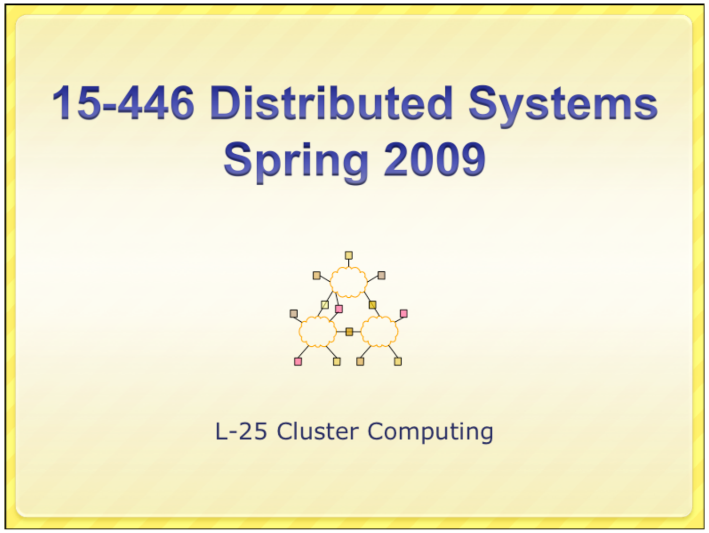
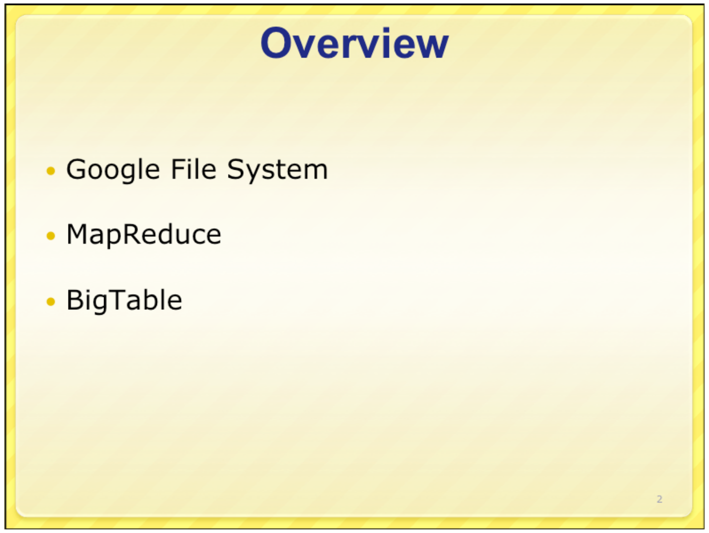


15-446 Distributed Systems Spring 2009

L-25 Cluster Computing



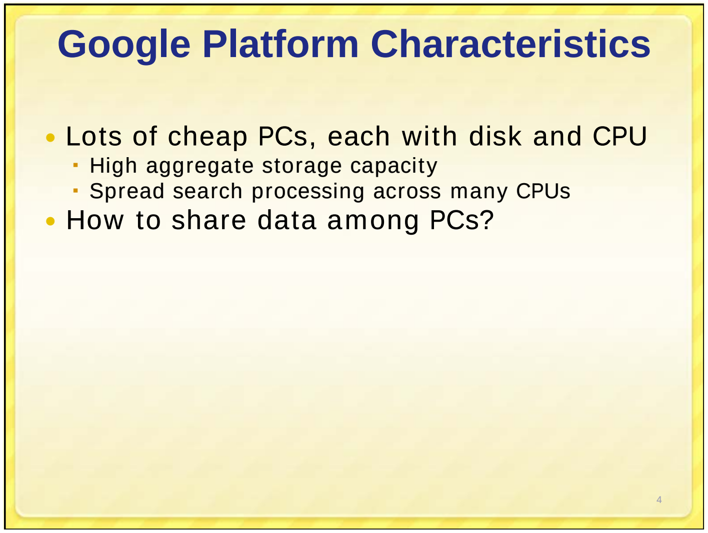
Overview

- Google File System
- MapReduce
- BigTable

Google Disk Farm

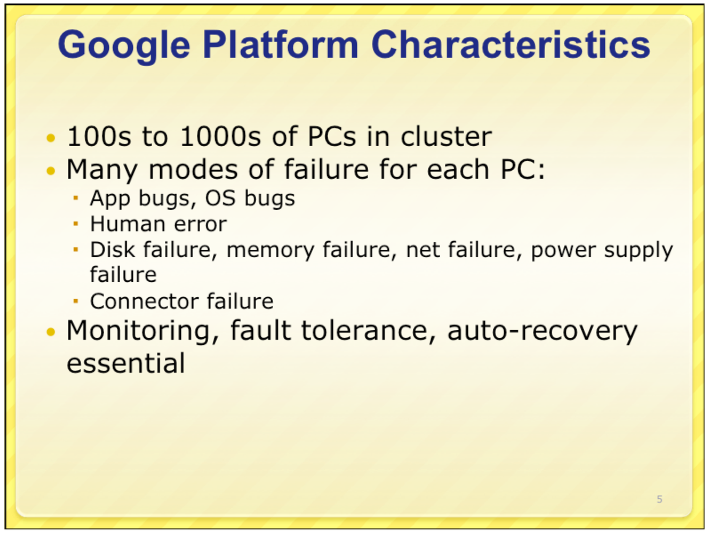
Early days…

…today



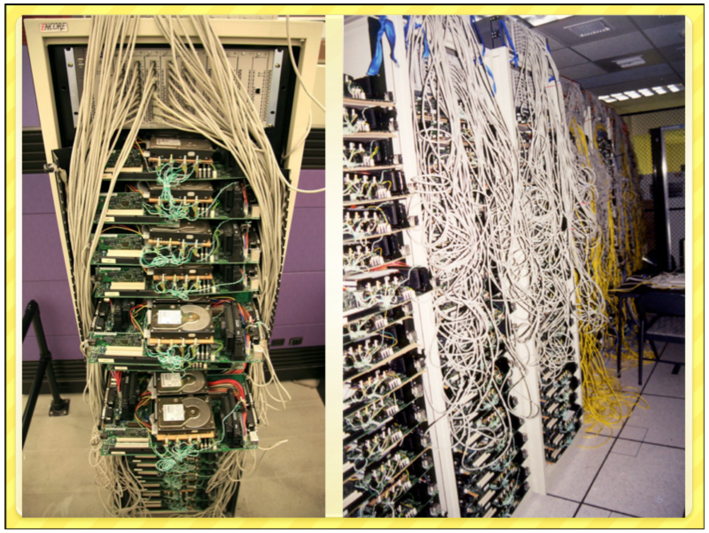
Google Platform Characteristics

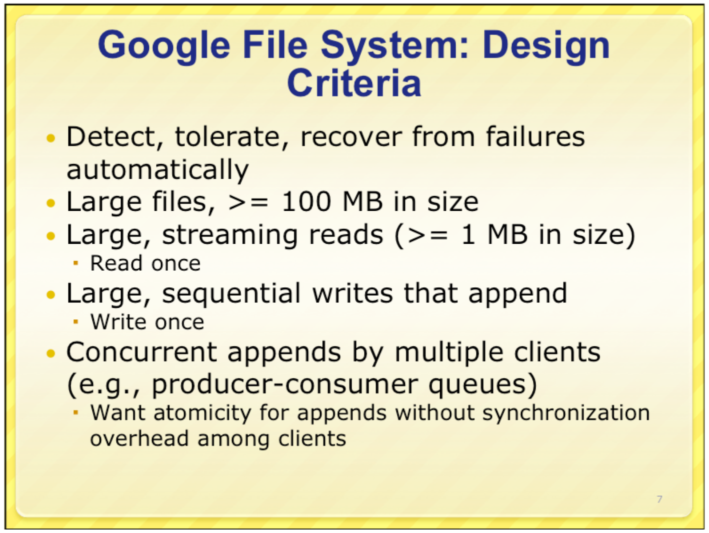
- Lots of cheap PCs, each with disk and CPU
  - High aggregate storage capacity
  - Spread search processing across many CPUs
- How to share data among PCs?



Google Platform Characteristics

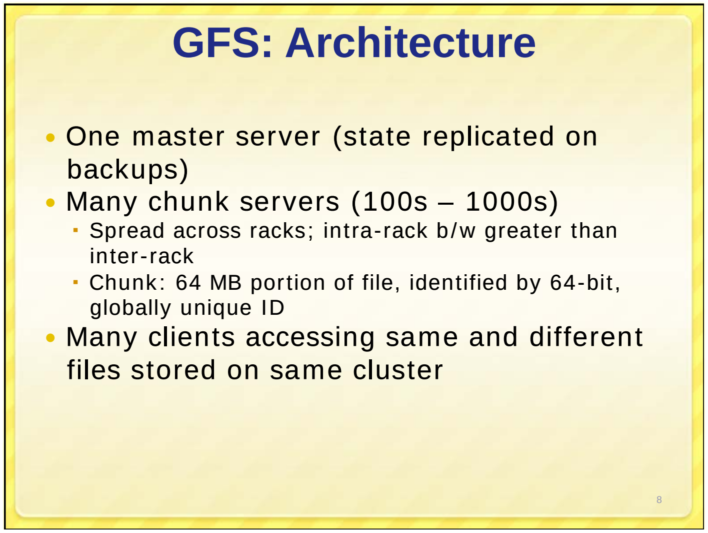
- 100s to 1000s of PCs in cluster
- Many modes of failure for each PC:
  - App bugs, OS bugs
  - Human error
  - Disk failure, memory failure, net failure, power supply failure
  - Connector failure
- Monitoring, fault tolerance, auto-recovery essential





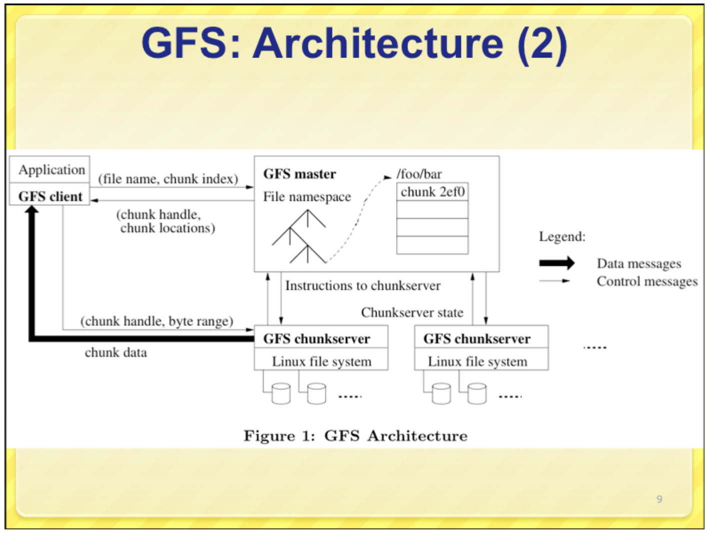
Google File System: Design Criteria

- Detect, tolerate, recover from failures automatically
- Large files, >= 100 MB in size
- Large, streaming reads (>= 1 MB in size)
  - Read once
- Large, sequential writes that append
  - Write once
- Concurrent appends by multiple clients (e.g., producer-consumer queues)
  - Want atomicity for appends without synchronization overhead among clients



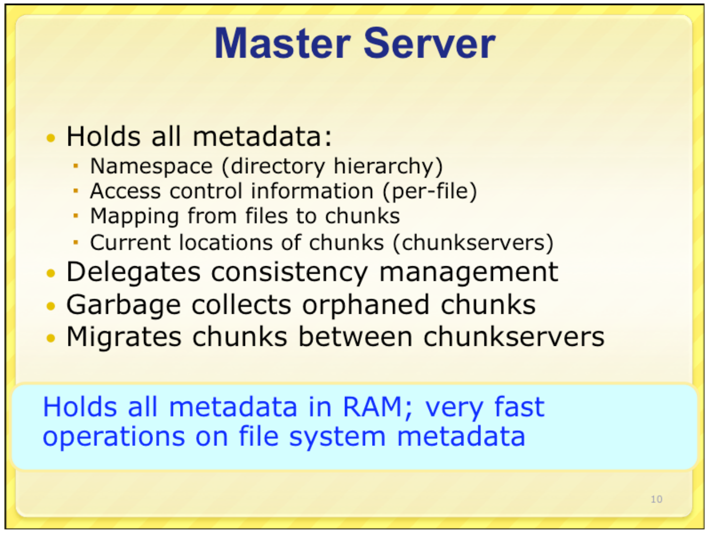
GFS: Architecture

- One master server (state replicated on backups)
- Many chunk servers (100s – 1000s)
  - Spread across racks; intra-rack b/w greater than inter-rack
  - Chunk: 64 MB portion of file, identified by 64-bit, globally unique ID
- Many clients accessing same and different files stored on same cluster



GFS: Architecture (2)

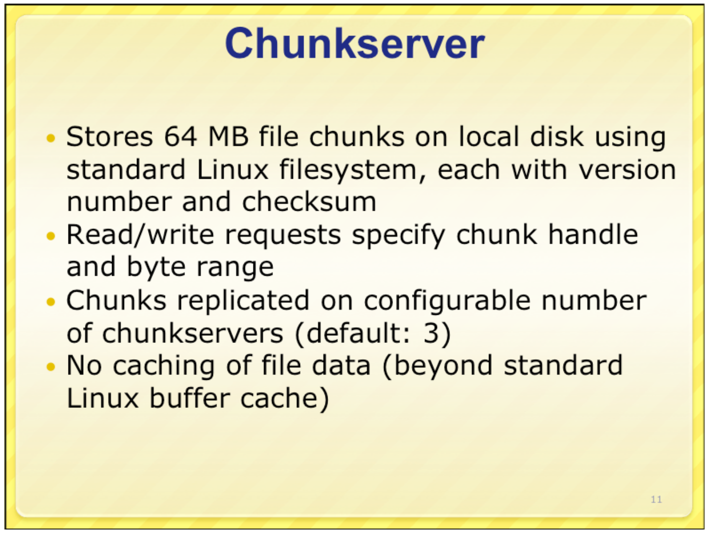
Figure 1: GFS Architecture



Master Server

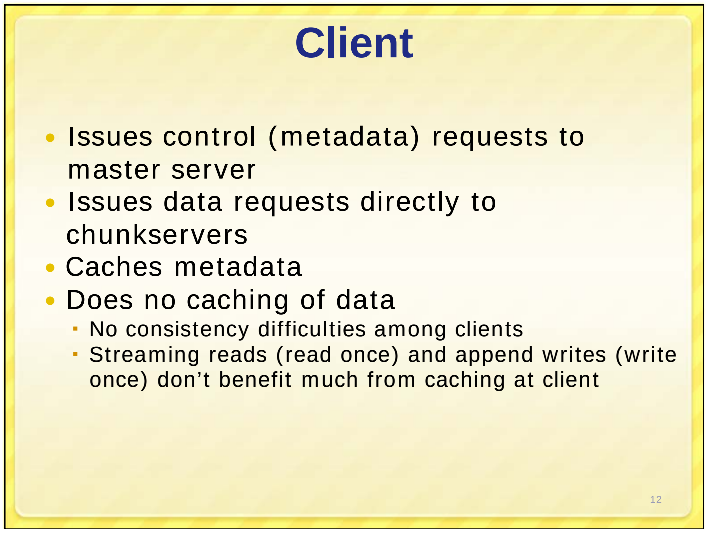
- Holds all metadata:
  - Namespace (directory hierarchy)
  - Access control information (per-file)
  - Mapping from files to chunks
  - Current locations of chunks (chunkservers)
- Delegates consistency management
- Garbage collects orphaned chunks
- Migrates chunks between chunkservers

Holds all metadata in RAM; very fast operations on file system metadata



Chunkserver

- Stores 64 MB file chunks on local disk using standard Linux filesystem, each with version number and checksum
- Read/write requests specify chunk handle and byte range
- Chunks replicated on configurable number of chunkservers (default: 3)
- No caching of file data (beyond standard Linux buffer cache)



Client

- Issues control (metadata) requests to master server
- Issues data requests directly to chunkservers
- Caches metadata
- Does no caching of data
  - No consistency difficulties among clients
  - Streaming reads (read once) and append writes (write once) don't benefit much from caching at client

Client Read

- Client sends master:
  - read(file name, chunk index)
- Master's reply:
  - chunk ID, chunk version number, locations of replicas
- Client sends "closest" chunkserver w/ replica:
  - read(chunk ID, byte range)
  - "Closest" determined by IP address on simple rack-based network topology
- Chunkserver replies with data

Client Write

- Some chunkserver is primary for each chunk
  - Master grants lease to primary (typically for 60 sec.)
  - Leases renewed using periodic heartbeat messages between master and chunkservers
- Client asks master for primary and secondary replicas for each chunk
- Client sends data to replicas in daisy chain
  - Pipelined: each replica forwards as it receives
  - Takes advantage of full-duplex Ethernet links

Client Write (2)

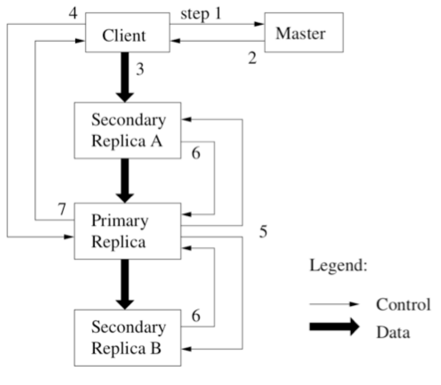
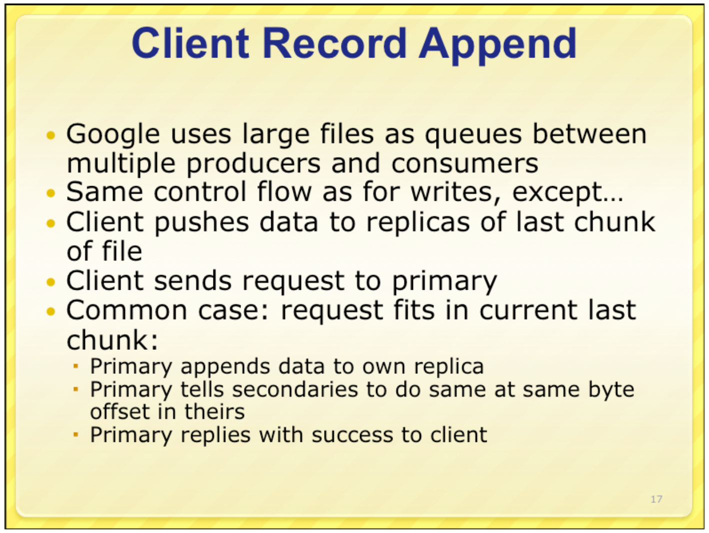


Figure 2: Write Control and Data Flow

Client Write (3)

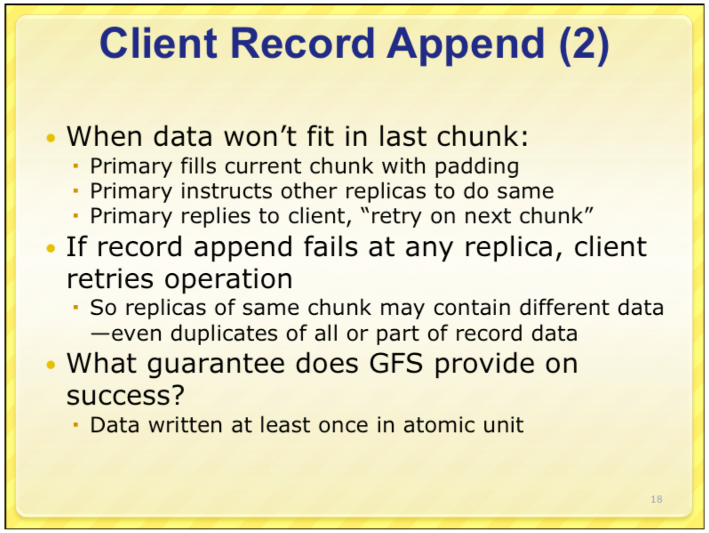
- All replicas acknowledge data write to client
- Client sends write request to primary
- Primary assigns serial number to write request, providing ordering
- Primary forwards write request with same serial number to secondaries
- Secondaries all reply to primary after completing write
- Primary replies to client



Client Record Append

- Google uses large files as queues between multiple producers and consumers
- Same control flow as for writes, except…
- Client pushes data to replicas of last chunk of file
- Client sends request to primary
- Common case: request fits in current last chunk:
  - Primary appends data to own replica
  - Primary tells secondaries to do same at same byte offset in theirs
  - Primary replies with success to client

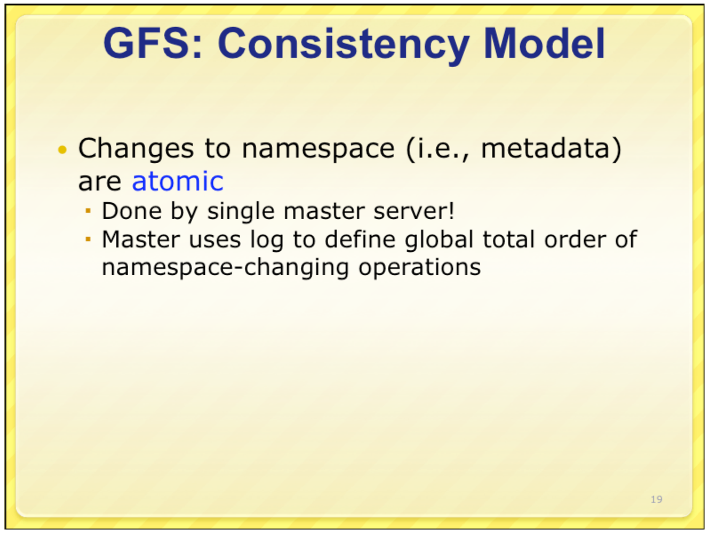
17



Client Record Append (2)

- When data won't fit in last chunk:
  - Primary fills current chunk with padding
  - Primary instructs other replicas to do same
  - Primary replies to client, "retry on next chunk"
- If record append fails at any replica, client retries operation
  - So replicas of same chunk may contain different data—even duplicates of all or part of record data
- What guarantee does GFS provide on success?
  - Data written at least once in atomic unit

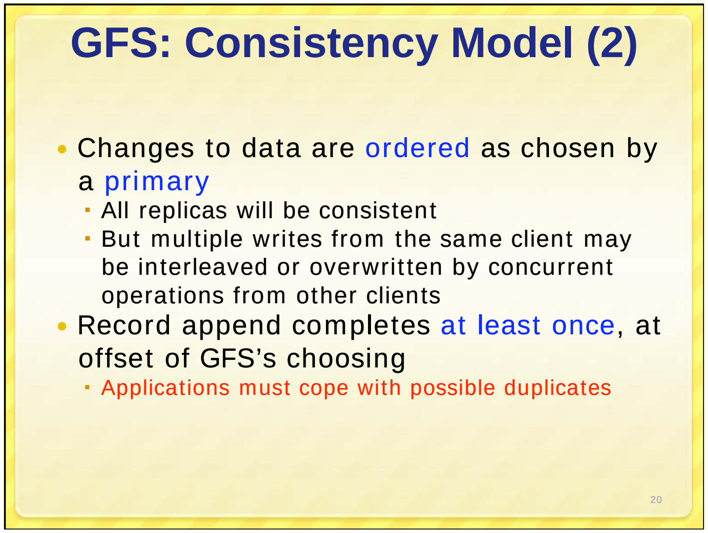
18



GFS: Consistency Model

- Changes to namespace (i.e., metadata) are atomic
  - Done by single master server!
  - Master uses log to define global total order of namespace-changing operations

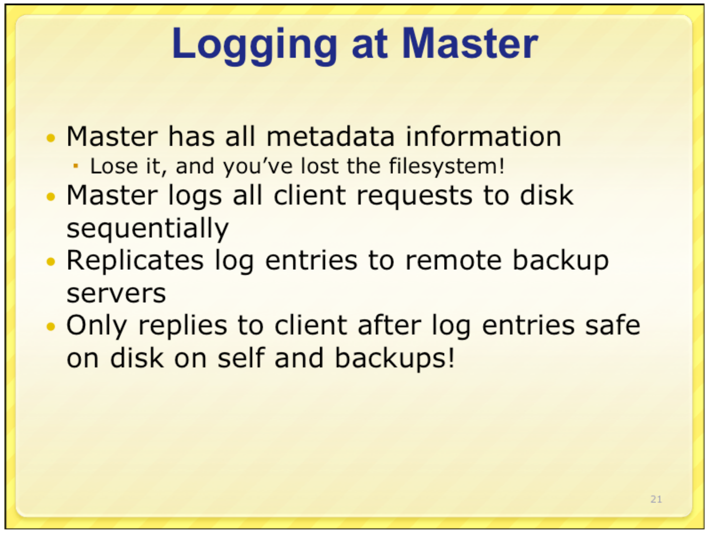
19



GFS: Consistency Model (2)

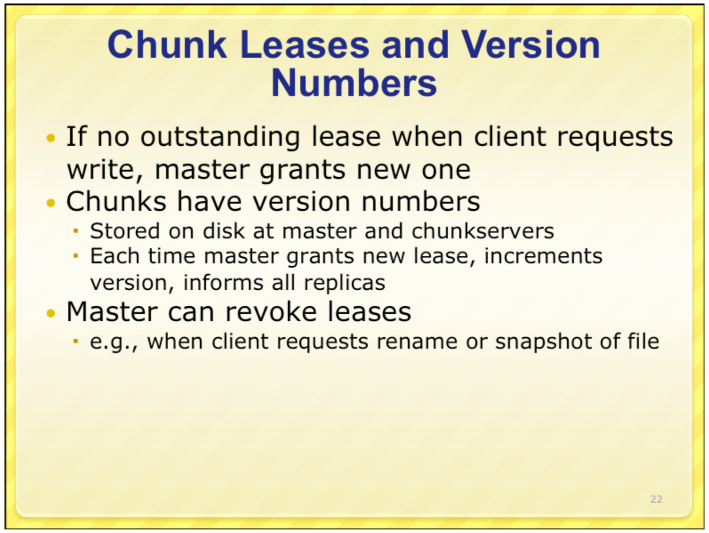
- Changes to data are ordered as chosen by a primary
  - All replicas will be consistent
  - But multiple writes from the same client may be interleaved or overwritten by concurrent operations from other clients
- Record append completes at least once, at offset of GFS's choosing
  - Applications must cope with possible duplicates

20



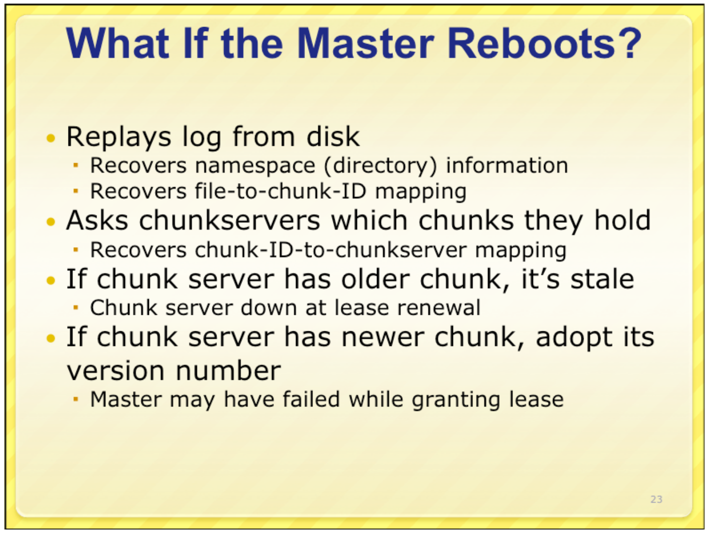
Logging at Master

- Master has all metadata information
  - Lose it, and you've lost the filesystem!
- Master logs all client requests to disk sequentially
- Replicates log entries to remote backup servers
- Only replies to client after log entries safe on disk on self and backups!



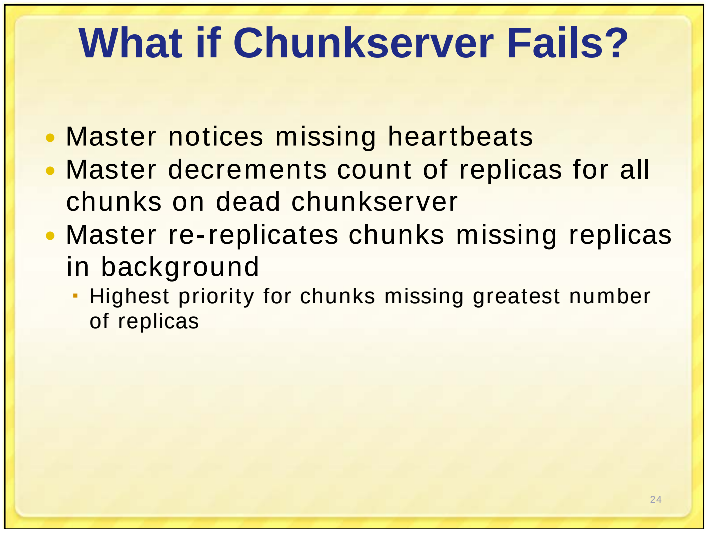
Chunk Leases and Version Numbers

- If no outstanding lease when client requests write, master grants new one
- Chunks have version numbers
  - Stored on disk at master and chunkservers
  - Each time master grants new lease, increments version, informs all replicas
- Master can revoke leases
  - e.g., when client requests rename or snapshot of file



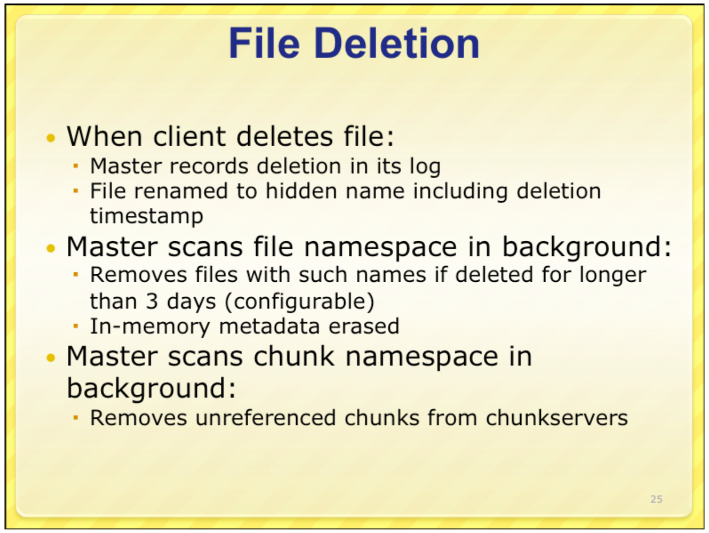
What If the Master Reboots?

- Replays log from disk
  - Recovers namespace (directory) information
  - Recovers file-to-chunk-ID mapping
- Asks chunkservers which chunks they hold
  - Recovers chunk-ID-to-chunkserver mapping
- If chunk server has older chunk, it's stale
  - Chunk server down at lease renewal
- If chunk server has newer chunk, adopt its version number
  - Master may have failed while granting lease



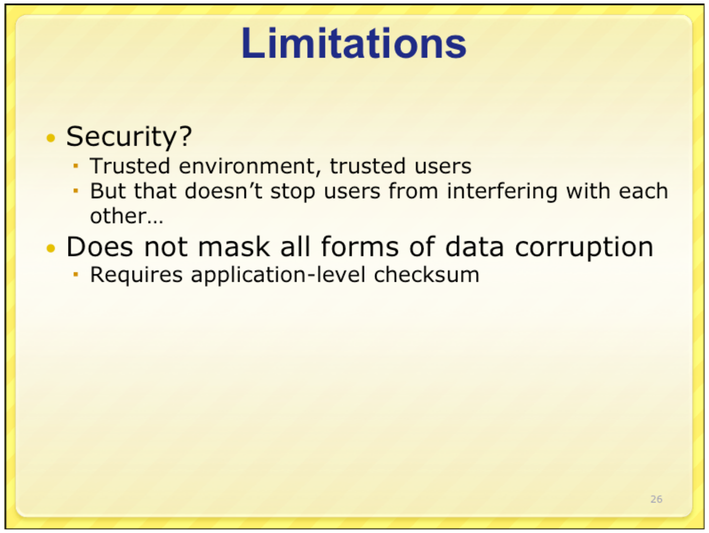
What if Chunkserver Fails?

- Master notices missing heartbeats
- Master decrements count of replicas for all chunks on dead chunkserver
- Master re-replicates chunks missing replicas in background
  - Highest priority for chunks missing greatest number of replicas



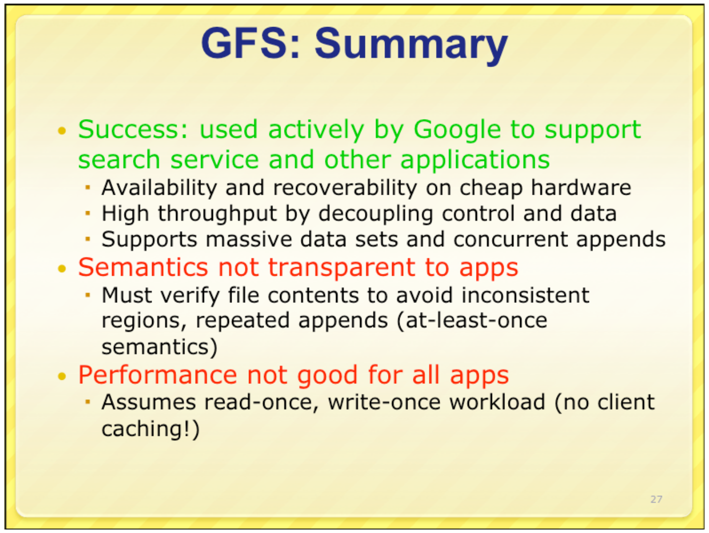
File Deletion

- When client deletes file:
  - Master records deletion in its log
  - File renamed to hidden name including deletion timestamp
- Master scans file namespace in background:
  - Removes files with such names if deleted for longer than 3 days (configurable)
  - In-memory metadata erased
- Master scans chunk namespace in background:
  - Removes unreferenced chunks from chunkservers



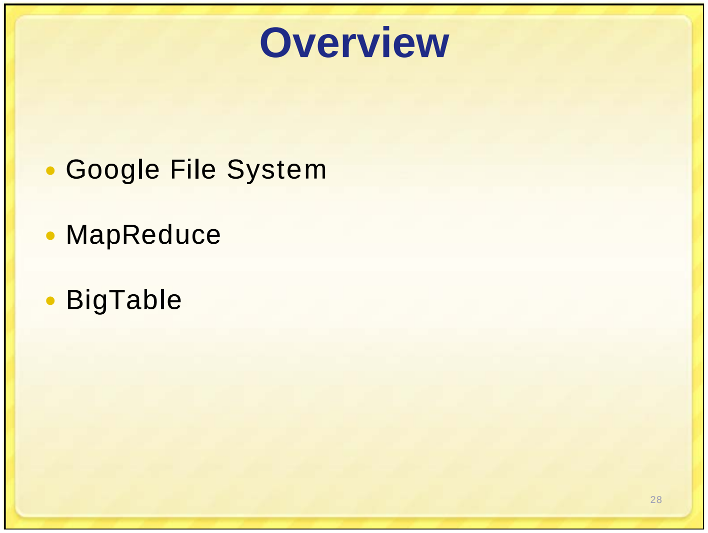
Limitations

- Security?
  - Trusted environment, trusted users
  - But that doesn’t stop users from interfering with each other…
- Does not mask all forms of data corruption
  - Requires application-level checksum



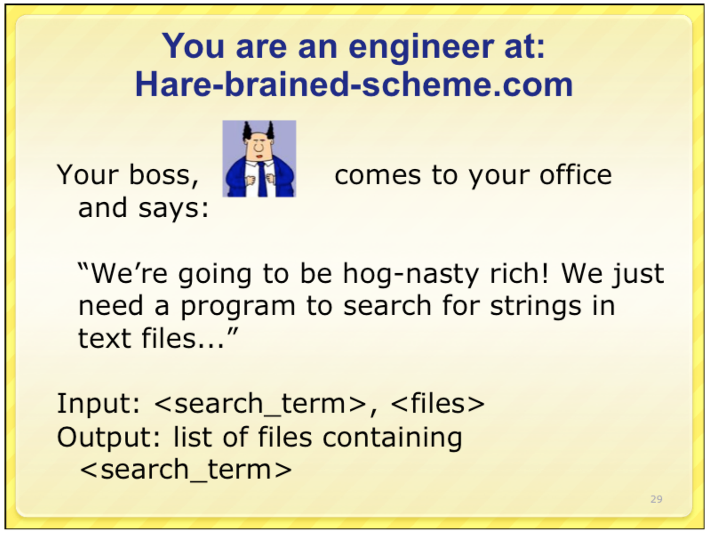
GFS: Summary

- Success: used actively by Google to support search service and other applications
  - Availability and recoverability on cheap hardware
  - High throughput by decoupling control and data
  - Supports massive data sets and concurrent appends
- Semantics not transparent to apps
  - Must verify file contents to avoid inconsistent regions, repeated appends (at-least-once semantics)
- Performance not good for all apps
  - Assumes read-once, write-once workload (no client caching!)



Overview

- Google File System
- MapReduce
- BigTable

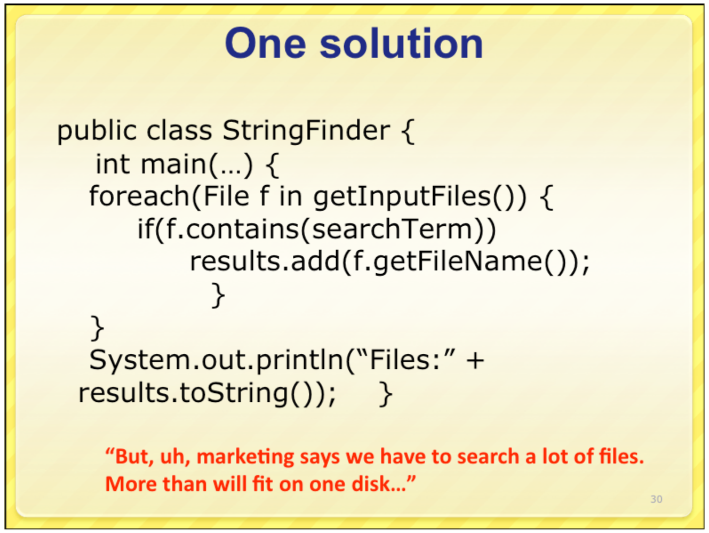


You are an engineer at: Hare-brained-scheme.com

Your boss, comes to your office and says:

"We're going to be hog-nasty rich! We just need a program to search for strings in text files…"

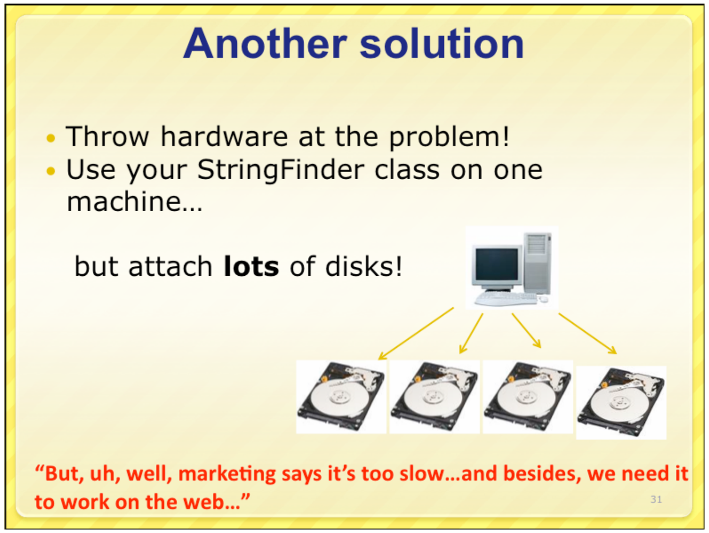
Input: <search_term>, <files>
Output: list of files containing <search_term>



One solution

```
public class StringFinder {
    int main(…) {
   foreach(File f in getInputFiles()) {
        if(f.contains(searchTerm))
             results.add(f.getFileName());
               }
   }
   System.out.println("Files:" +
  results.toString());    }
```

"But, uh, marketing says we have to search a lot of files. More than will fit on one disk…"

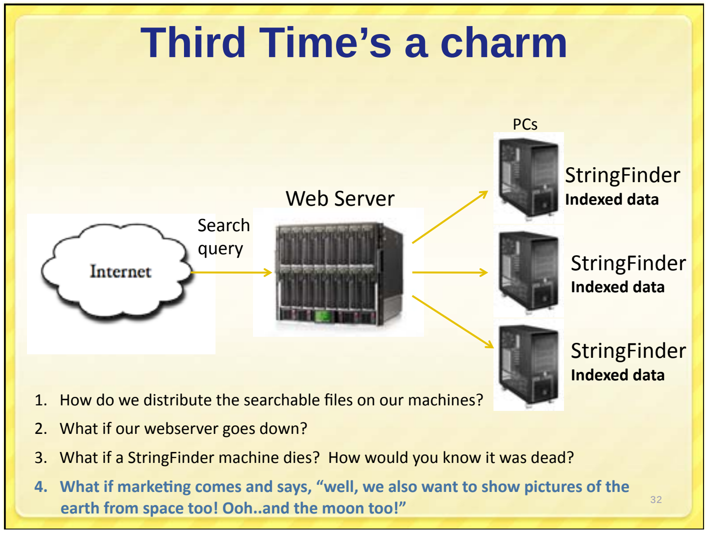


Another solution

- Throw hardware at the problem!
- Use your StringFinder class on one machine…

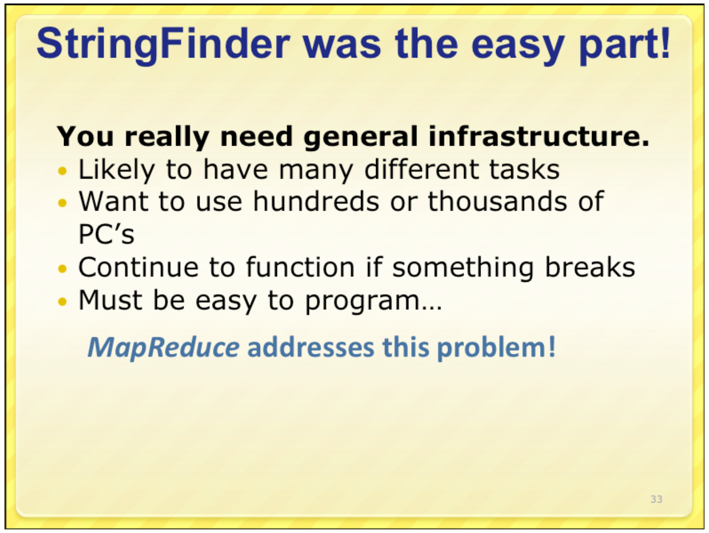
but attach **lots** of disks!

"But, uh, well, marketing says it's too slow…and besides, we need it to work on the web…"



Third Time's a charm

1. How do we distribute the searchable files on our machines?
2. What if our webserver goes down?
3. What if a StringFinder machine dies? How would you know it was dead?
4. What if marketing comes and says, "well, we also want to show pictures of the earth from space too! Ooh..and the moon too!"

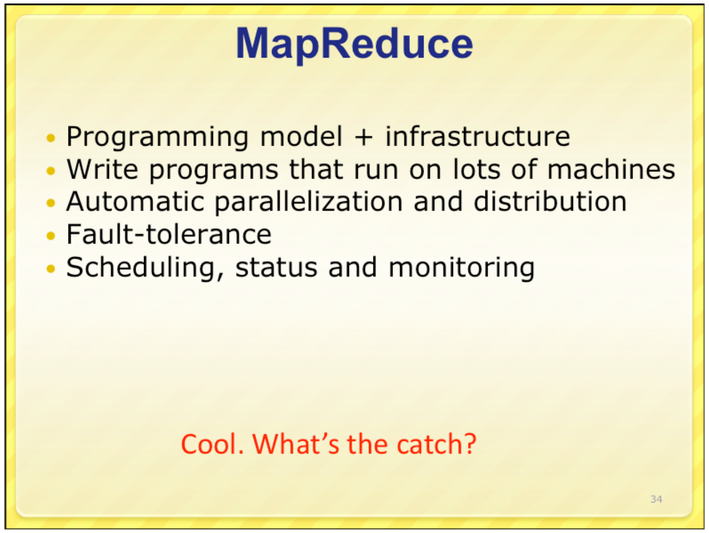


StringFinder was the easy part!

You really need general infrastructure.

- Likely to have many different tasks
- Want to use hundreds or thousands of PC's
- Continue to function if something breaks
- Must be easy to program…

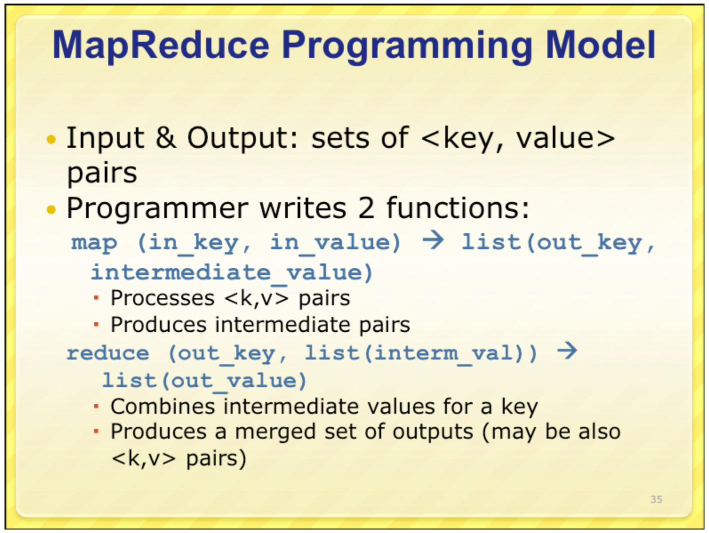
MapReduce addresses this problem!



MapReduce

- Programming model + infrastructure
- Write programs that run on lots of machines
- Automatic parallelization and distribution
- Fault-tolerance
- Scheduling, status and monitoring

Cool. What's the catch?



MapReduce Programming Model

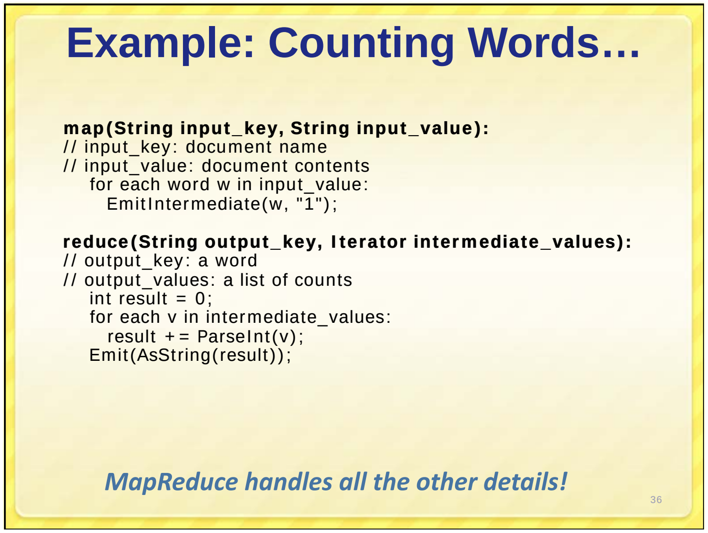
- Input & Output: sets of <key, value> pairs
- Programmer writes 2 functions:

`map (in_key, in_value) → list(out_key, intermediate_value)`

- Processes <k,v> pairs
- Produces intermediate pairs

`reduce (out_key, list(interm_val)) → list(out_value)`

- Combines intermediate values for a key
- Produces a merged set of outputs (may be also <k,v> pairs)

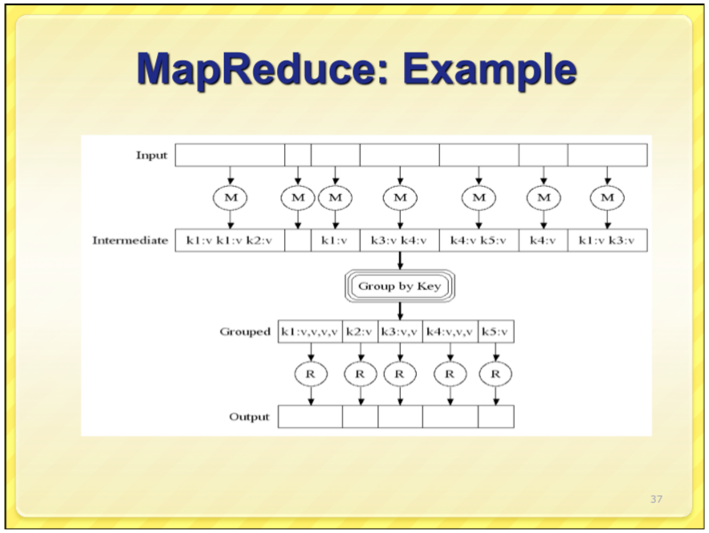


Example: Counting Words…

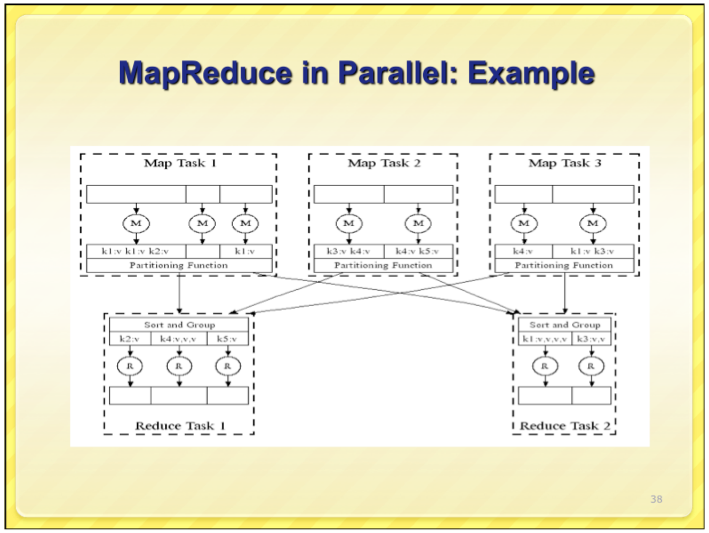
```
map(String input_key, String input_value):
// input_key: document name
// input_value: document contents
   for each word w in input_value:
     EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
// output_key: a word
// output_values: a list of counts
   int result = 0;
   for each v in intermediate_values:
     result += ParseInt(v);
   Emit(AsString(result));
```

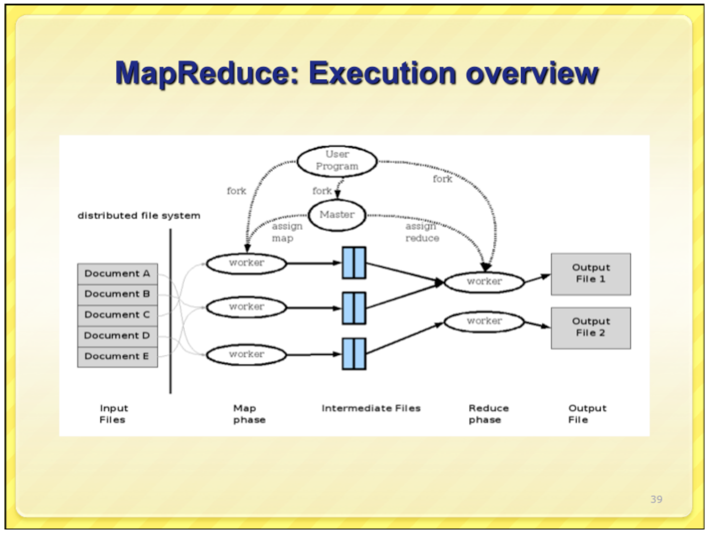
MapReduce handles all the other details!



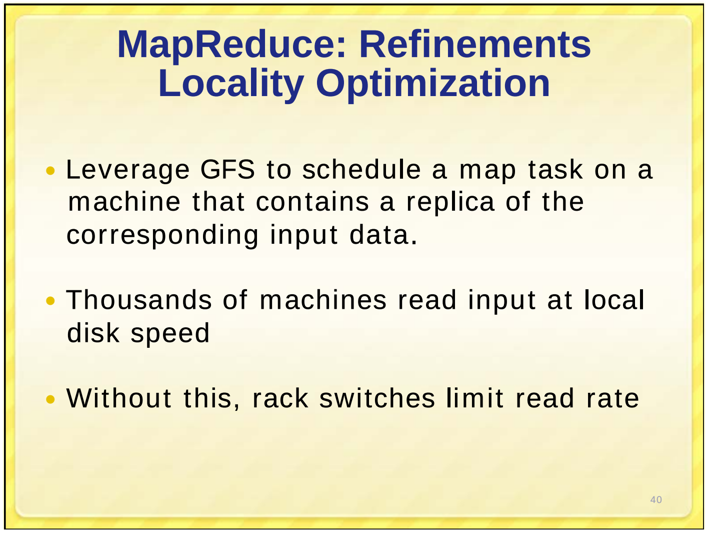
MapReduce: Example



MapReduce in Parallel: Example

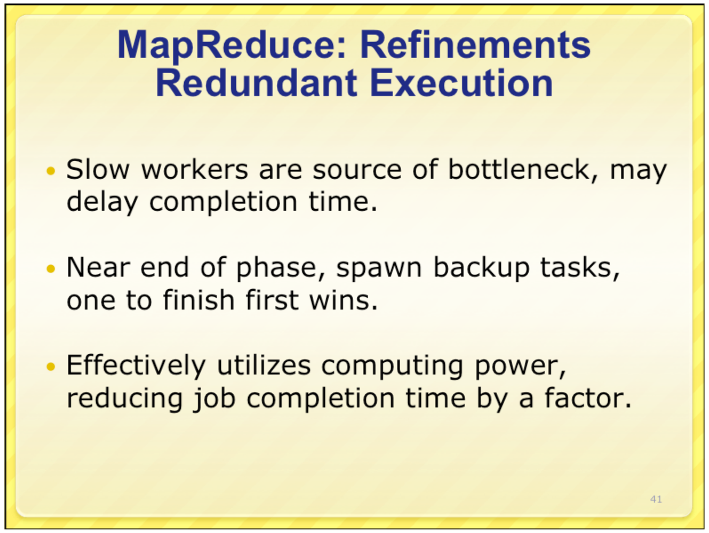


MapReduce: Execution overview



MapReduce: Refinements Locality Optimization

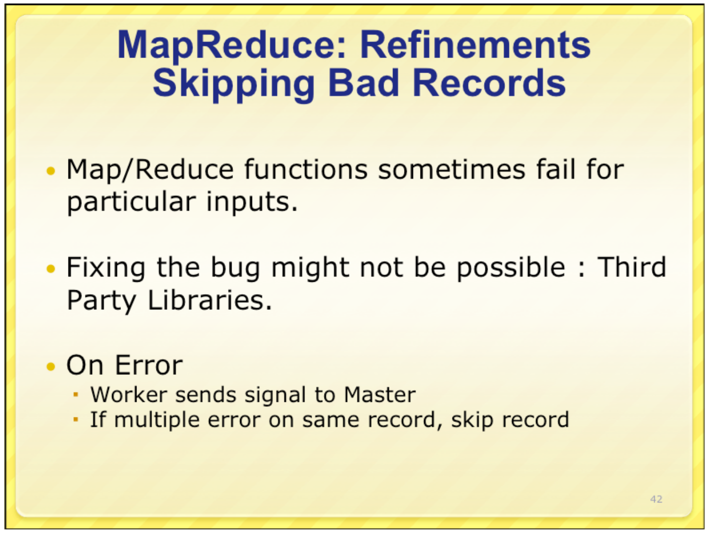
- Leverage GFS to schedule a map task on a machine that contains a replica of the corresponding input data.
- Thousands of machines read input at local disk speed
- Without this, rack switches limit read rate



MapReduce: Refinements Redundant Execution

- Slow workers are source of bottleneck, may delay completion time.
- Near end of phase, spawn backup tasks, one to finish first wins.
- Effectively utilizes computing power, reducing job completion time by a factor.

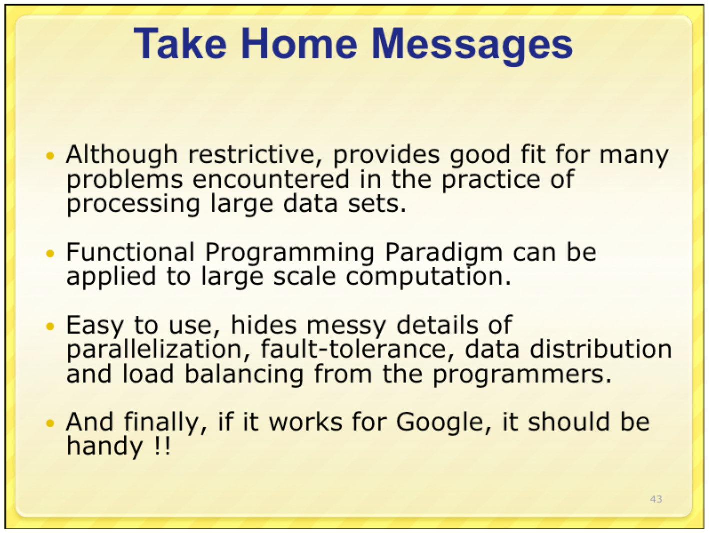
41



MapReduce: Refinements Skipping Bad Records

- Map/Reduce functions sometimes fail for particular inputs.
- Fixing the bug might not be possible : Third Party Libraries.
- On Error
  - Worker sends signal to Master
  - If multiple error on same record, skip record

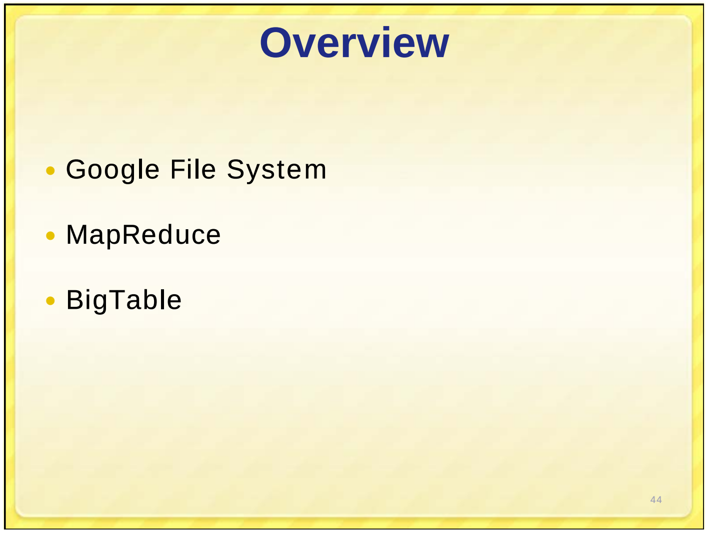
42



Take Home Messages

- Although restrictive, provides good fit for many problems encountered in the practice of processing large data sets.
- Functional Programming Paradigm can be applied to large scale computation.
- Easy to use, hides messy details of parallelization, fault-tolerance, data distribution and load balancing from the programmers.
- And finally, if it works for Google, it should be handy !!

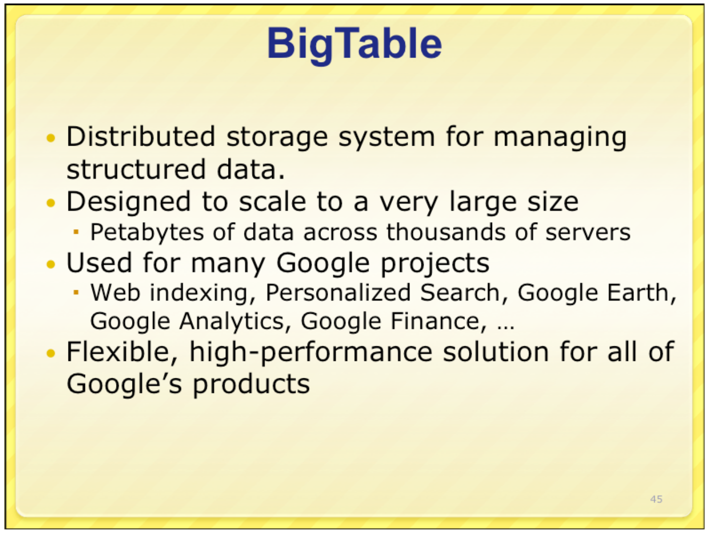
43



Overview

- Google File System
- MapReduce
- BigTable

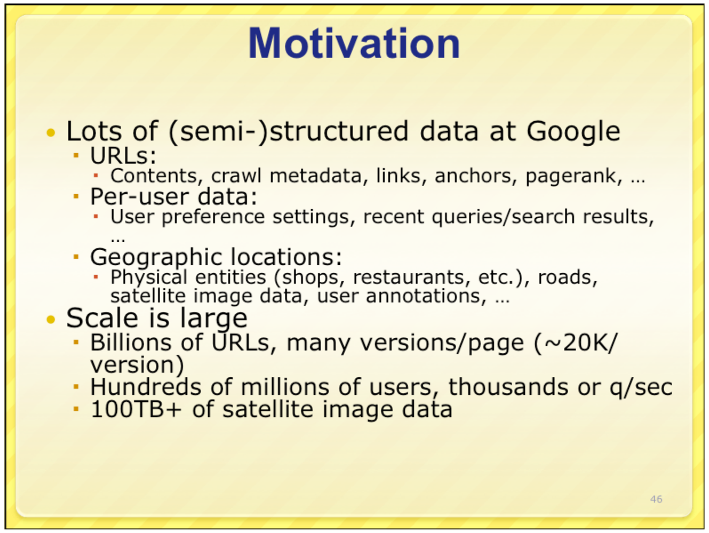
44



BigTable

- Distributed storage system for managing structured data.
- Designed to scale to a very large size
  - Petabytes of data across thousands of servers
- Used for many Google projects
  - Web indexing, Personalized Search, Google Earth, Google Analytics, Google Finance, …
- Flexible, high-performance solution for all of Google’s products

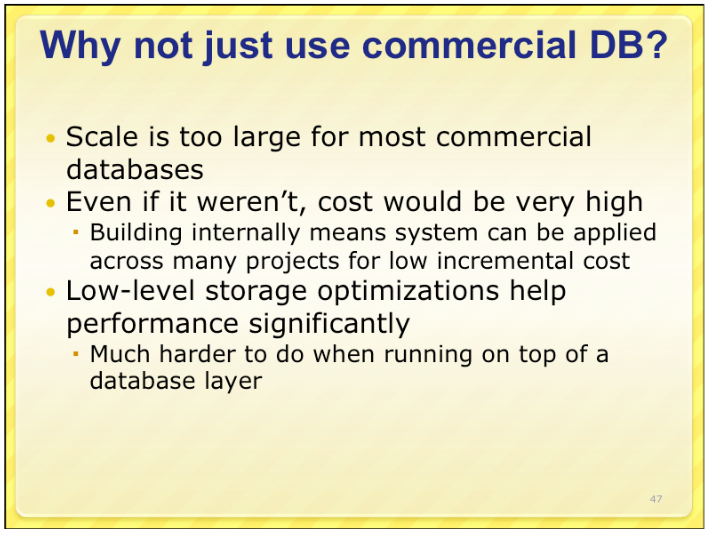
45



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 or q/sec
  - 100TB+ of satellite image data

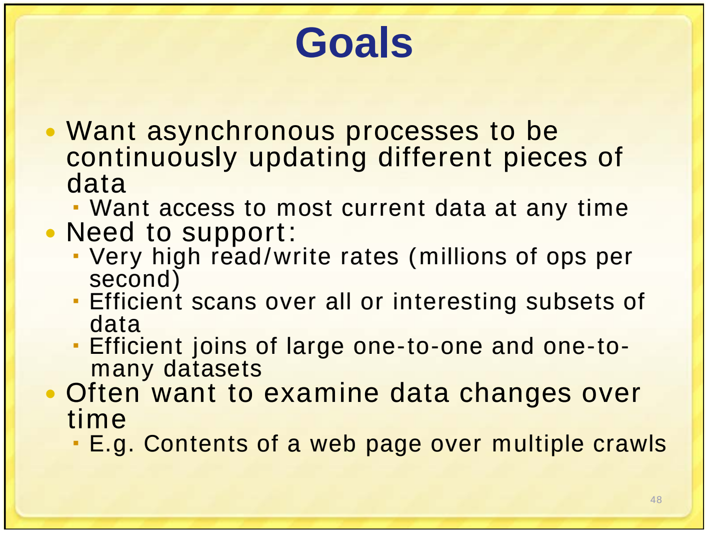
46



Why not just 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 for low incremental cost
- Low-level storage optimizations help performance significantly
  - Much harder to do when running on top of a database layer

47



Goals

- Want asynchronous processes to be continuously updating different pieces of data
  - Want access to most current data at any time
- Need to support:
  - Very high read/write rates (millions of ops per second)
  - Efficient scans over all or interesting subsets of data
  - Efficient joins of large one-to-one and one-to-many datasets
- Often want to examine data changes over time
  - E.g. Contents of a web page over multiple crawls

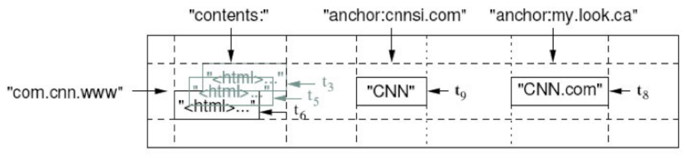
48

BigTable

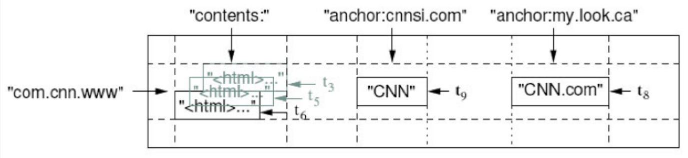
- Distributed multi-level map
- Fault-tolerant, persistent
- Scalable
  - Thousands of servers
  - Terabytes of in-memory data
  - Petabyte of disk-based data
  - Millions of reads/writes per second, efficient scans
- Self-managing
  - Servers can be added/removed dynamically
  - Servers adjust to load imbalance

Basic Data Model

- A BigTable is a sparse, distributed persistent multi-dimensional sorted map *(row, column, timestamp) -> cell contents*

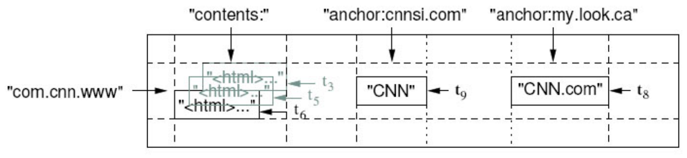


WebTable Example



- Want to keep copy of a large collection of web pages and related information
- Use URLs as row keys
- Various aspects of web page as column names
- Store contents of web pages in the `contents:` column under the timestamps when they were fetched.

Rows



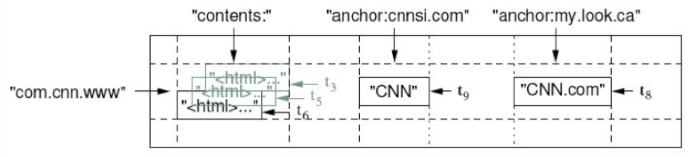
- Name is an arbitrary string
  - Access to data in a row is atomic
  - Row creation is implicit upon storing data
- Rows ordered lexicographically
  - Rows close together lexicographically usually on one or a small number of machines

Rows (cont.)

- Reads of short row ranges are efficient and typically require communication with a small number of machines.
- Can exploit this property by selecting row keys so they get good locality for data access.
- Example:

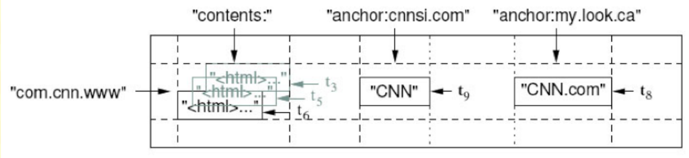
  math.gatech.edu, math.uga.edu, phys.gatech.edu, phys.uga.edu
  VS
  edu.gatech.math, edu.gatech.phys, edu.uga.math, edu.uga.phys

Columns



- Columns have two-level name structure:
  - family:optional_qualifier
- Column family
  - Unit of access control
  - Has associated type information
- Qualifier gives unbounded columns
  - Additional levels of indexing, if desired

Timestamps



- Used to store different versions of data in a cell
  - New writes default to current time, but timestamps for writes can also be set explicitly by clients
- Lookup options:
  - *"Return most recent K values"*
  - *"Return all values in timestamp range (or all values)"*
- Column families can be marked w/ attributes:
  - *"Only retain most recent K values in a cell"*
  - *"Keep values until they are older than K seconds"*

Implementation – Three Major Components

- Library linked into every client
- One master server
  - Responsible for:
    - Assigning tablets to tablet servers
    - Detecting addition and expiration of tablet servers
    - Balancing tablet-server load
    - Garbage collection
- Many tablet servers
  - Tablet servers handle read and write requests to its table
  - Splits tablets that have grown too large

Implementation (cont.)

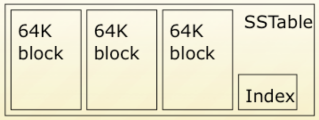
- Client data doesn’t move through master server. Clients communicate directly with tablet servers for reads and writes.
- Most clients never communicate with the master server, leaving it lightly loaded in practice.

Tablets

- Large tables broken into tablets at row boundaries
  - Tablet holds contiguous range of rows
    - Clients can often choose row keys to achieve locality
  - Aim for ~100MB to 200MB of data per tablet
- Serving machine responsible for ~100 tablets
  - Fast recovery:
    - 100 machines each pick up 1 tablet for failed machine
  - Fine-grained load balancing:
    - Migrate tablets away from overloaded machine
    - Master makes load-balancing decisions

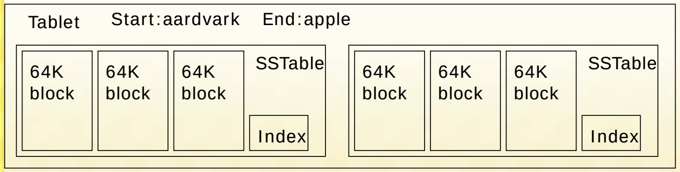
SSTable

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


64K block
64K block
64K block
SSTable
Index

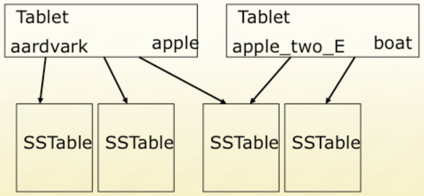
Tablet

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


Tablet Start:aardvark End:apple
64K block
64K block
64K block
SSTable
Index
64K block
64K block
64K block
SSTable
Index

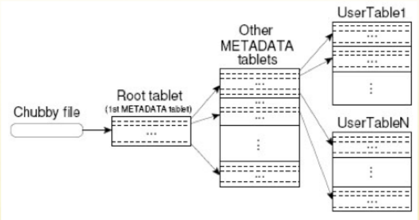
Table

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



Tablet Location

- 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

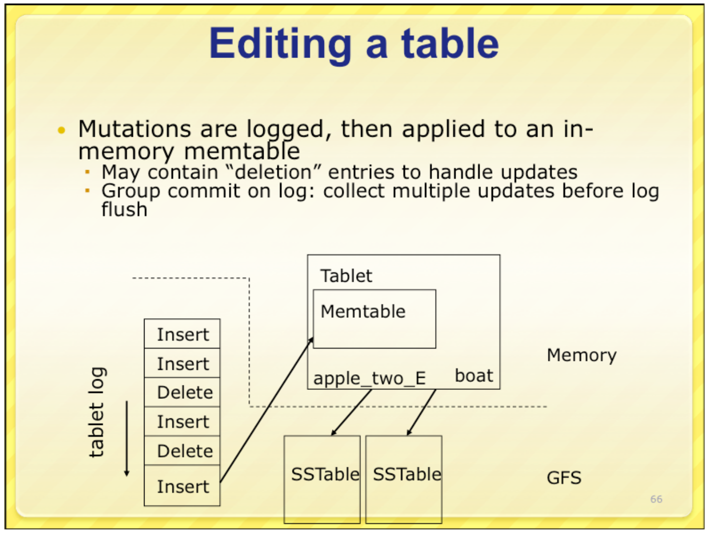


Chubby

- {lock/file/name} service
- Coarse-grained locks, can store small amount of data in a lock
- 5 replicas, need a majority vote to be active
- Also an OSDI '06 Paper

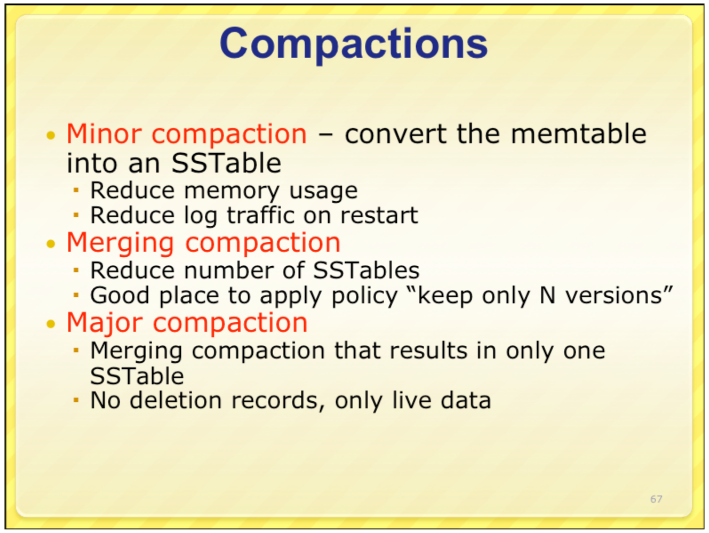
Servers

- 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



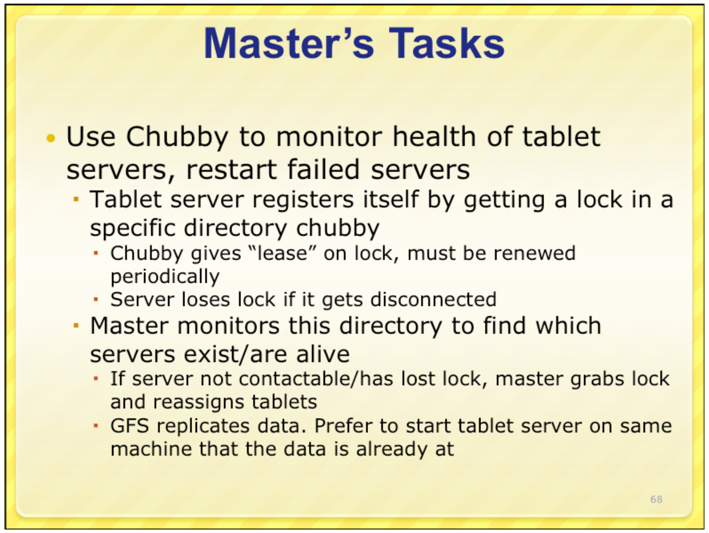
Editing a table

- 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



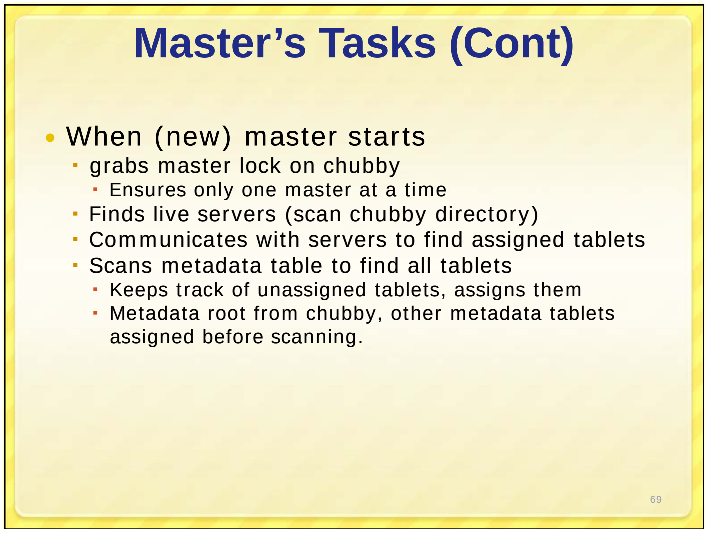
Compactions

- 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



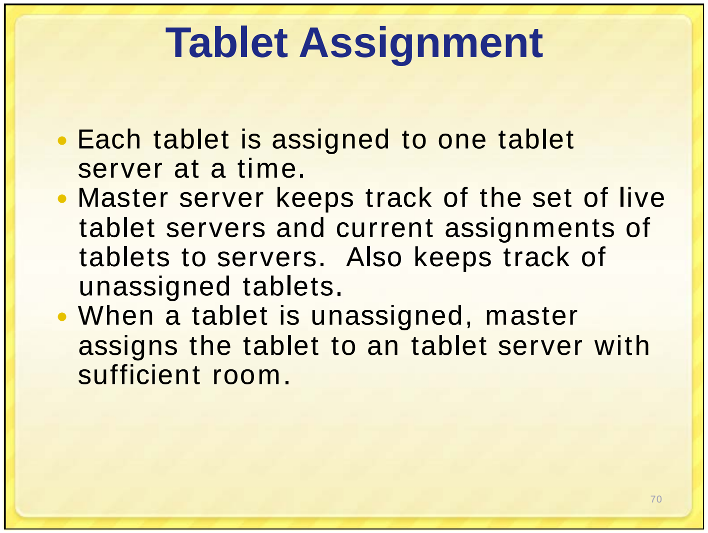
Master's Tasks

- 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.



Tablet Assignment

- Each tablet is assigned to one tablet server at a time.
- Master server keeps track of the set of live tablet servers and current assignments of tablets to servers. Also keeps track of unassigned tablets.
- When a tablet is unassigned, master assigns the tablet to an tablet server with sufficient room.