Building Large-Scale Internet Services

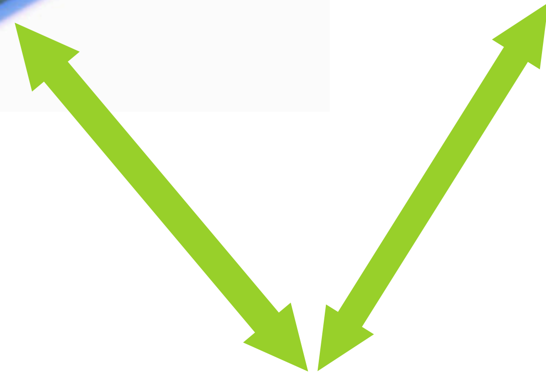
Jeff Dean
jeff@google.com

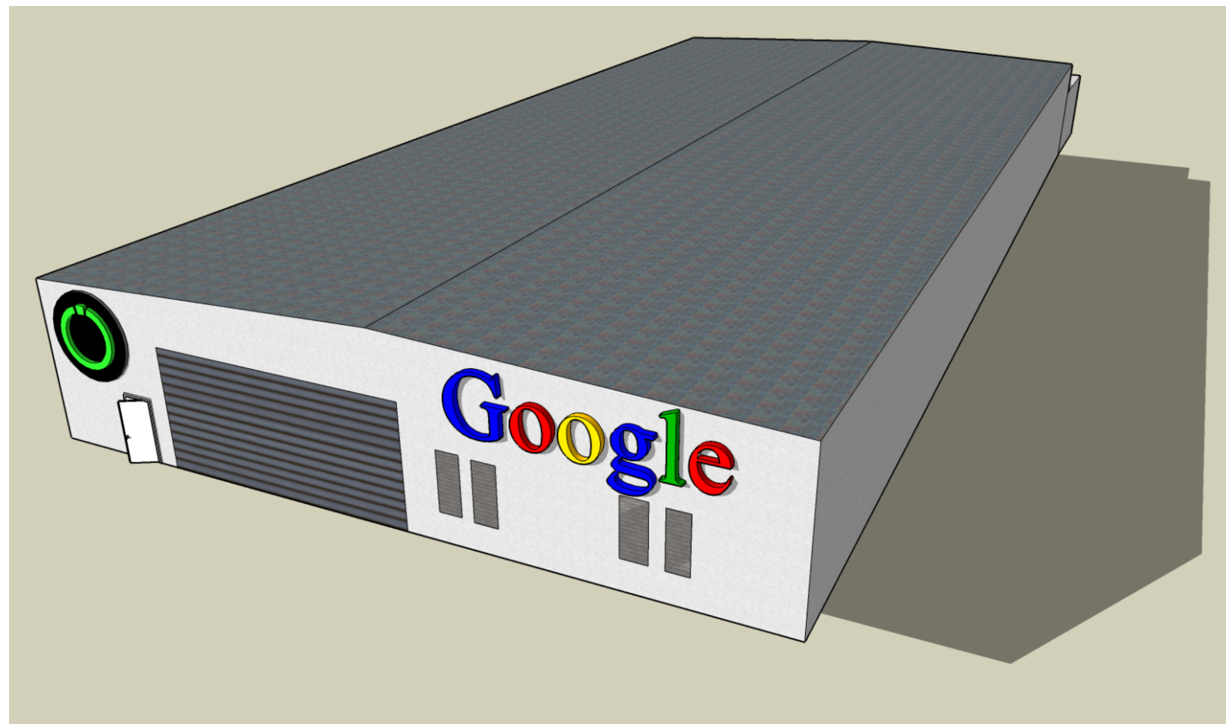
Plan for Today

- Google's computational environment
  - hardware
  - system software stack & its evolution
- Techniques for building large-scale systems
  - decomposition into services
  - common design patterns
- Challenging areas for current and future work

Computing shifting to really small and really big devices

UI-centric devices





Large consolidated computing farms

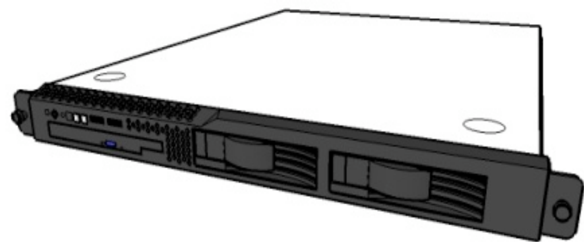
Implications

- Users have many devices
  - expect to be able to access their data on any of them
  - devices have wide range of capabilities/capacities
- Disconnected operation
  - want to provide at least some functionality when disconnected
  - bigger problem in short to medium term
    - long term we'll be able to assume network connection (almost) always available
- Interactive apps require moving at least some computation to client
  - Javascript, Java, native binaries, ...
- Opportunity to build interesting services:
  - can use much larger bursts of computational power than strictly client-side apps

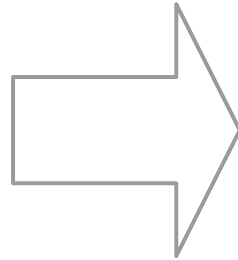
Google

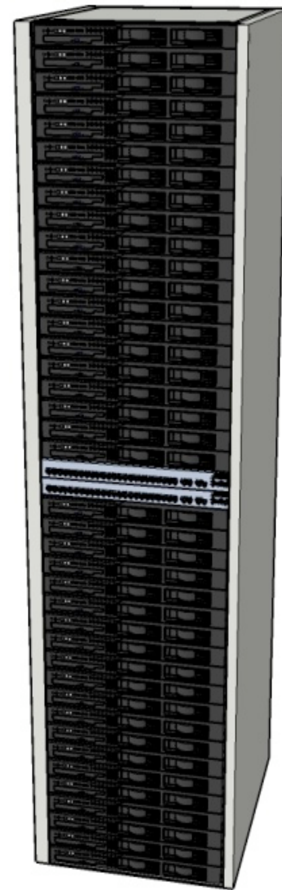
Google’s data center at The Dalles, OR

The Machinery

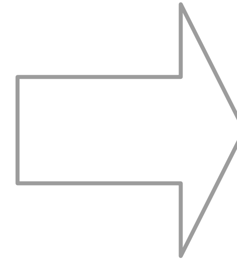


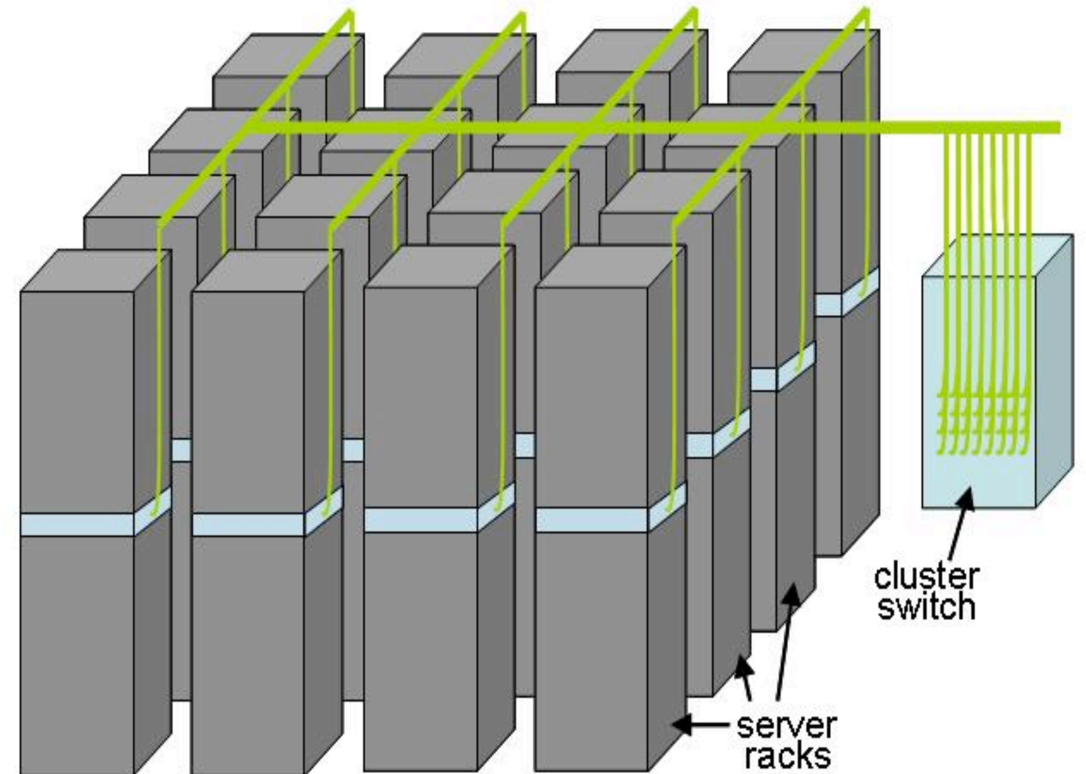
Servers
- CPUs
- DRAM
- Disks





Racks
- 40-80 servers
- Ethernet switch





Clusters

The Joys of Real Hardware

Typical first year for a new cluster:

~1 network rewiring (rolling ~5% of machines down over 2-day span)

~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)

~5 racks go wonky (40-80 machines see 50% packetloss)

~8 network maintenances (4 might cause ~30-minute random connectivity losses)

~12 router reloads (takes out DNS and external vips for a couple minutes)

~3 router failures (have to immediately pull traffic for an hour)

~dozens of minor 30-second blips for dns

~1000 individual machine failures

~thousands of hard drive failures

slow disks, bad memory, misconfigured machines, flaky machines, etc.

Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

The Joys of Real Hardware

Typical first year for a new cluster:

~1 network rewiring (rolling ~5% of machines down over 2-day span)

~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)

~5 racks go wonky (40-80 machines see 50% packetloss)

~8 network maintenances (4 might cause ~30-minute random connectivity losses)

~12 router reloads (takes out DNS and external vips for a couple minutes)

~3 router failures (have to immediately pull traffic for an hour)

~dozens of minor 30-second blips for dns

~1000 individual machine failures

~thousands of hard drive failures

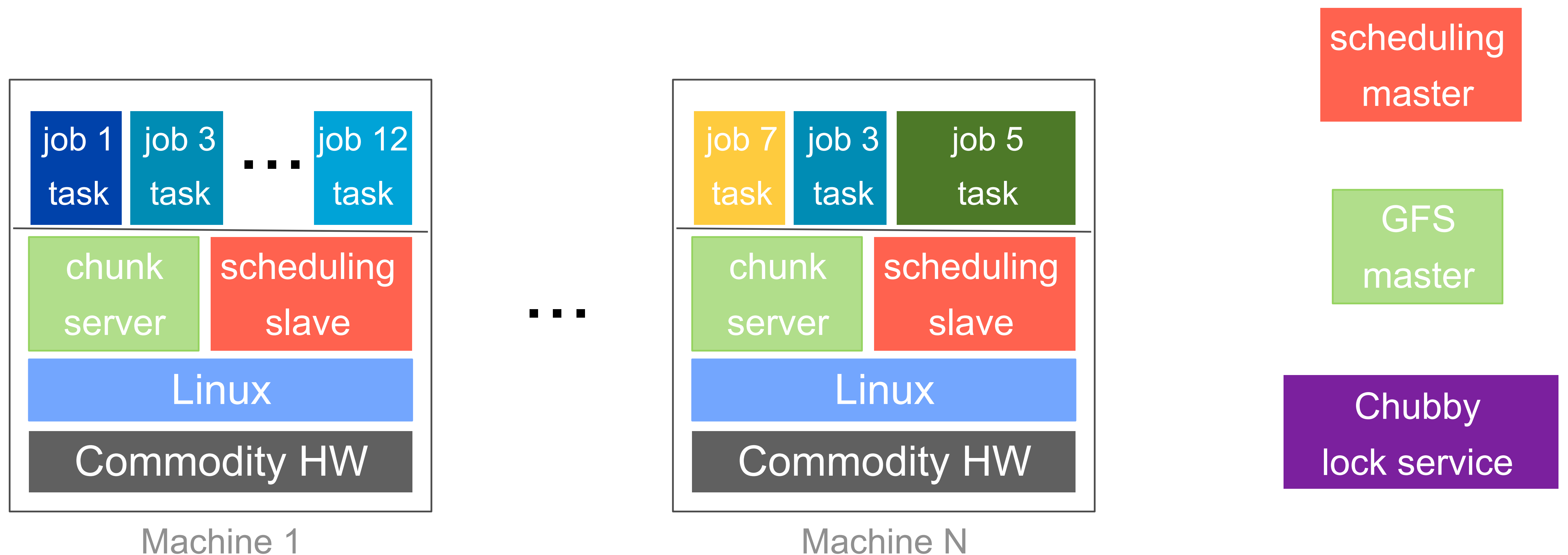
slow disks, bad memory, misconfigured machines, flaky machines, etc.

Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

- Reliability/availability must come from software!

Google Cluster Software Environment

- Cluster is 1000s of machines, typically one or handful of configurations
- File system (GFS or Colossus) + cluster scheduling system are core services
- Typically 100s to 1000s of active jobs (some w/1 task, some w/1000s)
  - mix of batch and low-latency, user-facing production jobs



Google

Some Commonly Used Systems Infrastructure at Google

- GFS & Colossus (next gen GFS)
  - cluster-level file system (distributed across thousands of nodes)
- Cluster scheduling system
  - assigns resources to jobs made up of tasks
- MapReduce
  - programming model and implementation for large-scale computation
- Bigtable
  - distributed semi-structured storage system
  - adaptively spreads data across thousands of nodes

Google

MapReduce

- A simple programming model that applies to many large-scale computing problems

- Hide messy details in MapReduce runtime library:
  - automatic parallelization
  - load balancing
  - network and disk transfer optimizations
  - handling of machine failures
  - robustness
  - **improvements to core library benefit all users of library!**

Google

Typical problem solved by MapReduce

- Read a lot of data
- Map: extract something you care about from each record
- Shuffle and Sort
- Reduce: aggregate, summarize, filter, or transform
- Write the results

Outline stays the same,
map and reduce change to fit the problem

Google

Example: Rendering Map Tiles

Input	Map	Shuffle	Reduce	Output
Geographic feature list	Emit each to all overlapping latitude-longitude rectangles	Sort by key (key= Rect. Id)	Render tile using data for all enclosed features	Rendered tiles

Input

I-5
Lake Washington
WA-520
I-90
...

Map

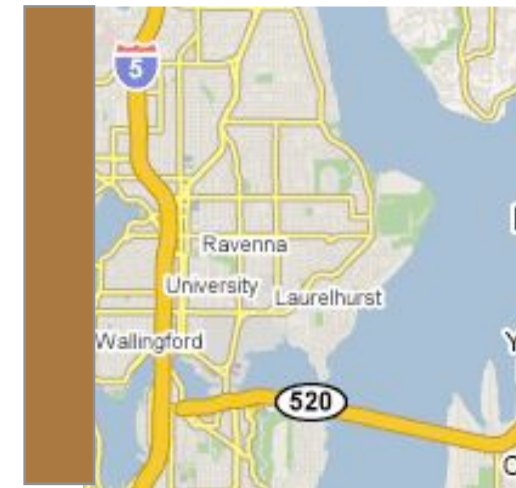
(0, I-5)
(1, I-5)
(0, Lake Wash.)
(1, Lake Wash.)
(0, WA-520)
(1, I-90)
...

Shuffle / Reduce

0
(0, I-5)
(0, Lake Wash.)
(0, WA-520)
...

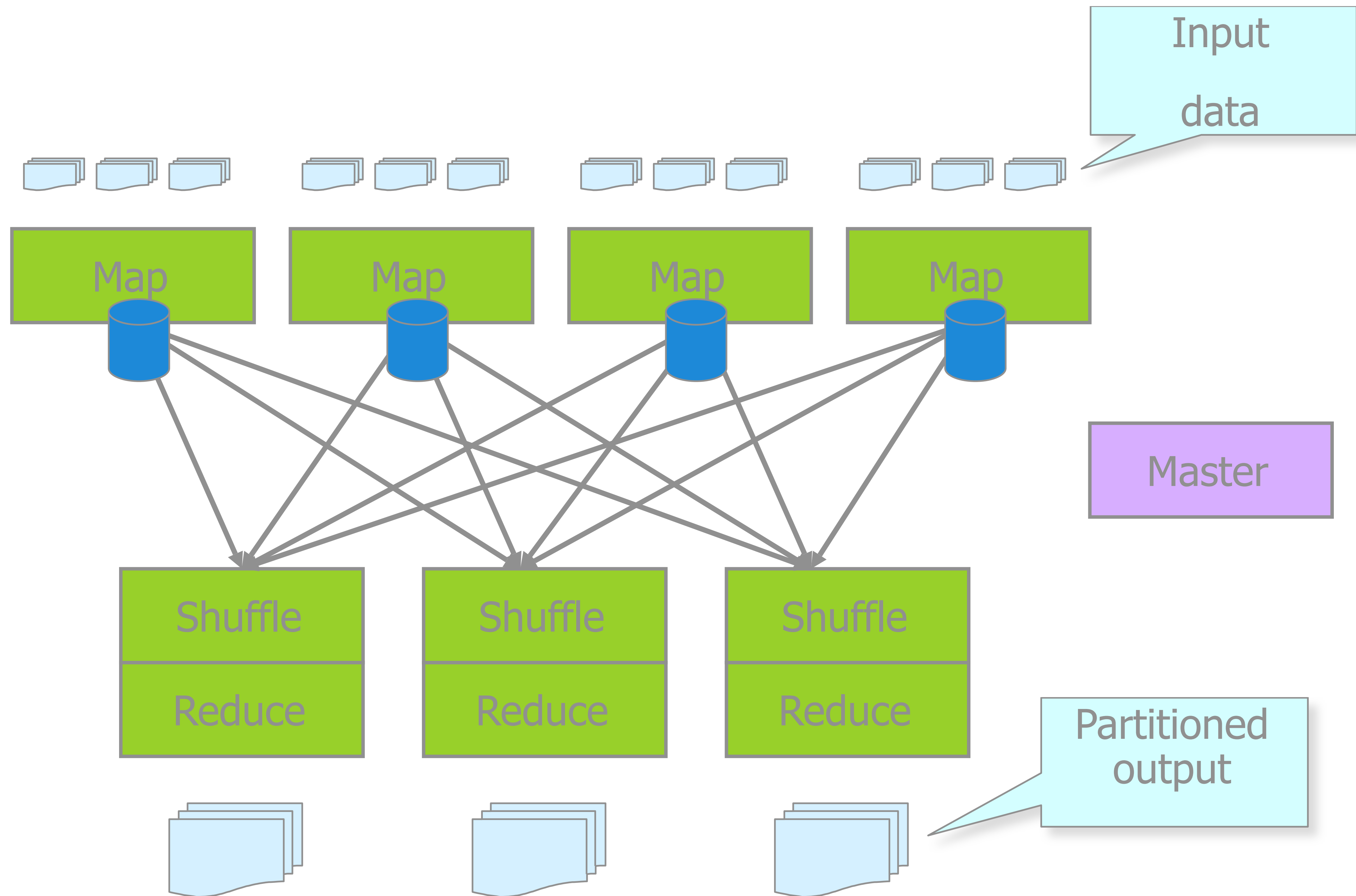
1
(1, I-5)
(1, Lake Wash.)
(1, I-90)
...

Output



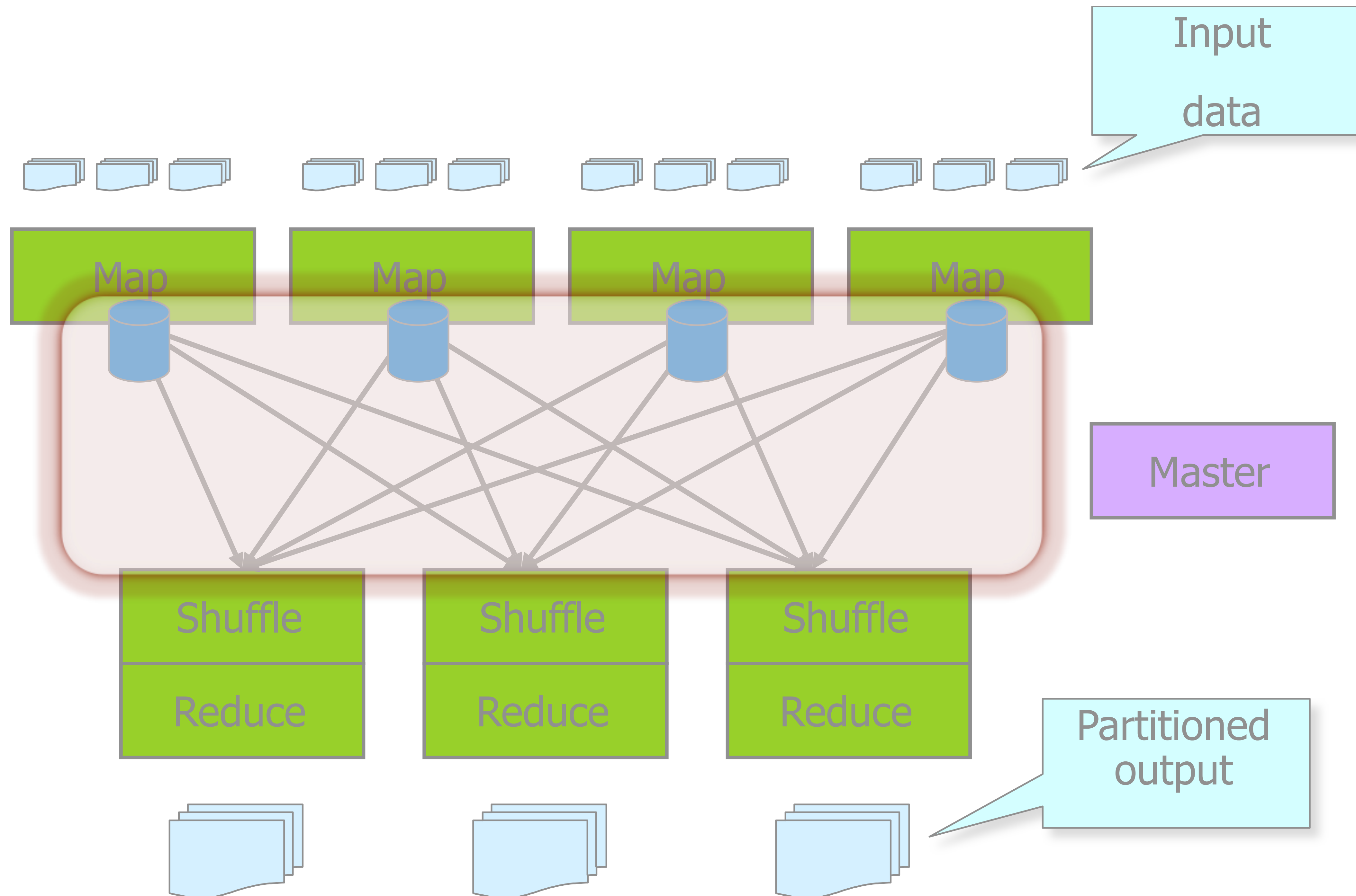
Google

Parallel MapReduce



Google

Parallel MapReduce



For large enough problems, it's more about disk and network performance than CPU & DRAM

Google

MapReduce Usage Statistics Over Time

	Aug, '04	Mar, '06	Sep, '07	May, '10
Number of jobs	29K	171K	2,217K	4,474K
Average completion time (secs)	634	874	395	748
Machine years used	217	2,002	11,081	39,121
Input data read (TB)	3,288	52,254	403,152	946,460
Intermediate data (TB)	758	6,743	34,774	132,960
Output data written (TB)	193	2,970	14,018	45,720
Average worker machines	157	268	394	368

Google

MapReduce in Practice

- Abstract input and output interfaces
  - lots of MR operations don't just read/write simple files
    - B-tree files
    - memory-mapped key-value stores
    - complex inverted index file formats
    - BigTable tables
    - SQL databases, etc.
    - ...
- Low-level MR interfaces are in terms of byte arrays
  - Hardly ever use textual formats, though: slow, hard to parse
  - Most input & output is in encoded Protocol Buffer format
- See "*MapReduce: A Flexible Data Processing Tool*" (CACM, 2010)

BigTable: Motivation

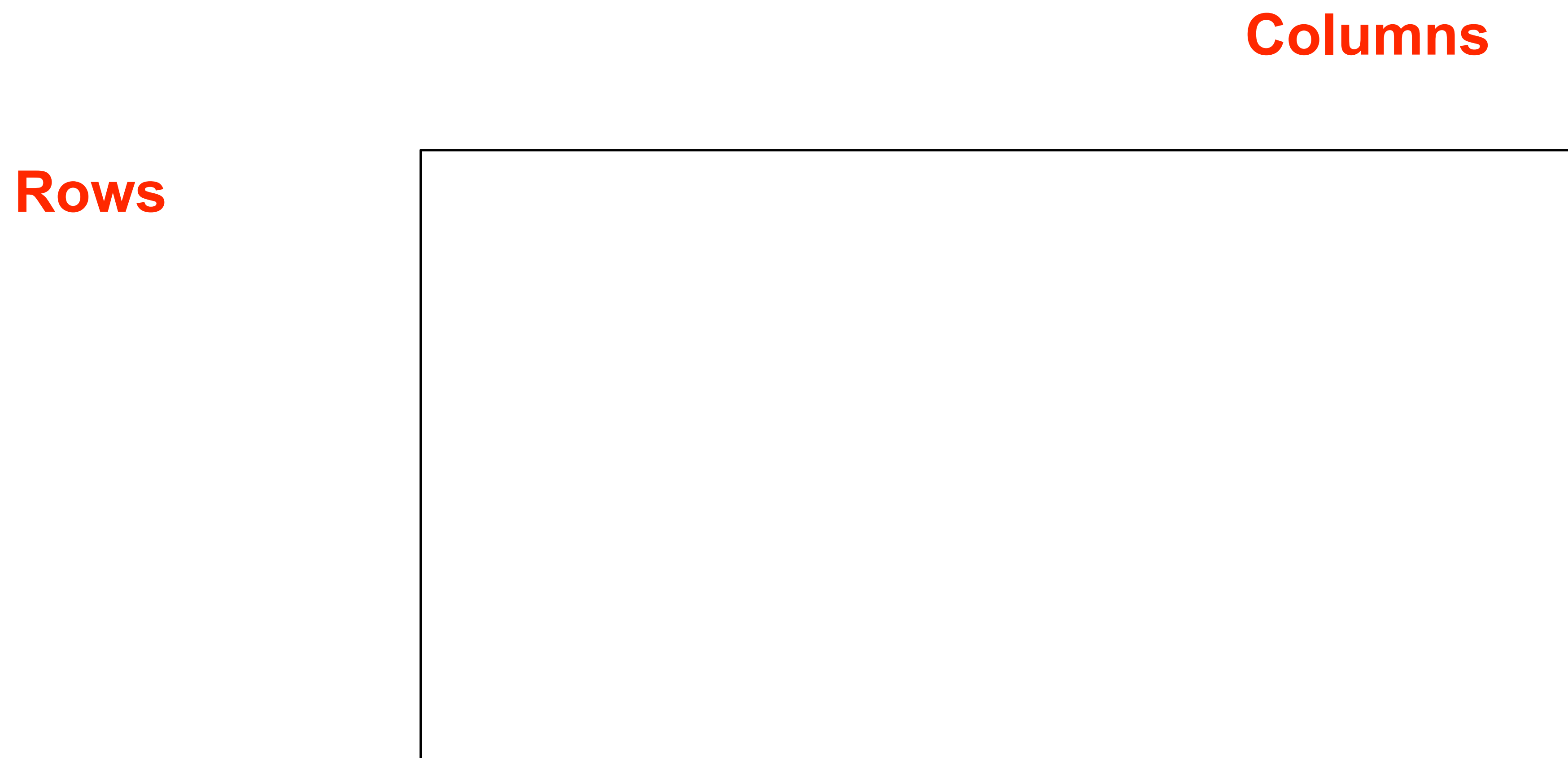
- Lots of (semi-)structured data at Google
  - URLs:
    - Contents, crawl metadata, links, anchors, pagerank, …
  - Per-user data:
    - User preference settings, recent queries/search results, …
  - Geographic locations:
    - Physical entities (shops, restaurants, etc.), roads, satellite image data, user annotations, …
- Scale is large
  - billions of URLs, many versions/page (~20K/version)
  - Hundreds of millions of users, thousands of q/sec
  - 100TB+ of satellite image data

Google

Basic Data Model

- Distributed multi-dimensional sparse map

  (row, column, timestamp) → cell contents

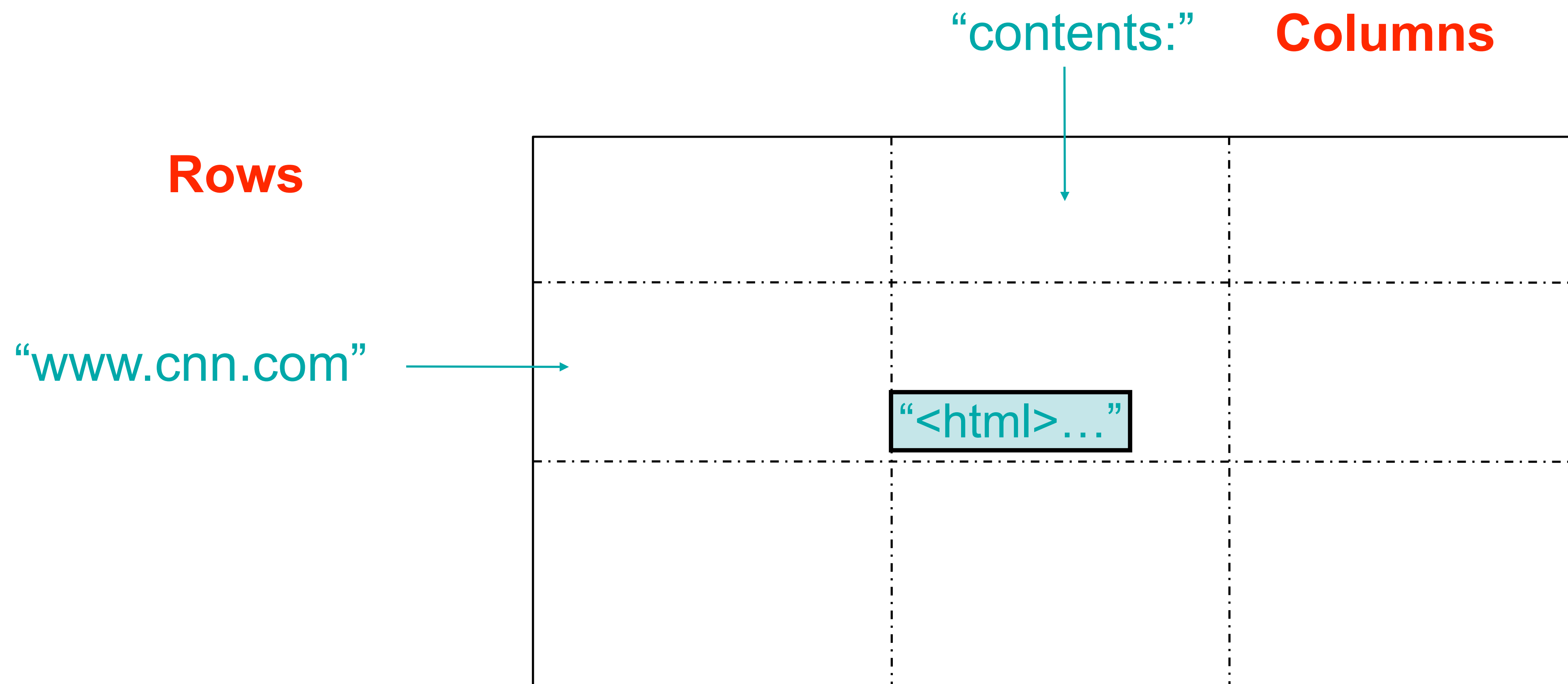


- Rows are ordered lexicographically
- Good match for most of our applications

Basic Data Model

- Distributed multi-dimensional sparse map

  (row, column, timestamp) → cell contents

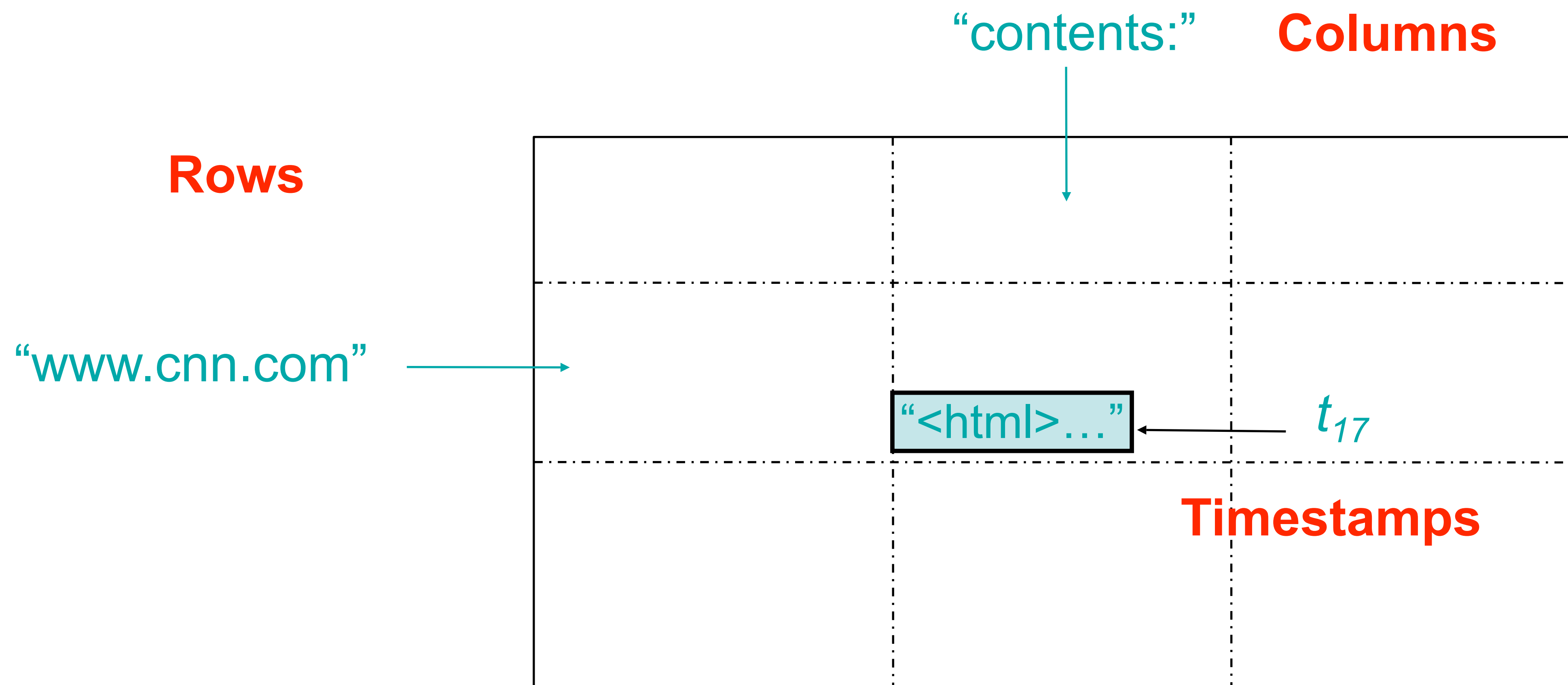


- Rows are ordered lexicographically
- Good match for most of our applications

Basic Data Model

- Distributed multi-dimensional sparse map

  (row, column, timestamp) → cell contents

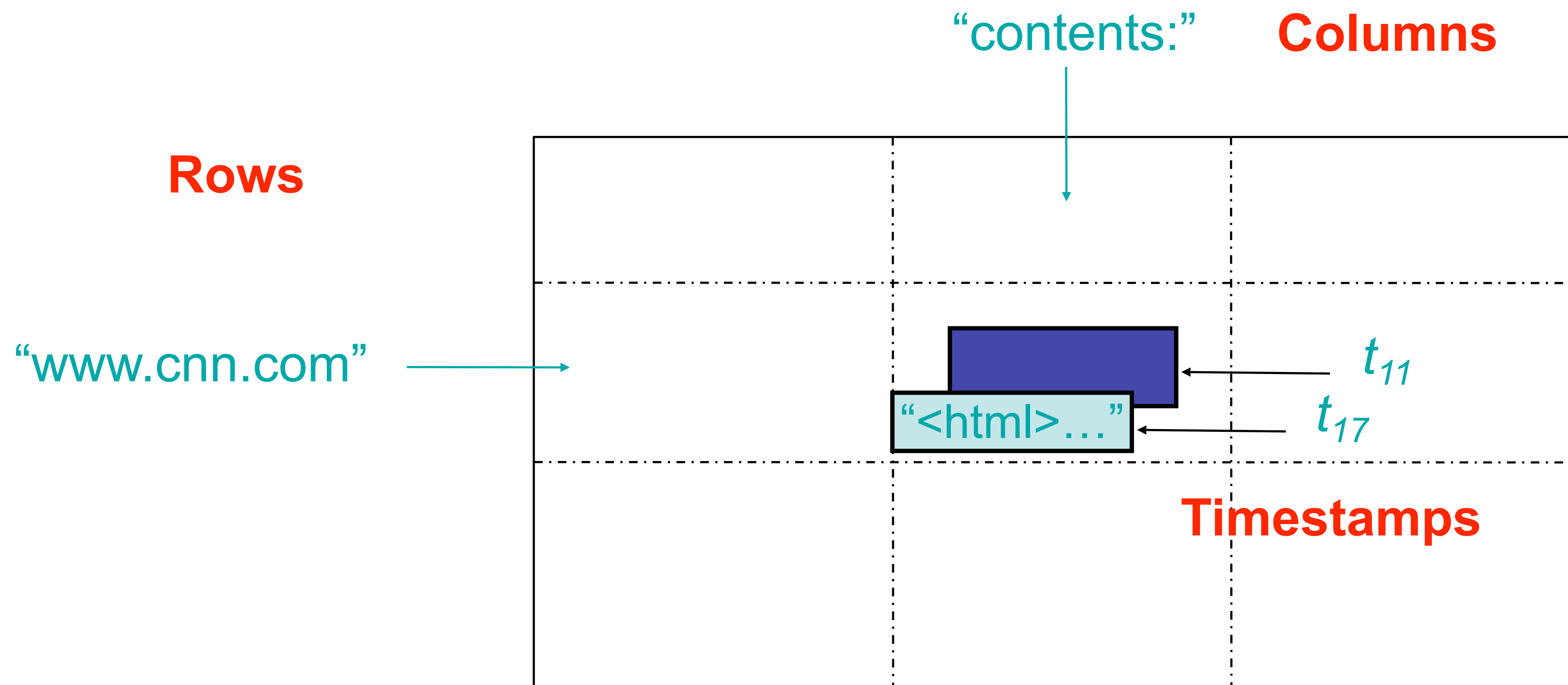


- Rows are ordered lexicographically
- Good match for most of our applications

Basic Data Model

- Distributed multi-dimensional sparse map

  (row, column, timestamp) → cell contents

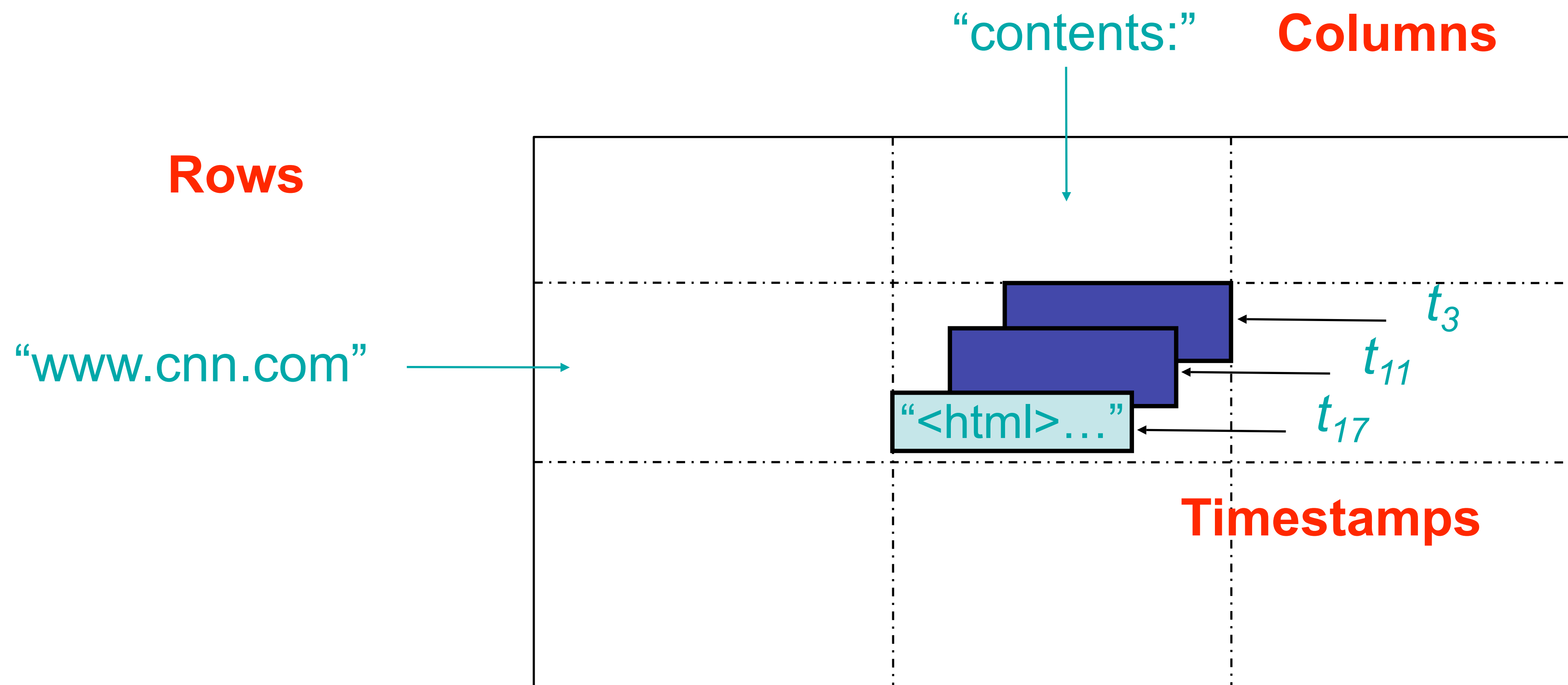


- Rows are ordered lexicographically
- Good match for most of our applications

Basic Data Model

- Distributed multi-dimensional sparse map

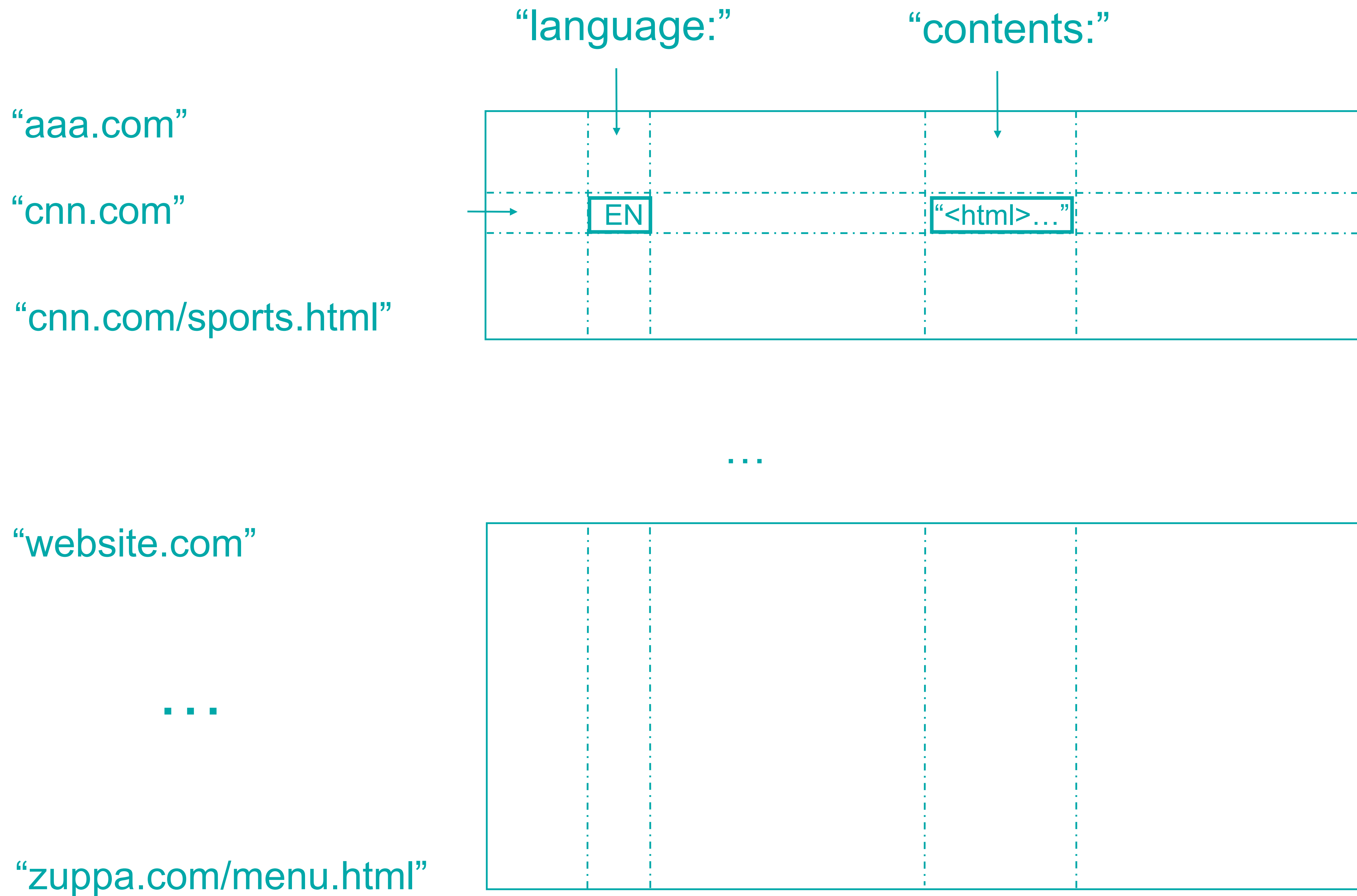
  (row, column, timestamp) → cell contents



- Rows are ordered lexicographically
- Good match for most of our applications

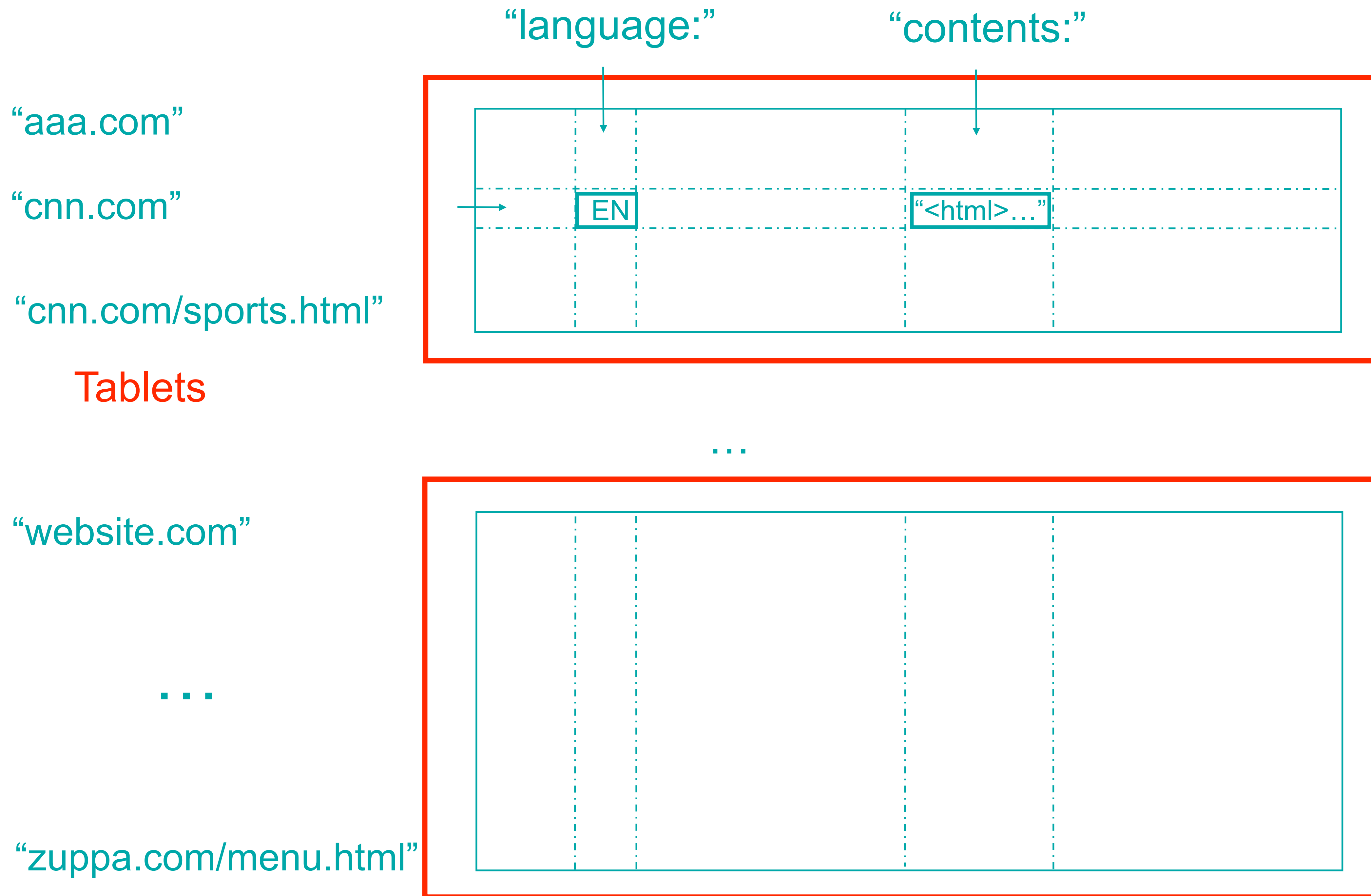
Google

Tablets & Splitting



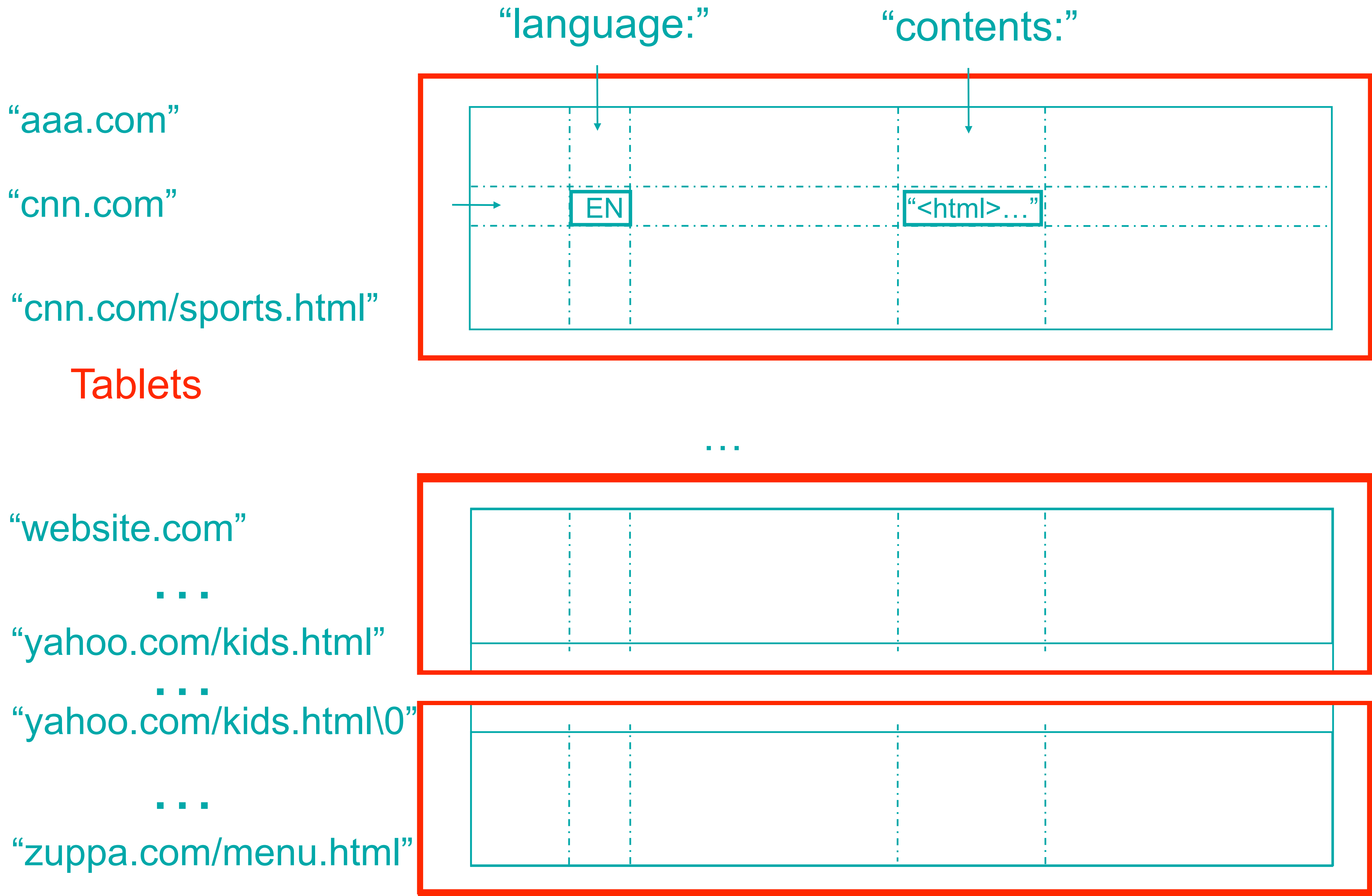
Google

Tablets & Splitting



Google

Tablets & Splitting



Google

BigTable System Structure

Bigtable Cell

Bigtable master

Bigtable tablet server

Bigtable tablet server

…

Bigtable tablet server

Google

BigTable System Structure

Bigtable Cell

Bigtable master

performs metadata ops +
load balancing

Bigtable tablet server

Bigtable tablet server

…

Bigtable tablet server

Google

BigTable System Structure

Bigtable Cell

Bigtable master

performs metadata ops +
load balancing

Bigtable tablet server

serves data

Bigtable tablet server

serves data

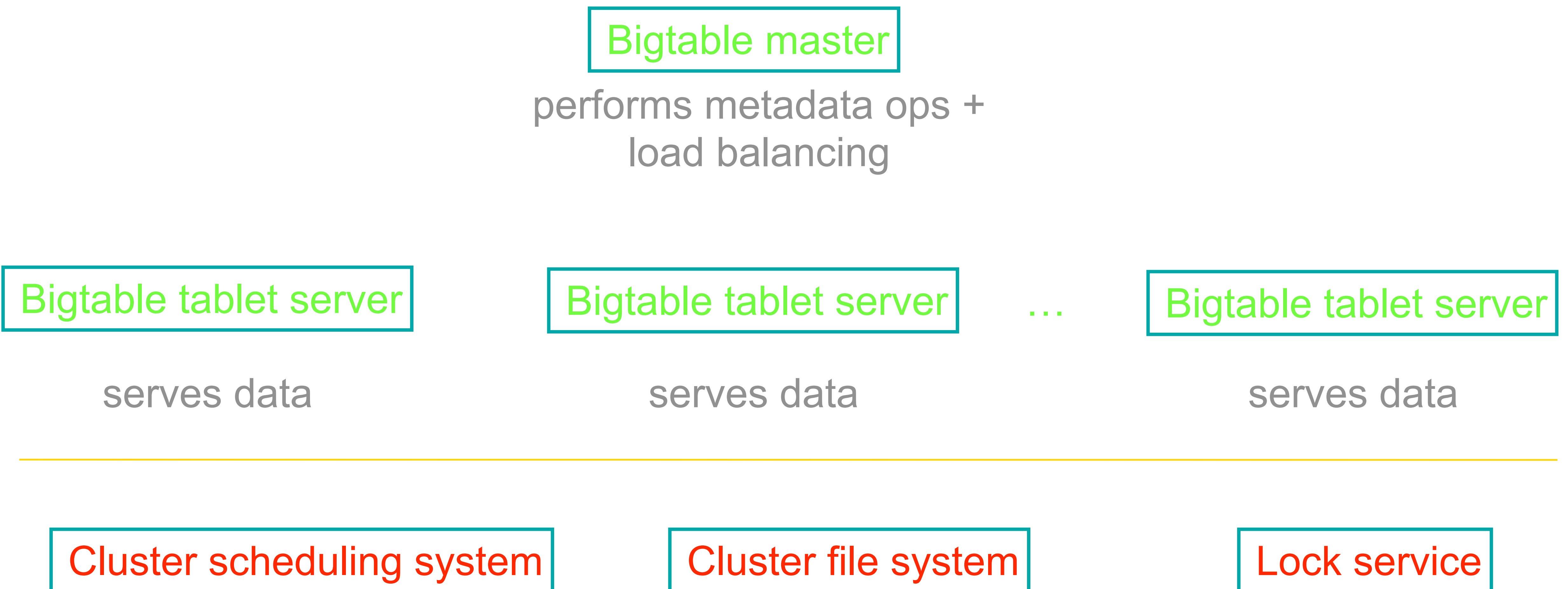
…

Bigtable tablet server

serves data

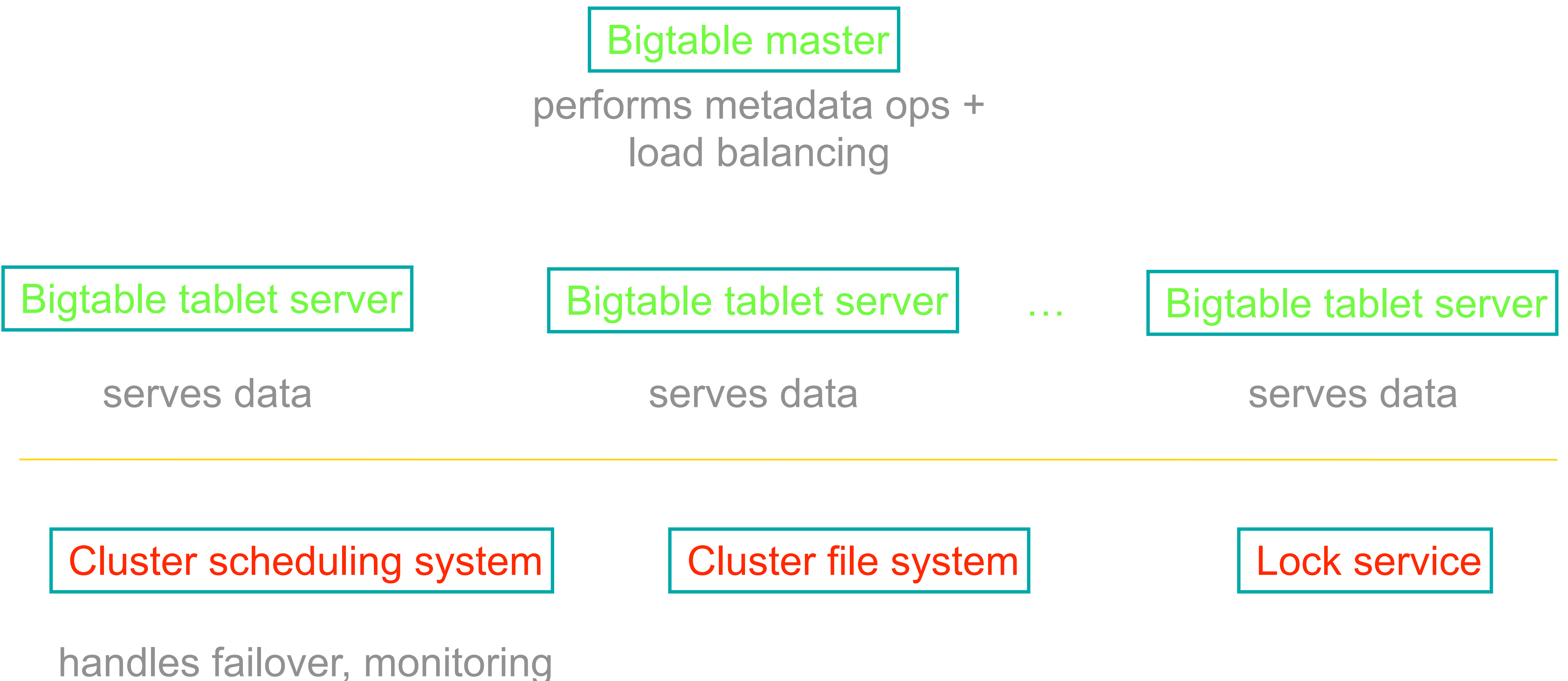
Google

BigTable System Structure



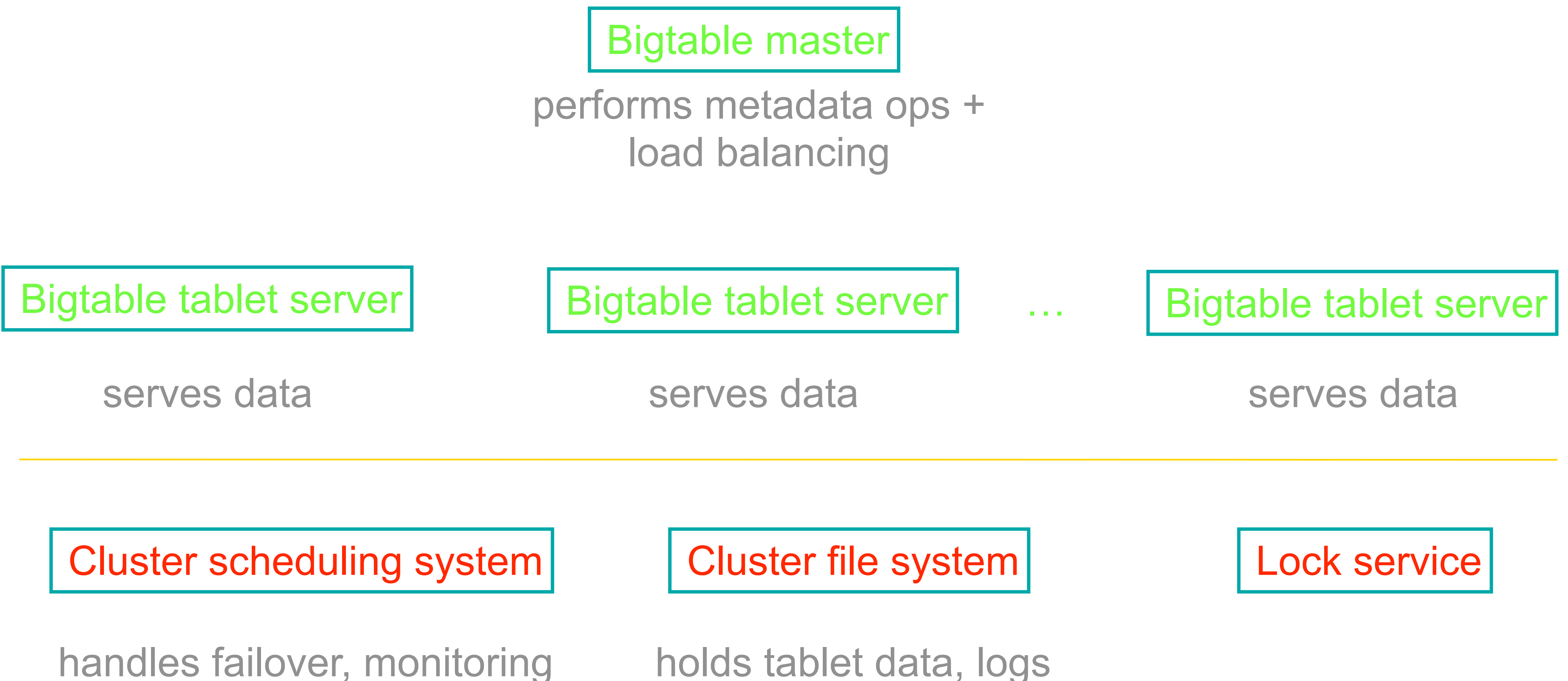
Google

BigTable System Structure

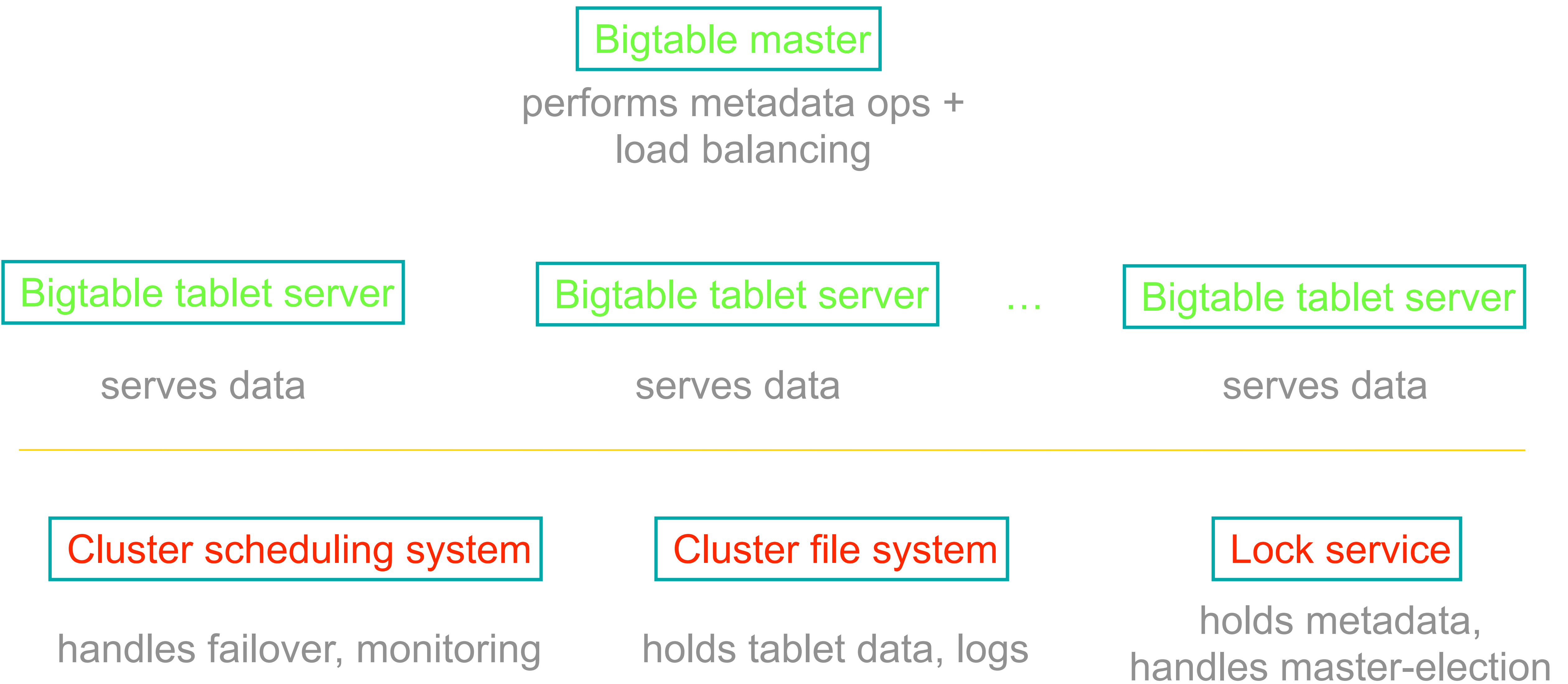


Google

BigTable System Structure



BigTable System Structure



Google

BigTable System Structure

Bigtable Cell

Bigtable client

Bigtable client library

Bigtable master

performs metadata ops + load balancing

Bigtable tablet server

serves data

Bigtable tablet server

serves data

…

Bigtable tablet server

serves data

Cluster scheduling system

handles failover, monitoring

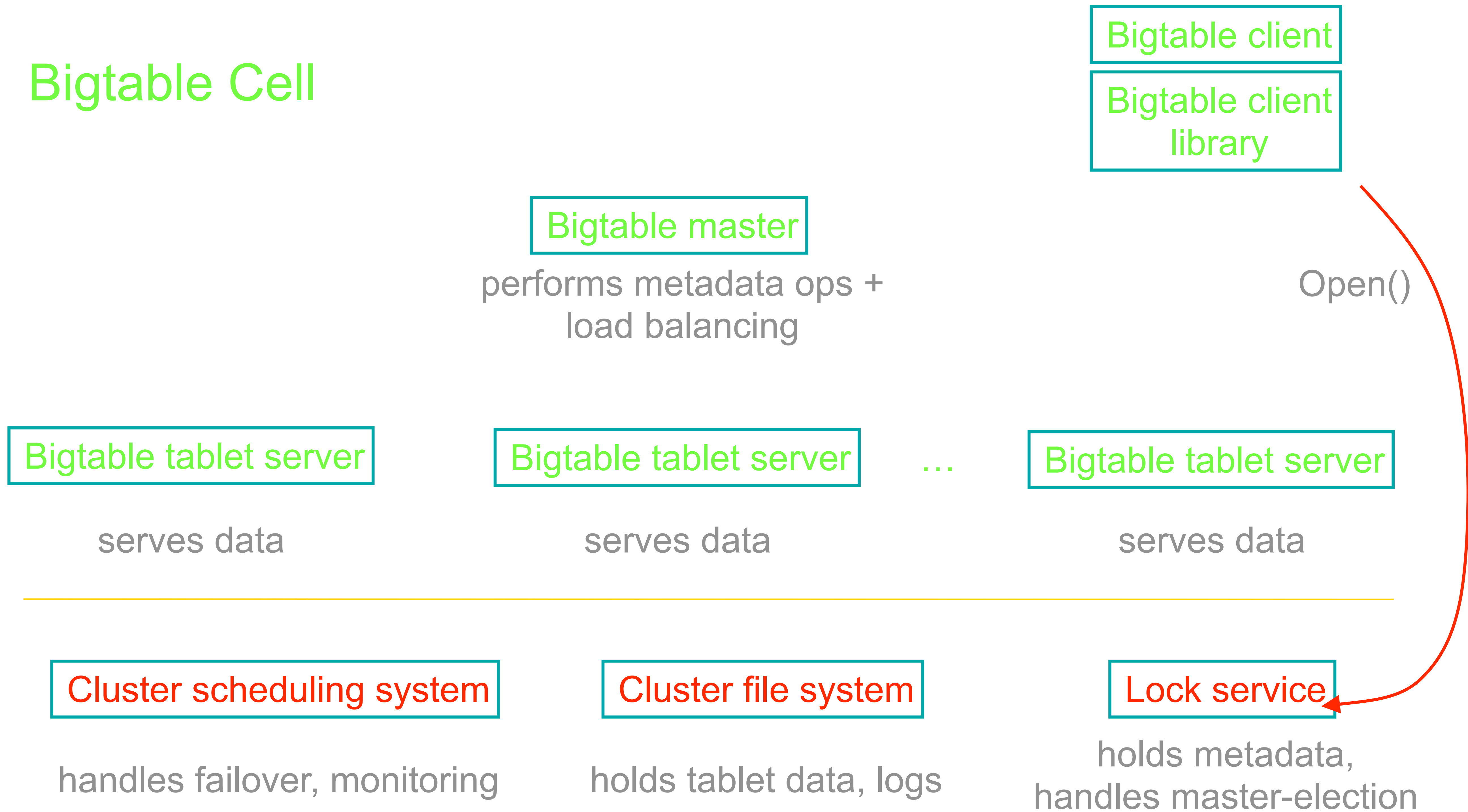
Cluster file system

holds tablet data, logs

Lock service

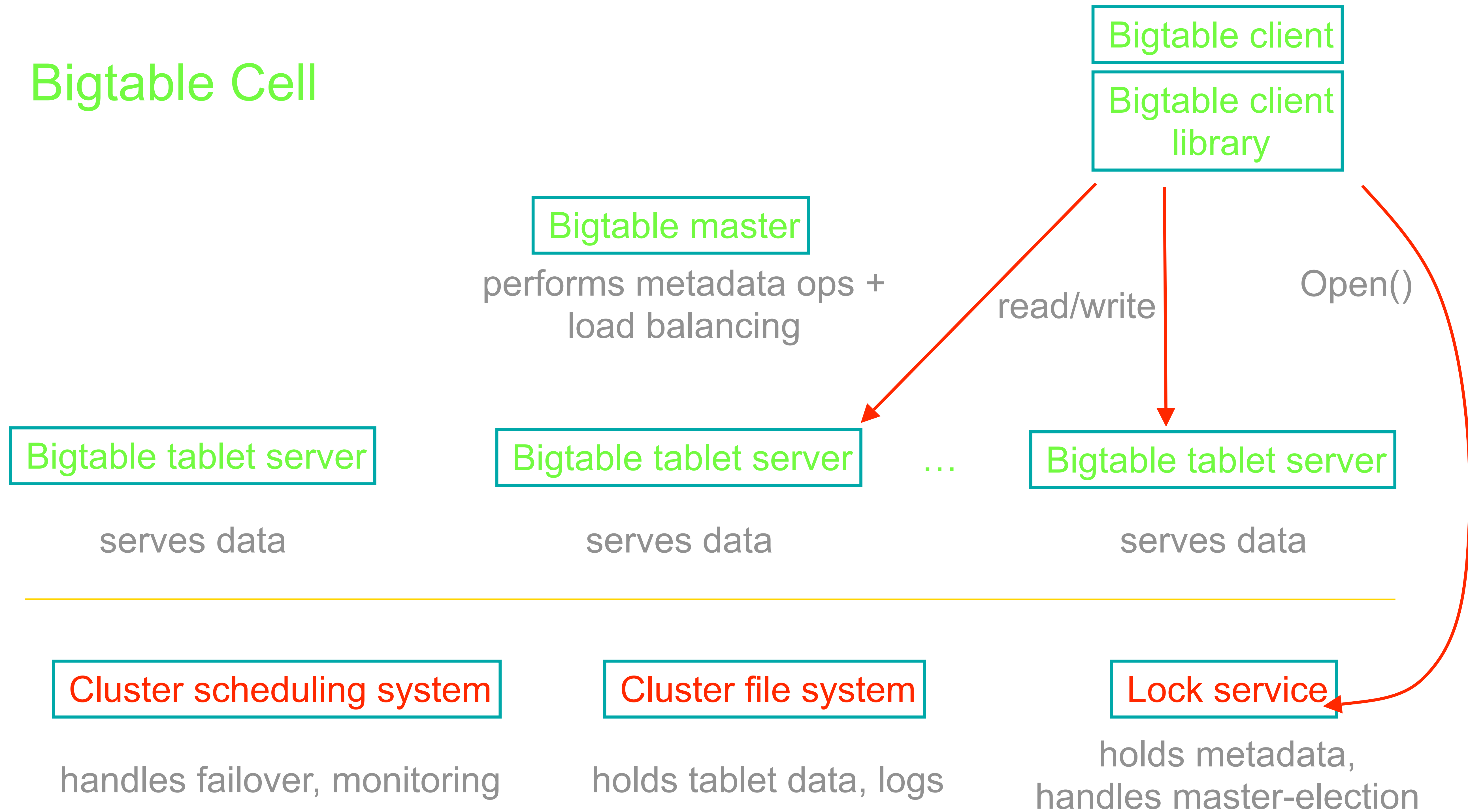
holds metadata, handles master-election

BigTable System Structure



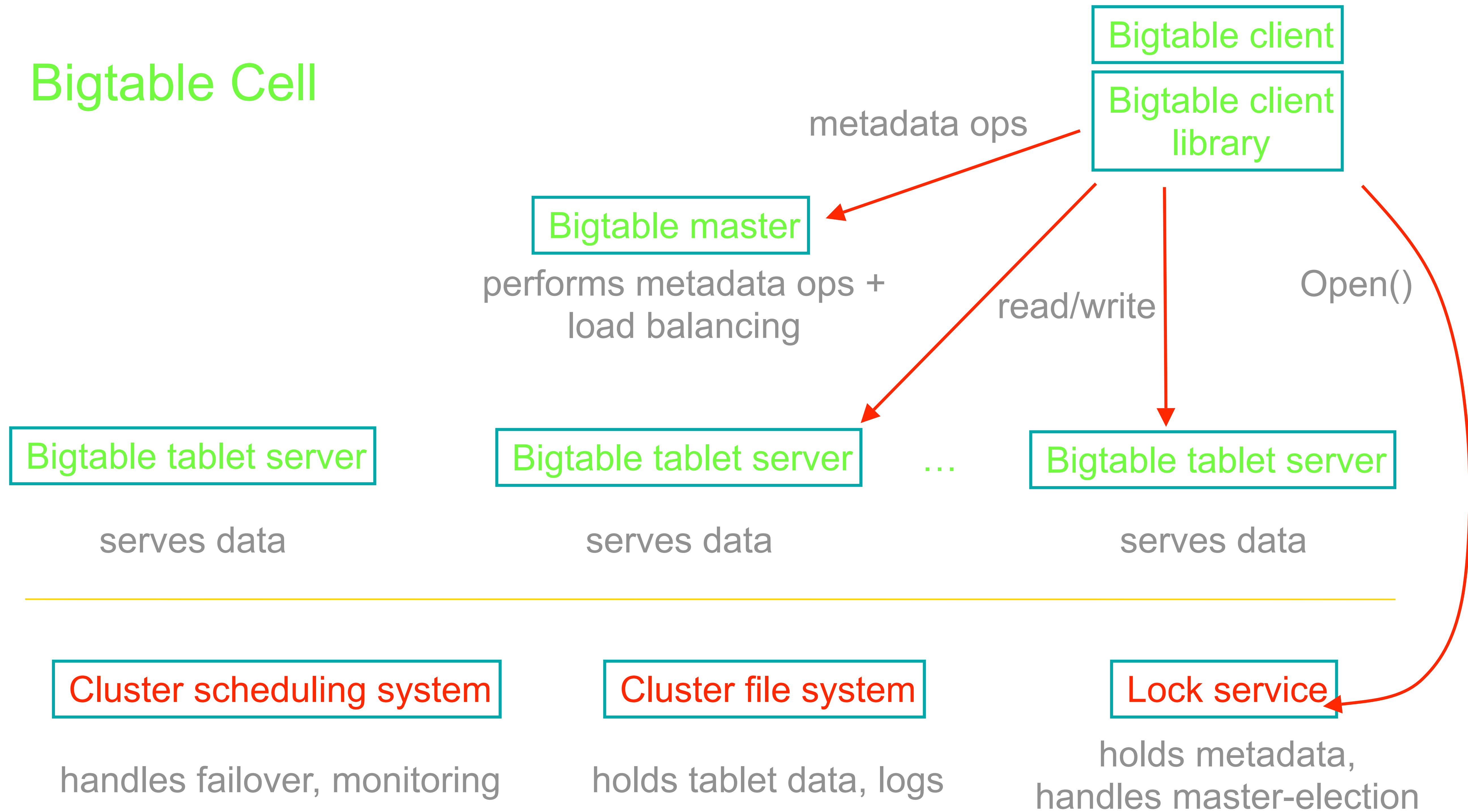
Google

BigTable System Structure



Google

BigTable System Structure



Google

BigTable Status

- Production use for 100+ projects:
  - Crawling/indexing pipeline
  - Google Maps/Google Earth
  - My Search History
  - Google Print
  - Orkut
  - Blogger
  - …
- Currently ~500 BigTable clusters
- Largest cluster:
  - 70+ PB data; sustained: 10M ops/sec; 30+ GB/s I/O

Google

BigTable: What's New Since OSDI'06?

- Lots of work on scaling
- Service clusters, managed by dedicated team
- Improved performance isolation
  - fair-share scheduler within each server, better accounting of memory used per user (caches, etc.)
  - can partition servers within a cluster for different users or tables
- Improved protection against corruption
  - many small changes
  - e.g. immediately read results of every compaction, compare with CRC.
    - Catches ~1 corruption/5.4 PB of data compacted

Google

BigTable Replication (New Since OSDI'06)

- Configured on a per-table basis
- Typically used to replicate data to multiple bigtable clusters in different data centers
- *Eventual consistency model*: writes to table in one cluster eventually appear in all configured replicas
- Nearly all user-facing production uses of BigTable use replication

Google

BigTable Coprocessors (New Since OSDI'06)

- Arbitrary code that runs run next to each tablet in table
  - as tablets split and move, coprocessor code automatically splits/moves too
- High-level call interface for clients
  - Unlike RPC, calls addressed to rows or ranges of rows
    - coprocessor client library resolves to actual locations
  - Calls across multiple rows automatically split into multiple parallelized RPCs
- Very flexible model for building distributed services
  - automatic scaling, load balancing, request routing for apps

Google

Example Coprocessor Uses

- Scalable filesystem metadata management for Colossus (next gen GFS-like file system)
- Distributed language model serving for machine translation system
- Distributed query processing for full-text indexing support
- Regular expression search support for code repository
- ...

Google

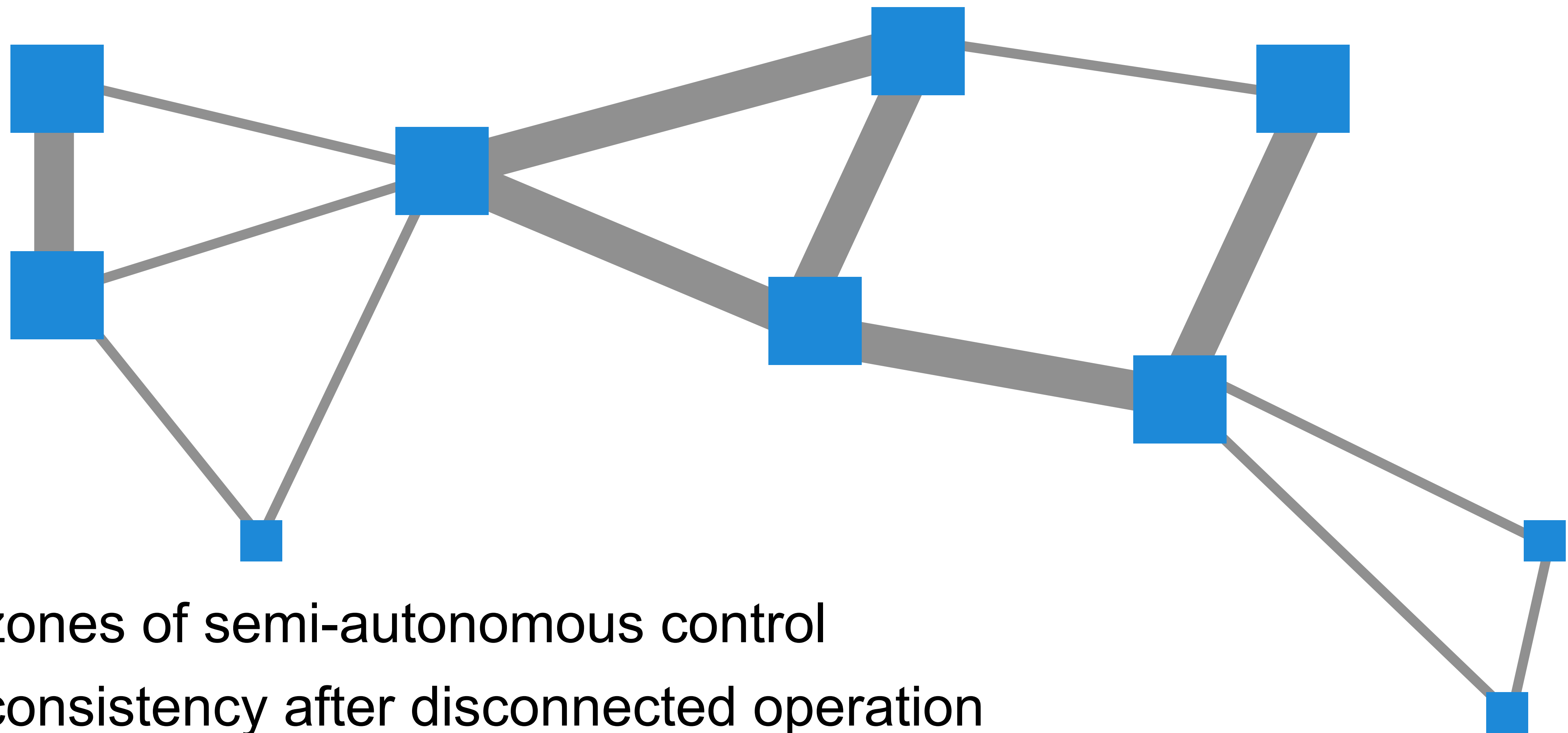
Current Work: Spanner

- Storage & computation system that runs across many datacenters
  - single global namespace
    - Names are independent of location(s) of data
    - Similarities to Bigtable: tables, families, locality groups, coprocessors, ...
    - Differences: directories instead of rows, fine-grained replication configurations
  - support mix of strong and weak consistency across datacenters
    - Strong consistency implemented with Paxos across tablet replicas
    - Full support for distributed transactions across directories/machines
  - much more automated operation
    - automatically changes replication based on constraints and usage patterns
    - automated allocation of resources across entire fleet of machines

Google

Design Goals for Spanner

- Future scale: ~10^5 to 10^7 machines, ~10^{13} directories, ~10^{18} bytes of storage, spread at 100s to 1000s of locations around the world, ~10^9 client machines



  - zones of semi-autonomous control
  - consistency after disconnected operation
  - users specify high-level desires:

    "99%ile latency for accessing this data should be <50ms"

    "Store this data on at least 2 disks in EU, 2 in U.S. & 1 in Asia"

System Building Experiences and Design Patterns

- Experiences from building a variety of systems

- Not all-encompassing, but a collection of patterns have cropped up several times across different systems

Google

Many Internal Services

- Break large complex systems down into many services!
- Simpler from a software engineering standpoint
  - few dependencies, clearly specified
  - easy to test and deploy new versions of individual services
  - ability to run lots of experiments
  - easy to reimplement service without affecting clients
- Development cycles largely decoupled
  - lots of benefits: small teams can work independently
  - easier to have many engineering offices around the world
- e.g. `google.com` search touches 100s of services
  - ads, web search, books, news, spelling correction, ...

Protocol Description Language is a Must

- Desires:
  - extensible
  - efficient
  - compact
  - easy-to-use
  - cross-language
  - self-describing

Google

Protocol Buffers

Our solution: Protocol Buffers (in active use since 2000)

```
message SearchResult {
  required int32 estimated_results = 1;  // (1 is the tag number)
  optional string error_message = 2;
  repeated group Result = 3 {
    required float score = 4;
    required fixed64 docid = 5;
    optional message<WebResultDetails> details = 6;
    …
  }
};
```

- Automatically generated wrappers: C++, Java, Python, ...
- Graceful client and server upgrades
  - servers ignore tags they don't understand, but pass the information through (no need to upgrade intermediate servers)

Google

Protocol Buffers (cont)

- Serialization/deserialization
  - high performance (200+ MB/s encode/decode)
  - fairly compact (uses variable length encodings, binary format)
  - format used to store data persistently (not just for RPCs)
- Also allow service specifications:

```
service Search {
  rpc DoSearch(SearchRequest) returns (SearchResponse);
  rpc Ping(EmptyMessage) returns (EmptyMessage) {
    protocol=udp; };
};
```

- Open source version: http://code.google.com/p/protobuf/

Google

Designing Efficient Systems

Given a basic problem definition, how do you choose "best" solution?

- Best could be simplest, highest performance, easiest to extend, etc.

Important skill: ability to estimate performance of a system design

- – without actually having to build it!

Google

Numbers Everyone Should Know

Operation	Time
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K w/cheap compression algorithm	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Google

Back of the Envelope Calculations

How long to generate image results page (30 thumbnails)?

Design 1: Read serially, thumbnail 256K images on the fly

```
30 seeks * 10 ms/seek + 30 * 256K / 30 MB/s = 560 ms
```

Google

Back of the Envelope Calculations

How long to generate image results page (30 thumbnails)?

Design 1: Read serially, thumbnail 256K images on the fly

```
30 seeks * 10 ms/seek + 30 * 256K / 30 MB/s = 560 ms
```

Design 2: Issue reads in parallel:

```
10 ms/seek + 256K read / 30 MB/s = 18 ms
```

(Ignores variance, so really more like 30-60 ms, probably)

Google

Back of the Envelope Calculations

How long to generate image results page (30 thumbnails)?

Design 1: Read serially, thumbnail 256K images on the fly

```
30 seeks * 10 ms/seek + 30 * 256K / 30 MB/s = 560 ms
```

Design 2: Issue reads in parallel:

```
10 ms/seek + 256K read / 30 MB/s = 18 ms
```

(Ignores variance, so really more like 30-60 ms, probably)

Lots of variations:

- caching (single images? whole sets of thumbnails?)
- pre-computing thumbnails
- …

Back of the envelope helps identify most promising…

Google

Know Your Basic Building Blocks

Core language libraries, basic data structures, protocol buffers, GFS, BigTable, indexing systems, MapReduce, …

Not just their interfaces, but understand their implementations (at least at a high level)

If you don't know what's going on, you can't do decent back-of-the-envelope calculations!

Google

Know Your Basic Building Blocks

Core language libraries, basic data structures, protocol buffers, GFS, BigTable, indexing systems, MapReduce, …

Not just their interfaces, but understand their implementations (at least at a high level)

If you don't know what's going on, you can't do decent back-of-the-envelope calculations!

- Corollary: implementations with unpredictable 1000X variations in performance are not very helpful if latency or throughput matters
  - e.g. VM paging

Google

Designing & Building Infrastructure

Identify common problems, and build software systems to address them in a general way

- Important not to try to be all things to all people
  - Clients might be demanding 8 different things
  - Doing 6 of them is easy
  - …handling 7 of them requires real thought
  - …dealing with all 8 usually results in a worse system
    - more complex, compromises other clients in trying to satisfy everyone

Don't build infrastructure just for its own sake

- Identify common needs and address them
- Don't imagine unlikely potential needs that aren't really there
- Best approach: use your own infrastructure (especially at first!)
  - (much more rapid feedback about what works, what doesn't)

Design for Growth

Try to anticipate how requirements will evolve

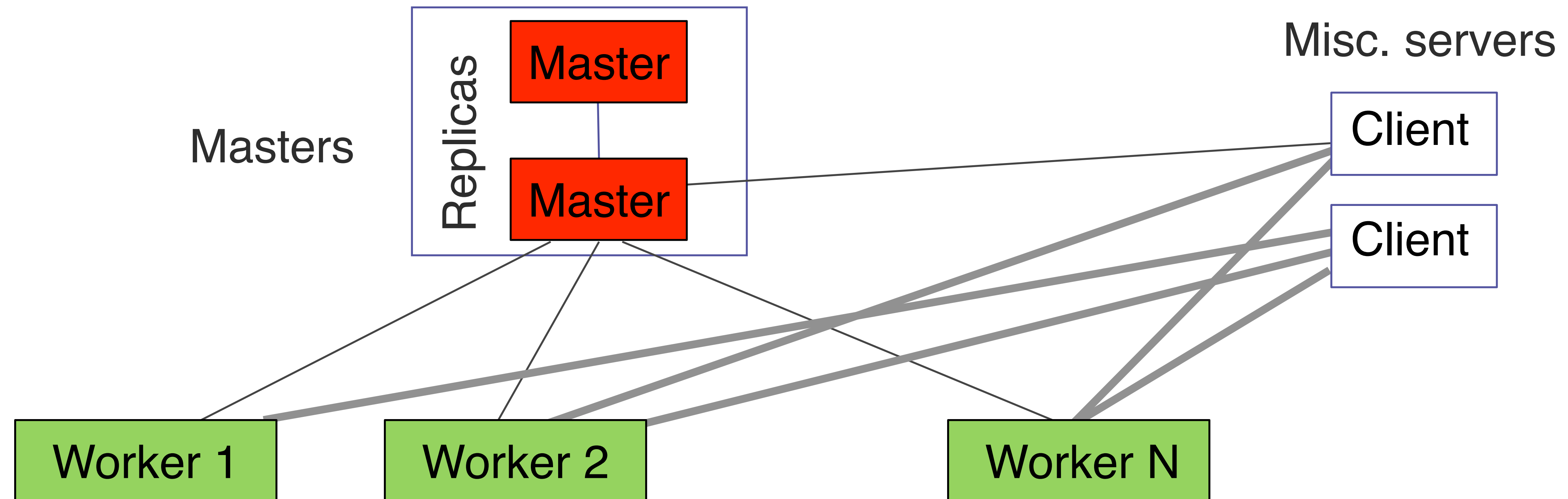
- keep likely features in mind as you design base system

Don’t design to scale infinitely:

- ~5X - 50X growth good to consider
- >100X probably requires rethink and rewrite

Pattern: Single Master, 1000s of Workers

- Master orchestrates global operation of system
  - load balancing, assignment of work, reassignment when machines fail, etc.
  - ... but client interaction with master is fairly minimal



- Examples:
  - GFS, BigTable, MapReduce, Transfer Service, cluster scheduling system, ...

Pattern: Single Master, 1000s of Workers (cont)

- Often: hot standby of master waiting to take over
- Always: bulk of data transfer directly between clients and workers

- Pro:
  - simpler to reason about state of system with centralized master

- Caveats:
  - careful design required to keep master out of common case ops
  - scales to 1000s of workers, but not 100,000s of workers

Pattern: Canary Requests

- Problem: odd requests sometimes cause server process to crash
  - testing can help reduce probability, but can't eliminate

- If sending same or similar request to 1000s of machines:
  - they all might crash!
  - recovery time for 1000s of processes pretty slow

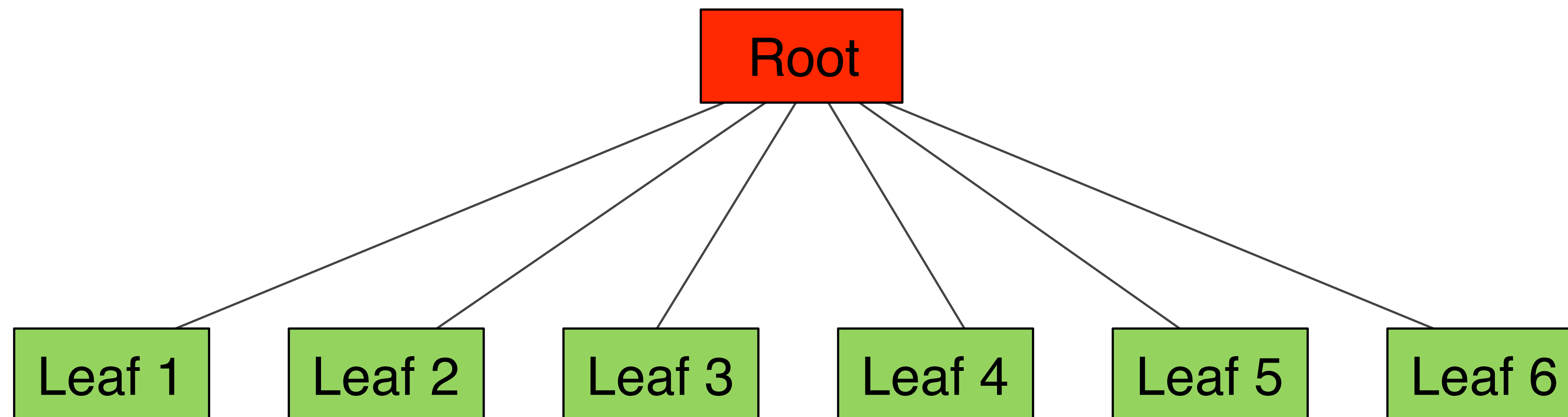
- Solution: send canary request first to one machine
  - if RPC finishes successfully, go ahead and send to all the rest
  - if RPC fails unexpectedly, try another machine
    (might have just been coincidence)
  - if fails *K* times, reject request

- Crash only a few servers, not 1000s

Google™

Pattern: Tree Distribution of Requests

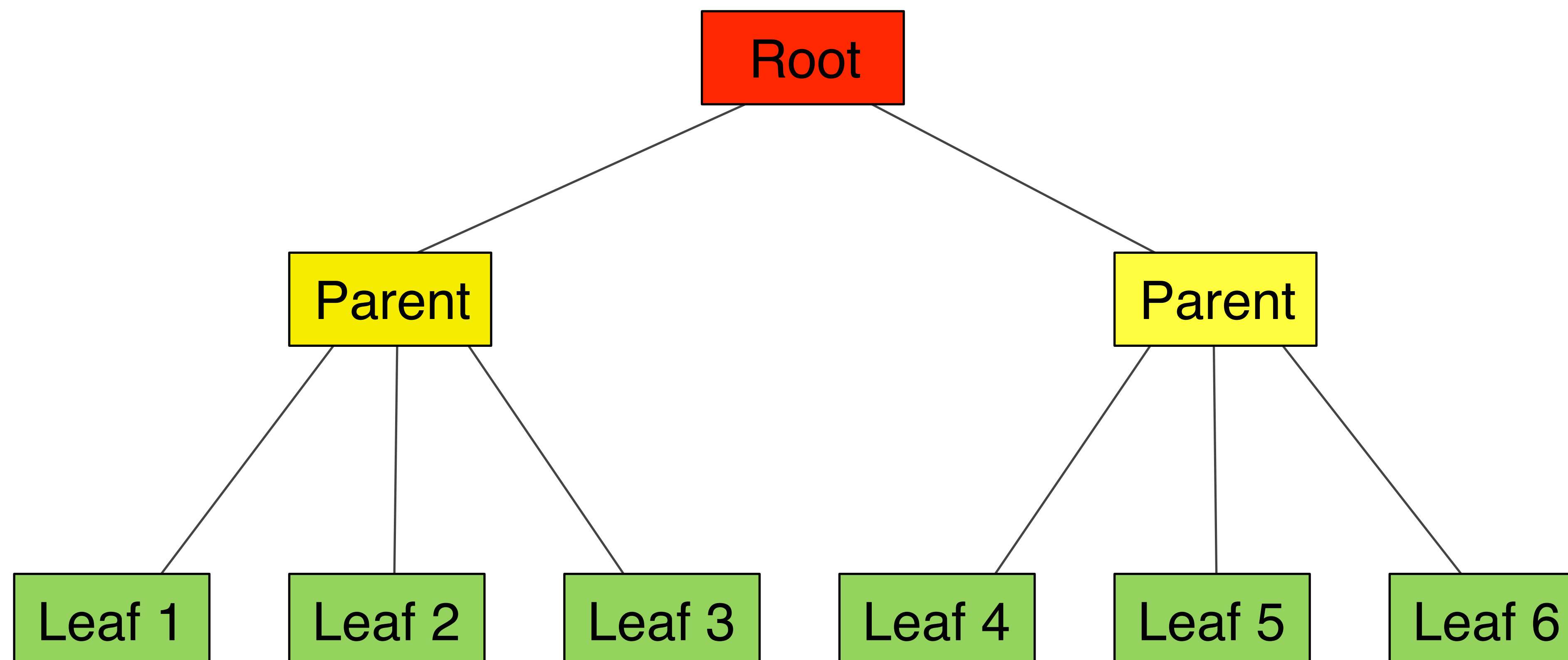
- Problem: Single machine sending 1000s of RPCs overloads NIC on machine when handling replies
  - wide fan in causes TCP drops/retransmits, significant latency
  - CPU becomes bottleneck on single machine



Google

Pattern: Tree Distribution of Requests

- Solution: Use tree distribution of requests/responses
  – fan in at root is smaller
  – cost of processing leaf responses spread across many parents
- Most effective when parent processing can trim/combine leaf data
  – can also co-locate parents on same rack as leaves



Google

Pattern: Backup Requests to Minimize Latency

- Problem: variance high when requests go to 1000s of machines
  - last few machines stretch out latency tail

- Often, multiple replicas can handle same kind of request
- When few tasks remaining, send backup requests to other replicas
- Whichever duplicate request finishes first wins
  - useful when variance is unrelated to specifics of request
  - increases overall load by a tiny percentage
  - decreases latency tail significantly

Pattern: Backup Requests to Minimize Latency

- Problem: variance high when requests go to 1000s of machines
  - last few machines stretch out latency tail

- Often, multiple replicas can handle same kind of request
- When few tasks remaining, send backup requests to other replicas
- Whichever duplicate request finishes first wins
  - useful when variance is unrelated to specifics of request
  - increases overall load by a tiny percentage
  - decreases latency tail significantly

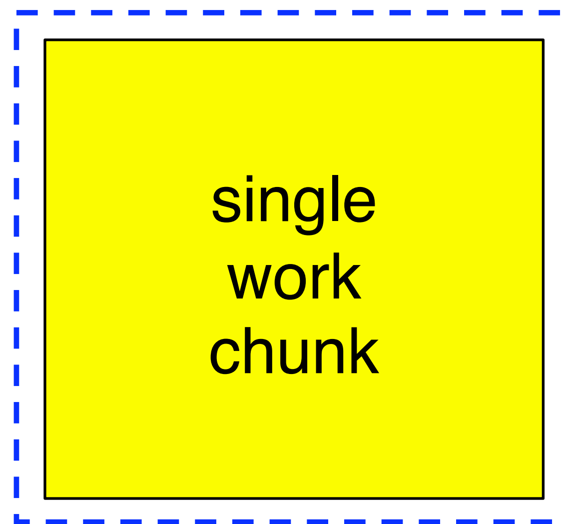
- Examples:
  - MapReduce backup tasks (granularity: many seconds)
  - various query serving systems (granularity: milliseconds)

Google™

Pattern: Multiple Smaller Units per Machine

- Problems:
  - want to minimize recovery time when machine crashes
  - want to do fine-grained load balancing

- Having each machine manage 1 unit of work is inflexible
  - slow recovery: new replica must recover data that is O(machine state) in size
  - load balancing much harder

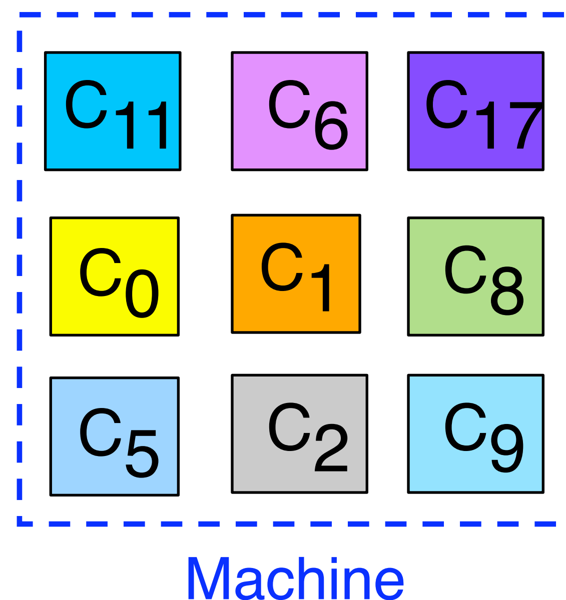


Google

Pattern: Multiple Smaller Units per Machine

- Have each machine manage many smaller units of work/data
  - typical: ~10-100 units/machine
  - allows fine grained load balancing (shed or add one unit)
  - fast recovery from failure (N machines each pick up 1 unit)

- Examples:
  - map and reduce tasks, GFS chunks, Bigtable tablets, query serving system index shards



Google

Pattern: Range Distribution of Data, not Hash

- Problem: manage growing set of keys/values in distributed system
  - need to spread data out across K machines
  - need to adapt to K+1 machines as data grows
- Consistent hashing of keys to map to machines:
  - Pro: gives nice even distribution of data across machines
  - Con: hard for users to generate or understand locality across multiple different keys

Google™

Pattern: Range Distribution of Data, not Hash

- Problem: manage growing set of keys/values in distributed system
  - need to spread data out across K machines
  - need to adapt to K+1 machines as data grows
- Consistent hashing of keys to map to machines:
  - Pro: gives nice even distribution of data across machines
  - Con: hard for users to generate or understand locality across multiple different keys
- Range distribution: break space of keys into multiple ranges
- Machines manage a small number of different ranges at a time
  - e.g. Bigtable tablets
  - Con: harder to implement than hashing (string ranges, rather than simple numeric hashes, location metadata is larger)
  - Pro: users can reason about & control locality across keys

Google™

Pattern: Elastic Systems

- Problem: Planning for exact peak load is hard
  - overcapacity: wasted resources
  - undercapacity: meltdown

- Design system to adapt:
  - automatically shrink capacity during idle period
  - automatically grow capacity as load grows

- Make system resilient to overload:
  - do something reasonable even up to 2X planned capacity
    - e.g. shrink size of index searched, back off to less CPU intensive algorithms, drop spelling correction tip, etc.
  - more aggressive load balancing when imbalance more severe

Pattern: One Interface, Multiple Implementations

- Example: Google web search system wants all of these:
  - freshness (update documents in ~1 second)
  - massive capacity (10000s of requests per second)
  - high quality retrieval (lots of information about each document)
  - massive size (billions of documents)

- Very difficult to accomplish in single implementation

- Partition problem into several subproblems with different engineering tradeoffs. E.g.
  - realtime system: few docs, ok to pay lots of $$$/doc
  - base system: high # of docs, optimized for low $/doc
  - realtime+base: high # of docs, fresh, low $/doc

Add Sufficient Monitoring/Status/Debugging Hooks

All our servers:

- Export HTML-based status pages for easy diagnosis
- Export a collection of key-value pairs via a standard interface
  - monitoring systems periodically collect this from running servers
- RPC subsystem collects sample of all requests, all error requests, all requests >0.0s, >0.05s, >0.1s, >0.5s, >1s, etc.

- Support low-overhead online profiling
  - cpu profiling
  - memory profiling
  - lock contention profiling

If your system is slow or misbehaving, can you figure out why?

Some Interesting Challenges

- A collection of problems germane to building large-scale datacenter services

- Again, not all inclusive..

- Some of these problems have substantial related work, some are relatively new..

Google

Adaptivity in World-Wide Systems

- Challenge: automatic, dynamic world-wide placement of data & computation to minimize latency and/or cost, given constraints on:
  - bandwidth
  - packet loss
  - power
  - resource usage
  - failure modes
  - ...
- Users specify high-level desires:

  "99%ile latency for accessing this data should be <50ms"

  "Store this data on at least 2 disks in EU, 2 in U.S. & 1 in Asia"

  "Store three replicas in three different datacenters less than 20ms apart"

Building Applications on top of Weakly Consistent Storage Systems

- Many applications need state replicated across a wide area
  - For reliability and availability
- Two main choices:
  - consistent operations (e.g. use Paxos)
    - often imposes additional latency for common case
  - inconsistent operations
    - better performance/availability, but apps harder to write and reason about in this model
- Many apps need to use a mix of both of these:
  - e.g. Gmail: marking a message as read is asynchronous, sending a message is a heavier-weight consistent operation

Google

Building Applications on top of Weakly Consistent Storage Systems

- Challenge: General model of consistency choices, explained and codified
  – ideally would have one or more “knobs” controlling performance vs. consistency
  – “knob” would provide easy-to-understand tradeoffs
- Challenge: Easy-to-use abstractions for resolving conflicting updates to multiple versions of a piece of state
  – Useful for reconciling client state with servers after disconnected operation
  – Also useful for reconciling replicated state in different data centers after repairing a network partition

Distributed Systems Abstractions

- High-level tools/languages/abstractions for building distributed systems
  - e.g. For batch processing, MapReduce handles parallelization, load balancing, fault tolerance, I/O scheduling automatically within a simple programming model

- Challenge: Are there unifying abstractions for other kinds of distributed systems problems?
  - e.g. systems for handling interactive requests & dealing with ***intra***-operation parallelism
    - load balancing, fault-tolerance, service location & request distribution, ...
  - e.g. client-side AJAX apps with rich server-side APIs
    - better ways of constructing client-side applications?

Google™

Sharing in Storage & Retrieval Systems

- Storage and retrieval systems with mix of private, semi-private, widely shared and public documents
  - e.g. e-mail vs. shared doc among 10 people vs. messages in group with 100,000 members vs. public web pages
- Challenge: building storage and retrieval systems that efficiently deal with ACLs that vary widely in size
  - best solution for doc shared with 10 people is different than for doc shared with the world
  - sharing patterns of a document might change over time

Google™

Final Thoughts

- Large-scale datacenters + many client devices offer many interesting opportunities
  - planetary scale distributed systems
  - interesting CPU and data intensive services
- Tools and techniques for building such systems are evolving
- Fun and interesting times are ahead of us!

Google™

Thanks! Questions...?

Further reading:

- Ghemawat, Gobioff, & Leung. *Google File System*, SOSP 2003.
- Barroso, Dean, & Hölzle. *Web Search for a Planet: The Google Cluster Architecture*, IEEE Micro, 2003.
- Dean & Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*, OSDI 2004.
- Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, & Gruber. *Bigtable: A Distributed Storage System for Structured Data*, OSDI 2006.
- Burrows. *The Chubby Lock Service for Loosely-Coupled Distributed Systems*. OSDI 2006.
- Pinheiro, Weber, & Barroso. *Failure Trends in a Large Disk Drive Population*. FAST 2007.
- Brants, Popat, Xu, Och, & Dean. *Large Language Models in Machine Translation*, EMNLP 2007.
- Barroso & Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, Morgan & Claypool Synthesis Series on Computer Architecture, 2009.
- Malewicz *et al*. *Pregel: A System for Large-Scale Graph Processing*. PODC, 2009.
- Schroeder, Pinheiro, & Weber. *DRAM Errors in the Wild: A Large-Scale Field Study*. SEGMETRICS'09.
- Protocol Buffers. http://code.google.com/p/protobuf/

These and many more available at: http://labs.google.com/papers.html

Google