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

Building blocks

- GFS: stores persistent state
- Scheduler: schedules jobs/nodes for tasks
- Lock service: master election
- MapReduce: data analytics
- BigTable: semi-structured data store

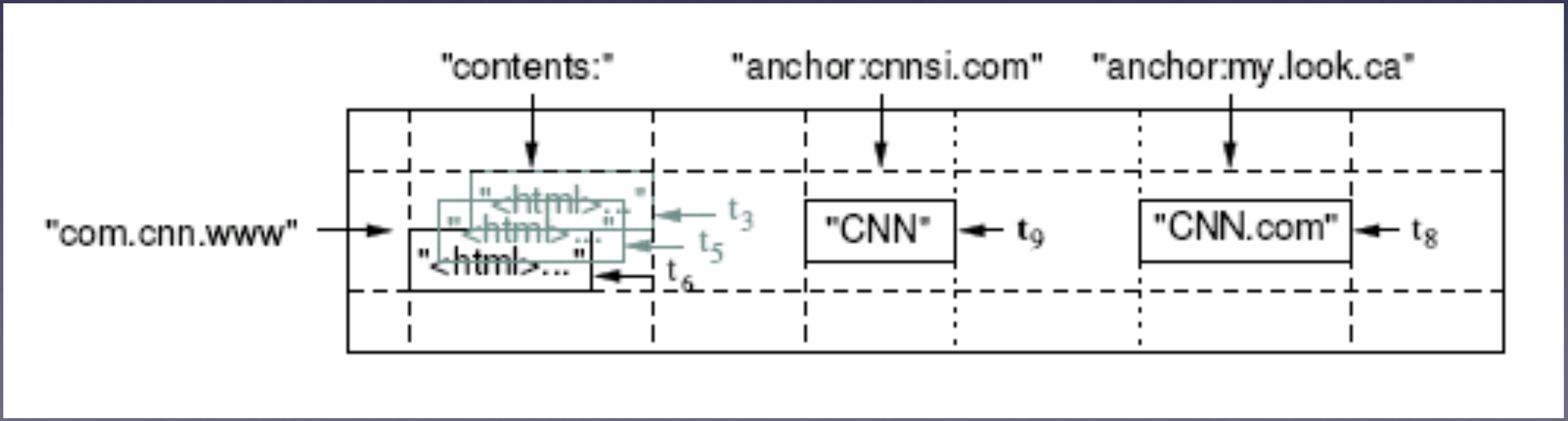
- Question: how do these pieces fit together?

BigTable Overview

- Data Model, API
- 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 Google’s applications



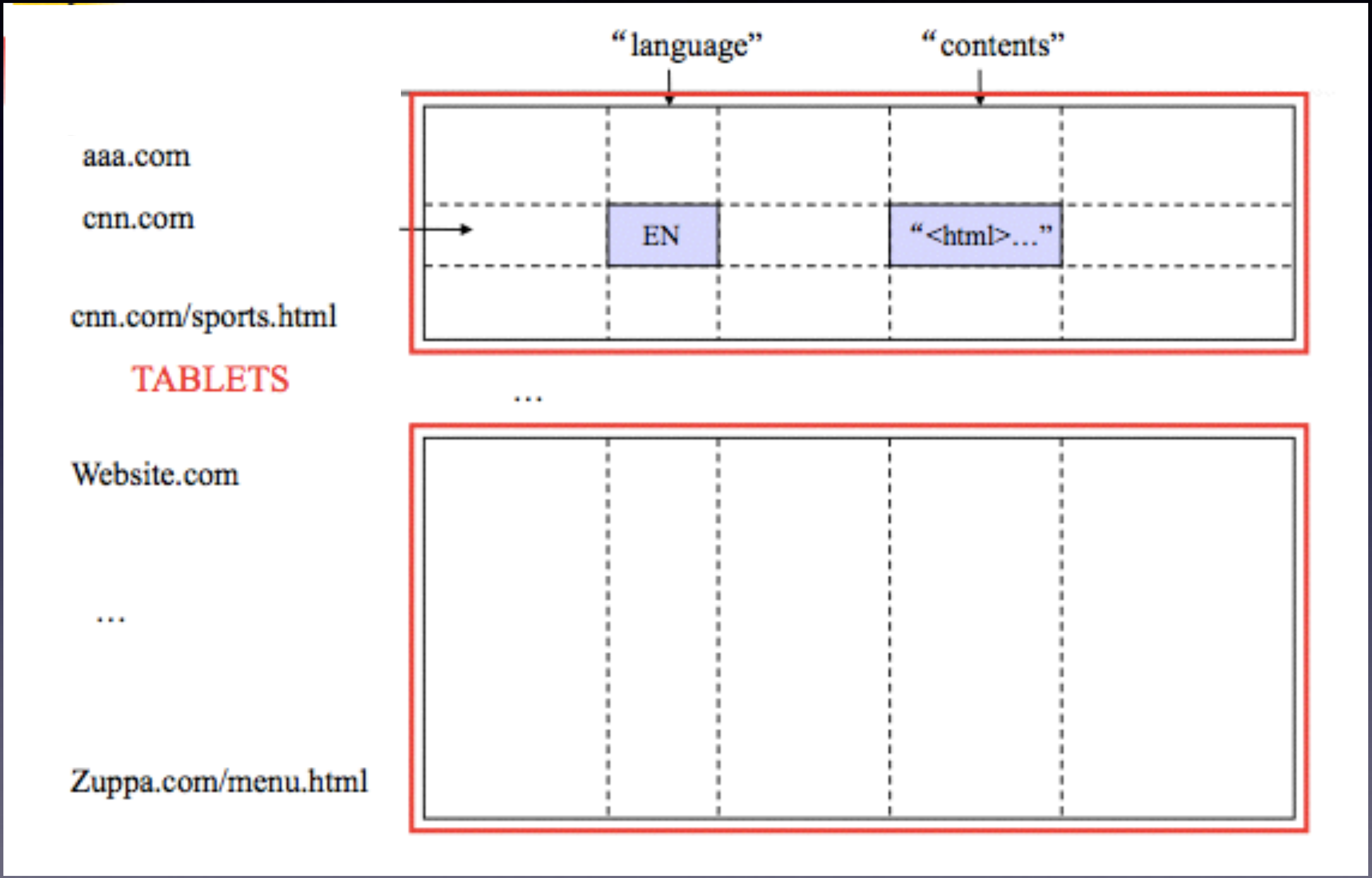
Rows

- 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

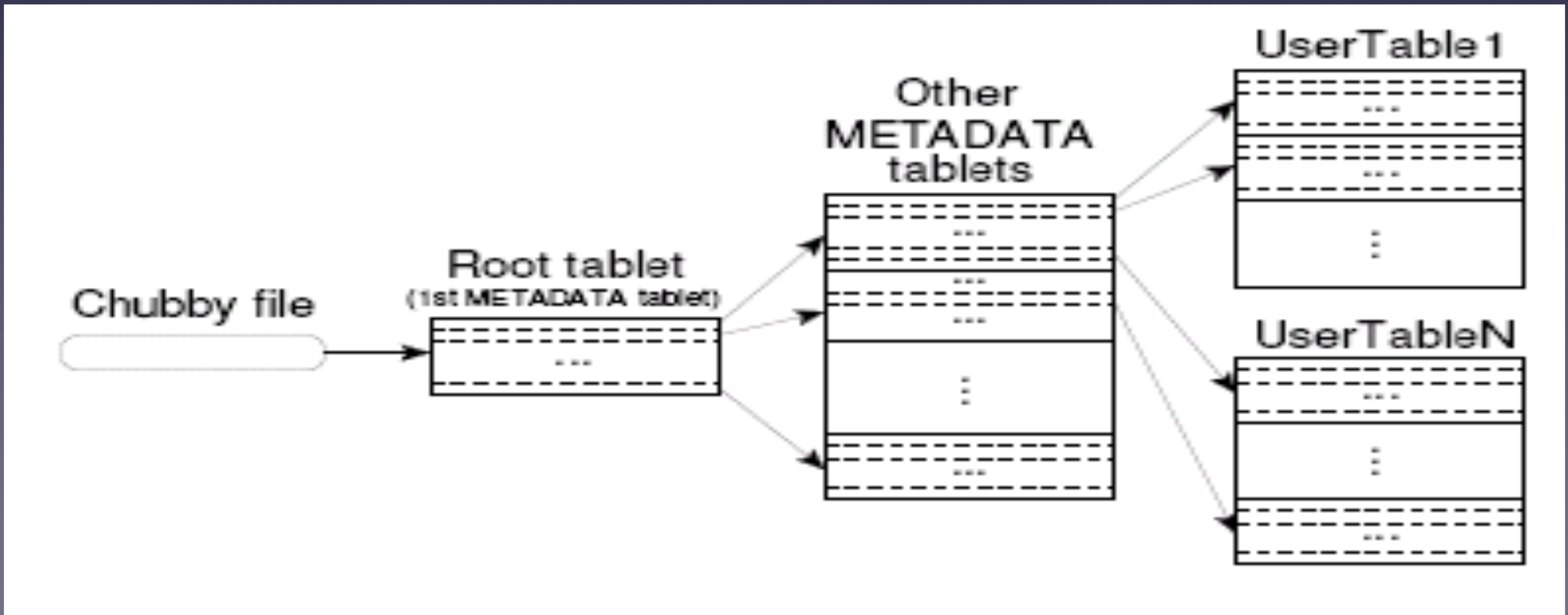


Locating Tablets

- 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



Basic 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 maybe 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
- Compaction:
  - SSTables similar to segments in LFS
  - need to clean old SSTables to reclaim space
  - clean by merging multiple SSTables into new one

- 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