CS 582: Distributed Systems

MapReduce: Simplified Data Processing on Large Clusters

Dr. Zafar Ayyub Qazi

Fall 2024

Acknowledgement: Slides use material from Jeff Dean and Sanjay Ghemawat (Google)

Agenda: MapReduce

- Motivation
- Programming Interface
- Framework design
- Dealing with failures and slow nodes

Specific learning outcomes

By the end of today's lecture, you should be able to:

- ❑ Explain the programming model of MapReduce framework and how it simplifies big data processing
- ❑ Describe and analyze the design of the MapReduce framework
- ❑ Analyze how MapReduce deals with failures and stragglers
- ❑ Evaluate the performance speedup through MapReduce framework

A note on the authors

Jeff Dean
(Chief Scientist at Google)

Sanjay Ghemawat
(Senior Fellow at Google)

A Friendship That Made Google (New Yorker)

"Big Data"

- Huge amounts of data being collected daily

- Many applications domains
  - Business intelligence, health care, security, etc.


How to handle big data?
google.com/datacenters

Distributed grep

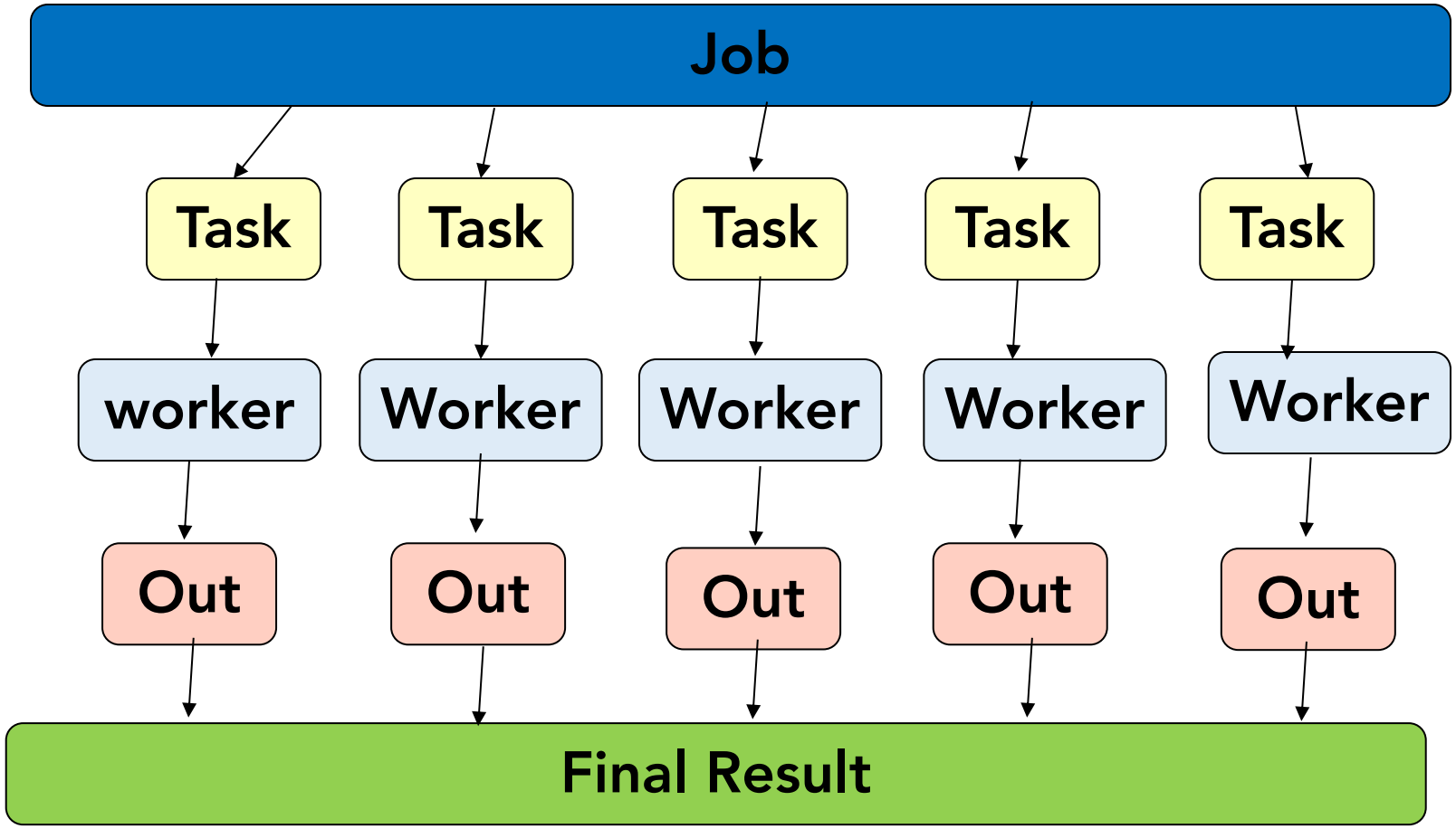
URL Access frequency

Need to process petabytes of data

Thousands of machines

Objective: Minimize the time to process data

Basic Idea: Divide and Conquer



Some Challenges

- How to partition input data?
- How to schedule tasks onto the worker nodes?
- How to communicate with workers?
- How to collect/aggregate results?
- What if workers become fail or become slow?

How can we design a distributed system that handles all the challenges of parallelism, while allowing programmers to focus on the high-level logic, not low-level implementation details?

How can we design a **distributed system** that handles all the challenges of parallelism, while allowing programmers to focus on the high-level logic, not low-level implementation details?

Q: What abstractions to provide to the programmer?

Why Abstractions?

- A fundamental design component in systems

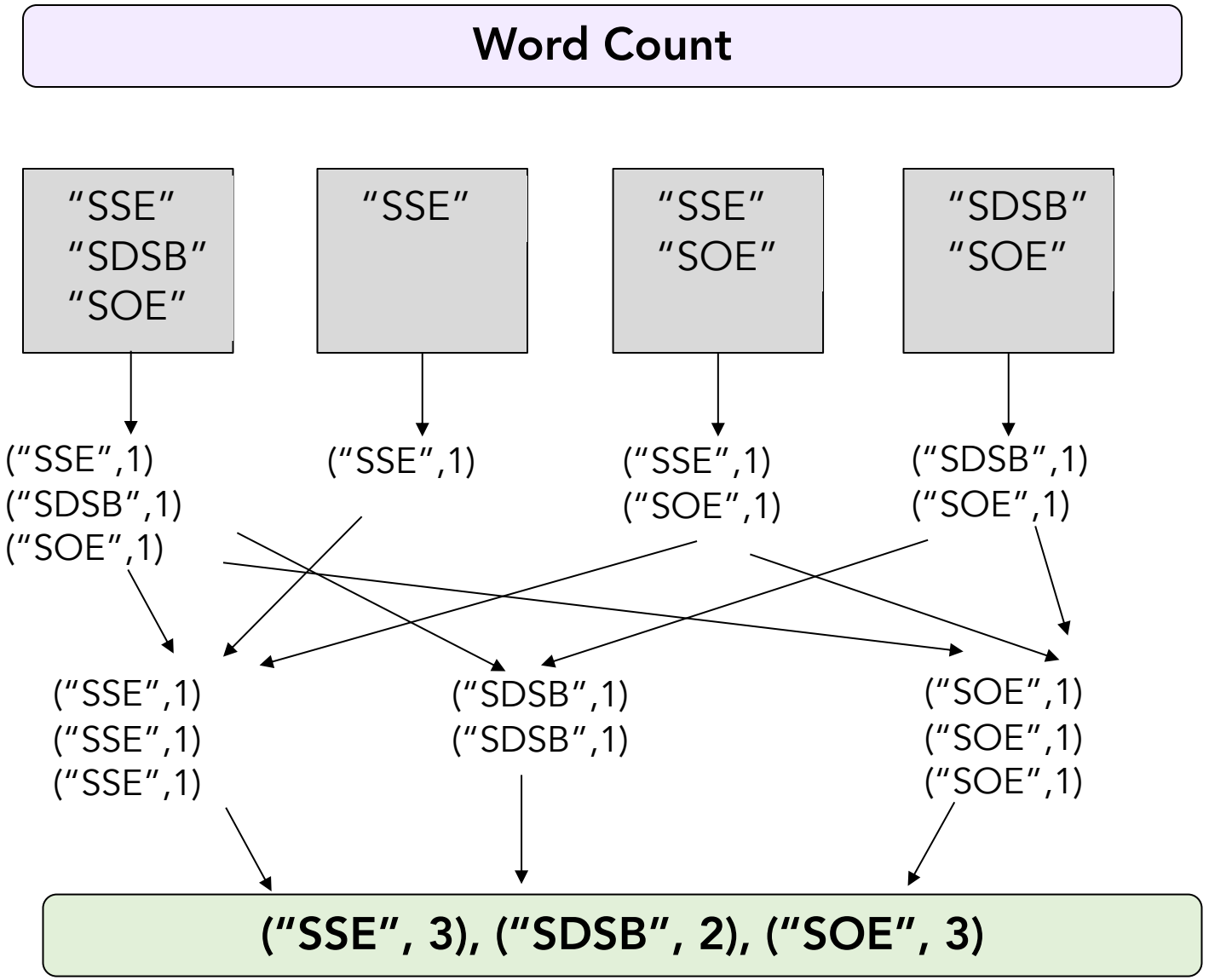
- Abstractions hide unnecessary details from the user
  - They limit what the user can do
  - Hence, the goal is to provide rich abstractions while hiding most details

- Examples
  - Sockets and RPC abstraction
  - Put/get abstraction of hash tables

Typical Operations

- Iterate over a large number of records across servers
- Extract some intermediate results from each
- Shuffle and sort intermediate results
- Collect and aggregate
- Generate final output

Example


Word Count
"SSE"
"SDSB"
"SOE"
"SSE"
"SSE"
"SOE"
"SDSB"
"SOE"
("SSE",1)
("SDSB",1)
("SOE",1)
("SSE",1)
("SSE",1)
("SOE",1)
("SDSB",1)
("SOE",1)
("SSE",1)
("SSE",1)
("SSE",1)
("SDSB",1)
("SDSB",1)
("SOE",1)
("SOE",1)
("SOE",1)
("SSE", 3), ("SDSB", 2), ("SOE", 3)

abstract..abstract..abstract..

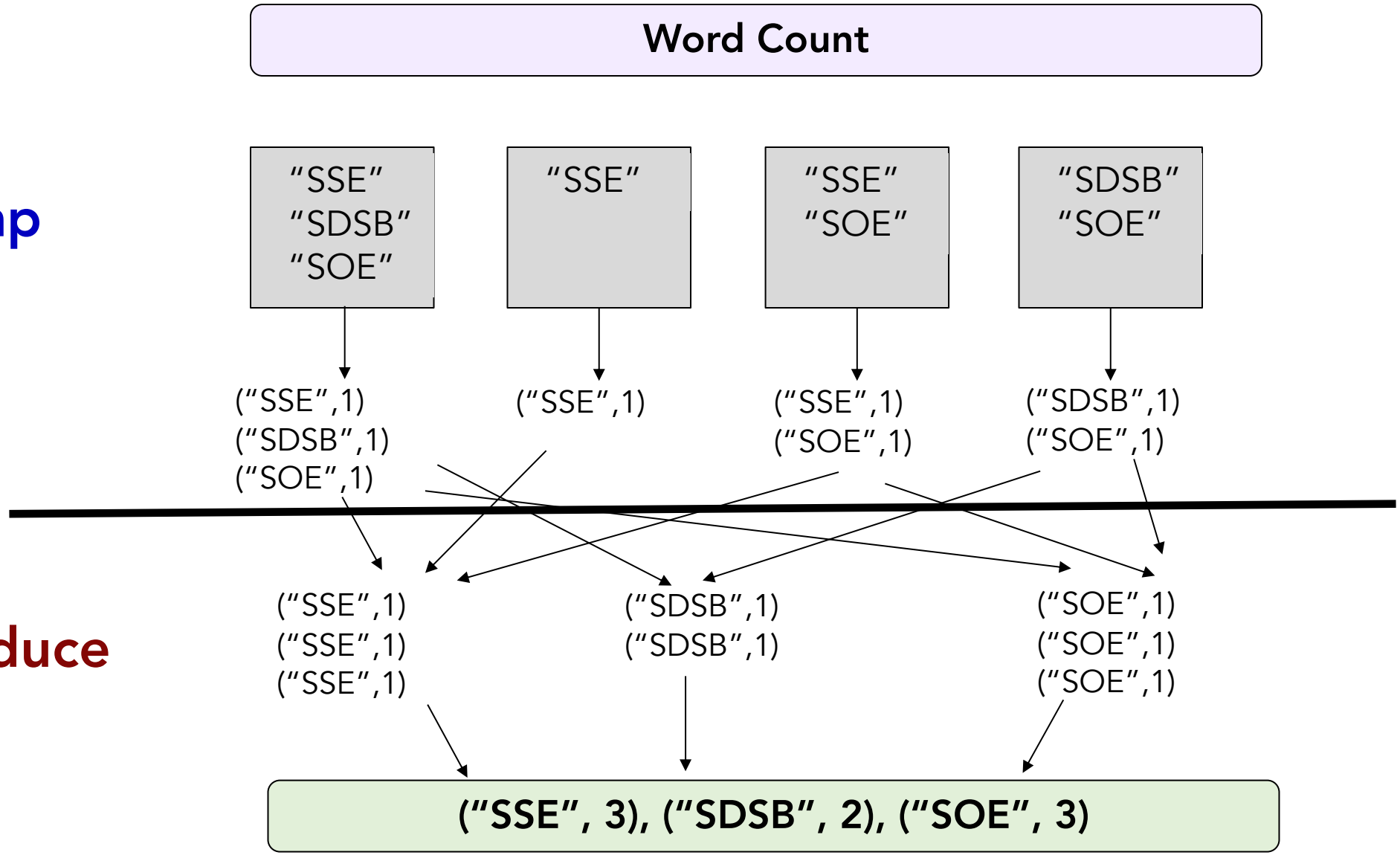
- Iterate over a large number of records across servers
- Extract some intermediate results from each record

Map

Reduce

- Shuffle and sort intermediate results
- Collect and aggregate
- Generate final output

Example


Word Count
Map
"SSE"
"SDSB"
"SOE"
"SSE"
"SSE"
"SOE"
"SDSB"
"SOE"
("SSE",1)
("SDSB",1)
("SOE",1)
("SSE",1)
("SSE",1)
("SOE",1)
("SDSB",1)
("SOE",1)
Reduce
("SSE",1)
("SSE",1)
("SSE",1)
("SDSB",1)
("SDSB",1)
("SOE",1)
("SOE",1)
("SOE",1)
("SSE", 3), ("SDSB", 2), ("SOE", 3)

MapReduce

- Framework for parallel computing
  - Allows one to process huge amounts of data (terabytes and petabytes) on thousands of processors
- Programmers get simple API
  - Write Map and Reduce functions
- The framework takes care of
  - Parallelization
  - Data distribution
  - Fault tolerance

Who Has It?

Who Has it?

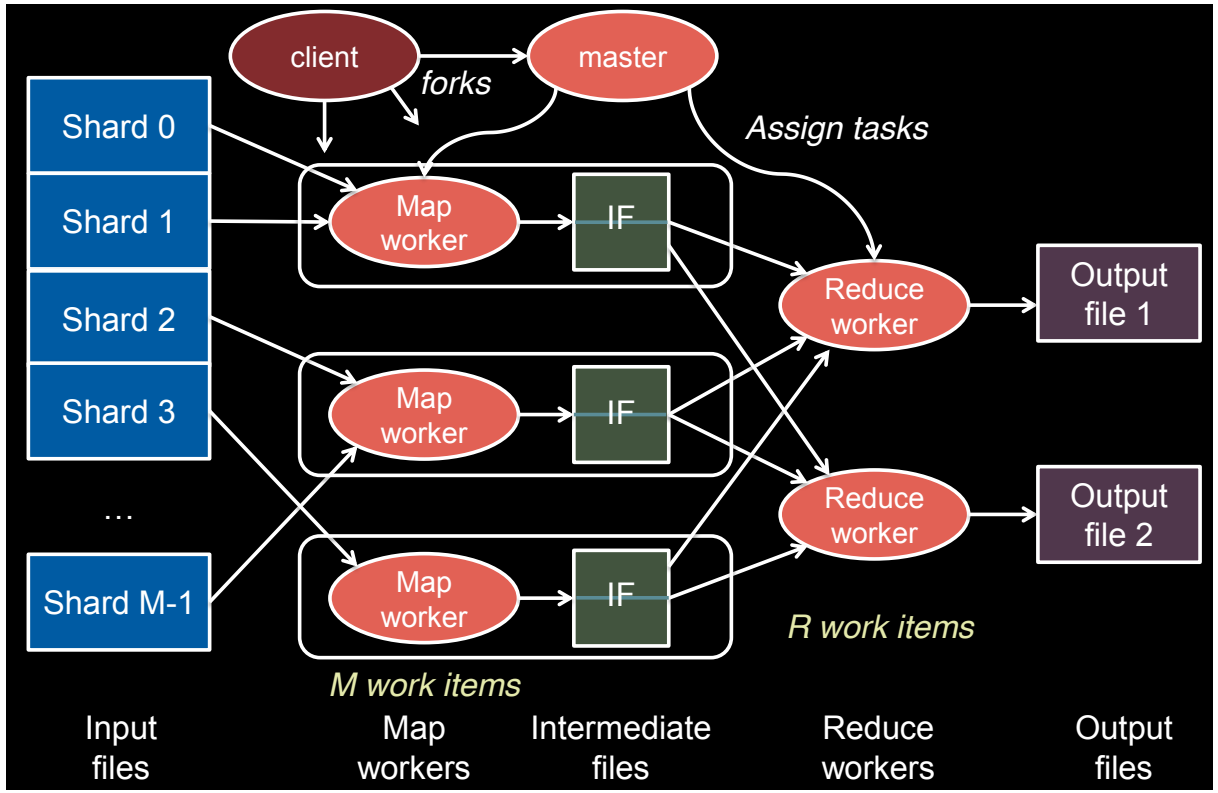
- Google
  - Original proprietary implementation
- Apache Hadoop MapReduce
  - Most common (open-source) implementation
  - Built to specs defined by google
- Amazon Elastic MapReduce
  - Uses Hadoop MapReduce running on Amazon EC2

Zooming in…

MapReduce (MR) Computational Model

- Programmer defines Map and Reduce functions
- Input is <key,value> pairs, divided into shards
  - Perhaps lots of files, <k,v> is filename/content
- MR framework calls Map() on each shard, produces a set of <k2,v2>
- MR framework gathers all Maps' v2's for a given k2, and passes them to a Reduce call
- Final output is set of <k2,v3> pairs from Reduce()

MapReduce: The Complete Picture



Step 1: Split Input Files Into Shards

- Break the Input data into M pieces (typically 16-64MB)

Input files

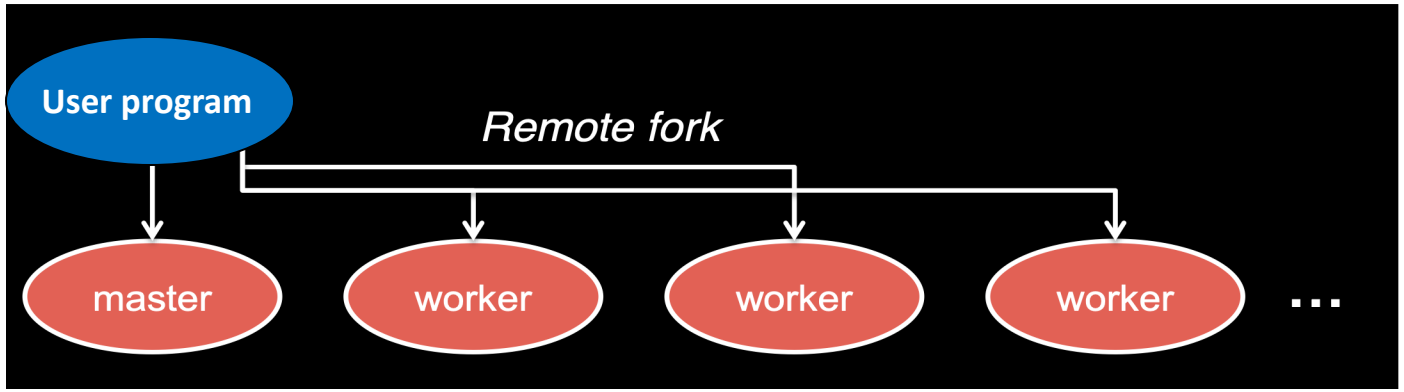
Divided into M shards

Input data is stored in the local disks of the cluster of servers and replicated by a file system, e.g., Google File System

Step 2: Fork Processes

- Start up many copies of the program on a cluster of machines
  - 1 Master: scheduler & coordinator
  - Lots of workers

- Idle workers are assigned tasks by Master:
  - Map tasks (each works on a shard) – there are M map tasks
  - Reduce tasks (each works on intermediate files) – there are R
    - R is defined by the user

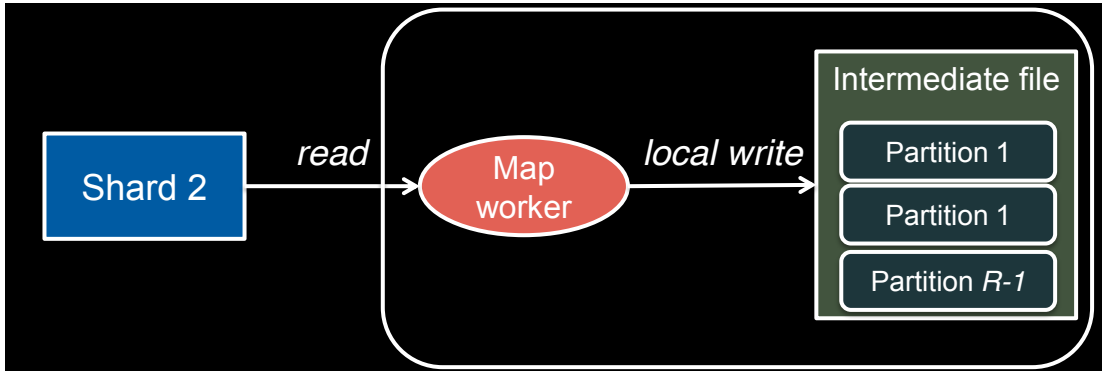


Step 3: Map Tasks

- Reads contents of the input shard assigned to it
- Parses key/value pairs out of the input data
- Passes each pair to a user-defined map function
  - Produces intermediate key/value pairs
  - These are buffered in memory

Step 4: Create Intermediate Files

- Intermediate key/value pairs produced by the user's map function and buffered in memory
  - And are periodically written to the local disk
- Partitioned into R regions by a partitioning function
- When data is written in the partitions, the map worker notifies Master
  - Map worker passes locations of intermediate data to the Master
  - Master forwards these locations to the reduce worker



Step 5: Partitioning

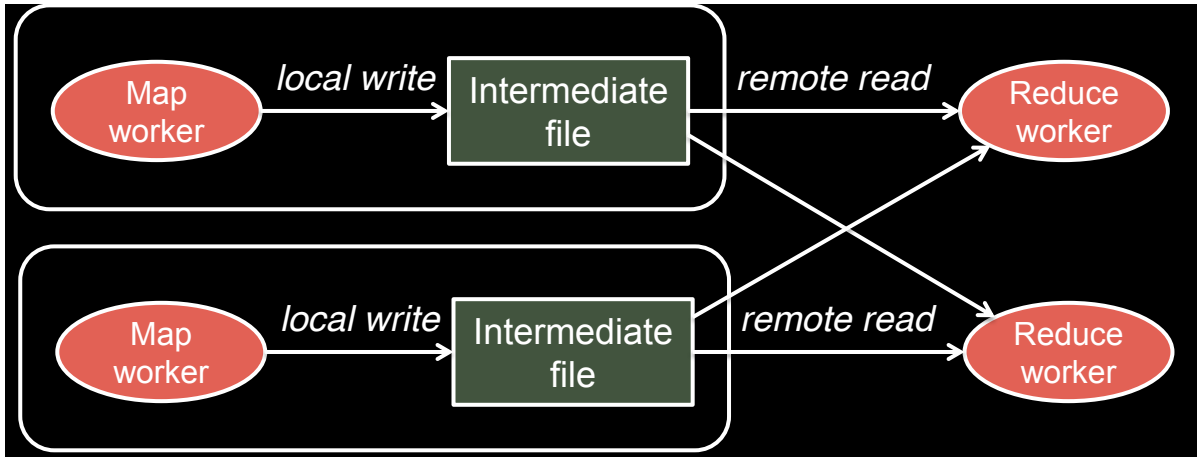
- Map data is processed by reduce workers
  - The user's reduce function is called once per unique key generated by Map

- This means we will need to sort all the (key, value) data by keys and decide which reduce worker processes which keys

- Partition function: Which reduce worker is responsible for which key
  - Default function: hash(key) mod R
  - Map worker partitions the data by keys

Step 6: Reduce Task -- *Sorting*

- Reduce worker gets notified by the master about the location of intermediate files for its partition
- Reads the data from the local disks of the map workers through RPC
- When the reduce worker reads intermediate data for its partitions
  - It sorts the data by the intermediate keys
  - All occurrences of the same key are grouped together

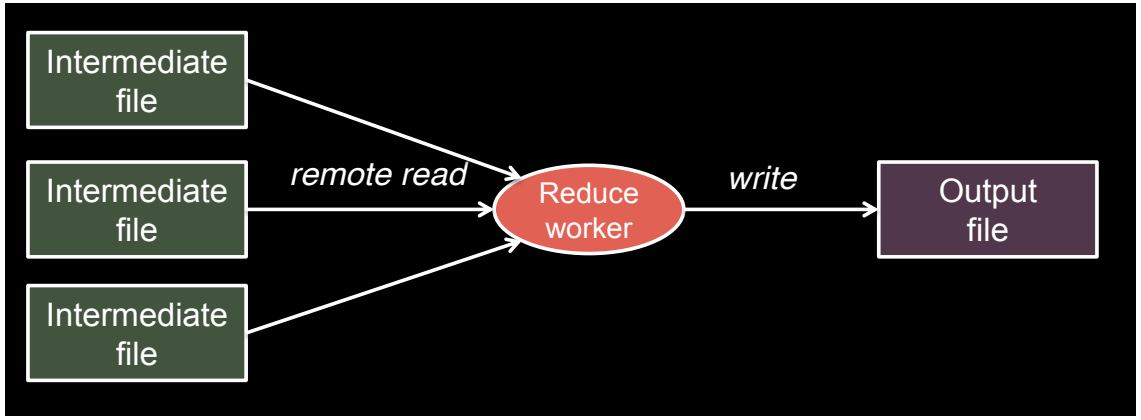


Step 7: Reduce Task -- *Reduce*

- The sort phase grouped data with a unique intermediate key

- The user's Reduce function is given the key and the set of intermediate values for that key
  - < key, (value1, value2, value3, value4, ...) >

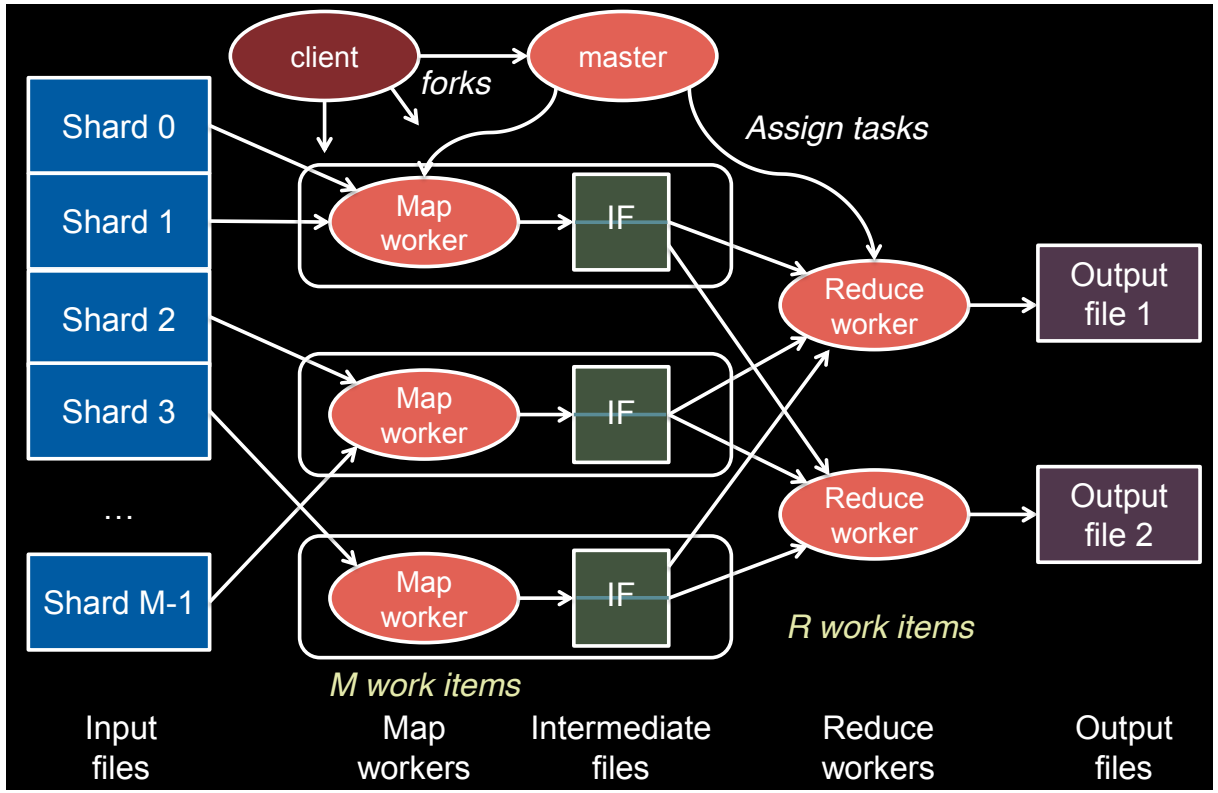
- The output of the Reduce function is appended to an output file



Step 8: Return To User

- When all map and reduce tasks have been completed, the Master wakes up the user program
- MapReduce call in the user program returns and the program can resume execution
- Output of MapReduce is available in R output files

MapReduce: The Complete Picture



What does MR framework do for word count?

```
[master, input files, map workers, map output, reduce workers, output files]
```

- Input files:
  - f1: a b
  - f2: b c
- Send "f1" to map worker 1
- Map("f1", "a b") -> <a 1> <b 1>
- Send "f2" to map worker 2
- Map("f2", "b c") -> <b 1> <c 1>
- Framework waits for Map jobs to finish

- Framework tells each reduce worker what key to reduce
  - worker 1: a
  - worker 2: b
  - worker 2: c
- Each reduce worker pulls needed Map output from Map workers
  - worker 1 pulls "a" Map output from every worker
- Each reduce worker calls Reduce once for each of its keys
  - worker 1: Reduce("a", [1]) -> 1
  - worker 2: Reduce("b", [1, 1]) -> 2
    Reduce("c", [1]) -> 1

How Google Used MapReduce?

- *"We often performed MapReduce computations with M= 200,000 and R = 5,000, suing 2000 worker machines"*

Why might MR have good performance?

- Map and Reduce functions run in parallel on different workers
- Ideally, with N workers; divide run-time by N
- But not quite that good:
  - move map output to reduce workers
  - can have slow nodes
  - read/write from network file system
  - failures

Failures

- Google uses MR with 1000s of workers
  - Suppose each worker only crashes once per year → That's ~3 per day!
  - Hence a big MR job is very likely to suffer worker failures

- What other things can go wrong?
  - Worker may be slow
  - Master may crash
  - Parts of the network may fail, lose packets

Dealing with Failures

- Master pings each worker periodically
  - If no response is received within a certain time, the worker is marked as failed

- Map or reduce tasks given to this worker are reset back to the initial state and rescheduled for other workers

Dealing with with Slow nodes

- Stragglers: slow nodes

- Solution: start a backup execution when “close to finishing”
  - Does not increase computational resources substantially

Summary

- Framework for parallel computing
- Programmers get simple API
  - Write Map and Reduce functions
- Framework deals with
  - Parallelization
  - Data distribution
  - Fault tolerance

Next Lecture

- Dealing with limitations of MapReduce
- Spark: A Fault-Tolerant Abstraction for In-Memory Cluster Computing