Column-Stores – Google Bigtable

Iztok Savnik, FAMNIT

November, 2022.

Literature

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, *Bigtable: A Distributed Storage System for Structured Data*, OSDI, 2006.

Romain Jacotin, *Lecture: The Google Bigtable*, Google Research, 2014.

Outline

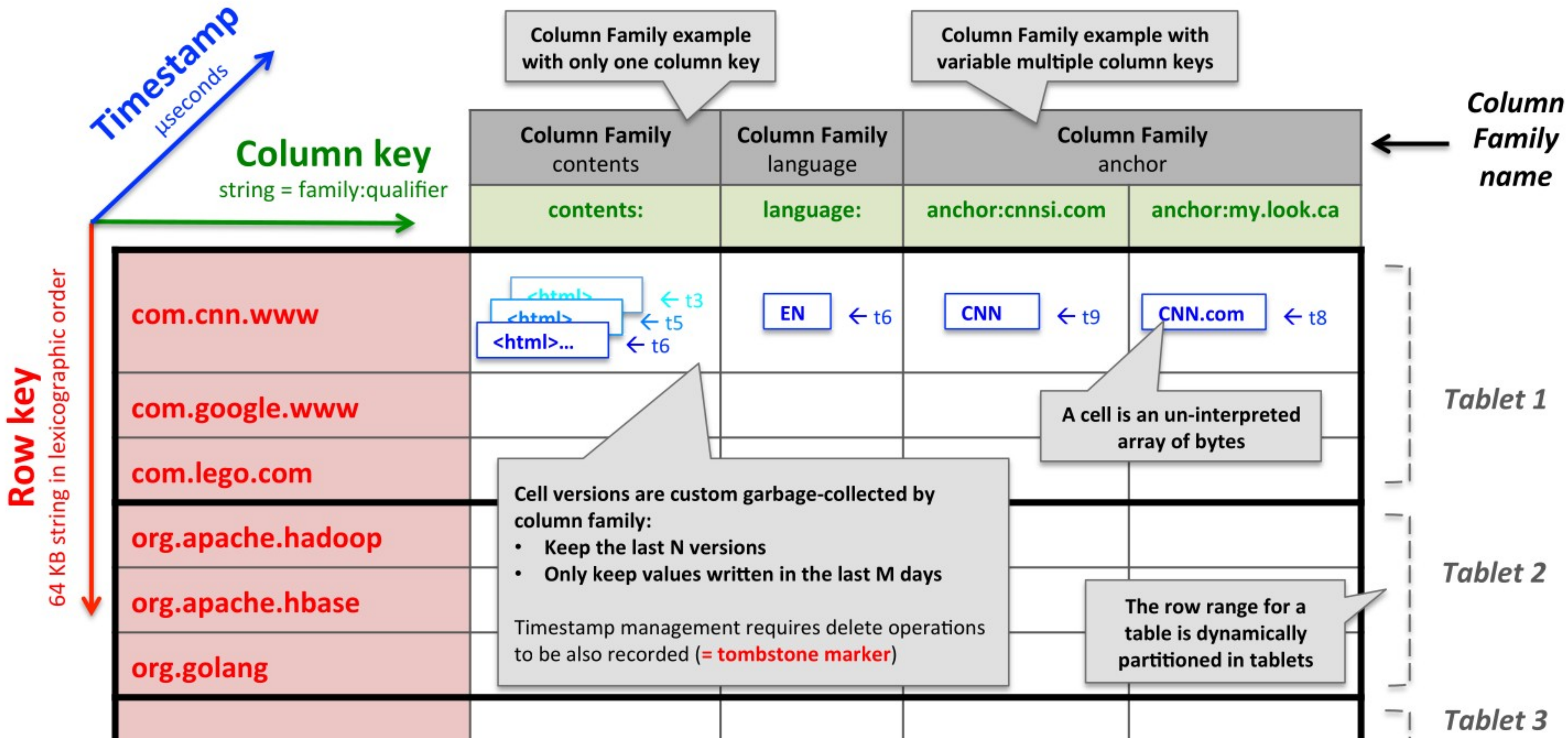
- Introduction
- Data model
- API
- Building blocks
- Implementation
- Refinements
- Performance evaluation
- Experience
- Conclusions

Introduction

Abstract:

- Bigtable is a distributed storage system for managing structured data
  - Designed to scale to a very large size: petabytes of data across thousands of commodity servers.
  - It is not a relational database, it is a sparse, distributed, persistent multi-dimensional sorted map (key/value store).
- Many projects at Google store data in Bigtable
  - Web indexing, Google earth and Google finance, …
  - Different data sizes: URL, web pages, satellite imagery, …
  - Different latency requirements: backend bulk processing to real-time data serving.
- Bigtable is a flexible, high performance solution

Data model



API

- Tablet server
  - AddServer(tabletServer) / RemoveServer(tabletServer)!
- Table
  - CreateTable(table) / DeleteTable(table)!
- Column family
  - CreateColumnFamily(columnFamily) / DeleteColumnFamily(columnFamily)!
- Table access control rights and metadata
  - SetTableFlag(table, flags) / . . .!
- Colum family access control rights and metadata
  - SetColumnFamilyFlag(table, colfamily, flags) / . . .!

API (2)

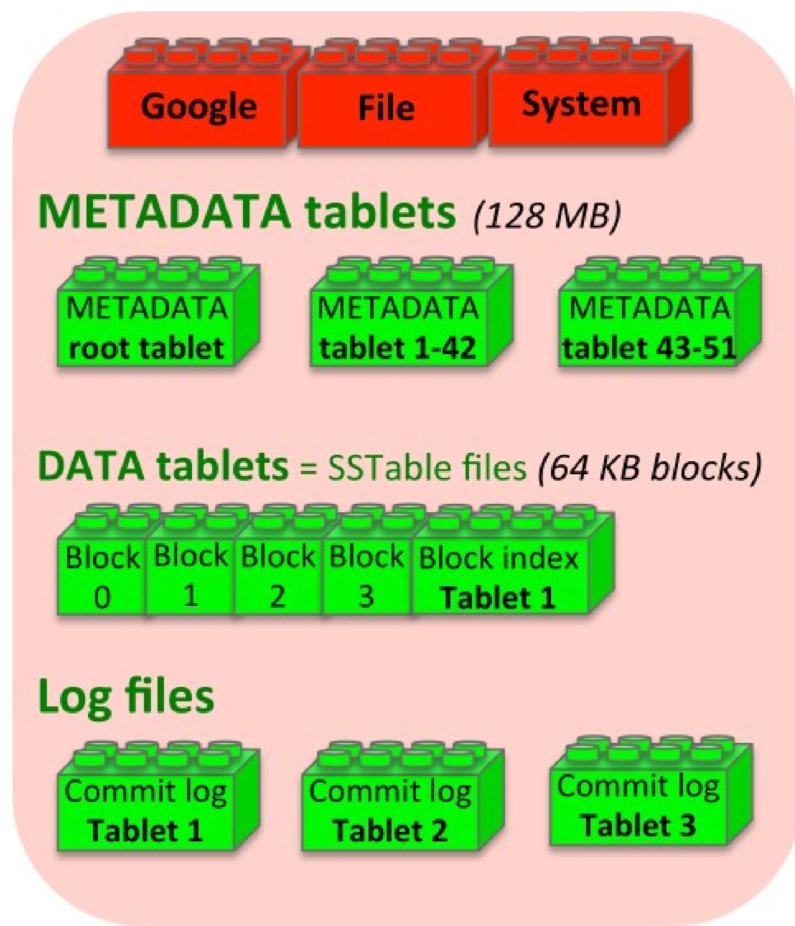
- Cell value
  - Put(rowkey, columnkey, value)
  - Get(rowkey, columnkey)
  - Delete(rowkey, columnkey)!
- Look up value from individual row
  - Has(rowkey, columnfamily), ...
- Look up values from table (=MapReduce like RPC)
  - Scan(rowFilter, columnFilter, timestampFilter)!
    - Can iterate over multiple column families
    - Can limit rows/colums/timestamps
- Single-row transactions (atomic read-modify-write sequence)
- No support for general transactions across row keys

API (3)

- Cells can be used as integer counters
  - Increment(rowkey, columnkey, increment)
- Execution of read-only client-supplied scripts in the address spaces of the servers: Sawzall
  - http://research.google.com/archive/sawzall.html
- Bigtable can be used with MapReduce (for input and/or output)

Building blocks

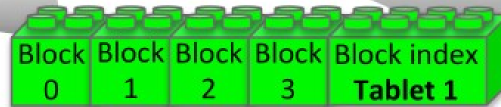
- Google File System (GFS)
  - Bigtable uses the fault-tolerant and scalable distributed GFS file system to store log and data files
  - metadata = METADATA tablets (store tablets location)
  - data = SSTables collection by tablet
  - log = Tablet logs
- Google SSTable file format (Sorted String Table)
  - Used to store table data in GFS
    - Persistent, ordered immutable map from keys to values



Building blocks

rowkey 1	col key	timestamp	value
	col key	timestamp	value
rowkey 2	col key	timestamp	value
	col key	timestamp	value
rowkey 3	col key	timestamp	value

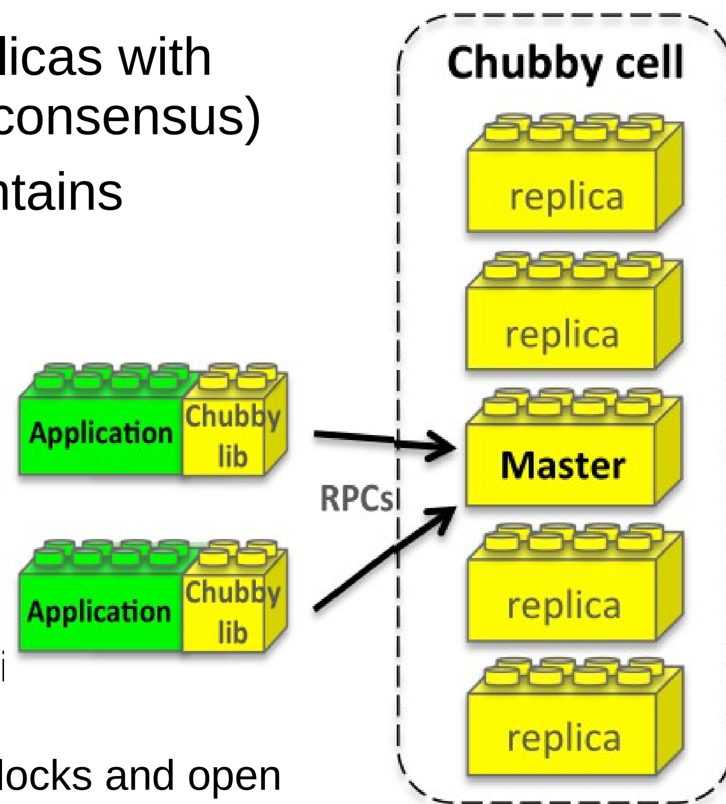
Block 0		Block 1		Block 2		Block 3	
File offset		File offset		File offset		File offset	
First row	rowkey 1	First row	rowkey 4	First row	rowkey 10	First row	rowkey 15
Last row	rowkey 3	Last row	rowkey 9	Last row	rowkey 14	Last row	rowkey 18



- Google SSTable file format (Sorted String Table)
  - Contains a sequence of 64 KB Blocks (size configurable)
    - Optionally, Blocks can be completely mapped into memory (= lookups and scans without touching disk)
- Block index stored at the end of the file
  - Used to locate blocks
  - Index loaded in memory when the SSTable is opened
  - Lookup with a single seek
    - Find the appropriate block by performing a binary search in the in-memory index
    - Reading the appropriate block from disk

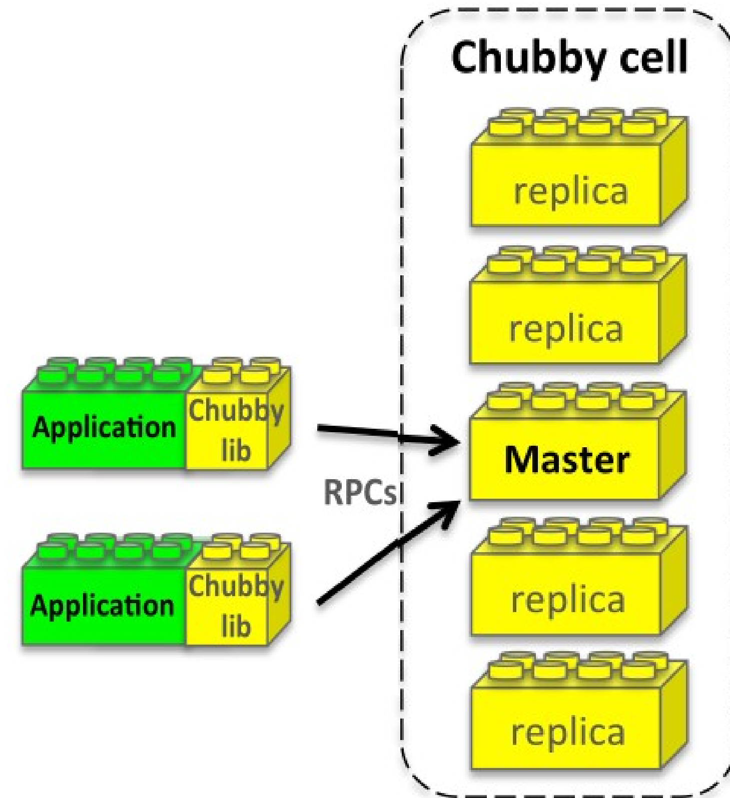
Building blocks

- Google Chubby
  - Chubby service consists of 5 active replicas with one master to serve requests (PAXOS consensus)
  - Chubby provides a namespace that contains directories and small files (<256 KB)
    - Each directory or file can be used as a lock
    - Reads and writes to a file are atomic
    - Chubby client library provides consistent caching of Chubby files
    - Each Chubby client maintains a session with a Chubby service
    - Client's session expires if is unable to renew i lease within the lease expiration time.
    - When a client's session expires, it loses any locks and open handles.
    - Chubby clients can also register callbacks on Chubby files and directories for notification of changes or session expiration.

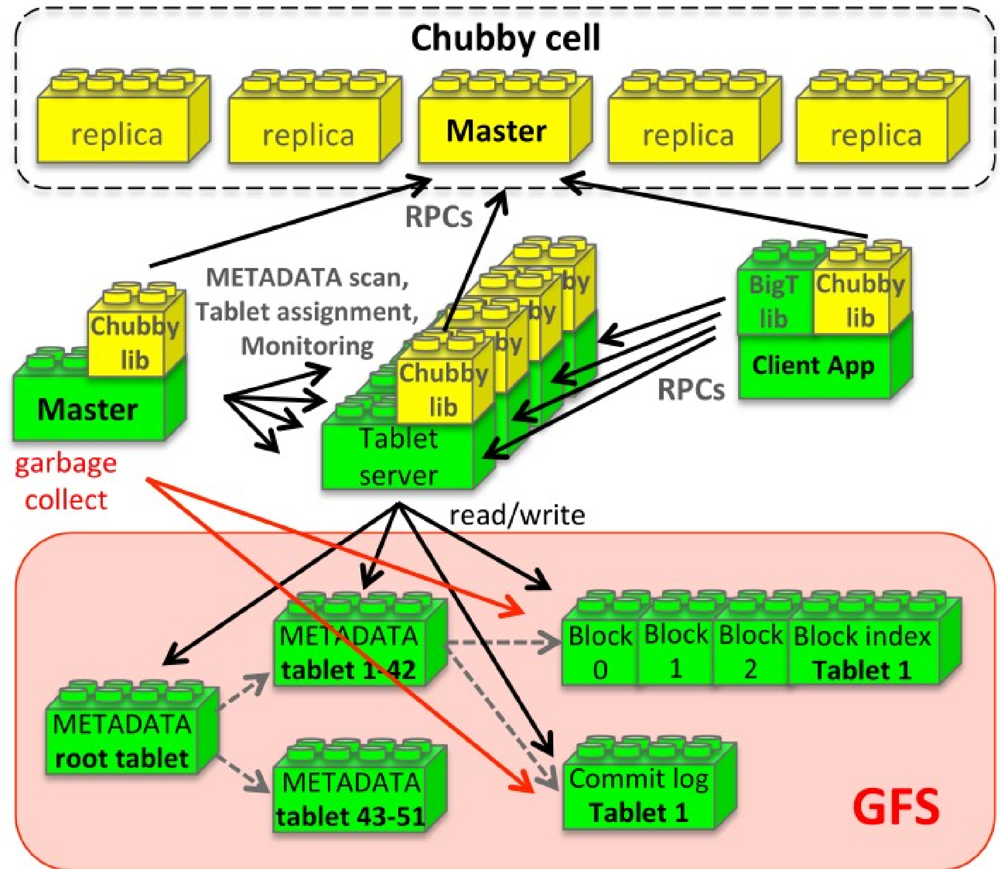


Building blocks

- Google Chubby
  - Bigtable uses Chubby for a variety of tasks
    - To ensure there is at most one active master at any time
    - To store the bootstrap location of Bigtable data (Root tablet)
    - To discover tablet servers and finalize tablet server deaths
    - To store Bigtable schema information (column family information for each table)
    - To store access control lists (ACL)
  - Chubby unavailable = Bigtable unavailable
    - 14 Bigtable clusters spanning 11 Chubby instances: average % server hours unavailability = 0,0047%



Implementation



- Major components
  - One master server
  - Many tablet servers
  - A library linked into every client
- Master
  - Assigning tablets to tablet servers
  - Detecting addition and expiration of tablet servers
  - Balancing tablet server load
  - Garbage collecting of files in GFS
  - Handling schema changes (table creation, column family creation/deletion
- Tablet server
  - manages a set of tablets
  - Handles read and write request to the tablets
  - Splits tablets that grow too large (100-200 MB)
- Client
  - Do not rely on master for tablet location information
  - Communicates directly with tablet servers for reads and writes

Implementation

Tablet Location

"DNS like" Tablet server location ! ;-)

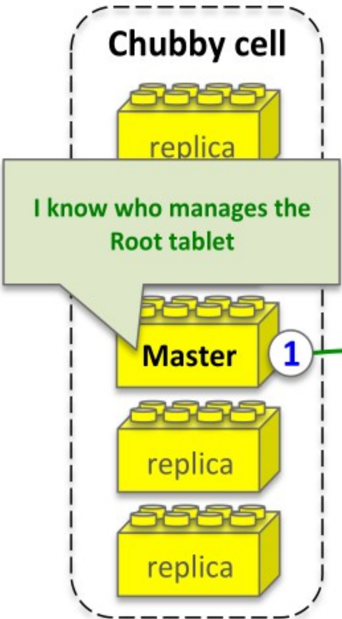
- 3 level hierarchy to store tablet location information (B+ tree)

Who manages the Tablet for table "Users" and row "`Romain Jacotin`" ??? **?**

- Empty cache = 3 network round-trips:
  1. **Request to Chubby** for Root tablet location
  2. **Request to Root** for METADATA tablet location
  3. **Request to MEDATA tablet** for Tablet location
- Stale cache: worst case = 6 network round-trips
  stale cache entries only discovered upon misses

For the table named "`Users`" and given row key "`Romain Jacotin`" i know who manages Tablet 1 with a row range that could possibly contain this row key
(and i know what are the GFS files used for Tablet 1)

I know who manages each METADATA tablets based on "`table name +end row`" entry



METADATA Tablet server 2 → METADATA Tablet server 3 → DATA Tablet server

Client App

Read write

Read write

Read write

Block 0 | Block 1 | Block 2 | Block index Tablet 1

Commit log Tablet 1

METADATA root tablet

METADATA tablet 1-42

METADATA tablet 43-51

Block 0 | Block 1 | Bloc Tab

Commit log Tablet 42

B+ tree

GFS

In fact METADATA tablets are "normal" Tablets (SST+Log) with specifics access control rights and Column Family schema
→ Root tablet is never split !

Implementation

- Tablet Assignment
  - Each tablet is assigned to one tablet server at a time
  - Master keeps tracks of
    - the set of live tablet servers (tracking via Chubby)
    - the current assignment of tablets to tablet servers
    - the currently unassigned tablets
  - When a tablet is unassigned, the master assigns the tablet to an available tablet server by sending a tablet load request to that tablet server
- Tablet Server discovery
  - When a tablet server starts, it creates, and acquires an exclusive lock on uniquely-named file in a specific Chubby directory (“servers” directory)
  - Master monitors this directory to discover tablet servers
  - A tablet server stops serving its tablets if it loses its exclusive Chubby lock
  - If the Chubby file no longer exists, then the tablet server will never be able to serve again, so it kills itself

Implementation

- Tablet Server monitoring
  - Master is responsible for detecting when a tablet server is no longer serving its tablets, and for reassigning those tablets
  - Master periodically asks each tablet server for the status of its lock to detect when a tablet server is no longer serving its tablets
  - If a tablet server reports that it has lost its lock, or if the master was unable to reach a server during its last attempts, the master attempts to acquire the lock for the Chubby file
    - If the master is able to acquire the lock then Chubby is live and the tablet server is dead or isolated, the master deletes its server file to ensure that the tablet server can never serve again
    - Then master can move all the tablets that were previously assigned to this tablet server into the set of unassigned tablets

Implementation

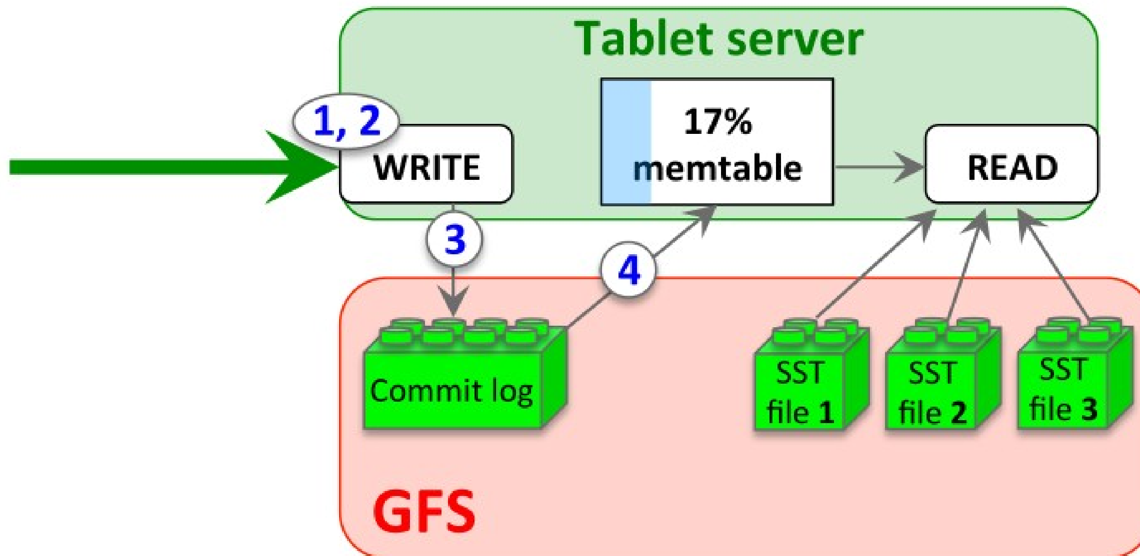
- Master isolated
  - To ensure that a Bigtable cluster is not vulnerable to networking issues between the master and Chubby, the master kills itself if its Chubby session expires (master failures do not change the assignment of tablets to tablet servers)
- Master startup
  - When a master is started by the cluster management system, it needs to discover the current tablet assignments before it can change them:
    1) Master grabs a unique master lock in Chubby to prevent concurrent master instantiations
    2) Master scans the servers directory in Chubby to find the live tablet servers
    3) Master communicate with every live tablet servers to discover what tablets are already assigned to each server
    4) Master adds the root tablet to the set of unassigned tablets if an assignment for the root tablet is not discovered in step 3,
    5) Master scans the METADATA tablets to learn the set of tablets (and detect unassigned tablets).

Implementation

- Tablet : merging / splitting
  - The set of existing tablets only change when:
    - A table is created or deleted
    - Two existing tablets are merged to form one larger tablet
    - Existing tablet is split into two smaller
  - Master initiates Tablets merging
  - Tablet server initiate tablet spliting
    - Commit the split by recording information for new tablet in the METADATA table
    - After committed, the tablet server notifies the master

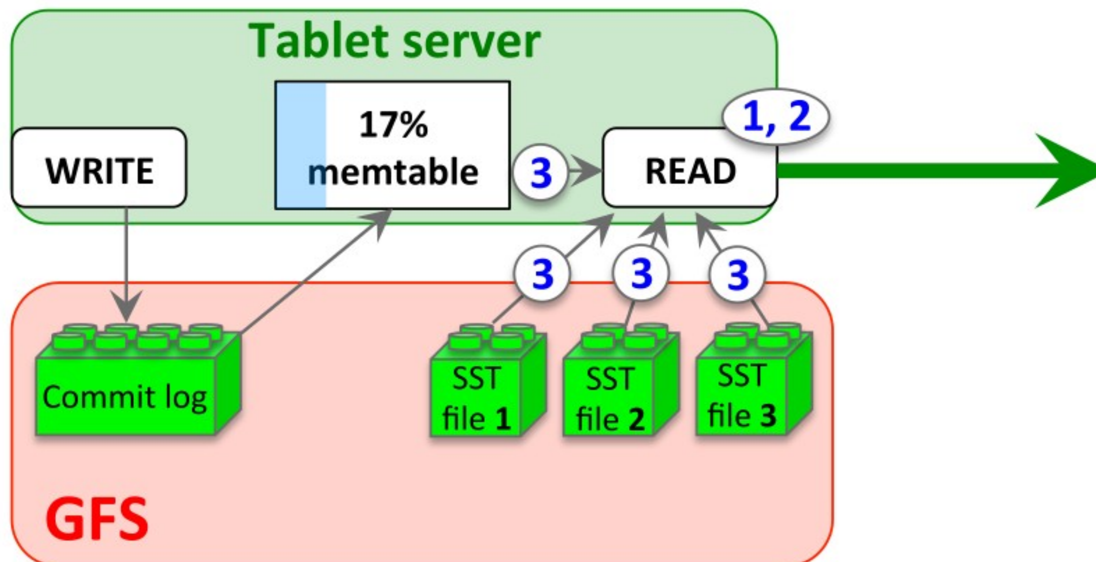
Implementation

- Tablet Serving
  - Write operation
    - Server checks that the request is well-formed
    - Server checks that the sender is authorized to write (list of permitted writers in a Chubby file)
    - A valid mutation is written to the commit log that stores redo records (group commit to improve throughput)
    - After the mutation has been committed, its contents are inserted into the memtable (= in memory sorted buffer)



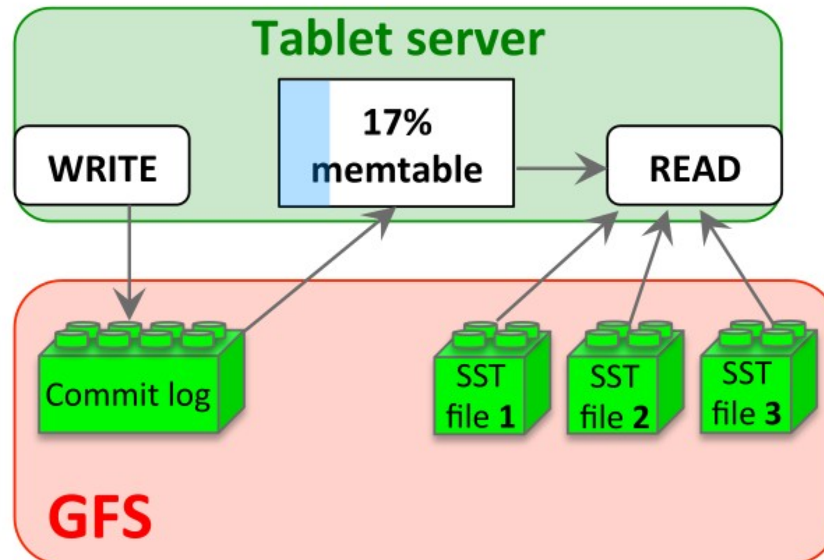
Implementation

- Tablet Serving
  - Read operation
    - Server checks that the request it is well-formed
    - Server checks that the sender is authorized to read (list of permitted readers from a Chubby file)
    - Valid read operation is executed on a merged view of the sequence of SSTables and the memtable



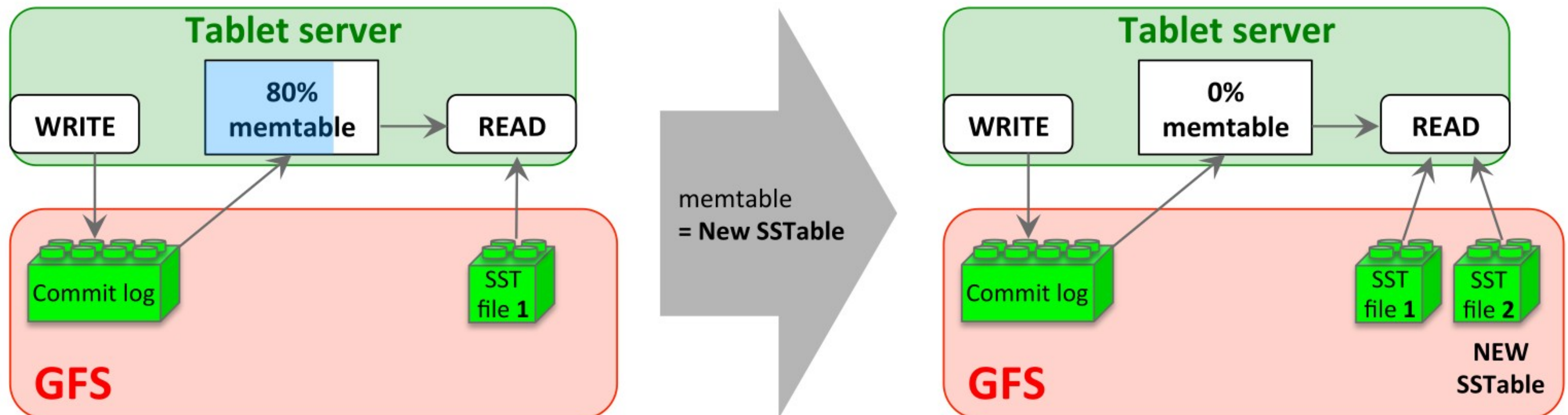
Implementation

- Tablet Serving
  - Tablet Recovery
    1) Tablet server reads its metadata from the METADATA table (lists of SSTables that comprise a tablet and a set of a redo points, which are pointers into any commit logs that may contain data for the tablet)
    2) The tablet server reads the indices of the SSTables into memory and reconstructs the memtable by applying all of the updates that have a committed since the redo points



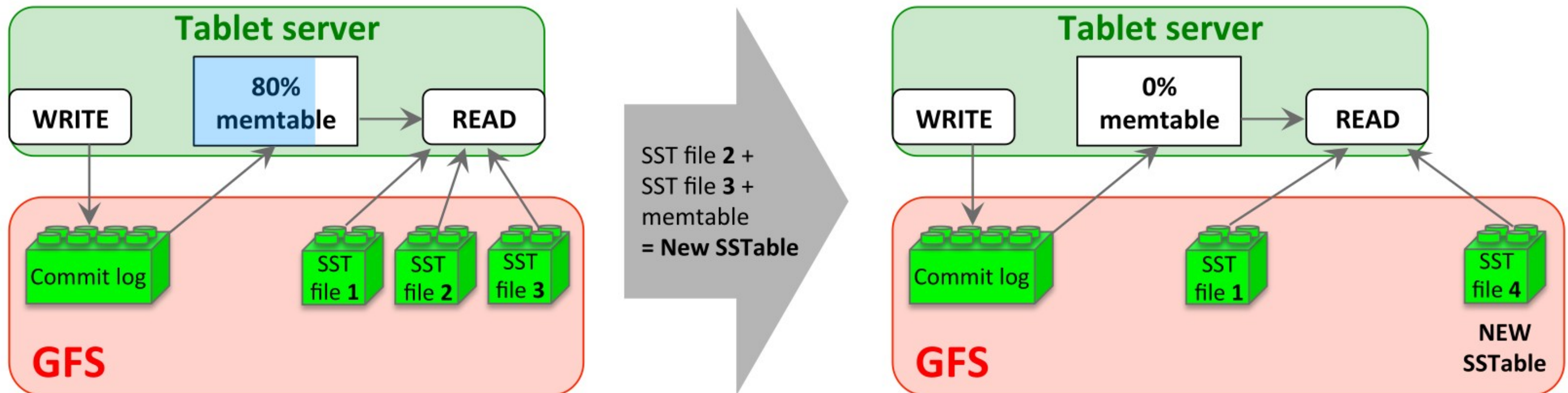
Implementation

- Compactions
  - Minor compaction
    - When memtable size reaches a threshold, memtable is frozen, a new memtable is created, and the frozen memtable is converted to a new SSTable and wri#en to GFS
    - Two goals: shrinks the memory usage of the tablet server, reduces the amount of data that has to be read from the commit log during a recovery



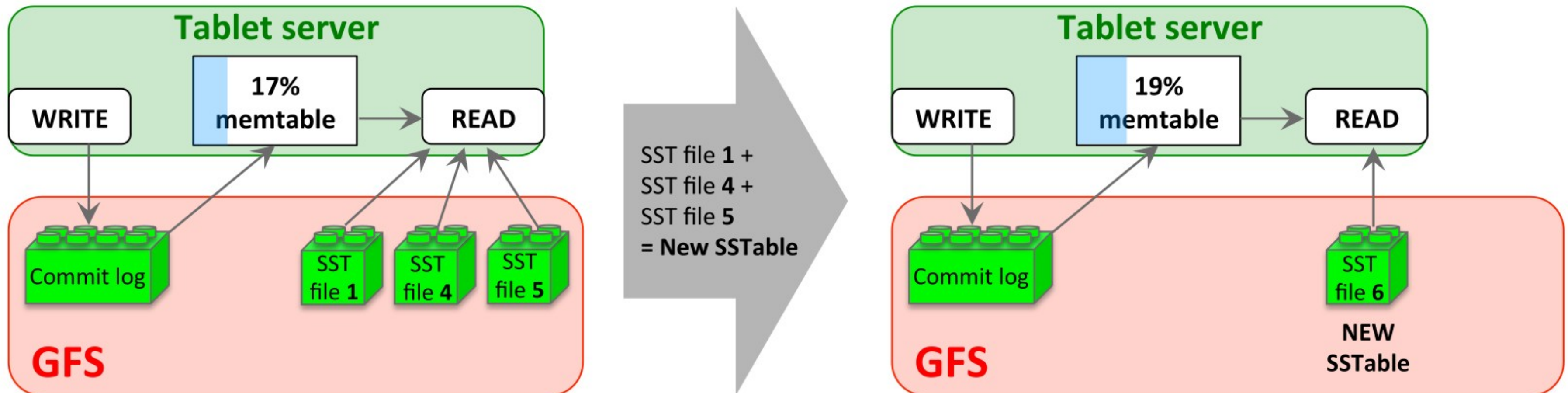
Implementation

- Compactions
  - Merging compaction
    - Problem: every minor compaction creates a new SSTable (=> arbitrary number of SSTables !)
    - Solution: periodic merging of a few SSTables and the memtable



Implementation

- Compactions
  - Major compaction
    - It is a merging compaction that rewrites all SSTables into exactly one SSTable that contains no deletion information or deleted data
    - Bigtable cycles throught all of it tablets and regularly applies major compaction to them (=reclaim ressources used by deleted data in a timely fashion)

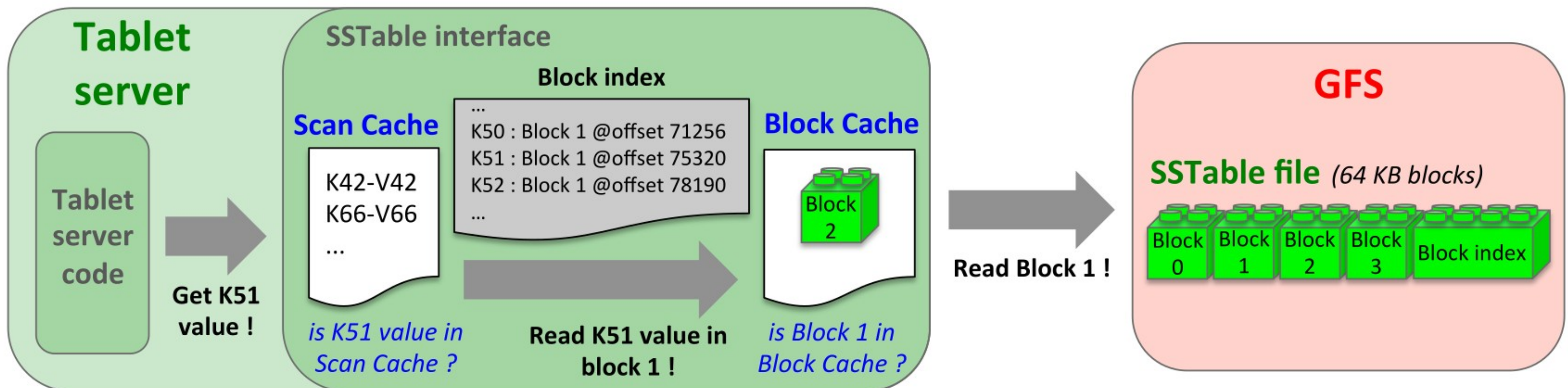


Refinements

- Locality groups
  - Clients can group multiple column families together into a locality group => more efficient reads!
    - A separate SSTable is generated for each locality group in each tablet
- In-memory
  - A locality group can be declared to be in-memory => no need for disk access!
- Compression
  - Clients can control wether or not the SSTables for a locality group are compressed
    - Compression format is applied to each SSTable block (64KB)
    - Two-pass Compression: fist pass uses Bentley and McIlroy's scheme, second pass uses a fast compression that looks for repetions in 16 KB window of the data (encode rate = 100-200 MB/s, decode rate = 400-1000 MB/s)

Refinements

- Caching for read performance
  - Tablet servers use two levels of caching
    - The Scan Cache : high level cache for key-value pairs returned by the SSTable interface to the tablet server code
    - The Block Cache : low level cache for SSTables blocks read from GFS

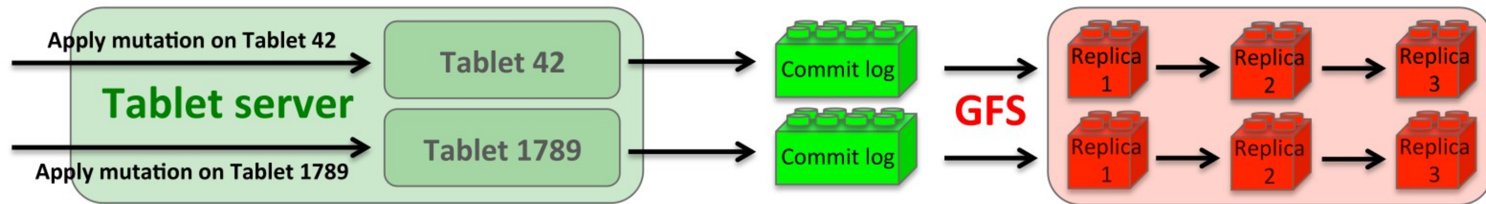


Refinements

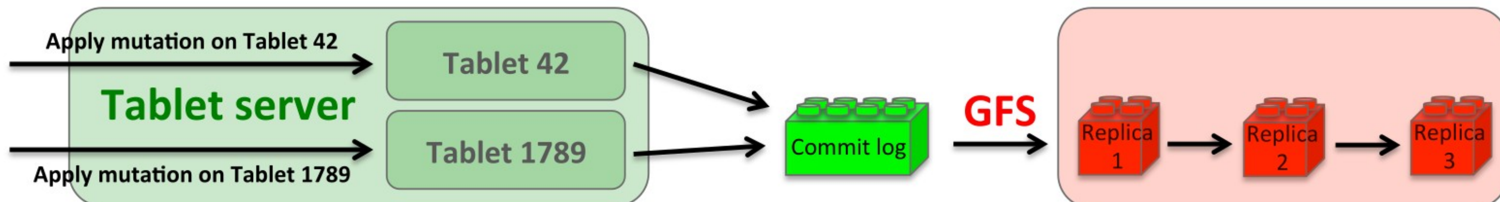
- Bloom filters
  - *Problem*: read operation has to read from all SSTables that make up the state of a tablet
    - Lot of disk access
  - *Solution*: use Bloom filers for each SSTable in a particular locality group
    - Bloom filter uses a small amount of memory and permit to know if a SSTable doesn’t contain a specified row/column pair
    - Most lookups for non existent rows or columns do not need to touch disk.

Refinements

- Commit-log implementation
  - Problem: If we kept the commit log for each tablet in a separate log file, a very large number of files would be written concurrently
    - in GFS = large number of disk seeks to write to different physical log files …



  - Solution: Append mutations to a single commit log per tablet server, co-mingling mutations for different tablets in the same physical log file
    - One log provides significant performance benefits during normal operation
    - Using one log complicates recovery … ! Sorting commit-log in a distributed way before reassigned the tablets



Refinements

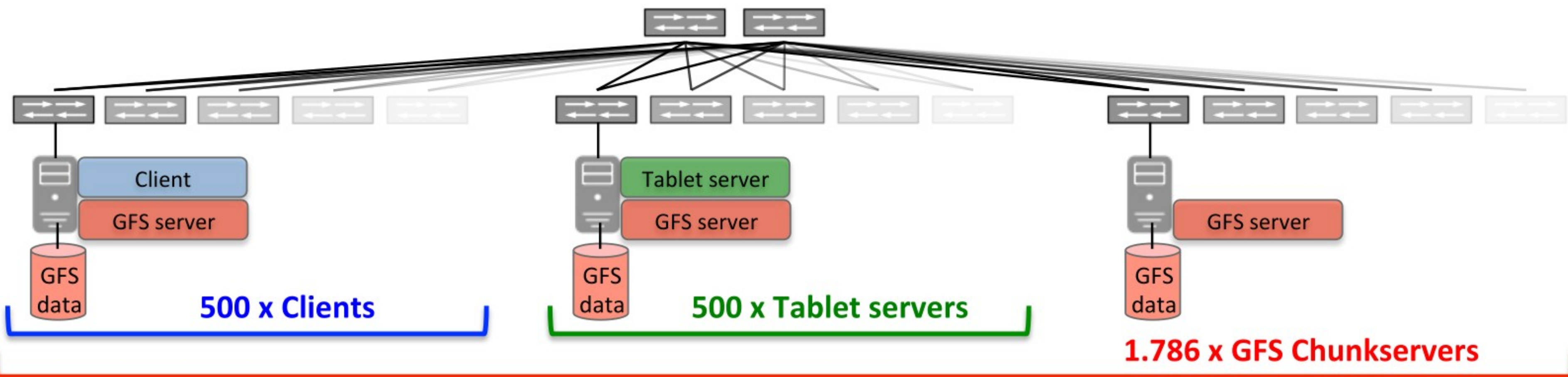
- Speeding up tablet recovery
  - If Master moves a tablet from one tablet server to another, the source tablet server first does minor compaction on that tablet
  - After this compaction, the source tablet server stops serving the tablet
  - Before unloads the tablet, the source tablet server does another (very fast) minor compaction to eliminate any remaining un-compacted state in the tablet server’s log that arrived while the first minor compacon was being performed
  - Tablet can now be loaded on another tablet server

Refinements

- Exploiting immutability
  - Various parts of the Bigtable system have been simplified by the fact that all of the SSTables generated are immutable
    - No need for synchronization when reading from SSTables => easy concurrency control over rows
    - The only mutable data structures accessed by both reads and writes is the “memtable” => each memtable row use copy-on-write and allow reads and writes to proceed in parallel
  - Garbage collection on obsolete SSTables
    - Master removes obsolete SSTables (in the METADATA table) as a mark-and-sweep garbage collection over the sate of SSTables
  - Immutability of SSTables permit to split tablets quickly
    - Child tablets share the SSTables of the parent tablet

Performance evaluation (2006)

- Bigtable cluster
  - 500 tablet servers
    - Configured to use 1 GB RAM
    - Dual-core Opteron 2 GHz, Gigabit Ethernet NIC
    - Write to a GFS cell (1786 machines with 2 x 400 GB IDE)
  - 500 clients
  - Network round-trip time between any machine < 1 millisecond



Performance evaluation (2006)

- Sequential writes
  - Used R row keys partitioned and assigned to N clients
  - Single unique random string row key (uncompressible)
- Random writes
  - Similar to Sequential writes except row key hashed modulo R
- Sequential reads
  - Used R row keys partitioned and assigned to N clients
- Random reads
  - Similar to Sequential reads except row key hashed modulo R
- Random reads (memory)
  - Similar to Random reads benchmark except locality group that contains the data is marked as in-memory
- Scans
  - Similar to Random reads but uses support provided by Bigtable API for scanning over all values in a row range (reduces RPC)

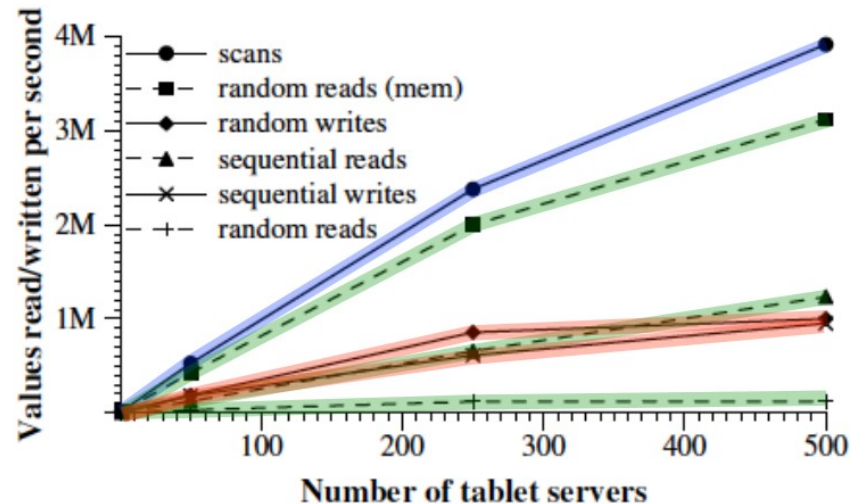
Performance evaluation (2006)

- Reads
- Writes
- Scans

Number of 1000-byte values read/write per second
(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/write per second
(aggregate rate)



Experience (2006)

- Real applications
  - Google indexing, Gmail, Google Analytics, Google Maps, Google Books, Google Earth, Personalized search, … also Youtube
  - It is in use currently! (above apps)

Distribution of number of tablet servers in Bigtable clusters

# of tablet servers	# of clusters
0 .. 19	259
20 .. 49	47
50 .. 99	20
100 .. 499	50
> 500	12

Tables in production *(table size = before compression)*

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

Experience (2006)

- Lessons
  - Large distributed systems are vulnerable to many types of failures
    - memory and network corruption
    - large clock skew
    - hung machines
    - extended and asymmetric partitions
    - bugs,
    - overflow
    - planned and unplanned hardware maintenance, …
  - Importance of proper system-level monitoring
    - lock contention detection on tablet data structures
    - slow writes to GFS while committing Bigtable mutations
    - stuck access to METADATA when METADATA tablets unavailable
    - track down all clusters via Chubby
  - The value is in simple design

Experience (2022)

- When to use
  - Low latency access to big data
  - Good for > 1TB of data, data elems < 10MB
  - Map-reduce, Spark applications
  - Real-time analytics
  - Not for interactive SQL queries
- Current usage
  - Jan 2022: BT manages over 10 Exabytes of data, 5 billion requests per second