CE 528 Cloud Computing

Lecture 3: Data-parallel Computation

Fall 2026

Prof. Yigong Hu

BOSTON UNIVERSITY

Administrivia

Paper quizzes start today

- 10 multiple-choice questions in 10 minutes, at the end of class
- Open book and open internet

Demo 1 is 09/23: push your design proposal to branch demo-1 by 12:00 noon

Check you can push to your team repository - do not wait until demo day

AWS Credits

Your team's AWS credit codes are in your email

- The amount differs by team

Use one team account - redeem every code into it

- Redeeming needs a credit card on the account

Set a billing alarm the same day - overruns are billed to you

Never commit AWS keys - your repo is public

- Full instructions: ec528.github.io/ec528/fall26/setup

What Is a Distribute System?

A set of cooperating computers that are communicating with each other over network to get some coherent task done

- multiple cooperating computers
- storage for big web sites, MapReduce,
- peer-to-peer sharing

Why Do People Build Distributed Systems

High performance

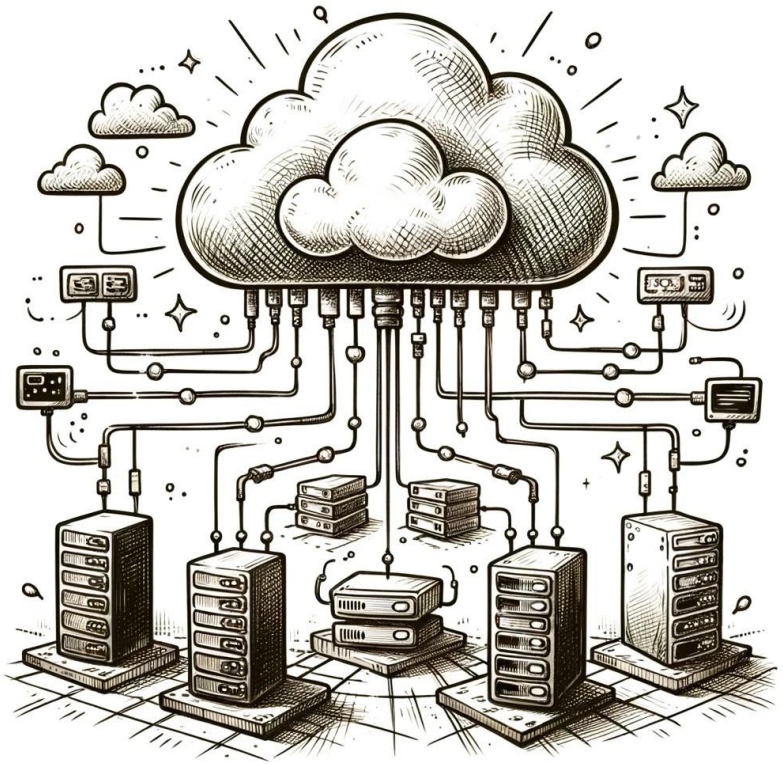
- Achieve parallelism

Fault Tolerance

Physical reason

Security

Why building distributed systems is hard?



Imagine you are operating a Starbucks

Case Study: Starbucks

Case Study: Starbucks

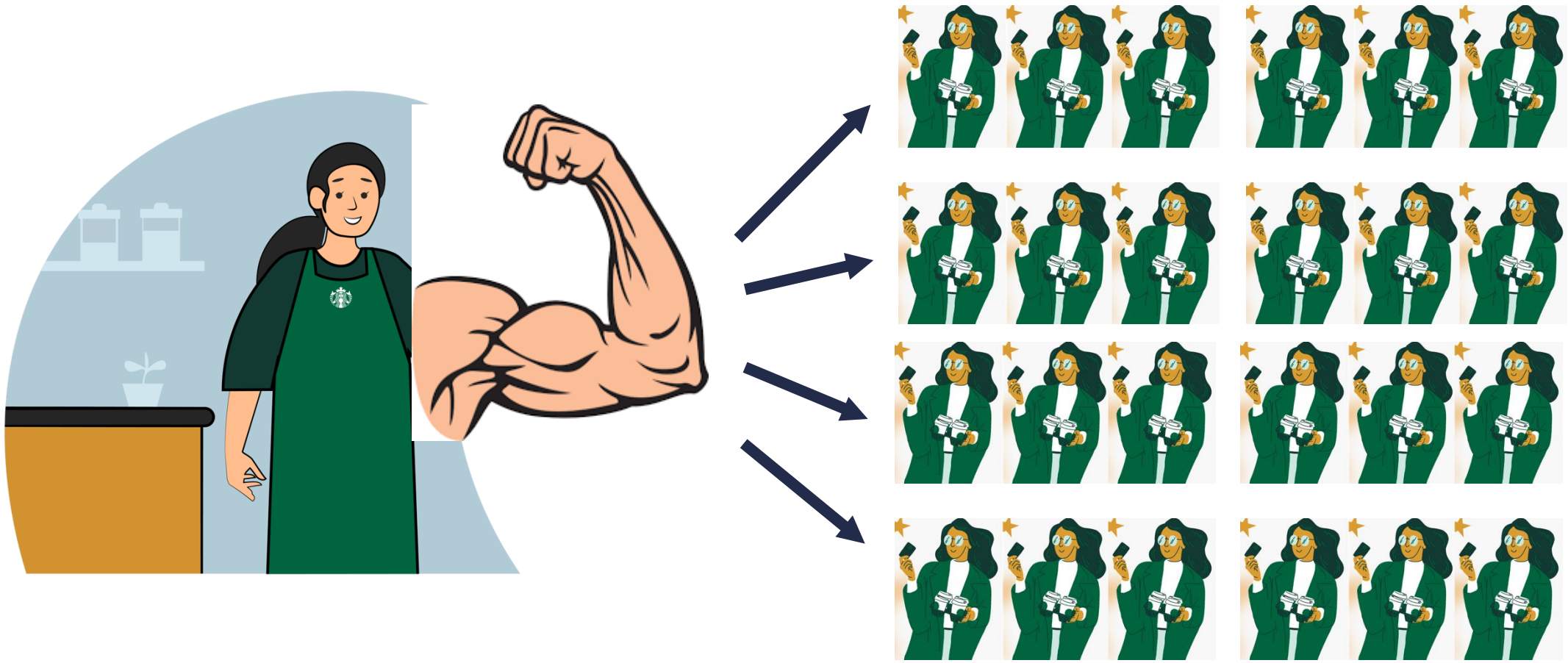
Now imagine 1000x customers?

Challenge 1: Ever-growing Load

Data is big. Users are many. Requests are even more.

Google get 8.5 billion searches per day.

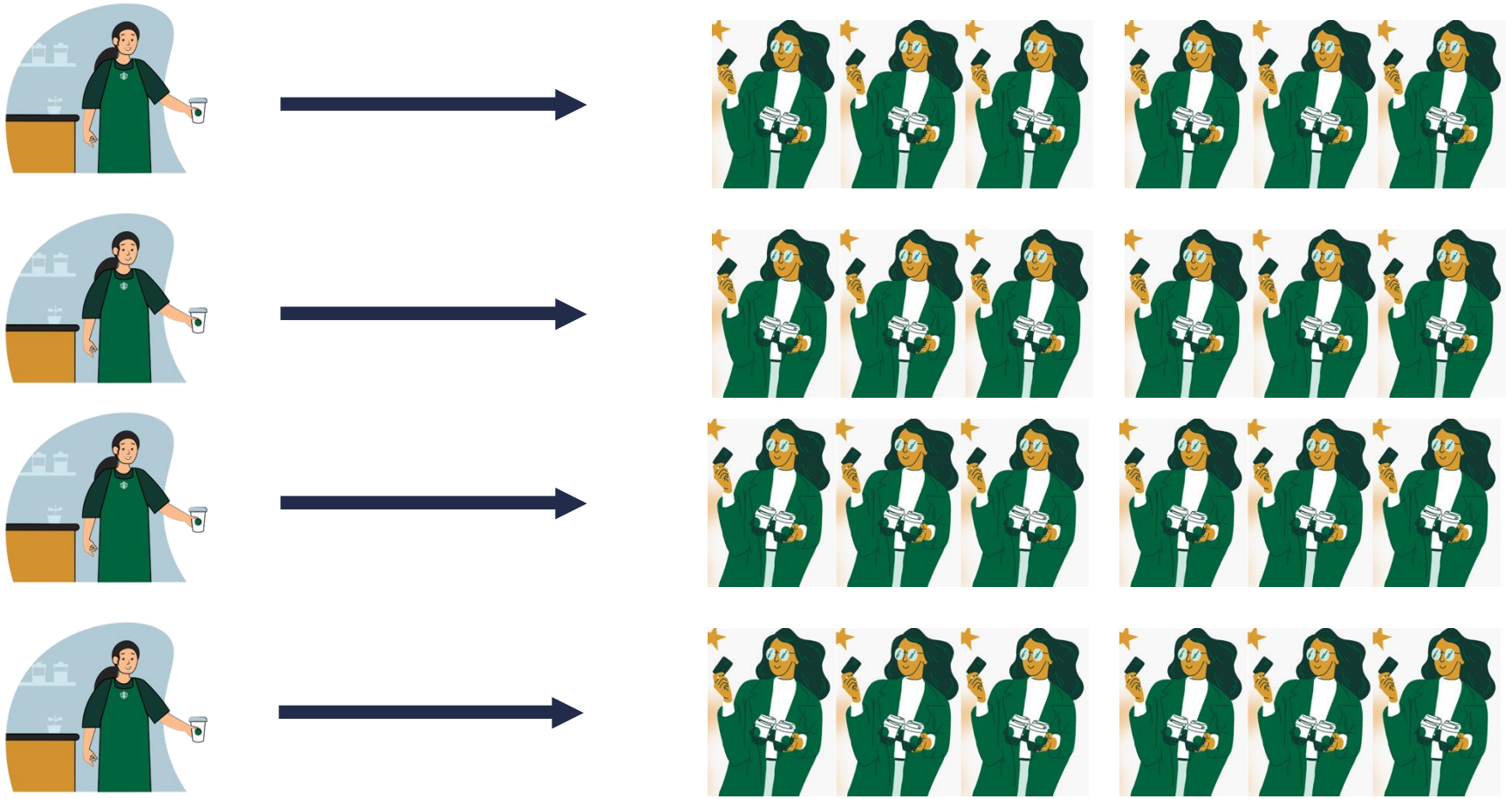
Scale-up?



Scale-up?

- You can always add more compute resources—such as CPU, memory, and disk capacity.
- But no single machine can handle the ever-growing load

Approach 1: Scale-out (Sharing)!



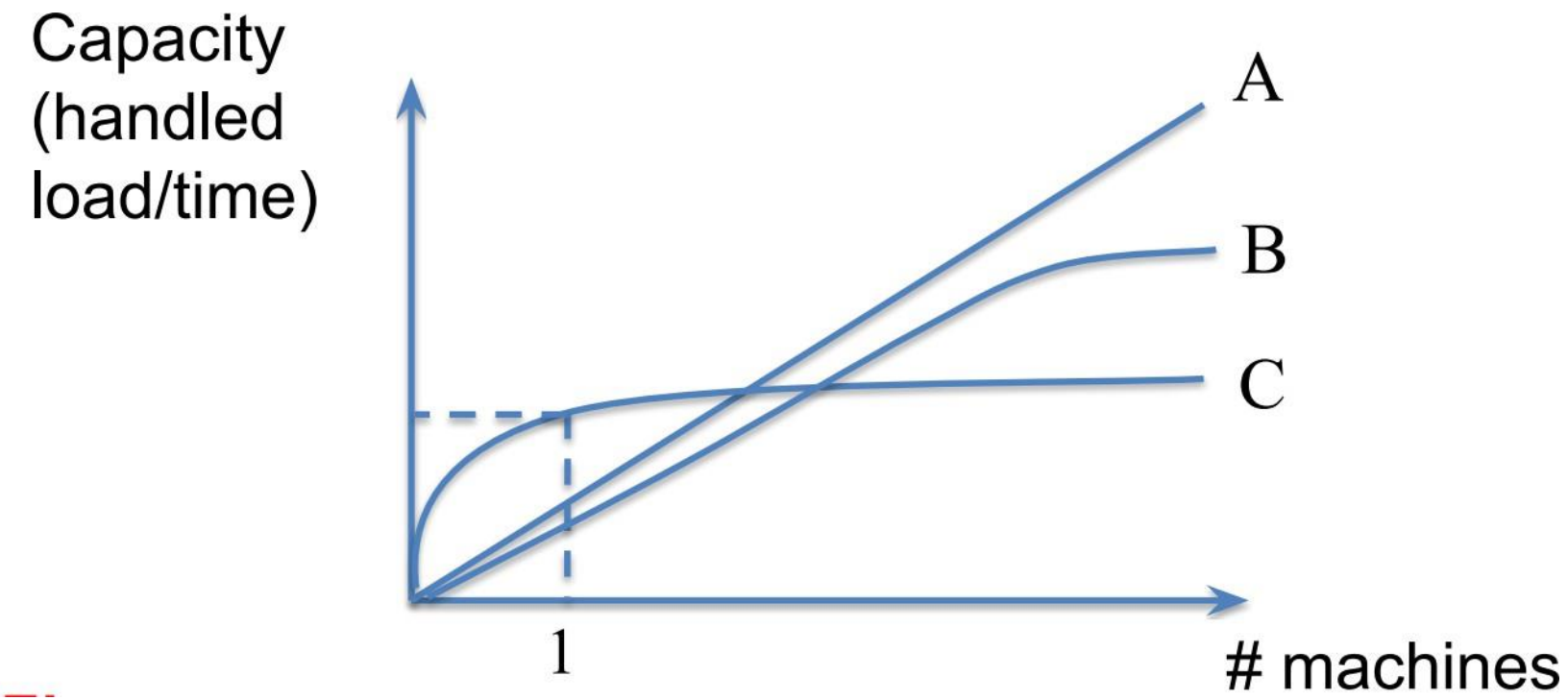
Goal 1: Scalability

The more resource you add, the more requests you can serve.

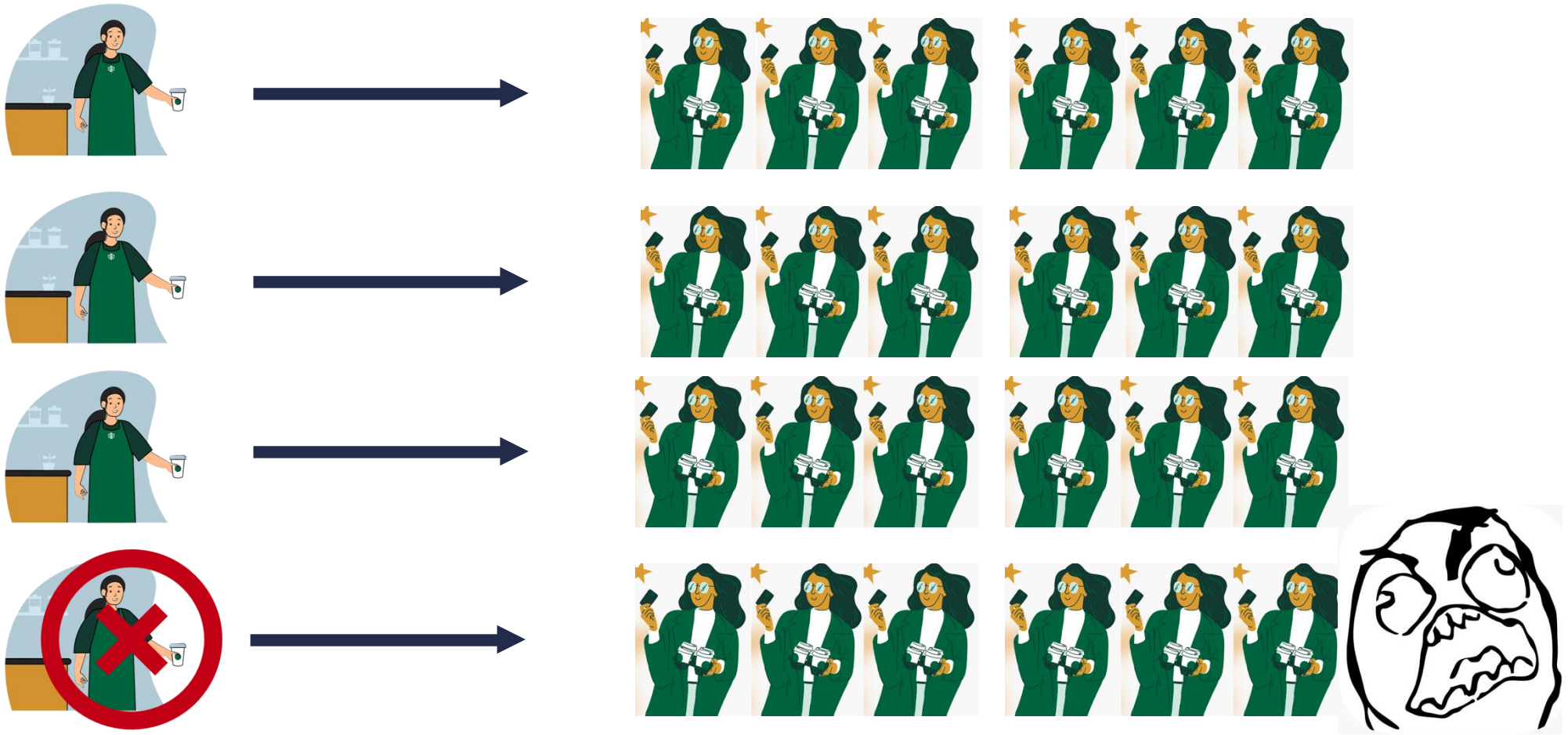
But never hope for perfect scalability

- add one machine, increase your capacity proportionally forever?

Sample Scalability Curves



Challenge 2: Failure



Goal 2: Fault Tolerance

Goal is to hide failures as much as possible to provide a service that e.g., finishes the computation fast despite failures, stores some data reliably despite failures, …

Fault tolerance subsumes:

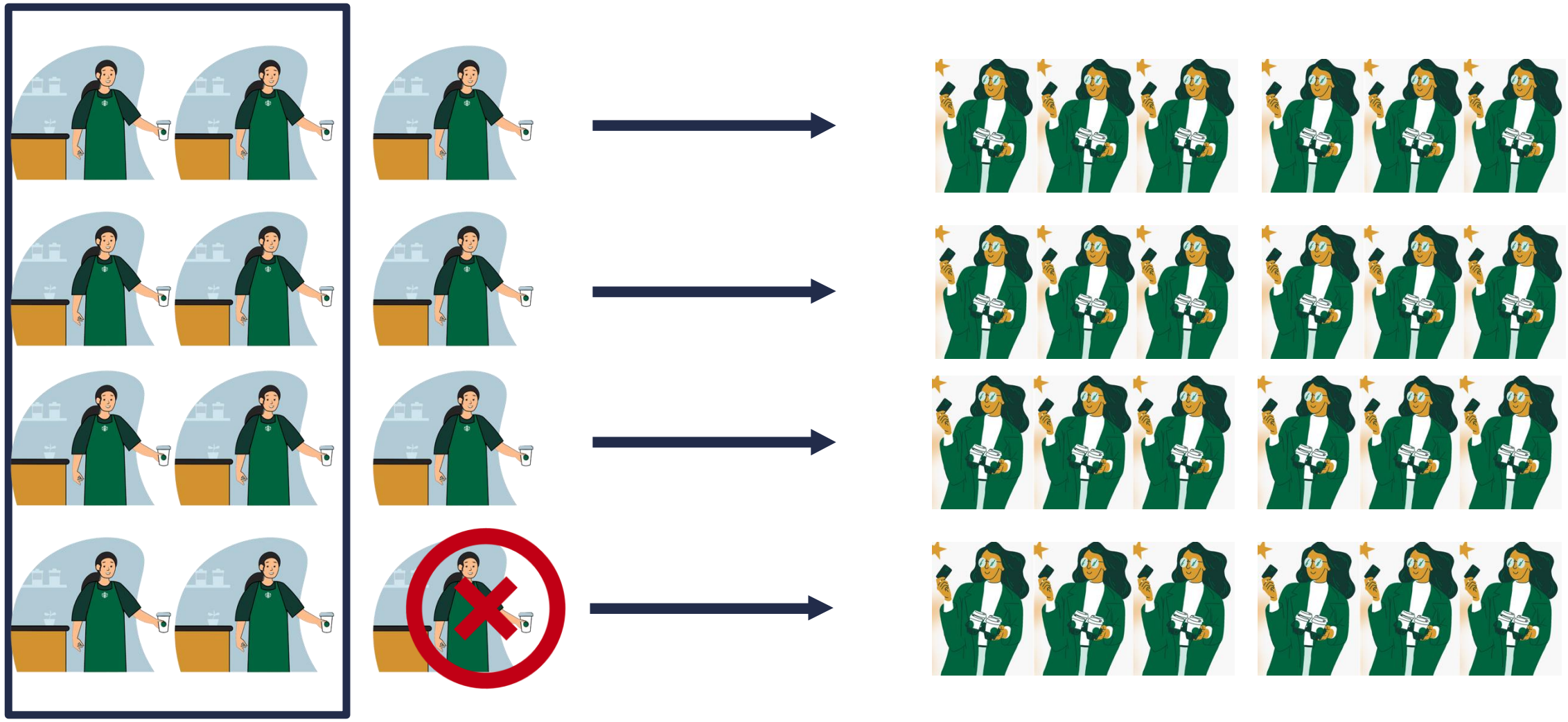
- Availability: the service/data continues to be operational despite failures.
- Durability: some data or updates that have been acknowledged by the system will persist despite failures and will eventually become available

Goal 2: Fault Tolerance

Availability: I expect someone will always take my order ..

Durability: staff remember my choice and wouldn't ask me again (even w/ failures)..

Approach 2: Replication

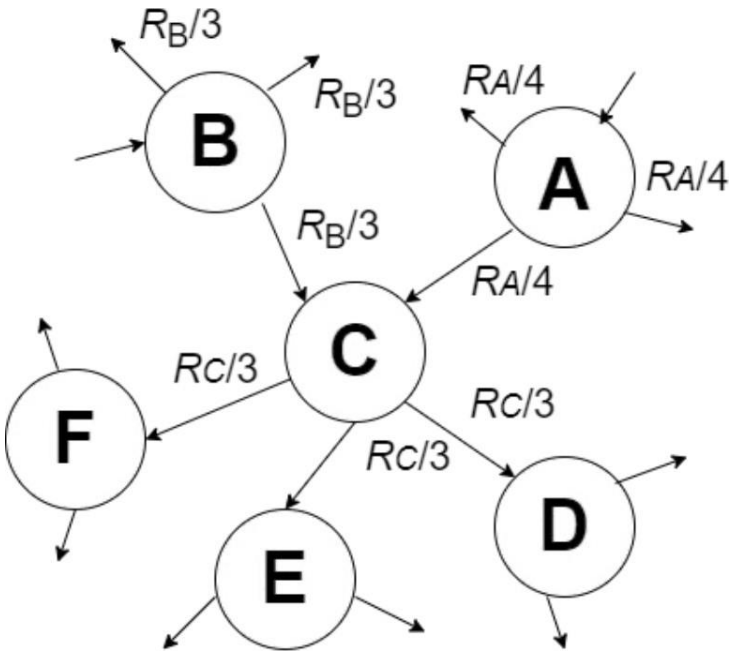


MapReduce: Background

- **Early 2000s at Google:** web indexing, PageRank, and log analysis required recurring batch jobs over tens of terabytes.
- **The recurring systems work:** each team had to partition data, schedule tasks, move intermediate results, and recover from failures.
- **MapReduce, Dean and Ghemawat, 2004:** the programmer supplies Map and Reduce functions.
- **A shared runtime:** executes the job across thousands of commodity machines and hides most distribution details.

Large-Scale Analytics

- WordCount: Compute the occurrence of words in a corpus of documents.
- Count how many times users have clicked on each of a (large) set of ads.
- PageRank: Compute the “importance” of a web page based on the “importances” of the pages that link to it



Option 1: SQL

Before MapReduce, analytics mostly done in SQL, or manually.

Example: Count word appearances in a corpus of documents.

With SQL, the rough query might be:

```
SELECT COUNT(*) FROM (
    SELECT UNNEST(string_to_array(doc_content, ' ')) as
    word FROM Corpus)
 GROUP BY word
```

Very expressive, convenient to program

But no one knew how to scale SQL execution!

Option 2: Manual

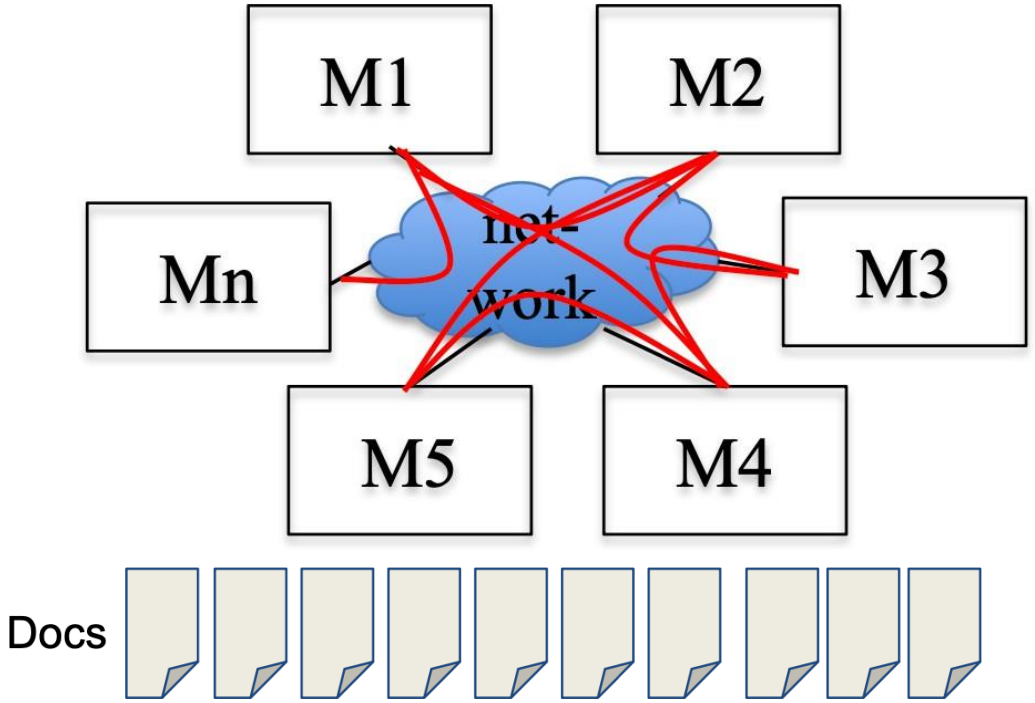
Example: Count word appearances in a large corpus.

Phase 1: Split documents across workers.

- Each worker computes {word: local_freq}.

Phase 2: Group partial counts by word.

- Send each (word, local_freq) pair to worker hash(word) % N.
- Each worker sums all values for its assigned words.



Limitation of Option 2

How to generalize to other applications?

How to deal with failures?

How to deal with slow nodes?

How to deal with load balancing (some docs are very large, others small)?

How to deal with skew (some words are very frequent, so nodes designated to aggregate them will be pounded)?

They are common challenges for large-scale analytics!

Part 1: The Programming Model

- **What the programmer writes: Map and Reduce**
- **What the model promises: all values for the same key are grouped together**
- **Not yet: machines, tasks, networks, failures**

MapReduce

Parallelizable programming model:

- Consists of three phases, each intrinsically parallelizable:
  - Map: processes input elements independently to emit relevant (key, value) pairs from each.
  - Transparently, the runtime system groups all the values for each key together: (key, [list of values]), called "shuffle".
  - Reduce: aggregates all the values for each key to emit a global value for each key.
- Applies to a broad class of analytics applications.
- Isn't as expressive as SQL but it is easier to scale.

Scalable, efficient, fault tolerant runtime system (discuss in later slides).

How MapReduce Works

Input: a collection of elements of (key, value) pair type.

Programmer defines two functions:

- Map(key, value) → a list of (key', value') pairs
- Reduce(key, value-list) → output

Execution

- Apply Map to each input key-value pair, in parallel for different keys.
- Sort emitted (key', value') pairs to produce (key' value'-list) pairs.
- Apply Reduce to each (key', value'-list) pair, in parallel for different keys.

Output is the union of all Reduce invocations' outputs.

Word Count With MapReduce

```
Map(key, value): // key: document ID; value: document content
  FOR (each word w IN value)
    emit(w, 1);
```

```
Reduce(key, value-list): // key: a word; value-list: a list of integers
  result = 0;
  FOR (each integer v on value-list)
    result += v;
  emit(key, result);
```

Live Demo 1: The Programming Model

The code we write: Map and Reduce in Python

- Handed unchanged to Hadoop on this laptop (no cluster)
- Map only: (word, 1) pairs
- Map + Reduce: word count
- A new job by changing only Map

Exercise

(MapReduce) You are given a symmetric social network (like Facebook) where a is a friend of b implies that b is also a friend of a. The input is a dataset D (sharded) containing such pairs (a, b) – note that either a or b may be a lexicographically lower name. Pairs appear exactly once and are not repeated. Find the last names of those users whose first name is “Taylor” and who have at least 300 friends.

- Write pseudocode code for Map() and Reduce(). Your pseudocode may assume the presence of appropriate primitives (e.g., “first_name(user_id)”, etc.). The Map function takes as input a tuple (key=a,value=b)

Example: Word Frequency

Suppose instead of word count, we wanted to compute word frequency: the probability that a word would appear in a document.

This means computing the fraction of times a word appears, out of the total number of words in the corpus.

Solution: Chain two mapreduce's

First Map/Reduce: Word Count (like before)

- Map: process documents and output pairs.
- Multiple Reducers: emit for each word.

Second MapReduce:

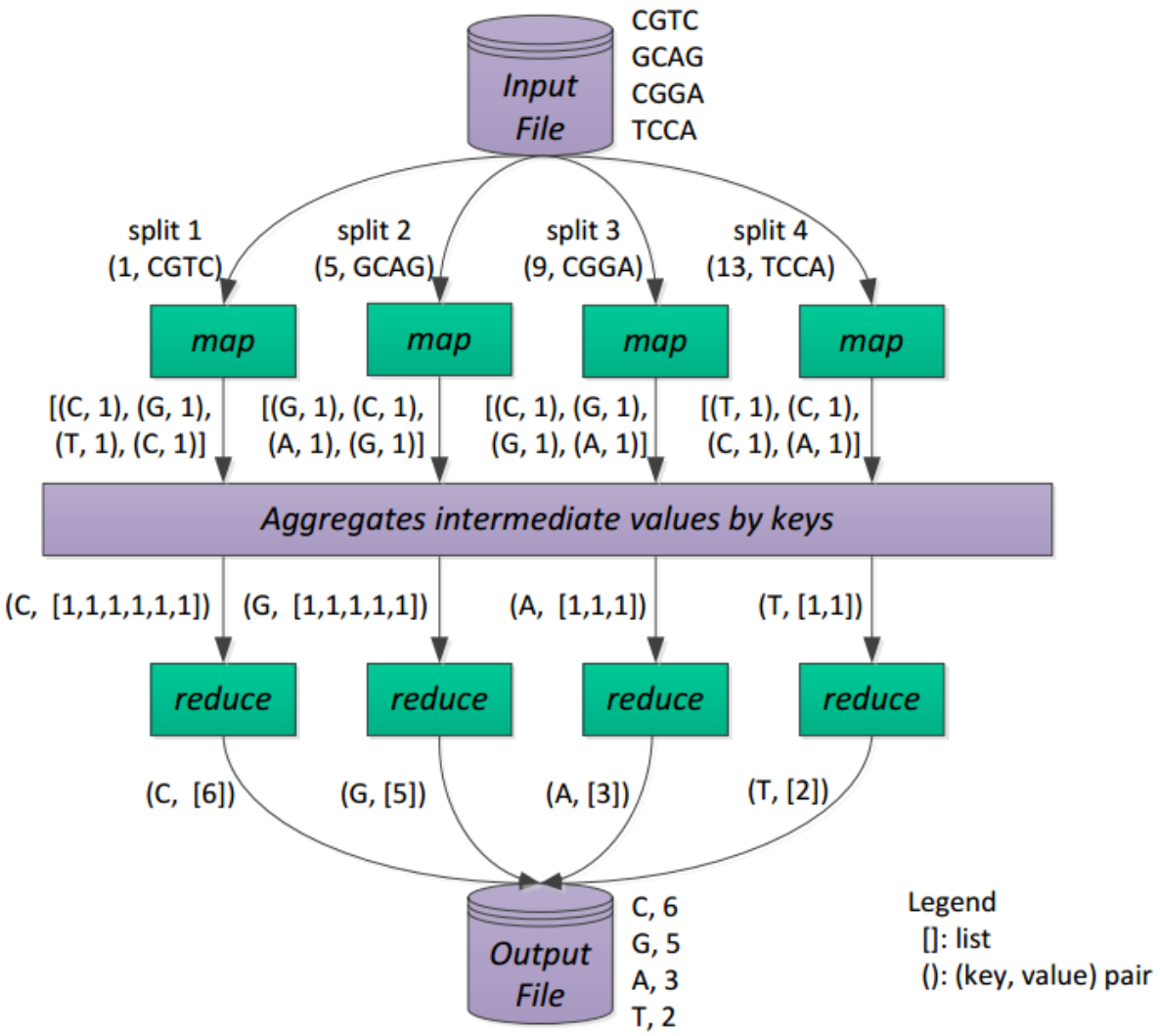
- Map: process<word, word_count> and output (1, (word, word_count)).
- 1 Reducer: perform two passes:
  - In first pass, sum up all word_count's to calculate overall_count.
  - In second pass calculate fractions and emit multiple .

Scalability is not too bad, as first stage's output is a rather small dictionary (maximum # of English words with an integer for each).

Part 2: The System Design

- **How the runtime executes the same Map and Reduce on thousands of machines**
- **Input splits and tasks, master and workers, partitioning and the shuffle**
- **Data locality, fault tolerance, stragglers**

WordCount Example



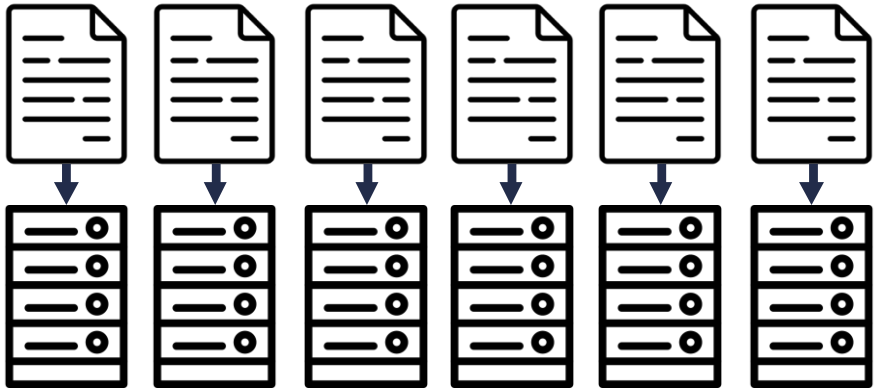
Map Phase

Mapper is given key: document ID; value: document content, say:

> D1,“The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am.”)

It will emit the following pairs:

<The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store,1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store,1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>



Intermediary Phase (Shuffle)

Transparently, the runtime sorts emitted (key, value) pairs by key:

<9am, 1>
<at, 1>
<closed, 1>
<in, 1>
<morning, 1>
<opens, 1>
<opens, 1>
<store, 1>
<store,1>
<store, 1>
<store,1>

Reducer 1

<teacher, 1>
<the, 1>
<the, 1>
<the, 1>
<the, 1>
<the 1>
<to, 1>
<went, 1>
<was, 1>
<The, 1>

Reducer 2

Reduce Phase

For each unique key emitted from the Map Phase, function Reduce(key, value-list) is invoked on Reducer 1 or Reducer 2.

Across their invocations, these Reducers will emit:

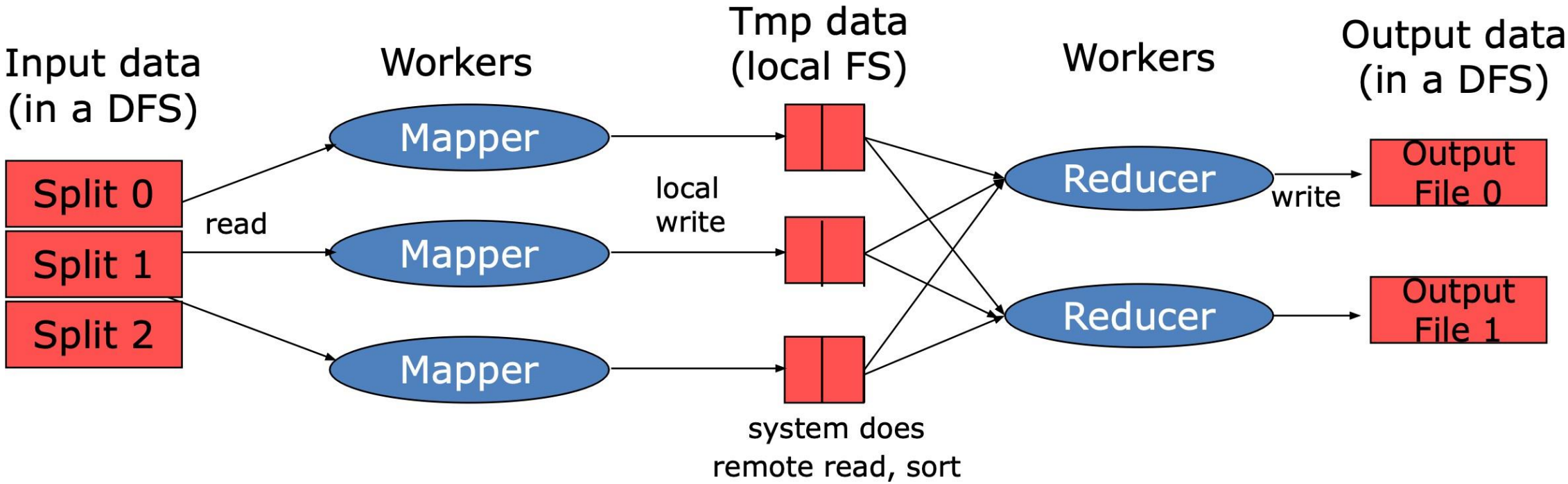
<9am, 1>
<at, 1>
<closed, 1>
<in, 1>
<morning, 1>
<opens, 2>
<store, 4>

Reducer 1

<teacher, 1>
<the, 5>
<to, 1>
<went, 1>
<was, 1>
<The, 1>

Reducer 2

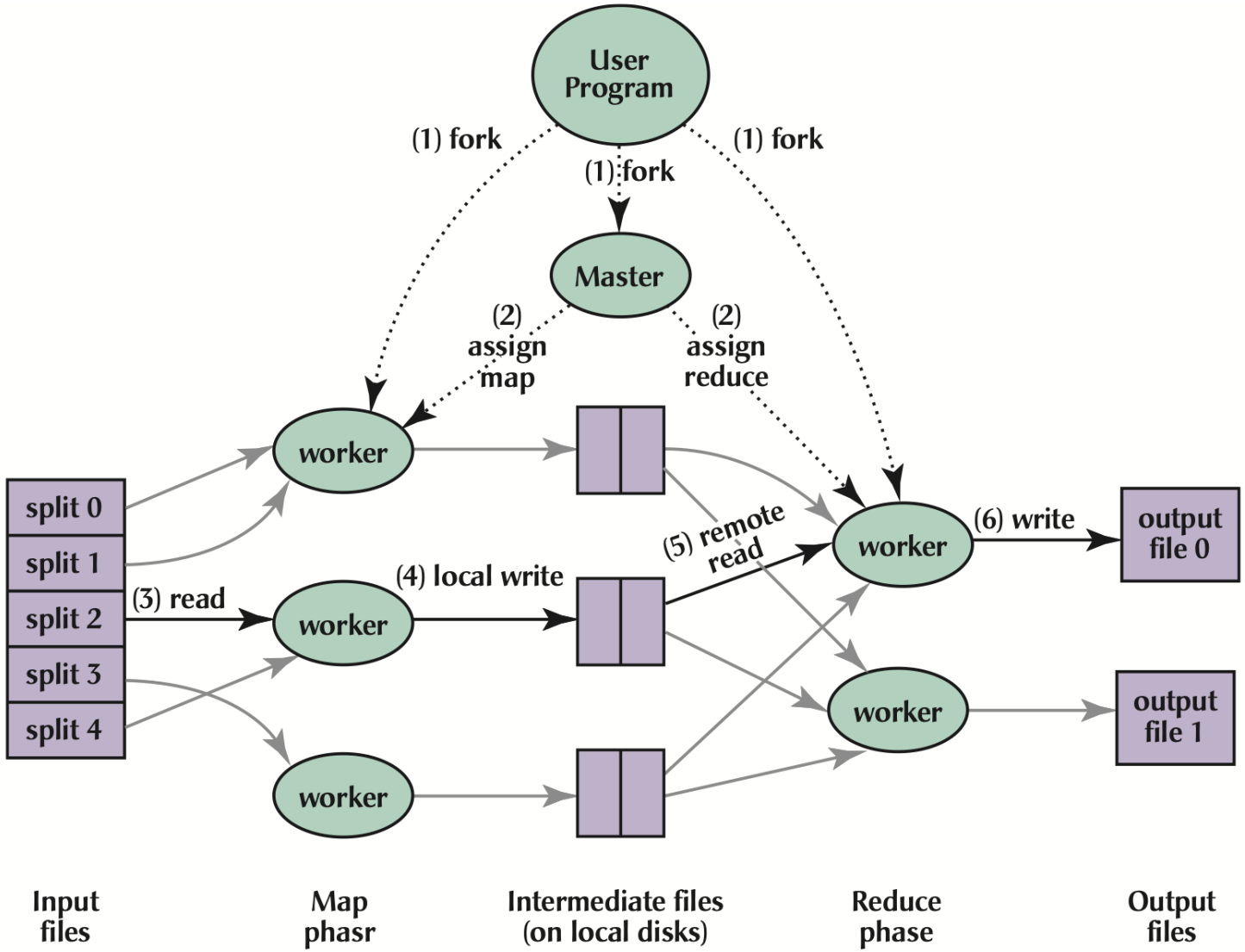
MapReduce: Workflow



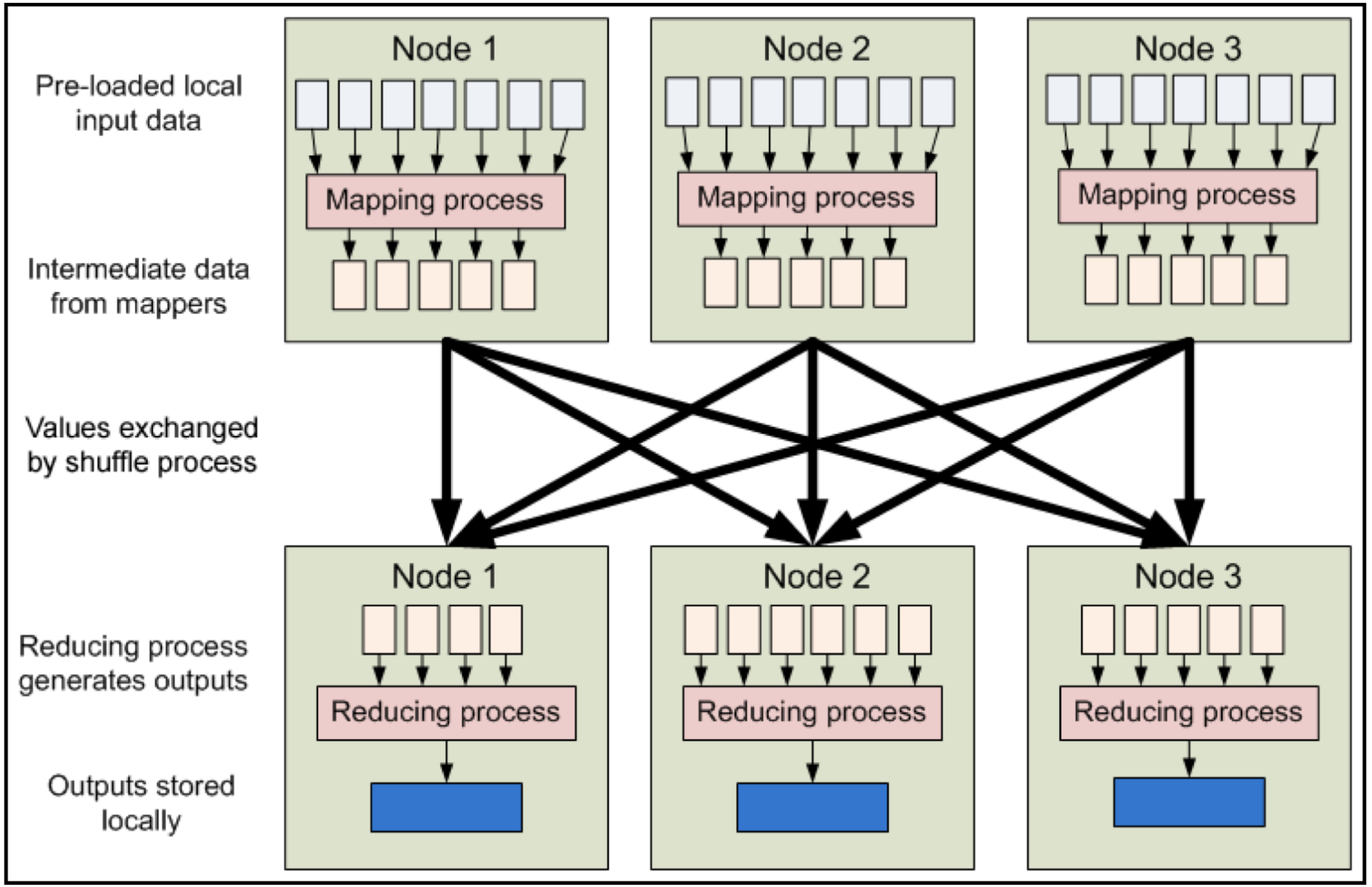
Map phase:
extract something you care about from each record

Reduce phase:
aggregate

MapReduce Architecture



Detailed Architecture

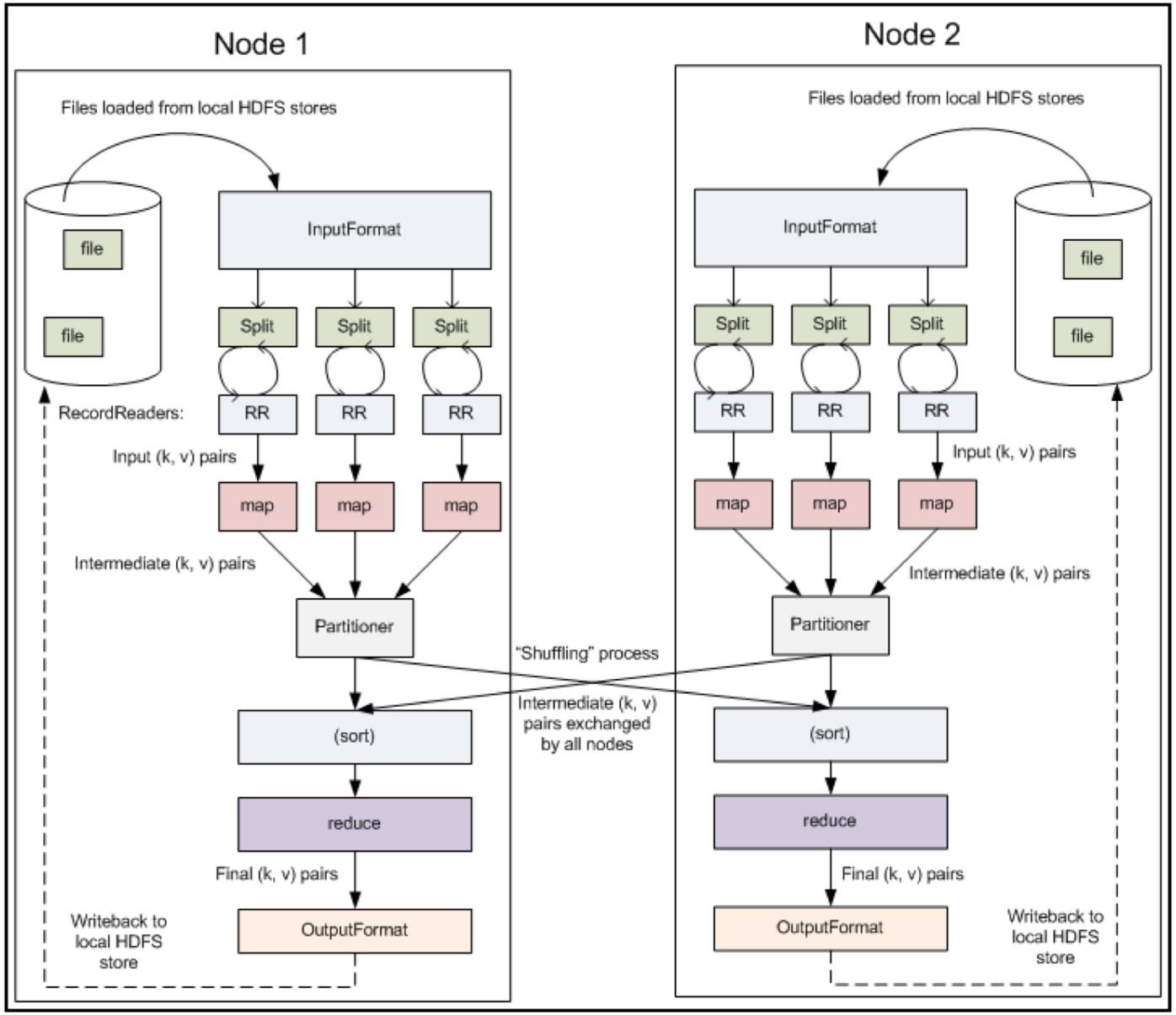


Live Demo 2: The MapReduce System

The same job: what the system did

- Input splits become map tasks
- One output file per task; output never overwritten
- The shuffle: pairs grouped by key before Reduce
- 2 reduce tasks: hash partitioning
- Same answer with 1 or 2 reduce tasks

GFS in MapReduce



Performance

A common bottleneck: network

- input data transferred all over the cluster, how to solve?

Solution: scheduling policy for data locality

- Asks GFS for locations of replicas of input file blocks.
- Map tasks are scheduled so GFS input block replica are on the same machine or on the same rack.

Effect: Thousands of machines read input at local speed.

- eliminate network bottleneck!
- But shuffle is still heavy

Fault Tolerance

Failures are the norm in data centers.

- Worker failure
- Master failure

Worker failure solution: Heartbeats

- Master detects if workers failed by periodically pinging them.
- Re-execute in-progress map/reduce tasks.

Master failure solution: Replication

- Initially, was single point of failure; Resume from Execution Log. Subsequent versions used replication and consensus.

Effect: From Google's paper: once, a Map/Reduce job lost 1600 of 1800 machines, but it still finished fine.

Stragglers

Slow workers or stragglers significantly lengthen completion time.

Slowest worker can determine the total latency!

- Other jobs consuming resources on machine.
- Bad disks with errors transfer data very slowly.
- This is why many systems measure 99th percentile latency.

Solution: spawn backup copies of tasks (redundant execution).

- Whichever one finishes first "wins.”
- I.e., treat slow executions as failed executions!