MapReduce

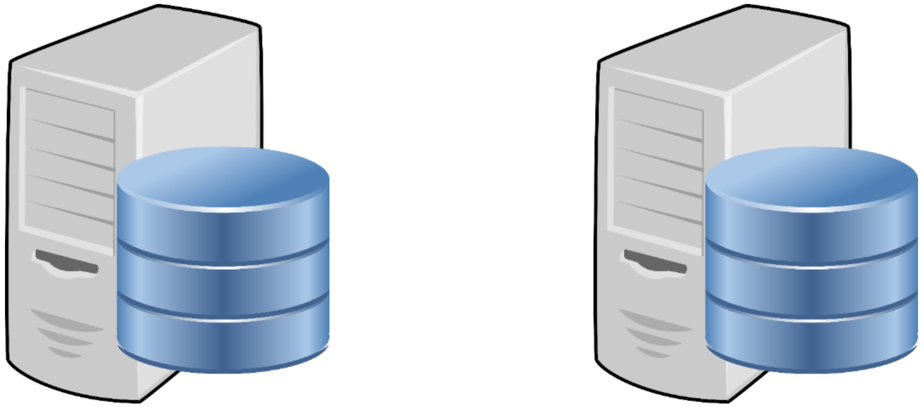
Agenda

- Why distributed data processing?
- Simple data parallelization
- The MapReduce system
- How MapReduce works
- Google File System for distributed storage
- MapReduce examples
- Fault-tolerance
- Map reduce performance Issues
- Limitations

Processing Data

- Data on file system on disk
- Simple processing flow:
  - 1. Read into memory
  - 2. Process data (apply some function)
  - 3. Write back to disk

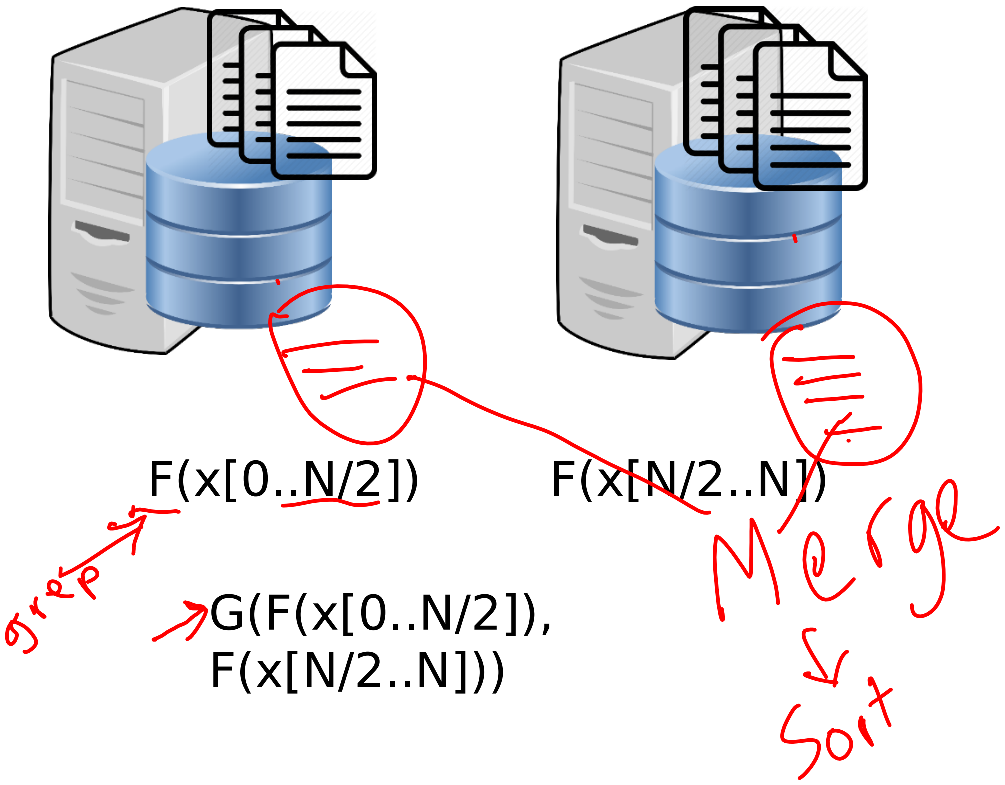
Why Multiple Servers?



- Data size larger than disk capacity
- Sequentially processing data can be slow
  - Limited by disk read/write speeds
  - Parallel processing can significantly reduce time
- Single point of failure with a single server

Multiple Servers: Divide and Conquer

grep



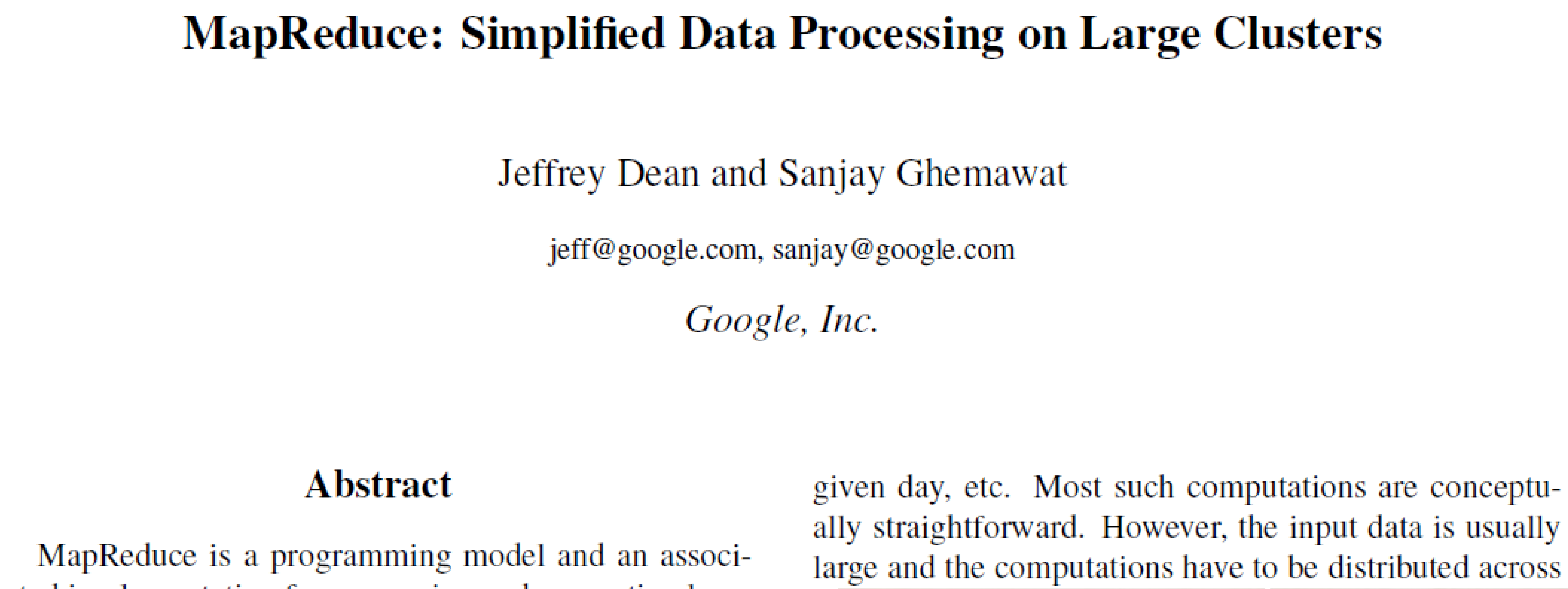
F(x[0..N/2])

F(x[N/2..N])

G(F(x[0..N/2]),
F(x[N/2..N]))

- Divide input data among multiple servers
- Each server processes a partition in parallel
- ???
- Profit!?

- Data processed by individual servers must be "aggregated" or collected
- May require significant communication



MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across
hu
a
al

Published at OSDI 2004.
>25,000 citations

Why MapReduce?

- Automated parallelization and distribution of data and processing
- Clean, powerful, well-understood abstraction (Map and Reduce)
- Fault-tolerance
- Scalability: 1000s of servers, TBs of data, …
- Apache Hadoop: widely used open source Java implementation

Some MapReduce Problems

- Word-count
- Creating an inverted index
  - Which documents does a word occur in?
  - Useful if you are building a search engine
- Log-processing
  - Filter log messages which match some condition
  - IP address == 156.54.61.*
- Matrix processing
  - PageRank, matrix multiplication, …

(List (Processing (Like) It's) 1959)

- LISP: Everything is a List
- '(1 2 3 4 5 6)
- Map: Apply a function element-wise
- (map square '(1 2 3 4 5))
  - -> '(1 4 9 16 25)

+ 1 (2 3 4 5)

+ 1 (+ 2 ·('3 4 5)

- Reduce: Aggregate values in a list
  - Also called fold 4R
- f(x, y) -> z
- (reduce sum '(1 2 3 4 5))
  - -> 15
- Can pass any associative function

+ x y z

≡ (+ x y) + z

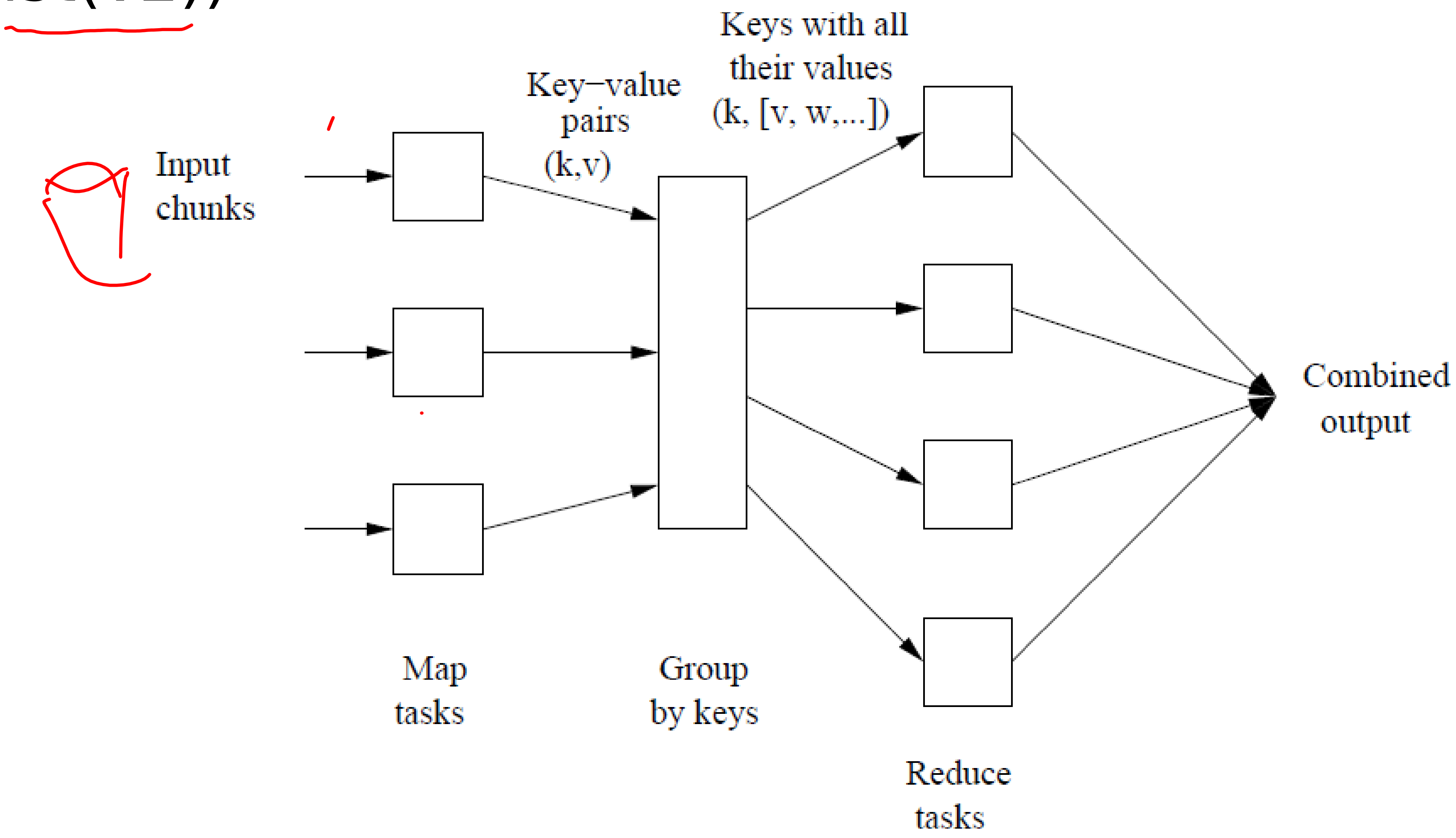
≡ x + (y + z)

(reduce sum (map square '(1 2 3 4 5)))

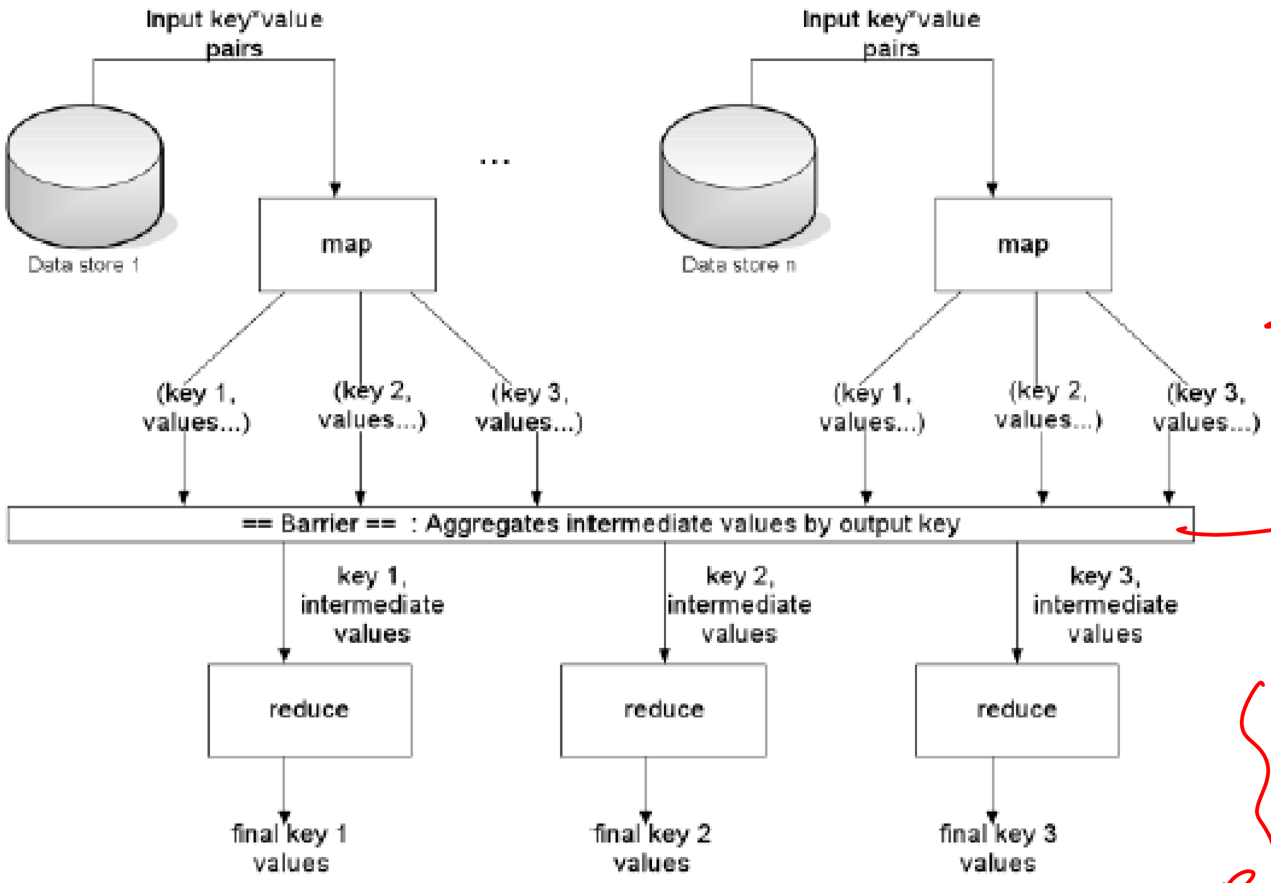
-> Sum of squares

Map Reduce Semantics

- Map: (k1, v1) -> list(k2, v2)
- Reduce: (k2, list(v2)) -> list(v3)



MapReduce System Architecture



Intermediate Data

Barrier: Wait for all processes to finish

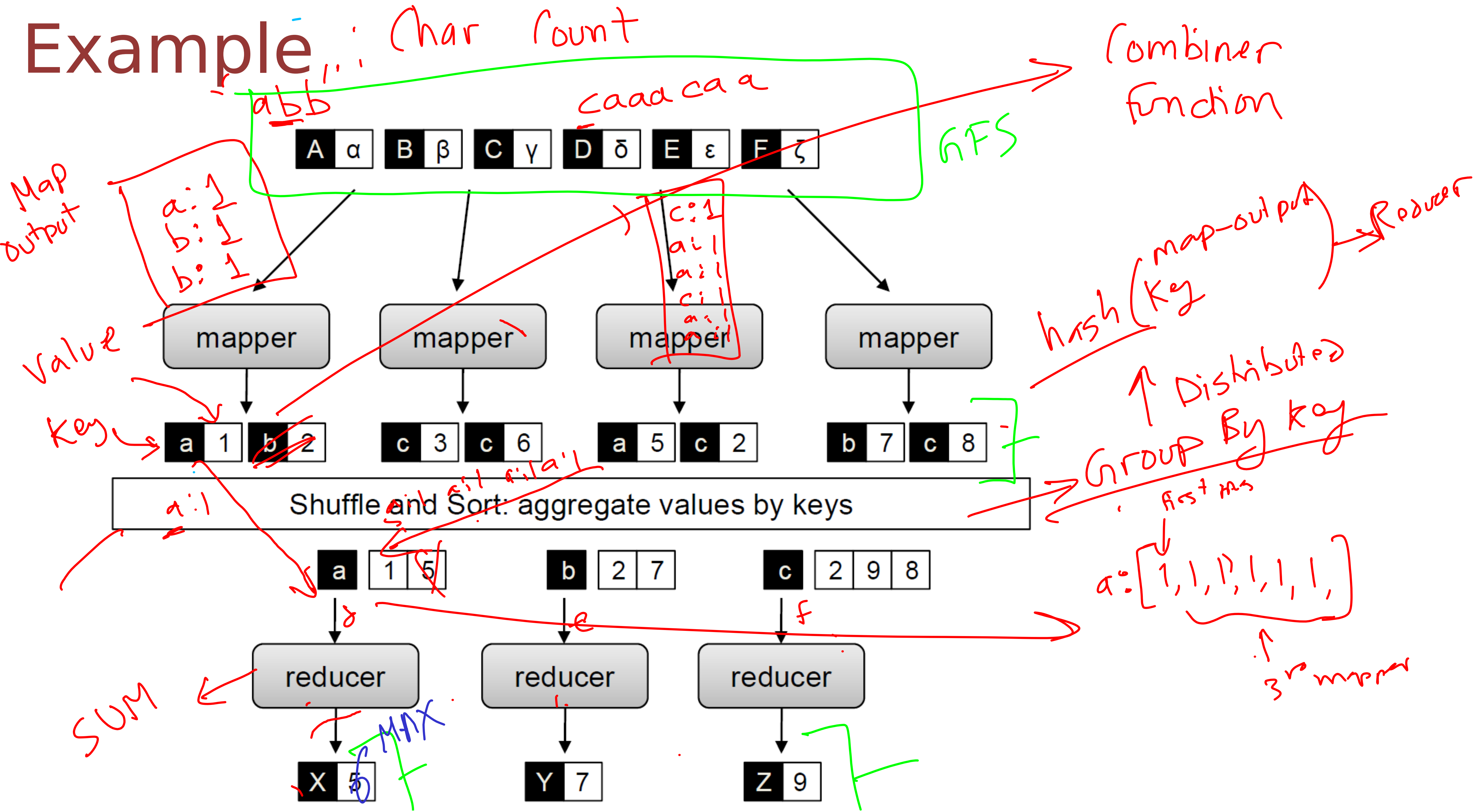
Barrier()
Wait

Barrier()

Computation Flow

- Map tasks turn chunks into series of key-value pairs based on user-provided fn
- Key-value pairs from all map tasks are collected and sorted/grouped by key
- Each reducer task gets a subset of keys
  - Master uses a hash function to assign a key to one of R reducers
  - All key-val pairs with the same key are processed by the same reducer task
- Reducer tasks process one key at a time, and combine all values associated with that key based on reduce function

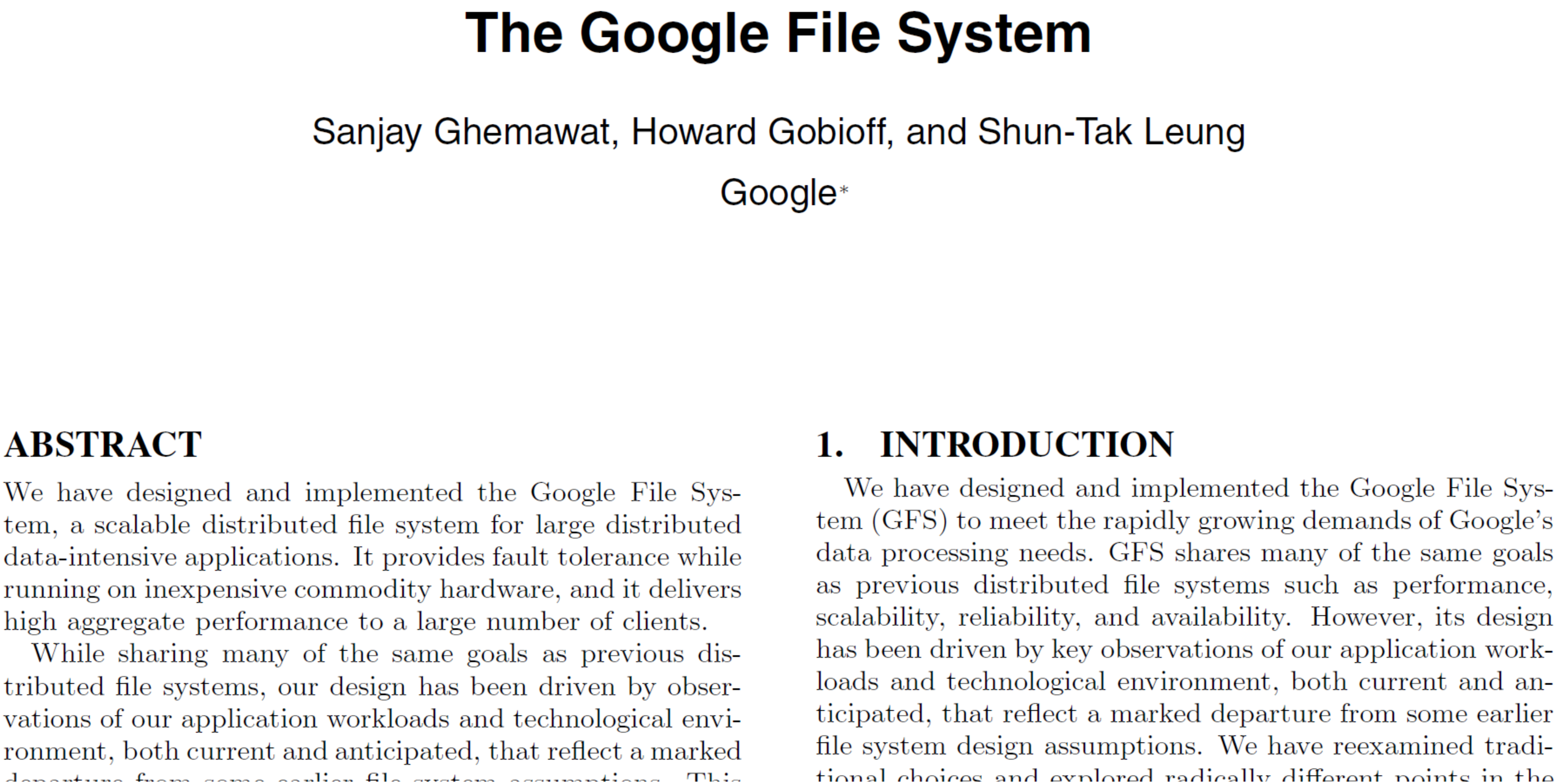
Example


Char Count
abb
caaa ca a
A α
B β
C γ
D δ
E ε
F ζ
GFS
Combiner function
Map output
a: 1
b: 1
b: 1
c: 1
a: 1
a: 1
c: 1
a: 1
mapper
mapper
mapper
mapper
Value
Key
a 1
b 2
c 3
c 6
a 5
c 2
b 7
c 8
hash(Key
map-output
Reducer
Distributed
Group By key
Shuffle and Sort: aggregate values by keys
a:1
c:1
a:1 a:1
first map
a 1 5
b 2 7
c 2 9 8
a: [1, 1, 1, 1, 1, 1,]
3rd mapper
d
e
f
reducer
reducer
reducer
SUM
MAX
X 5
6
Y 7
Z 9

Distributed File System

- MapReduce distributes **computation**
- Distributing and managing **data** is as important, if not more
- Storing data centrally:
  - I/O bottleneck
  - Processing data in parallel not very useful if limited by single disk

Data Storage: Google File System



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This

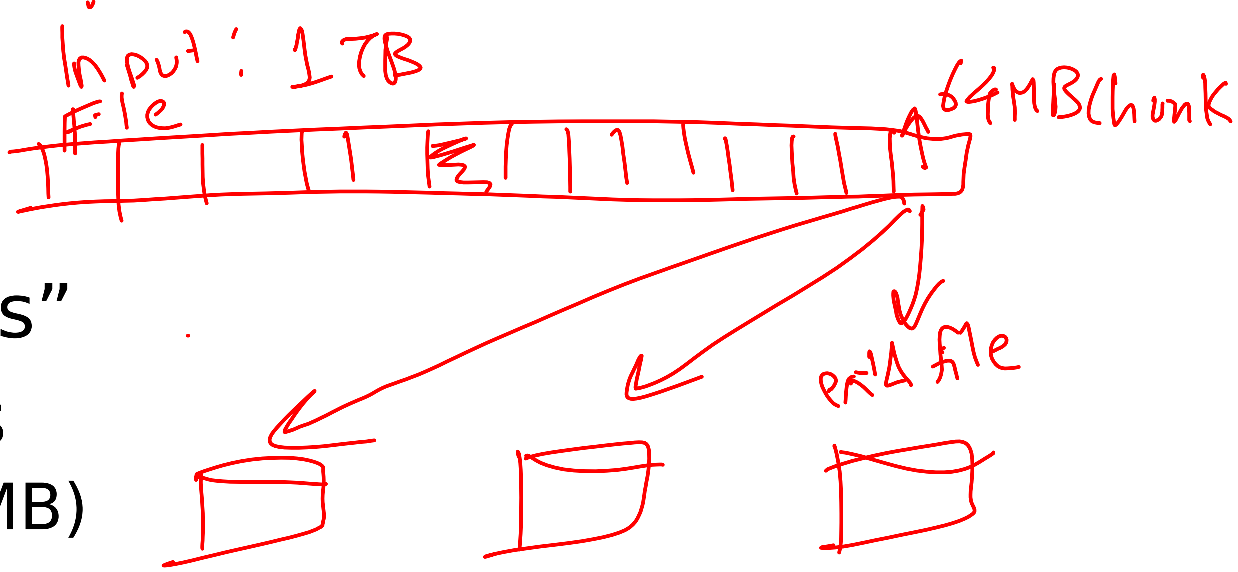
1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. We have reexamined traditional choices and explored radically different points in the

Google File System

- Distributed FS for large datasets
- Supports read, write, open, close, record-append operations
- Can run on off-the-shelf computing clusters
  - No specialized hardware required
- Provides fault tolerance via replicating each piece of data three times
  - Replicas can also be used for load balancing

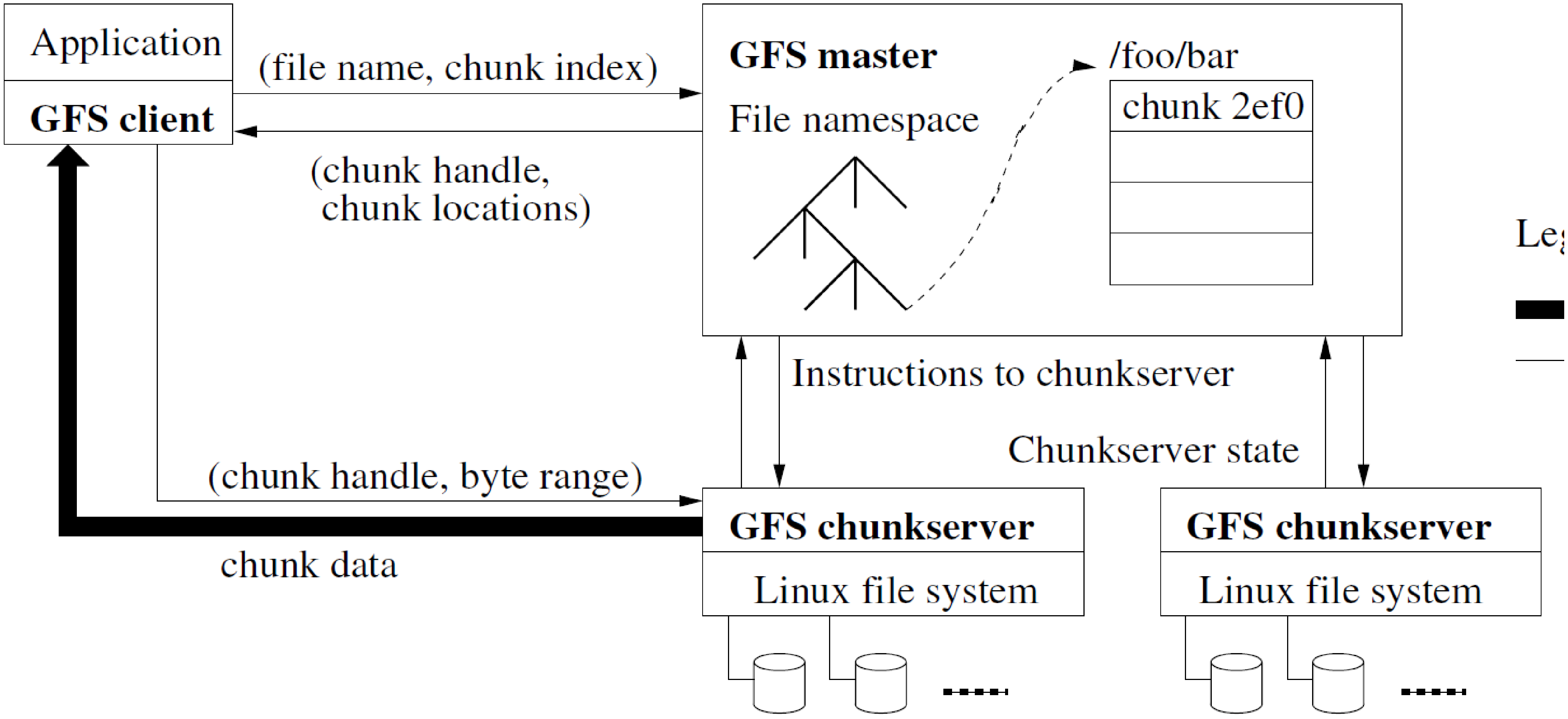
Overlay File Systems



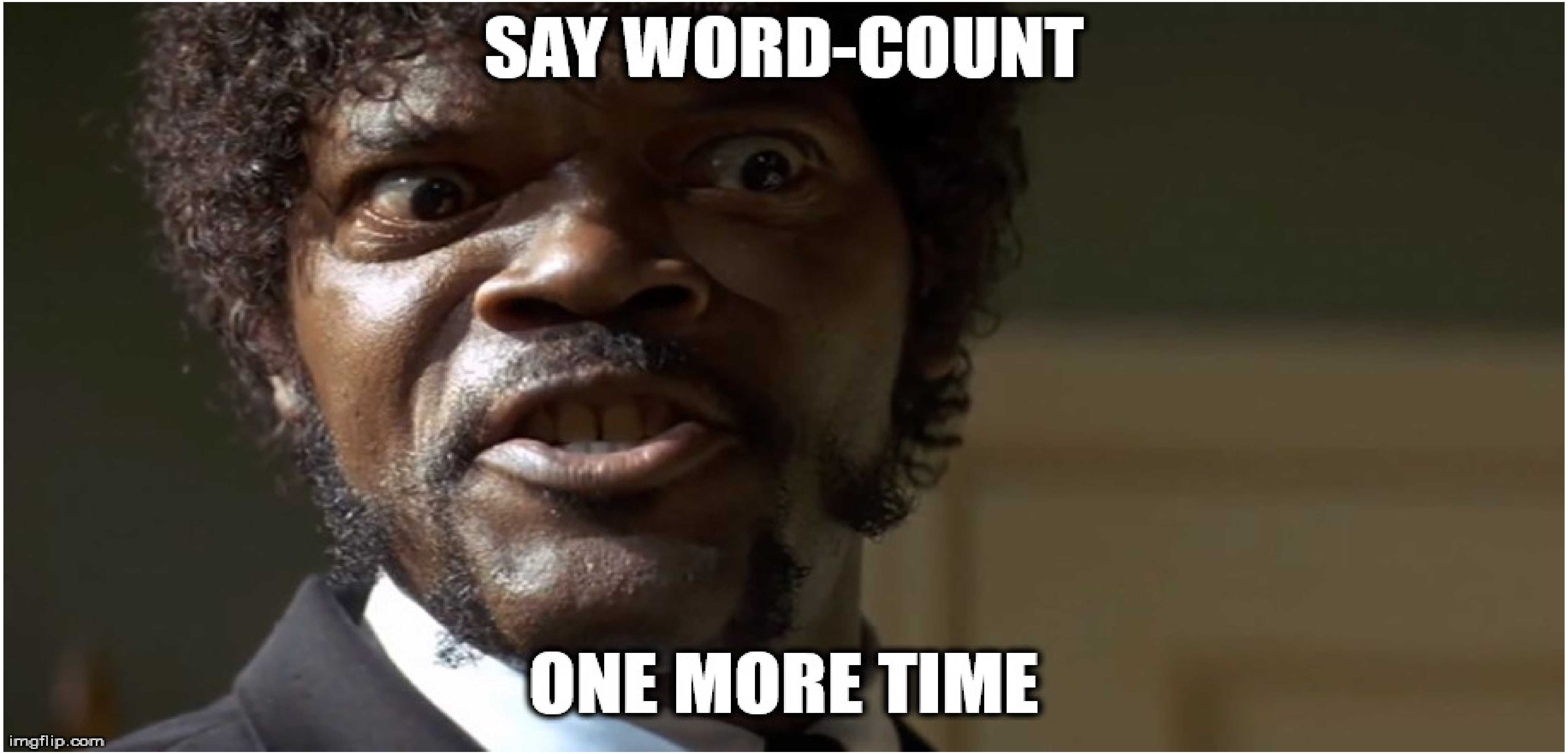
- Data is stored in “chunk servers”
- HDFS files: sequence of chunks
  - Chunks are just large blocks (64MB)
- Chunk-servers store chunks as **files in a local file-system (such as ext4)**
- HDFS files are indexed by a central “namenode”
- (HDFS-file-name, chunk-number) -> Chunk-servers, chunk-handle

(Input-file-1.txt (1TB), 100) → 10.1.1.1, 1-chunk-100.out

GFS Architecture



MapReduce Examples



Word-count

Input: [list of words]

Input 2: [Words are lame]

[(list, 1), (of, 1), (words, 1)]

[(words, 1), (are, 1), (lame, 1)]

hash(list) % 3

- Mapper: Emit (word,1) pairs
- Grouping by key :
  - (the,1), (the,1),…. : Sent to Reducer 0
  - (apple, 1), (apple,1),…. : Sent to Reducer 1
  - (car, 1), (car,1)….. : Sent to Reducer 2
  - (dog, 1), (dog,1),…. : Sent to Reducer 0
  - ….

list = 1
words = 2

- Reducer function is simple addition, and each reducer outputs:
  - (the, 102)
  - (apple, 4)
  - …

Grep

- Mapper: Emit (word, line-number) pairs
- Grouping by key :
  - (word1, [line-1, line-4, line-3, line-2, line-100])
  - (word2, [line-3, line-4, line-6, line-7])

Intermediate output

- Map output is usually sorted by key:
  - (word1, [1,2,3,4,100])
  - (word2, [3,4,6,7]

Reducer → Merge

- Reducer:
  - Do nothing / Identity / Null

Distributed Sort

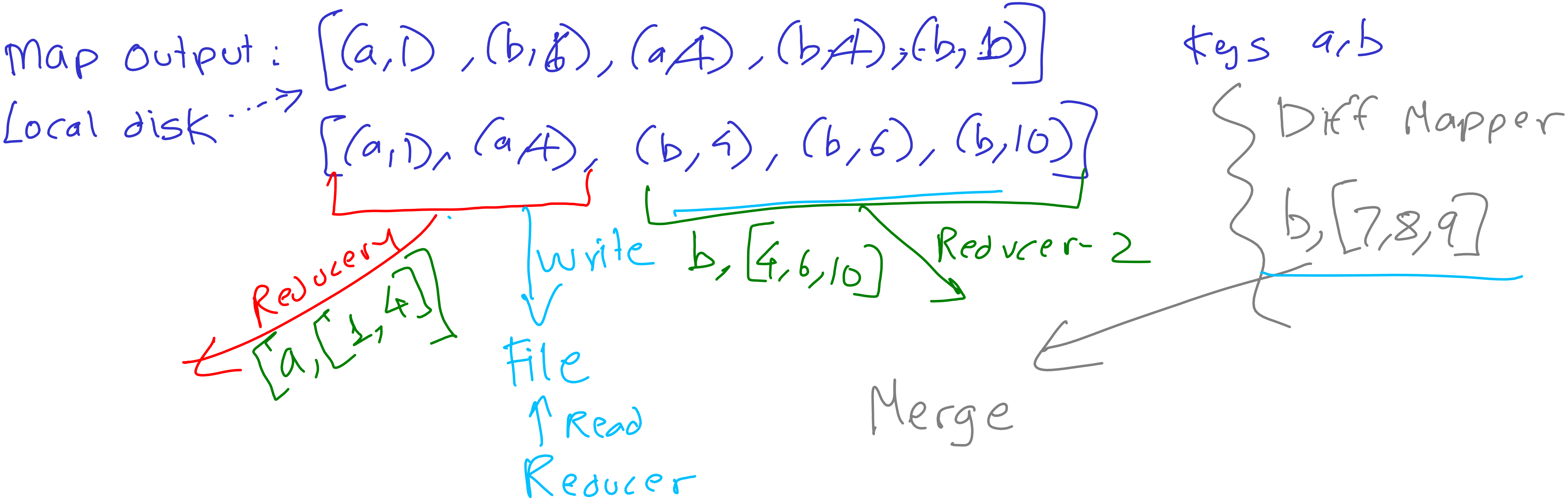
Combiners

- Sometimes, Reduce function is associative and commutative
  - f(f(a, b), c) =f(a, f(b, c))
  - f(a, b) = f(b, a)
  - Example: Addition
- In such cases, can combine map-output before sending to reducer
- In case of word-count example:
  - Map-output: [(word-1,1), (word-1, 1),…,, (word-2,1)…]
  - With combiner: [(word-1, 20), (word-2, 12)]

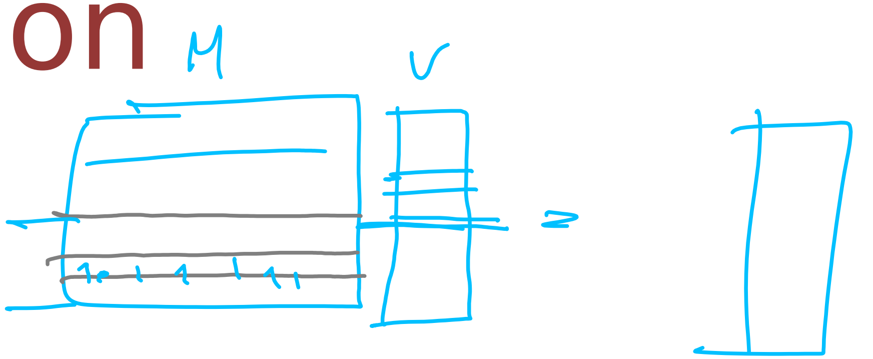
User specified

Map Output (Intermediate Data)

- Each mapper groups its output by key
  - Usually done by sorting entire local map output
- Creates an intermediate file for each reducer
- If M mappers and R reducers: up to M*R total intermediate files



Matrix-Vector Multiplication

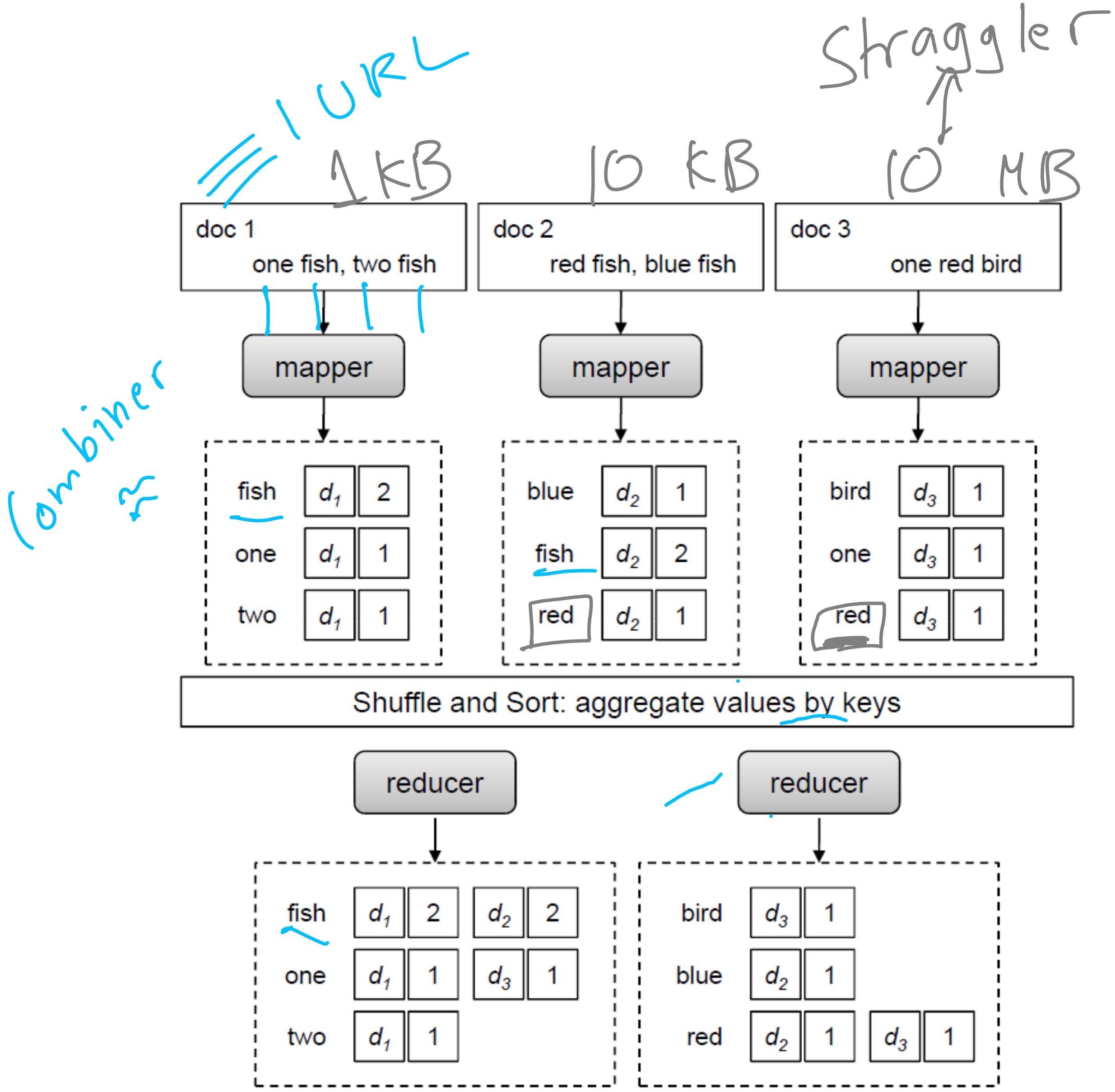


- Matrix, M and vector, v. Want to multiply : Mv
  - V=[v1,…vn]. M is nXn matrix
- Matrix partitioned by row
- Map: outputs $\boxed{(i, m_{ij} * v_j) \quad \forall j}$
- Reduce: sum up all values for each key to output (i, x_j)

- QS: What if we can't fit v in memory?
- QS: Co-occurrence matrix of words in a corpus

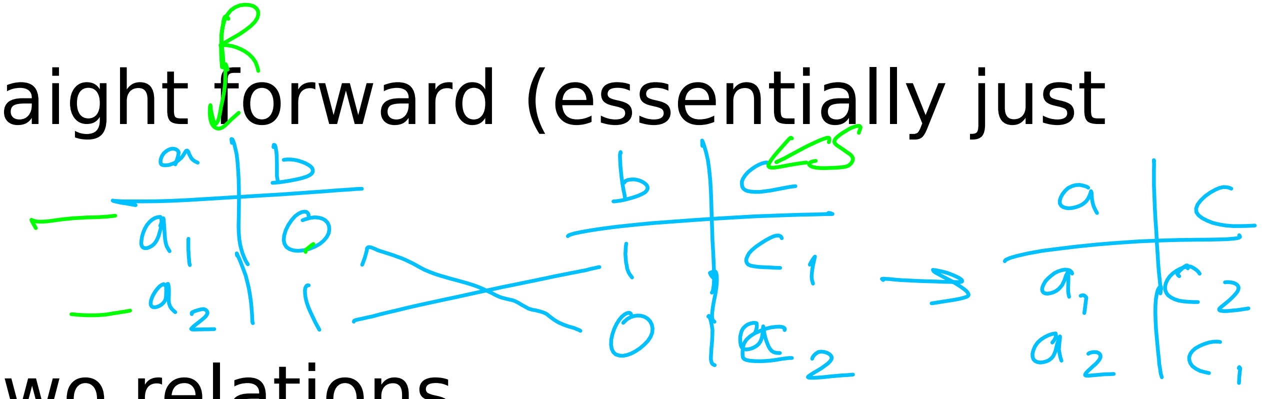
Inverted Index

- How many times does a word occur in each document?



Relational Operations

- MapReduce can also be used to run relational operations
  - Select, Project, Joins, …
- Selection and projection are straight forward (essentially just filtering using map)
- Natural joins with MapReduce
- Assume: R(a,b) and S(b,c) are two relations
- Map: Emit (b, (R, a)) Where R is just the relation name and a "tag"
- Reduce: For each key (b), output all tuples in the values list (a,c)
- Apache Hive translates SQL queries into a MapReduce program



a	b
a_1	0
a_2	1

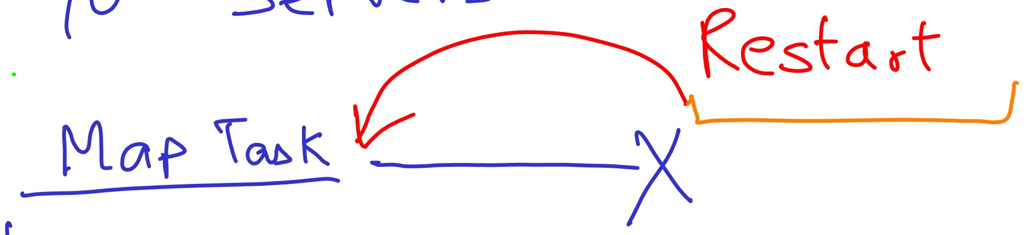
b	c
1	c_1
0	c_2

a	c
a_1	c_2
a_2	c_1

Fault-Tolerance

10^4 Servers

- Master fails → Primary Secondary Replication
  - Switch to secondary master
  - Restart entire job if no replicated master
- Worker fails
  - If running map: restart map tasks on available/free worker nodes
  - If running reduce: restart reducer tasks
- File-system (HDFS/GFS) has a single master node
  - FS index stored in memory

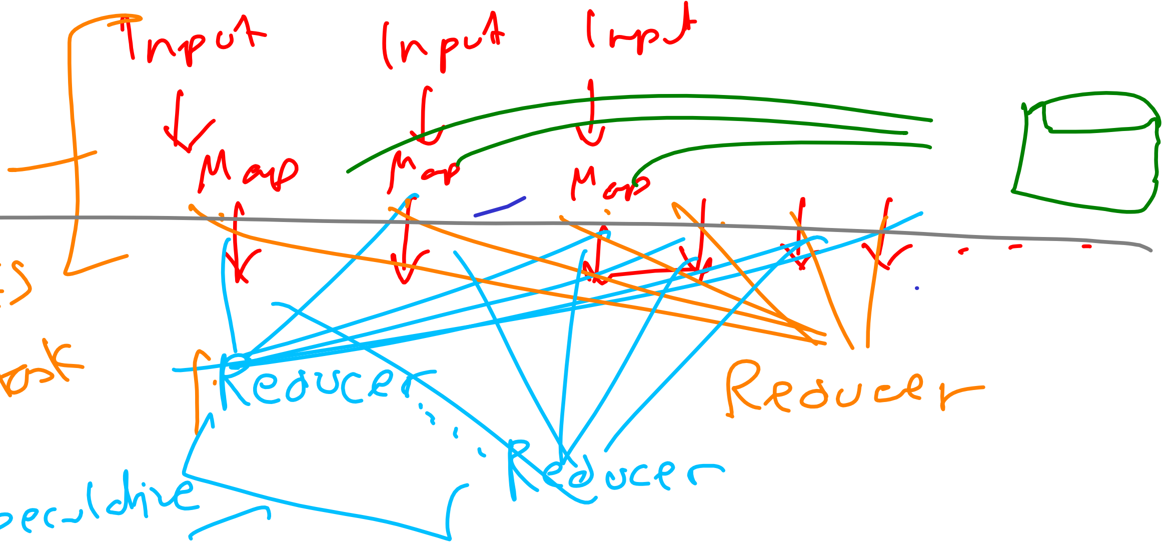


Idempotent: Same output for same input

timestamp

Barrier

Worst case: Rerun all map tasks + reducer task



Performance Issues

- Map output barrier : Even partitioning of mappers workload required
  - Usually achieved by evenly splitting the input
  - Assumes that element-wise map function has uniform cost
- Speculative execution (backup tasks)
  - Run the **same** task on multiple workers
  - If some workers are slow (stragglers)

Hardware slowdowns

- Shared H/w

- Reducer skew: A major problem
- Recall that all values of a specific key must be handled by same reducer
- What if there's a really popular key?

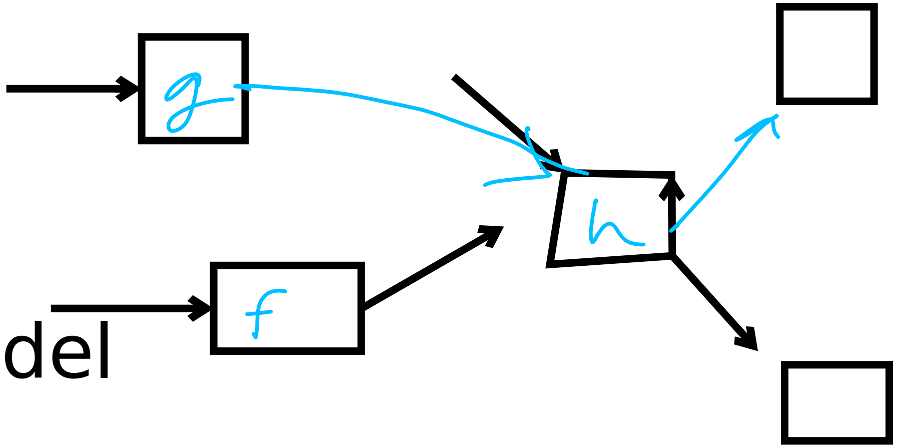
"the" — 10^6

Perf dictated by time(slowest Mapper) + GrpBy + time (slowest Reducer)

MapReduce Limitations

Idempotent

- Static data
  - Append-only usually OK
- Restrictive programming model
  - No support for dataflows



- How many mappers and reducers? → CPU, Mem
- How much memory to allocate? (
- Purely batch processing
  - Jobs can take hours to complete
  - No streaming, interactive analytics

Stream of Input data

When NOT To Use MapReduce

- Modern computing hardware has plenty of computing power:
  - A typical laptop: 8 cores, 16 GB RAM, 512 GB SSD (usually NVMe)
  - A typical server: 64 cores, 256 GB RAM, 2 TB SSD, …
- Is your dataset really that large?
  - Wikipedia: 30 GB
- In many cases, an optimized non-distributed implementation beats a large cluster (!)
  - Frank McSherry’s blog

+ Laptop