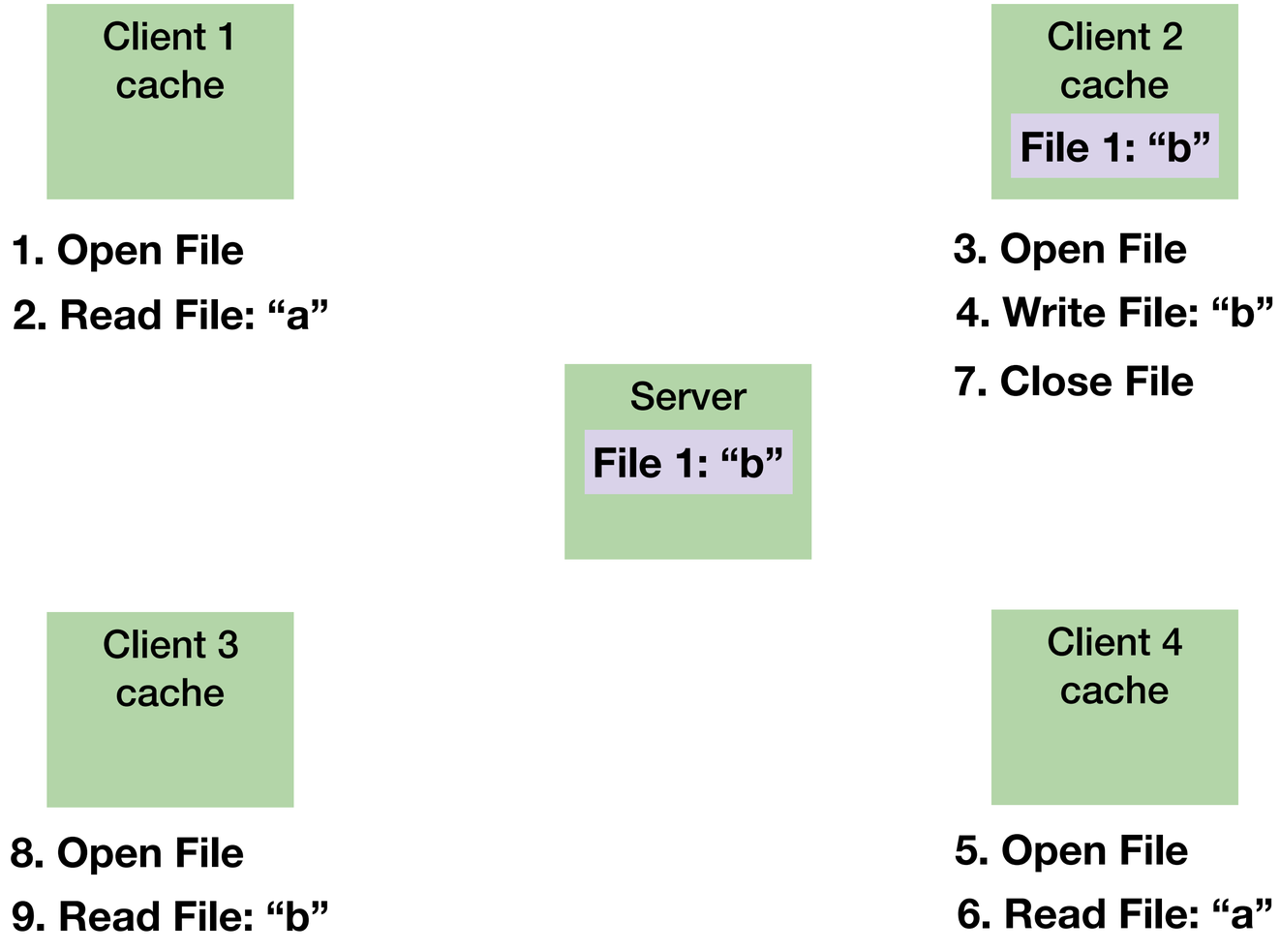
GFS + MapReduce

CS 475, Fall 2019
Concurrent & Distributed Systems

GEORGE MASON UNIVERSITY

NFS Caching - Close-to-open



Note: in practice, client caches periodically check server to see if still valid

NFS Limitations

- Security: what if untrusted users can be root on client machines?
- Scalability: how many clients can share one server?
  - Writes always go through to server
  - Some writes are to “private,” unshared files that are deleted soon after creation
- Can you run NFS on a large, complex network?
  - Effects of latency? Packet loss? Bottlenecks?
- Important question: whose fault are these limitations? Are they intractable (because of the very problem we are trying to solve)? Or are we just not thinking hard enough?

Today

- Today:
  - Big data, big problems
- Additional readings for reference:
  - GFS, MapReduce papers, Podcast about Dropbox
- Project is out next week!
  - Fault-tolerant, sequentially consistent replicated key value store
  - Start thinking of groups (1 to **3** students per group)

More data, more problems

- I have a 1TB file
- I need to sort it
- …My computer can only read 60MB/sec
- …
- …
- …
- 1 day later, it's done

More data, more problems

- Think about scale:
  - Google indexes ~20 petabytes of web pages per **day** (as of 2008!)
  - Facebook has 2.5 petabytes of user data, increases by 15 terabytes/day (as of 2009!)

Distributing Computation

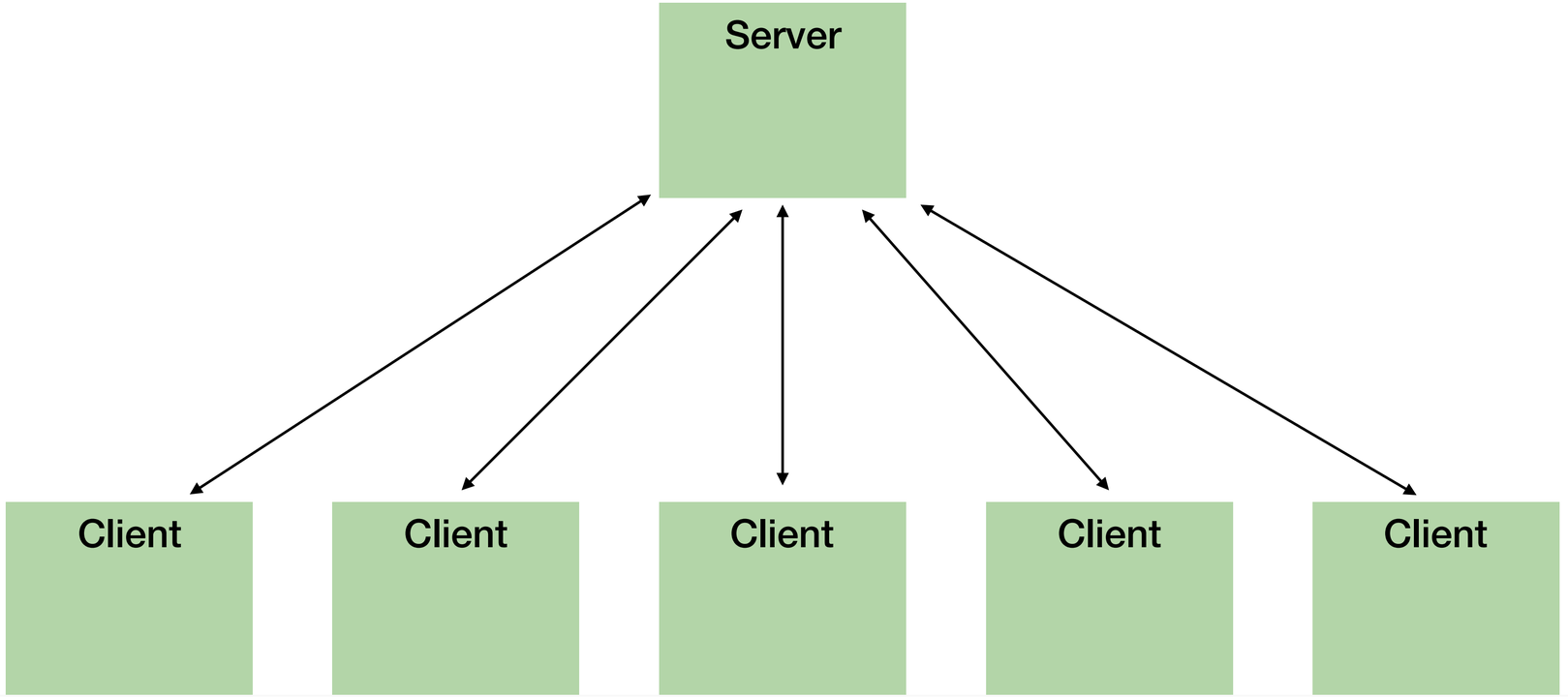
Distributing Computation

- Can't I just add 100 nodes and sort my file 100 times faster?
- Not so easy:
  - Sending data to/from nodes
  - Coordinating among nodes
  - Recovering when one node fails
  - Optimizing for locality
  - Debugging

Distributing Computation

- We begin to answer
  - 1. How do we store the data?
  - 2. How do we compute on this data?

NFS to the Rescue?



All files stored on the same server is bad because:
Fault tolerance (what if it crashes?)
Performance (what if we need to access 100's of GBs at a time?)
Scale (what if we need to store PBs of files?)
Plus, NFS' open-to-close caching can be weird

GFS (Google File System)

- Google apps observed to have specific R/W patterns (usually read recent data, lots of data, etc)
- Normal FS API (POSIX) is constraining (consider: NFS contains a ton of annoying glue to make it work with open/close/sync/seek etc)
- Hence, Google made their own FS

GFS

- Hundreds of thousands of regular servers
- Millions of regular disks
- Failures are normal
  - App bugs, OS bugs
  - Human Error
  - Disk failure, memory failure, network failure, etc
- Huge number of concurrent reads, writes

GFS Workload

- (Relatively) small total number of large files (>100MB) - millions
- Large, streaming reads (reading > 1MB at a time)
  - Throughput is more important than latency
- Large, sequential writes that always append to end of a file
  - Optimize for appends
- Multiple clients might append concurrently

GFS Design Goals

- Unified FS for all google platforms (e.g. gmail, youtube)
- Data + system availability
- Graceful + transparent failure handling
- Low synchronization overhead
- Exploit parallelism
- High throughput and low latency

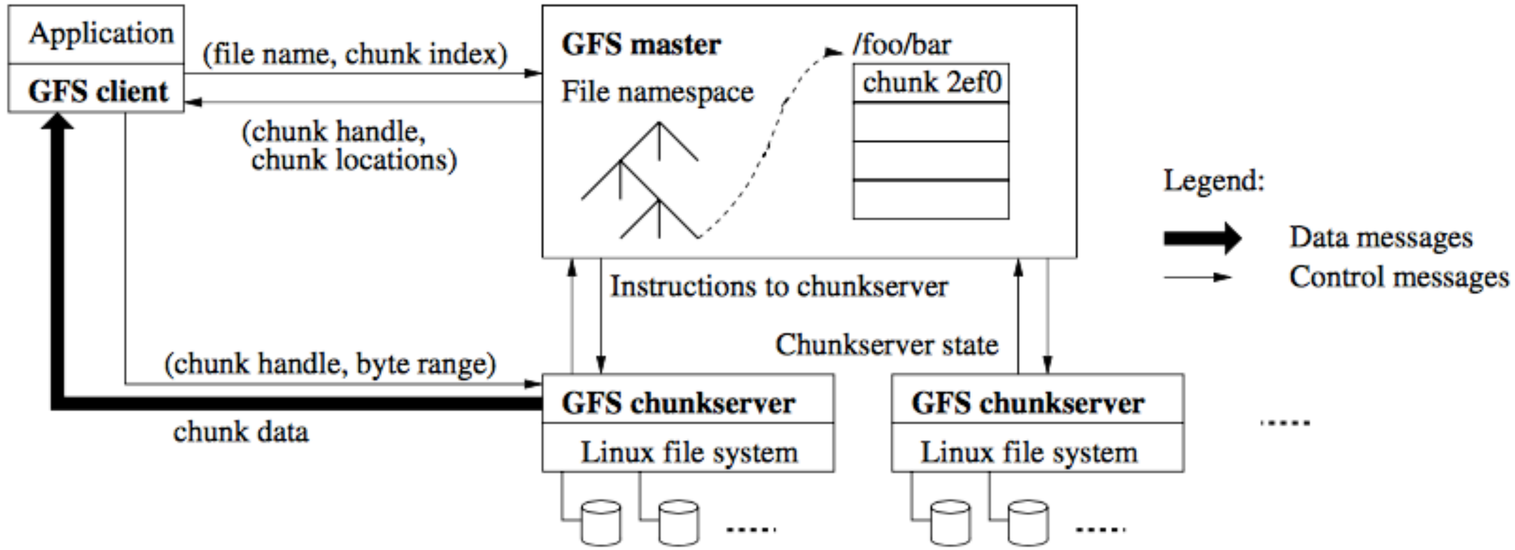
GFS Interface Design

- Non-standard API (e.g. not POSIX system calls)
- Normal filesystem hierarchy (directories, paths)
- Extra operations:
  - Snapshot (low cost file/directory tree copying)
  - Record append (append without locking)

GFS Architecture

- Servers:
  - Single master
  - Multiple backups (“chunk servers”)
- Base unit is a “Chunk”
  - Fixed-part of a file (typically 64MB)
  - Global ID: 64 bit unique ID, assigned by master upon creation
  - Read/write: specify chunk ID + byte range into file
  - Each chunk is replicated to at least 3 servers
  - Chunks are stored as plain files on chunk servers

GFS Architecture



GFS Architecture

- Single master server (can replicate to a backup too)
- Holds all metadata (in RAM!) - namespace, ACL, file-chunk mapping
  - In charge of migrating chunks, GC'ing chunks
- Data stored in 64MB chunks each with some ID
  - Compare to EXT-4's 4KB block
- Thousands of chunk servers
  - Chunks are replicated
  - Chunk servers don't cache anything in RAM, store chunks as regular files

GFS Architecture

GFS Metadata

- Master server is in charge of metadata
- File-chunk mapping
  - Stored on disk
- For each chunk:
  - Location of replicas holding the chunk
  - Identity of the master chunk server for this chunk
  - Stored ONLY in RAM - if crash, recover by asking all chunk servers

GFS Metadata Example

Chunk ID	Filename	Part of file	Master Chunk Server	Other Chunk Servers
1	/foo/bar	1 of 1	A, valid for 1 more minute	B, C
2	/another/file	1 of 2	B, valid for 1 more minute	A, C
3	/another/file	2 of 2	D, valid for 1 more minute	C, E

Note - can get very good parallelism by splitting chunks of the same file across different chunk servers

GFS Metadata

- File-chunk mapping stored on disk
- Chunk-chunk server mapping stored only in RAM: why?
  - If stored on disk, would need to update every time a chunk server comes/leaves, OR new chunk comes (remember: optimize for appends)
  - If stored on disk, would need to ensure that each update is flushed to disk to ensure consistency if master fails (and RAM is lost)
  - Hence: only store in RAM, recover from other chunk servers

GFS Client

- Makes metadata requests to master server
- Makes chunk requests to chunk servers
- Caches metadata
- Does not cache data (chunks)
  - Google's workload (streaming reads, appending writes) doesn't benefit from caching, so why bother with consistency nightmare

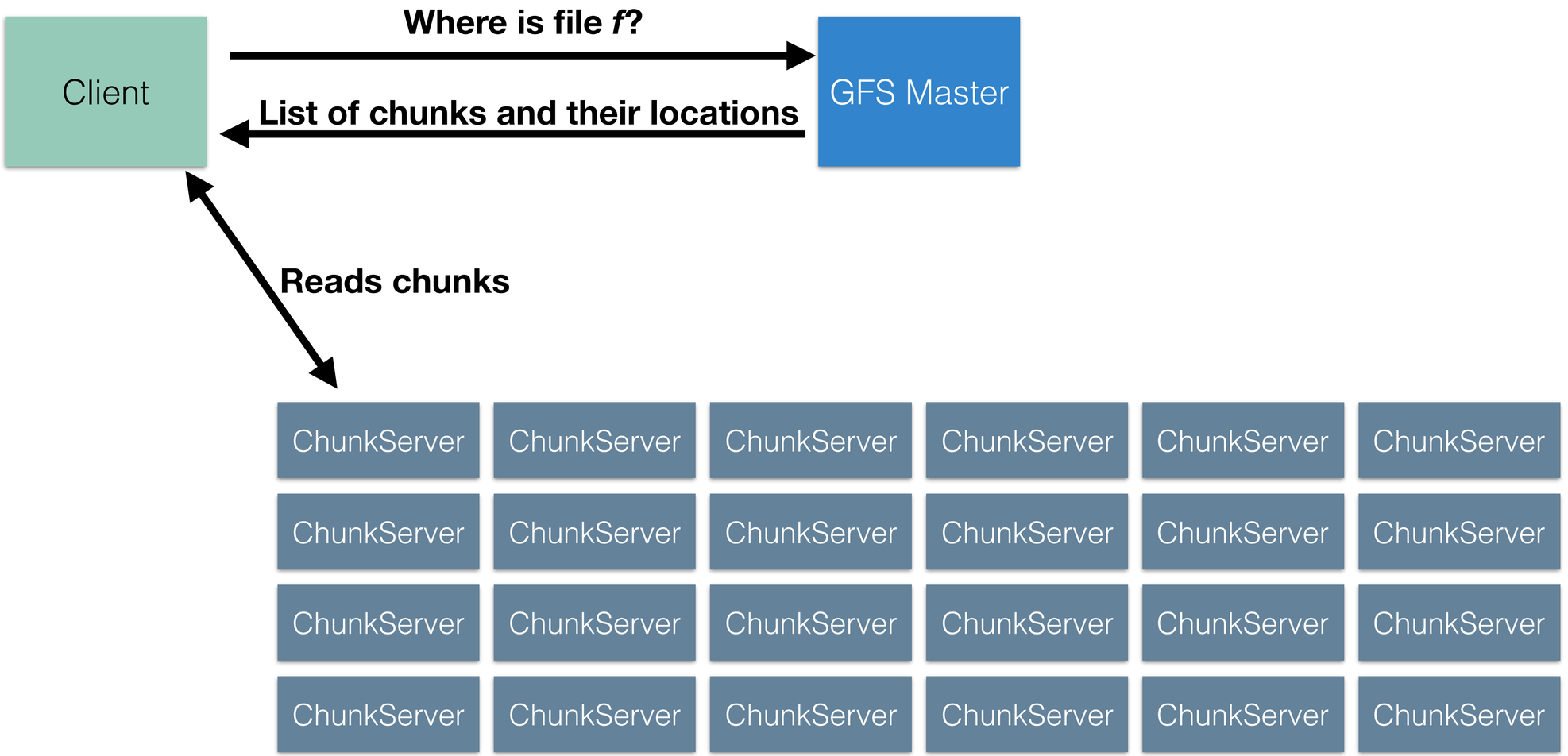
GFS Chunk Primaries

- There needs to be exactly one primary for each chunk
- GFS ensures this using *leases*
  - Master selects a chunk server and grants it a lease
  - The chunk server holds the lease for T seconds, and is primary
  - Chunk server can *refresh* lease endlessly
  - If chunk server fails to refresh it, falls out of being primary
- Like a lock, but needs to be renewed (like with a heart beat)

GFS Reads

- Client asks master for chunk ID, chunk version number, and location of replicas given a file name
- By default, GFS replicates each chunk to 3 servers
- Client sends read request to closest (in network topology) chunk server

GFS - Reads


Client
Where is file *f*?
GFS Master
List of chunks and their locations
Reads chunks
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer
ChunkServer

GFS Writes

- Client asks master for replicas storing a chunk (one is arbitrarily declared primary)
- Client sends write request to all replicas
- Each replica acknowledges write to primary replica
- Primary coordinates commit between all of the replicas
- On success, primary replies to client

GFS Consistency

- Metadata changes are atomic. Occur only on a single machine, so no distributed issues.
- Changes to data are ordered as arbitrarily chosen by the primary chunk server for a chunk

Comparing GFS/NFS Issues

- Fault tolerance (what if it crashes?)
  - NFS: Crashing the server is bad
  - GFS: Crashing a chunk server is fine, crashing the primary is bad
- Performance (what if we need to access 100's of GBs at a time?)
  - NFS: Limited by single server's bandwidth
  - GFS: Limited only by number of chunk servers (can get good parallelism between 100 chunk servers each at 1GB/sec)
- Scale (what if we need to store PBs of files?)
  - NFS: Limited by storage of single server
  - GFS: Limited by amount of metadata that can be stored on single master
- Plus, NFS' open-to-close caching can be weird
  - GFS: No caching

GFS Summary

- Much more attractive than NFS for reading/writing large files
- Limitations:
  - Master is a huge bottleneck
  - Recovery of master is slow
- Lots of success at Google
- Performance isn't great for all apps (lots of small files?)
- Consistency needs to be managed by apps
- Replaced in 2010 by Google's Colossus system - eliminates master

Distributing Computation

- Lots of these challenges re-appear, regardless of our specific problem
  - How to split up the task
  - How to put the results back together
  - How to store the data (GFS)
- Enter, MapReduce

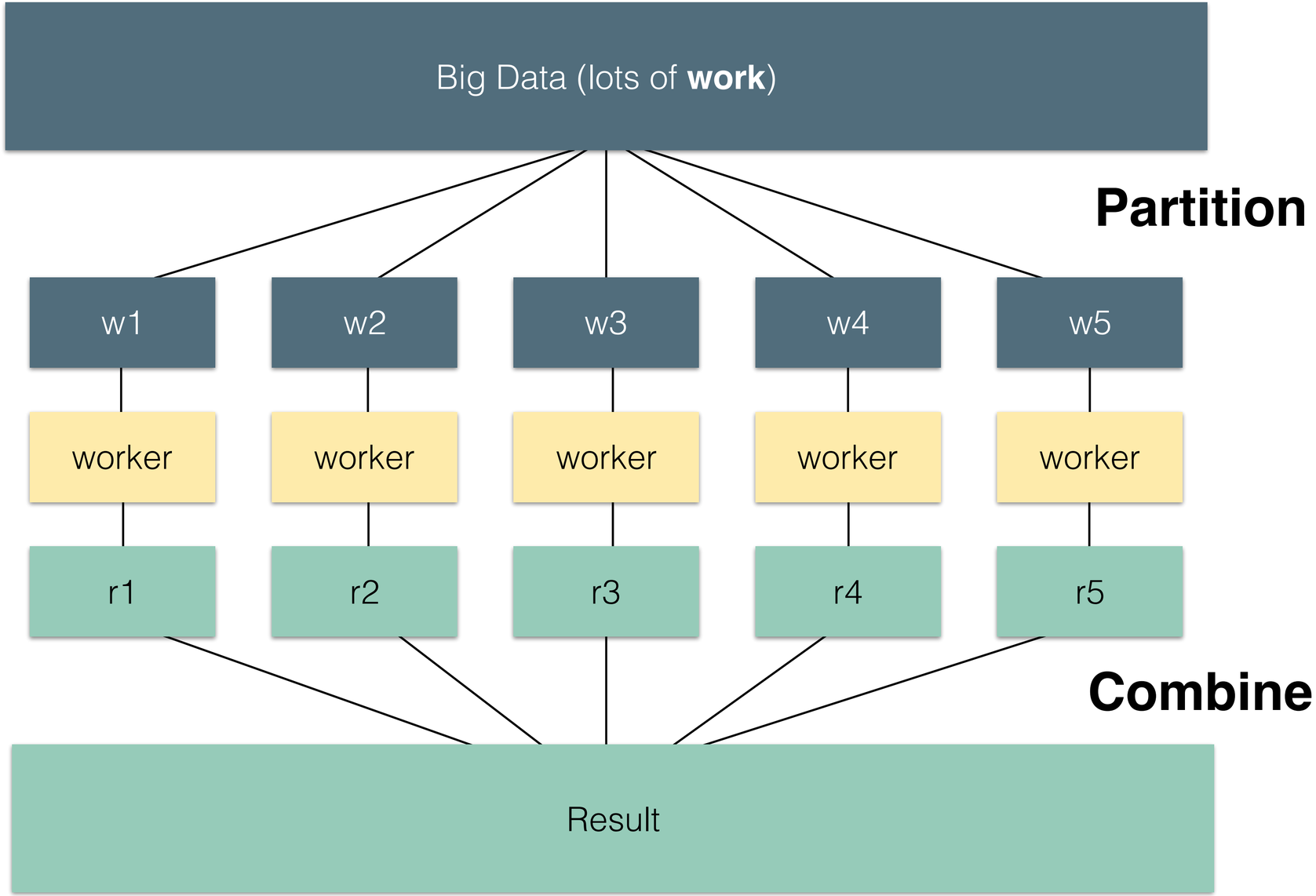
MapReduce

- A programming model for large-scale computations
  - Takes large inputs, produces output
  - No side-effects or persistent state other than that input and output
- Runtime library
  - Automatic parallelization
  - Load balancing
  - Locality optimization
  - Fault tolerance

MapReduce

- Partition data into splits (**map**)
- Aggregate, summarize, filter or transform that data (**reduce**)
- Programmer provides these two methods

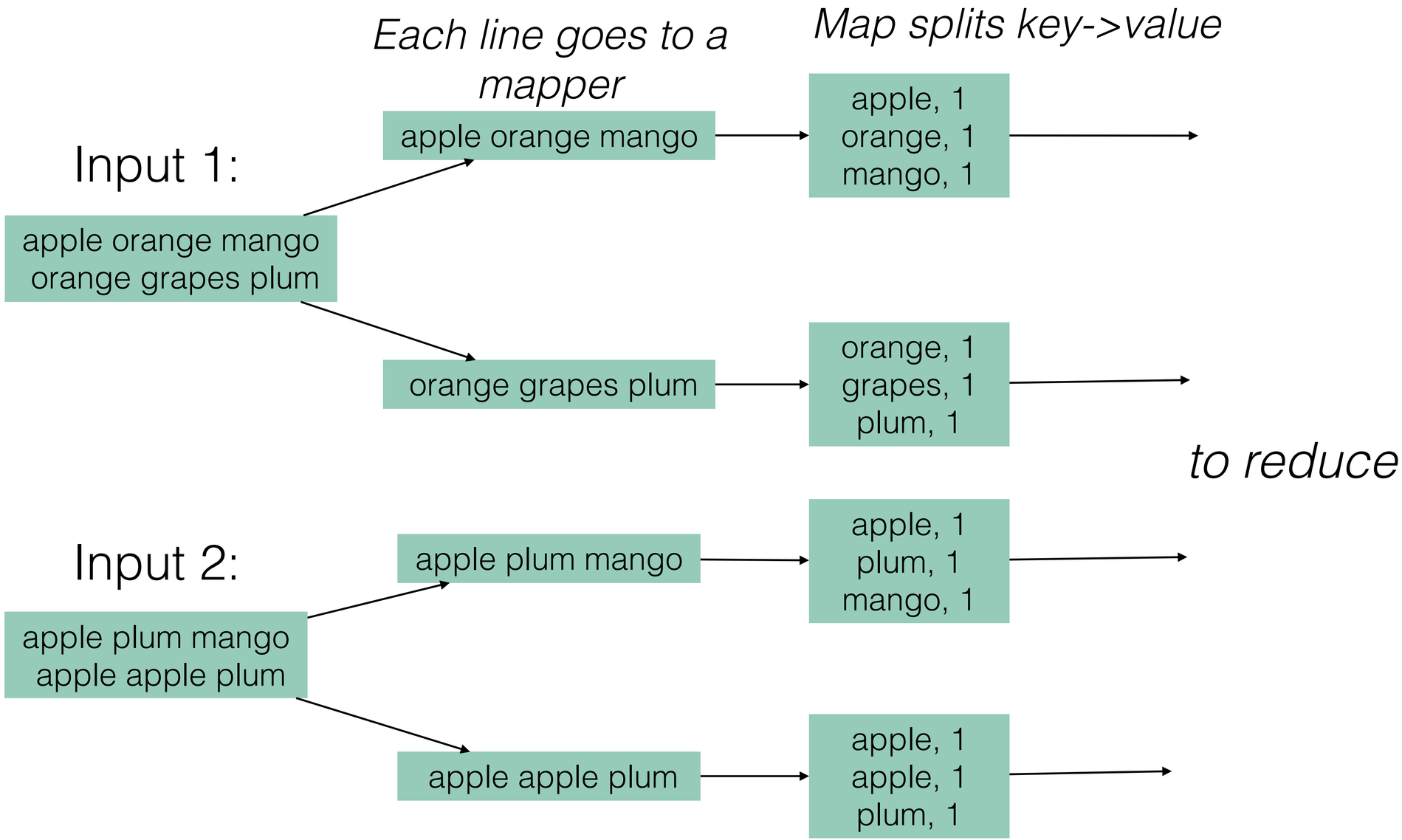
MapReduce: Divide & Conquer



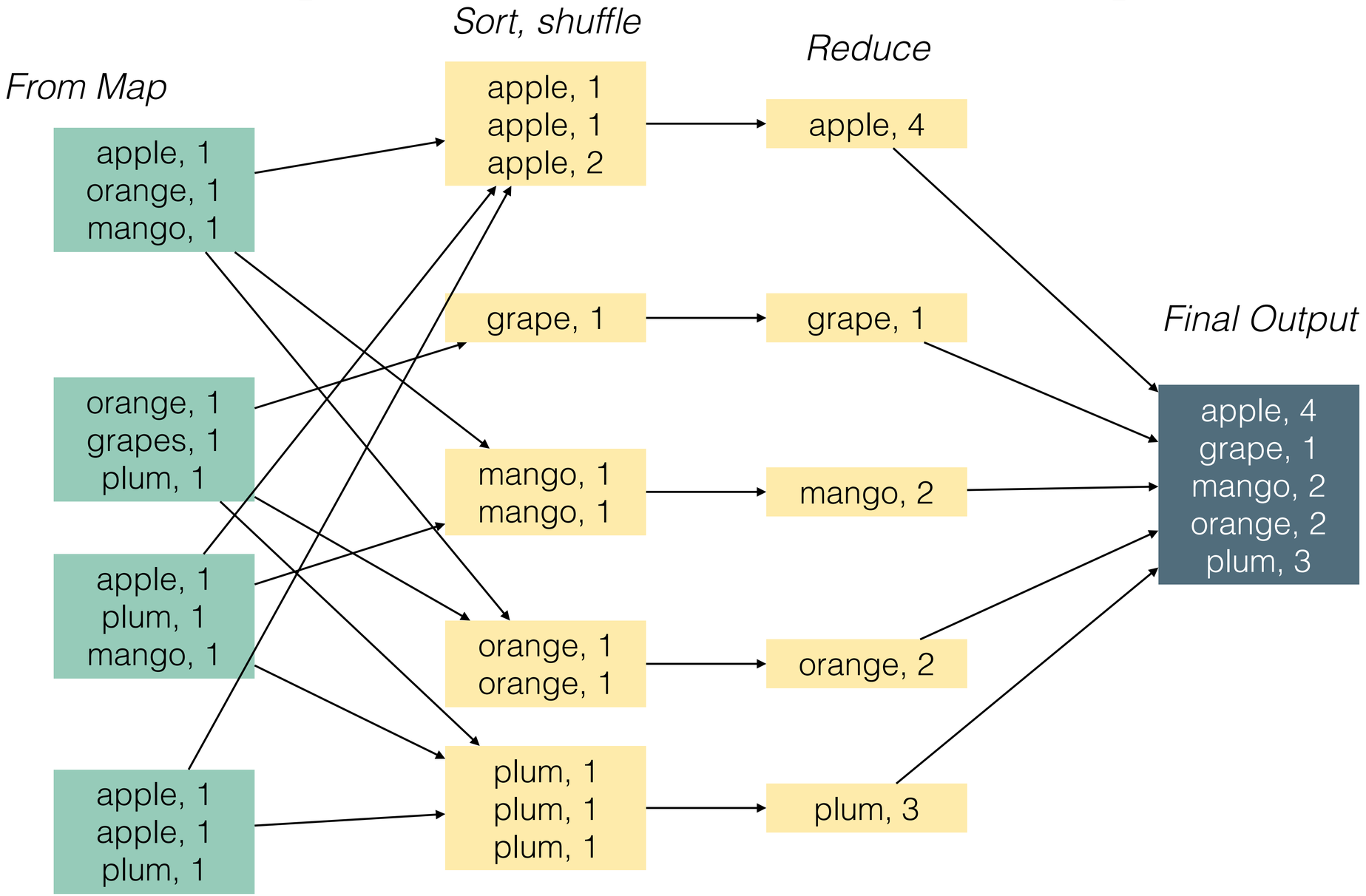
MapReduce: Example

- Calculate word frequencies in documents
- Input: files, one document per record
- **Map** parses documents into words
  - Key - Word
  - Value - Frequency of word
- **Reduce**: compute sum for each key

MapReduce: Example



MapReduce: Example

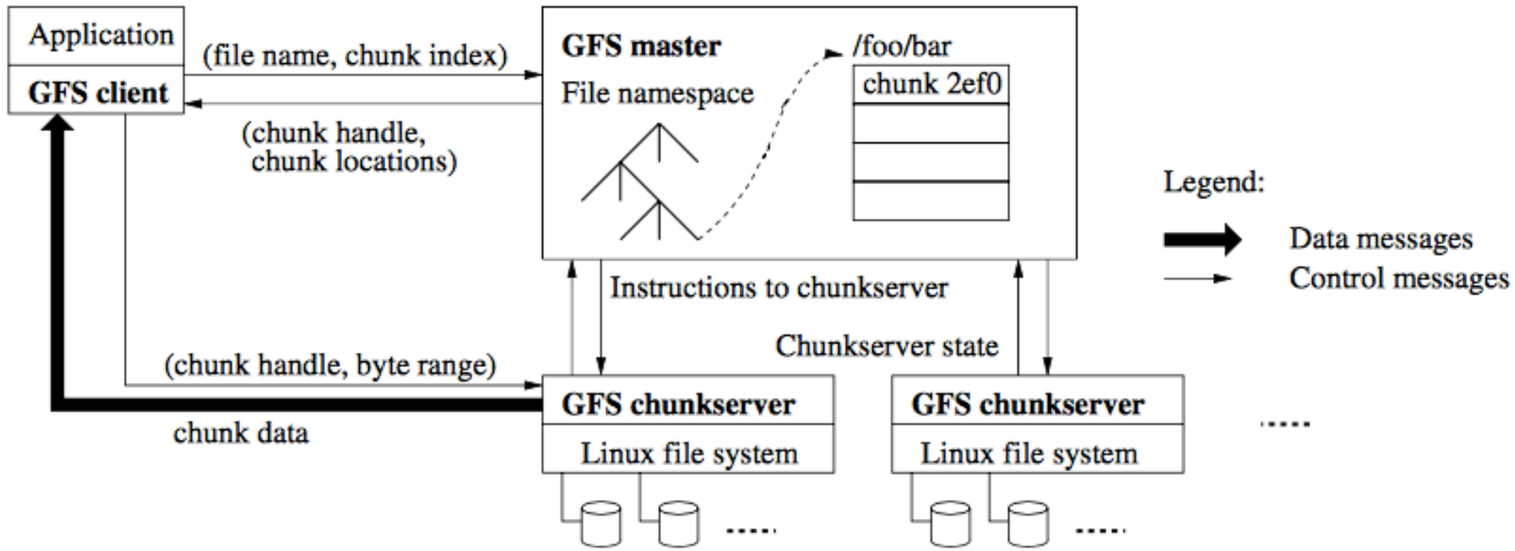


MapReduce Applications

- Distributed grep
- Distributed clustering
- Web link graph traversal
- Detecting duplicate web pages

MapReduce: Implementation

- Each worker node is **also** a GFS chunk server!



MapReduce: Scheduling

- One master, many workers
- Input data split into *M* map tasks (typically 64MB ea)
- *R* reduce tasks
- Tasks assigned to works dynamically; stateless and idempotent -> easy fault tolerance for workers
- Typical numbers:
  - 200,000 map tasks, 4,000 reduce tasks across 2,000 workers

MapReduce: Scheduling

- Master assigns map task to a free worker
  - Prefer "close-by" workers for each task (based on data locality)
  - Worker reads task input, produces intermediate output, stores locally (K/V pairs)
- Master assigns reduce task to a free worker
  - Reads intermediate K/V pairs from map workers
  - Reduce worker sorts and applies some *reduce* operation to get the output

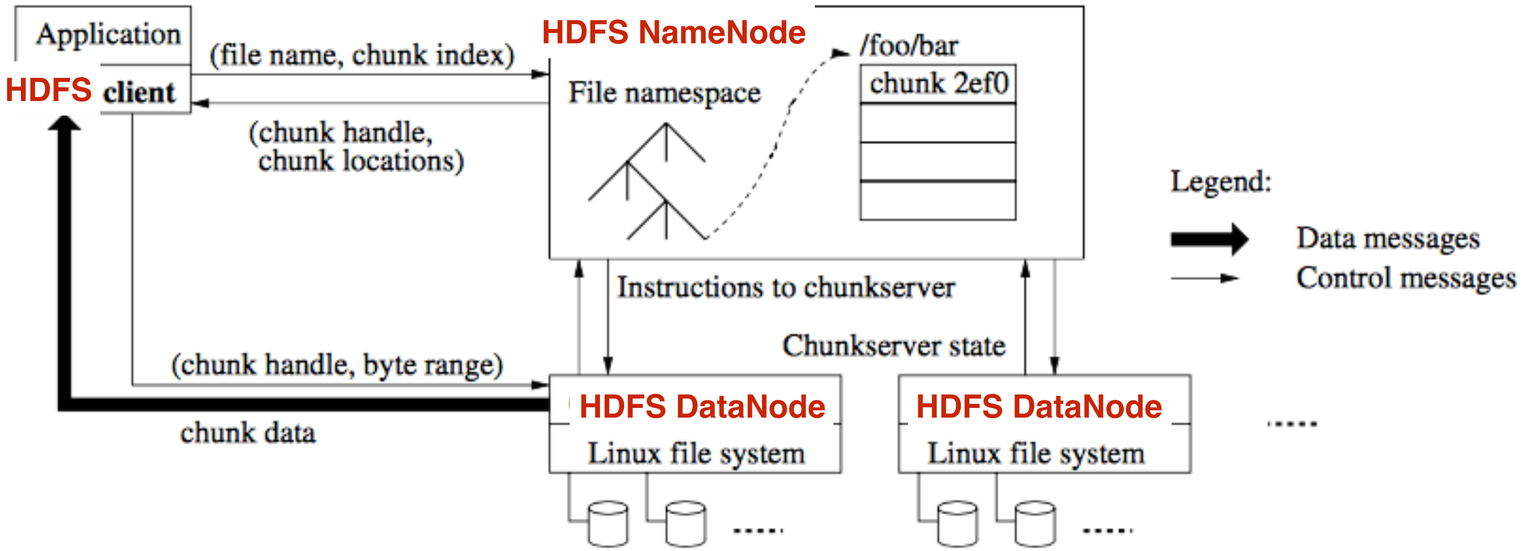
Fault tolerance via re-execution

- Ideally, fine granularity tasks (more tasks than machines)
- On worker-failure:
  - Re-execute completed and in-progress map tasks
  - Re-executes in-progress reduce tasks
  - Commit completion to master
- On master-failure:
  - Recover state (master checkpoints in a primary-backup mechanism)

MapReduce in Practice

- Originally presented by Google in 2003
- Widely used today (**Hadoop** is an open source implementation)
- Many systems designed to have easier programming models that compile into MapReduce code (Pig, Hive)

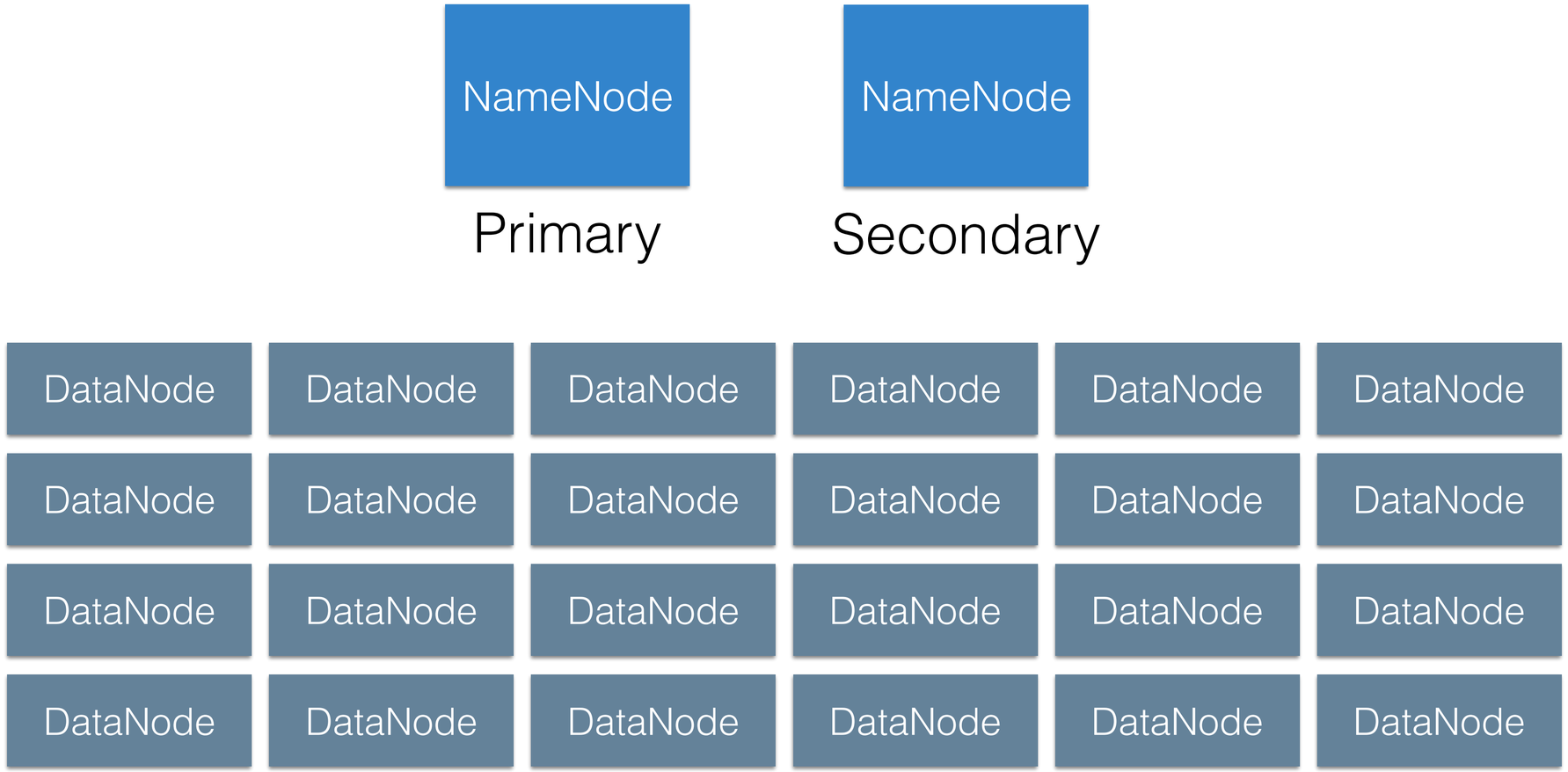
Hadoop: HDFS


Application
HDFS client
(file name, chunk index)
(chunk handle, chunk locations)
HDFS NameNode
File namespace
/foo/bar
chunk 2ef0
Instructions to chunkserver
Chunkserver state
(chunk handle, byte range)
chunk data
HDFS DataNode
Linux file system
HDFS DataNode
Linux file system
Legend:
Data messages
Control messages

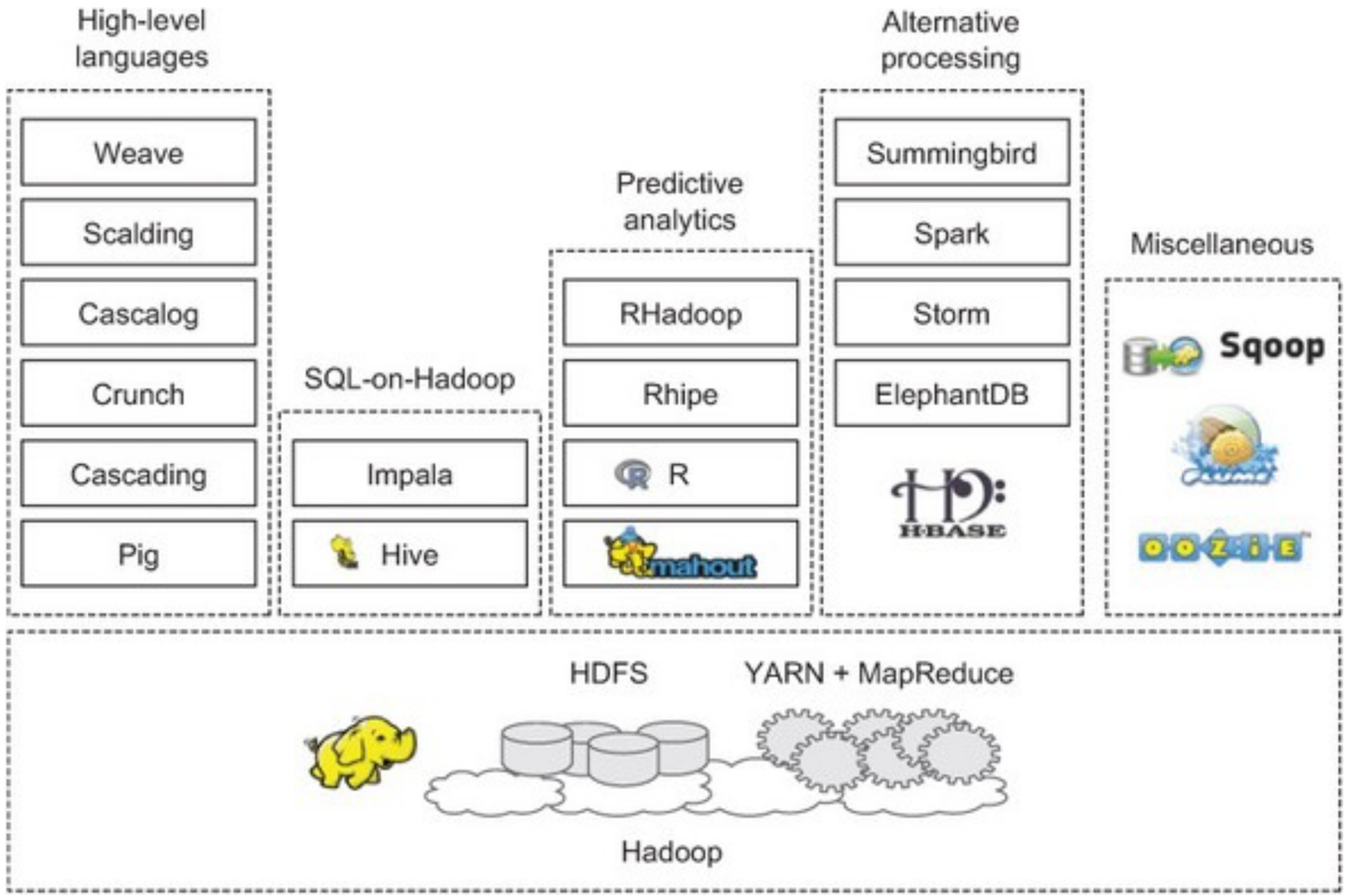
HDFS (GFS Review)

- Files are split into blocks (128MB)
- Each block is replicated (default 3 block servers)
- If a host crashes, all blocks are re-replicated somewhere else
- If a host is added, blocks are rebalanced
- Can get awesome locality by pushing the map tasks to the nodes with the blocks (just like MapReduce)

Hadoop


NameNode
NameNode
Primary
Secondary
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode
DataNode

Hadoop Ecosystem



This work is licensed under a Creative Commons Attribution-ShareAlike license