CS 471 Operating Systems

Yue Cheng

George Mason University

Spring 2019

Google File System
MapReduce
Key-Value Store

Google File System

MapReduce

Key-Value Store

Google File System (GFS) Overview

- Motivation
- Architecture

GFS

- Goal: a global (distributed) file system that stores data across many machines
  – Need to handle 100’s TBs
- Google published details in 2003
- Open source implementation:
  – Hadoop Distributed File System (HDFS)

Workload-driven Design

- Google workload characteristics
  - Huge files (GBs)
  - Almost all writes are appends
  - Concurrent appends common
  - High throughput is valuable
  - Low latency is not

Example Workloads

- Read entire dataset, do computation over it
  - Batch processing

- Producer/consumer: many producers append work to file concurrently; one consumer reads and does work

Workload-driven Design

- Build a global (distributed) file system that incorporates all these application properties

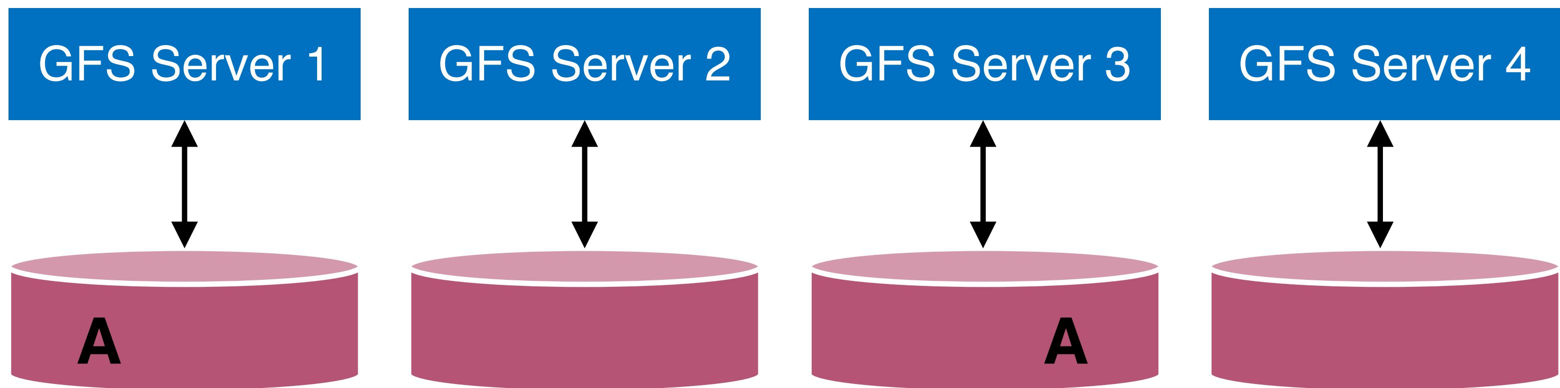
- Only supports features required by applications

- Avoid difficult local file system features, e.g.:
  - rename dir
  - links

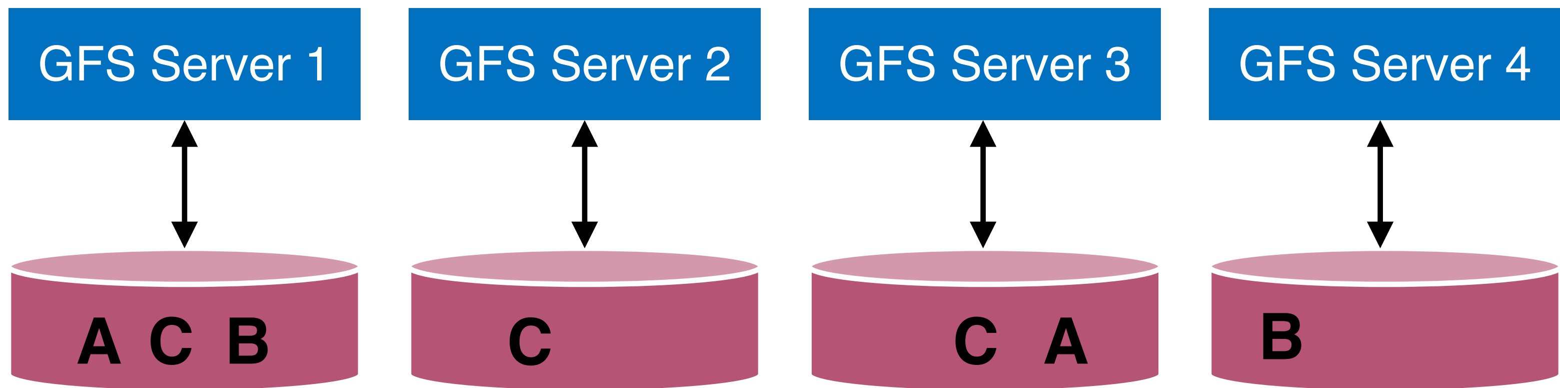
Google File System (GFS) Overview

- Motivation
- Architecture

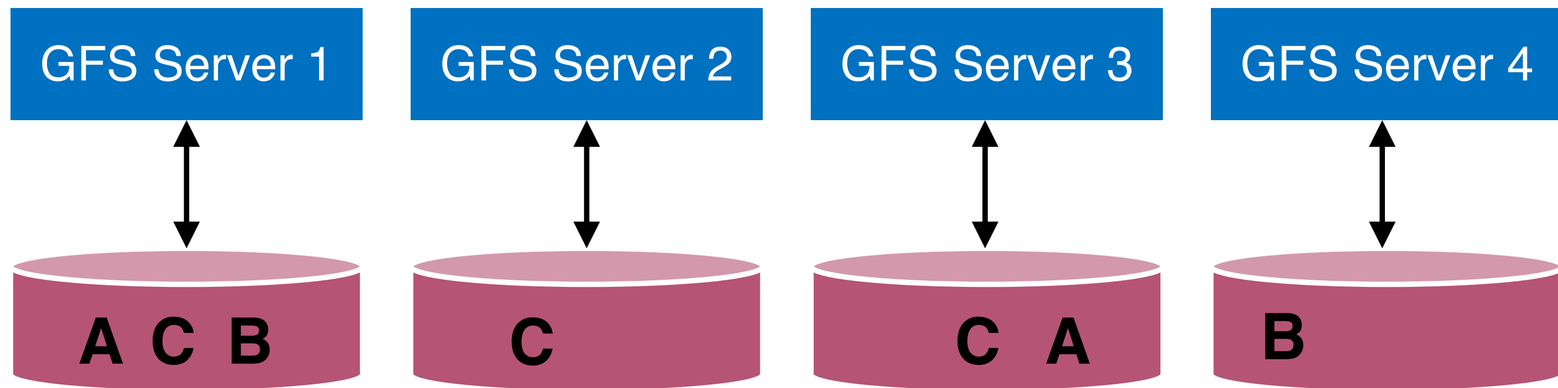
Replication



Replication



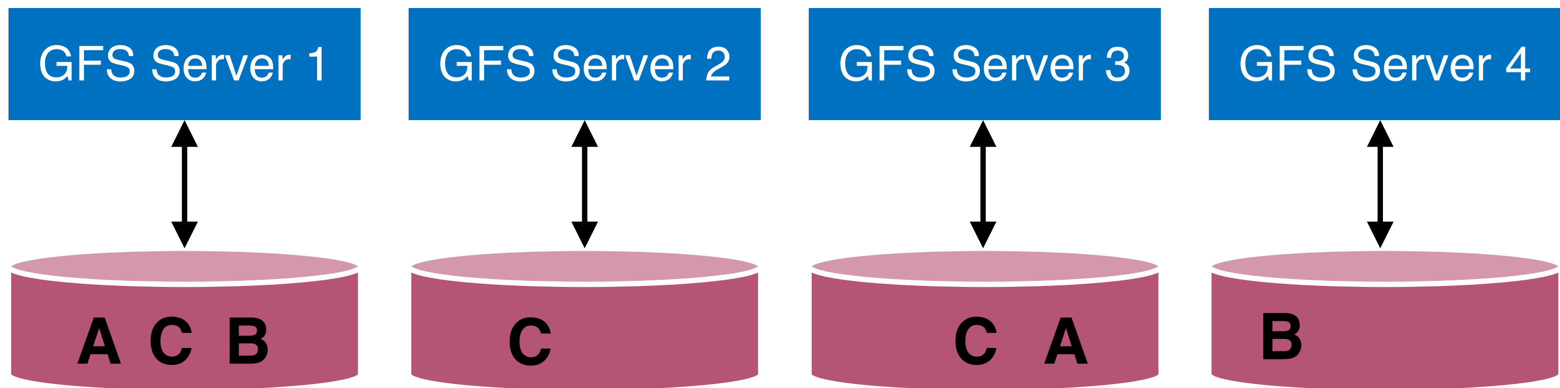
Replication



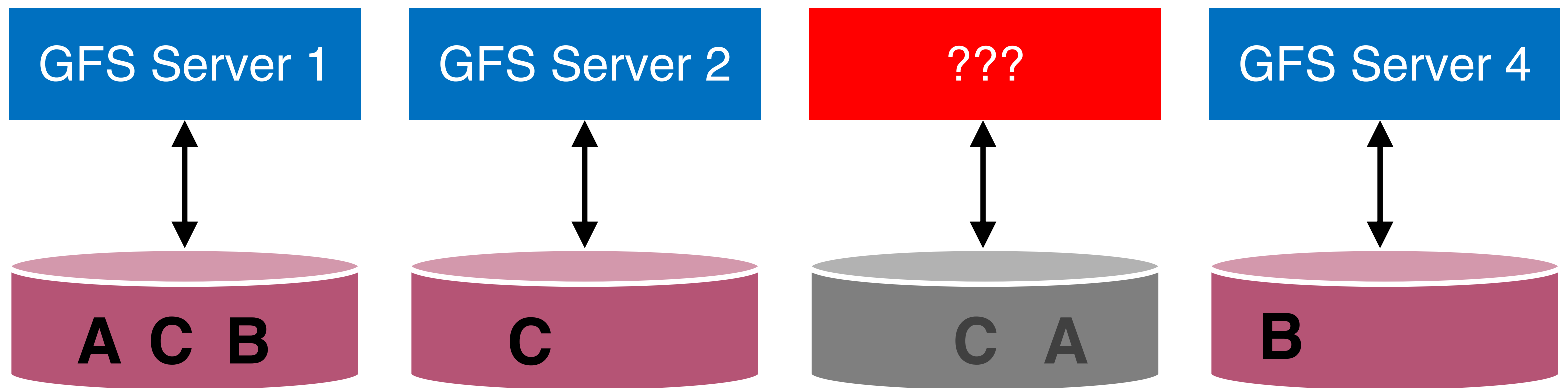
Similar to RAID, but less orderly than RAID

- Machines' capacity may vary (resource heterogeneity)
- Different data may have different replication factors (application-driven)

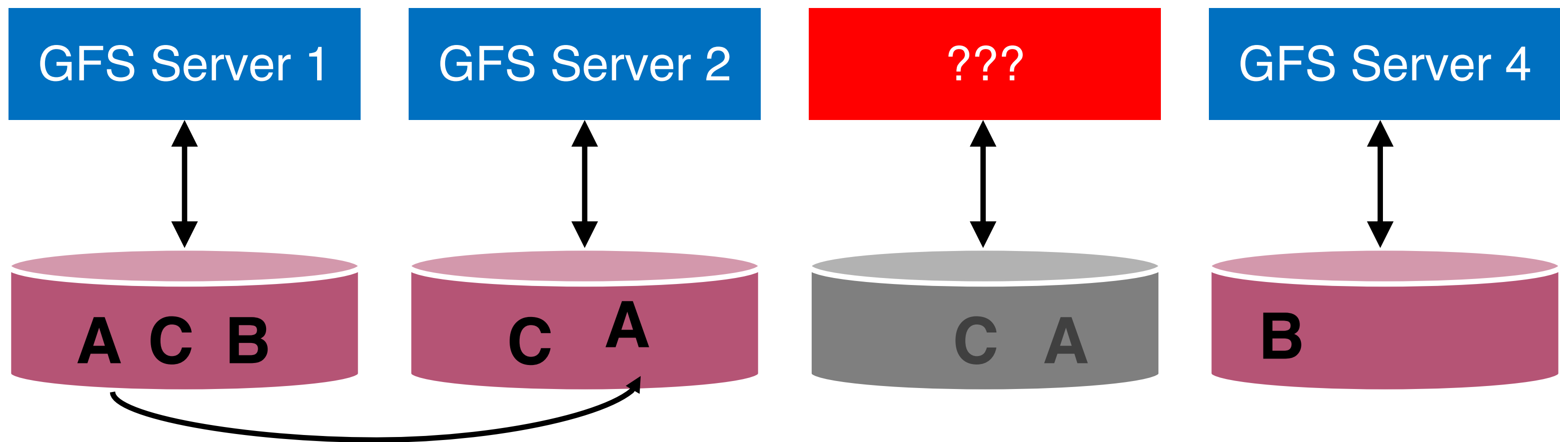
Data Recovery



Data Recovery

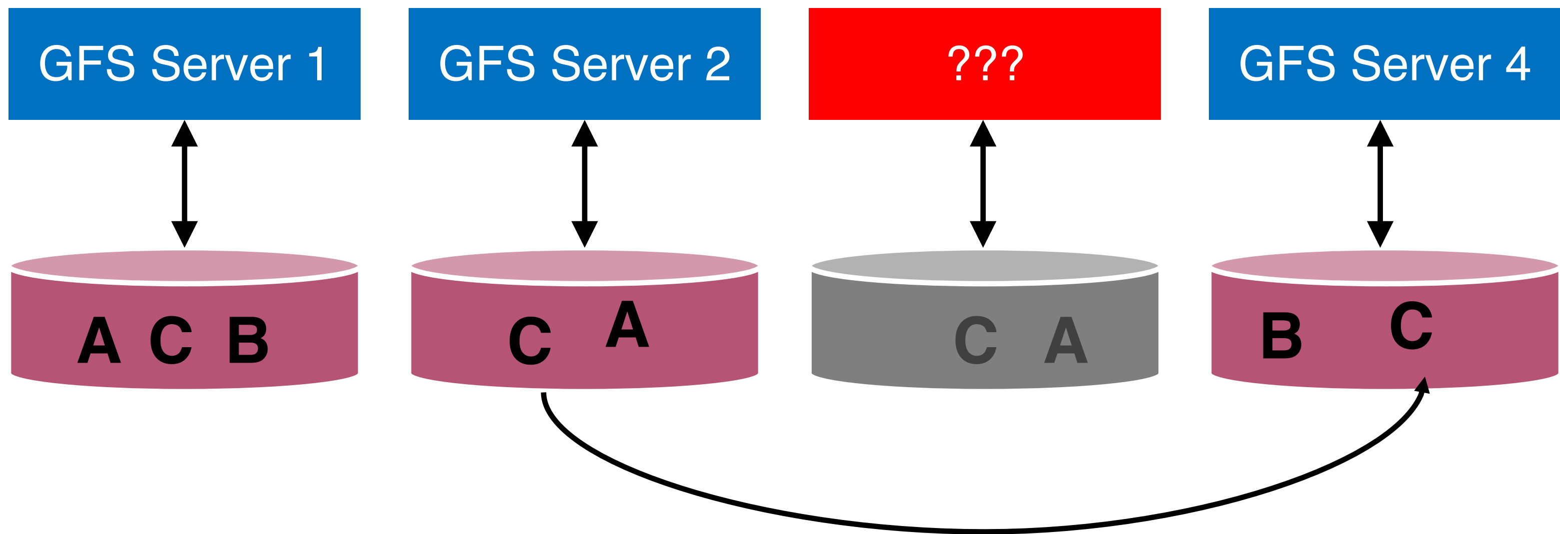


Data Recovery



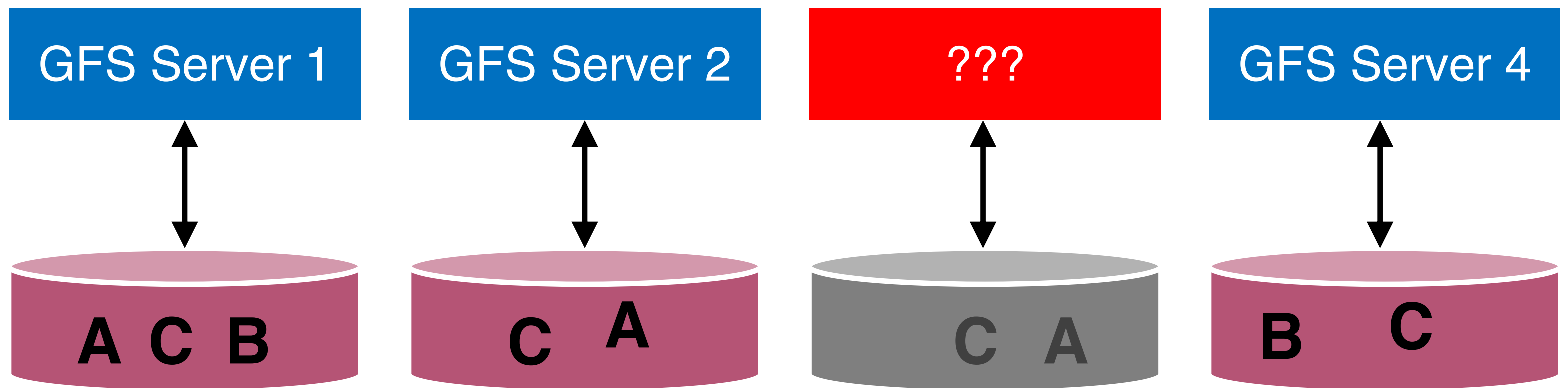
Replicating A to maintain a replication factor of 2

Data Recovery



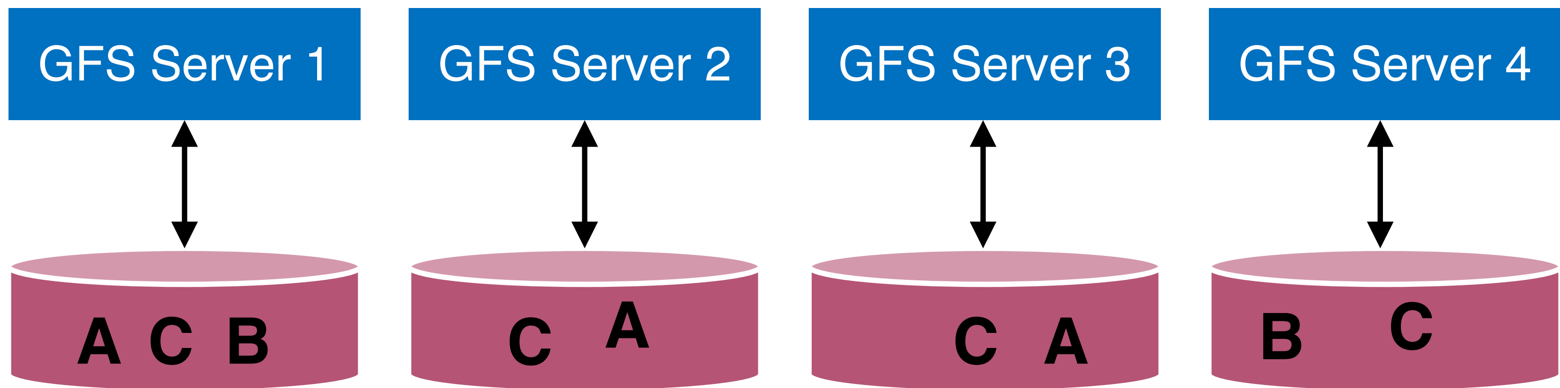
Replicating C to maintain a replication factor of 3

Data Recovery



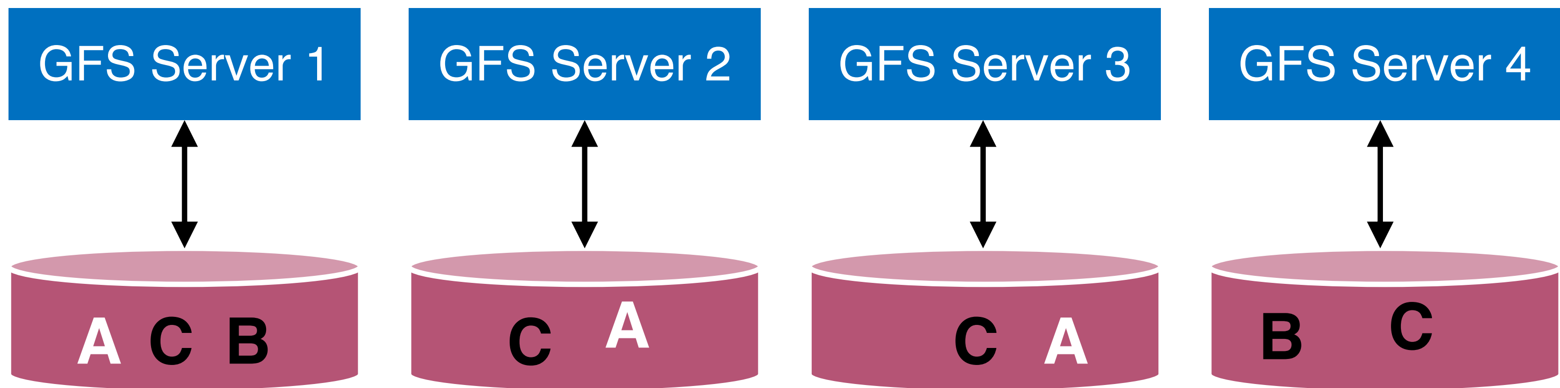
Machine may be dead forever, or it may come back

Data Recovery

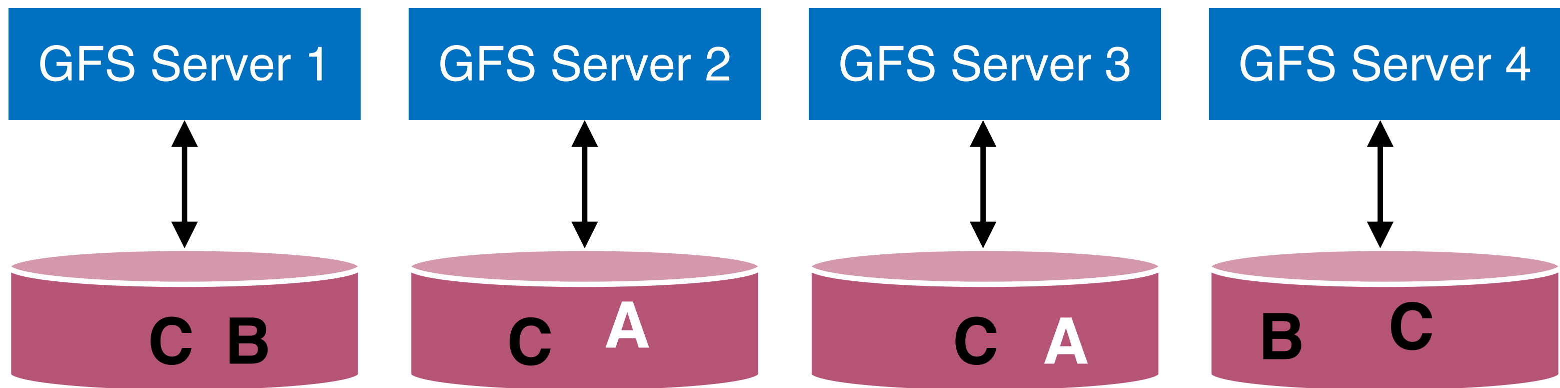


Machine may be dead forever, or it may come back

Data Recovery



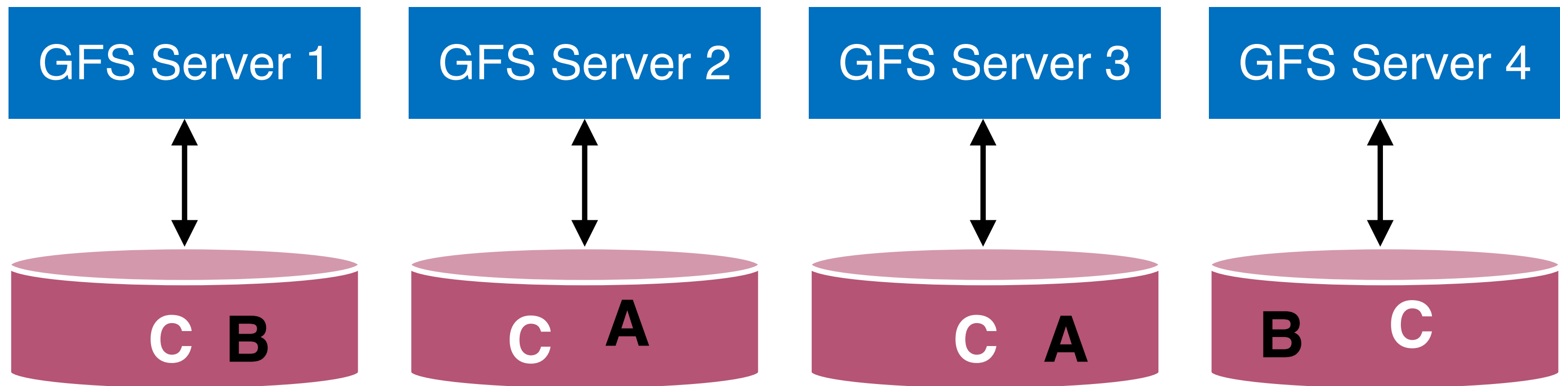
Data Recovery



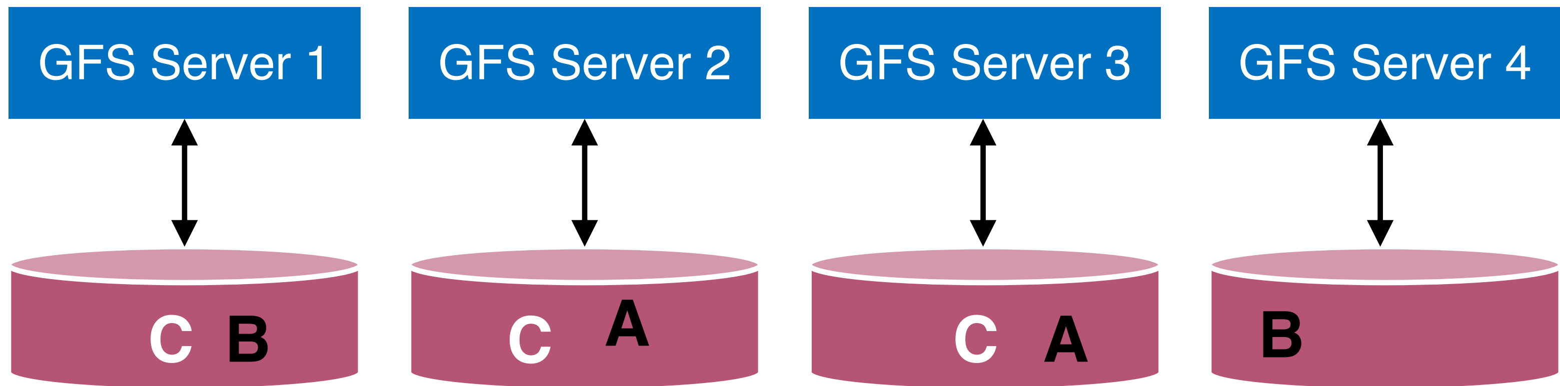
Data Rebalancing

Deleting one A to maintain a replication factor of 2

Data Recovery



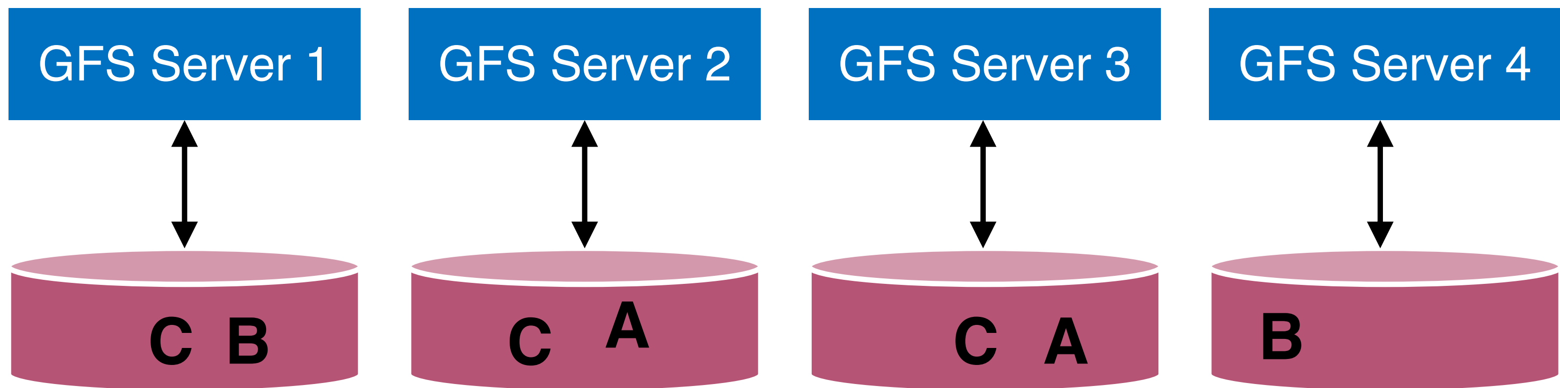
Data Recovery



Data Rebalancing

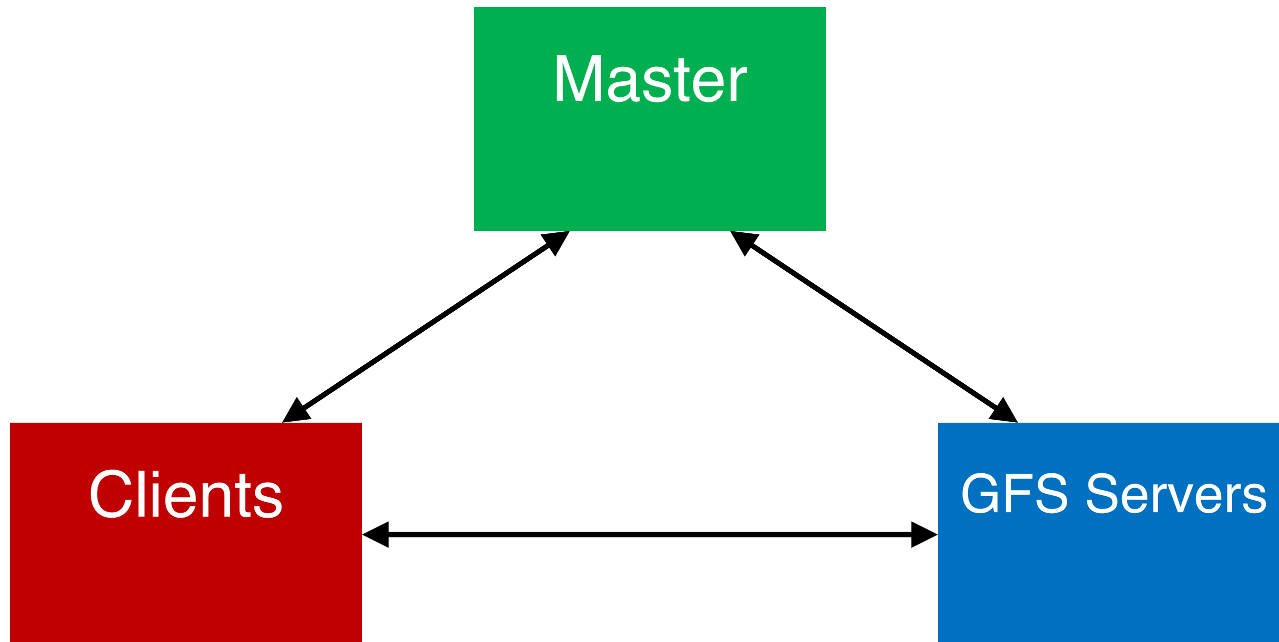
Deleting one C to maintain a replication factor of 3

Data Recovery

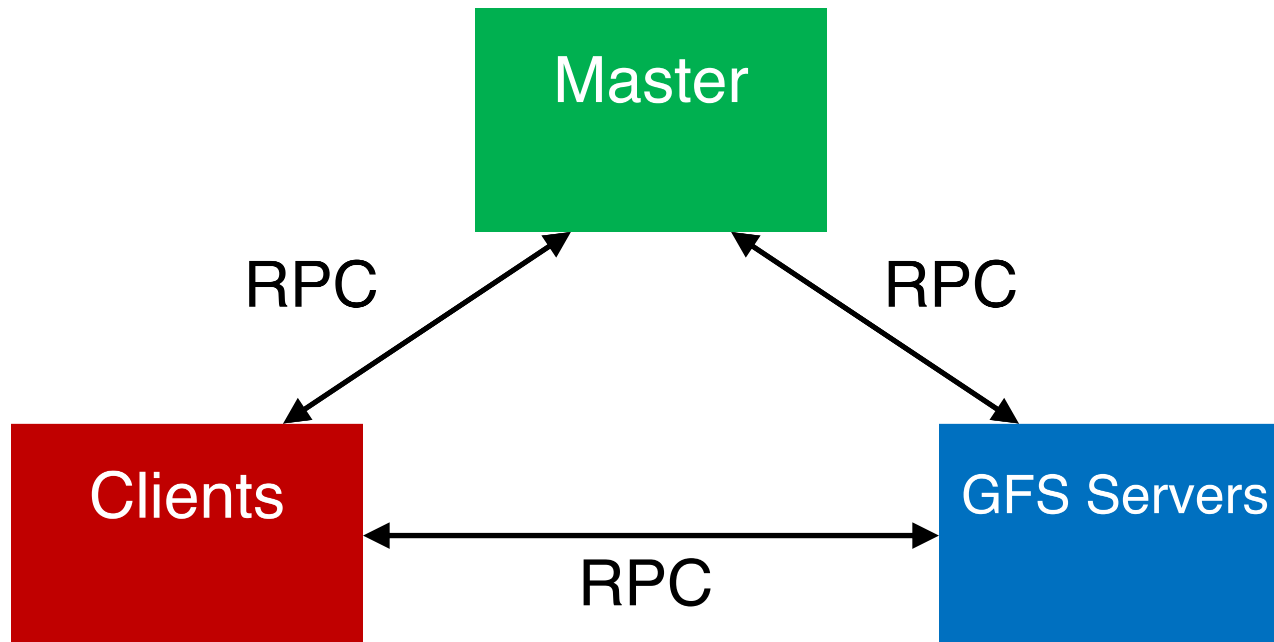


Question: how to maintain a global view of all data distributed across machines?

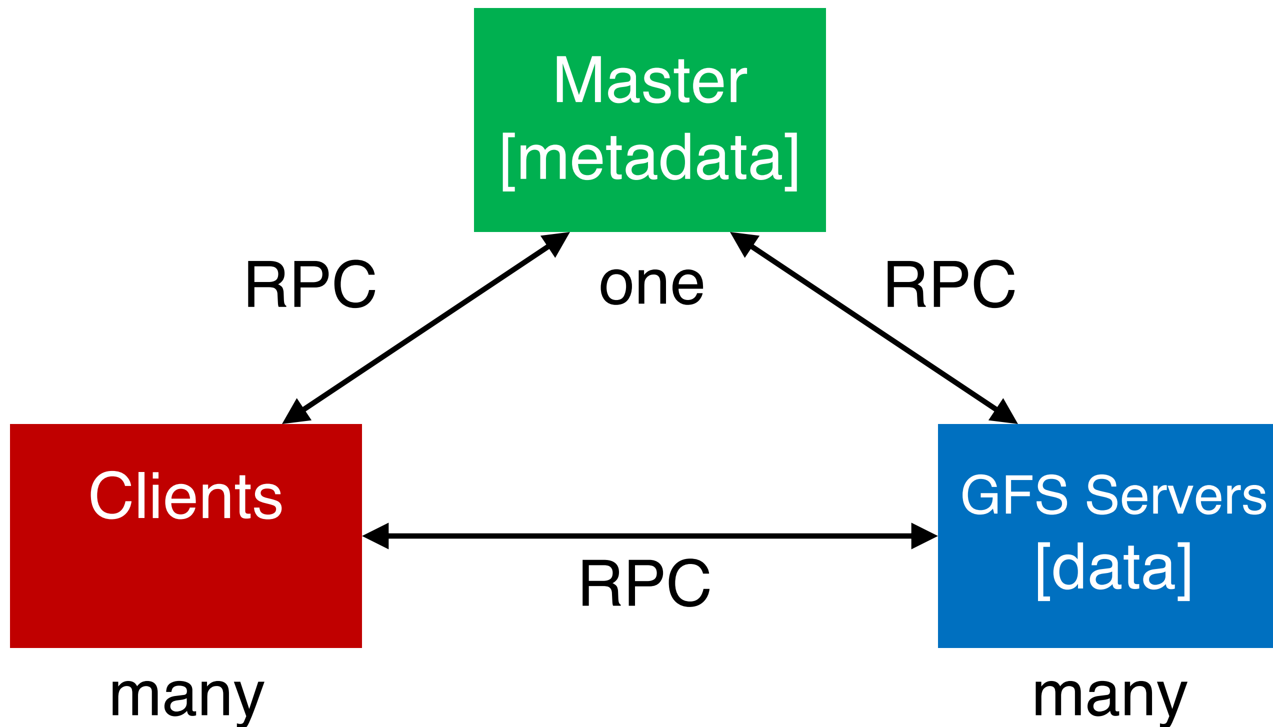
GFS Architecture



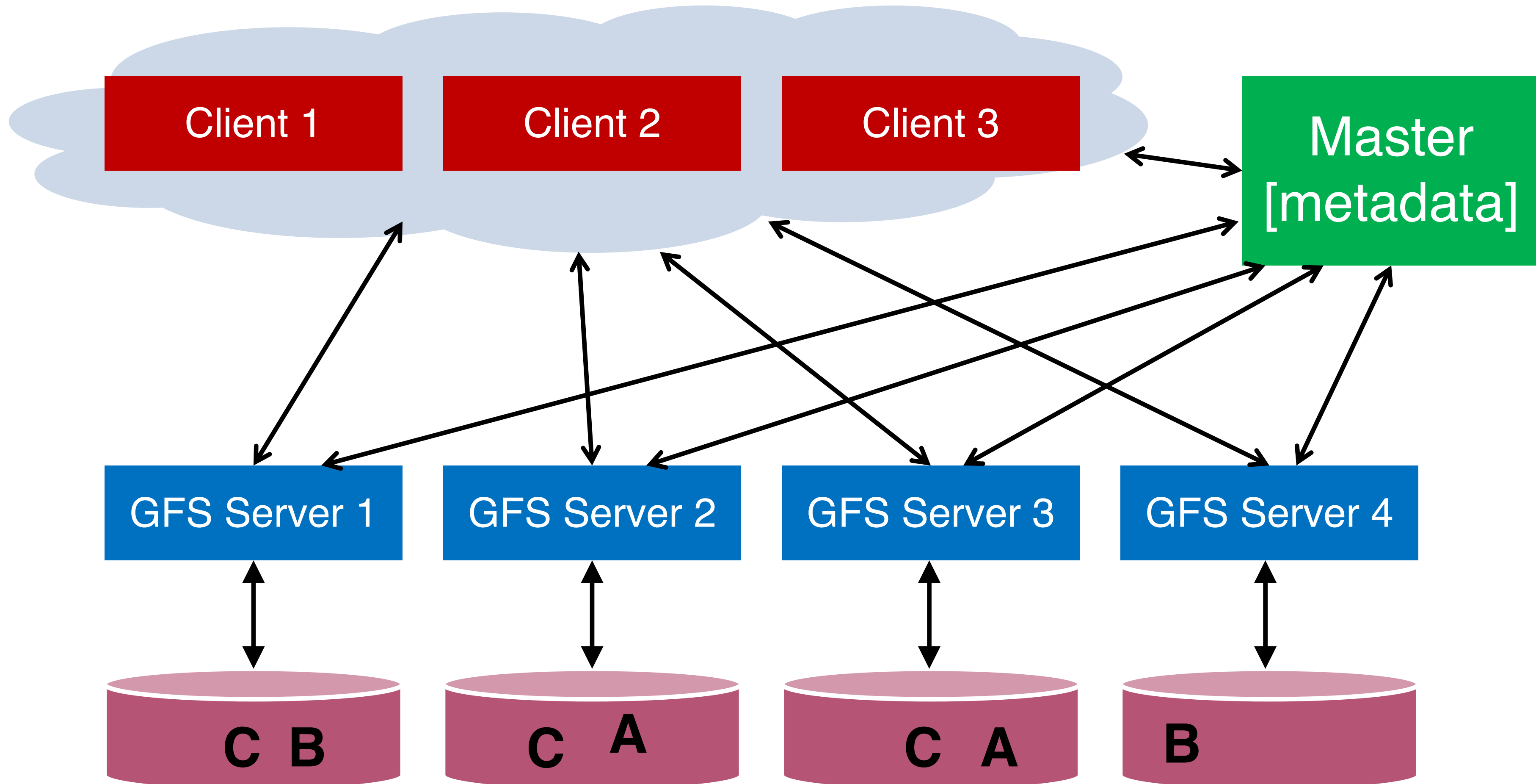
GFS Architecture



GFS Architecture



GFS Architecture



Data Chunks

- Break large GFS files into coarse-grained data chunks (e.g., 64MB)
- GFS servers store physical data chunks in local Linux file system (detail discussed in lec-6a/6b)
- Centralized master keeps track of mapping between logical and physical chunks

Chunk Map

Master

chunk map

logical	phys
924	s2,s5,s7
521	s2,s9,s11
…	…

GFS Server s2

Master

chunk map

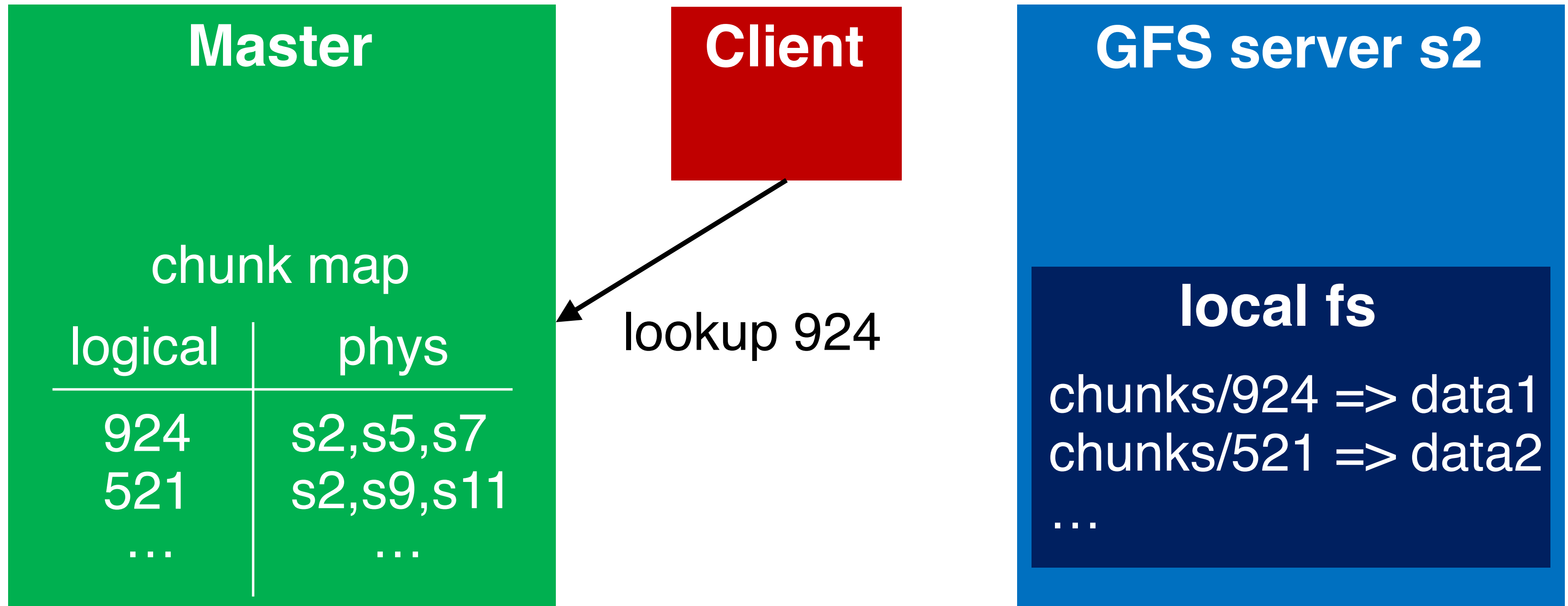
logical	phys
924	s2,s5,s7
521	s2,s9,s11
…	…

GFS server s2

local fs

chunks/924 => data1
chunks/521 => data2
…

Client Reads a Chunk



Client Reads a Chunk

Client Reads a Chunk

Master

chunk map

logical	phys
924	s2,s5,s7
521	s2,s9,s11
…	…

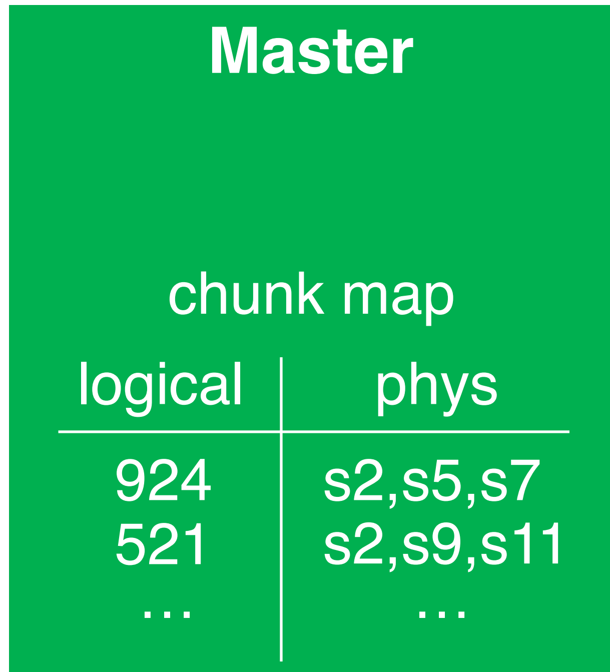
Client

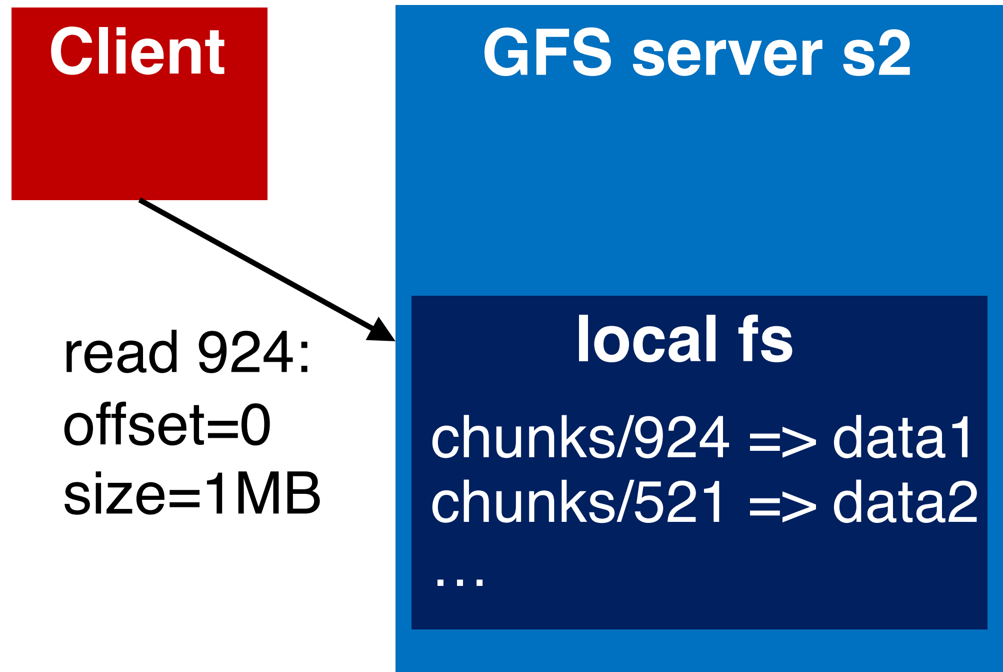
GFS server s2

local fs

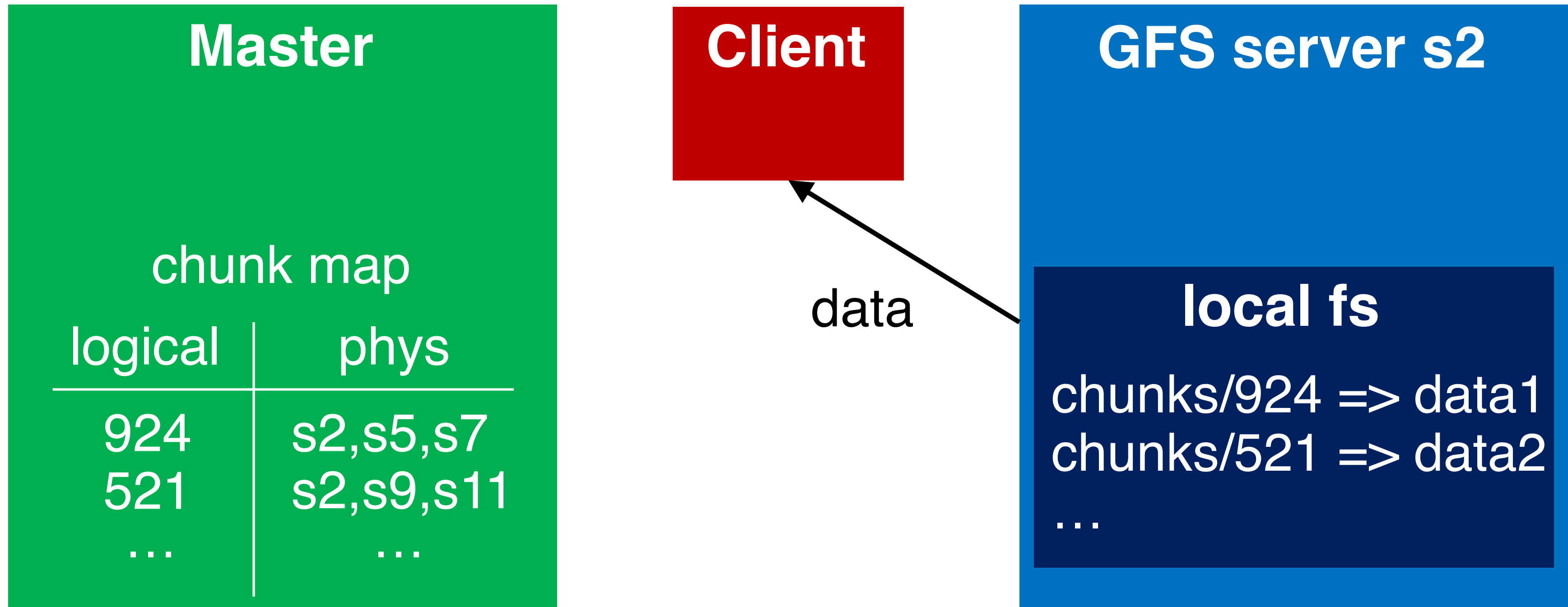
chunks/924 => data1
chunks/521 => data2
…

Client Reads a Chunk

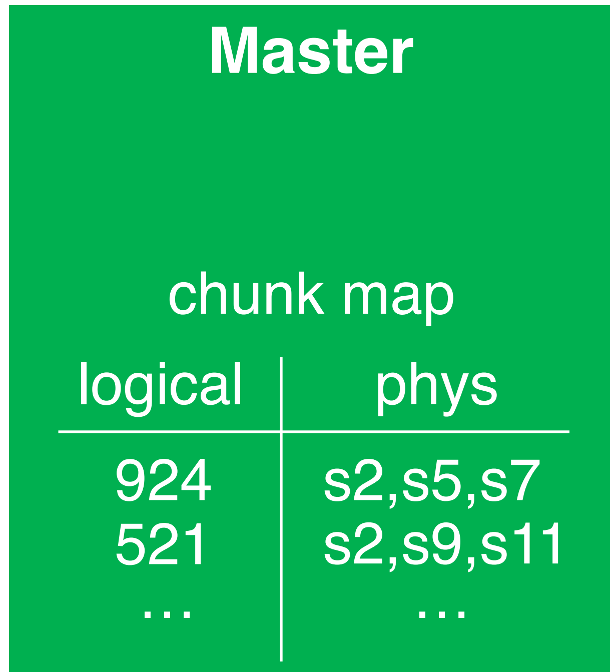




Client Reads a Chunk

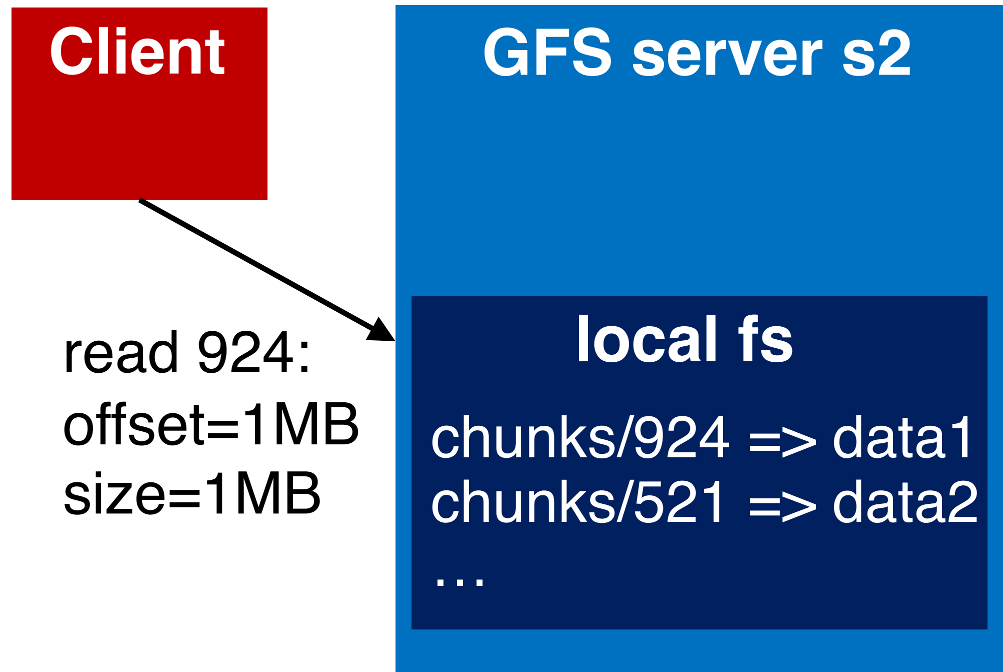


Client Reads a Chunk

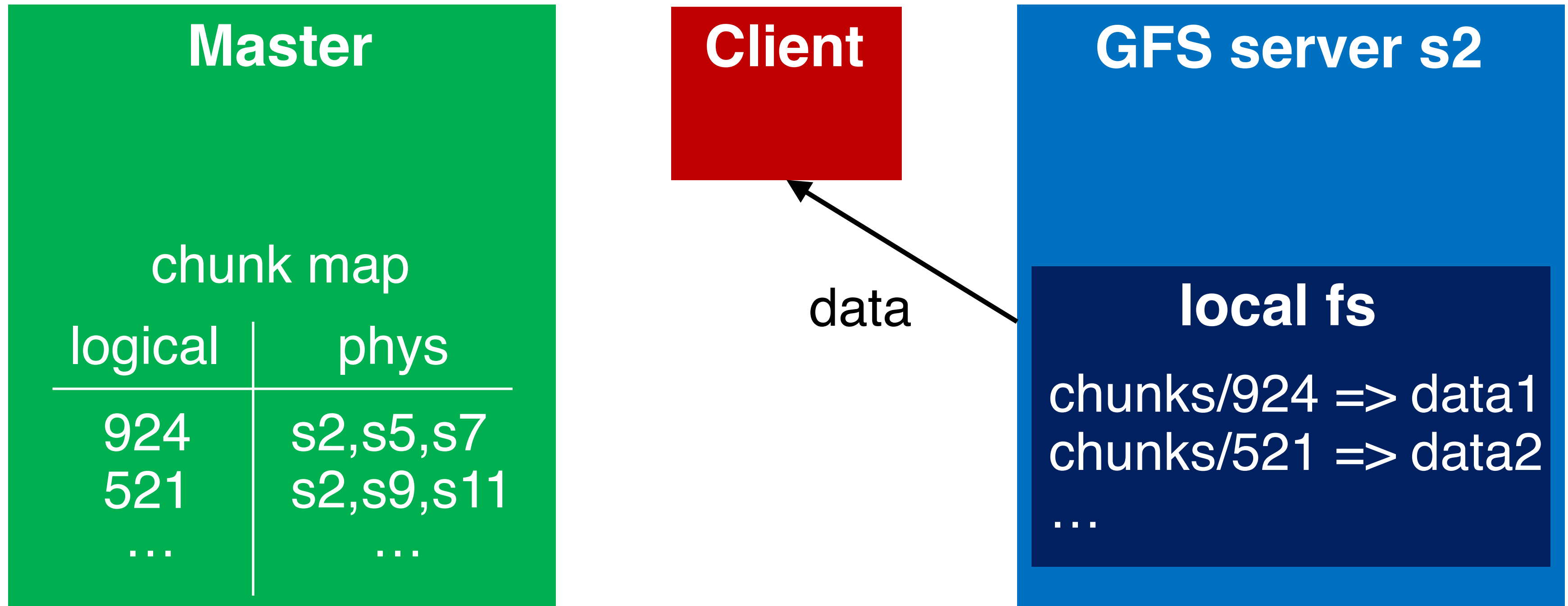


Client

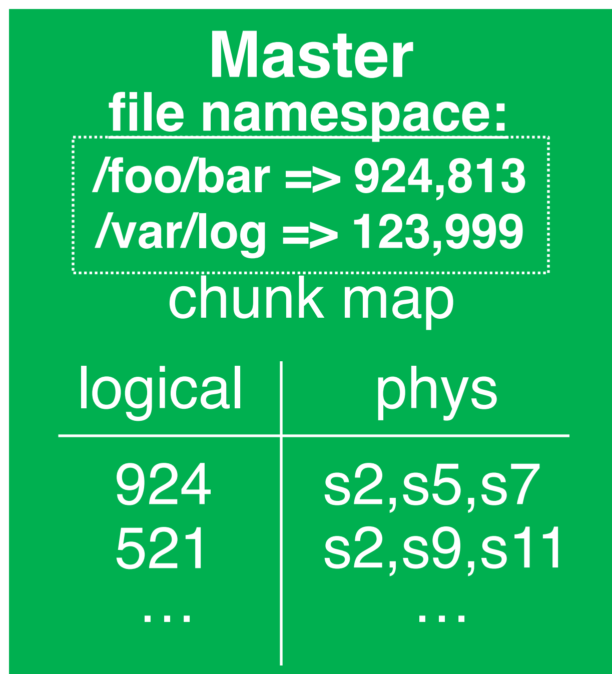
read 924:
offset=1MB
size=1MB

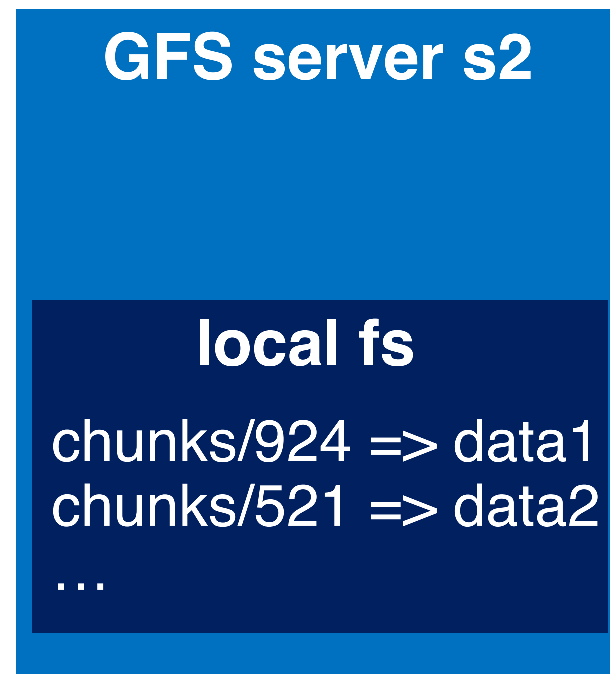


Client Reads a Chunk



File Namespace





path names mapped to logical names

Google File System

MapReduce

Key-Value Store

MapReduce Overview

- Motivation
- Architecture
- Programming Model

Problem

- Datasets are too big to process using single machine

- Good concurrent processing engines are rare (back then in the late 90s)

- Want a concurrent processing framework that is:
  – easy to use (no locks, CVs, race conditions)
  – general (works for many problems)

MapReduce

- Strategy: break data into buckets, do computation over each bucket

- Google published details in 2004

- Open source implementation: Hadoop

Example: Word Count

Word	Count
was	28
what	129
was	54
what	18
was	32
map	10

How to quickly sum word counts with multiple machines concurrently?

Example: Word Count

Word	Count
was	28
what	129
was	54
what	18
was	32
map	10

mapper 1

was	28
what	129
was	54

mapper 2

what	18
was	32
map	10

Example: Word Count

Word	Count
was	28
what	129
was	54
what	18
was	32
map	10

mapper 1

was	28
what	129
was	54

→

was	28+54
what	129

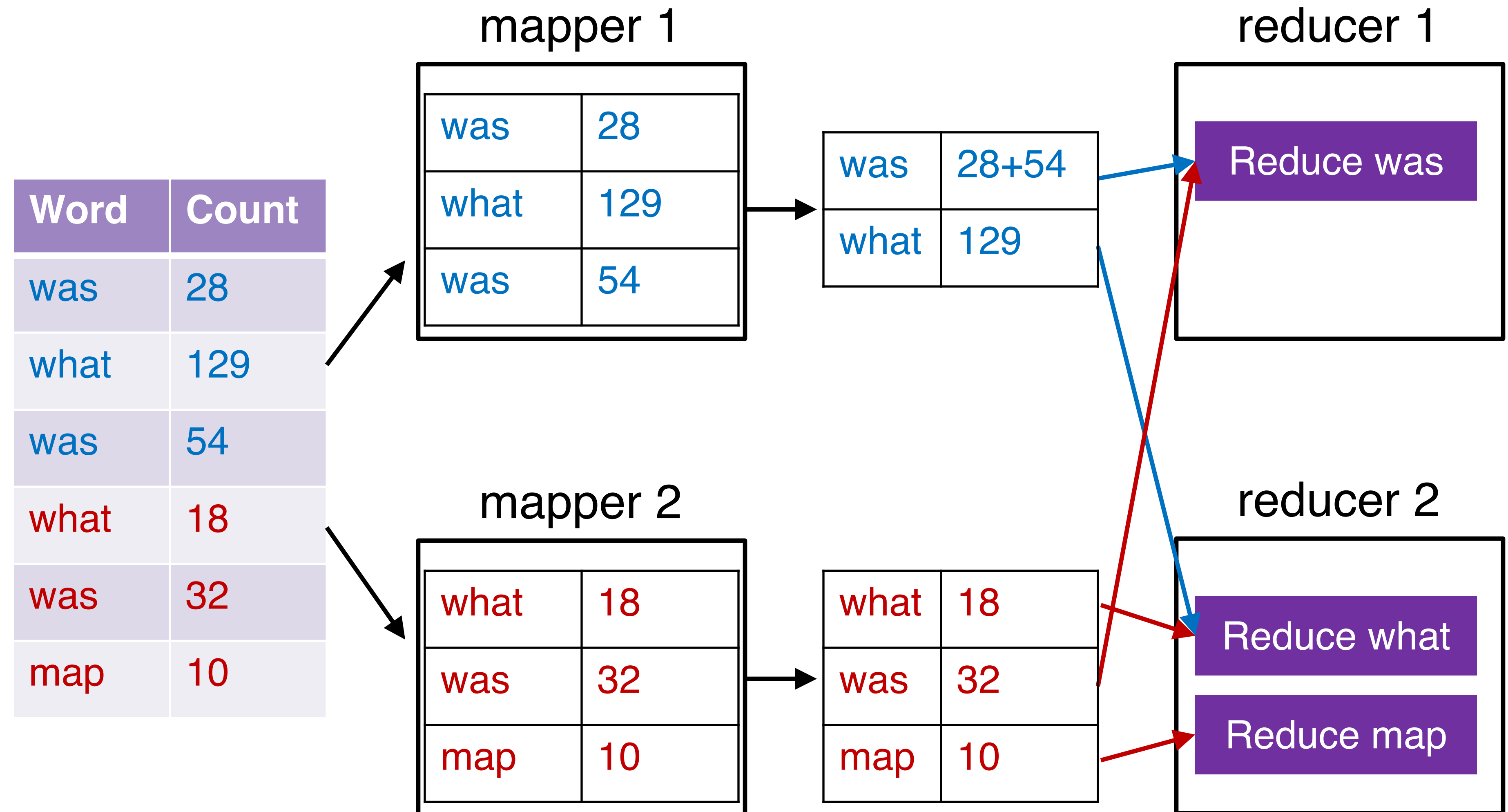
mapper 2

what	18
was	32
map	10

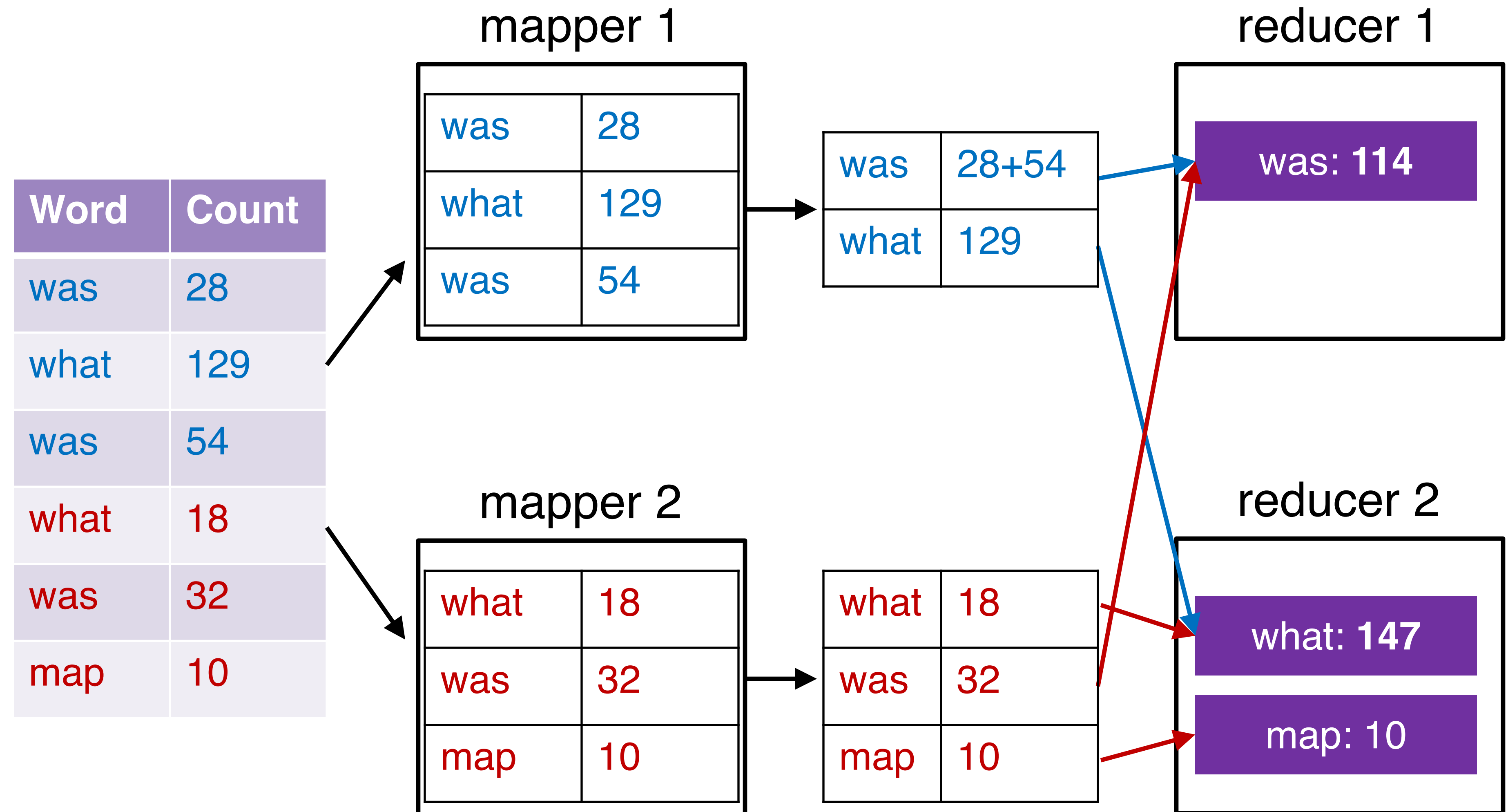
→

what	18
was	32
map	10

Example: Word Count



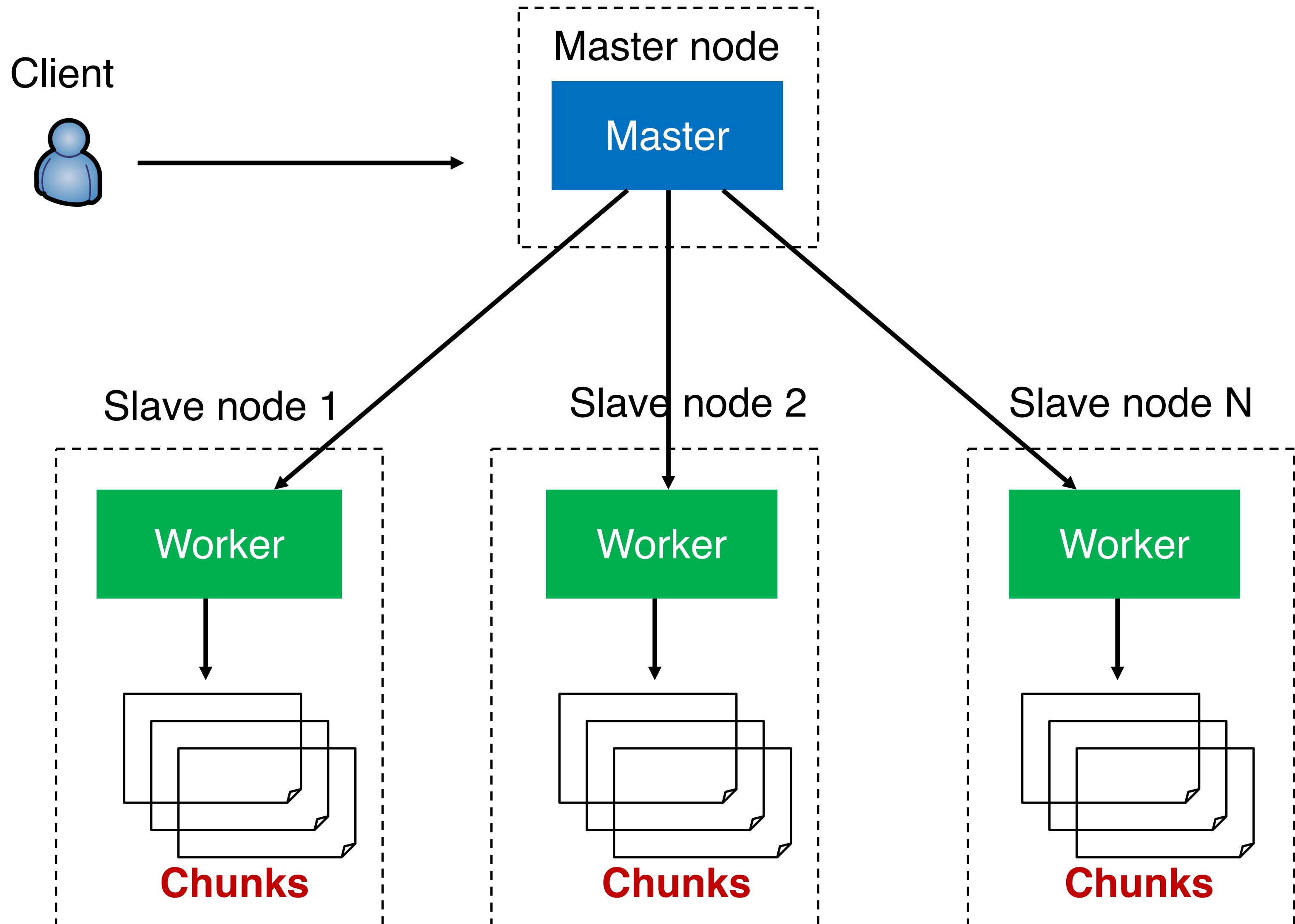
Example: Word Count



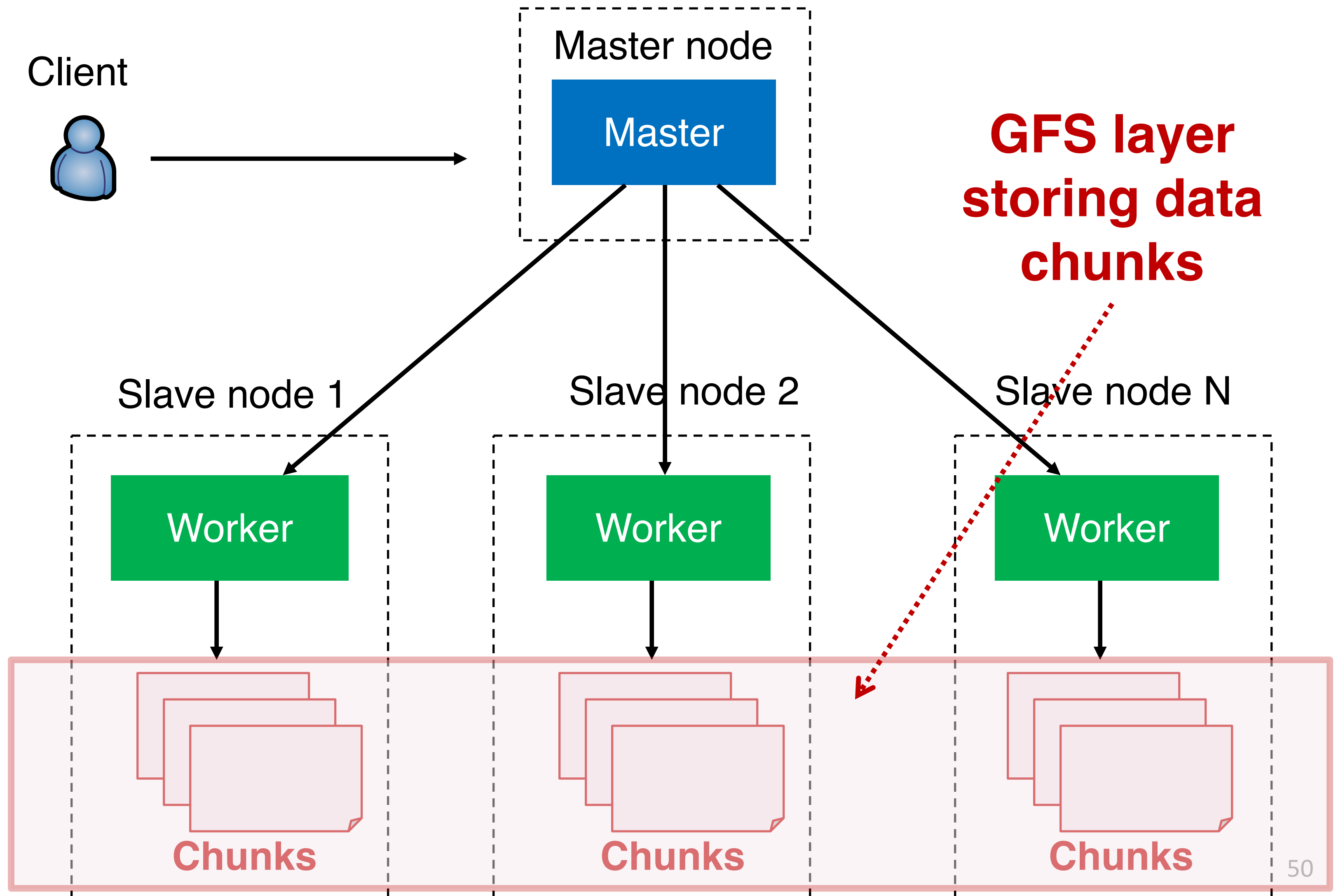
MapReduce Overview

- Motivation
- Architecture
- Programming Model

MapReduce Architecture



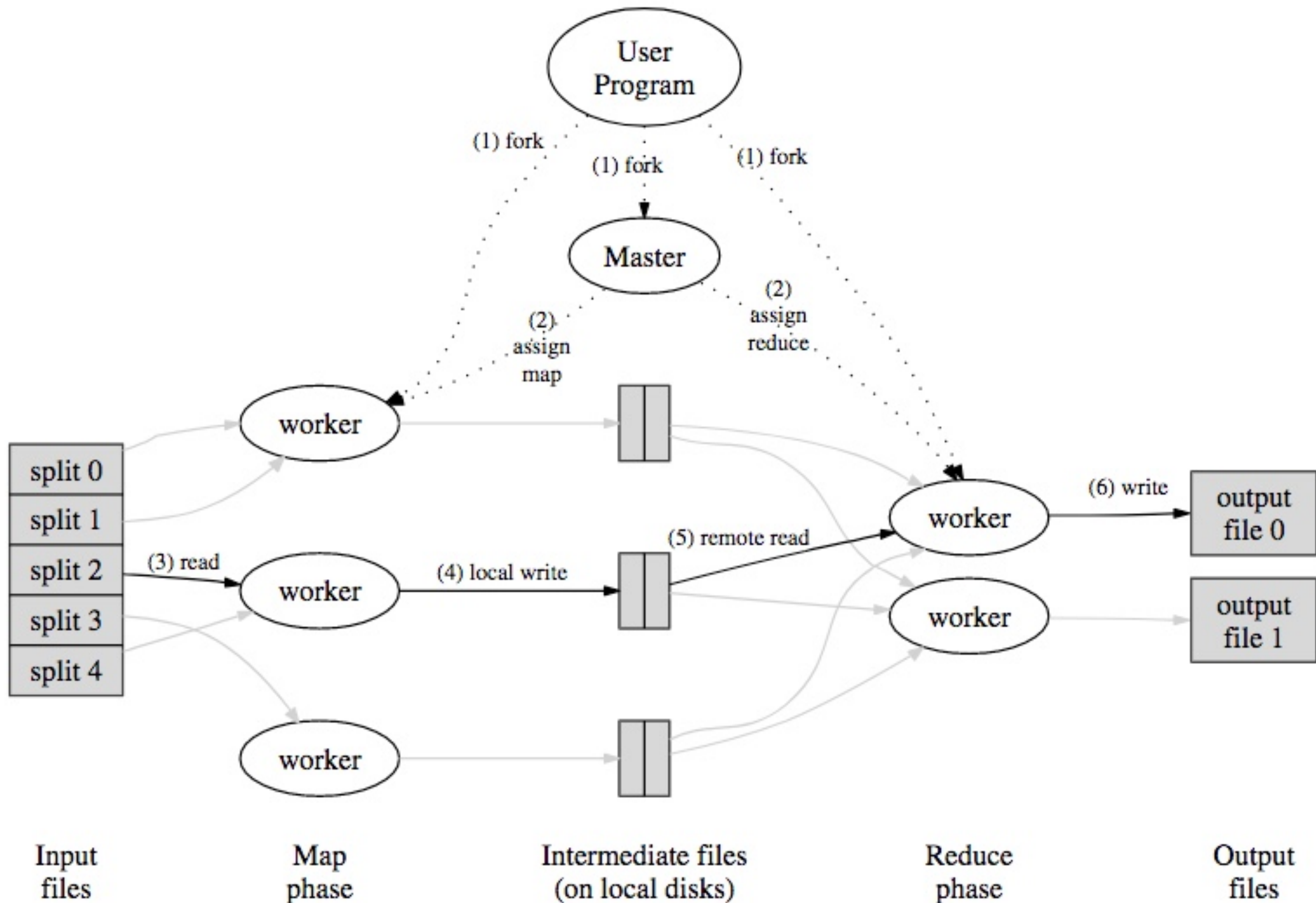
MapReduce Architecture



MapReduce over GFS

- MapReduce writes and reads data to/from GFS
- MapReduce workers run on same machines as GFS server daemons

MapReduce Data Flows & Executions



User Program

(1) fork
(1) fork
(1) fork

Master

(2) assign map
(2) assign reduce

split 0
split 1
split 2
split 3
split 4

(3) read

worker
worker
worker

(4) local write

(5) remote read

worker
worker

(6) write

output file 0
output file 1

Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

MapReduce Overview

- Motivation
- Architecture
- Programming Model

Map/Reduce Function Types

- map(k1, v1) → list(k2, v2)
- reduce(k2, list(v2)) → list(k3, v3)

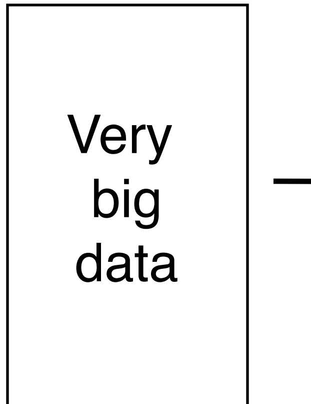
Hadoop API

```
public void map(LongWritable key, Text value) {
    // WRITE CODE HERE
}

public void reduce(Text key, Iterator<IntWritable> values)
{
    // WRITE CODE HERE
}
```

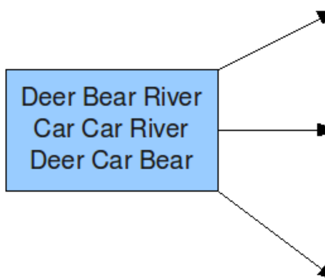
MapReduce Word Count Pseudo Code

```
func mapper(key, line) {
    for word in line.split()
        yield word, 1
}

func reducer(word, occurrences) {
    yield word, sum(occurrences)
}
```

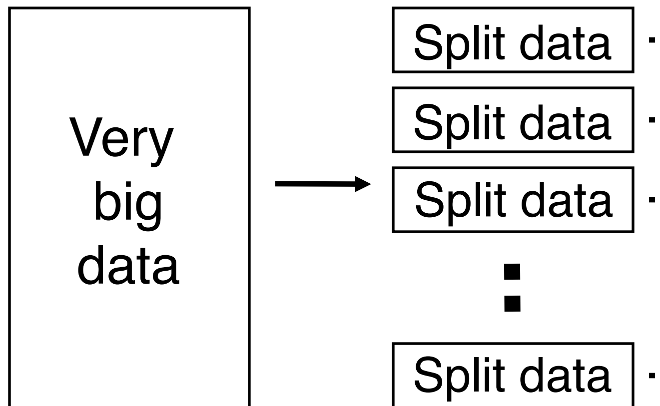
MapReduce Word Count



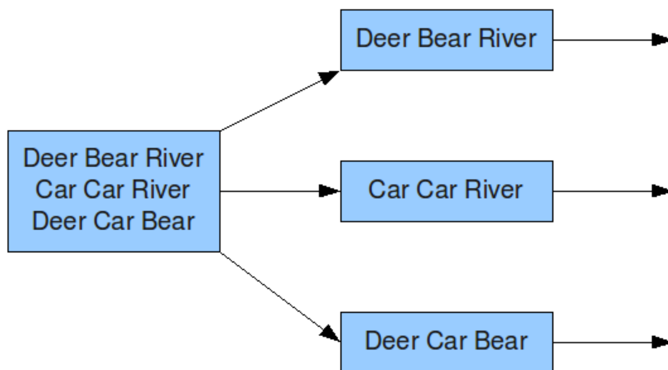
Input



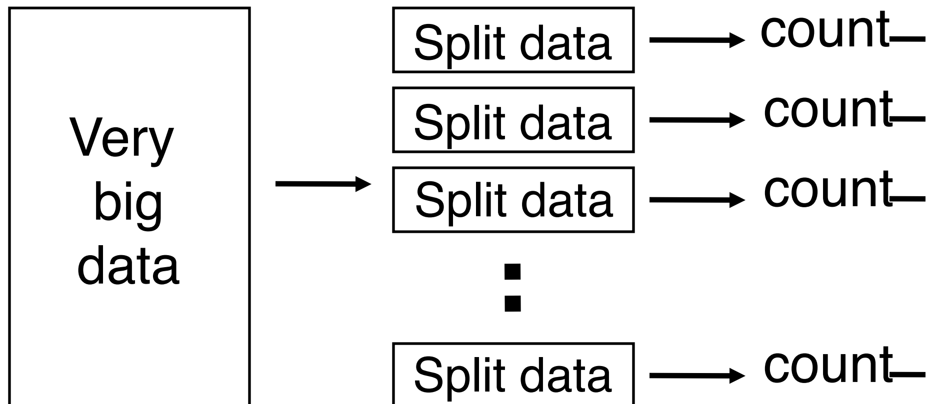
MapReduce Word Count

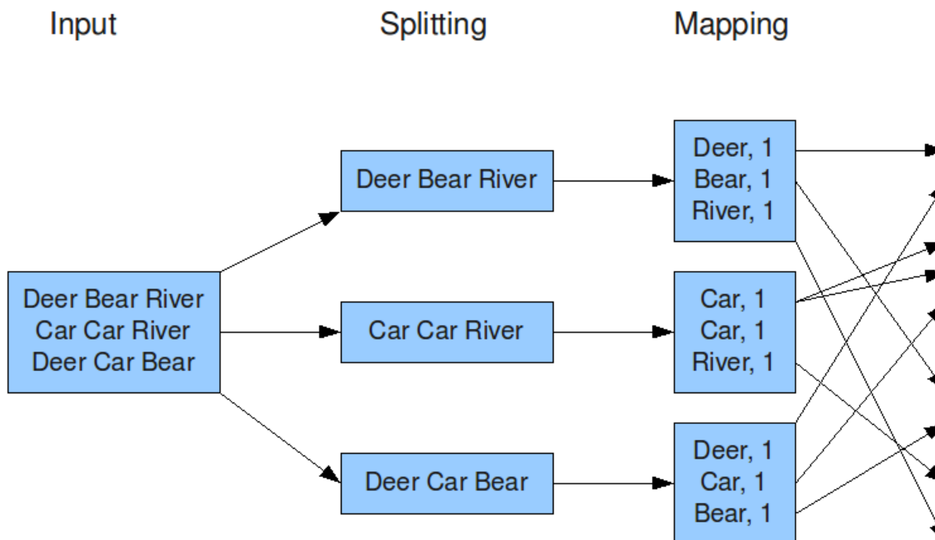


The overall M

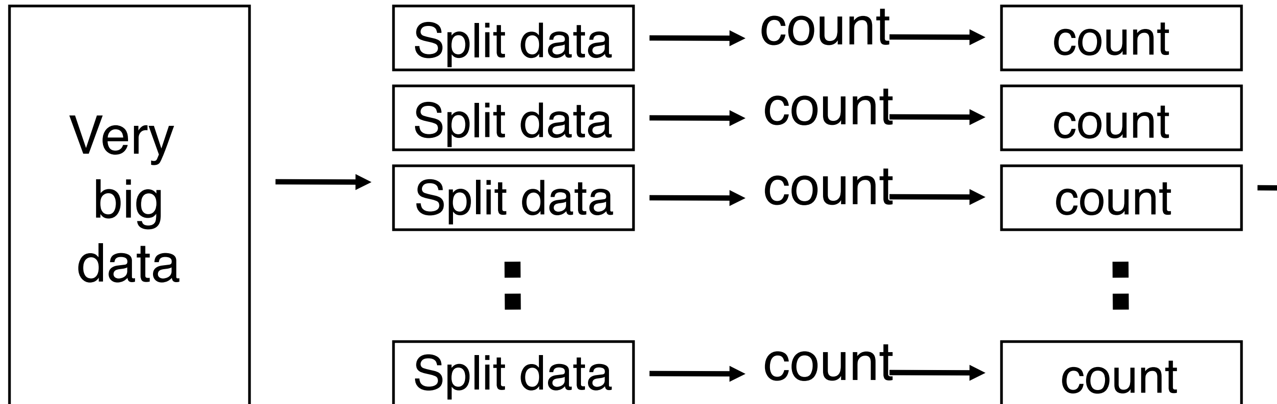


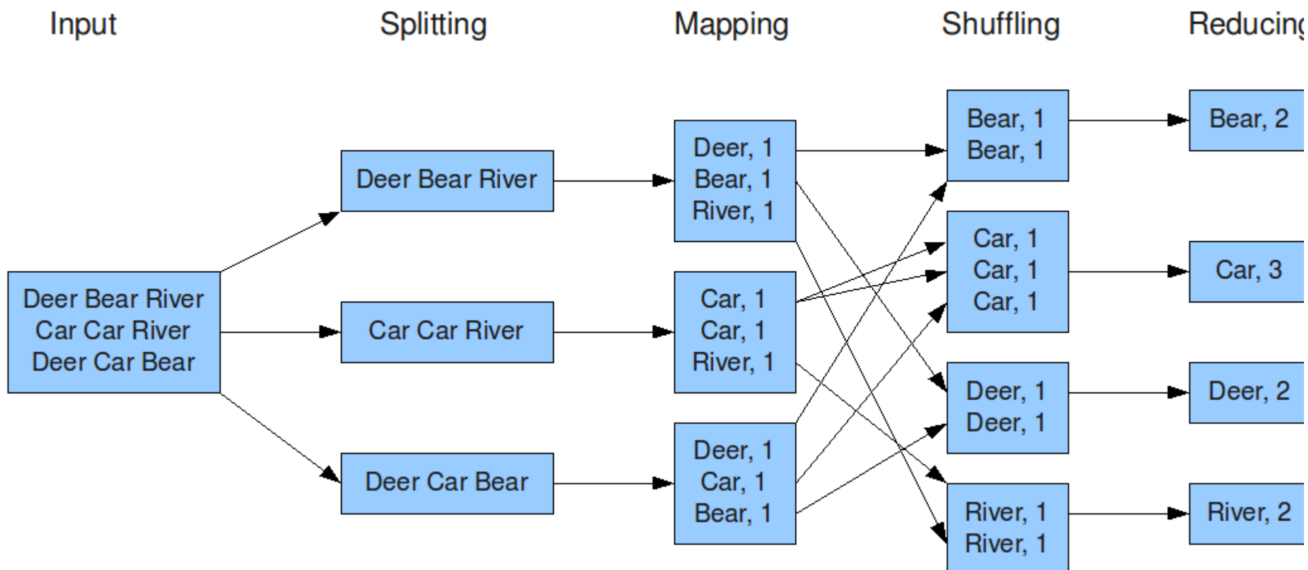
MapReduce Word Count



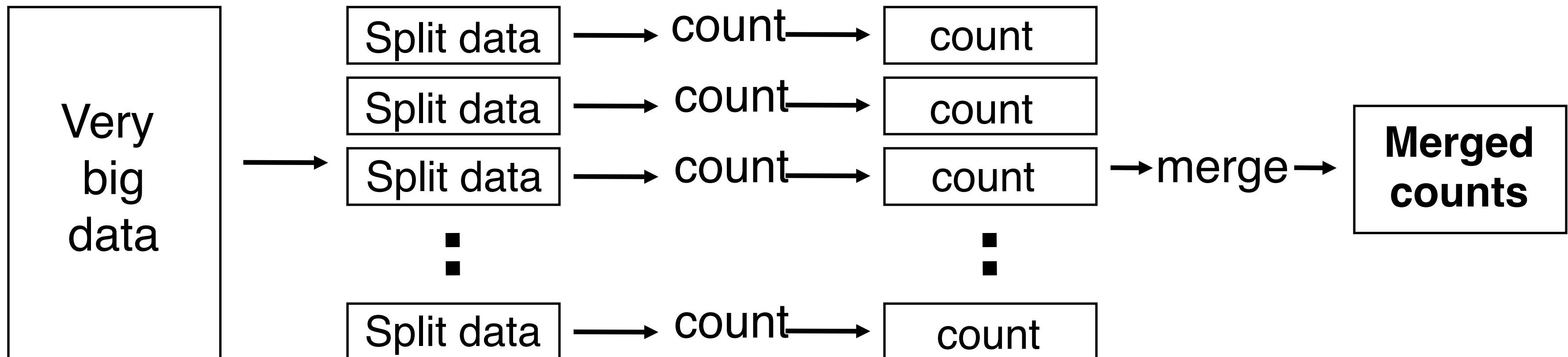


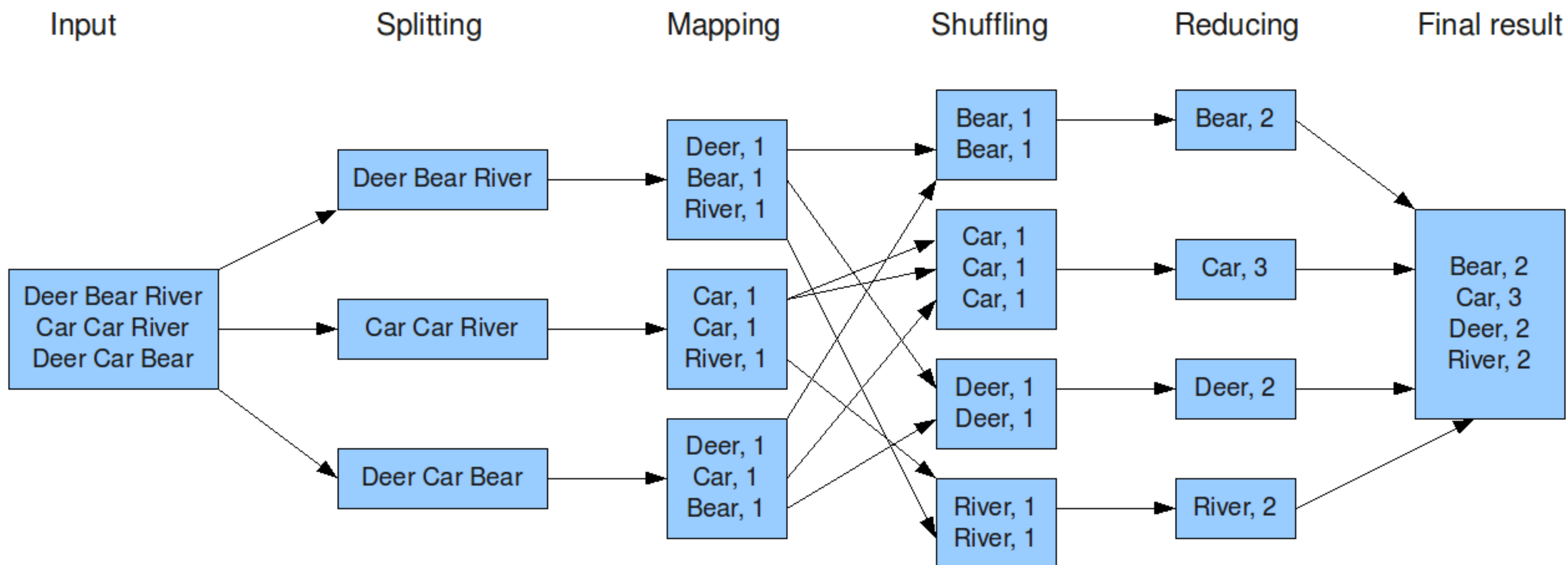
MapReduce Word Count



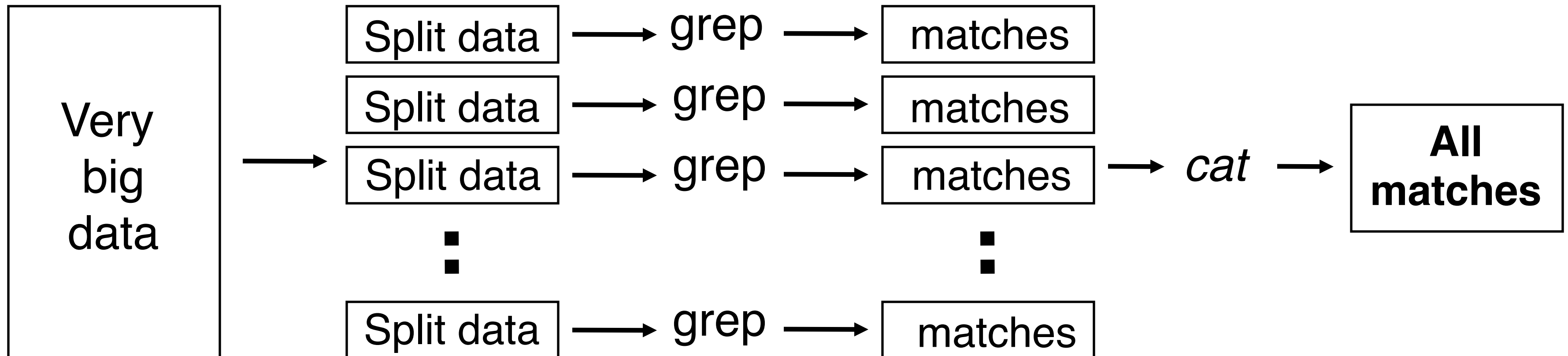


MapReduce Word Count





MapReduce Grep



Google File System
MapReduce
Key-Value Store

Credit: Prof. Hector Garcia-Molina@Stanford

Key-Value Store

Table T:

key	value
k1	v1
k2	v2
k3	v3
k4	v4

Key-Value Store

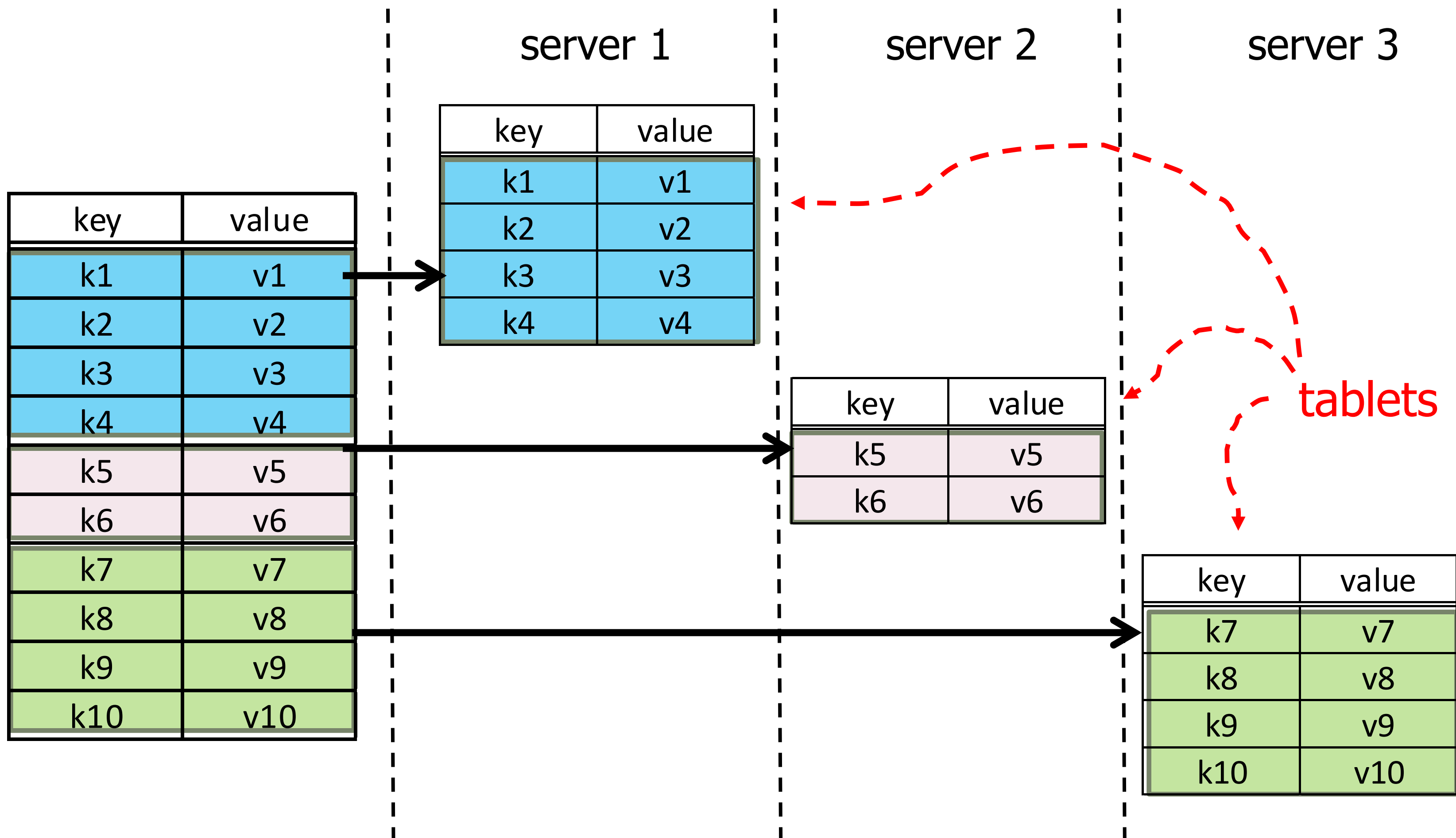
Table T:

key	value
k1	v1
k2	v2
k3	v3
k4	v4

keys are sorted

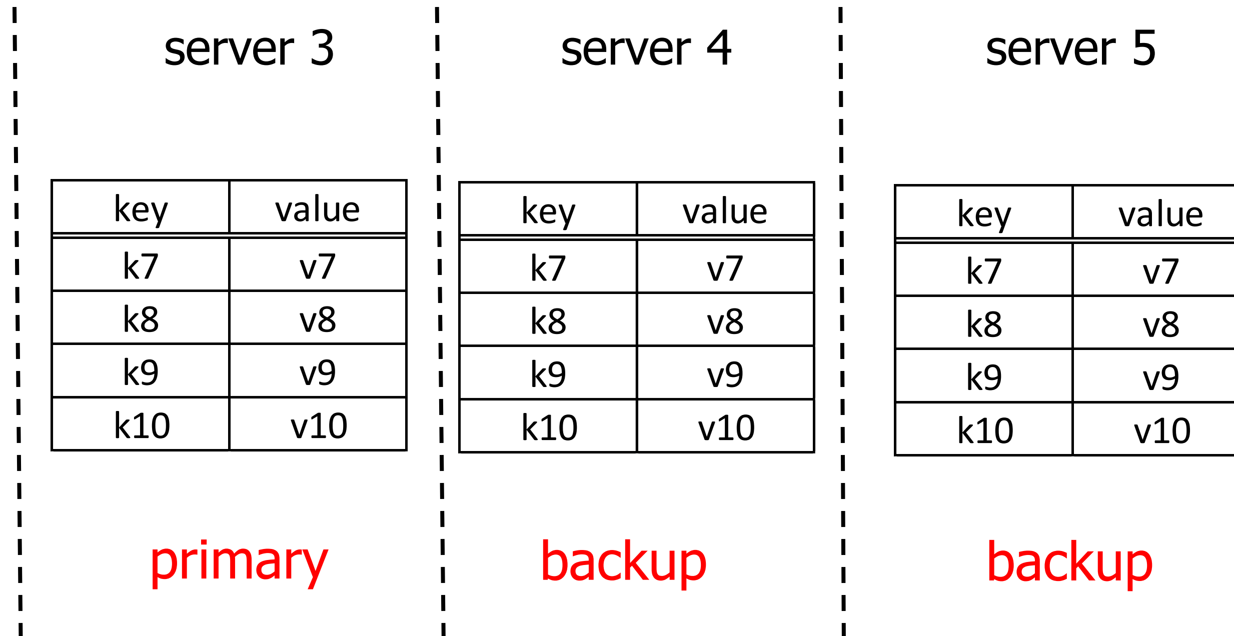
- API:
  - `lookup(key)` → `value`
  - `lookup(key range)` → `values`
  - `getNext` → `value`
  - `insert(key, value)`
  - `delete(key)`
- Each row has timestemp
- Single row actions atomic
  (but not persistent in some systems?)
- No multi-key transactions
- No query language!

Partitioning (Sharding)



- use a partition vector
- “auto-sharding”: vector selected automatically

Tablet Replication



- <u>Cassandra:</u>
  Replication Factor (# copies)
  R/W Rule: One, Quorum, All
  Policy (e.g., Rack Unaware, Rack Aware, ...)
  Read all copies (return fastest reply, do repairs if necessary)
- <u>HBase:</u> Does not manage replication, relies on HDFS

Need a "directory"

- Add naming hierarchy to a flat namespace
- Table Name:
  - Key $\rightarrow$ Servers: stores key $\rightarrow$ Backup servers
- Can be implemented as a special table

Tablet Internals

memory

key	value
k3	v3
k8	v8
k9	delete
k15	v15

---

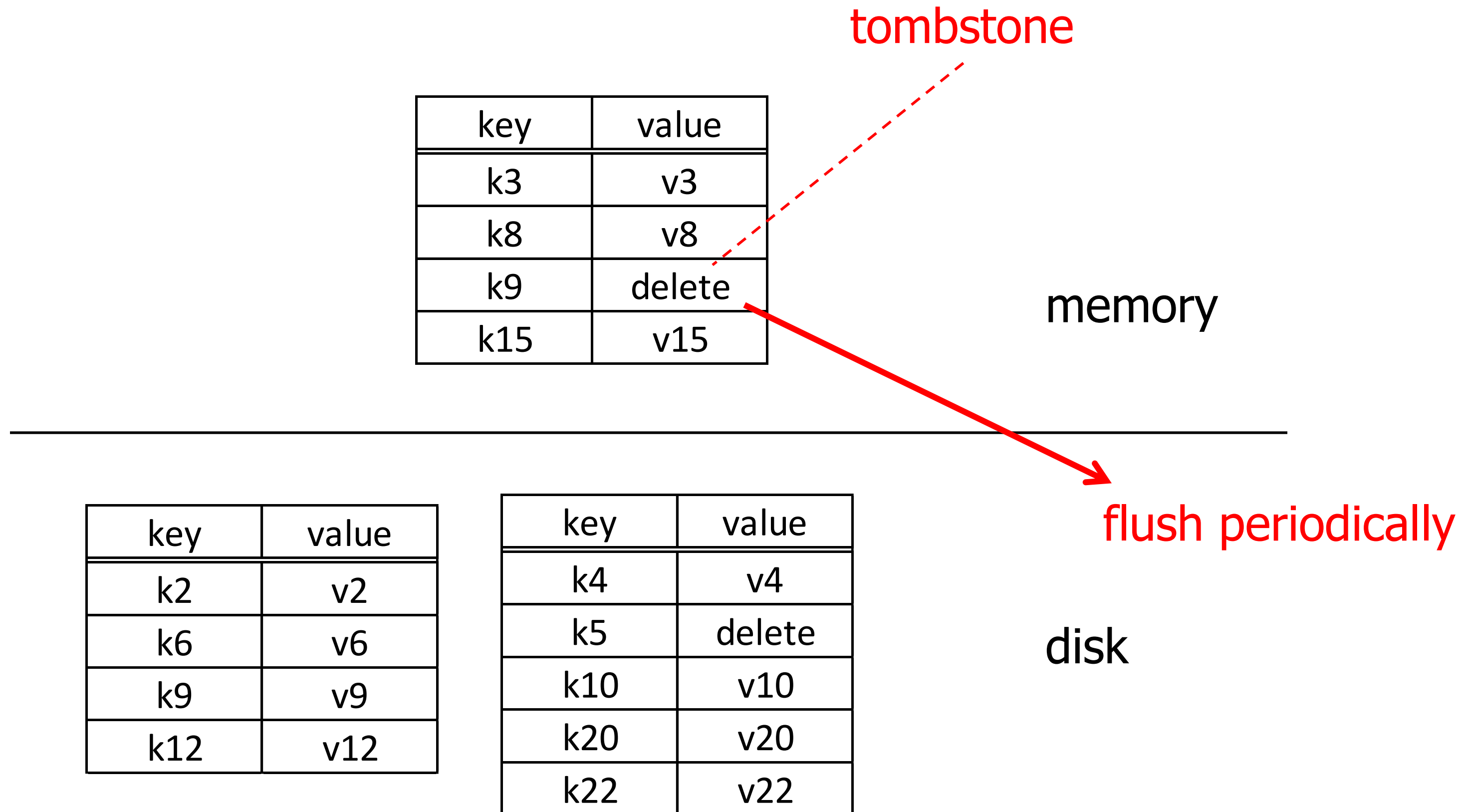
disk

key	value
k2	v2
k6	v6
k9	v9
k12	v12

key	value
k4	v4
k5	delete
k10	v10
k20	v20
k22	v22

Design Philosophy (?): Primary scenario is where all data is in memory
Disk storage added as an afterthought

Tablet Internals



key	value
k3	v3
k8	v8
k9	delete
k15	v15

key	value
k2	v2
k6	v6
k9	v9
k12	v12

key	value
k4	v4
k5	delete
k10	v10
k20	v20
k22	v22

- tablet is merge of all segments (files)
- disk segments imutable
- writes efficient; reads only efficient when all data in memory
- periodically reorganize into single segment

Column Family

K	A	B	C	D	E
k1	a1	b1	c1	d1	e1
k2	a2	null	c2	d2	e2
k3	null	null	null	d3	e3
k4	a4	b4	c4	e4	e4
k5	a5	b5	null	null	null

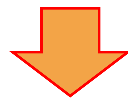
Column Family

K	A	B	C	D	E
k1	a1	b1	c1	d1	e1
k2	a2	null	c2	d2	e2
k3	null	null	null	d3	e3
k4	a4	b4	c4	e4	e4
k5	a5	b5	null	null	null

- for storage, treat each row as a single "super value"
- API provides access to sub-values
  (use family:qualifier to refer to sub-values
  e.g., price:euros, price:dollars)
- Cassandra allows "super-column":
  two level nesting of columns
  (e.g., Column A can have sub-columns X & Y)

Vertical Partitions

K	A	B	C	D	E
k1	a1	b1	c1	d1	e1
k2	a2	null	c2	d2	e2
k3	null	null	null	d3	e3
k4	a4	b4	c4	e4	e4
k5	a5	b5	null	null	null



can be manually implemented as

K	A
k1	a1
k2	a2
k4	a4
k5	a5

server 1

K	B
k1	b1
k4	b4
k5	b5

server 2

K	C
k1	c1
k2	c2
k4	c4

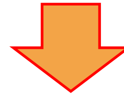
server 3

K	D	E
k1	d1	e1
k2	d2	e2
k3	d3	e3
k4	e4	e4

server 4

Vertical Partitions

K	A	B	C	D	E
k1	a1	b1	c1	d1	e1
k2	a2	null	c2	d2	e2
k3	null	null	null	d3	e3
k4	a4	b4	c4	e4	e4
k5	a5	b5	null	null	null



column family

K	A
k1	a1
k2	a2
k4	a4
k5	a5

K	B
k1	b1
k4	b4
k5	b5

K	C
k1	c1
k2	c2
k4	c4

K	D	E
k1	d1	e1
k2	d2	e2
k3	d3	e3
k4	e4	e4

- good for sparse data;
- good for column scans
- not so good for tuple reads
- are atomic updates to row still supported?
- API supports actions on full table; mapped to actions on column tables
- API supports column "project"
- To decide on vertical partition, need to know access patterns