# GFS and BigTable

Lecture 17, cs262a

Ion Stoica & Ali Ghodsi,
UC Berkeley
March 19, 2018

## Today’s Papers

The Google File System,

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, SOSP’03

http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

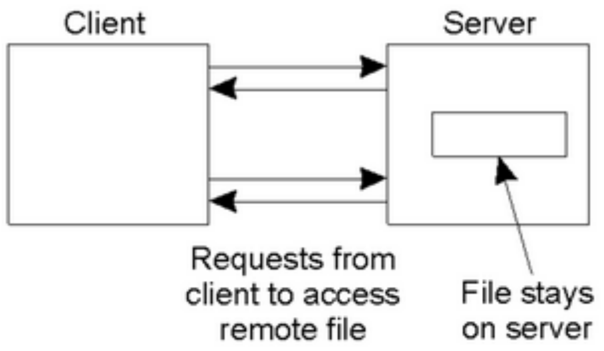
BigTable: A Distributed Storage System for Structured Data.

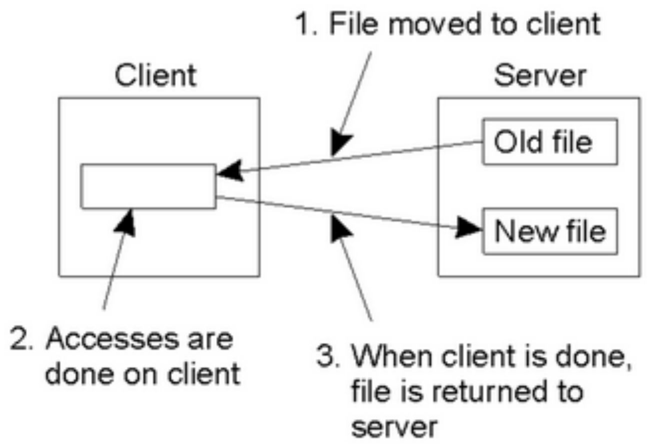
Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, OSDI’06

http://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf

## Google File System

# Distributed File Systems before GFS




1. File moved to client
Client
Server
Old file
New file
2. Accesses are done on client
3. When client is done, file is returned to server

a) The remote access model.

b) The upload/download model

**Pros/Cons?**

## What is Different?

Component failures are norm rather than exception, why?

- File system consists of 100s/1,000s commodity storage servers
- Comparable number of clients

Much bigger files, how big?

- GBs file sizes common
- TBs data sizes: hard to manipulate billions of KB size blocks

## What is Different?

Files are mutated by appending new data, and not overwriting

- Random writes practically non-existent
- Once written, files are only read sequentially

Own both applications and file system → can effectively co-designing applications and file system APIs

- Relaxed consistency model
- Atomic append operation: allow multiple clients to append data to a file with no extra synchronization between them

## Assumptions and Design Requirements (1/2)

Should monitor, detect, tolerate, and recover from failures

Store a modest number of large files (millions of 100s MB files)

- Small files supported, but no need to be efficient

Workload

- Large sequential reads
- Small reads supported, but ok to batch them
- Large, sequential writes that append data to files
- Small random writes supported but no need to be efficient

## Assumptions and Design Requirements

Implement well-defined semantics for concurrent appends

- Producer-consumer model
- Support hundreds of producers concurrently appending to a file
- Provide atomicity with minimal synchronization overhead
- Support consumers reading the file as it is written

High sustained bandwidth more important than low latency

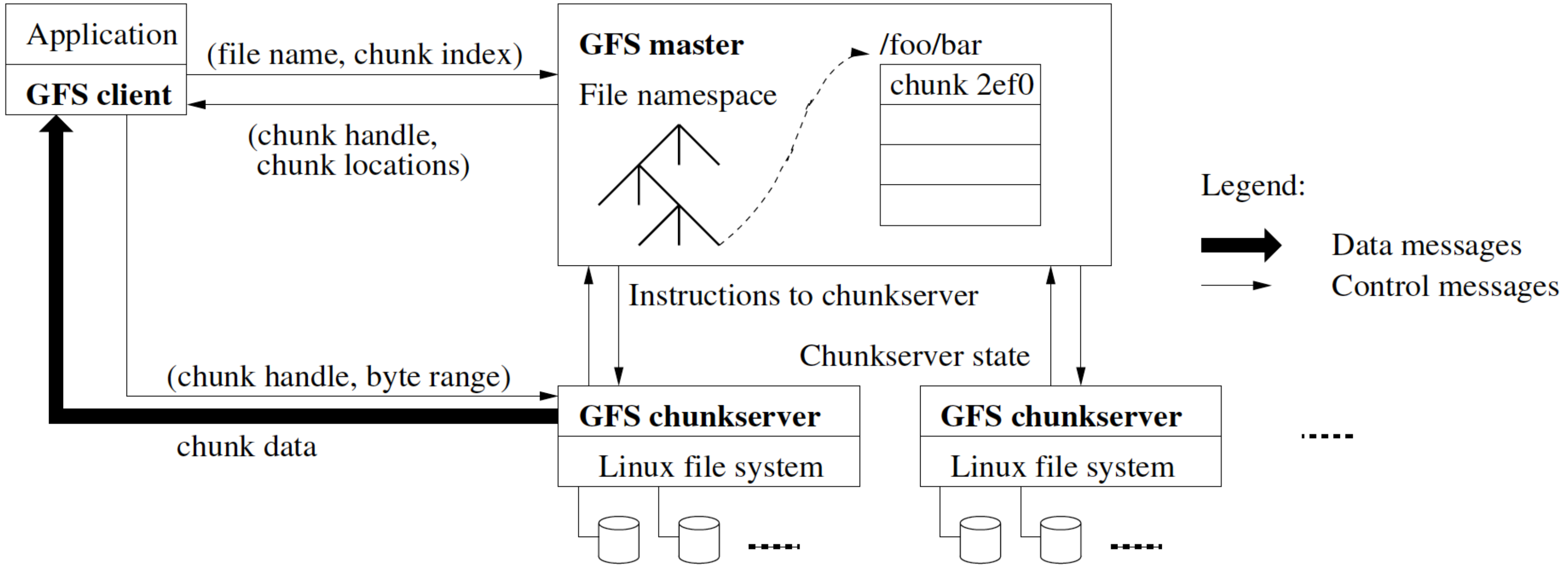
## API

Not POSIX, but..

- Support usual ops: create, delete, open, close, read, and write
- Support snapshot and record append operations

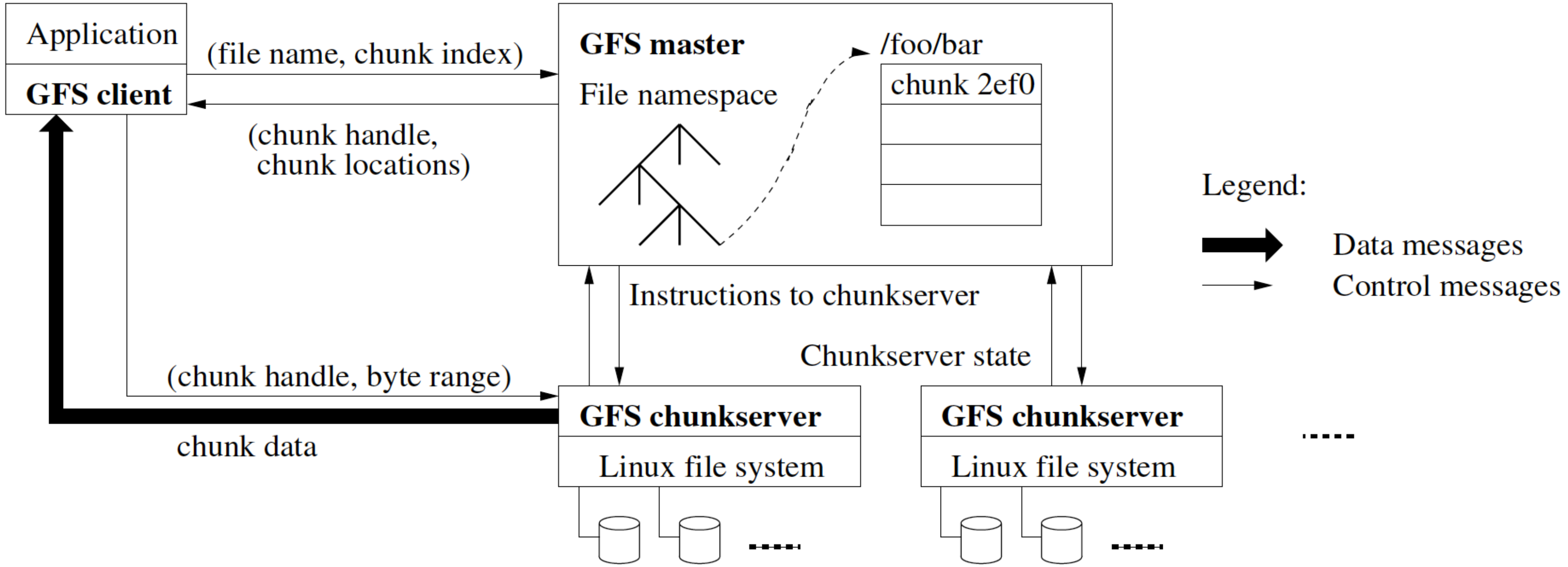
# Architecture

Single Master (why?)



## Architecture

64MB chunks identified by unique 64 bit identifier



## Discussion

64MB chunks: pros and cons?

How do they deal with hotspots?

How do they achieve master fault tolerance?

How does master achieve high throughput, perform load balancing, etc?

## Consistency Model

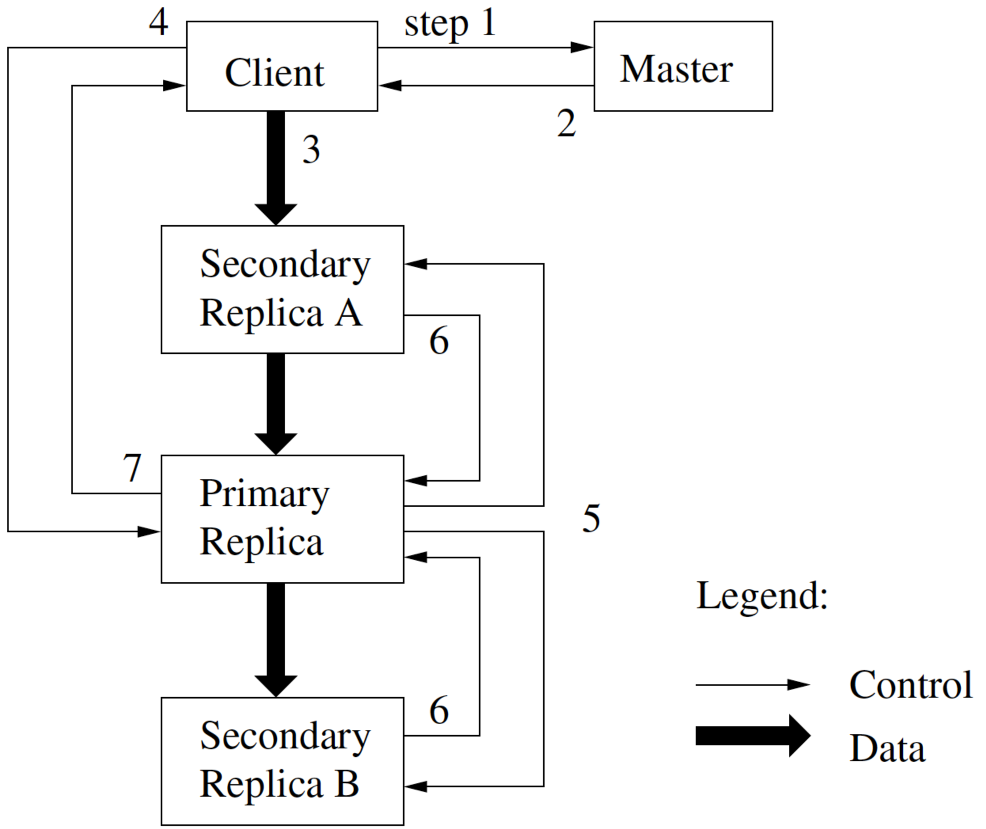
Mutation: write/append operation to a chunk

Use lease mechanism to ensure consistency:

- Master grants a chunk lease to one of replicas (primary)
- Primary picks a serial order for all mutations to the chunk
- All replicas follow this order when applying mutations
- Global mutation order defined first by
  - lease grant order chosen by the master
  - serial numbers assigned by the primary within lease

If master doesn't hear from primary, it grant lease to another replica after lease expires

# Write Control and data Flow



## Atomic Writes Appends

Client specifies only the data

GFS picks offset to append data and returns it to client

Primary checks if appending record will exceed chunk max size

If yes

- Pad chunk to maximum size
- Tell client to try on a new chunk

# Master Operation

## Namespace locking management

- Each master operation acquires a set of locks
- Locks are acquired in a consistent total order to prevent deadlock

## Replica Placement

- Spread chunk replicas across racks to maximize availability, reliability, and network bandwidth

## Others

### Chunk creation

- Equalize disk utilization
- Limit the number of creations on same chunk server
- Spread replicas across racks

### Rebalancing

- Move replica for better disk space and load balancing.
- Remove replicas on chunk servers with below average free space

## Summary

GFS meets Google storage requirements

- Optimized for given workload
- Simple architecture: highly scalable, fault tolerant

Why is this paper so highly cited?

- Changed all DFS assumptions on its head
- Thanks for new application assumptions at Google

BigTable

## Motivation

Highly available distributed storage for structured data, e.g.,

- URLs: content, metadata, links, anchors, page rank
- User data: preferences, account info, recent queries
- Geography: roads, satellite images, points of interest, annotations

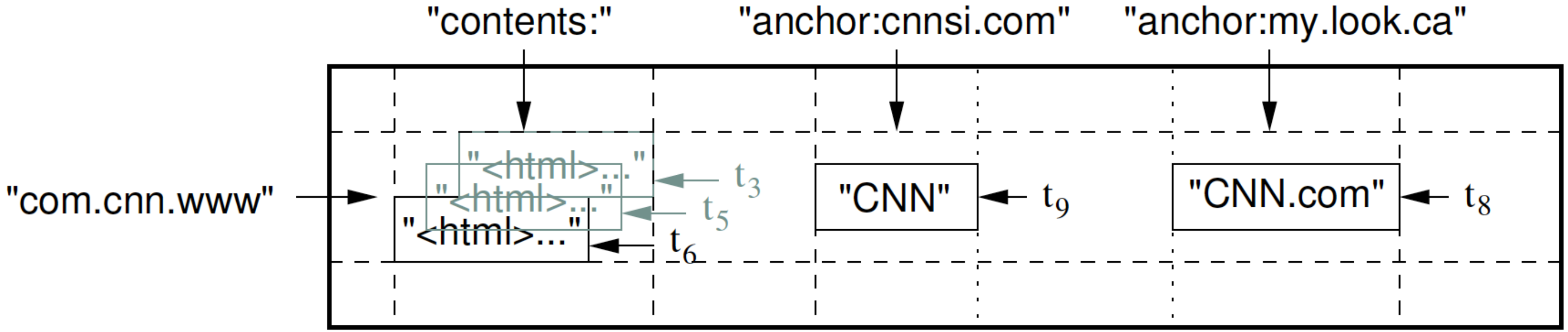
Large scale

- Petabytes of data across thousands of servers
- Billions of URLs with many versions per page
- Hundreds of millions of users
- Thousands of queries per second
- 100TB+ satellite image data

# (Big) Tables

“A BigTable is a sparse, distributed, persistent multidimensional sorted map”

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



## Column Families

Column Family

- Group of column keys
- Basic unit of data access
- Data in a column family is typically of the same type
- Data within the same column family is compressed

Identified by family:qualifier, e.g.,

- “language”:language_id
- “anchor”:referring_site
  - Example: <a href="http://www.w3.org/">CERN</a> appearing in www.berkeley.edu

Cell content

Referring site

## Timestamps

Each cell in a Bigtable 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

## Tablets

Table partitioned dynamically by rows into tablets

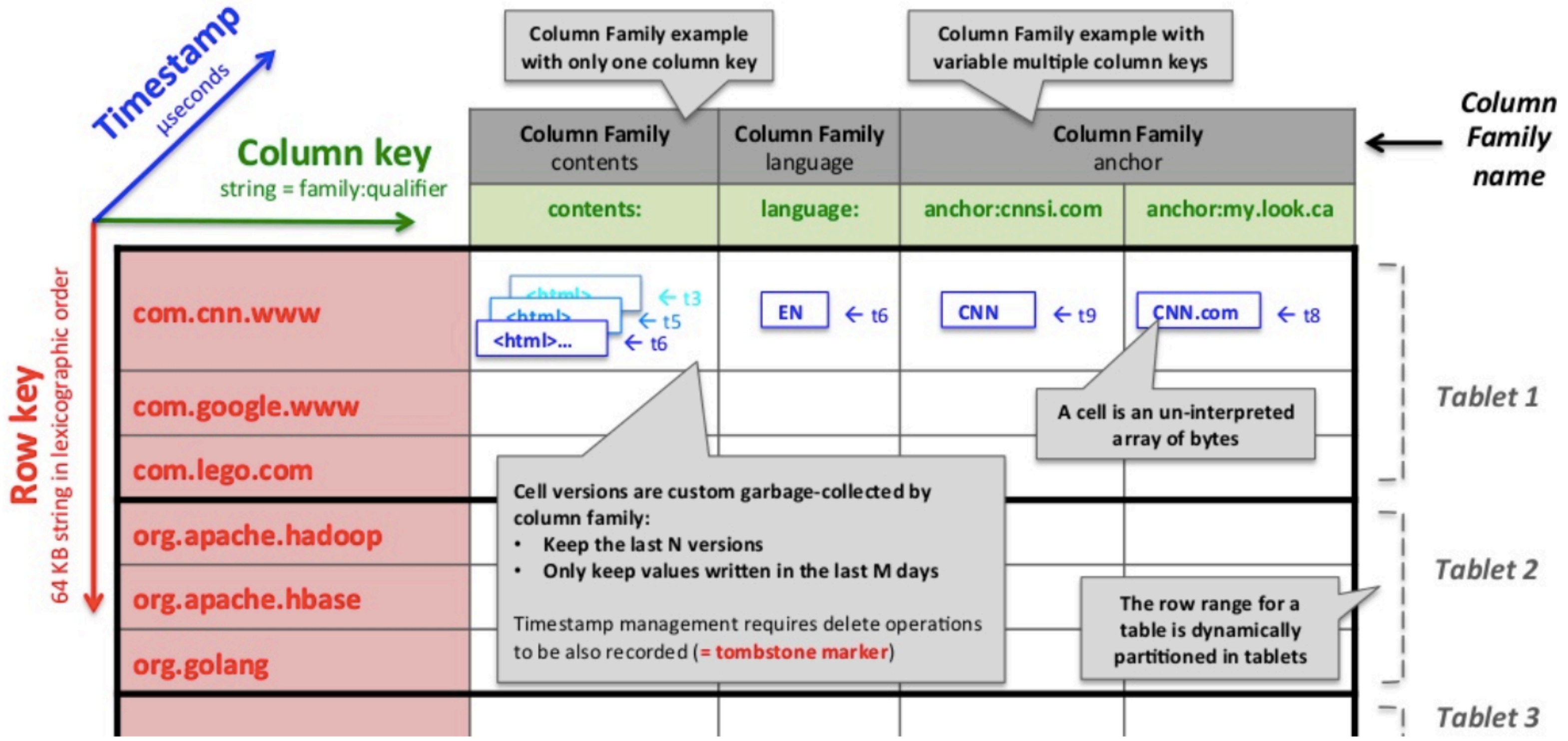
Tablet = range of contiguous rows

- Unit of distribution and load balancing
- Nearby rows will usually be served by same server
- All rows are sorted
- Accessing nearby rows requires communication with small # of servers
- Usually, 100-200 MB per tablet

Users can control related rows to be in same tablet by row keys

- E.g., store maps.google.com/index.html under key com.google.maps/index.html

# Data model



A cell is an un-interpreted array of bytes

Cell versions are custom garbage-collected by column family:

- Keep the last N versions
- Only keep values written in the last M days

Timestamp management requires delete operations to be also recorded (= tombstone marker)

The row range for a table is dynamically partitioned in tablets

from https://www.slideshare.net/romain_jacotin/undestand-google-bigtable-is-as-easy-as-playing-lego-bricks-lecture-by-romain-jacotin

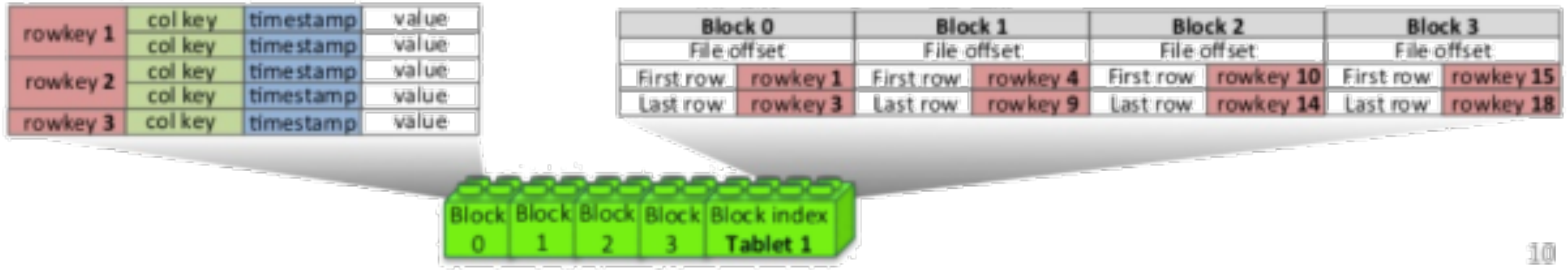
## SSTable

Immutable, sorted file of key-value pairs

Chunks of Blocks plus an index

- Index of block ranges, not values
- Index loaded into memory when SSTable is opened
- Lookup is a single disk seek

Client can map SSTable into mem

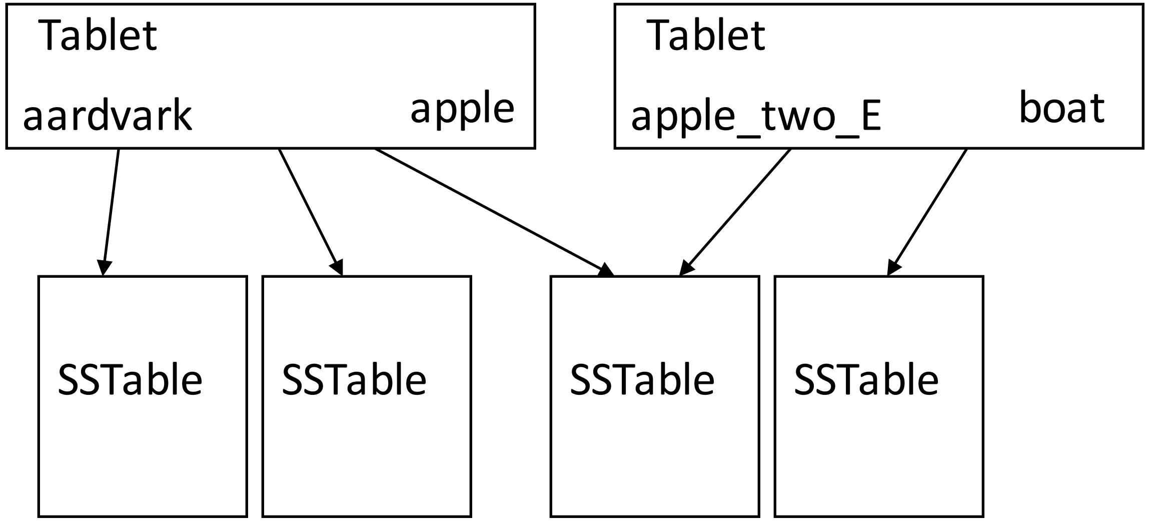


from https://www.slideshare.net/romain_jacotin/undestand-google-bigtable-is-as-easy-as-playing-lego-bricks-lecture-by-romain-jacotin

## Putting Everything Together

SSTables can be shared

Tablets do not overlap, SSTables can overlap



(from www.cs.cmu.edu/~chensm/Big_Data_reading.../Gibbons_bigtable-updated.ppt)

## API

Tables and column families: create, delete, update, control rights

Rows: atomic read and write, read-modify-write sequences

Values: delete, and lookup values in individual rows

Others

- Iterate over subset of data in table
- No transactions across rows, but support batching writes across rows

## Architecture

Google-File-System (GFS) to store log and data files.

- SSTable file format

Chubby as a lock service (another lecture)

- Ensure at most one active master exists
- Store bootstrap location of Bigtable data
- Discover tablet servers
- Store Bigtable schema information (column family info for each table)
- Store access control lists

## Implementation

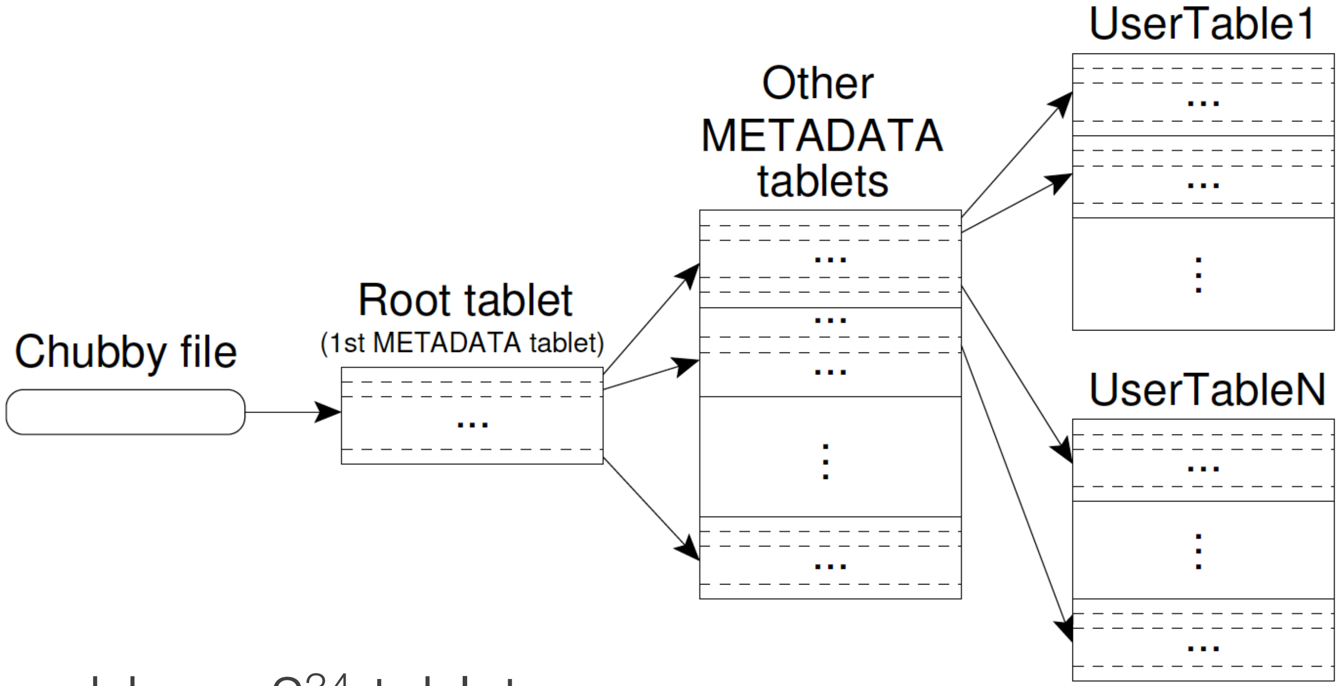
Components:

- Library linked with every client
- Master server
- Many tablet servers

Master responsible for assigning tablets to tablet servers

- Tablet servers can be added or removed dynamically
- Tablet server store typically 10-1000 tablets
- Tablet server handle read and writes and splitting of tablets
- Client data does not move through master

# Tablet Location Hierarchy



Able to address $2^{34}$ tablets

## Tablet Assignment

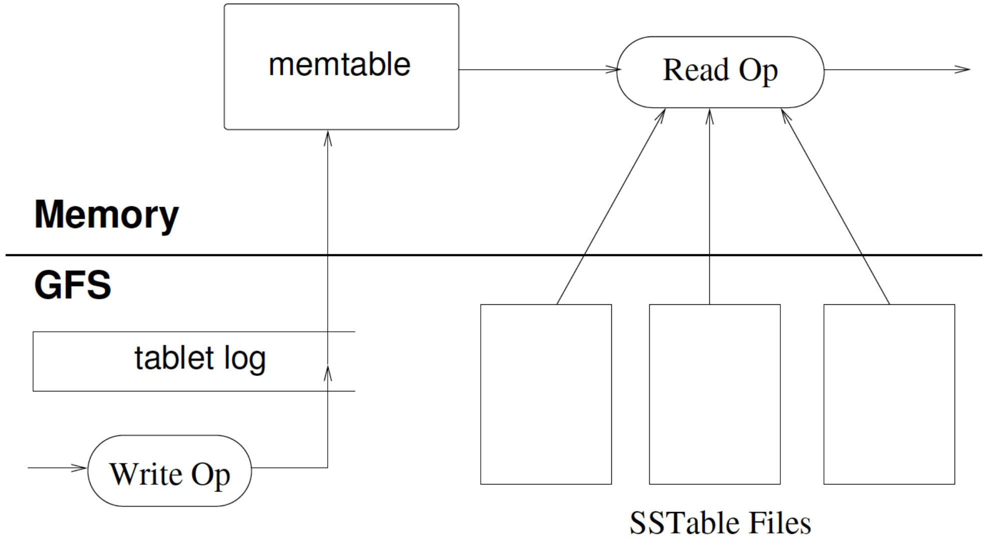
Master keeps track of live tablet servers, current assignments, and unassigned tablets

- Master assigns unassigned tablets to tablet servers
- Tablet servers are linked to files in Chubby directory

Upon a master starting

- Acquire master lock in Chubby
- Scan live tablet servers
- Get list of tablets from each tablet server, to find out assigned tablets
- Learn set of existing tablets → adds unassigned tablets to list

## Tablet Representation



Tablet recovery:

- METADATA: list of SSTables that comprise a tablet and set of redo points
- Reconstructs the memtable by applying all of the updates that have committed since the redo points

## BigTable vs. Relational DB

No data independence:

- Clients dynamically control whether to serve data form memory or disk
- Client control locality by key names

Uninterpreted values: clients can serialize, deserialize data

No multi-row transactions

No table-wide integrity constraints

API: C++, not SQL(no complex queries)

(from www.cs.cmu.edu/~chensm/Big_Data_reading.../Gibbons_bigtable-updated.ppt)

## Lessons learned

Many types of failure possible, not only fail-stop:

- Memory and network corruption, large clock skew, hung machines, extended and asymmetric network partitions, bugs in other systems, planned and unplanned hardware maintenance

Delay adding new features until it is clear they are needed

- E.g., Initially planned for multi-row transaction APIs

Big systems need proper systems-level monitoring

Most important: value of simple designs

- Related: use widely used features of systems you depend on as less widely features more likely to be buggy

## Summary

Huge impact

- GFS → HDFS
- BigTable → HBase, HyperTable

Demonstrate the value of

- Deeply understanding the workload, use case
- Make hard tradeoffs to simplify system design
- Simple systems much easier to scale and make them fault tolerant