Google File System

Google File System

- Google needed a good distributed file system
- Why not use an existing file system?
  - Different workload and design priorities
  - GFS is designed for Google apps
  - Google apps are designed for GFS!

Workload Considerations

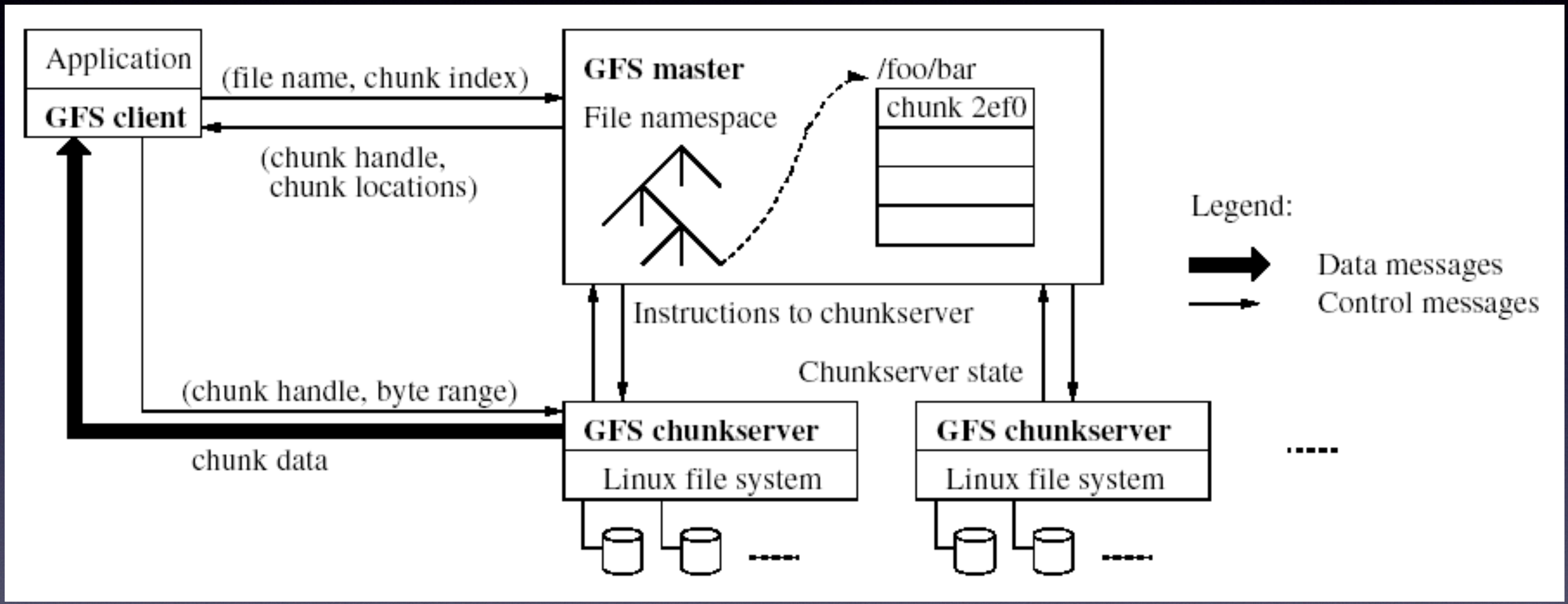
- Optimize cost: don't use high-end machines, instead tolerate failures when they happen
- Dedicated computers
  - In 2000, 2500+ for search; 15K+ by 2004, and 250K+ by 2007
- "Modest" number of huge files; few million of 100MB files
- Files are write-once, mostly appended to (perhaps concurrently)
- Large streaming reads; high throughput favored over low latency

- What are the design choices made by GFS?

GFS Design Decisions

- Files stored as chunks (fixed size: 64MB)
- Reliability through replication
  - each chunk replicated over 3+ chunkservers
- Simple master to coordinate access, keep metadata
- No data caching! Why?
- Familiar interface, but customize the API
  - focus on Google apps; add snapshot and record append operations

GFS Architecture



Design Aspects

- Shadow masters
- Minimize master involvement
  - Never move data through it (only metadata)
  - Cache metadata at clients
  - Large chunk size
  - Master delegates authority to primary replicas in data mutations

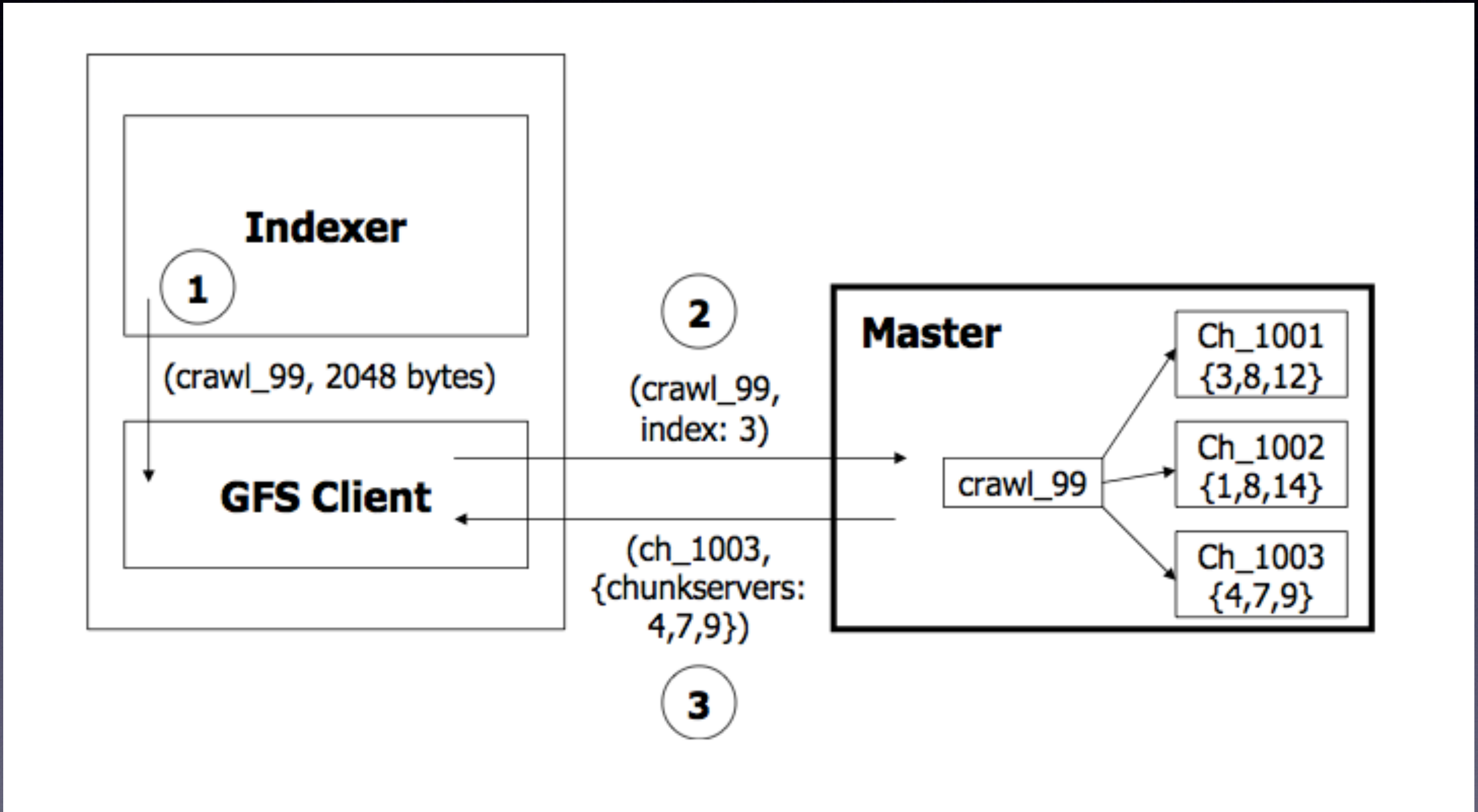
Metadata

- Global metadata is stored on the master
  - File and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk’s replicas
- All in memory (64B/chunk)
  - Few million files ==> can fit all in memory

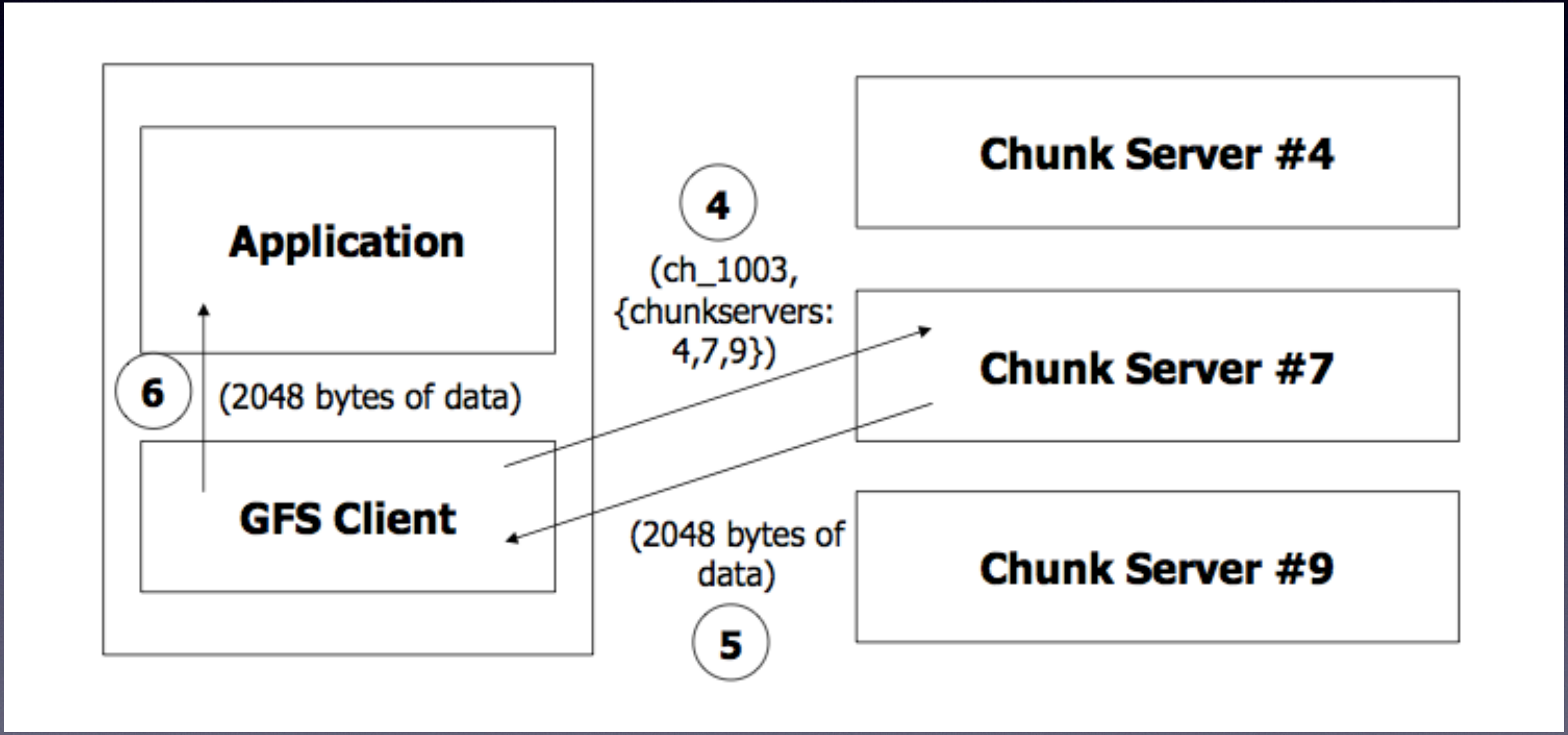
Durability

- Master has an operation log for persistent logging of critical metadata updates
  - each log write is 2PC to multiple remote machines
  - replicated transactional redo log
  - group commit to reduce the overhead
  - checkpoint all state periodically
  - checkpoint: switch to new log and copy snapshot in background

Read Operations

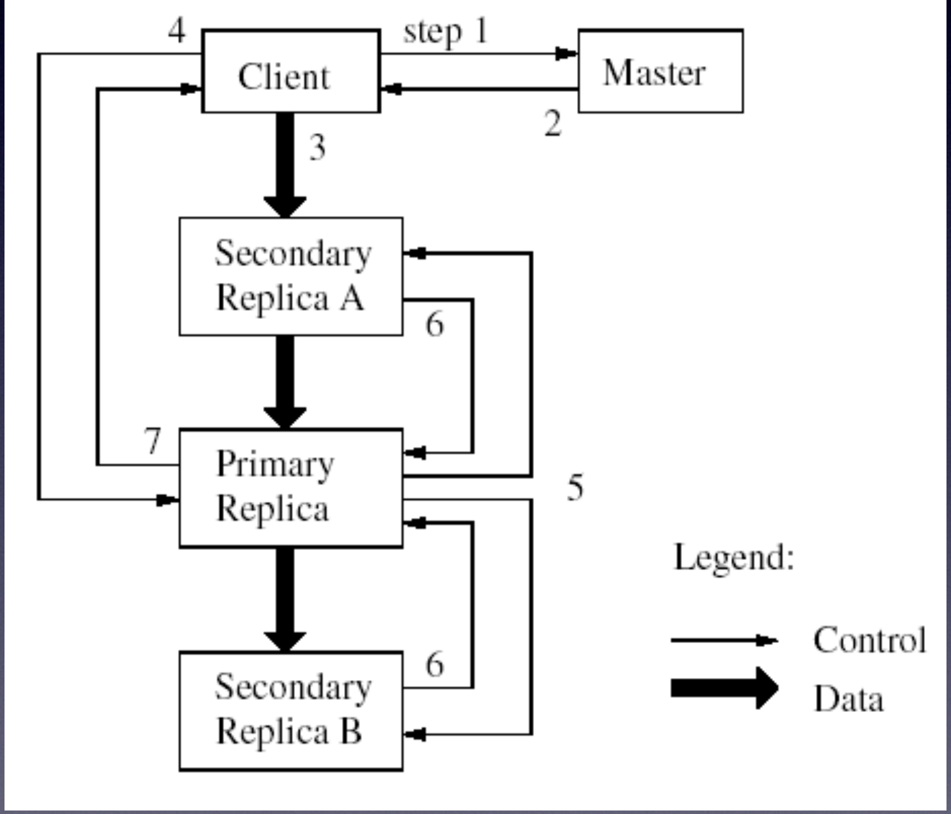


Read Operations



Mutable Operations

- Mutation is write or append
- Goal: minimize master involvement
- Lease mechanism
  - Master picks one replica as primary; gives it a lease
  - Primary defines a serial order of mutations
- Data flow decoupled from control flow



Write Operations

- Application originates write request
- GFS client translates request from (fname, data) --> (fname, chunk-index) sends it to master
- Master responds with chunk handle and (primary+secondary) replica locations
- Client pushes write data to all locations; data is stored in chunkservers' internal buffers
- Client sends write command to primary

Write Operations (contd.)

- Primary determines serial order and writes data in that order to the chunk (also Atomic Appends)
- Primary sends serial order to the secondaries and tells them to perform the write
- Secondaries respond to the primary
- Primary responds back to client
- Note: if write fails at one of the chunkservers, client is informed and retries the write

Data Corruption

- Files stored on Linux and Linux has bugs
  - sometimes silent corruptions
- Files stored on disks and disks are not fail stop
  - stored blocks could be corrupted
  - rare events become common at scale
- Chunkserver maintains per-chunk CRC (64KB)

- Discussion: Identify one thing that you would improve about GFS and suggest an alternative design

~15 years later

- Scale is much bigger
  - now 10K servers instead of 1K, 100 PB instead of 100 TB
- Bigger change: updates to small files
- Around 2010: incremental updates of the Google search index

GFS -> Colossus

- Main scalability limit of GFS: single master
  - fixed by partitioning the metadata
  - ~100M files per master, smaller chunk sizes (1MB)
- Reduce storage overhead using erasure coding

BigTable Motivation

- Lots of (semi)-structured data at Google
  - URLs: contents, crawl metadata, links
  - Per-user data: preference settings, recent queries
  - Geographic locations: physical entities, roads, satellite image data
- Scale is large:
  - Billions of URLs, many versions/page
  - Hundreds of millions of users, queries/sec
  - 100TB+ of 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 projects
- Low-level storage optimizations help performance significantly
    - Much harder to do 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
- Need to support:
  - very high read/write rates (million ops/s)
  - efficient scans over all or interesting subsets
  - efficient joins of large datasets
- Often want to examine data changes over time
  - E.g., contents of web page over multiple crawls

BigTable

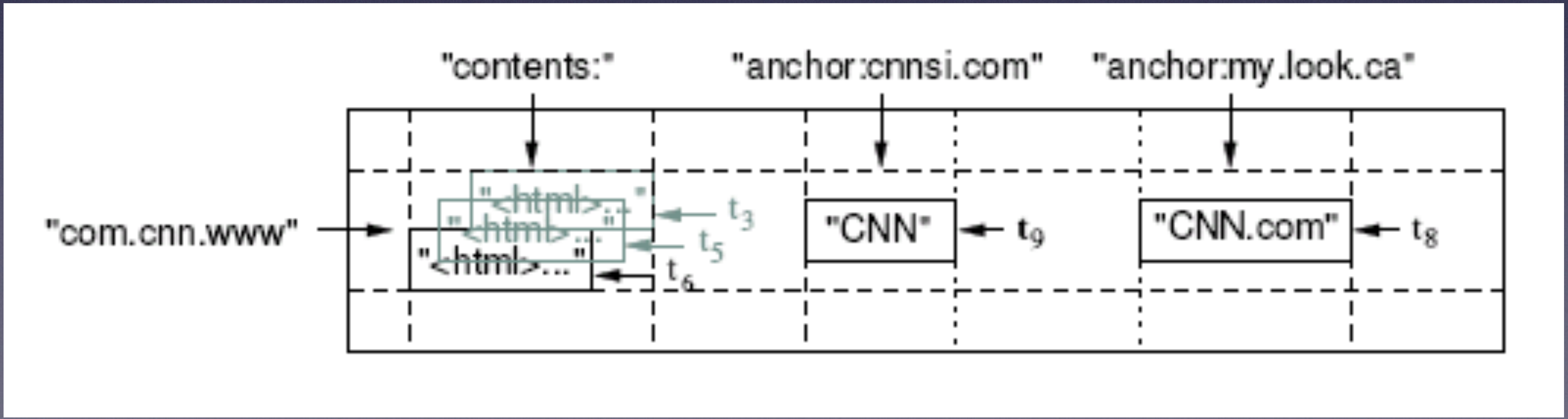
- Distributed multi-level map
- Fault-tolerant, persistent
- Scalable
  - Thousands of servers
  - Terabytes of in-memory data
  - Petabytes of disk-based data
  - Millions of reads/writes per sec, efficient scans
- Self-managing: add servers, adjust load, etc.

BigTable Overview

- Data Model, API
- Implementation structure
  - Tablets, compactions, locality groups, ...
- Details
  - Optimizations, compression, replication, ...

Basic Data Model

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



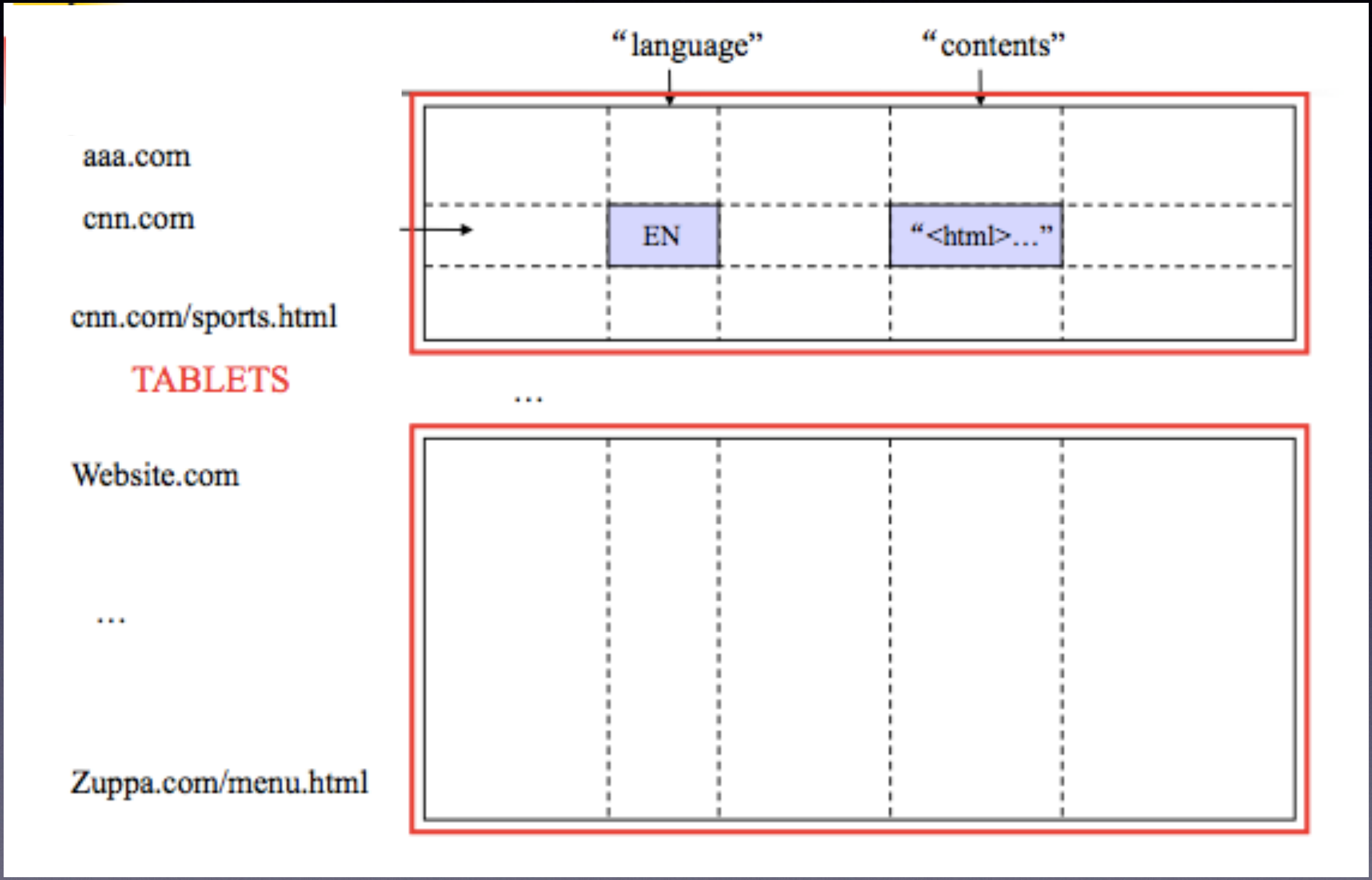
Rows

- Name is an arbitrary string
  - Access to data in a row is atomic
  - Row creation is implicit upon storing data
- Rows ordered lexicographically
  - Rows close together lexicographically usually on one or a small number of machines

Tablets

- Large tables broken into “tablets” at row boundaries
  - Tablet holds contiguous range of rows
  - Aim for 100MB to 200MB of data/tablet
- Serving machine responsible for about 100 tablets
  - Fast recovery (100 machines each pick up 1 tablet from failed machine)
  - Fine-grained load balancing

Tablets & Splitting



Building blocks

- GFS: stores persistent state
- Scheduler: schedules jobs/nodes for tasks
- Lock service: leader election, small metadata storage
- MapReduce: data analytics
- BigTable: semi-structured data store

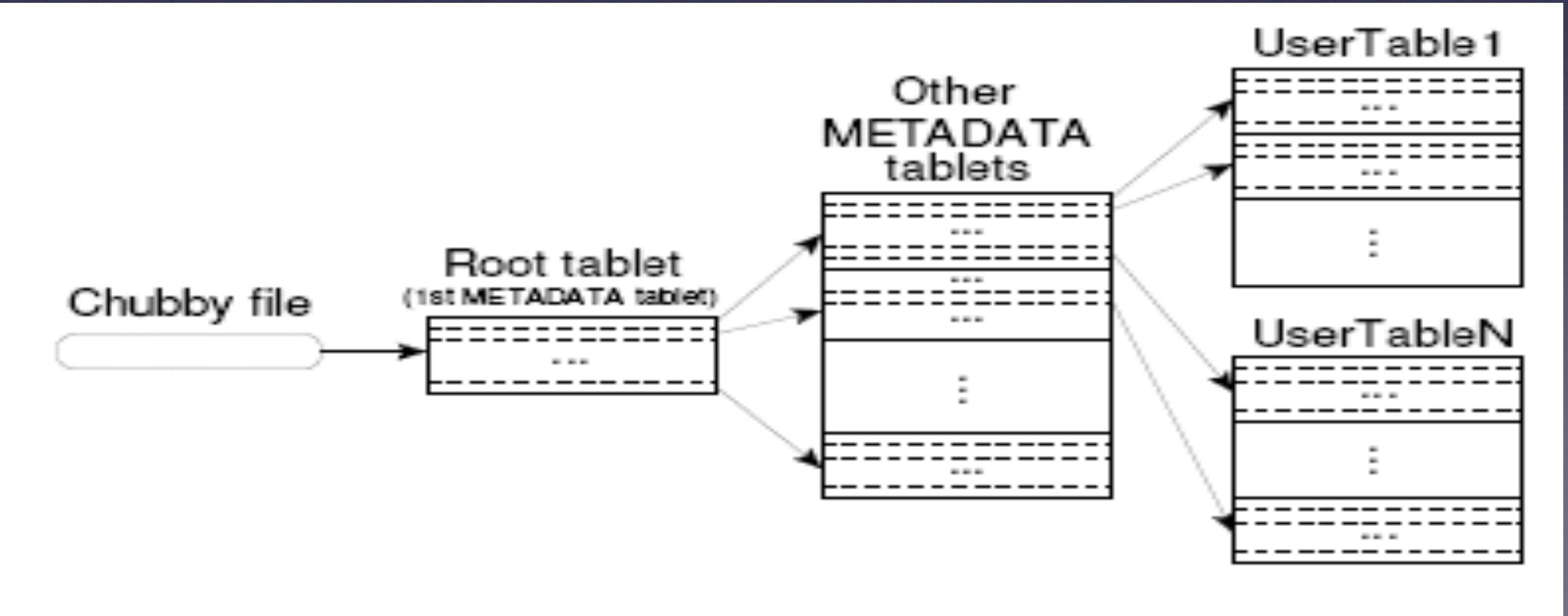
- Question: how do these pieces fit together?

Locating Tablets

- Since tablets move around from server to server, given a row, how do clients find the right machine?
  - Need to find tablet whose row range covers the target row
- One approach: could use the BigTable master
  - Central server almost certainly would be bottleneck in large system
- Instead store special tables containing tablet location info in BigTable cell itself

Locating Tablets

- Approach: 3-level hierarchical lookup scheme for tablets
  - Location is ip:port of relevant server
  - 1st level: bootstrapped from lock server, points to META0
  - 2nd level: Uses META0 data to find owner of META1 tablet
  - 3rd level: META1 table holds location of tablets of all other tables



Tablet Implementation

- Writes go to log then to in-memory table “memtable” (key, value)
- Periodically move in-memory table to disk
  - SSTable is immutable ordered subset of table; range of keys & subset of their columns
  - Tablet = all of the SSTables for one key range plus the memtable
  - some values may be stale (due to new writes)

Basic Implementation

- Reads: maintain in-memory map of keys to SSTables
  - current version is in exactly one SSTable or memtable
  - may have to read many SSTables to get all of the columns
  - in-memory MANIFEST providing key ranges of SSTables
- Compaction:
  - SSTables similar to segments in LFS
  - need to clean old SSTables to reclaim space
  - clean by merging multiple SSTables into new one

LevelDB Settings

- When log reaches 1MB, create Level 0 SSTable
- Up to 4 SSTables in Level 0 with overlapping keys
- Level k has typically a total of 10^k MB of data in 2MB file segments
- Level 0 compaction could do 14MB of read/writes
- Level L+1 compaction could read/write 26MB
  - time is 0.5secs on today’s disks but could be 5secs if backgrounded

- Why does the above structure make sense? How do you optimize the system outlined above?

Bloom filters

- Goal: efficient test for set membership: member(key) -> true/false
  - false ==> definitely not in the set
  - true ==> probably is in the set
- Generally supports adding elements but not removing them
- Basic version: m bit positions, k hash functions
  - For insert: compute k bit locations, set to 1
  - For lookup: compute k locations, check for 1
- BigTable: avoid reading SSTables for elements that are not present; saves many seeks

Bloom Filter