CompSci 516
Data Intensive Computing Systems

Lecture 3
Map-Reduce
and
Spark

Guest Lecturer: Junghoon Kang

Reading Material

- Recommended (optional) readings:
  - Chapter 2 (Sections 1,2,3) of Mining of Massive Datasets, by Rajaraman and Ullman: http://i.stanford.edu/~ullman/mmds.html
  - MapReduce: Simplified Data Processing on Large Cluster - Jeffrey Dean, et al. – 2004
  - The Google File System - Sanjay Ghemawat, et al. – 2003
  - Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing - Matei Zaharia, et al. - 2012

Announcement

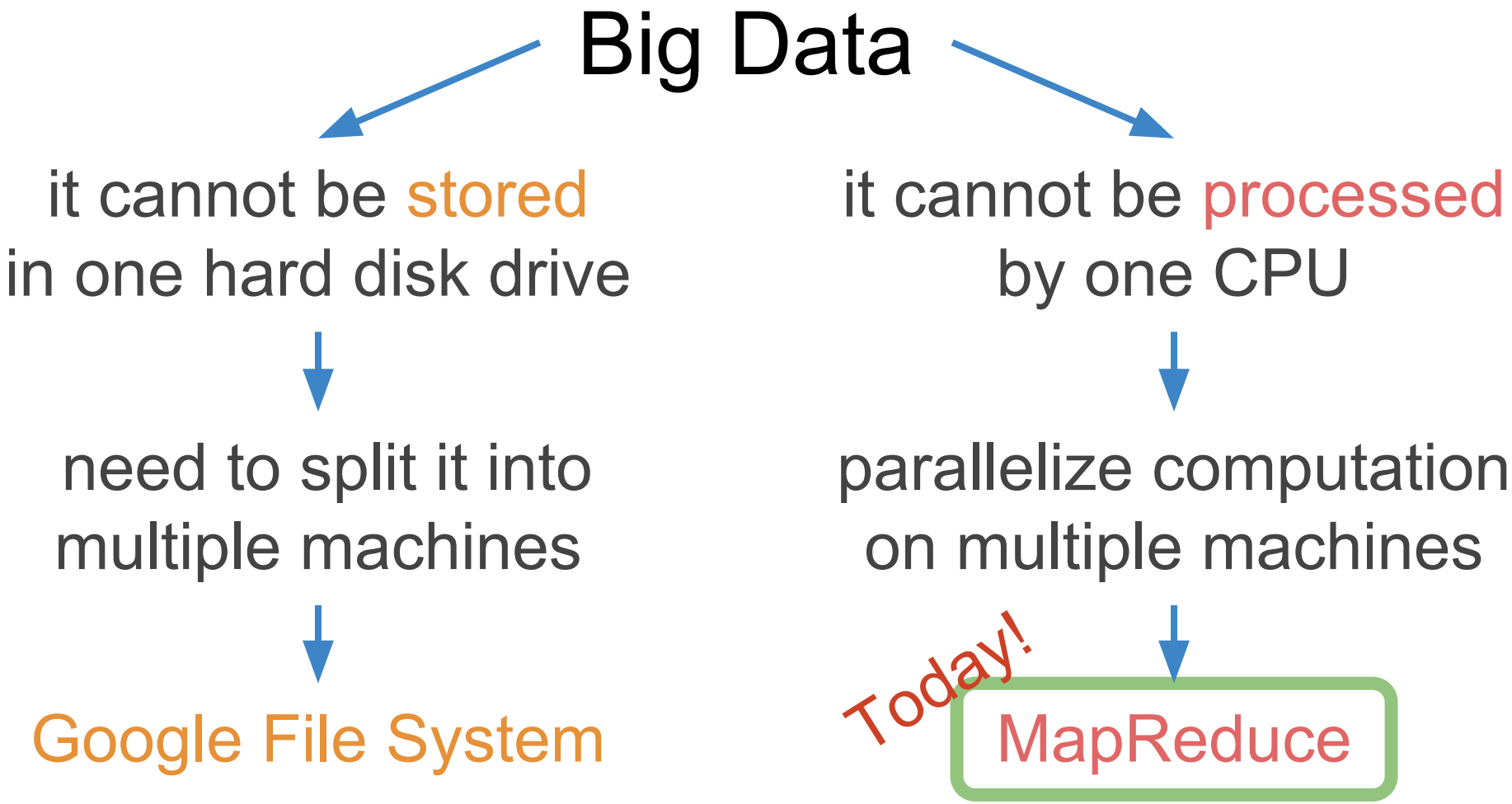
- Jung - I have switched my office hours
  from Thursdays 1pm - 2pm
  to Thursdays 3pm - 4pm
  at the same location, N303B.

- In HW2, you will be writing Spark applications and run them on AWS EC2 instances.

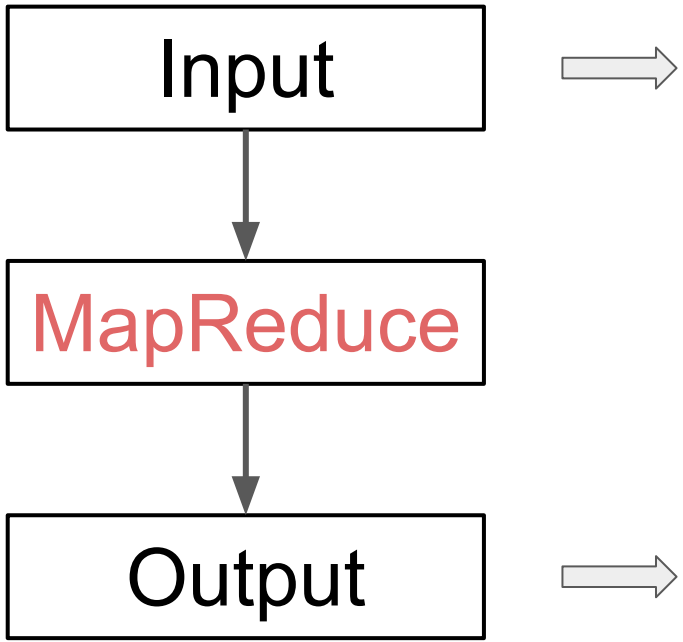
Google MapReduce

CompSci 516
Junghoon Kang


BIG
DATA
INFORMATION
STORAGE
LARGE
MANAGEMENT
ANALYSIS
SETS
MILLION
SEARCH
PETABYTES
RESEARCH
INTERNET
TECHNOLOGIES
GOVERNMENT
VOLUME
SOFTWARE
SAN
NAS
PROCESS
SIZE
EXABYTES
LEARNING
WORLD
EVERY
BASED
DEVELOPMENT
TECHNOLOGY
SCIENCE
FUTURE
BILLION
NEW
PARALLEL
INITIATIVE
ECONOMIC
TERABYTES
APPROACHES
PROCESSES
FACTOR
ONLINE
DAY
MANAGE
WORLDWIDE
SYSTEMS
RESULTS
SOCIAL
YEARS
DRIVE
REPLICATION
HANDLE
SOURCES
CHALLENGES
LIMITS
COMPLEX
ANNOUNCED
PAST
HAND
CAPACITY
ONE
OUT
INCLUDE
NETWORKS
HYPOTHESIS
END
MOST
NEARLY
MAPREDUCE
CRITIQUES
APPROACH
MANY
MAKING
INTEGRATION
MASSIVE
USING
DEFINITION
FRAMEWORK
BETWEEN
STATISTICS
BUSINESS
EFFECTIVE
VISUALIZATION
DATABASES
PROCESSING
ANALYTICS
SIMULATIONS
DETERMINE
WELL
SET
DEPARTMENT
QUERIES
HIGH
USERS
AMOUNT
REAL
STRUCTURE
WORK
RATE
APPLICATIONS
INFRASTRUCTURE
HIGHER
ALGORITHMS
COMPANIES
TERMS
PRIVATE
YEAR
SUCCESSFUL
NEEDED
USES
LESS
HUGE
CAPABILITIES
COMPARED
SHARED
VARIOUS
TIMES
ARCHITECTURE
PARADIGM
FUNDING
INSIGHT
TOOLS
CENTER
RECORDS
COST
PER
DISTRIBUTED
RELEVANT
USED
PROJECT
ANNUAL
NOW
VARIETY
MARKET
SENSOR
TIME
WOULD
RELATED
CURRENTLY
USE
STATE
CRITIQUE
PEOPLE
TRAFFIC
PRIVACY
INTERNATIONAL
IMPLEMENTATION
BECOMES
SECOND
NATIONAL
FLOW
INTELLIGENCE


Big Data
it cannot be stored
in one hard disk drive
need to split it into
multiple machines
Google File System
it cannot be processed
by one CPU
parallelize computation
on multiple machines
Today!
MapReduce

Where does Google use MapReduce?



- crawled documents
- web request logs

- inverted indices
- graph structure of web documents
- summaries of the number of pages crawled per host
- the set of most frequent queries in a day

What is MapReduce?

It is a programming model

that processes large data by:

apply a function to each logical record in the input (map)

categorize and combine the intermediate results
into summary values (reduce)

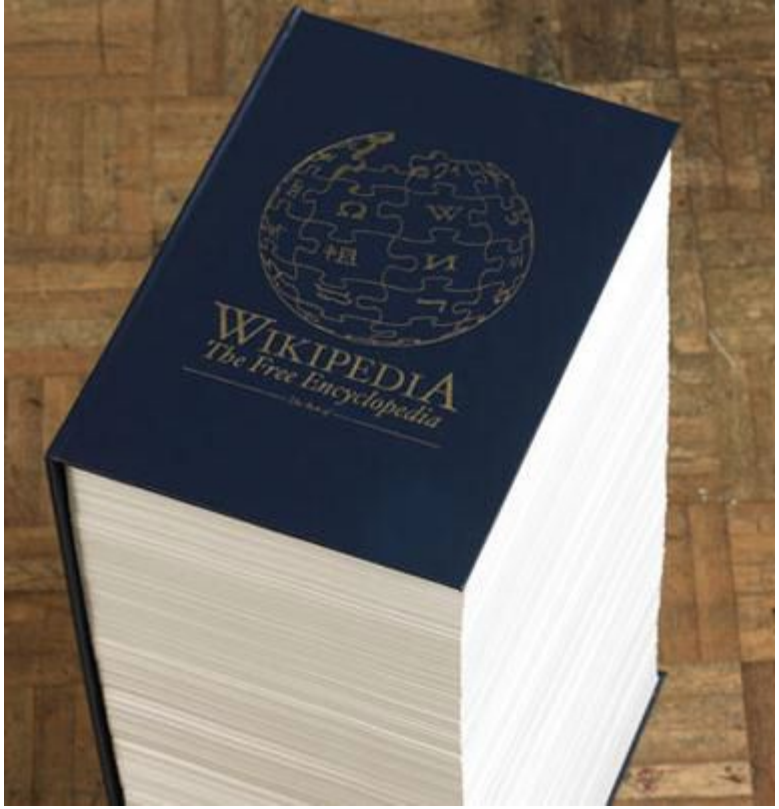
Understanding MapReduce

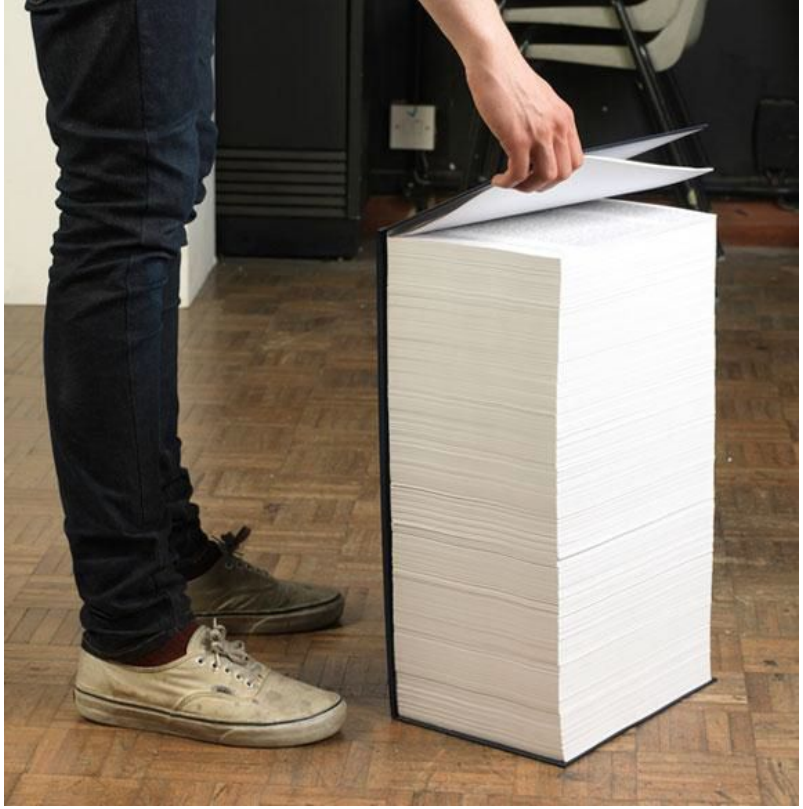
(example by Yongho Ha)

I am a class president

An English teacher asks you:

“Could you count the number of occurrences of each word in this book?”



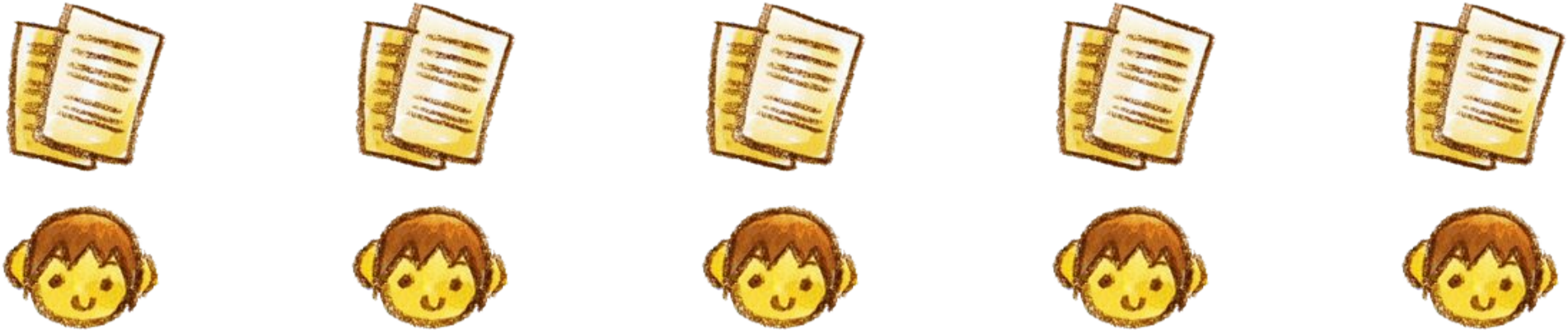


Um... Ok...

borrowed slide

Let’s divide the workload among classmates.

map



cloud 1
data 1

parallel 1
data 1
computer 1

map 1
cloud 1
parallel 1

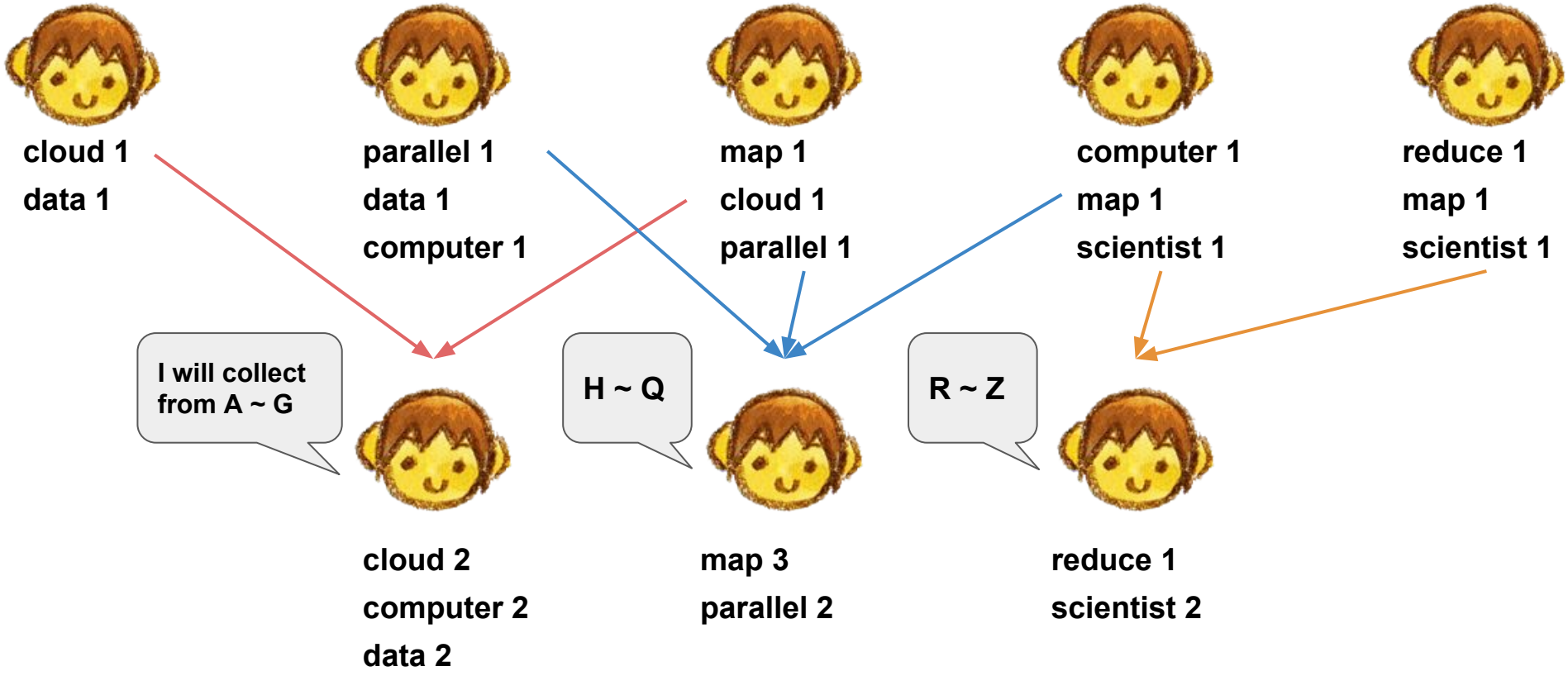
computer 1
map 1
scientist 1

reduce 1
map 1
scientist 1

borrowed slide

And let few combine the intermediate results.

reduce



borrowed slide

Why did MapReduce become so popular?

borrowed slide

Is it because Google uses it?

borrowed slide

Distributed Computation Before MapReduce

Things to consider:

- how to divide the workload among multiple machines?
- how to distribute data and program to other machines?
- how to schedule tasks?
- what happens if a task fails while running?
- … and … and ...

borrowed slide

Distributed Computation After MapReduce

Things to consider:

- how to write Map function?
- how to write Reduce function?

MapReduce has made distributed computation an easy thing to do!

Developers needed
before MapReduce

Developers needed
after MapReduce

borrowed slide

Given the brief intro to MapReduce,

let’s begin our journey to real implementation details in MapReduce !

Key Players in MapReduce

One Master

- coordinates many workers.
- assigns a task* to each worker.

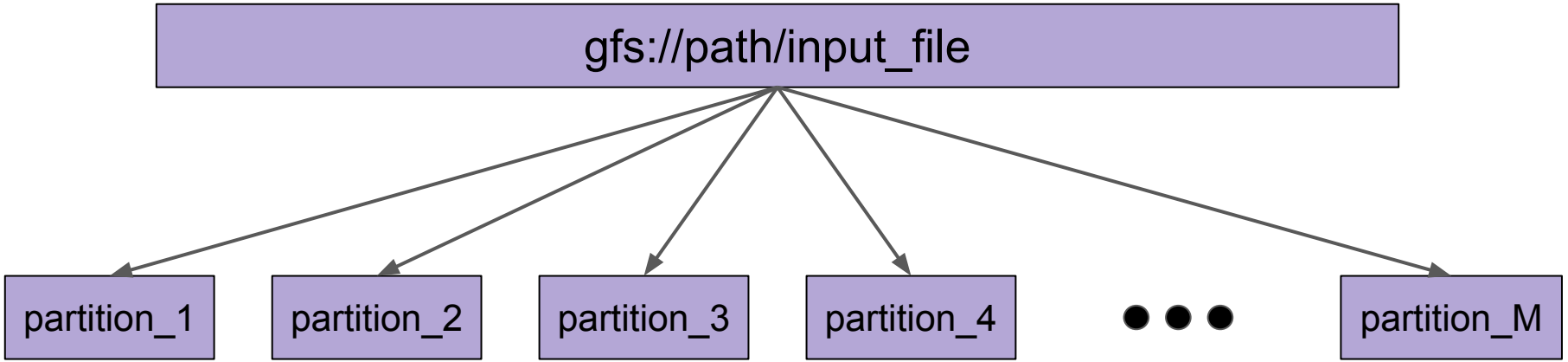
  (* task = partition of data + computation)

Multiple Workers

- Follow whatever the Master asks to do.

Execution Overview

1. The MapReduce library in the user program first splits the input file into **M** pieces.

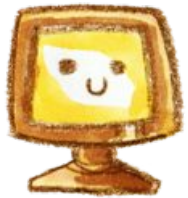


2. The MapReduce library in the user program then starts up many copies of the program on a cluster of machines: one master and multiple workers .

master

worker 1

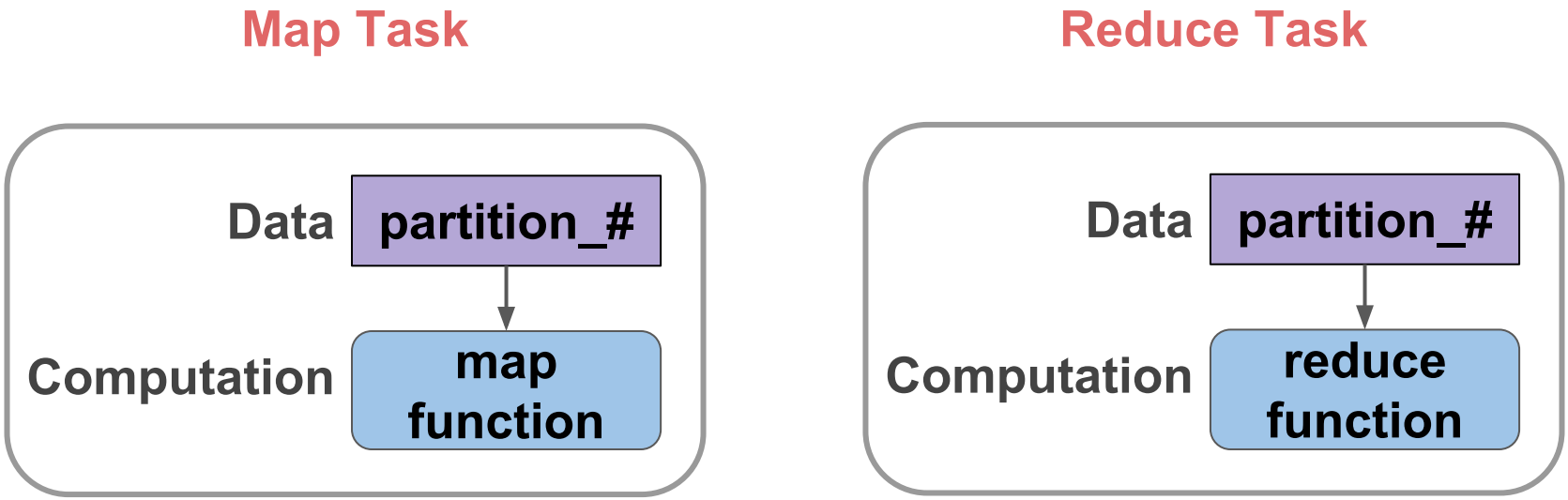
worker 2



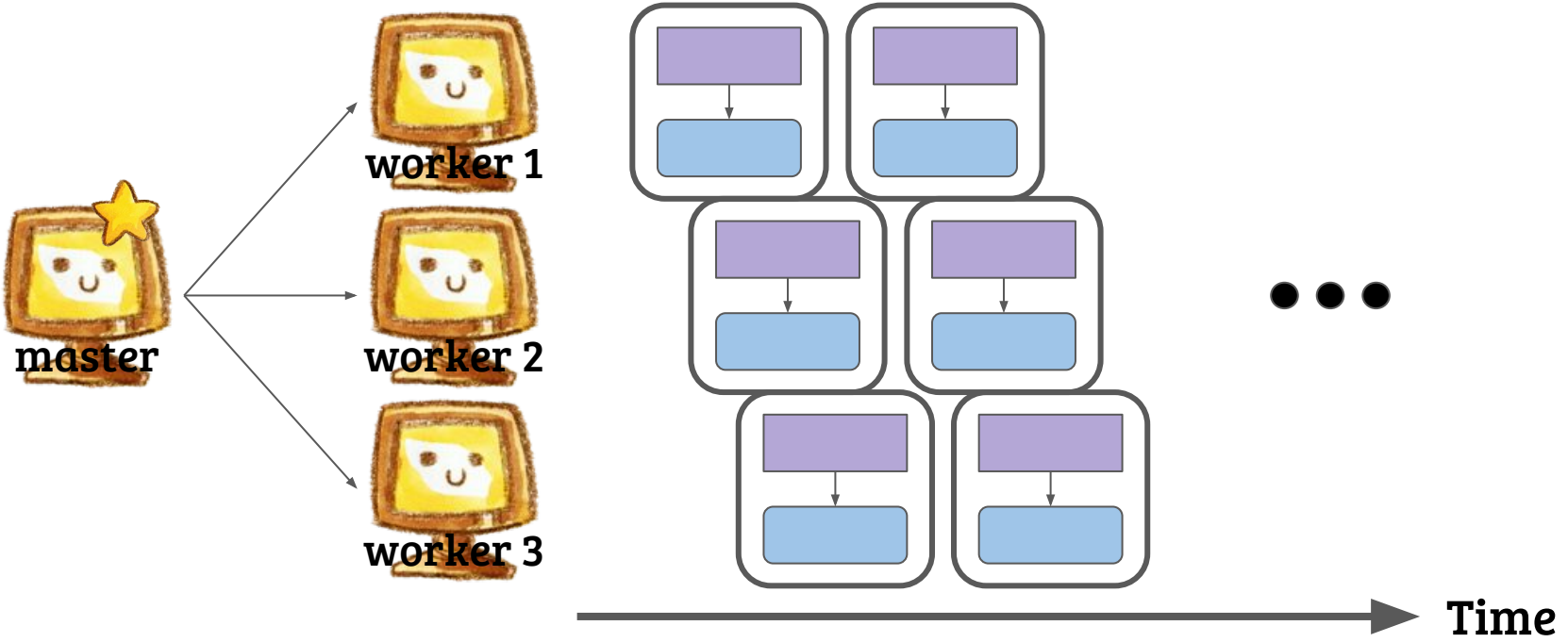
worker 3

There are **M** map tasks and **R** reduce tasks to assign.

(The figures below depicts task = data + computation)



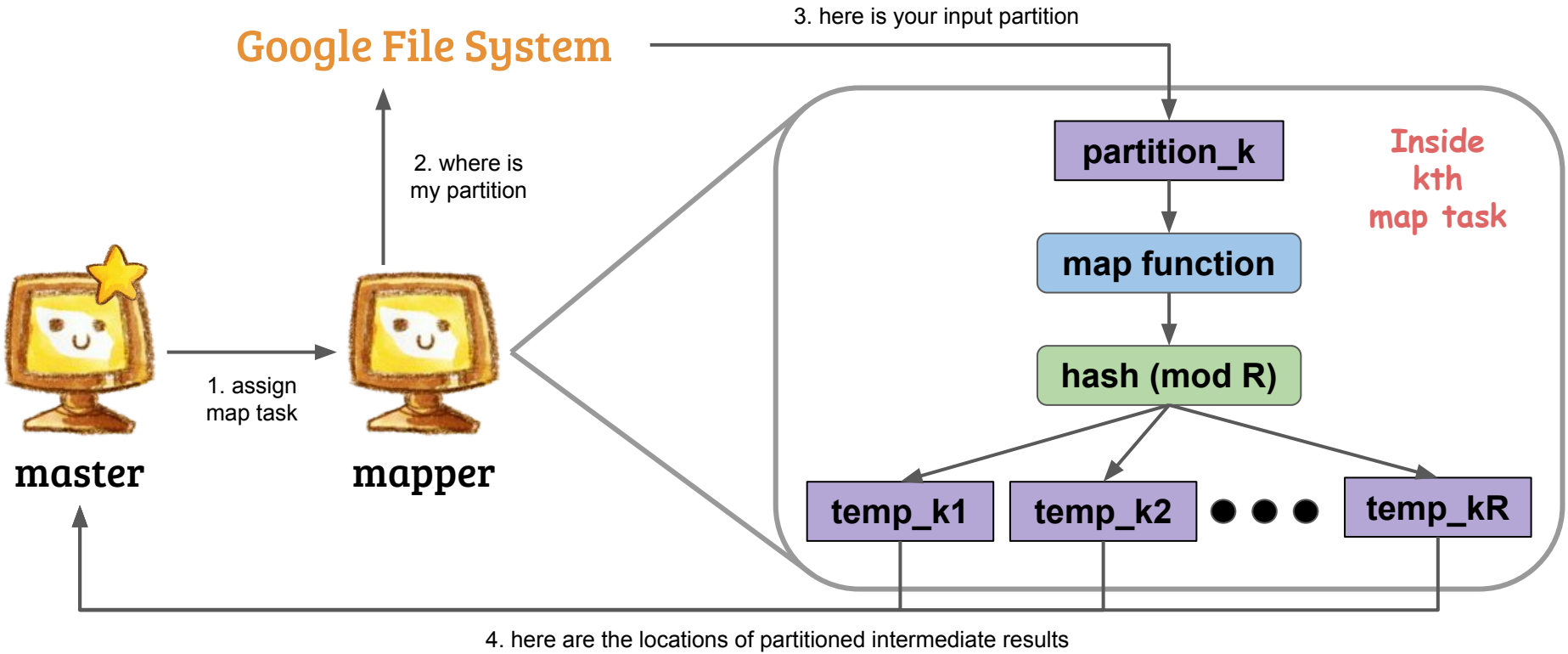
3. The master picks idle workers and assigns each one a map task.



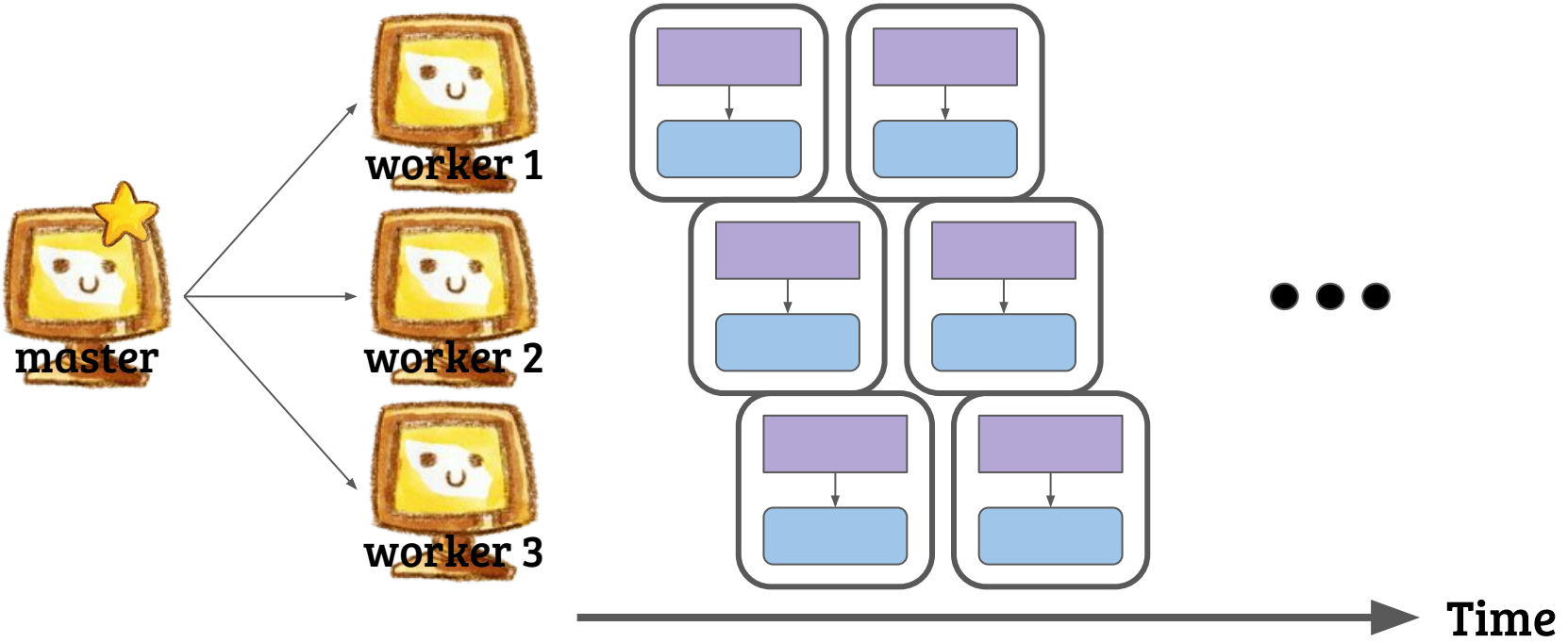
4. Map Phase (each mapper node)

1) Read in a corresponding input partition.
2) Apply the user-defined map function to each key/value pair in the partition.
3) Partition the result produced by the map function into **R** regions using the partitioning function.
4) Write the result into its local disk (not GFS).
5) Notify the master with the locations of each partitioned intermediate result.

Map Phase


Google File System
3. here is your input partition
2. where is my partition
partition_k
Inside kth map task
map function
hash (mod R)
1. assign map task
master
mapper
temp_k1
temp_k2
temp_kR
4. here are the locations of partitioned intermediate results

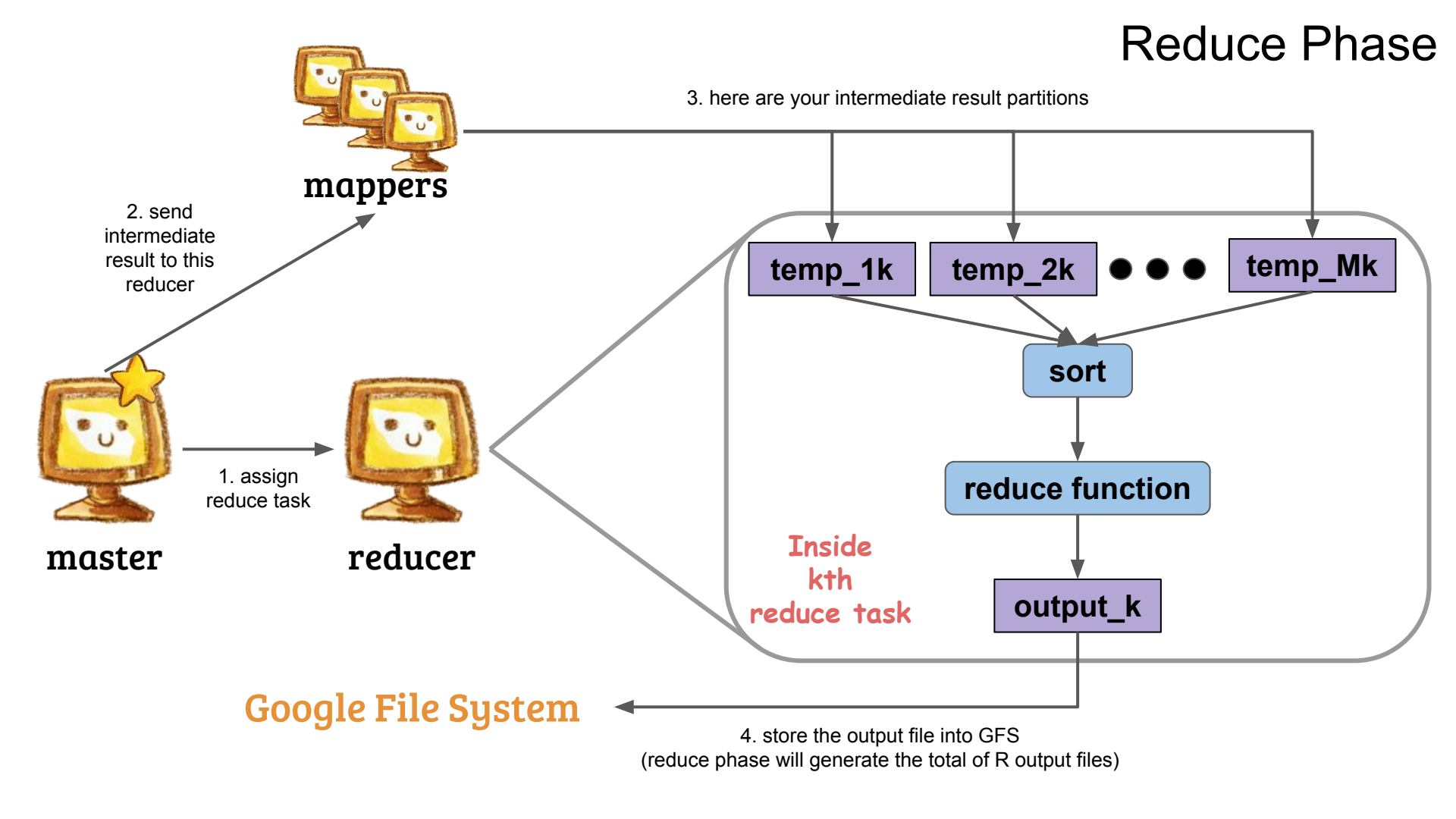
5. After all the map tasks are done, the master picks idle workers and assigns each one a reduce task.



6. Reduce Phase (each reducer node)

1) Read in all the corresponding intermediate result partitions from mapper nodes.
2) Sort the intermediate results by the intermediate keys.
3) Apply the user-defined reduce function on each intermediate key and the corresponding set of intermediate values.
4) Create one output file.

Reduce Phase



Fault Tolerance

Although the probability of a machine failure is low, the probability of a machine failing among thousands of machines is common.

How does MapReduce handle machine failures?

Worker Failure

- The master sends heartbeat to each worker node.
- If a worker node fails, the master reschedules the tasks handled by the worker.

Master Failure

- The whole MapReduce job gets restarted through a different master.

Locality

- The input data is managed by GFS.
- Choose the cluster of MapReduce machines such that those machines contain the input data on their local disk.
- We can conserve network bandwidth.

Task Granularity

- It is preferable to have the number of tasks to be multiples of worker nodes.
- Smaller the partition size, faster failover and better granularity in load balance.

  But it incurs more overhead. Need a balance.

Backup Tasks

- In order to cope with a straggler, the master schedules backup executions of the remaining *in-progress* tasks.

MapReduce Pros and Cons

- MapReduce is good for off-line batch jobs on large data sets.
- MapReduce is not good for iterative jobs due to high I/O overhead as each iteration needs to read/write data from/to GFS.
- MapReduce is bad for jobs on small datasets and jobs that require low-latency response.

Apache Hadoop

Apache Hadoop is an open-source version of GFS and Google MapReduce.

	Google	Apache Hadoop
File System	GFS	HDFS
Data Processing Engine	Google MapReduce	Hadoop MapReduce

References

- MapReduce: Simplified Data Processing on Large Cluster - Jeffrey Dean, et al. - 2004
- The Google File System - Sanjay Ghemawat, et al. - 2003
- http://www.slideshare.net/yongho/2011-h3

Apache Spark

CompSci 516
Junghoon Kang

About Spark

- Spark is a distributed large-scale data processing engine that exploits in-memory computation and other optimizations.

- One of the most popular data processing engine in the industry these days; many large companies like Netflix, Yahoo, and eBay use Spark at massive scale.

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.*, between two MapReduce jobs) is to write it to an external stable storage system, *e.g.*, a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.*, looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.*, to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.*, cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.*, map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[1]Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

More about Spark

- It started as a research project at UC Berkeley.
- Published the **Resilient Distributed Datasets (RDD)** paper in NSDI 2012.
- **Best Paper** award that year.

Motivation

Hadoop MapReduce indeed made analyzing large datasets easy.

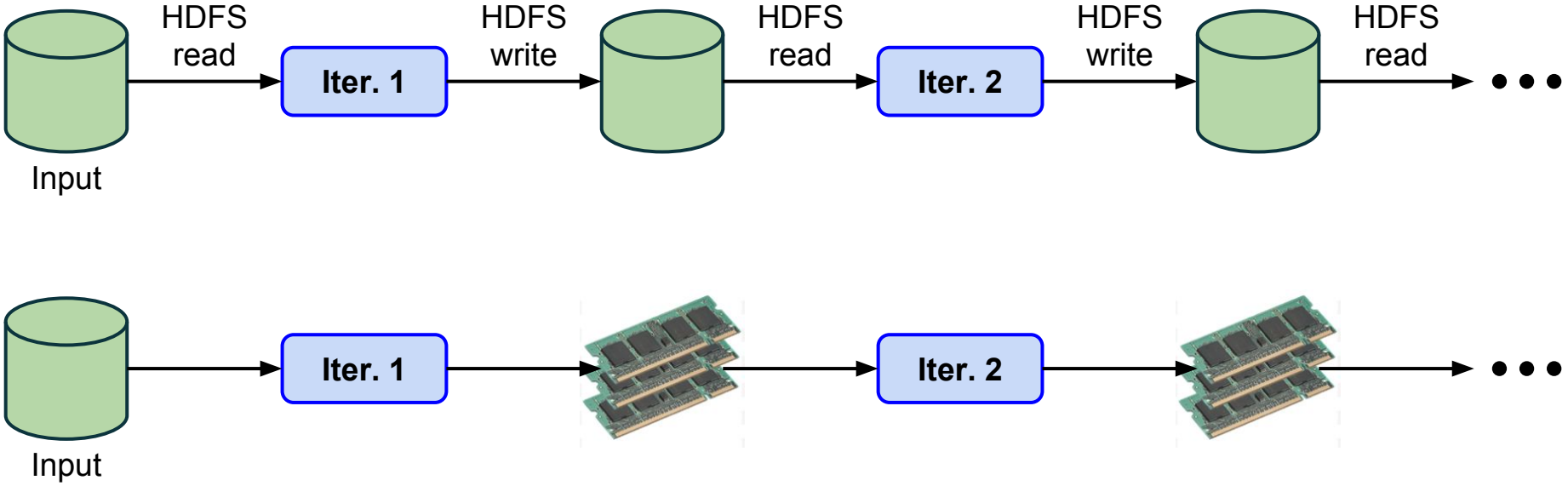
However, MapReduce was still not good for:

- iterative jobs, such as machine learning and graph computation
- interactive and ad-hoc queries

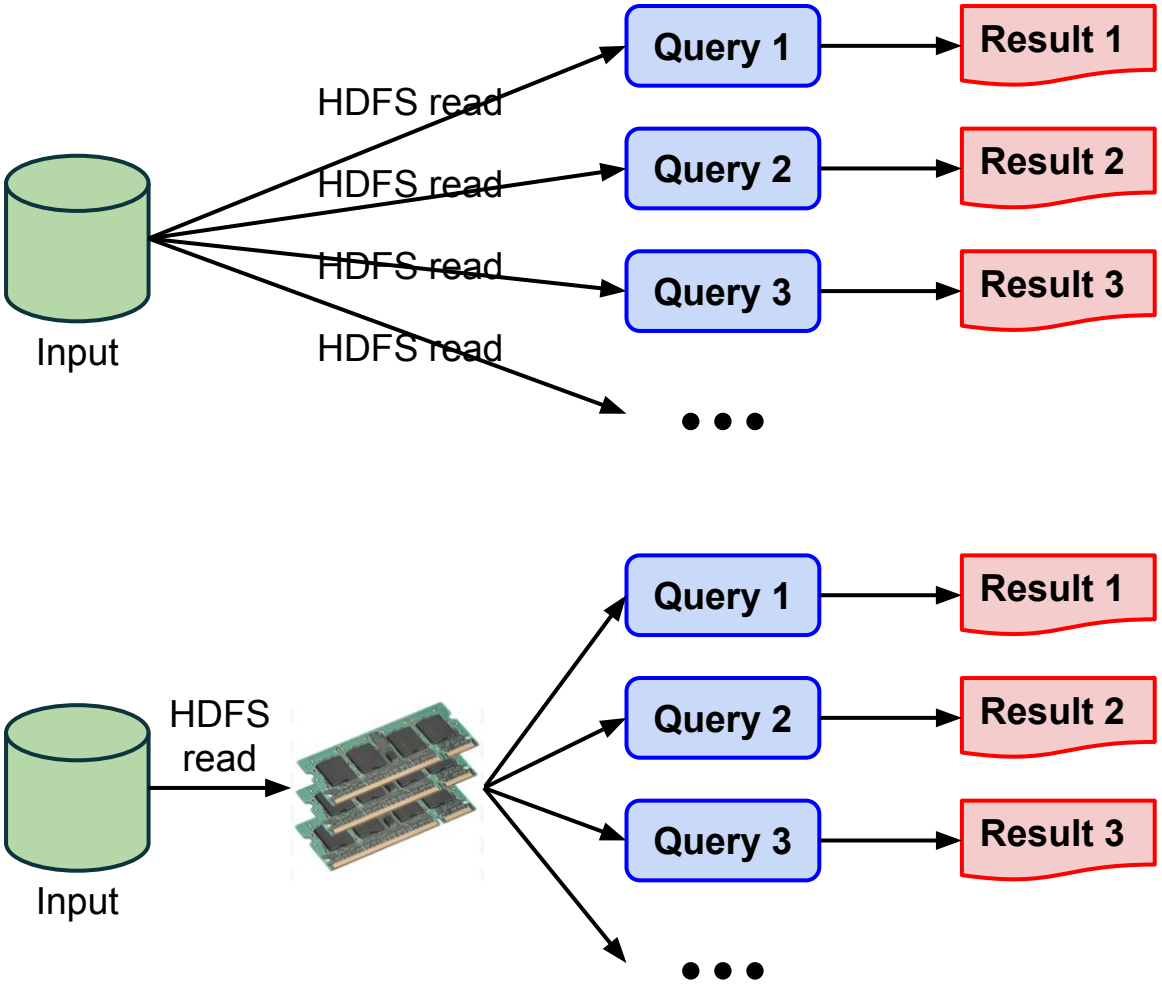
Can we do better?

The reason why MapReduce is not good for iterative jobs is because of the high I/O overhead as each iteration needs to read/write data from/to HDFS.

So, what if we use RAM between each iteration?



Instead of storing intermediate outputs into HDFS, using RAM would be faster

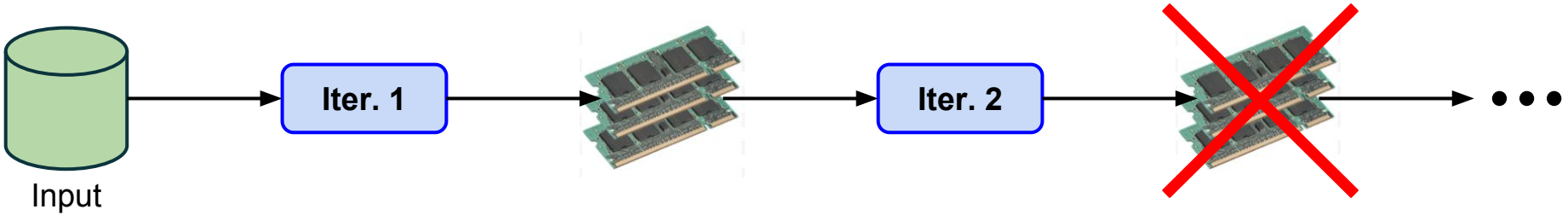


Instead of reading input from HDFS every time you run query,
bring the input into RAM first then run multiple queries.

Challenge

But RAM is a volatile storage…

What happens if a machine faults?



Although the probability of a machine failure is low, the probability of a machine failing among thousands of machines is common.

In other words, how can we create an efficient, fault-tolerant, and distributed RAM storage?

Some Approaches

Some data processing frameworks, such as RAMCloud or Piccolo, also used RAM to improve the performance.

And they supported fine-grained update of data in RAM.

But it is hard to achieve fault tolerance with fine-grained update while having a good performance.

Spark’s Approach

What if we use RAM as read-only?

This idea is RDD, Resilient Distributed Datasets!

Which is the title of the Spark paper and the core idea behind Spark!

Resilient Distributed Datasets

What are the properties of RDD?

- read-only, partitioned collections of records

- you can only create RDD from input files in a storage or RDD

RDD (cont.)

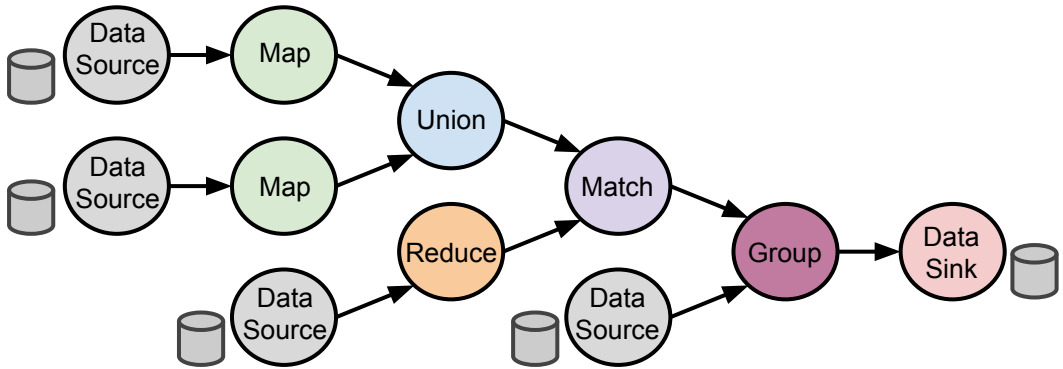
What's good about RDD again?

- RDD is read-only (immutable).
  Thus, it hasn't been changed since it got created.
- That means
  if we just record how the RDD got created
  from its parent RDD (lineage),
  it becomes fault-tolerant!

RDD (cont.)

But how do you code in Spark using RDD?

- Coding in Spark is
  creating a lineage of RDDs
  in a directed acyclic graph (DAG) form.



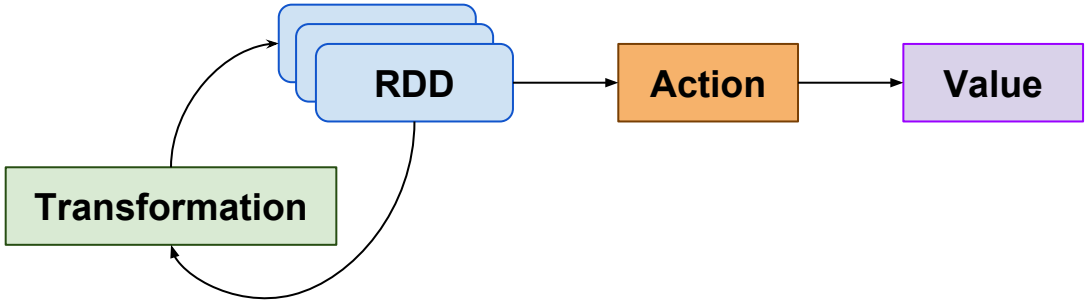
RDD Operators

Transformations & Actions

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Lazy Execution



- Transformation functions simply creates a lineage of RDDs.
- An action function that gets called in the end triggers the computation of the whole lineage of transformation functions and outputs the final value.

Two Types of Dependencies

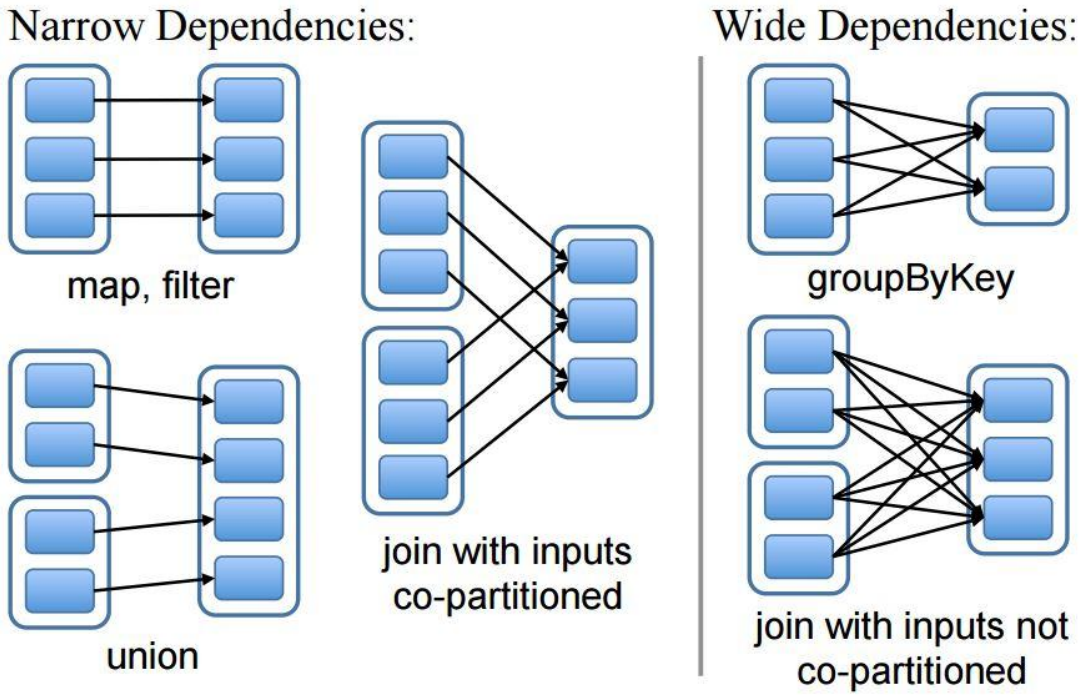
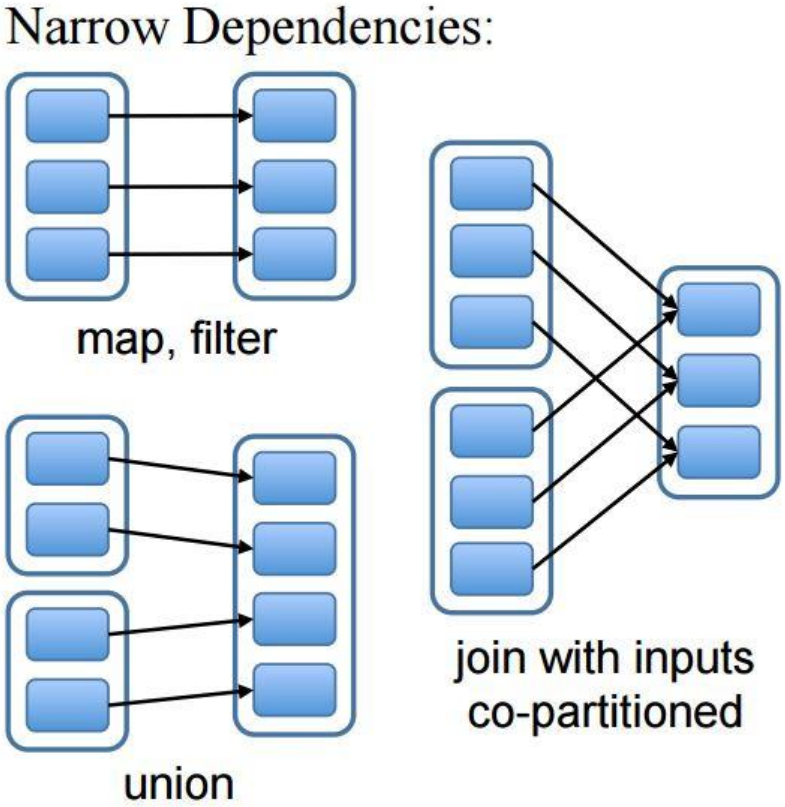


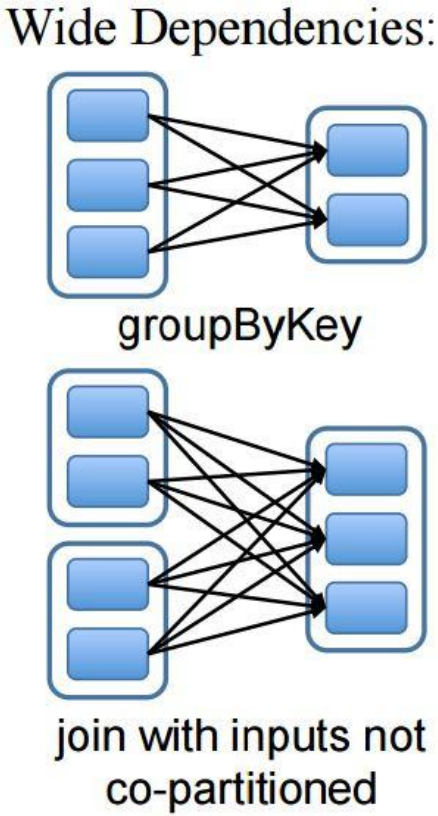
Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

Narrow Dependency



- The task can be done in one node.
- No need to send data over network to complete the task.
- Fast.

Wide Dependency



- The task needs shuffle.
- Need to pull data from other nodes via network.
- Slow.
- Use wide dependencies wisely.

Job Scheduling

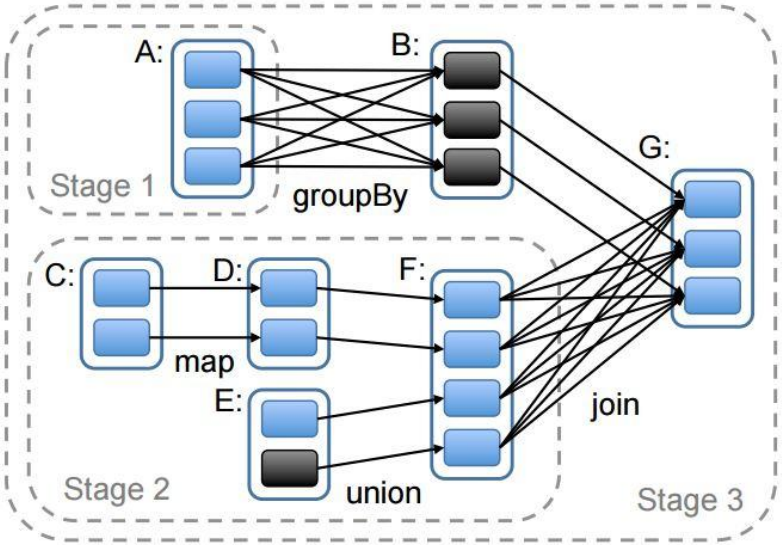


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

- One job contains one action function and possibly many transformation functions.
- A job is represented by the DAG of RDDs.
- Compute the job following the DAG.
- New stage gets created if a RDD requires shuffle from an input RDD.

Task Distribution

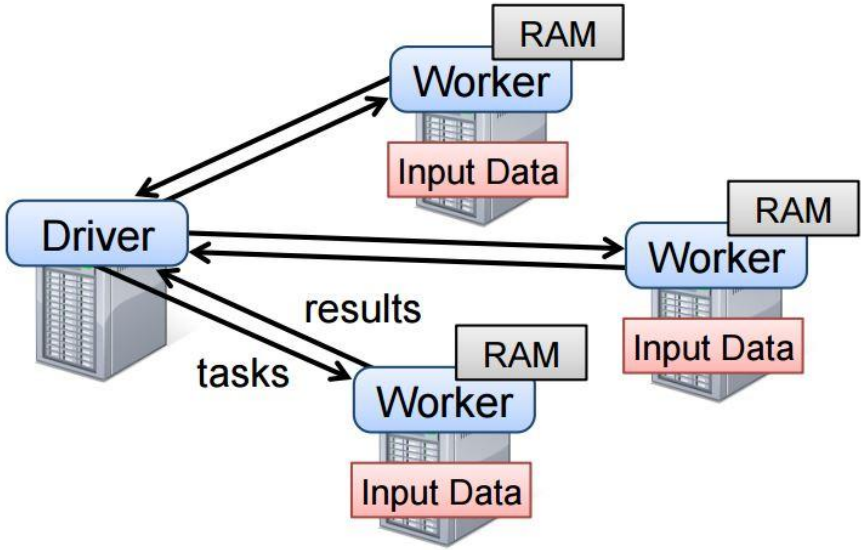


Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

- Similar to MR
- One master, multiple workers
- One RDD is divided into multiple partitions

How fast is Spark?

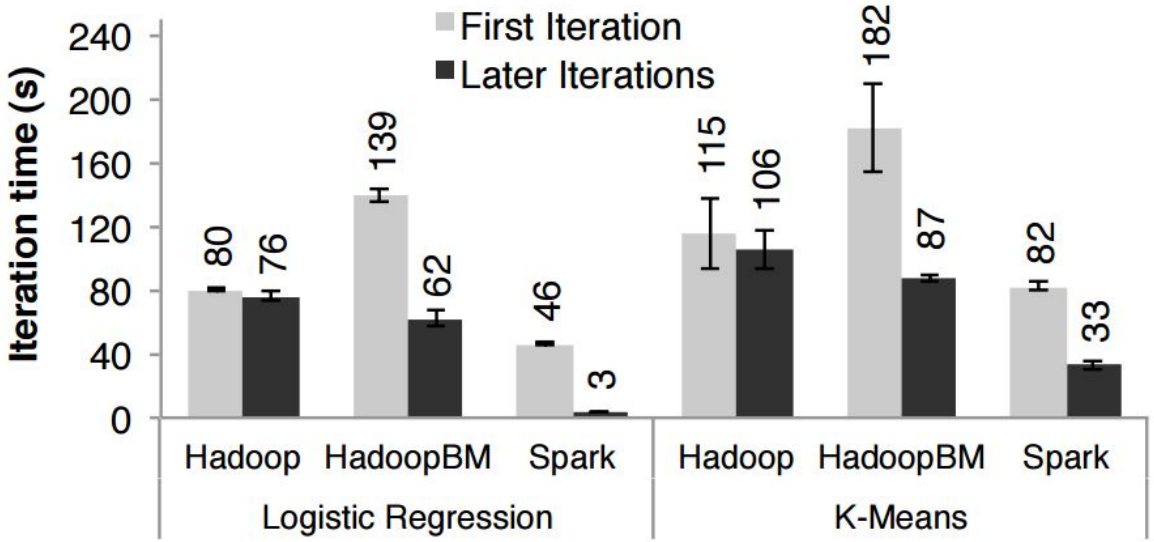


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

- Skip the first iteration, since it's just text parsing.
- In later iterations, Spark is much faster (black bar).
- HadoopBM writes intermediate data in memory not HDFS.

What if the number of nodes increases?

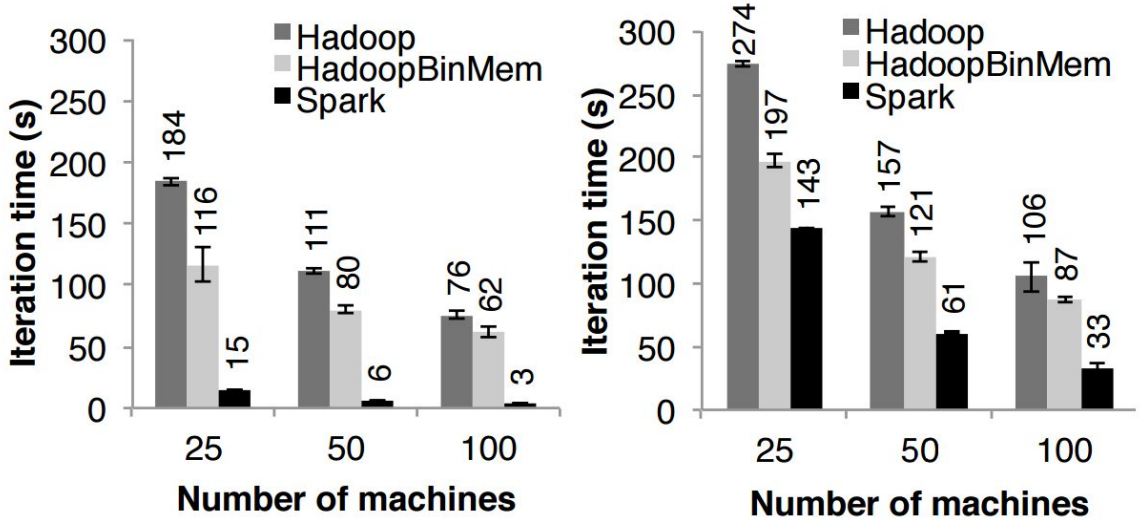
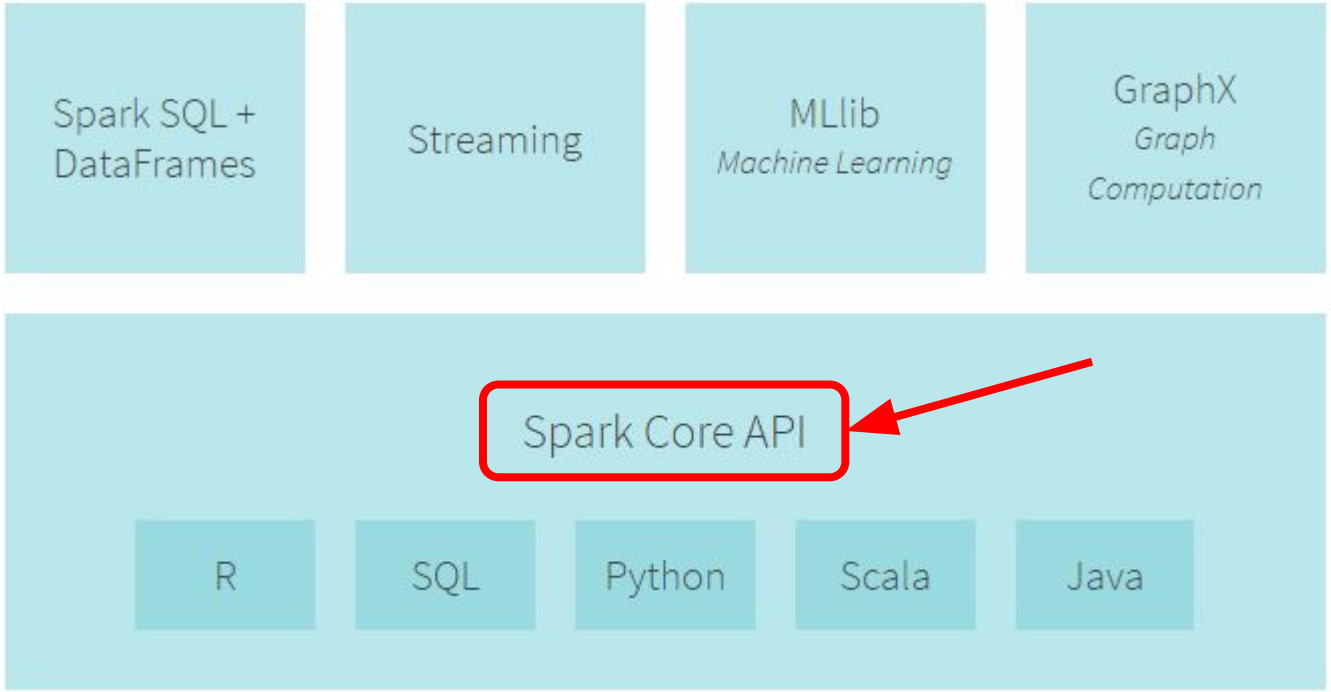


Figure 8: Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.

Apache Spark Ecosystem



References

- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing - Matei Zaharia, et al. - 2012
- https://databricks.com/spark/about
- http://www.slideshare.net/yongho/rdd-paper-review
- https://www.youtube.com/watch?v=dmL0N3qfSc8
- http://www.tothenew.com/blog/spark-1o3-spark-internals/
- https://trongkhoanguyenblog.wordpress.com/2014/11/27/understand-rdd-operations-transformations-and-actions/