CS5412: OTHER DATA CENTER SERVICES

Lecture V

Ken Birman

Tier two and Inner Tiers

- If tier one faces the user and constructs responses, what lives in tier two?
  - Caching services are very common (many flavors)
  - Other kinds of rapidly responsive lightweight services that are massively scaled
- Inner tier services might still have "online" roles, but tend to live on smaller numbers of nodes: maybe tens rather than hundreds or thousands
  - Tiers one and two soak up the load
  - This reduces load on the inner tiers
  - Many inner services accept asynchronous streams of events

Contrast with “Back office”

- A term often used for services and systems that don’t play online roles
  - In some sense the whole cloud has an outward facing side, handling users in real-time, and an inward side, doing “offline” tasks
  - Still can have immense numbers of nodes involved but the programming model has more of a batch feel to it
- For example, MapReduce (Hadoop)

Some interesting services we'll consider

- Memcached: In-memory caching subsystem
- Dynamo: Amazon's shopping cart
- BigTable: A "sparse table" for structured data
- GFS: Google File System
- Chubby: Google's locking service
- Zookeeper: File system with locking, strong semantics
- Sinfonia: A flexible append-only logging service
- MapReduce: "Functional" computing for big datasets

Connection to DHT concept

- Last time we focused on a P2P style of DHT
- These services are mostly built as layers *over* a data center DHT deployment
  - Same idea and similar low-level functionality
  - But inside the data center we can avoid costly indirect routing. We'll discuss that next time.

Memcached

- Very simple concept:
  - High performance distributed in-memory caching service that manages “objects”
  - Key-value API has become an accepted standard
  - Many implementations

- Simplest versions: just a library that manages a list or a dictionary
- Fanciest versions: distributed services implemented using a cluster of machines

Memcached API

- Memcached defines a standard API
  - Defines the calls the application can issue to the library or the server (either way, it looks like library)
  - In theory, this means an application can be coded and tested using one version of memcached, then migrated to a different one

```
function get_foo(foo_id)
        foo = memcached_get("foo:" . foo_id)
        if foo != null return foo
        foo = fetch_foo_from_database(foo_id)
        memcached_set("foo:" .  foo_id, foo)
        return foo end
```

A single memcached server is easy

- Today’s tools make it trivial to build a server
  - Build a program
  - Designate some of its methods as ones that expose service APIs
  - Tools will create stubs: library procedures that automate binding to the service
  - Now run your service at a suitable place and register it in the local registry
- Applications can do remote procedure calls, and these code paths are heavily optimized: quite fast

How do they build clusters?

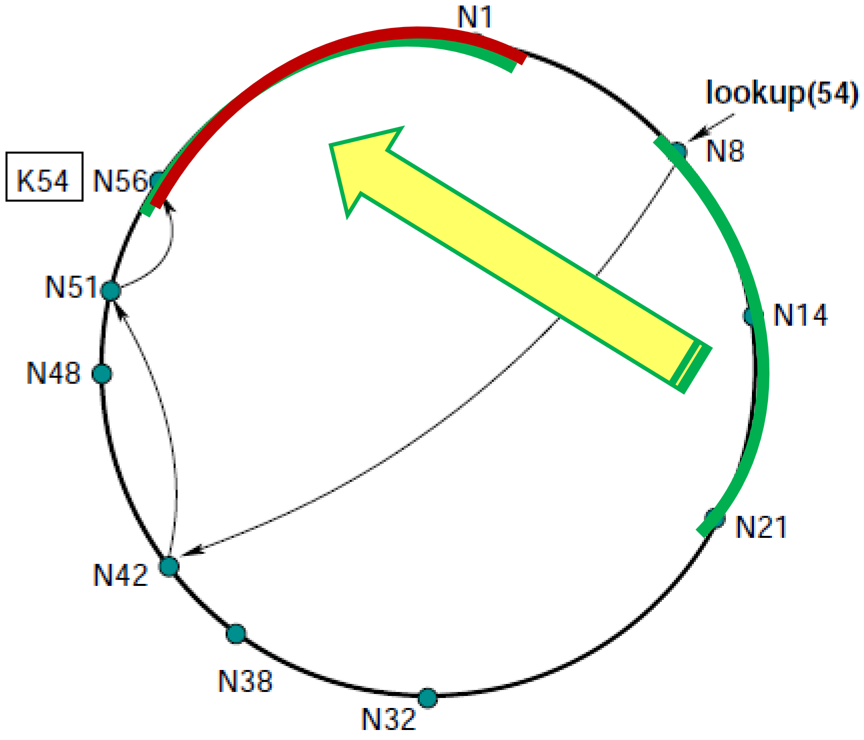
- Much trickier challenge!
  - Trivial approach just hashes the memcached key to decide which server to send data to
  - But this could lead to load imbalances, plus some objects are probably popular, while others are probably “cold spots”.
    - Would prefer to replicate the hot data to improve capacity
    - But this means we need to track popularity (like Beehive!)
- Solutions to this are being offered as products
- We have it as one of the possible cs5412 projects!

Dynamo

- Amazon's massive collaborative key-value store
- Built over a version of Chord DHT
  - Basic idea is to offer a key-value API, like memcached
  - But now we'll have thousands of service instances
  - Used for shopping cart: a very high-load application
- Basic innovation?
  - To speed things up (think BASE), Dynamo sometimes puts data at the "wrong place"
  - Idea is that if the right nodes can't be reached, put the data *somewhere* in the DHT, then allow repair mechanisms to migrate the information to the right place asynchronously

Dynamo in practice

- Suppose key should map to N56
- Dynamo replicates data on neighboring nodes (N1 here)
- Will also save key,value on subsequent nodes if targets don't respond
- Data migrates to correct location eventually

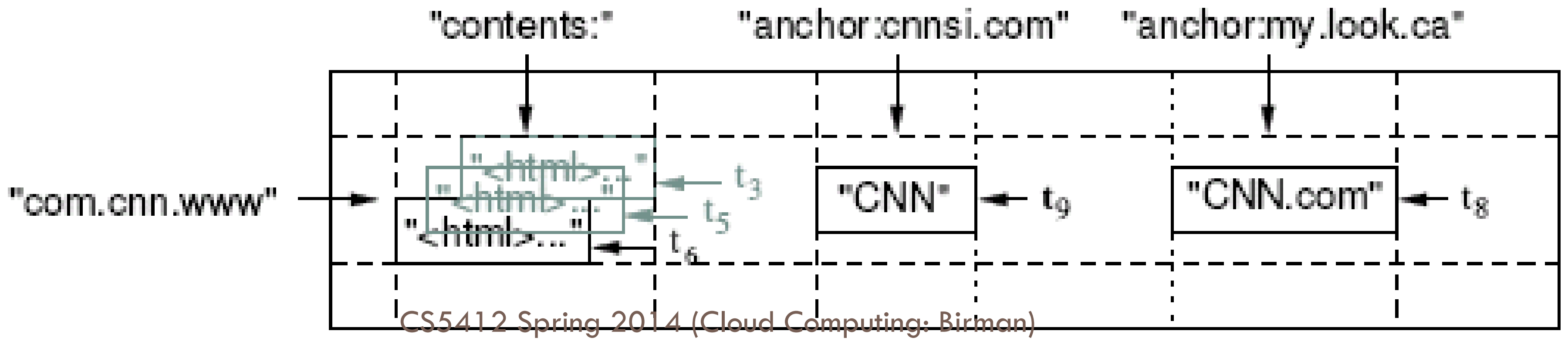


BigTable

- Yet another key-value store!
- Built by Google over their GFS file system and Chubby lock service
- Idea is to create a flexible kind of table that can be expanded as needed dynamically
- Slides from a talk the developers gave on it

Data model: a big map

- <Row, Column, Timestamp> triple for key Arbitrary "columns" on a row-by-row basis
  - Column family:qualifier. Family is heavyweight, qualifier lightweight
  - Column-oriented physical store- rows are sparse!
- Does not support a relational model
  - No table-wide integrity constraints
  - No multirow transactions



API

- Metadata operations
  - Create/delete tables, column families, change metadata
- Writes (atomic)
  - Set(): write cells in a row
  - DeleteCells(): delete cells in a row
  - DeleteRow(): delete all cells in a row
- Reads
  - Scanner: read arbitrary cells in a bigtable
    - Each row read is atomic
    - Can restrict returned rows to a particular range
    - Can ask for just data from 1 row, all rows, etc.
    - Can ask for all columns, just certain column families, or specific columns

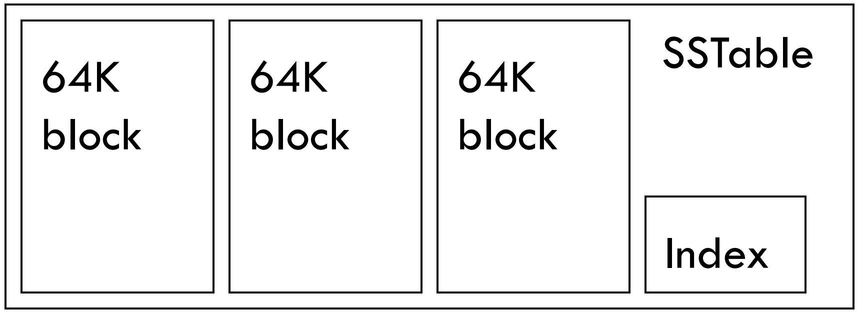
Versions

- Data has associated version numbers
  - To perform a transaction, create a set of pages all using some new version number
  - Then can atomically install them

- For reads can let BigTable select the version or can tell it which one to access

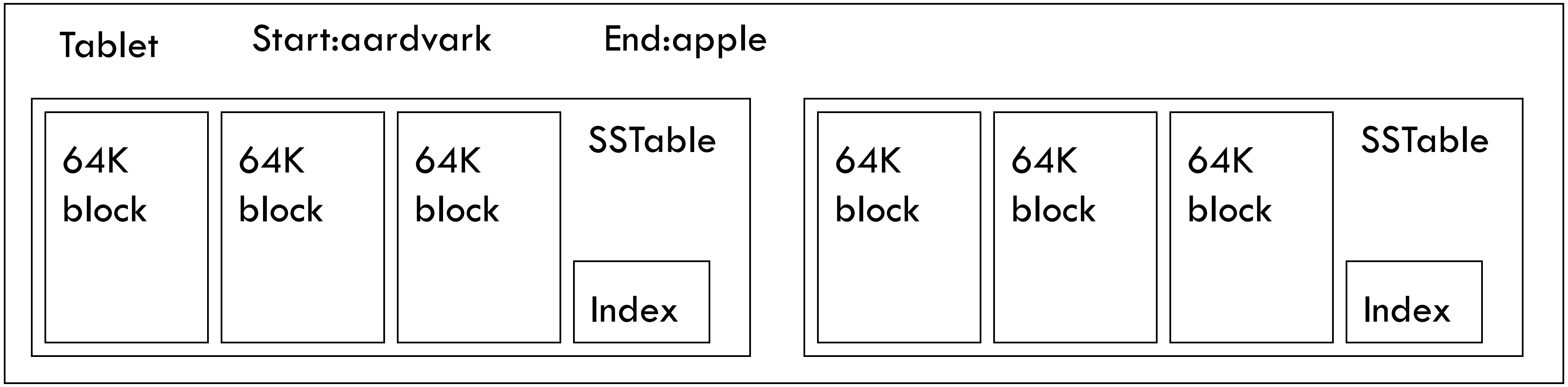
SSTable

- Immutable, sorted file of key-value pairs
- Chunks of data plus an index
  - Index is of block ranges, not values



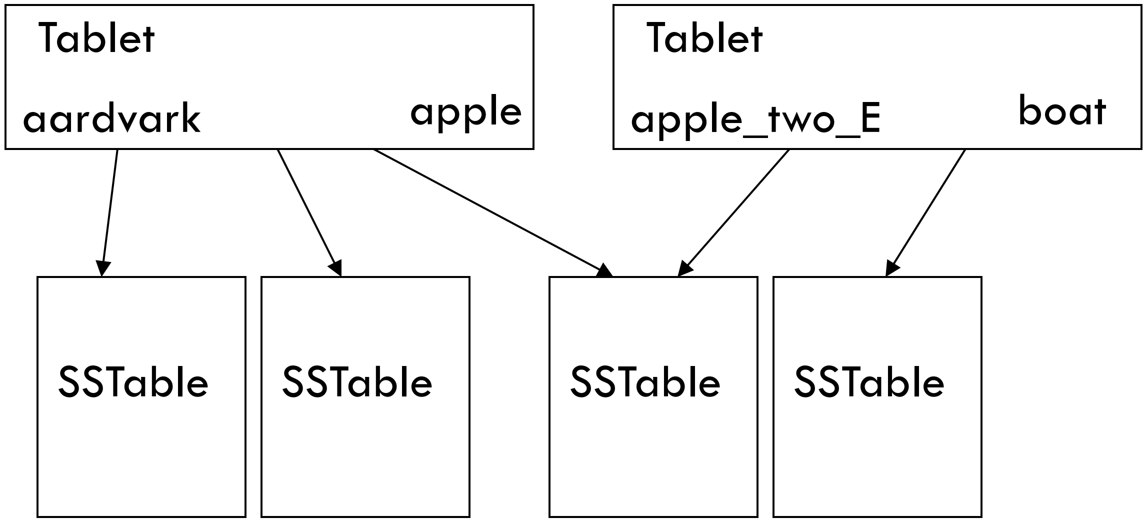
Tablet

- Contains some range of rows of the table
- Built out of multiple SSTables

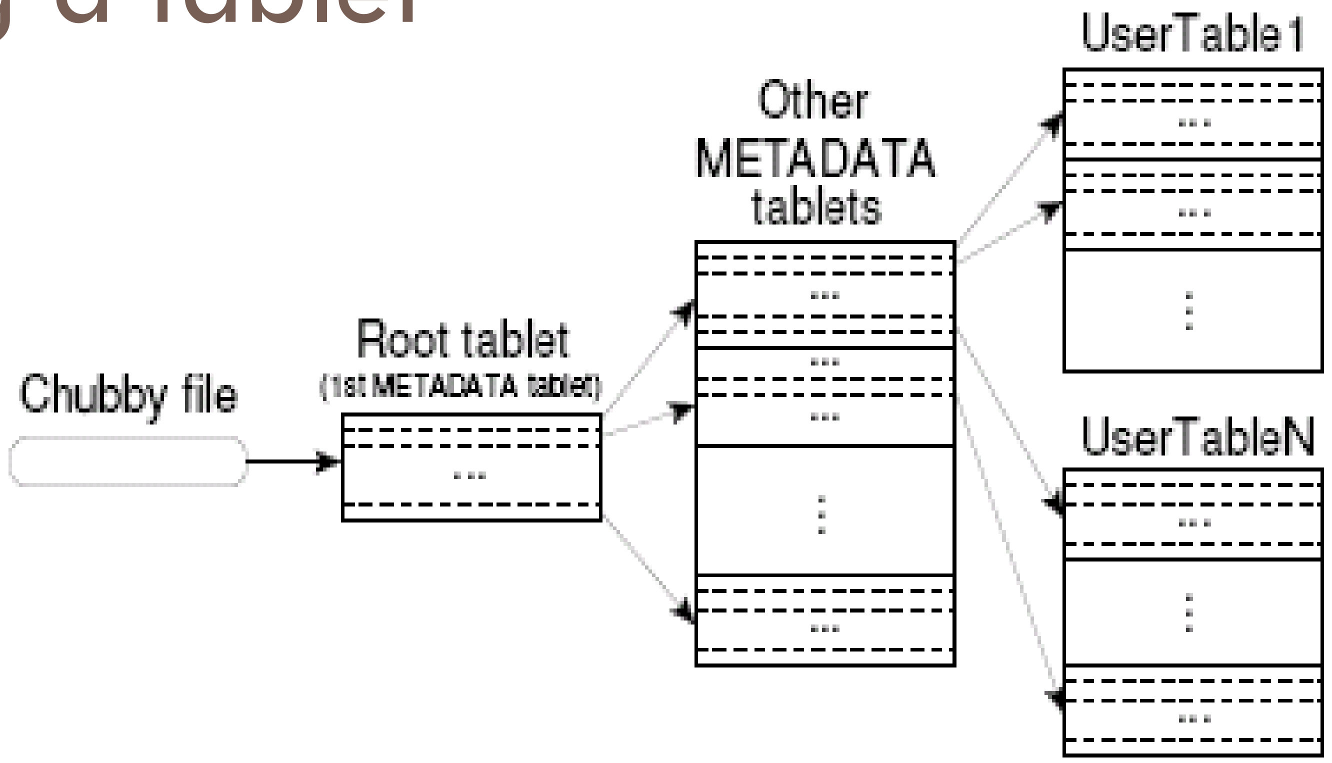


Table

- Multiple tablets make up the table
- SSTables can be shared
- Tablets do not overlap, SSTables can overlap



Finding a tablet



- Stores: Key: table id + end row,   Data: location
- Cached at clients, which may detect data to be incorrect
  - in which case, lookup on hierarchy performed
- Also prefetched (for range queries)

Servers

- Tablet servers manage tablets, multiple tablets per server. Each tablet is 100-200 MB
  - Each tablet lives at only one server
  - Tablet server splits tablets that get too big

- Master responsible for load balancing and fault tolerance

Master's Tasks

- Use Chubby to monitor health of tablet servers, restart failed servers
  - Tablet server registers itself by getting a lock in a specific directory chubby
    - Chubby gives "lease" on lock, must be renewed periodically
    - Server loses lock if it gets disconnected
  - Master monitors this directory to find which servers exist/are alive
    - If server not contactable/has lost lock, master grabs lock and reassigns tablets
    - GFS replicates data. Prefer to start tablet server on same machine that the data is already at

Master's Tasks (Cont)

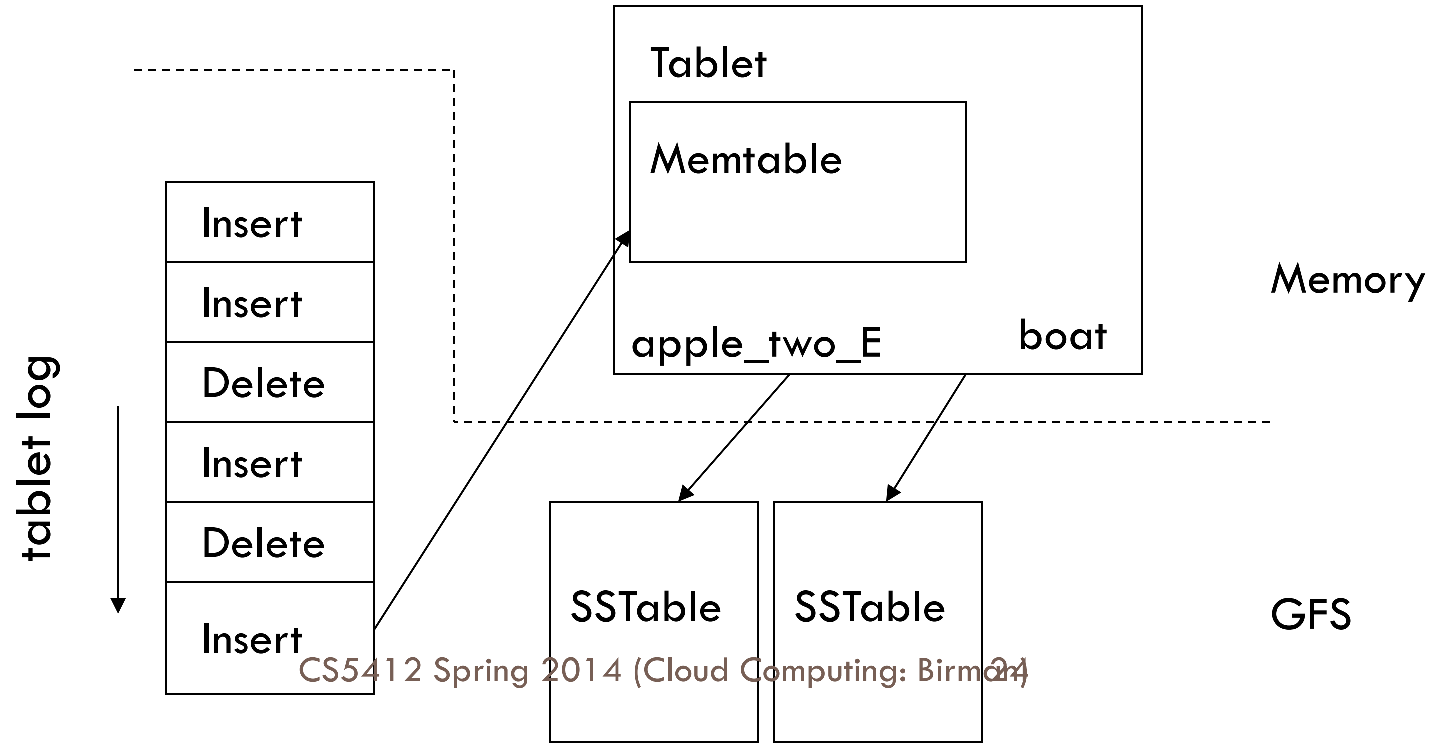
- When (new) master starts
  - grabs master lock on chubby
    - Ensures only one master at a time
  - Finds live servers (scan chubby directory)
  - Communicates with servers to find assigned tablets
  - Scans metadata table to find all tablets
    - Keeps track of unassigned tablets, assigns them
    - Metadata root from chubby, other metadata tablets assigned before scanning.

Metadata Management

- Master handles
  - table creation, and merging of tablet
- Tablet servers directly update metadata on tablet split, then notify master
  - lost notification may be detected lazily by master

Editing a table

- Mutations are logged, then applied to an in-memory memtable
  - May contain “deletion” entries to handle updates
  - Group commit on log: collect multiple updates before log flush



Programming model

- Application reads information
- Uses it to create a group of updates
- Then uses group commit to install them atomically
  - Conflicts? One "wins" and the other "fails", or perhaps both attempts fail
  - But this ensures that data moves in a predictable manner version by version: a form of the ACID model!
- Thus BigTable offers strong consistency

Compactions

- Minor compaction – convert the memtable into an SSTable
  - Reduce memory usage
  - Reduce log traffic on restart
- Merging compaction
  - Reduce number of SSTables
  - Good place to apply policy "keep only N versions"
- Major compaction
  - Merging compaction that results in only one SSTable
  - No deletion records, only live data

Locality Groups

- Group column families together into an SSTable
  - Avoid mingling data, e.g. page contents and page metadata
  - Can keep some groups all in memory
- Can compress locality groups
- Bloom Filters on SSTables in a locality group
  - bitmap on keyvalue hash, used to overestimate which records exist
  - avoid searching SSTable if bit not set
- Tablet movement
  - Major compaction (with concurrent updates)
  - Minor compaction (to catch up with updates) without any concurrent updates
  - Load on new server without requiring any recovery action

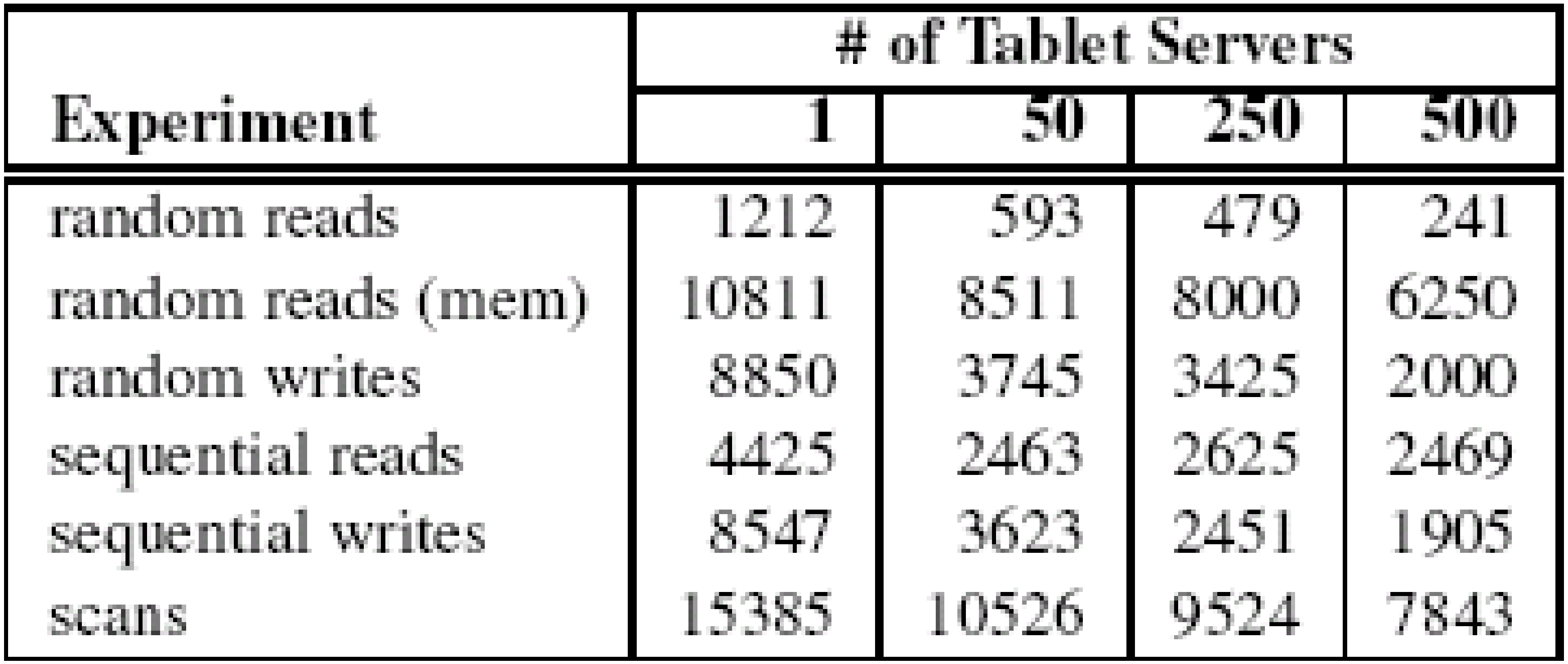
Log Handling

- Commit log is per server, not per tablet (why?)
  - complicates tablet movement
  - when server fails, tablets divided among multiple servers
    - can cause heavy scan load by each such server
    - optimization to avoid multiple separate scans: sort log by (table, rowname, LSN), so logs for a tablet are clustered, then distribute
- GFS delay spikes can mess up log write (time critical)
  - solution: two separate logs, one active at a time
  - can have duplicates between these two

Immutability

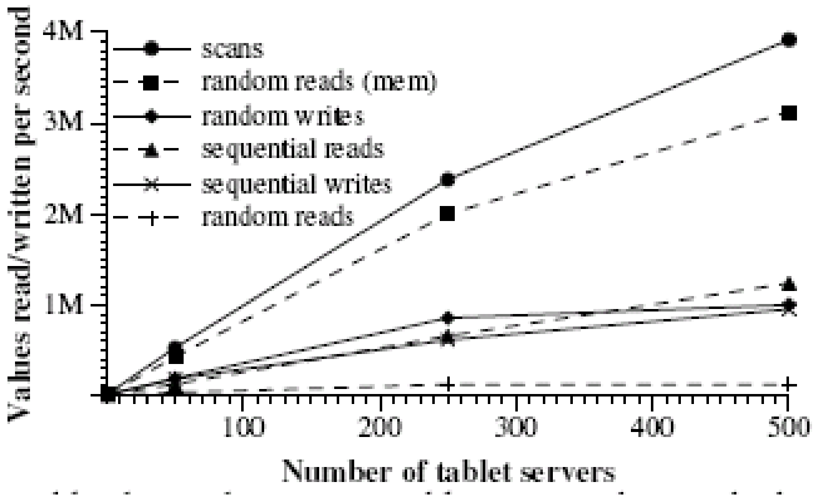
- SSTables are immutable
  - simplifies caching, sharing across GFS etc
  - no need for concurrency control
  - SSTables of a tablet recorded in METADATA table
  - Garbage collection of SSTables done by master
  - On tablet split, split tables can start off quickly on shared SSTables, splitting them lazily
- Only memtable has reads and updates concurrent
  - copy on write rows, allow concurrent read/write

Microbenchmarks

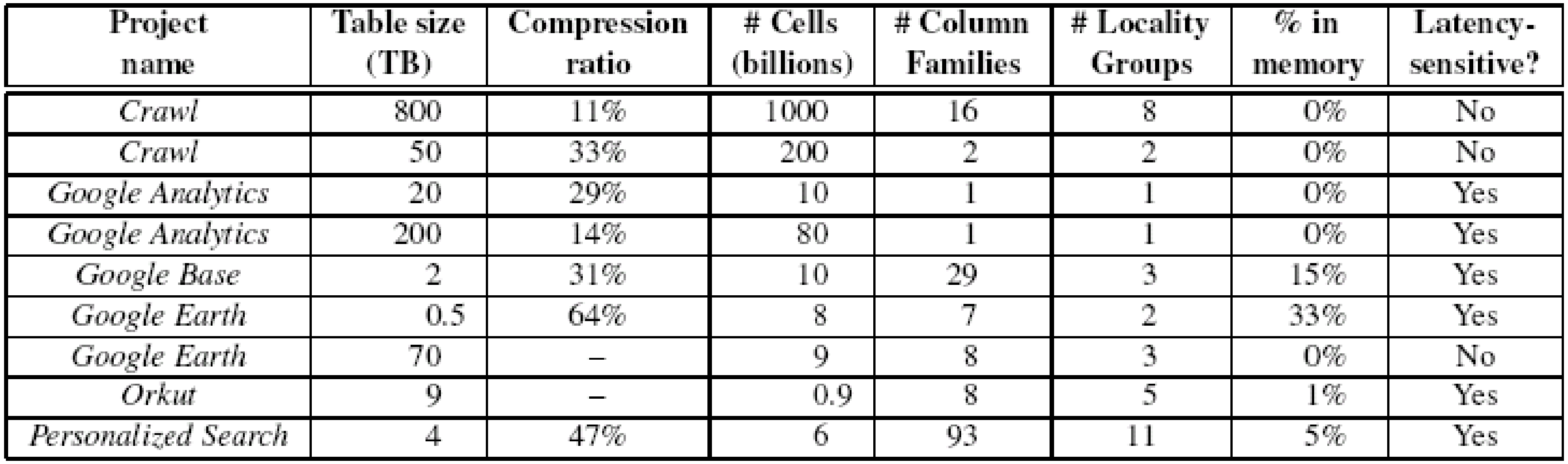


	# of Tablet Servers			
Experiment	**1**	**50**	**250**	**500**
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

Performance



Application at Google

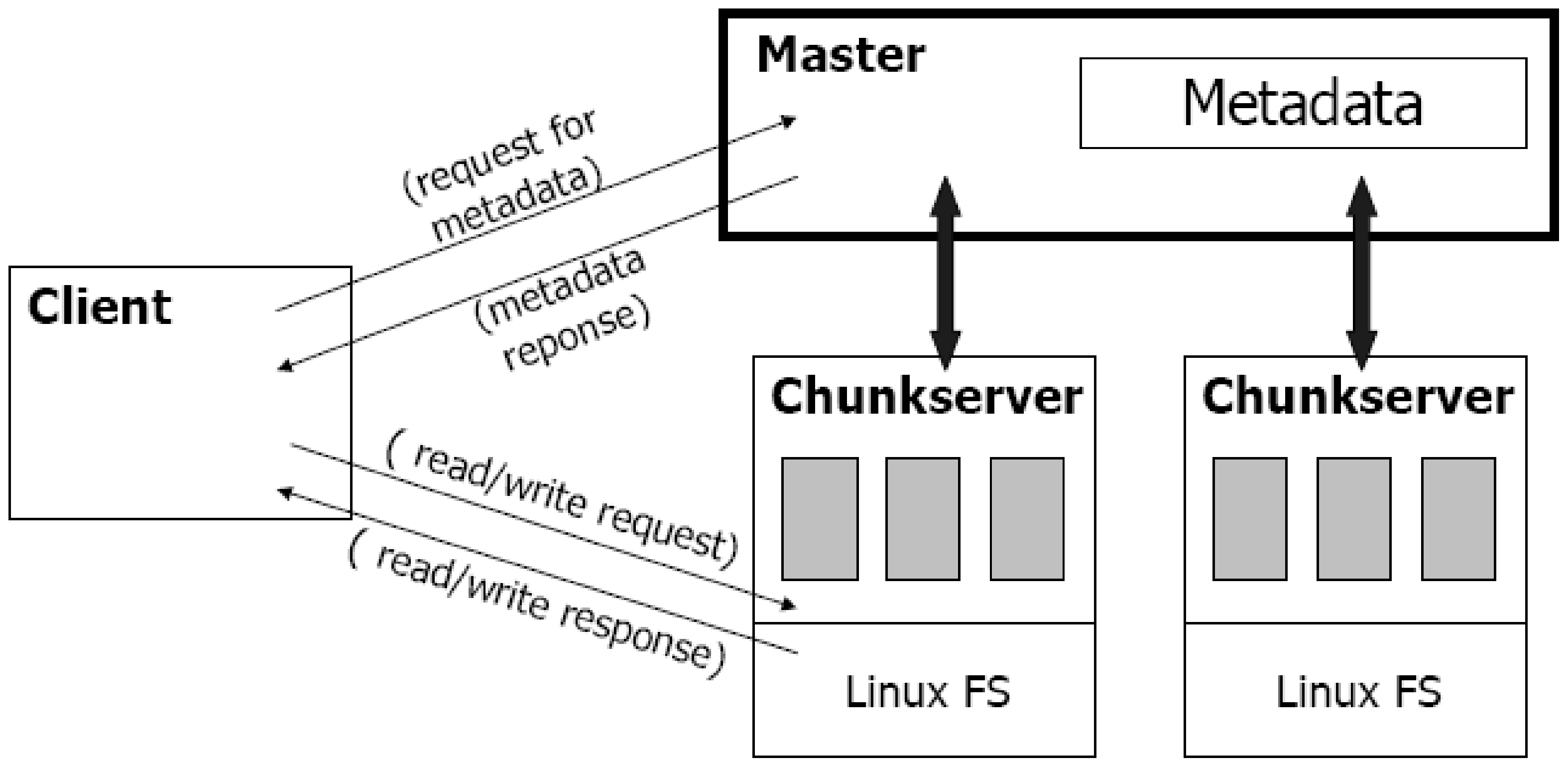


Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

GFS and Chubby

- GFS file system used under the surface for storage
  - Has a master and a set of chunk servers
  - To access a file, ask master… it directs you to some chunk server and provides a capability
  - That server sends you the data
- Chubby lock server
  - Implements locks with varying levels of durability
  - Implemented over Paxos, a protocol we'll look at a few lectures from now

GFS Architecture



Write Algorithm is trickier

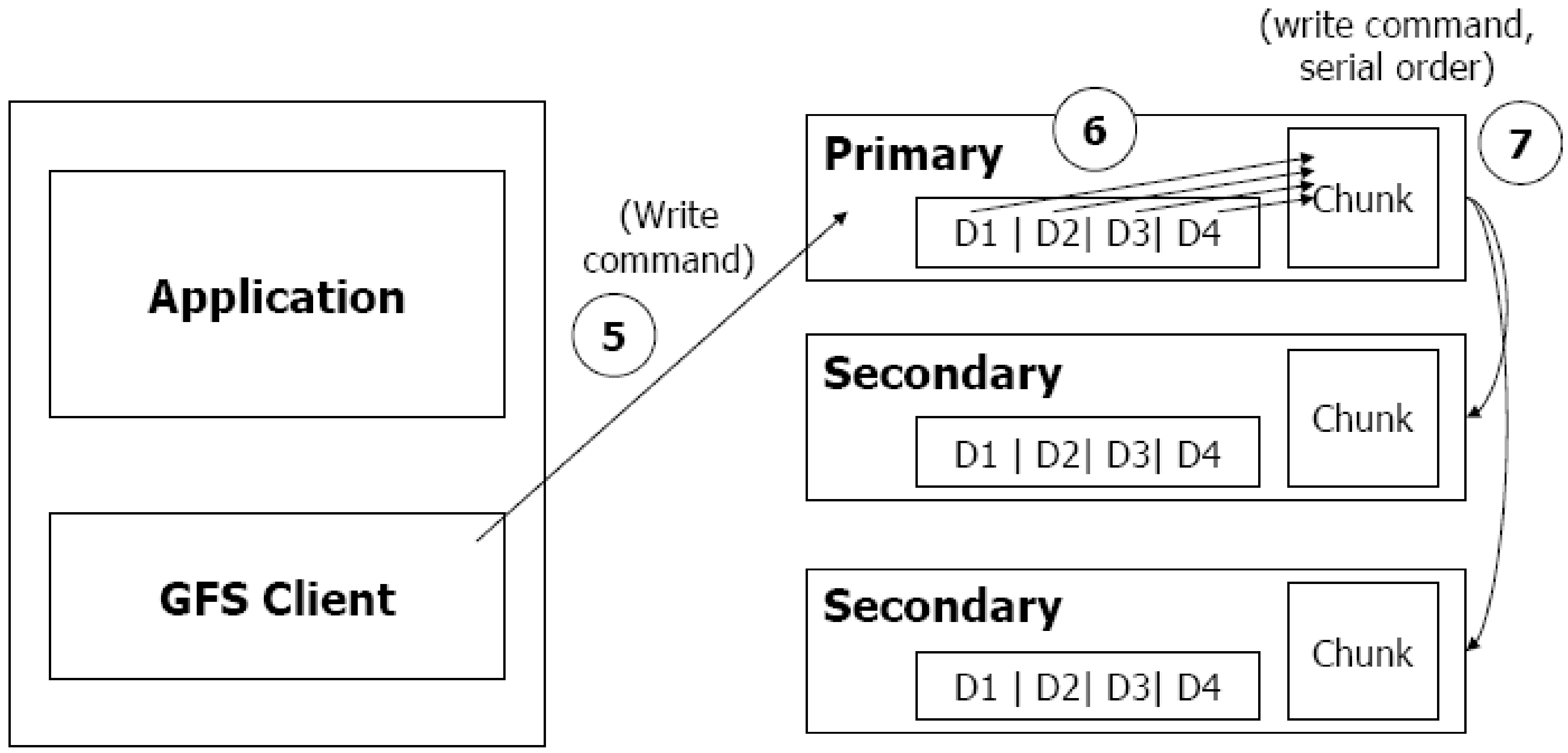
1. Application originates write request.
2. GFS client translates request from (filename, data) -> (filename, chunk index), and sends it to master.
3. Master responds with chunk handle and (primary + secondary) replica locations.
4. Client pushes write data to all locations. Data is stored in chunkservers'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

Write Algorithm is trickier

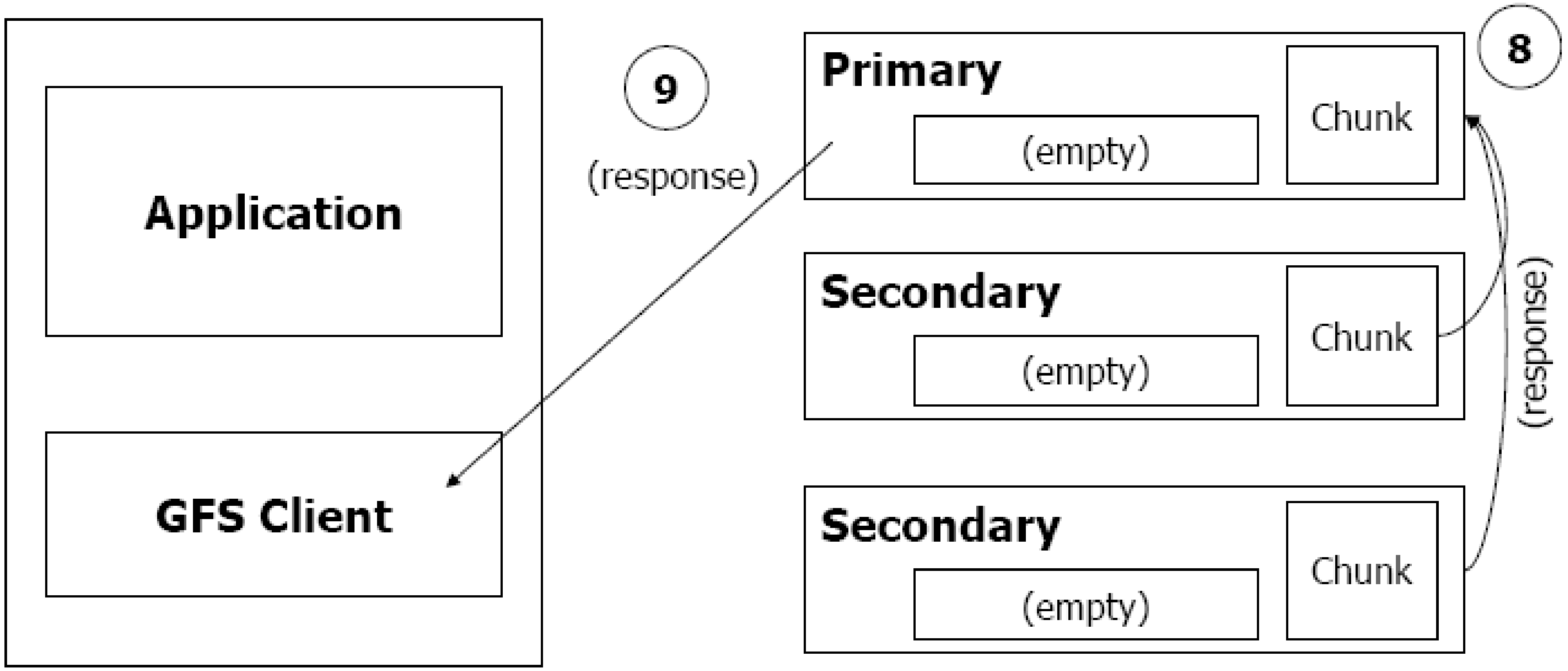
6. Primary determines serial order for data instances stored in its buffer and writes the instances in that order to the chunk.
7. Primary sends serial order to the secondaries and tells them to perform the write.
8. Secondaries respond to the primary.
9. Primary responds back to client.

Note: If write fails at one of chunkservers, client is informed and retries the write.

Write Algorithm is trickier



Write Algorithm is trickier



Zookeeper

- Created at Yahoo!
- Integrates locking and storage into a file system
  - Files play the role of locks
  - Also has a way to create unique version or sequence numbers
  - But basic API is just like a Linux file system
- Implemented using virtual synchrony protocols (we'll study those too, when we talk about Paxos)
- Extremely popular, widely used

Sinfonia

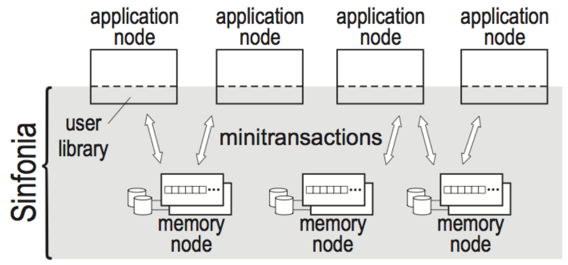
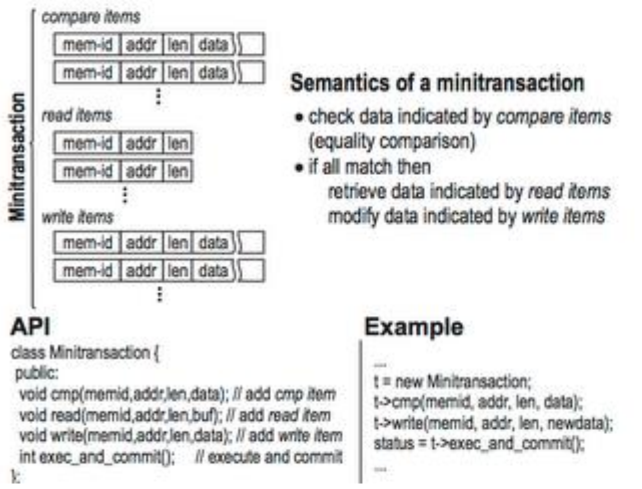


Figure 1: Sinfonia allows application nodes to share data in a fault tolerant, scalable, and consistent manner.

- Created at HP Labs
  - Core construct: durable append-only log replicated for high availability and fast load-balanced reads
  - Concept of a "mini-transaction" that appends to the state
  - Then "specialized" by a series of plug-in modules
    - Can support a file system
    - Lock service
    - Event notification service
    - Message queuing system
    - Database system…
- Like Chubby, uses Paxos at the core

Sinfonia

- To assist developer in gaining more speed, application can precompute transaction using cached data
- At transaction execution time we check validity of the data read during precomputation
- Thus the transation can just do a series of writes at high speed, without delay to think



Semantics of a minitransaction

- check data indicated by *compare items* (equality comparison)
- if all match then
  retrieve data indicated by *read items*
  modify data indicated by *write items*

API

```
class Minitransaction {
 public:
  void cmp(memid,addr,len,data); // add cmp item
  void read(memid,addr,len,buf); // add read item
  void write(memid,addr,len,data); // add write item
  int exec_and_commit();     // execute and commit
};
```

Example

```
...
t = new Minitransaction;
t->cmp(memid, addr, len, data);
t->write(memid, addr, len, newdata);
status = t->exec_and_commit();
...
```

Key idea in Sinfonia

- A persistent, append-oriented durable log offers
  - Strong guarantees of consistency
  - Very effective fault-tolerance, if implemented properly
  - A kind of version-history model

- We can generalize from this to implement all those other applications by using Sinfonia as a version store or a data history
  - Seen this way, very much like the BigTable "story"!

Second idea

- Precomputation allows us to create lots of read-only data replicas that can be used for offline computation
  - Sometimes it can be very slow to compute a database operation, like a big join
  - So we do this "offline" permitting massive speedups
- By validating that the data didn't change we can then apply just the updates in a very fast transaction after we've figured out the answer
  - Note that if we "re-ran" the whole computation we would get the same answers, since inputs are unchanged!

MapReduce

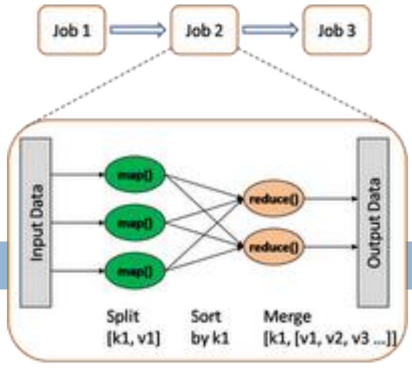


Fig 3. Map Reduce programming model

- Used for functional style of computing with massive numbers of machines and huge data sets
- Works in a series of stages
  - **Map** takes some operations and "maps" it on a set of servers so that each does some part
  - The operations are functional: they don't modify the data they read and can be reissued if needed
  - Result: a large number of partial results, each from running the function on some part of the data
  - **Reduce** combines these partial results to obtain a smaller set of result files (perhaps just one, perhaps a few)
- Often iterates with further map/reduce stages

Hadoop

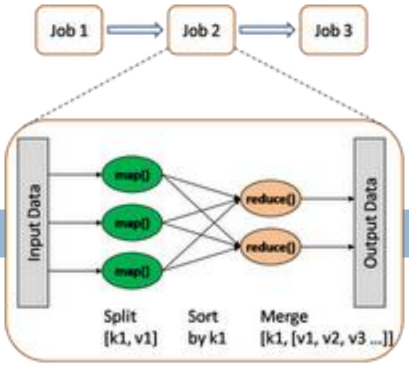


Fig 3. Map Reduce programming model

- Open source MapReduce
  - Has many refinements and improvements
  - Widely popular and used even at Google!

- Challenges
  - Dealing with variable sets of worker nodes
  - Computation is functional; hard to accommodate adaptive events such as changing parameter values based on rate of convergence of a computation

Classic MapReduce examples

- Make a list of terms appearing in some set of web pages, counting the frequency
- Find common misspellings for a word
- Sort a very large data set via a partitioning merge sort

- Nice features:
  - Relatively easy to program
  - Automates parallelism, failure handling, data management tasks

MapReduce debate

- The database community dislikes MapReduce
  - Databases can do the same things
  - In fact can do far more things
  - And database queries can be compiled automatically into MapReduce patterns; this is done in big parallel database products all the time!

- Counter-argument:
  - Easy to customize MapReduce for a new application
  - Hadoop is free, parallel databases not so much…

Summary

- We've touched upon a series of examples of cloud computing infrastructure components
  - Each really could have had a whole lecture

- They aren't simple systems and many were very hard to implement!
  - Hard to design… hard to build… hard to optimize for stable and high quality operation at scale
  - Major teams and huge resource investments
  - Design decisions that may sound simple often required very careful thought and much debate and experimentation!

Summary

- Some recurring themes
  - Data replication using (key,value) tuples
  - Anticipated update rates, sizes, scalability drive design
  - Use of multicast mechanisms: Paxos, virtual synchrony
  - Need to plan adaptive behaviors if nodes come and go, or crash, while system is running
  - High value for "latency tolerant" solutions
    - Extremely asynchronous structures
    - Parallel: work gets done "out there"
- Many offer strong consistency guarantees, "overcoming" the CAP theorem in various ways