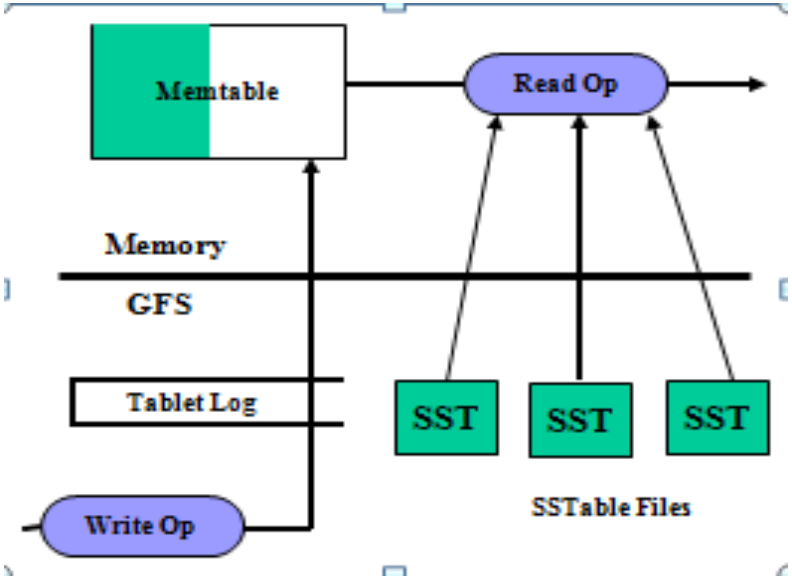


Tablet Server Failure



Tablet Serving

- Commit log stores the updates that are made to the data
- Recent updates are stored in memtable
- Older updates are stored in SStable files



Tablet Serving

- Recovery process
- Reads/Writes that arrive at tablet server

  - Is the request Well-formed?

  - Authorization: Chubby holds the permission file

  - If a mutation occurs it is wrote to commit log and finally a group commit is used

Tablet Serving

- Tablet recovery process
  - Read metadata containing SSTABLES and redo points
  - Redo points are pointers into any commit logs
  - Apply redo points

Compactions

- Minor compaction – convert the memtable into an SSTable
  - Reduce memory usage
  - Reduce log traffic on restart
- Merging compaction
  - 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

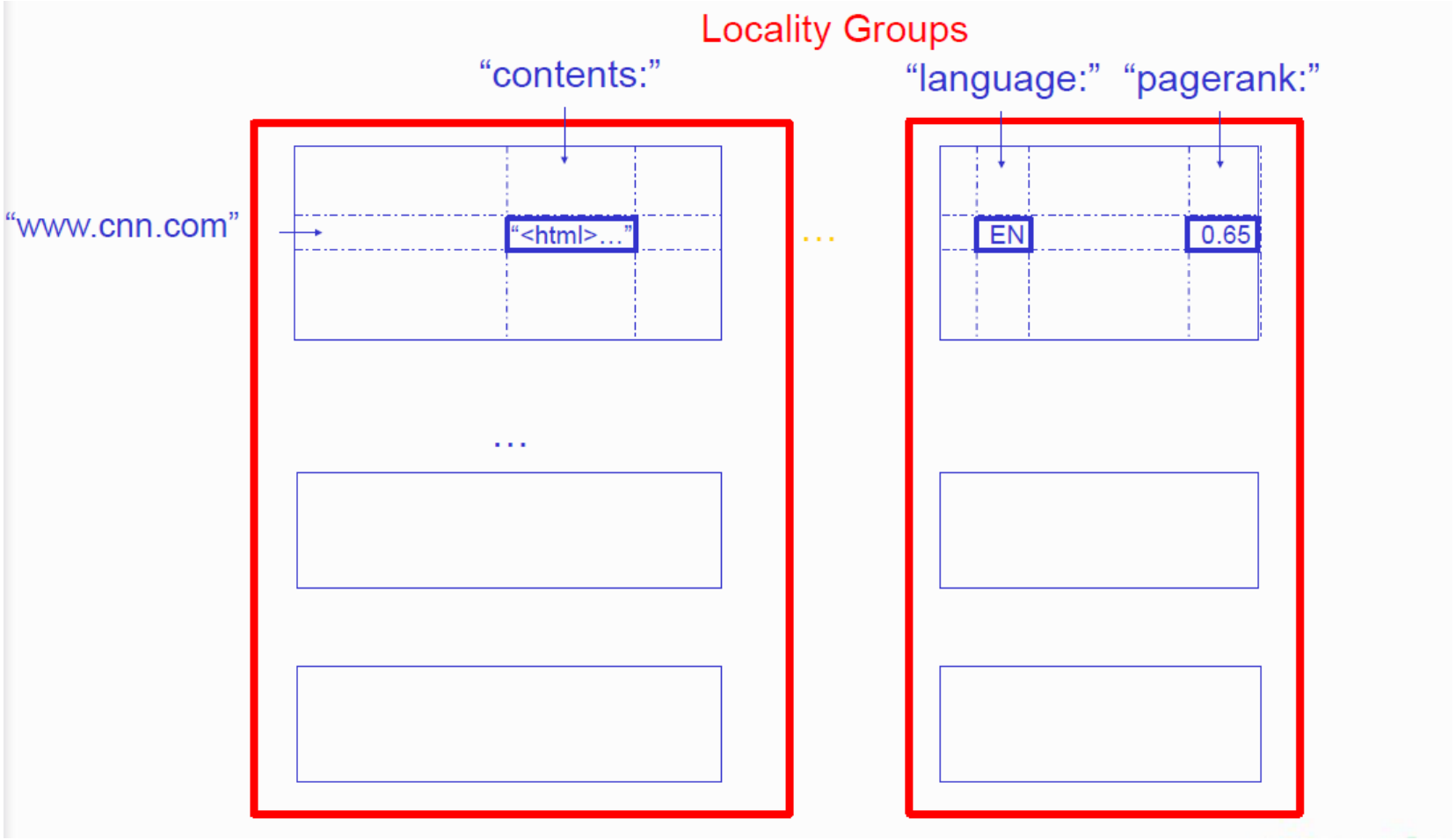
Schema Management

- Bigtable schemas are stored in Chubby.
- Chubby provides atomic whole-file writes and consistent caching of small files.
- Schema is made consistent by Master and can be accessed by tablet servers

Refinements: Locality Groups

- Can group multiple column families into a *locality group*
  - Separate SSTable is created for each locality group in each tablet.
- Segregating columns families that are not typically accessed together enables more efficient reads.
  - In WebTable, page metadata can be in one group and contents of the page in another group.

Locality Groups



Refinements: Compression

- Many opportunities for compression
  - Similar values in the same row/column at different timestamps
  - Similar values in different columns
  - Similar values across adjacent rows
- Two-pass custom compressions scheme
  - First pass: compress long common strings across a large window
  - Second pass: look for repetitions in small window
- Speed emphasized, but good space reduction (10-to-1)

Refinements: Bloom Filters

- Read operation has to read from disk when desired SSTable isn't in memory
- Reduce number of accesses by specifying a Bloom filter.
  - Allows us ask if an SSTable might contain data for a specified row/column pair.
  - Small amount of memory for Bloom filters drastically reduces the number of disk seeks for read operations
  - Use implies that most lookups for non-existent rows or columns do not need to touch disk

Refinements

- **Caching for read performance**
  - Uses Scan Cache and Block Cache
- **Commit-log implementation**
  - Suppose one log per tablet rather have one log per tablet server
- **Exploiting SSTable immutability**
  - No need to synchronize accesses to file system when reading SSTables
  - Concurrency control over rows efficient
  - Deletes work like garbage collection on removing obsolete SSTables
  - Enables quick tablet split: parent SSTables used by children

System Performance

- Experiments involving random reads (from GFS and main memory) and writes, sequential reads and writes, and scans.
  - Scan: A single RPC fetches a large sequence of values from the 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

Figure 6: Number of 1000-byte values read/written per second. aggregate rate.

Random Reads

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

Figure 6: Number of 1000-byte values read/written per second. aggregate rate.

Random Reads

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

Figure 6: Number of 1000-byte values read/written per second. aggregate rate

by an order of magnitude or more. Each random read involves the transfer of a 64 KB SSTable block over the network from GFS to a tablet server, out of which only a single 1000-byte value is used. The tablet server executes approximately 1200 reads per second, which translates into approximately 75 MB/s of data read from GFS. This bandwidth is enough to saturate the tablet server CPUs because of overheads in our networking stack, SSTable parsing, and Bigtable code, and is also almost enough to saturate the network links used in our system. Most Bigtable applications with this type of an access pattern reduce the block size to a smaller value, typically 8KB.

Sequential Reads

- A read request is for 1000 bytes.

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

Figure 6: Number of 1000-byte values read/written per second. aggregate rate

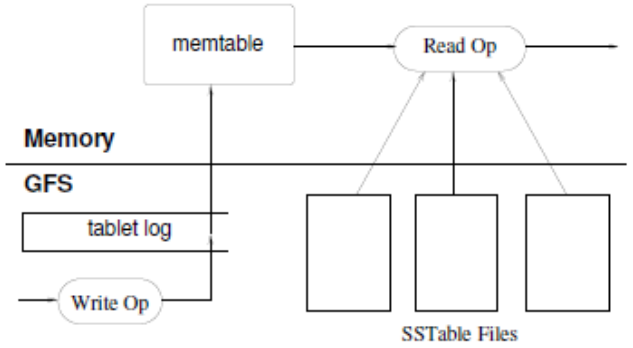
Sequential reads perform better because a tablet server caches the 64 KB SSTable block (from GFS) and uses it to serve the next 64 read requests.

Random Reads from Memory

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

Figure 6: Number of 1000-byte values read/written per second. aggregate rate.

Random reads from memory avoid the overhead of fetching a 64 KB block from GFS.
Data is mapped onto the memory of the tablet server



Writes

- Tablet server appends all incoming writes to a single commit log and uses group commit to stream these writes to GFS efficiently.

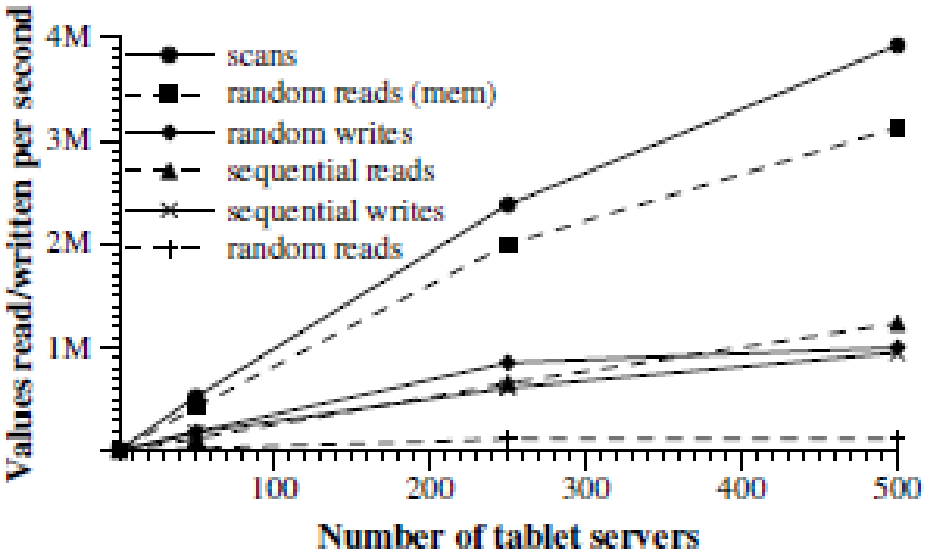
Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	[illegible]	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	[illegible]	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	[illegible]	10526	9524	7843

Figure 6: Number of 1000-byte values read/written per second. aggregate rate.

Scale-up

- As the number of tablet servers is increased by a factor of 500:
  - Performance of random reads from memory increases by a factor of 300.
  - Performance of scans increases by a factor of 260.

- Why?



Scale-up

- As the number of tablet servers is increased by a factor of 500:
  - Performance of random reads from memory increases by a factor of 300.
  - Performance of scans increases by a factor of 260.

- Why?

However, performance does not increase linearly. For most benchmarks, there is a significant drop in per-server throughput when going from 1 to 50 tablet servers. This drop is caused by imbalance in load in multiple server configurations, often due to other processes contending for CPU and network. Our load balancing algorithm attempts to deal with this imbalance, but cannot do a perfect job for two main reasons: rebalancing is throttled to reduce the number of tablet movements (a tablet is unavailable for a short time, typically less than one second, when it is moved), and the load generated by our benchmarks shifts around as the benchmark progresses.

Application 1: Google Analytics

- Enables webmasters to analyze traffic pattern at their web sites. Statistics such as:
  - Number of unique visitors per day and the page views per URL per day,
  - Percentage of users that made a purchase given that they earlier viewed a specific page.
- How?
  - A small JavaScript program that the webmaster embeds in their web pages.
  - Every time the page is visited, the program is executed.
  - Program records the following information about each request:
    - User identifier
    - The page being fetched

Application 1: Google Analytics

- Two of the Bigtables
  - Raw click table (~ 200 TB)
    - A row for each end-user session.
    - Row name include website's name and the time at which the session was created.
    - Clustering of sessions that visit the same web site. And a sorted chronological order.
    - Compression factor of 6-7.
  - Summary table (~ 20 TB)
    - Stores predefined summaries for each web site.
    - Generated from the raw click table by periodically scheduled MapReduce jobs.
    - Each MapReduce job extracts recent session data from the raw click table.
    - Row name includes website's name and the column family is the aggregate summaries.
    - Compression factor is 2-3.

Application 2: Google Earth & Maps

- Functionality: Pan, view, and annotate satellite imagery at different resolution levels.
- One Bigtable stores raw imagery (~ 70 TB):
  - Row name is a geographic segments. Names are chosen to ensure adjacent geographic segments are clustered together.
  - Column family maintains sources of data for each segment.

- There are different sets of tables for serving client data, e.g., index table.

Application 3: Personalized Search

- Records user queries and clicks across Google properties.
- Users browse their search histories and request for personalized search results based on their historical usage patterns.
- One Bigtable:
  - Row name is userid
  - A column family is reserved for each action type, e.g., web queries, clicks.
  - User profiles are generated using MapReduce.
    - These profiles personalize live search results.
  - Replicated geographically to reduce latency and increase availability.

Lessons

- Interesting Point – only implement some of the requirements, since the last is probably not needed.
- Many types of failure in real life system can occur.
- Big Systems need proper system level monitoring.
- Value simple designs.

Bottleneck

- Bigtable uses Chubby to:
  - Ensure there is at most one active master at a time,
  - Store the bootstrap location of Bigtable data (Root tablet),
  - Discover tablet servers and finalize tablet server deaths,
  - Store Bigtable schema information (column family information),
  - Store access control list.
  - If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable

Conclusion

- Designed to store enormous amounts of data
- Easily scalable to accommodate petabytes of data across thousands of nodes.
- Substantial amount of flexibility and advantages due to having a own data model at Google

Questions ??