BigTable: A Distributed Storage System for Structured Data

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

ECE1724

Authors: Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

Many slides adapted from Ion Stoica, Berkeley

Why Build BigTable?

- Need highly available, scalable structured data storage
  - Web crawler: url, content, anchors, page rank
  - Per-user data: account info, preferences, recent queries
  - Geography: roads, satellite image data, user annotations

- Google's workloads
  - Petabytes of data across thousands of servers
  - Billions of URLs with many versions per page (~20K/version)
  - Hundreds of millions of users
  - Thousands of queries per second
  - 100TB+ satellite image data

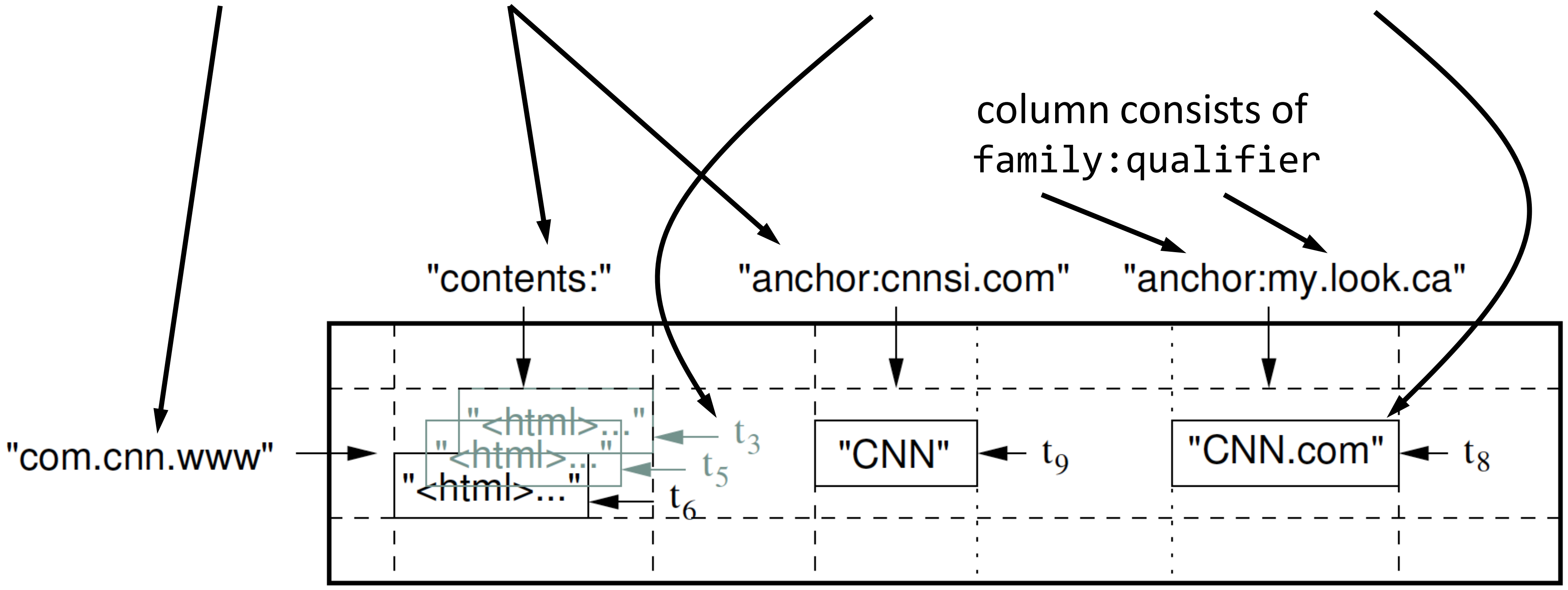
Why Not Use Commercial DB?

- Scale is too large for most commercial databases
- Even if it weren’t, cost would be very high
    - Building internally means system can be applied across many applications with low incremental cost
- Low-level storage optimizations improve performance
    - Much harder to do when running on top of a database layer

What is BigTable?

- A sparse, distributed, multi-level sorted map:

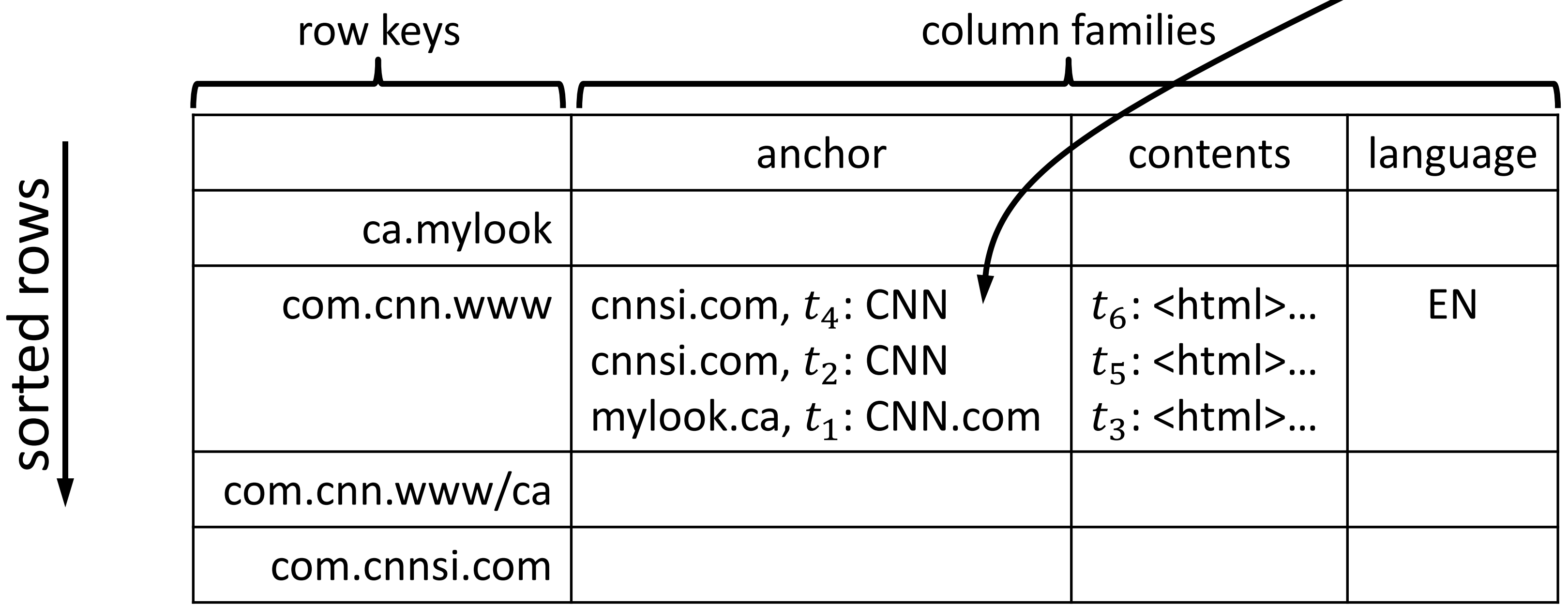
```
(row:string, column:string, time:int64) → cell content
```



Column Families

- Column family is a group of column keys
  - Column format is `family:qualifier`
    - Family specified on creation, like traditional column in DBs
    - New qualifiers can be created anytime
  - Each column family can be compressed and stored separately

You can think of each (row, family) as a KV store:
`(qualifier, time) -> value`



	anchor	contents	language
ca.mylook			
com.cnn.www	cnnsi.com, t_4: CNN	t_6: <html>…	EN
	cnnsi.com, t_2: CNN	t_5: <html>…	
	mylook.ca, t_1: CNN.com	t_3: <html>…	
com.cnn.www/ca			
com.cnnsi.com			

Timestamps

- Each cell can contain multiple versions of same data
  - Version indexed by a 64-bit timestamp
  - Real time or assigned by client
- Per-column-family settings for garbage collection
  - Keep only latest *n* versions
  - Or keep only versions written since time *t*
- Retrieve most recent version if no version specified
  - If specified, return version where timestamp ≤ requested time

BigTable API

- Tables and column families
  - create, delete, update, control rights
- Rows
  - create, delete
  - atomic per-row read and write, read-modify-write
  - Iterate over row ranges
- Multi-row access
  - No transactions across rows
  - Support batching writes across rows
- Client-provided server-side scripts for transformation, filtering, summarization, etc.

BigTable Goals

- Use a cluster of machines to provide a scalable, shared-nothing database
- Persistent and fault-tolerant
- Scalable
  - Support thousands of servers
  - Terabytes of in-memory data, petabyte of disk-based data
  - Millions of reads/writes per second, efficient scans
- Self-managing
  - Servers can be added/removed dynamically
  - Servers adjust to load imbalance

Key Design Ideas

- Goal: use a cluster of machines to provide a scalable, shared-nothing database
- Single master server
  - Performs database schema operation
    - Create table, column families, etc.
  - Uses a coordination server (Chubby lock server)
    - For leader election, tablet servers, storing schema metadata, etc.
  - Dynamically partitions tables across data servers
    - Migrates table partitions (tablets) for load balancing
  - Avoids performing any data operations
- Data (Tablet) servers …

Key Design Ideas

- Goal: use a cluster of machines to provide a scalable, shared-nothing database
- Master server …
- Data (Tablet) servers
  - Serve data, i.e., table rows
  - Row format is flexible (unbounded number of columns)
  - Provide low latency access by using write-optimized data store
  - Use GFS for storage and replication
  - Co-locate with GFS servers for locality

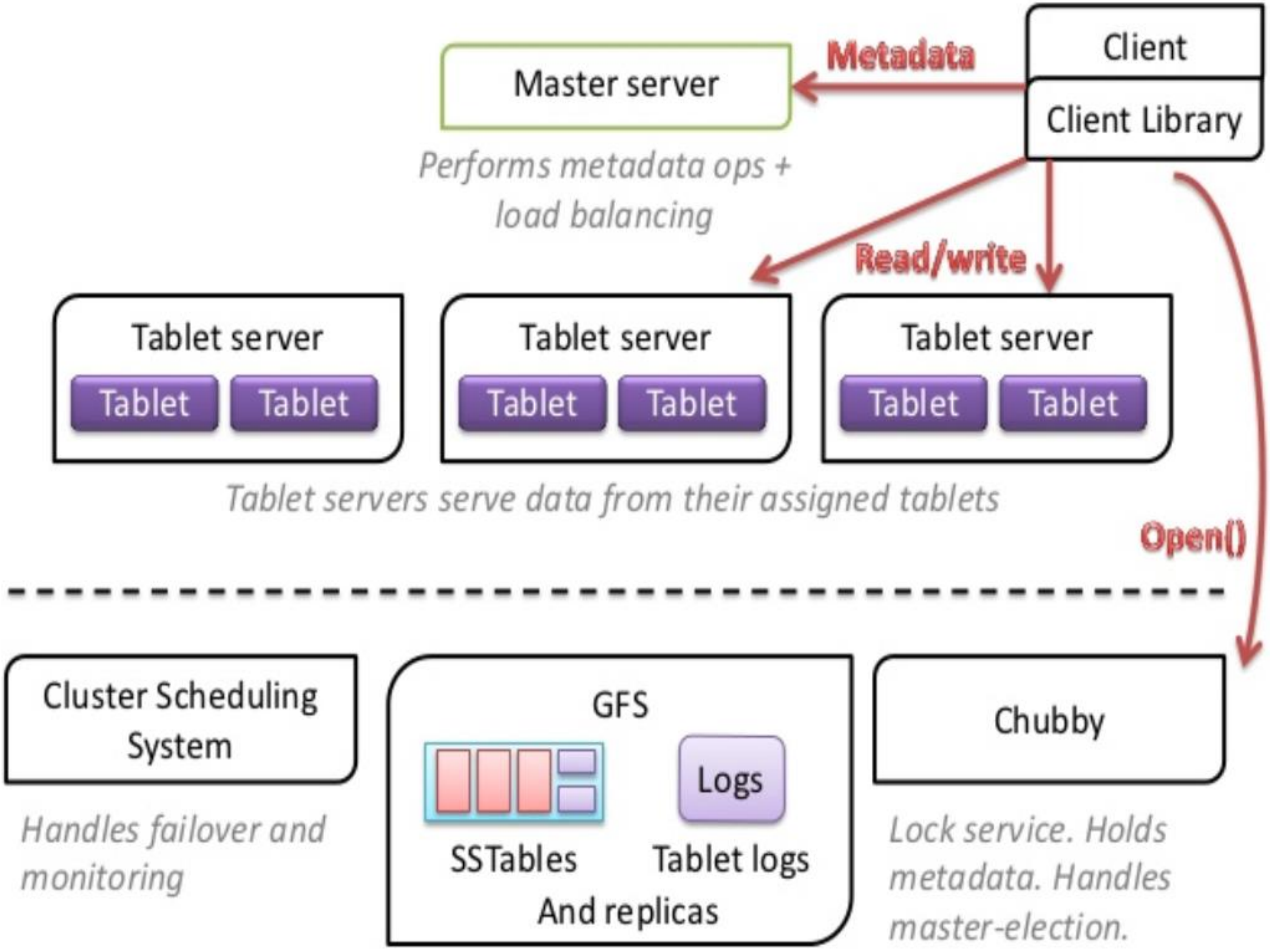
Partitioning Tables: Tablets

- Master partitions tables dynamically by ranges of contiguous rows into tablets, typically 100-200MB size

		anchor	contents	language
Tablet 1	ca.mylook			
Tablet 2	com.cnn.www	cnnsi.com, t_4: CNN cnnsi.com, t_2: CNN mylook.ca, t_1: CNN.com	t_6: <html>… t_5: <html>… t_3: <html>…	EN
	com.cnn.www/ca			
Tablet 3	com.cnnsi.com			

- A tablet is a unit of distribution and load balancing
  - Each tablet served by a single tablet server
- Users select keys to control placement of related rows
  - Nearby rows will usually be served by same server

Big Table Architecture


Master server
Metadata
Client
Client Library
Performs metadata ops + load balancing
Read/write
Tablet server
Tablet
Tablet
Tablet server
Tablet
Tablet
Tablet server
Tablet
Tablet
Tablet servers serve data from their assigned tablets
Open()
Cluster Scheduling System
GFS
SSTables
Logs
Tablet logs
And replicas
Chubby
Handles failover and monitoring
Lock service. Holds metadata. Handles master-election.

BigTable Storage

- Use Google file system (GFS) to store log and data files
  - SSTable file format (discussed later)

- Use Chubby distributed lock service for coordination
  - Store bootstrap location of Bigtable data
  - Store schema metadata (e.g., column families for each table)
  - Store access control lists
  - Helps ensure at most one active master exists
  - Helps keep track of live tablet servers

BigTable Implementation

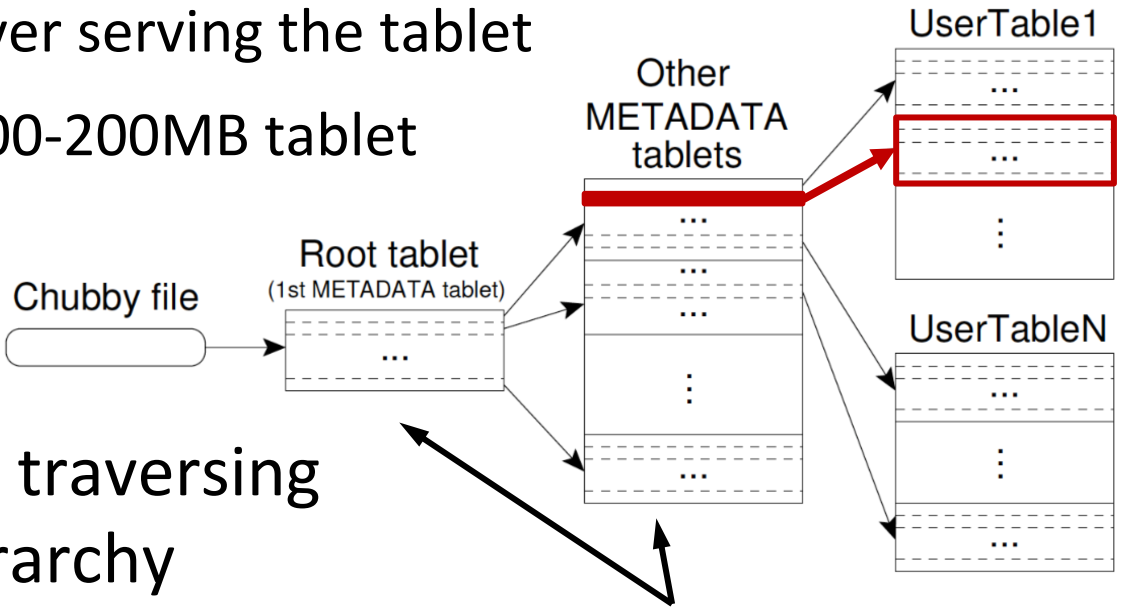
- Library linked with every client
- Master
  - Assigns tablets to tablet servers
  - Handles adding, deleting and merging of tablets
  - Handles addition and removal of tablet servers in the system
- Tablet server
  - Each tablet server typically serves 10-1000 tablets
  - Tablet servers handle read and writes and splitting of tablets
  - Clients access data from tablet servers directly

Locating Tablets

- Client needs to find tablet whose row range covers the target rows in a query
- Since tablets may be loaded on any tablet server and may be migrated, how do clients find tablets?
- One option would be to store tablet row-range to tablet server mapping at the BigTable master
  - Central server would become bottleneck in large system
- Instead, BigTable uses a special metadata table containing tablet location information
  - Metadata table is stored using BigTable itself

Metadata Table for Locating Tablets

- metadata table helps locate (up to 2^{34}) user tables
- Each metadata table row locates one tablet
  - Stores the (GFS) file locations that store a tablet
  - Stores current tablet server serving the tablet
  - Row size: 1KB for each 100-200MB tablet



- Clients look up a row by traversing 3-level B+-tree type hierarchy
  - With prefetching+caching, most client operations directly access user tablet servers

Metadata table stored on tablet servers, lookup does not require accessing master

Assigning Tablets to Tablet Servers

- Master keeps track of:
  - Current assignment to tablets to tablet servers
  - Unassigned tablets
- When a master starts up, it
  - Acquires a master lock in Chubby
  - Acquires list of live tablet servers from Chubby
  - Gets list of tablets served by asking each tablet server
    - These are assigned tablets
  - Scans the master table to find all tablets
    - Unassigned tablets = all tablets - assigned tablets
  - Assigns the unassigned tablets to tablet servers

Tablet Storage Layout

- The tablet data and logs are stored in GFS files
- How should the data be stored in the GFS files?
- Problem
  - GFS supports fast file appends, but not overwrites
  - GFS supports large file reads and writes
  - However, modern web applications require support for both
    - Fast indexed small reads, scans (search rows)
    - High-throughput updates (insert rows)

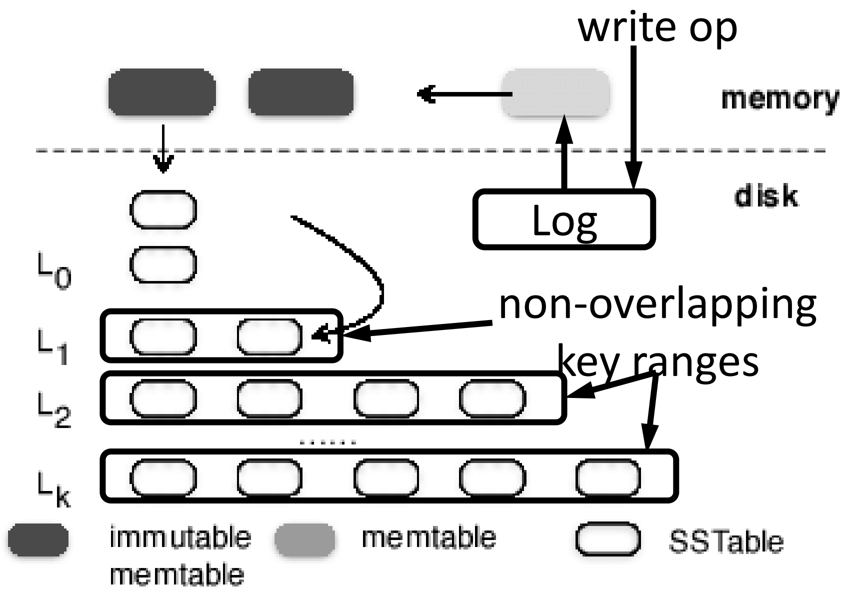
Storage Layout Options

	Sorted Array	Tree, e.g., B+-tree	Log
Search	O(log(n))	O(log(n))	O(n), very slow since a row may be located anywhere in the log
Insert	O(n), very slow since much of the array may need to be rewritten	O(log(n))	O(1)

- A log appends data, so is a good fit for GFS
- Need a structure that improves search performance on logs, without sacrificing much on insert?

Log-Structured Merge (LSM) Trees

- Uses logging + sorted structure
- Write: All data (key, value) is initially written to an in-memory sorted table called memtable
- Flush: memtable is periodically written sequentially to an on-disk sorted, immutable file called sstable (L0 level)
- Compaction: L0 sstables are periodically merged into sorted L1 sstables using immutable ops



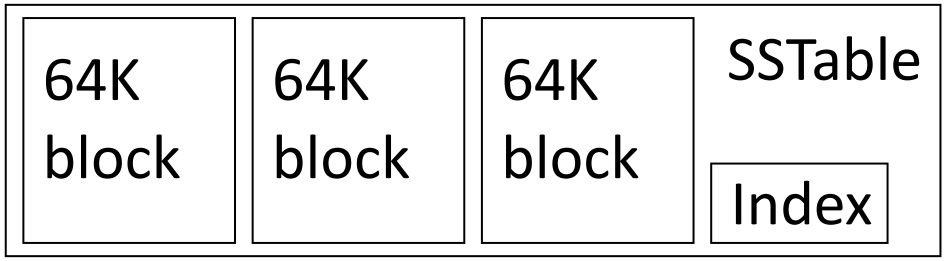
Performance:
  insert: O(1)
  search: O(log^2(n))

Immutable Structures

- Only memtable allows reads and writes
- All SSTables are immutable
  - Contain versioned (timestamped) data
- Allows asynchronous deletes
  - A delete is a new version (tombstone)
  - Previous versions deleted asynchronously during compaction
- Mitigates need for locking
  - Since data is not written in place

SSTable

- Immutable, sorted file of key-value pairs (both strings)
  - key is (row, column, timestamp)



- Contains blocks of data and an index
  - Index maps key range to block
  - Index loaded into memory when SSTable is opened
- Key lookup requires single disk seek, per SSTable
  - Read block into memory (slow)
  - Look up key using binary search within block (fast)

Putting Everything Together

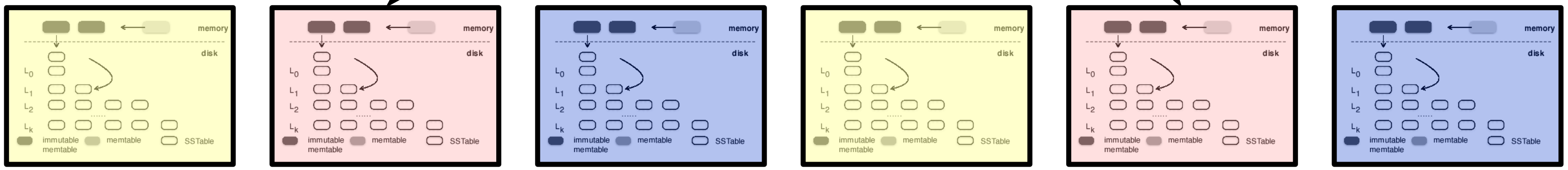
- Clients can group one or more column families in a table, each group in a tablet has its own SSTables
- All SSTables of a tablet served by same tablet server

column group

		anchor
T1	ca.mylook	
T2	com.cnn.www	cnnsi.com, t_4: CNN cnnsi.com, t_2: CNN mylook.ca, t_1: CNN.com
	com.cnn.www/ca	
T3	com.cnnsi.com	

column group

		contents	language
T1	ca.mylook		
T2	com.cnn.www	t_6: <html>… t_5: <html>… t_3: <html>…	EN
	com.cnn.www/ca		
T3	com.cnnsi.com		



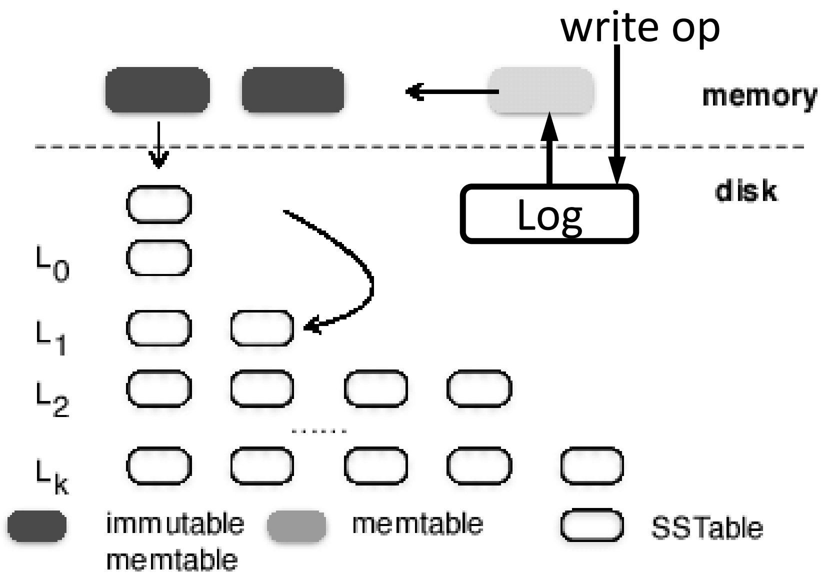
Optimizing Reads: Caching

- Cache reads at tablet servers with two-level caching
- Scan cache
  - Cache key-value pairs from SSTable
  - Temporal locality
- Block cache
  - SSTable blocks read from GFS
  - Spatial locality

Optimizing Reads: Bloom Filters

- Reads need to read from multiple SSTables that make up table
- Each SSTable stores a bloom filter
- Bloom filter is a space efficient data structure that returns true when the (key, value) pair exists in the SSTable (but may return false positives)
- Helps reduce disk accesses when the SSTable doesn’t have matching key, value pair

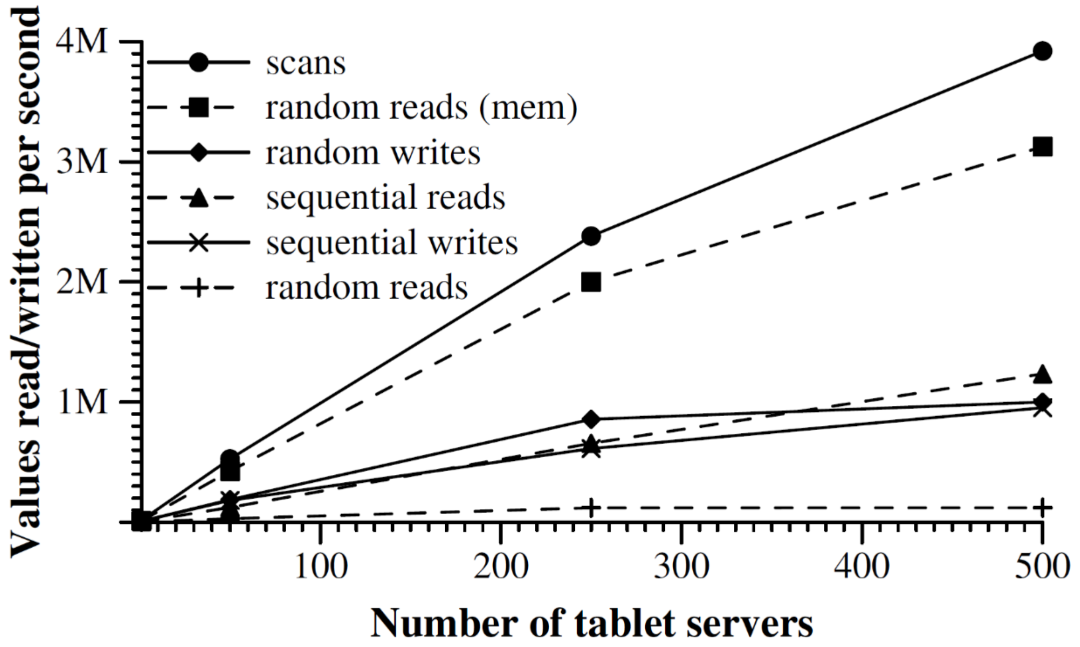
Optimizing Writes: Single Commit Log per Tablet Server



- Use one log per tablet server, not one per tablet
  - Reduces the number of files written, improves seek locality, reduces overhead, etc.
  - Different files would mean writes to different locations on disk
- Complicates recovery after table server fails, since tablets may be loaded on many live tablet servers
  - Few log entries associated with any one tablet in the log
  - Run a parallel sort by key, then log entries for each tablet are close together

Performance

- Random reads are much slower than all other operations
- Sequential reads/writes, random writes, perform better, are comparable
- Random reads from memory are much faster
- Scans are even faster



Bigtable: Pros, Cons

- Pros
  - Can handle massive data and massive objects scalably
  - Supports low-latency access for small data sizes
  - Supports tables with thousands of columns efficiently
  - Allows applications to control data locality
- Cons
  - Weak consistency model (row-level atomic updates)
    - No table-wide integrity constraints
    - However, sufficient for many applications
  - Writing large objects (e.g., videos) causes much write amplification

Some Lessons Learned

- Many types of failure possible, not only fail-stop
  - Memory and network corruption, large clock skew, hung machines, bugs in other systems, extended and asymmetric network partitions, planned and unplanned hardware maintenance
  - Big systems need constant systems-level monitoring
- Delay adding new features until needed
  - E.g., Initially planned for multi-row transaction APIs

Conclusions

- Bigtable is a highly available and scalable database
  - Easy to scale by adding tablet servers to the system
  - Separating storage from serving data simplifies design, fault tolerance, self management, etc.
- If you are Google
  - Significant advantages of building own storage system
  - Data model applicable to many of their applications
- Very influential
  - Apache Hbase based on BigTable design
  - Apache Cassandra offers BigTable data model

Discussion

Q1

- Bigtable is called a NoSQL database
  - What are the differences/tradeoffs between a NoSQL database and a traditional database?

Q2

- What are the most significant differences between GFS and Bigtable in terms of workloads?

Q3

- What are the most significant differences between GFS and Bigtable in terms of system architecture?

Q4

- How is fault tolerance provided in Bigtable? How does it compare with fault tolerance in GFS?

Q5

- BigTable ensures atomic reads/writes at row granularity. Why is this consistency guarantee relatively easy to implement in BigTable?