

Introduction to Big Data

Based on "Big Data: Hype or Hallelujah?" by Elena Baralis

http://dbdmg.polito.it/wordpress/wp-content/uploads/2010/12/BigData_2015_2x.pdf

Big data

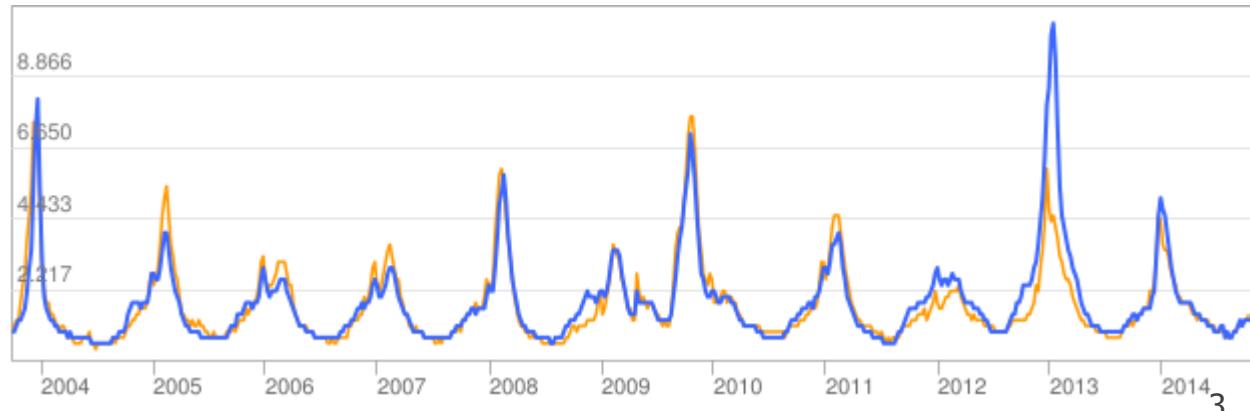


Google Flu trends



- February 2010
 - Google detected flu outbreak two weeks ahead of CDC data (Centers for Disease Control and Prevention – U.S.A)
 - Based on the analysis of Google search queries

it says that how to use our existed data to solve a problem



Google Flu trends

google.org Flu Trends

[Google.org home](#)

Flu Trends

Select country/region ▾

Home

[How does this work?](#)

FAQ

Flu activity

Intense

High

Moderate

Low

Minimal

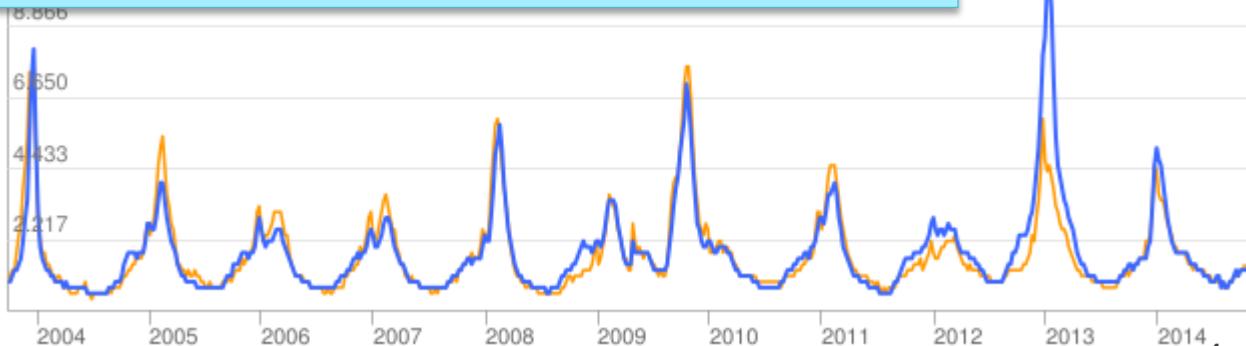
Explore flu trends around the world

We've found that certain search terms are good indicators of flu activity. Google Flu Trends uses aggregated Google search data to estimate flu activity. [Learn more ▾](#)



- February 2010
 - Google detected flu outbreak two weeks ahead of CDC data (Centers for Disease Control and Prevention)

Nowcasting analysis of search queries



it means how to predict our current problem solution

Data on the Internet...

■ Internet live stats

- <http://www.internetlivestats.com/>



4,485,508,861

Internet Users in the world



1,752,142,970

Total number of Websites



193,688,718,339

Emails sent [today](#)



5,222,289,027

Google searches [today](#)



4,990,992

Blog posts written [today](#)



572,159,945

Tweets sent [today](#)



5,348,093,035

Videos viewed [today](#) on YouTube



62,832,046

Photos uploaded [today](#) on Instagram



107,175,151

Tumblr posts [today](#)



2,435,900,914

Facebook active users



795,537,418

Google+ active users



357,398,865

Twitter active users



278,573,312

Pinterest active users



287,236,096

Skype calls [today](#)



110,341

Websites hacked [today](#)



5,664,059,486 GB

Internet traffic [today](#)



3,065,544 MWh

Electricity used [today](#) for the Internet



2,507,959 tons

CO₂ emissions [today](#)

Who generates big data?

- User Generated Content (Web & Mobile)
 - E.g., Facebook, Instagram, Yelp, TripAdvisor, Twitter, YouTube



- Health and scientific computing

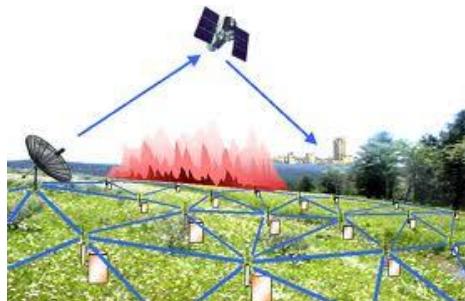


Who generates big data?

- Log files
 - Web server log files, machine system log files

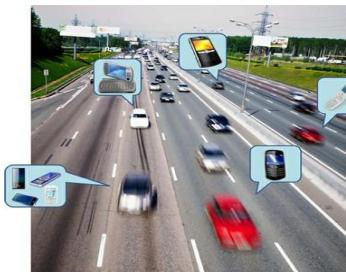


- Internet Of Things (IoT)
 - Sensor networks, RFIDs, smart meters



An example of Big data at work

Crowdsourcing



Sensing



Map data



Computing



Real time traffic info

Travel time forecast/nowcast

The Vs of big data

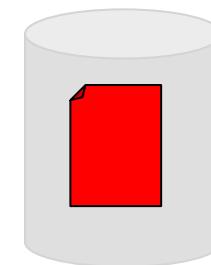
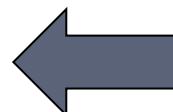
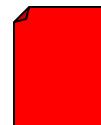
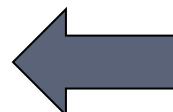
- The 3Vs of big data
 - Volume: scale of data
 - Variety: different forms of data
 - Velocity: analysis of streaming data
- ... but also
 - Veracity: uncertainty of data
 - Value: exploit information provided by data

Big data challenges

- Technology and infrastructure
 - New architectures, programming paradigms and techniques are needed
- Data management and analysis
 - New emphasis on “data”
 -  **Data science**

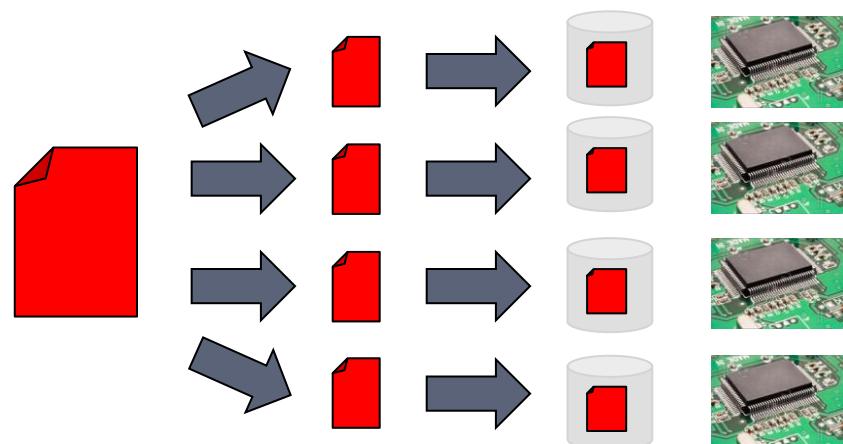
The bottleneck

- Processors process data
- Hard drives store data
- We need to transfer data from the disk to the processor



The solution

- Transfer the processing power to the data
- Multiple distributed disks
 - Each one holding a portion of a large dataset
- Process in parallel different file portions from different disks



Big data architectures

Big data architectures

- “A *big data architecture* is designed to handle the ingestion, processing, and analysis of data that is too large or complex for traditional database systems

...

Big data solutions typically involve one or more of the following types of workload:

- **Batch processing** of big data sources at rest
- **Real-time processing** of big data in motion
- **Interactive exploration** of big data
- **Predictive analytics** and **machine learning**

Big data architectures

Consider big data architectures when you need to:

- *Store and process **data in volumes too large** for a traditional database*
- *Transform **unstructured data** for analysis and reporting*
- *Capture, process, and analyze **unbounded streams of data** in real time, or **with low latency**"*

Big data architectures

- The most frequently used big data architecture is the **Lambda Architecture**
- The lambda architecture was proposed by Nathan Marz in 2011

Lambda architecture: Definition #1

- Nathan Marz

*“The past decade has seen a huge amount of innovation in scalable data systems. These include **large-scale computation systems** like Hadoop and databases such as Cassandra and Riak. These systems can handle very large amounts of data, but with serious trade-offs.*

Lambda architecture: Definition #1

Hadoop, for example, can parallelize large-scale batch computations on very large amounts of data, but the computations have high latency. You don't use Hadoop for anything where you need low-latency results.

Lambda architecture: Definition #1

NoSQL databases like Cassandra achieve their **scalability** by offering you a **much more limited data model** than you're used to with something like SQL. Squeezing your application into these limited data models can be very complex. And because the **databases are mutable**, they're **not human-fault tolerant**.

Lambda architecture: Definition #1

These tools on their own are not a panacea. But when intelligently used in conjunction with one another, you can produce scalable systems for arbitrary data problems with human-fault tolerance and a minimum of complexity. This is the Lambda Architecture you'll learn throughout the book. "

Lambda architecture: Definition #2

- Source: Databricks

“Lambda architecture is a way of processing massive quantities of data (i.e. “Big Data”) that provides access to batch-processing and stream-processing methods with a hybrid approach.

Lambda architecture is used to solve the problem of computing arbitrary functions.”

Lambda architecture: Definition #3

- Source: Wikipedia

*“Lambda architecture is a **data-processing architecture** designed to **handle massive quantities of data** by taking advantage of both **batch** and **stream-processing** methods.*

Lambda architecture: Definition #3

*This approach to architecture attempts to **balance latency**, **throughput**, and **fault-tolerance** by using **batch** processing to provide **comprehensive** and **accurate views of batch data**, while simultaneously using **real-time stream** processing to provide **views of online data**.*

The two view outputs may be joined before presentation.

Lambda architecture: Definition #3

*Lambda architecture depends on a **data model with an append-only, immutable data source** that serves as a system of record. It is intended for ingesting and processing timestamped events that are appended to existing events rather than overwriting them. State is determined from the natural time-based ordering of the data.”*

Lambda architecture: Requirements

- **Fault-tolerant** against both hardware failures and human errors
- **Support variety of use cases** that include low latency querying as well as updates
- Linear **scale-out capabilities** horizontal increase for computation
- **Extensible**, so that the system is manageable and can accommodate newer features easily

Lambda architecture: Queries

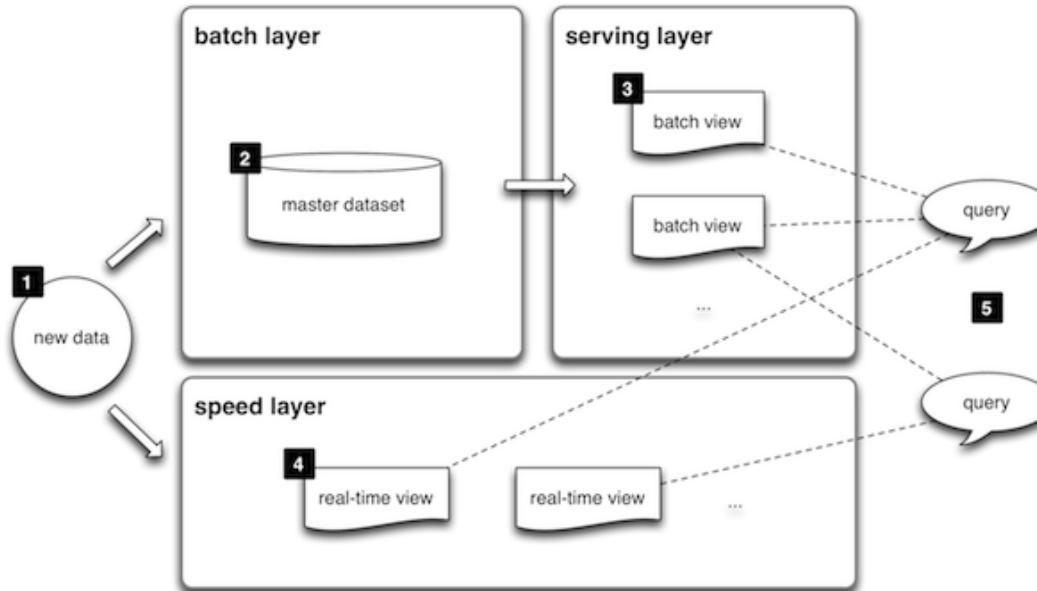
query = function(all data)

- Some query properties
 - Latency
 - The time it takes to run a query
 - Timeliness
 - How up to date the query results are (**freshness** and **consistency**)
 - Accuracy
 - Tradeoff between performance and scalability (**approximations**)

Lambda architecture

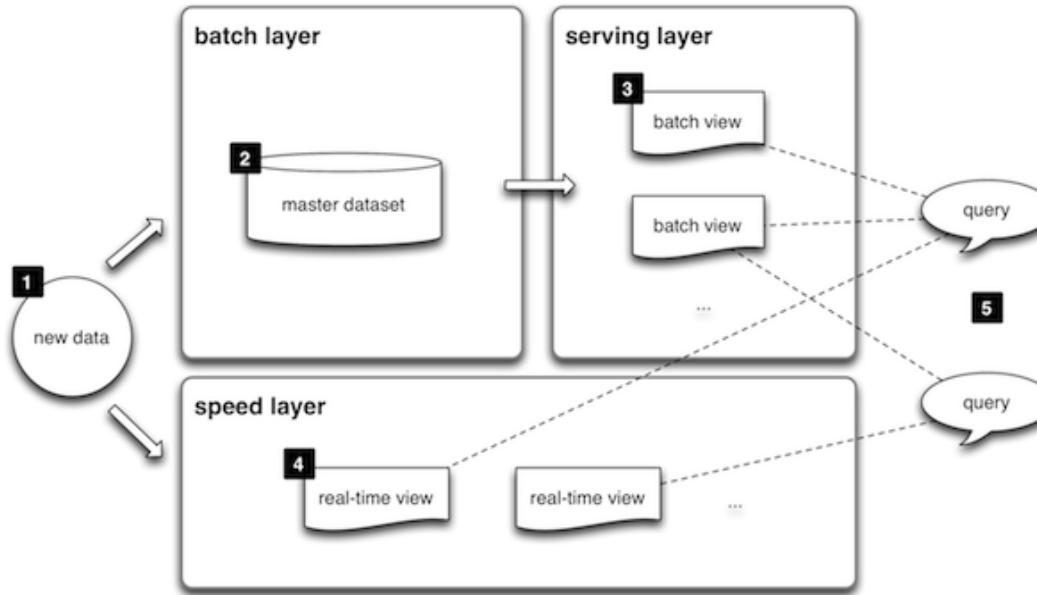
- It is based on two data paths:
 - **Cold path (batch layer)**
 - It stores all of the incoming **data in its raw form** and performs **batch processing on the data**
 - The result of this processing is stored as batch views
 - **Hot path (speed layer)**
 - It **analyzes data in real time**
 - This path is designed for **low latency, at the expense of accuracy**

Lambda architecture



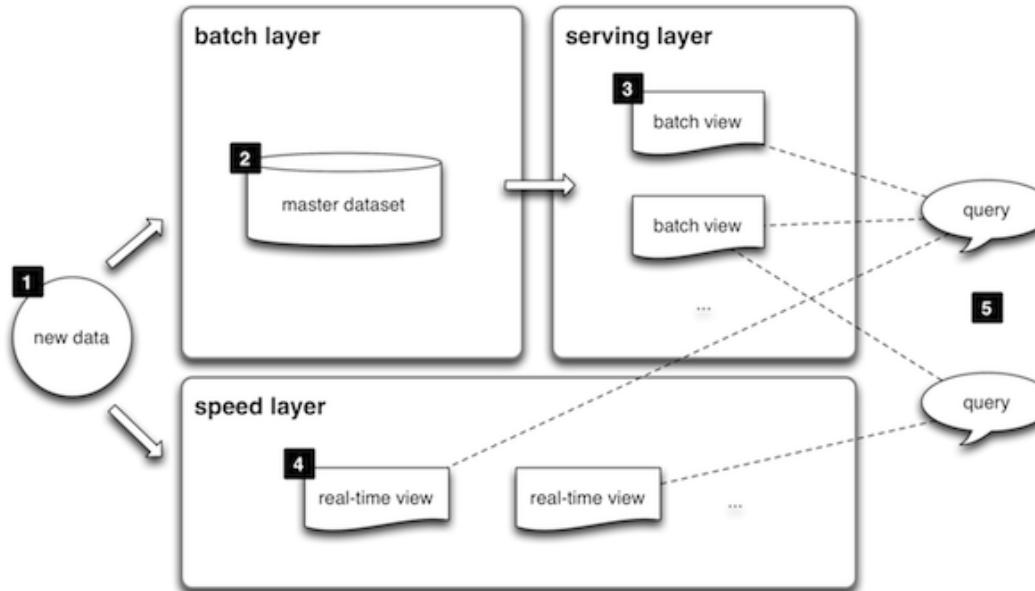
these two layers can be combined to get target queries

Lambda architecture



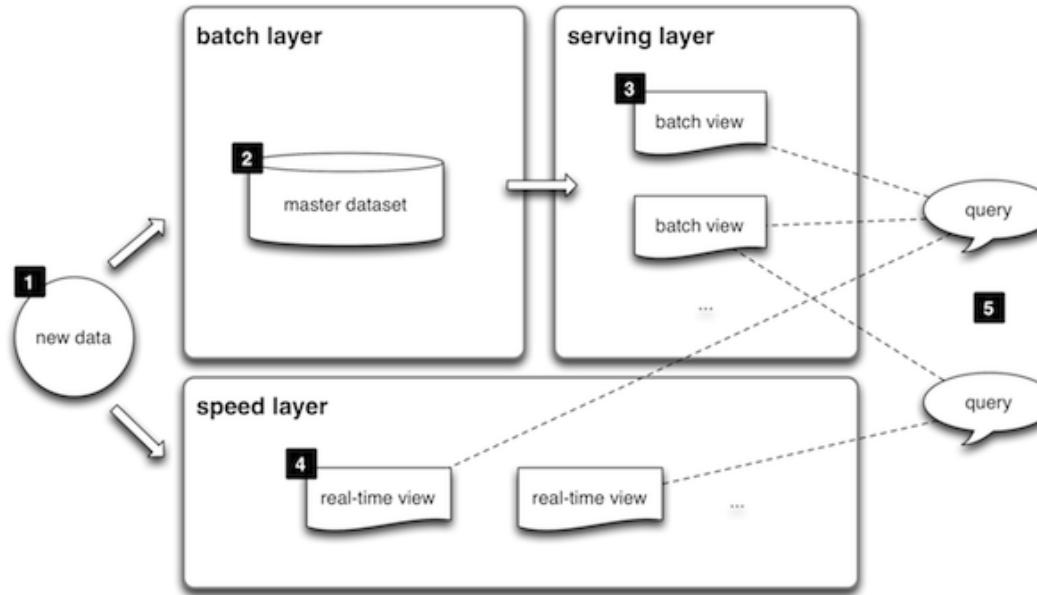
1. All data entering the system is dispatched to both the batch layer and the speed layer for processing

Lambda architecture



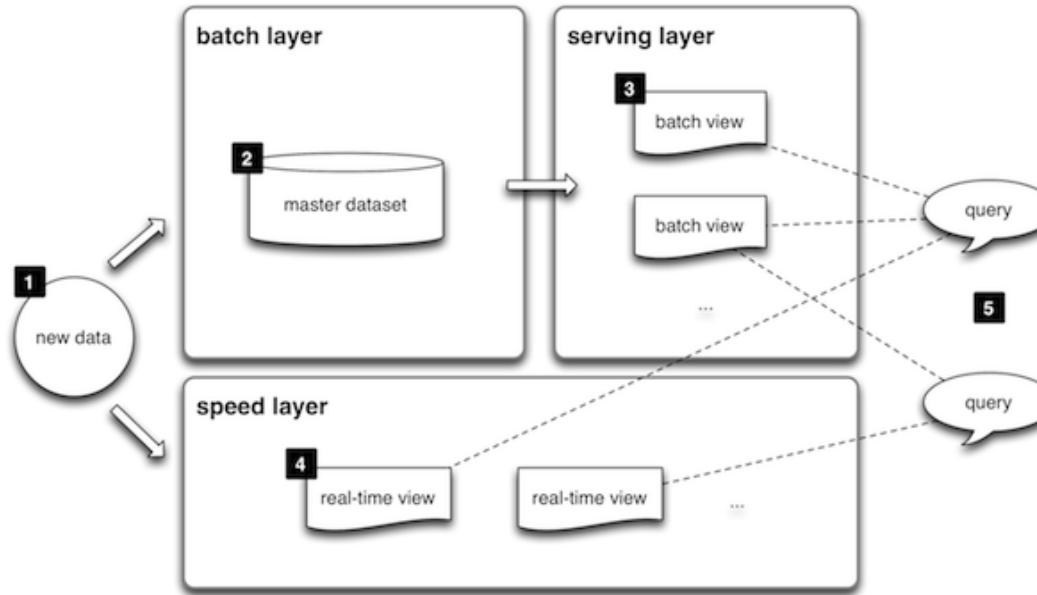
2. The **batch layer** has two functions:
 - (i) **managing** the **master dataset** (an **immutable, append-only** set of **raw data**), and
 - (ii) to **pre-compute** the **batch views**

Lambda architecture



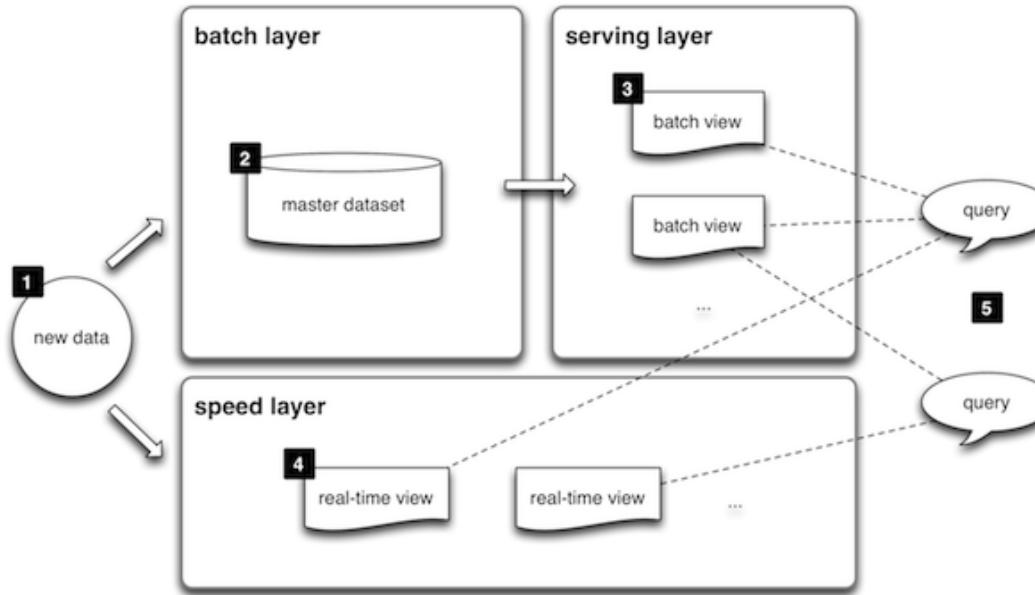
3. The **serving layer indexes** the **batch views** so that they can be **queried in low-latency, ad-hoc way**

Lambda architecture



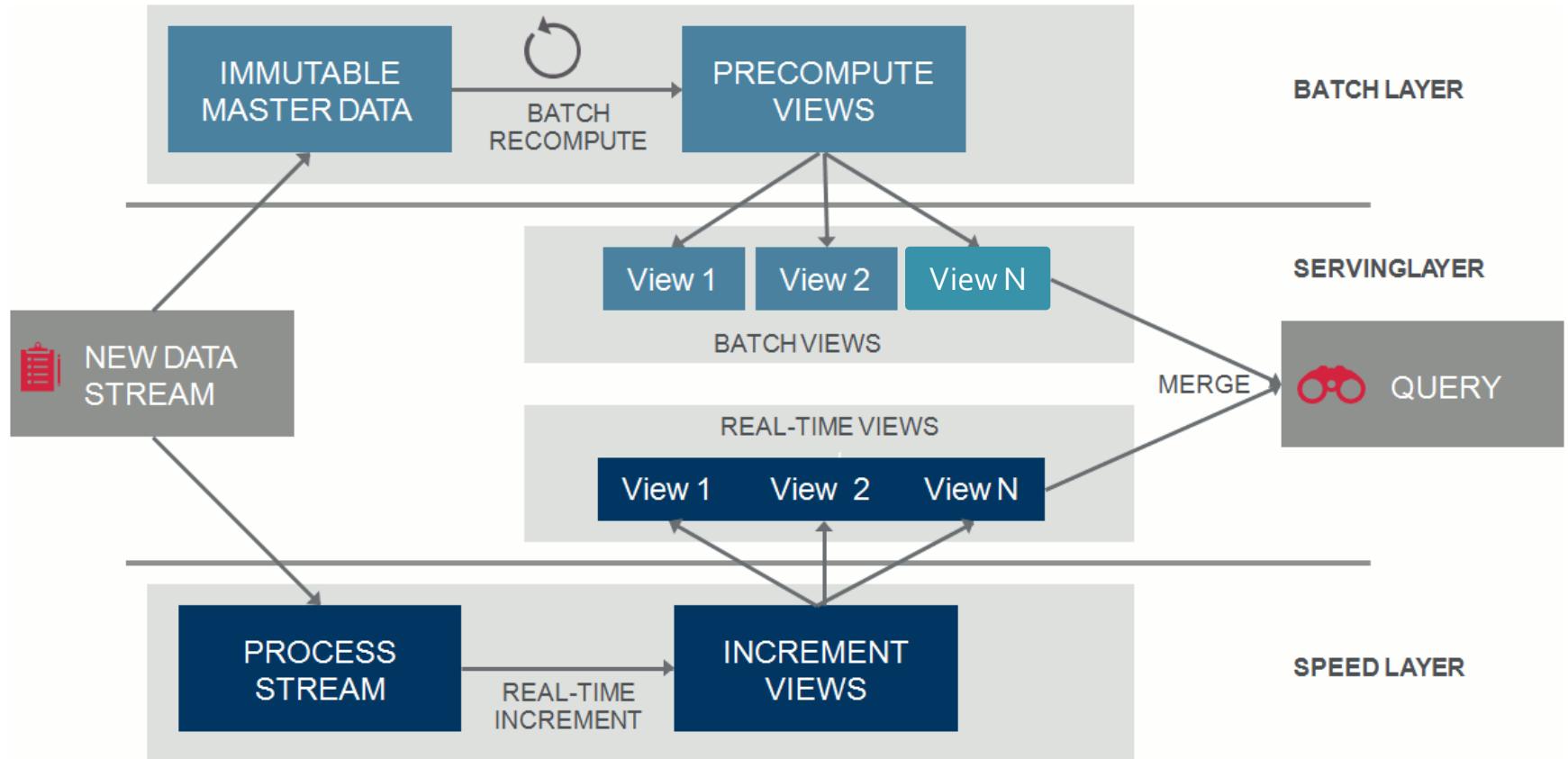
4. The **speed layer** compensates for the high latency of updates to the serving layer and **deals with recent data only**

Lambda architecture

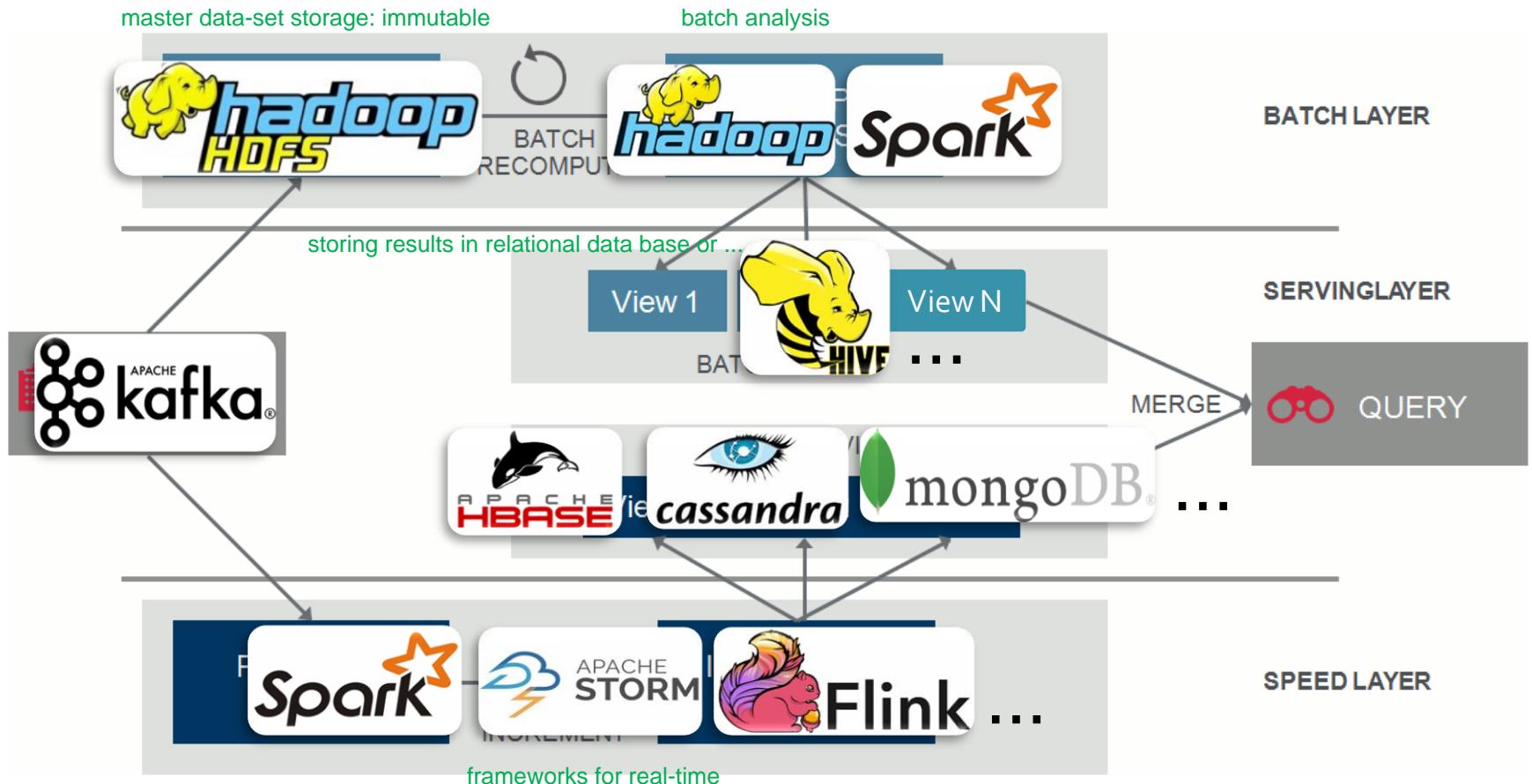


5. Any incoming **query** can be **answered by merging results from batch views and real-time views**

Lambda architecture: A more detailed view



Lambda architecture: Possible instances of each block



Introduction to Hadoop and MapReduce

Motivations of Hadoop and Big Data Frameworks

Data volumes

- The amount of data increases every day
- Some numbers (~ 2012):
 - Data processed by Google every day: 100+ PB
 - Data processed by Facebook every day: 10+ PB
- To analyze them, systems that scale with respect to the data volume are needed

Data volumes: Google Example

- Analyze **10 billion web pages**
- Average size of a webpage: **20KB**
- Size of the collection: $10 \text{ billion} \times 20\text{KBs} = 200\text{TB}$
- HDD hard disk read bandwidth: **150MB/sec**
- Time needed to **read all web pages (without analyzing them)**: 2 million seconds = more than **15 days**
- A single node architecture is not adequate

Data volumes: Google Example with SSD

- Analyze **10 billion web pages**
- Average size of a webpage: **20KB**
- Size of the collection: $10 \text{ billion} \times 20\text{KBs} = 200\text{TB}$
- SSD hard disk read bandwidth: **550MB/sec**
- Time needed to **read all web pages (without analyzing them)**: 2 million seconds = more than **4 days**
- A single node architecture is not adequate

Failures

- Failures are part of everyday life, especially in data center
 - A single server stays up for 3 years (~1000 days)
 - 10 servers → 1 failure every 100 days (~3 months)
 - 100 servers → 1 failure every 10 days
 - 1000 servers → 1 failure/day
- Sources of failures
 - Hardware/Software
 - Electrical, Cooling, ...
 - Unavailability of a resource due to overload

Failures

- LALN data [DSN 2006]
 - Data for 5000 machines, for 9 years
 - Hardware failures: 60%, Software: 20%, Network 5%
- DRAM error analysis [Sigmetrics 2009]
 - Data for 2.5 years
 - 8% of DIMMs affected by errors
- Disk drive failure analysis [FAST 2007]
 - Utilization and temperature major causes of failures

Failures

- Failure types
 - Permanent
 - E.g., Broken motherboard
 - Transient
 - E.g., Unavailability of a resource due to overload

Network bandwidth

- Network becomes the bottleneck if big amounts of data need to be exchanged between nodes/servers
 - Network bandwidth (in a data center): 10Gbps
 - Moving 10 TB from one server to another takes more than 2 hours
 - Data should be moved across nodes only when it is indispensable
 - Usually, codes/programs are small (few MBs)
 - Move code (programs) and computation to data

Network bandwidth

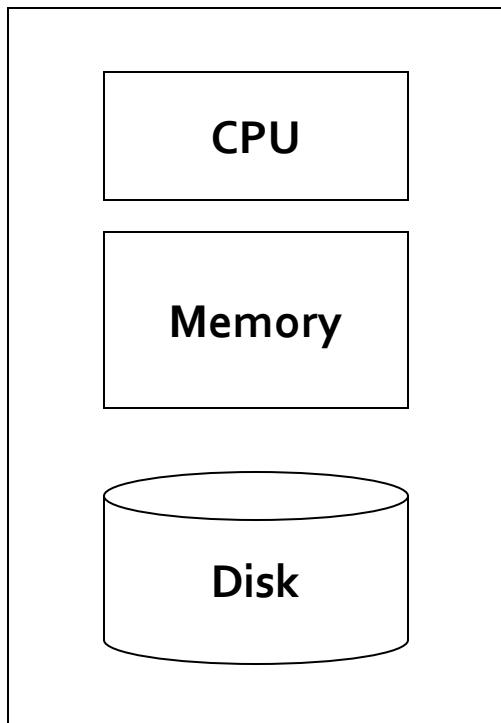
- Network becomes the bottleneck if big amounts of data need to be exchanged between nodes/servers
 - Network bandwidth (in a data center): 10Gbps
 - Moving 10 TB from one server to another takes more than 2 hours
 - **Data** should be **moved across nodes only when** it is **indispensable**
 - Usually, codes/programs are small (few MBs)
 - **Move code** (programs) and **computation** to data



Data locality

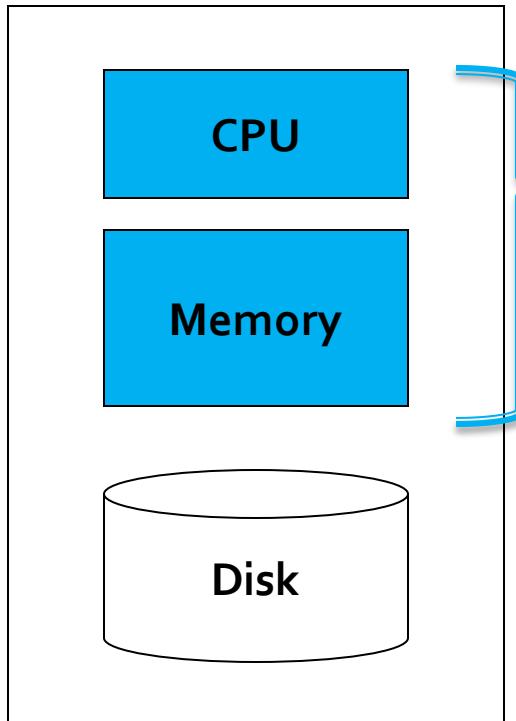
Single-node architecture

Server (Single node)



Single-node architecture

Server (Single node)

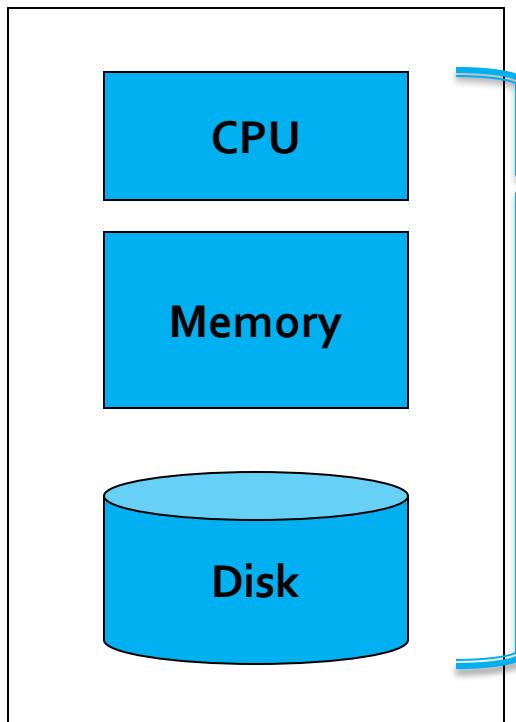


Machine Learning, Statistics

- Small data
 - Data can be completely loaded in main memory

Single-node architecture

Server (Single node)



here, since we have all data in main memory, it is possible to have global view of data and perform more complex algorithms or queries

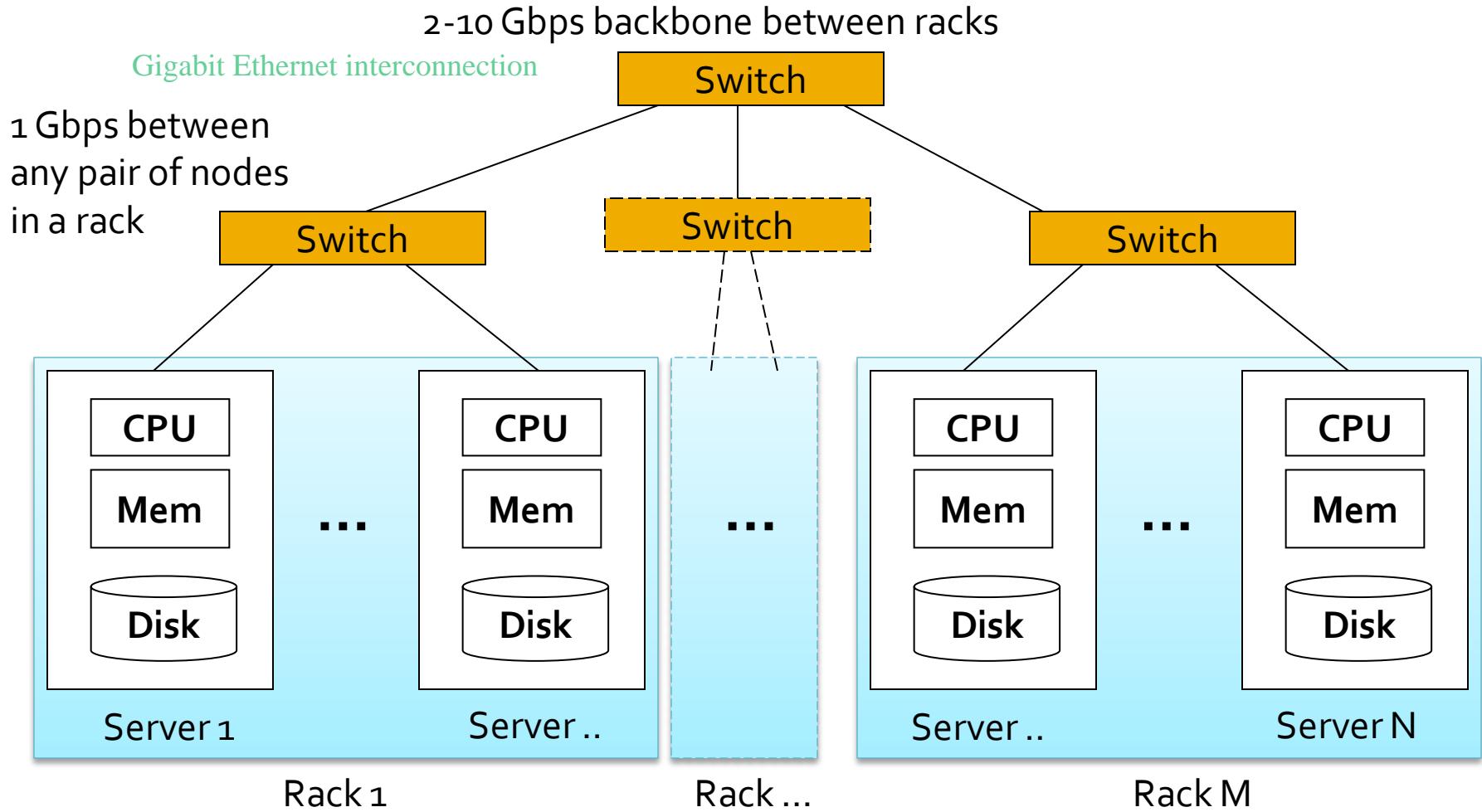
“Classical” data mining

- Large data
 - Data can not be completely loaded in main memory
 - Load in main memory one chunk of data at a time
 - Process it and store some statistics
 - Combine statistics to compute the final result

Cluster Architecture

- Cluster of servers (data center)
 - Computation is distributed across servers
 - Data are stored/distributed across servers
- Standard architecture in the Big data context
(~ 2012)
 - Cluster of commodity Linux nodes/servers
 - 32 GB of main memory per node
 - Gigabit Ethernet interconnection

Commodity Cluster Architecture



Each rack contains 16-64 nodes

the servers are needed to be cheap, bc we perform just simple algorithms but on large data

Data center



Data center



Scalability

- Current systems must scale to address
 - The increasing amount of data to analyze
 - The increasing number of users to serve
 - The increasing complexity of the problems
- Two approaches are usually used to address scalability issues
 - Vertical scalability (scale up)
 - Horizontal scalability (scale out)

Scale up vs. Scale out

- Vertical scalability (scale up)
 - Add more power/resources (main memory, CPUs) to a single node (high-performing server)
 - Cost of super-computers is not linear with respect to their resources
- Horizontal scalability (scale out)
 - Add more nodes (commodity servers) to a system
 - The cost scales approximately linearly with respect to the number of added nodes
 - But data center efficiency is a difficult problem to solve

Scale up vs. Scale out

- For data-intensive workloads, a large number of commodity servers is preferred over a small number of high-performing servers
 - At the same cost, we can deploy a system that processes data more efficiently and is more fault-tolerant
- Horizontal scalability (scale out) is preferred for big data applications
 - But distributed computing is hard
→ New systems hiding the complexity of the distributed part of the problem to developers are needed

Cluster computing challenges

- **Distributed programming** is hard
 - Problem decomposition and parallelization
 - Task synchronization
- **Task scheduling** of distributed applications is critical
 - Assign tasks to nodes by trying to
 - Speed up the execution of the application
 - Exploit (almost) all the available resources
 - Reduce the impact of node failures

Cluster computing challenges

- **Distributed data storage**
 - How do we store data persistently on disk and keep it available if nodes can fail?
 - Redundancy is the solution, but it increases the complexity of the system
- **Network bottleneck**
 - Reduce the amount of data send through the network
 - Move computation and code to data

Cluster computing challenges

- **Distributed computing** is not a new topic
 - HPC (High-performance computing) ~1960
 - Grid computing ~1990
 - Distributed databases ~1990
- Hence, many solutions to the mentioned challenges are already available
- But we are now facing big data driven-problems
 - The former solutions are not adequate to address big data volumes

Typical Big Data Problem

- Typical Big Data Problem
 - Iterate over a large number of records/objects
 - Extract something of interest from each record/object
 - Aggregate intermediate results
 - Generate final output
- The challenges:
 - Parallelization
 - Distributed storage of large data sets (Terabytes, Petabytes)
 - Node Failure management
 - Network bottleneck
 - Diverse input format (data diversity & heterogeneity)

Apache Hadoop

Apache Hadoop

- Scalable fault-tolerant distributed system for Big Data
 - Distributed Data Storage
 - Distributed Data Processing
 - Borrowed concepts/ideas from the systems designed at Google (Google File System for Google's MapReduce)
 - Open source project under the Apache license
 - But there are also many commercial implementations (e.g., Cloudera, Hortonworks, MapR)

Hadoop History

- Dec 2004 – Google published a paper about GFS
- July 2005 – Nutch uses MapReduce
- Feb 2006 – Hadoop becomes a Lucene subproject
- Apr 2007 – Yahoo! runs it on a 1000-node cluster
- Jan 2008 – Hadoop becomes an Apache Top Level Project
- Jul 2008 – Hadoop is tested on a 4000 node cluster

Hadoop History

- Feb 2009 – The Yahoo! Search Webmap is a Hadoop application that runs on more than 10,000 core Linux cluster
- June 2009 – Yahoo! made available the source code of its production version of Hadoop
- In 2010 Facebook claimed that they have the largest Hadoop cluster in the world with 21 PB of storage
 - On July 27, 2011 they announced the data has grown to 30 PB.

Who uses/have used Hadoop?

- Amazon
- Facebook
- Google
- IBM
- Joost
- Last.fm
- New York Times
- PowerSet
- Veoh
- Yahoo!
-

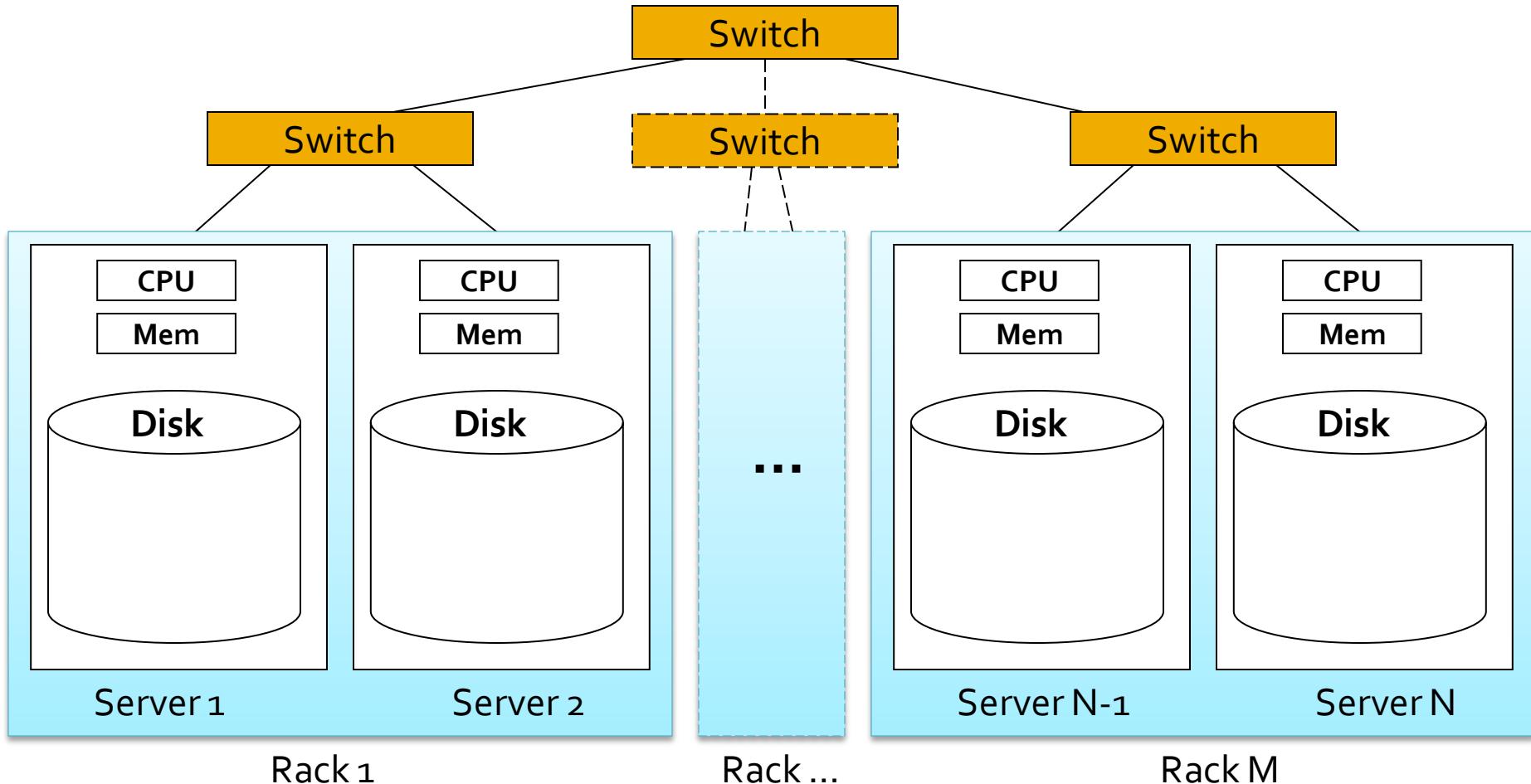
Hadoop vs. HPC

- **Hadoop**
 - Designed for **Data intensive workloads**
 - Usually, no CPU demanding/intensive tasks
- **HPC** (High-performance computing)
 - A supercomputer with a high-level computational capacity
 - Performance of a supercomputer is measured in floating-point operations per second (FLOPS)
 - Designed for **CPU intensive tasks**
 - Usually it is used to process “small” data sets

Hadoop: main components

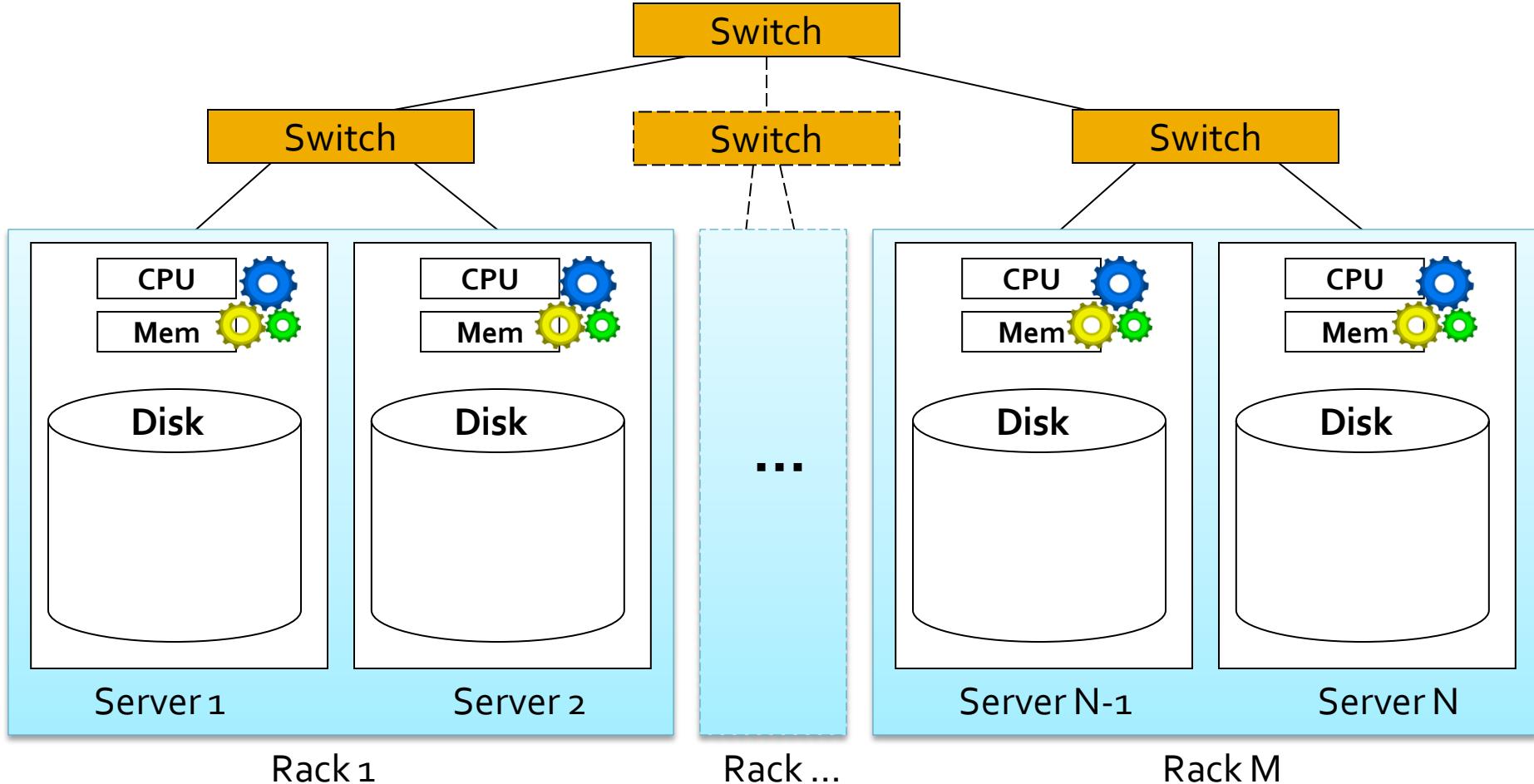
- Core components of Hadoop:
 - **Distributed Big Data Processing Infrastructure** based on the MapReduce programming paradigm
 - Provides a high-level abstraction view
 - Programmers do not need to care about task scheduling and synchronization
 - Fault-tolerant
 - Node and task failures are automatically managed by the Hadoop system
 - **HDFS (Hadoop Distributed File System)**
 - High availability distributed storage
 - Fault-tolerant

Hadoop: main components



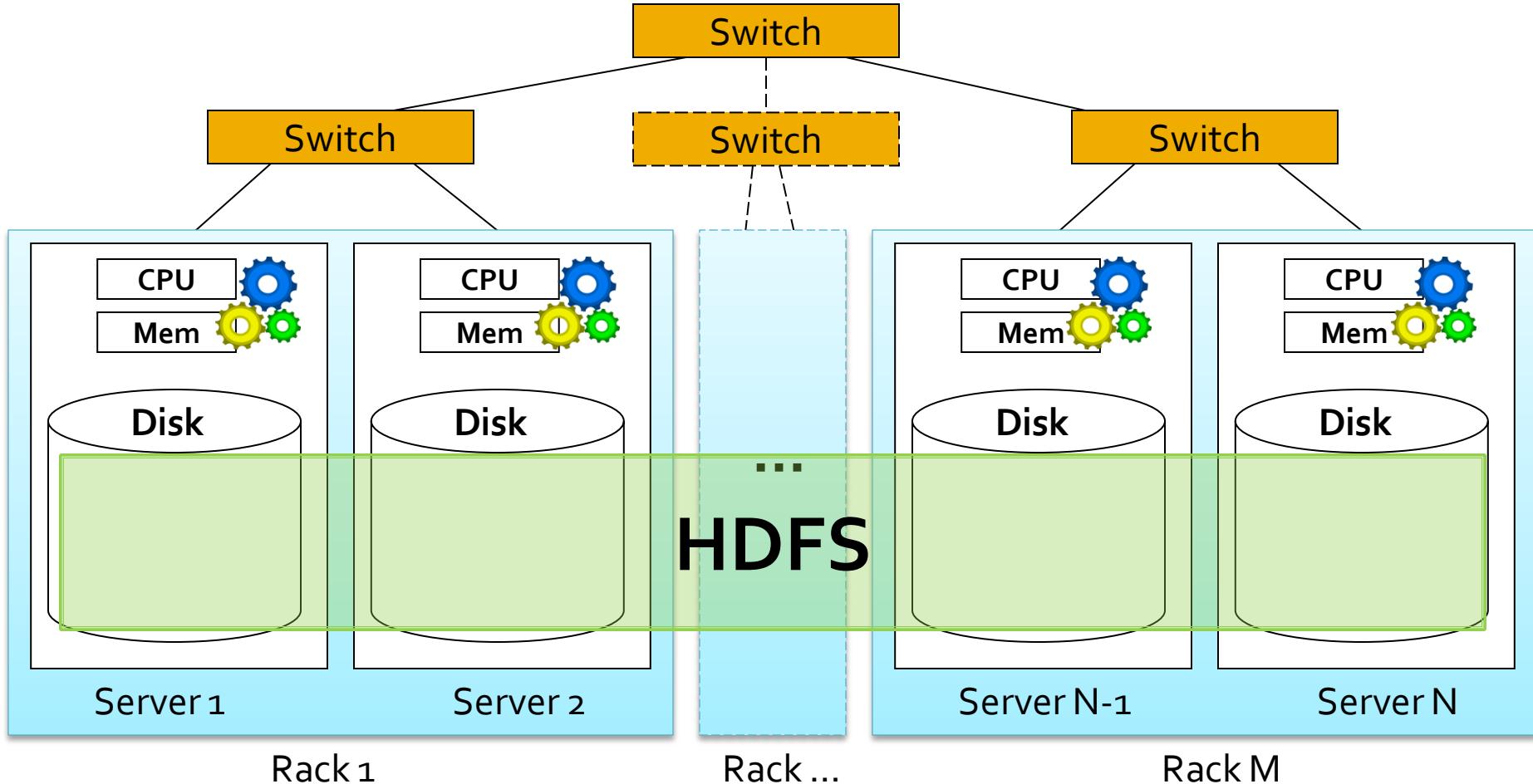
Example with number of replicas per chunk = 2

Hadoop: main components



Example with number of replicas per chunk = 2

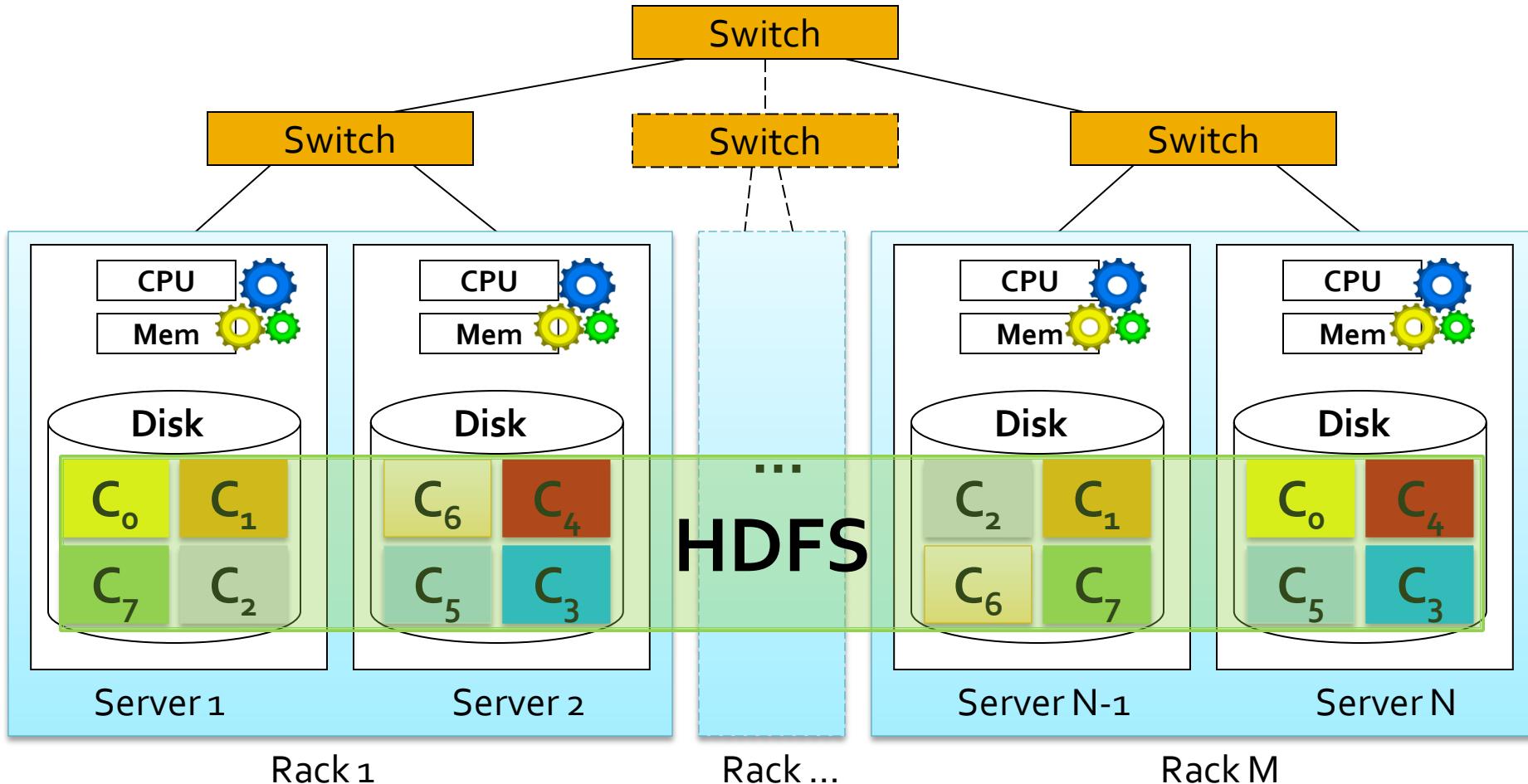
Hadoop: main components



Example with number of replicas per chunk = 2

Hadoop: main components

1. distributed file system = HDFS
2. yarn scheduler = to execute mapreduce application



Example with number of replicas per chunk = 2

Distributed Big Data Processing Infrastructure

- Separates the **what** from the **how**
 - Hadoop programs are based on the MapReduce programming paradigm
 - MapReduce abstracts away the “distributed” part of the problem (scheduling, synchronization, etc)
 - Programmers focus on **what**
 - The distributed part (scheduling, synchronization, etc) of the problem is handled by the framework
 - The Hadoop infrastructure focuses on **how**

Distributed Big Data Processing Infrastructure

- But an in-depth knowledge of the Hadoop framework is important to develop efficient applications
 - The **design of the application** must exploit data locality and limit network usage/data sharing

HDFS

- HDFS
 - Standard Apache Hadoop distributed file system
 - Provides global file namespace
 - Stores data redundantly on multiple nodes to provide persistence and availability
 - Fault-tolerant file system
- Typical usage pattern
 - Huge files (GB to TB)
 - Data is rarely updated
 - Reads and appends are common
 - Usually, random read/write operations are not performed

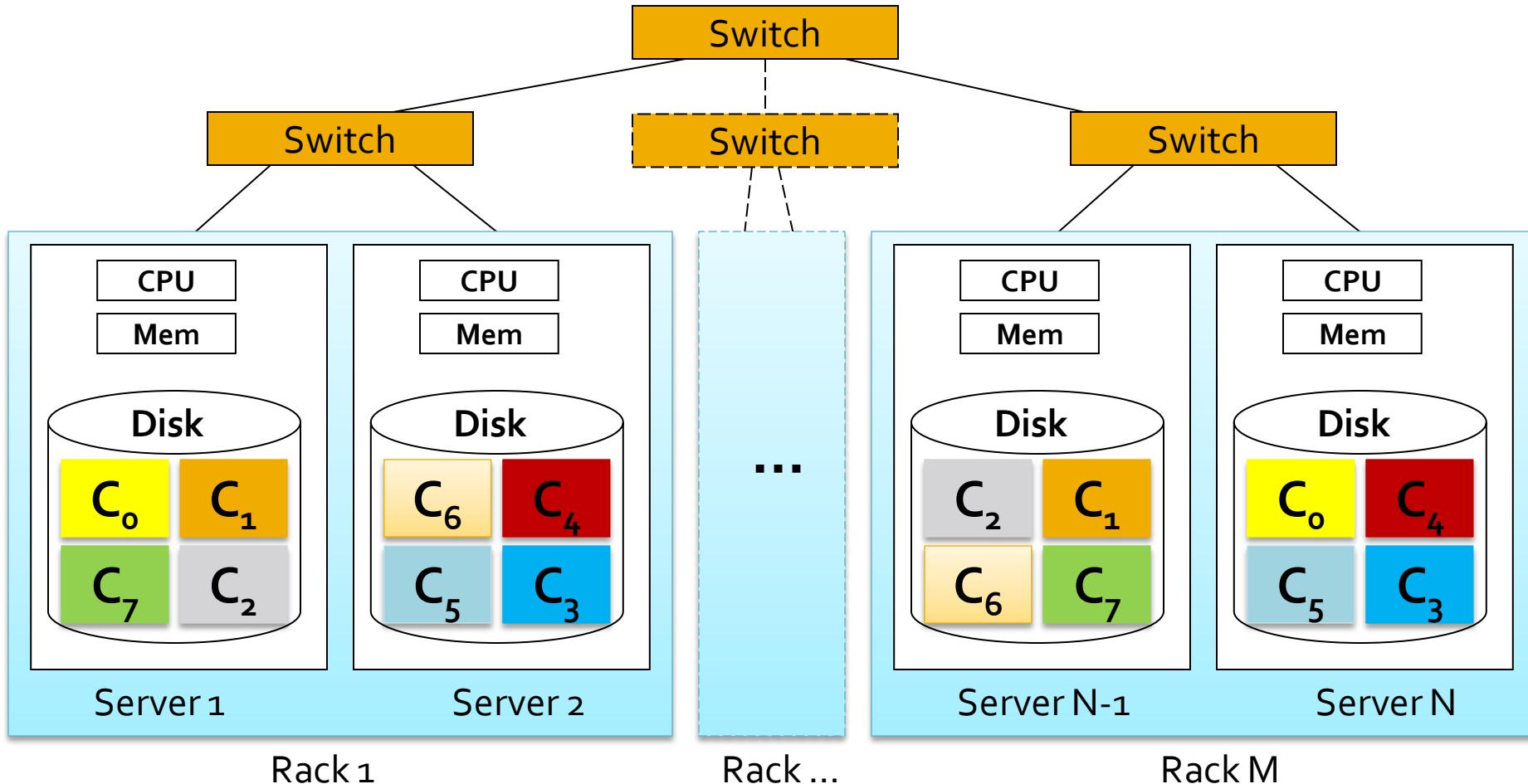
HDFS

- Each file is split in “chunks/blocks” that are spread across the servers
 - Each chunk is replicated on different servers (usually there are 3 replicas per chunk)
 - Ensures persistence and availability
 - To increase persistence and availability, replicas are stored in different racks, if it is possible
 - Each chunk/block contains a part of the content of one single file
 - You cannot have the content of two files in the same chunk/block
 - Typically each chunk is 64-128MB

f1.txt 129 MB -> 2 blocks * 3 = 6 blocks with replicas
f2.txt 10 MB -> 1 block * 3 = 3
so most of the blocks are empty

1 million files each 2KB, too many waste and network is slow and also they occupied a lot of space in meta data information, so we should merge them before uploading on distributed servers

HDFS



system can tolerate at most one failure here, after that it tries to make copies of failed server in the rest of servers

Example with number of replicas per chunk = 2

HDFS

- The **Master node, a.k.a. Name Nodes** in HDFS, is a special node/server that
 - Stores HDFS metadata
 - E.g., the mapping between the name of a file and the location of its chunks
 - Might be replicated
- **Client applications:** file access through HDFS APIs
 - Talk to the master node to find data/chunk servers associated with the file of interest
 - Connect to the selected chunk servers to access data

Hadoop ecosystem

- Many Hadoop-related projects/systems are available
 - **Hive**
 - A distributed relational database, based on MapReduce, for querying data stored in HDFS by means of a query language based on SQL
 - **HBase**
 - A distributed column-oriented database that uses HDFS for storing data
 - **Pig**
 - A data flow language and execution environment, based on MapReduce, for exploring very large datasets

Hadoop ecosystem

- **Sqoop**
 - A tool for efficiently moving data from traditional relational databases and external flat file sources to HDFS
- **ZooKeeper**
 - A distributed coordination service. It provides primitives such as distributed locks
-
- Each project/system addresses one specific class of problems

MapReduce: introduction

Warm up: Word Count

- Input
 - A large textual file of words
- Problem
 - Count the number of times each distinct word appears in the file
- Output
 - A list of pairs <word, number>, counting the number of occurrences of each specific word in the input file

Word Count

- Case 1: Entire file fits in main memory

Word Count

- Case 1: Entire file fits in main memory
 - A traditional single node approach is probably the most efficient solution in this case
 - The complexity and overheads of a distributed system affects the performance when files are “small”
 - “small” depends on the resources you have

Word Count

- Case 1: Entire file fits in main memory
 - A traditional single node approach is probably the most efficient solution in this case
 - The complexity and overheads of a distributed system affects the performance when files are “small”
 - “small” depends on the resources you have
- Case 2: File too large to fit in main memory

Word Count

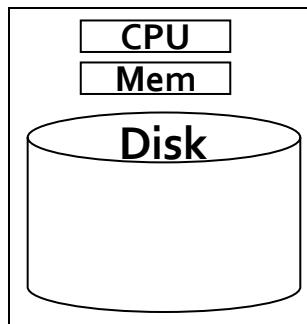
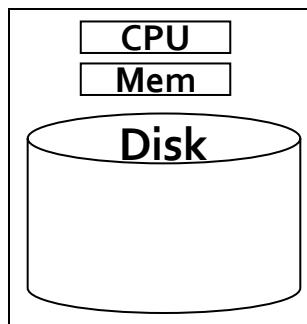
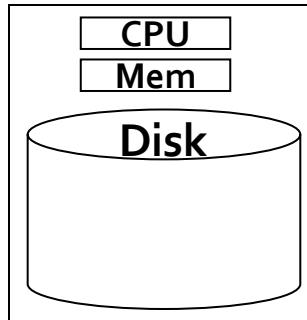
- Case 1: Entire file fits in main memory
 - A traditional single node approach is probably the most efficient solution in this case
 - The complexity and overheads of a distributed system affects the performance when files are “small”
 - “small” depends on the resources you have
- Case 2: File too large to fit in main memory
 - How can we split this problem in a set of (almost) **independent sub-tasks**, and
 - execute them **in parallel** on a cluster of servers?

Word Count with a very large file

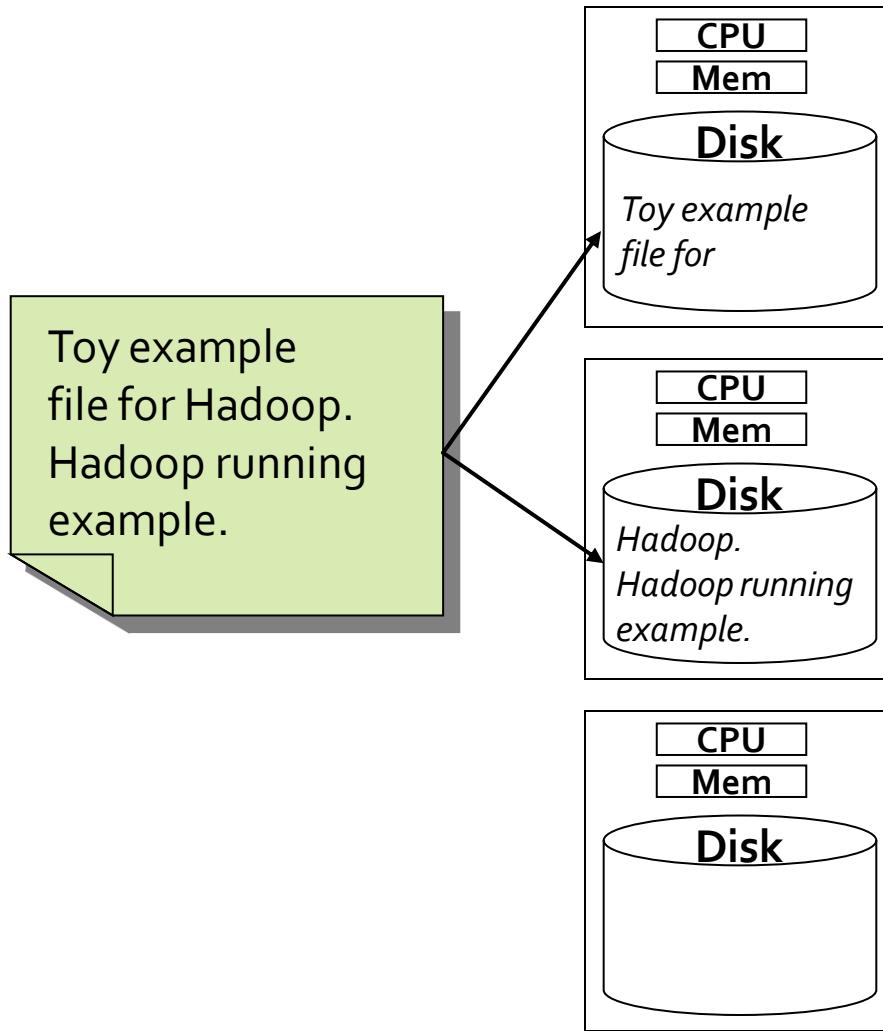
- Suppose that
 - The cluster has **3 servers**
 - The content of the input file is
 - “Toy example file for Hadoop. Hadoop running example.”
 - The input file is split into **2 chunks**
 - The number of **replicas** is **1**

Word Count with a very large file

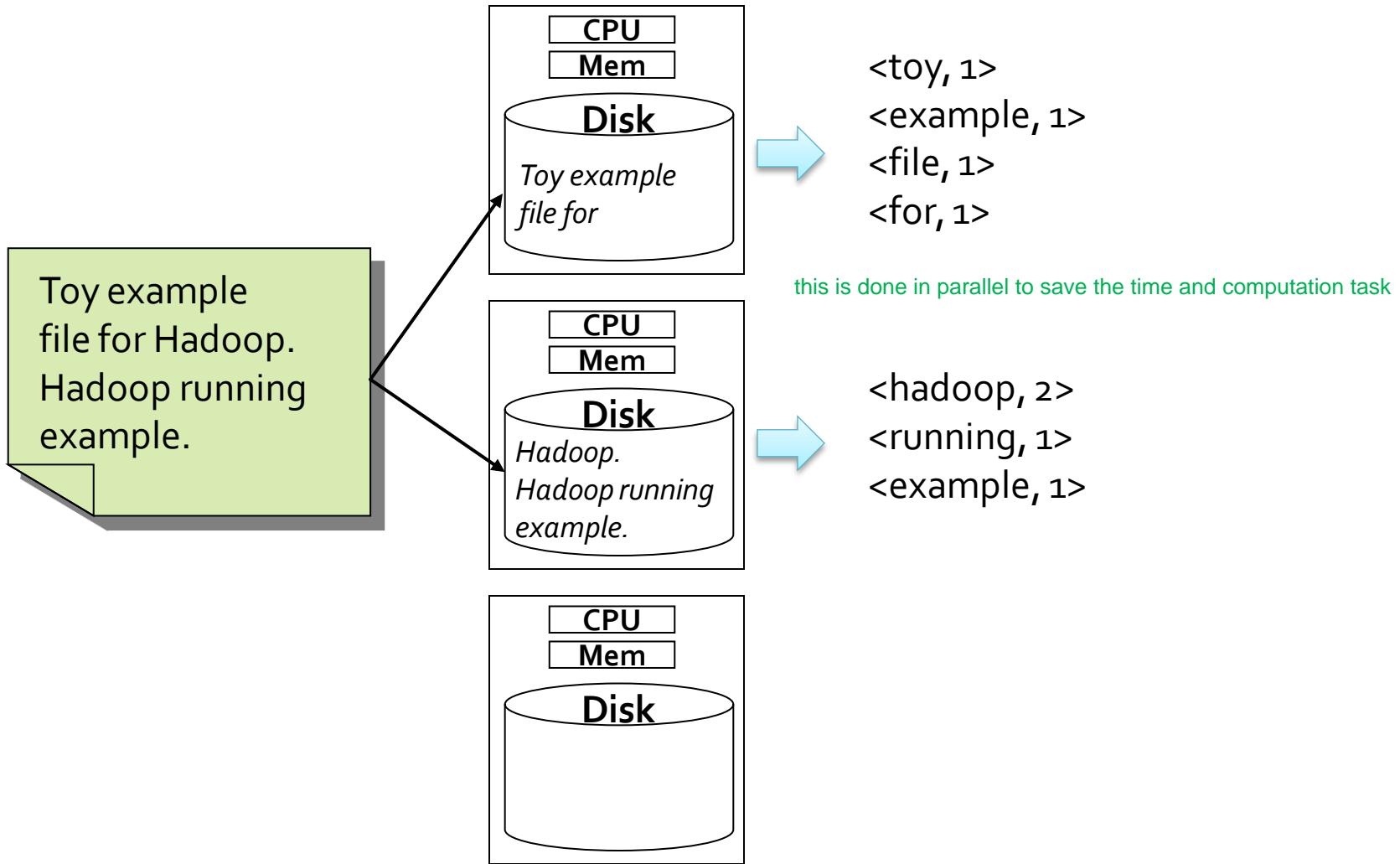
Toy example
file for Hadoop.
Hadoop running
example.



Word Count with a very large file



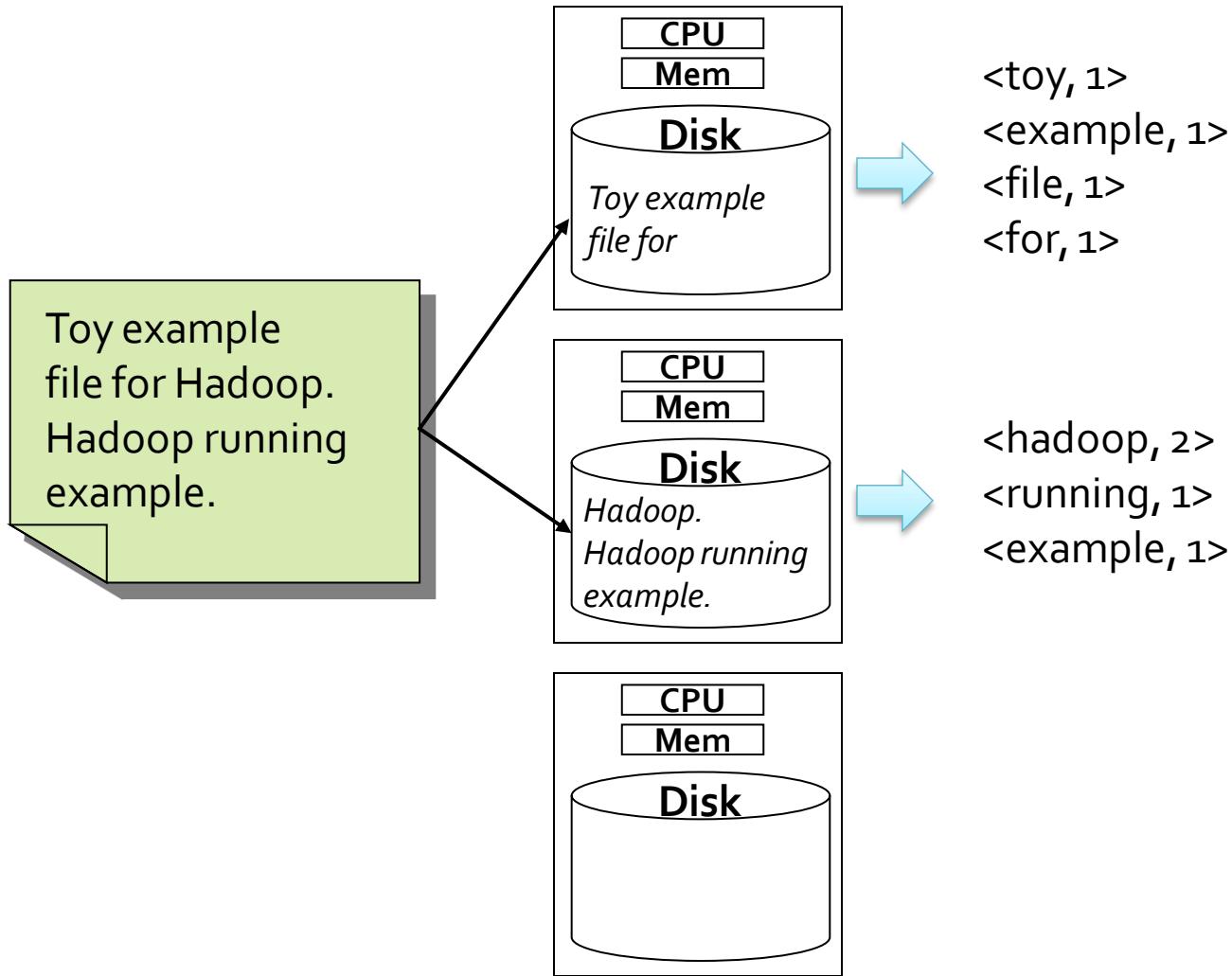
Word Count with a very large file



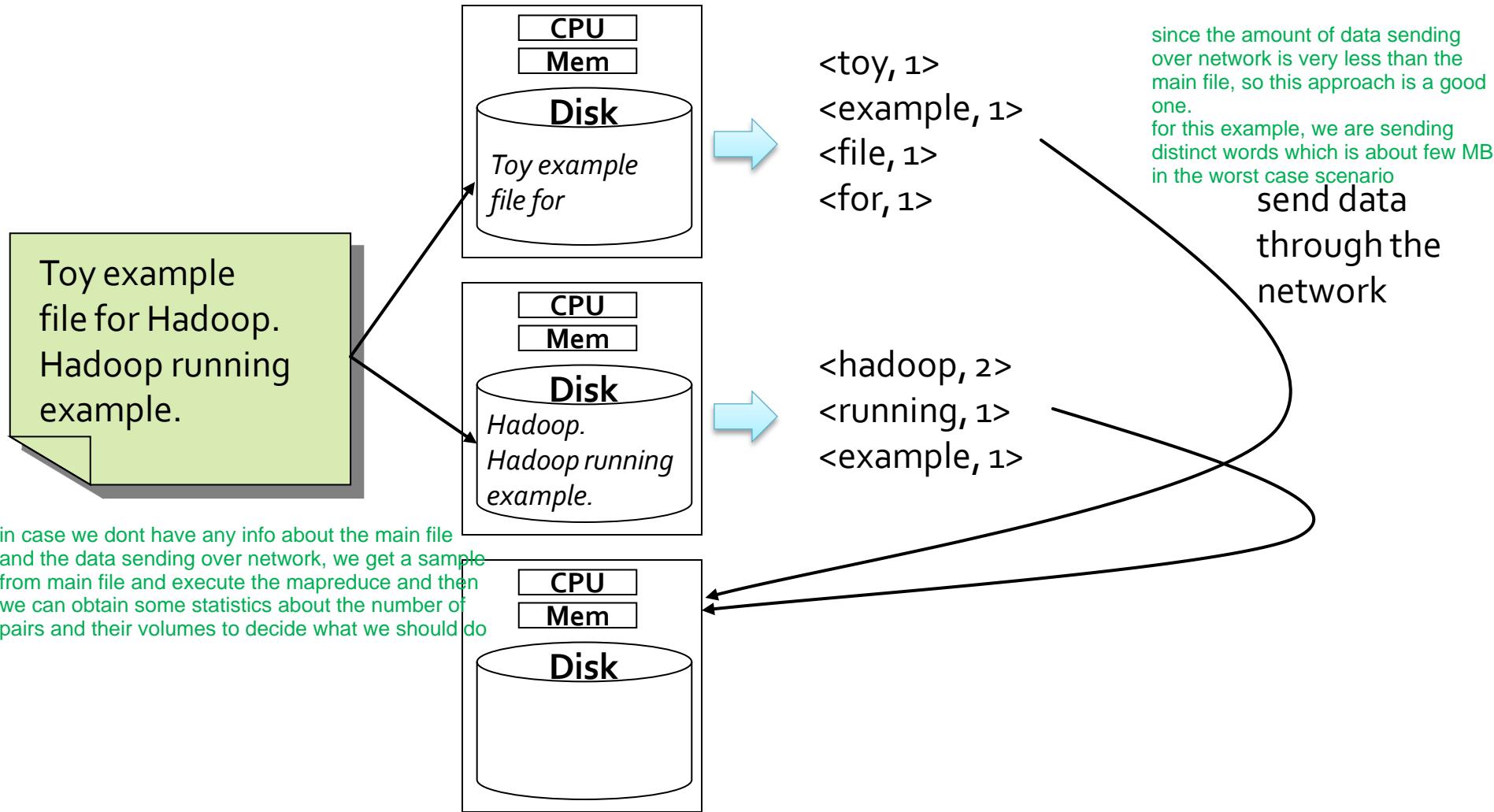
Word Count with a very large file

- The problem can be easily parallelized
 - 1. Each server processes its chunk of data and counts the number of times each word appears in its own chunk
 - Each server can execute its sub-task independently from the other servers of the cluster
→ synchronization is not needed in this phase
 - The output generated from each chunk by each server represents a partial result

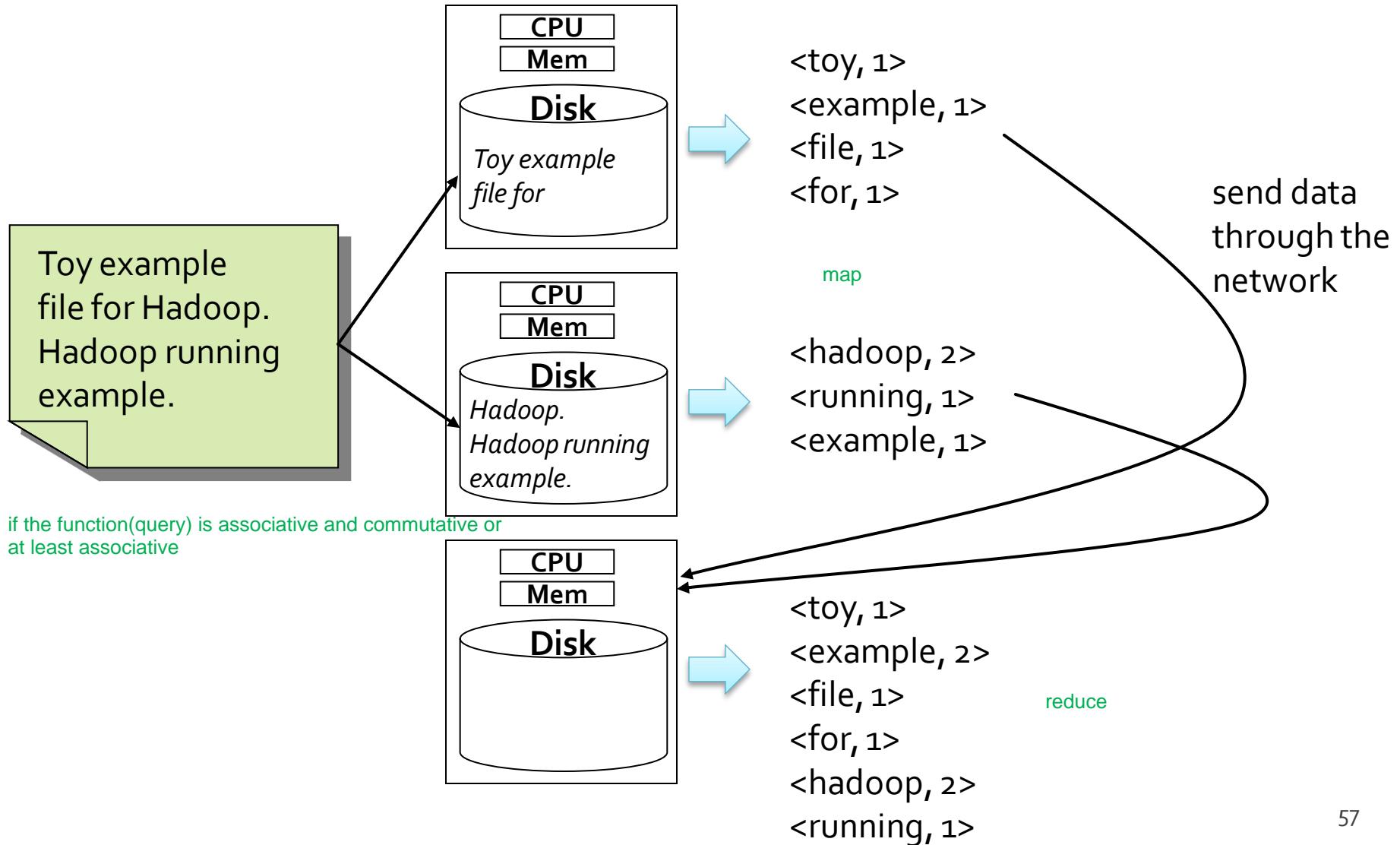
Word Count with a very large file



Word Count with a very large file



Word Count with a very large file



Word Count with a very large file

2. Each server sends its local (partial) list of pairs **<word, number of occurrences in its chunk>** to a server that is in charge of **aggregating** all local results and computing the global result
 - The server in charge of computing the global result needs to receive all the local (partial) results to compute and emit the final list
→ A synchronization operation is needed in this phase

Word Count: a more realistic example

- Case 2: File too large to fit in main memory
- Suppose that
 - The file size is 100 GB and the number of distinct words occurring in it is at most 1,000
 - The cluster has 101 servers
 - The file is spread across 100 servers and each of these servers contains one (different) chunk of the input file
 - i.e., the file is optimally spread across 100 servers (each server contains 1/100 of the file in its local hard drives)

Word Count: complexity

- Each server reads 1 GB of data from its local hard drive (it reads one chunk from HDFS)
 - Few seconds
- Each local list consists of at most 1,000 pairs (because the number of distinct words is 1,000)
 - Few MBs
- The maximum amount of data sent on the network is $100 \times$ size of local list (number of servers \times local list size)
 - Some MBs

Word Count: scalability

- We can define **scalability** along two dimensions
 - In terms of **data**:
 - Given twice the amount of data, the word count algorithm takes approximately no more than twice as long to run
 - Each server processes $2 \times$ data => $2 \times$ execution time to compute local list
 - In terms of **resources**
 - Given twice the number of servers, the word count algorithm takes approximately no more than half as long to run
 - Each server processes $\frac{1}{2} \times$ data => $\frac{1}{2} \times$ execution time to compute local list

Word Count: scalability

- The time needed to send local results to the node in charge of computing the final result and the computation of the final result are considered negligible in this running example
- Frequently, this assumption is not true
 - It depends
 - on the complexity of the problem
 - on the ability of the developer to limit the amount of data sent on the network

MapReduce-approach key ideas

- Scale “out”, not “up”
 - Increase the number of servers, avoiding to upgrade the resources (CPU, memory) of the current ones
- Move processing to data
 - The network has a limited bandwidth
- Process data sequentially, avoid random access
 - Seek operations are expensive
 - Big data applications usually read and analyze all input records/objects
 - Random access is useless

Data locality

- Traditional distributed systems (e.g., HPC) move data to computing nodes (servers)
 - This approach cannot be used to process TBs of data
 - The network bandwidth is limited
- Hadoop moves code to data
 - Code (few KB) is copied and executed on the servers where the chunks of data are stored
 - This approach is based on “**data locality**”

Hadoop and MapReduce

- Hadoop/MapReduce is designed for
 - Batch processing involving (mostly) full scans of the input data
 - Data-intensive applications
 - Read and process the whole Web (e.g., PageRank computation)
 - Read and process the whole Social Graph (e.g., LinkPrediction, a.k.a. “friend suggestion”)
 - Log analysis (e.g., Network traces, Smart-meter data, ..)

Hadoop and MapReduce

- Hadoop/MapReduce is not the panacea for all Big Data problems
- Hadoop/MapReduce **does not fit well**
 - Iterative problems
 - Recursive problems
 - Stream data processing
 - Real-time processing

just for off-line batch analysis

The MapReduce Programming Paradigm

MapReduce and Functional programming

- The MapReduce programming paradigm is based on the basic concepts of **Functional programming**
- MapReduce “implements” a subset of functional programming
 - The programming model appears quite limited and strict
 - Everything is based on two “functions” with predefined signatures
 - **Map and Reduce**

What can we do with MapReduce?

- Solving complex problems is difficult
- However, there are several important problems that can be adapted to MapReduce
 - Log analysis
 - PageRank computation
 - Social graph analysis
 - Sensor data analysis
 - Smart-city data analysis
 - Network capture analysis

Building blocks: Map and Reduce

- MapReduce is based on two main “building blocks”
 - **Map** and **Reduce** functions
- Map function
 - It is **applied over each element** of an input data set and **emits** a set of **(key, value) pairs**
- Reduce function
 - It is **applied over each set of (key, value) pairs** (emitted by the map function) **with the same key** and **emits** a set of **(key, value) pairs** → Final result

Word count running example

- Input
 - A textual file (i.e., a list of words)
- Problem
 - Count the number of times each distinct word appears in the file
- Output
 - A list of pairs <word, number of occurrences in the input file>

Word count running example

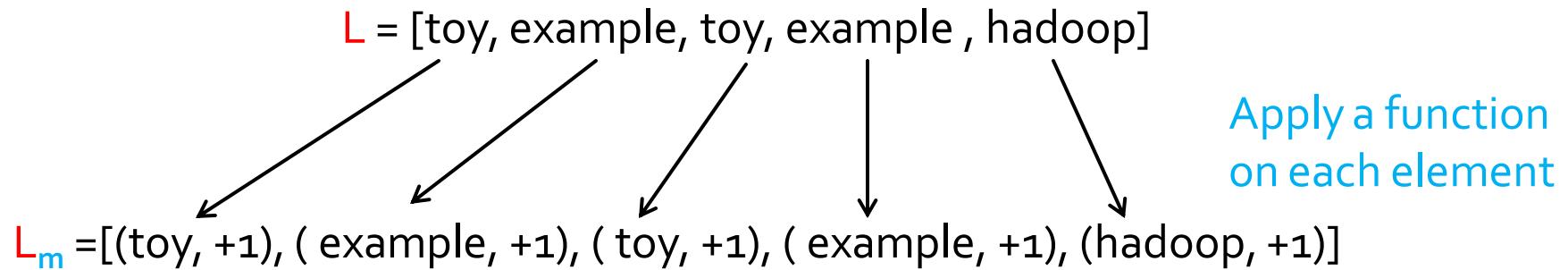
- The input textual file is considered as a list of words **L**

Word count running example

L = [toy, example, toy, example , hadoop]

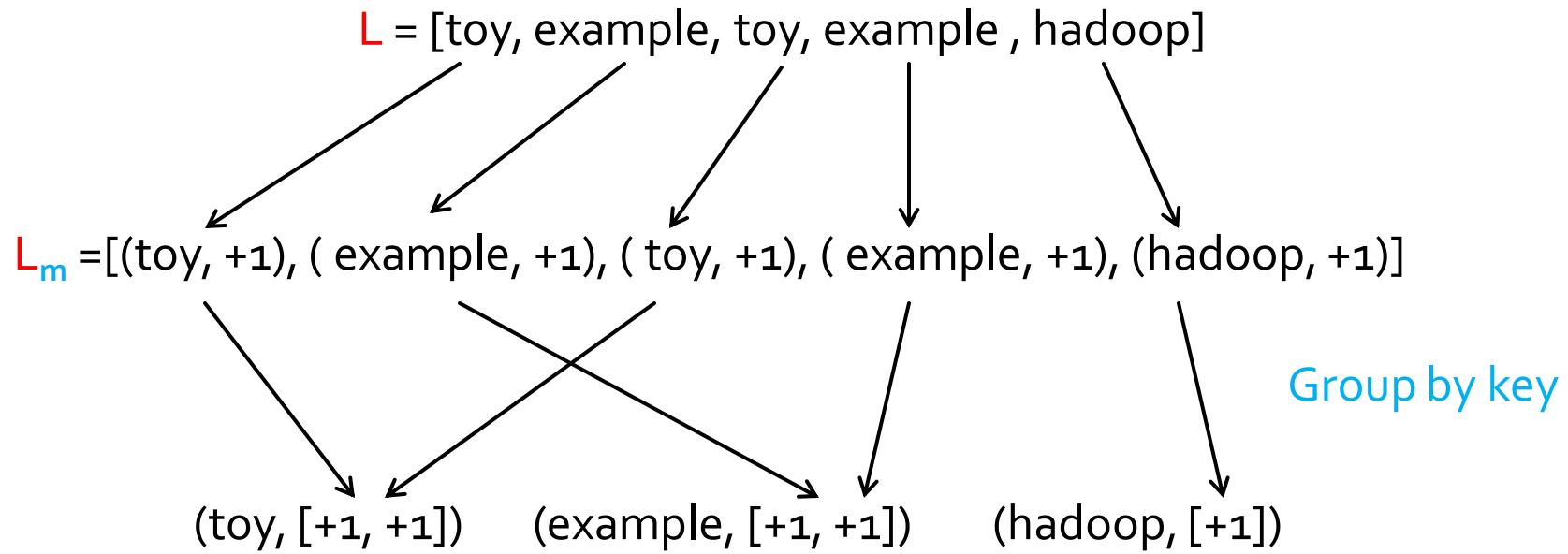
[...] denotes a list. (k, v) denotes a key-value pair.

Word count running example



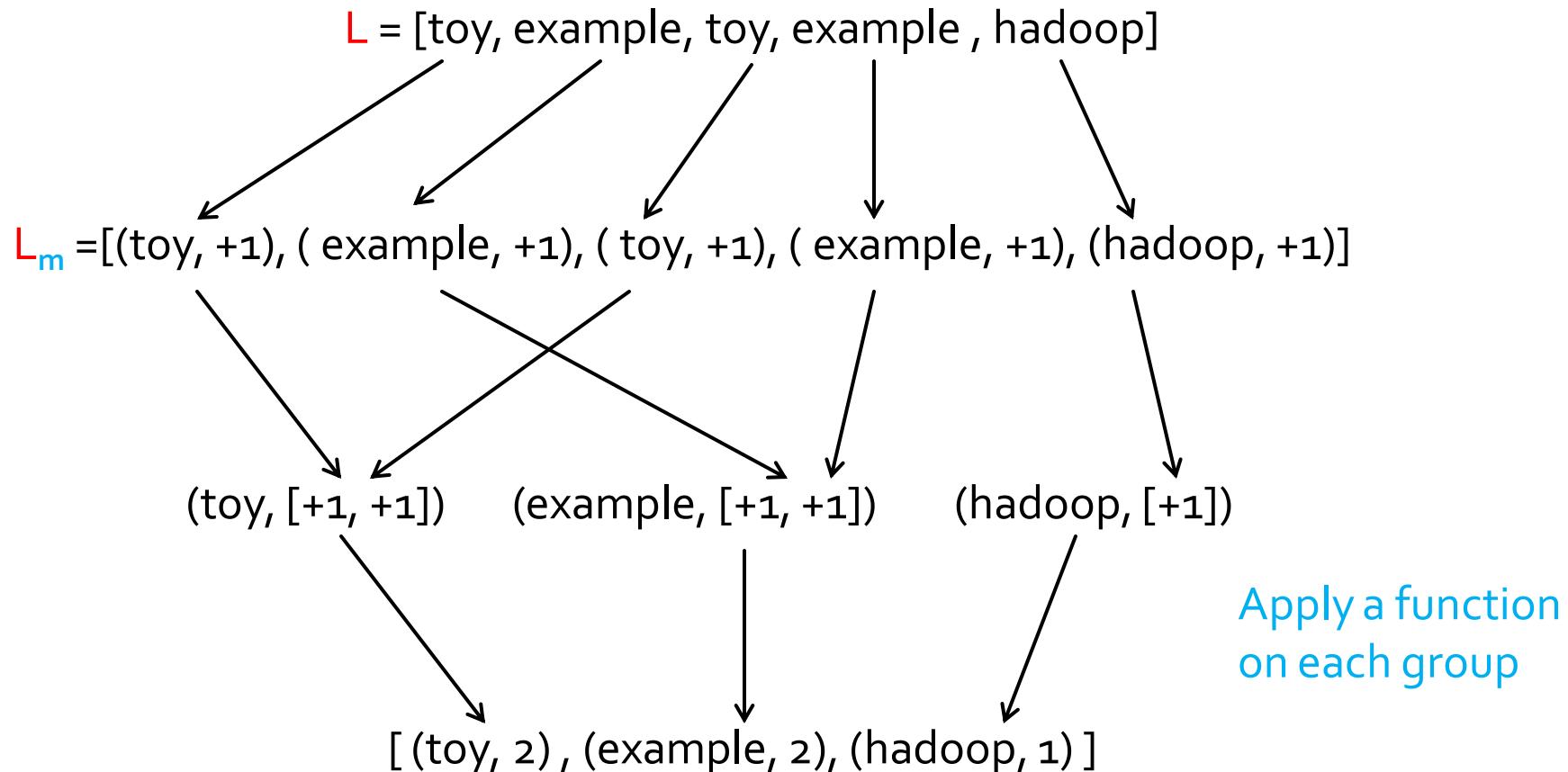
[...] denotes a set. (k, v) denotes a key-value pair.

Word count running example



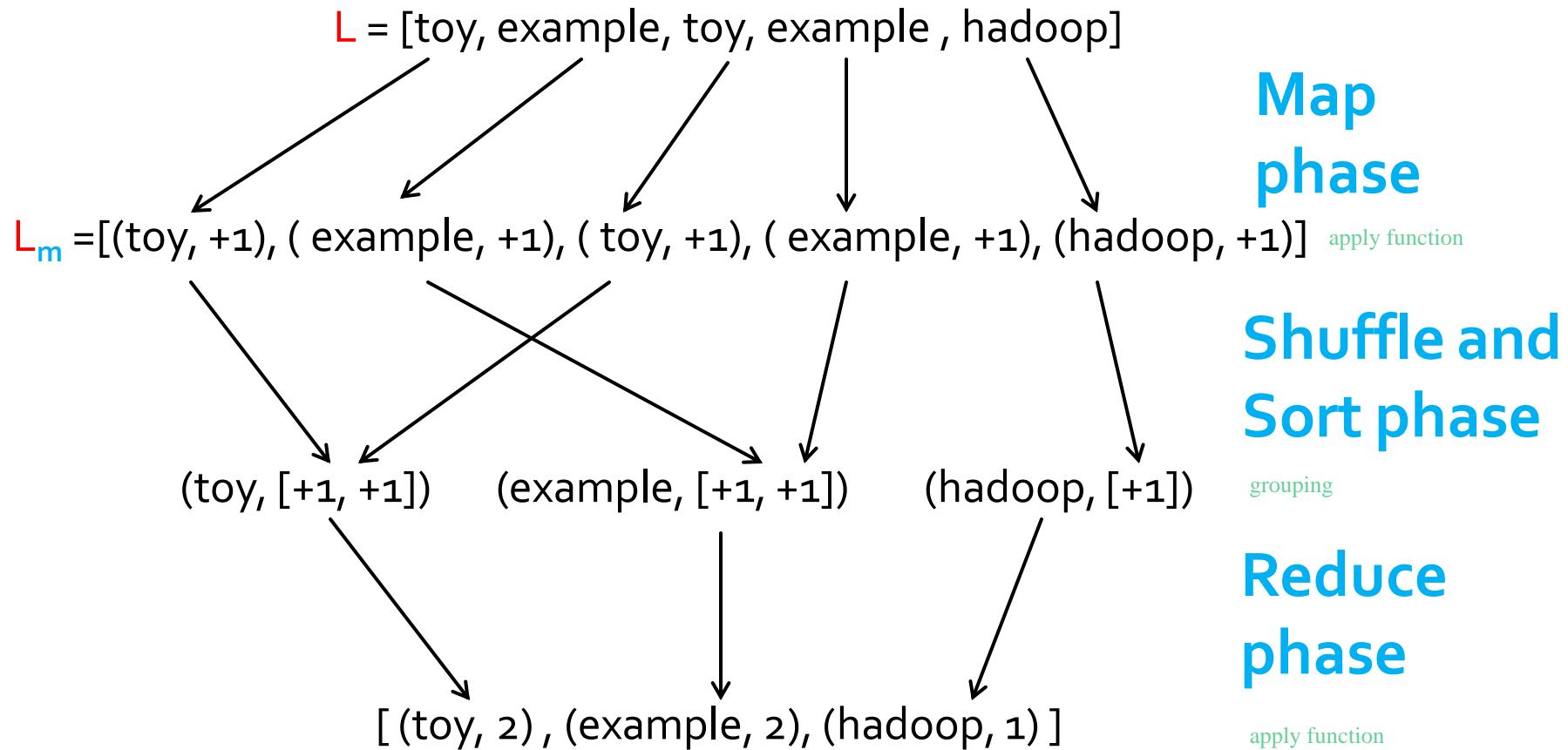
[...] denotes a list. (k, v) denotes a key-value pair.

Word count running example



[...] denotes a list. (k, v) denotes a key-value pair.

Word count running example



[...] denotes a list. (k, v) denotes a key-value pair.

Word count running example

- The input textual file is considered as a list of words **L**

Word count running example

- The input textual file is considered as a list of words L
- A key-value pair $(w, 1)$ is emitted for each word w in L
 - i.e., the **map** function is
$$m(w) = (w, 1)$$
 - A new list of (key, value) pairs L_m is generated

Word count running example

- The key-value pairs in L_m are aggregated by key (i.e., by word w in our example)
 - One group G_w is generated for each word w
 - Each group G_w is a key-list pair (w , [list of values]) where [list of values] contains all the values of the pairs associated with the word w
 - i.e., [list of values] is a list of [$1, 1, 1, \dots$] in our example
 - Given a group G_w , the number of ones [$1, 1, 1, \dots$] is equal to the occurrences of word w in the input file

Word count running example

- A key-value pair (w , sum G_w .[list of values]) is emitted for each group G_w
 - i.e., the **reduce** function is
$$r(G_w) = (w, \text{sum}(G_w.\text{[list of values]}))$$
- The list of emitted pairs is the result of the word count problem
 - One pair (word w , num. of occurrences) for each word in our running example

MapReduce: Map

- The **Map** phase can be viewed as a **transformation** over each element of a data set
 - This transformation is a function **m** defined by developers
 - **m** is invoked one time for each input element
 - Each invocation of **m** happens in **isolation**
 - The application of **m** to each element of a data set can be parallelized in a straightforward manner

MapReduce: Reduce

- The **Reduce** phase can be viewed as an **aggregate** operation
 - The aggregate function is a function **r** defined by developers
 - **r** is invoked one time for each distinct key and aggregates all the values associated with it
 - Also the reduce phase can be performed in parallel and in isolation
 - Each group of key-value pairs with the same key can be processed in isolation

MapReduce: Shuffle and Sort

- The shuffle and sort phase is always the same
 - i.e., group the output of the map phase by key
 - It does not need to be defined by developers
 - It is already provided by the Hadoop system

Data Structures

- Key-value pair is the basic data structure in MapReduce
 - Keys and values can be: integers, float, strings, ...
 - They can also be (almost) arbitrary data structures defined by the designer
- Both input and output of a MapReduce program are lists of key-value pairs
 - Note that also the input is a list of key-value pairs

Data Structures

- The design of MapReduce involves
 - Imposing the key-value structure on the input and output data sets
 - E.g., for a collection of Web pages, input keys may be URLs and values may be their HTML content

Formal definition of Map and Reduce functions

- The map and reduce functions are formally defined as follows:
 - $\text{map}: (k_1, v_1) \rightarrow [(k_2, v_2)]$
 - $\text{reduce}: (k_2, [v_2]) \rightarrow [(k_3, v_3)]$
- Since the input data set is a list of key-value pairs, the argument of the map function is a key-value pair

[...] denotes a list. (k, v) denotes a key-value pair

Formal definition of Map and Reduce functions

- Map function
 - map: $(k_1, v_1) \rightarrow [(k_2, v_2)]$
- The argument of the map function is a key-value pair
- Note that the map function emits a **list** of key-value pairs for each input record
 - The list can also be empty

[...] denotes a list. (k, v) denotes a key-value pair

Formal definition of Map and Reduce functions

- Reduce function
 - $\text{reduce}: (k_2, [v_2]) \rightarrow [(k_3, v_3)]$
- Note that the reduce function
 - Receives a list of values $[v_2]$ associated with a specific key k_2
 - Emits a list of key-value pairs

[...] denotes a list. (k, v) denotes a key-value pair

MapReduce Algorithms

- In many applications, the key part of the input data set is ignored
 - i.e., usually the map function does not consider the key of its key-value pair argument
 - E.g., word count problem
- Some specific applications exploit also the keys of the input data
 - E.g., keys can be used to uniquely identify records/objects it is like an index or id

Word Count using MapReduce: Pseudocode

Input file: a textual document with one word per line

The map function is invoked over each word of the input file

```
map(key, value):
    // key: offset of the word in the file
    // value: a word of the input document
    emit(value, 1)
```

```
reduce(key, values):
    // key: a word; values: a list of integers
    occurrences = 0
    for each c in values:
        occurrences = occurrences + c

    emit(key, occurrences)
```

HDFS: command line commands

HDFS

- The content of a HDFS file can be accessed by means of
 - Command line commands
 - A basic web interface provided by Apache Hadoop
 - The HDFS content can only be browsed and its files downloaded from HDFS to the local file system
 - Uploading functionalities are not available
 - Vendor-specific web interfaces providing a full set of functionalities (upload, download, rename, delete, ...)
 - E.g., the HUE web application of Cloudera

HDFS – user folder

- Each user of the Hadoop cluster has a personal folder in the HDFS file system
 - The default folder of a user is

*/user/**username***

HDFS – command line

- The **hdfs** command can be executed in a Linux shell to read/write/modify/delete the content of the distributed file system
- The parameters/arguments of hdfs command are used to specify the operation to execute

HDFS – command line

- List the content of a folder of the HDFS file system

hdfs dfs -ls *folder*

- Example

hdfs dfs -ls /user/garza

- shows the content (list of files and folders) of the /user/garza folder

HDFS – command line

- Example

```
hdfs dfs -ls .
```

- shows the content of the home of the current user
 - i.e., the content of /user/*current_username*
 - . = user home
- The mapping between the local linux user and the user of the cluster is based on
 - A Kerberos ticket if Kerberos is active
 - Otherwise the local linux user is considered

HDFS – command line

- Show the content of a file of the HDFS file system

hdfs dfs -cat *file*

- Example

hdfs dfs -cat /user/garza/document.txt

- Shows the content of the /user/garza/document.txt file stored in HDFS

HDFS – command line

- Copy a file from the local file system to the HDFS file system

hdfs dfs -put *local_file HDFS_path*

- Example

hdfs dfs -put /data/document.txt /user/garza/

- Copy the local file /data/document.txt in the folder /user/garza of HDFS

HDFS – command line

- Copy a file from the HDFS file system to the local file system
 - hdfs dfs -get *HDFS_path local_file*
- Example
 - hdfs dfs -get /user/garza/document.txt /data/
- Copy the HDFS file /user/garza/document.txt in the local file system folder /data/

HDFS – command line

- Delete a file from the HDFS file system
 `hdfs dfs -rm HDFS_path`
- Example
 `hdfs dfs -rm /user/garza/document.txt`
- Delete from HDFS the file
 `/user/garza/document.txt`

HDFS – command line

- There are many other linux-like commands
 - rmdir
 - du
 - tail
 - ...
- Useful link
 - <https://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html>

HDFS and Hadoop

Hadoop – command line

- The Hadoop programs are executed (submitted to the cluster) by using the hadoop command
 - It is a command line program
 - Hadoop is characterized by a set of parameters
 - E.g., the name of the jar file containing all the classes of the MapReduce application we want to execute
 - The name of the Driver class
 - The parameters/arguments of the MapReduce application

Hadoop – command line example

(1)

- The following command executes/submits a MapReduce application
hadoop jar *MyApplication.jar*
it.polito.bigdata.hadoop.DriverMyApplication 1
inputdatafolder/ outputdatafolder/
- It executes/submits the application contained in *MyApplication.jar*

Hadoop – command line example

(2)

- The Driver Class is
 - it.polito.bigdata.hadoop.DriverMyApplication
- The application has three arguments
 - Number of reducers (args[0])
 - Input data folder (args[1])
 - Output data folder (args[2])

How to write MapReduce programs in Hadoop

Hadoop implementation of MapReduce

MapReduce and Hadoop

- Designers/Developers focus on the definition of the Map and Reduce functions (i.e., **m** and **r**)
 - No need to manage the distributed execution of the map, shuffle and sort, and reduce phases
- The **Hadoop framework** coordinates the execution of the MapReduce program
 - Parallel execution of the map and reduce phases
 - Execution of the shuffle and sort phase
 - Scheduling of the subtasks
 - Synchronization

MapReduce programs

- The programming language is **Java**
- A Hadoop MapReduce program consists of three main parts
 - Driver
 - Mapper
 - Reducer
- Each part is “implemented” by means of a specific class

Terminology

- Driver class
 - The class containing the method/code that coordinates the configuration of the job and the “workflow” of the application
- Mapper class
 - A class “implementing” the map function
- Reducer class
 - A class “implementing” the reduce function
- Driver
 - Instance of the Driver class (i.e., an object)
- Mapper
 - Instance of the Mapper class (i.e., an object)
- Reducer
 - Instance of the Reducer class (i.e., an object)

Terminology

- (Hadoop) Job
 - Execution/run of a MapReduce code over a data set
- Task
 - Execution/run of a Mapper (Map task) or a Reducer (Reduce task) on a slice of data
 - Many tasks for each job
- Input split
 - Fixed-size piece of the input data
 - Usually each split has approximately the same size of a HDFS block/chunk

Driver

- The Driver
 - Is characterized by the main() method, which accepts arguments from the command line
 - i.e., it is the entry point of the application
 - Configures the job
 - Submits the job to the Hadoop Cluster
 - “Coordinates” the work flow of the application
 - Runs on the client machine
 - i.e., it does not run on the cluster

Mapper

- The Mapper
 - Is an instance of the Mapper class
 - “Implements” the map phase
 - Is characterized by the map(...) method
 - Processes the (key, value) pairs of the input file and emits (key, value) pairs
 - Is invoked one time for each input (key, value) pair
 - **Runs on the cluster**

Reducer

after shuffling and sort phase

■ The Reducer

- Is an instance of the Reduce class
- “Implements” the reduce phase
- Is characterized by the reduce(...) method
 - Processes (key, [list of values]) pairs and emits (key, value) pairs
 - Is invoked one time for each distinct key
- **Runs on the cluster**

Hadoop implementation of the MapReduce phases

- Input key-value pairs are read from the HDFS file system
- The map method of the Mapper
 - Is invoked over each input key-value pair
 - Emits a set of intermediate key-value pairs that are stored in the local file system of the computing server
(they are not stored in HDFS)
- Intermediate results
 - Are aggregated by means of a shuffle and sort procedure
 - A set of (key, [list of values]) pairs is generated
 - One (key, [list of values]) for each distinct key

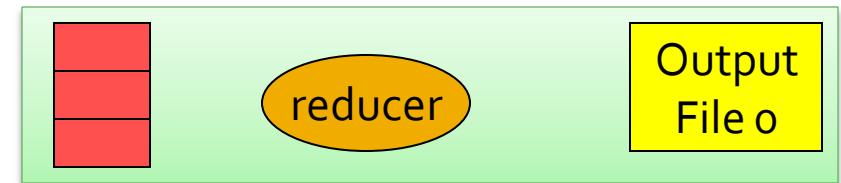
Hadoop implementation of the MapReduce phases

- The reduce method of the Reducer
 - Is applied over each (key, [list of values]) pair
 - Emits a set of key-value pairs that are stored in HDFS (the final result of the MapReduce application)
- Intermediate key-value pairs are transient:
 - They are not stored on the distributed files system
 - They are stored locally to the node producing or processing them

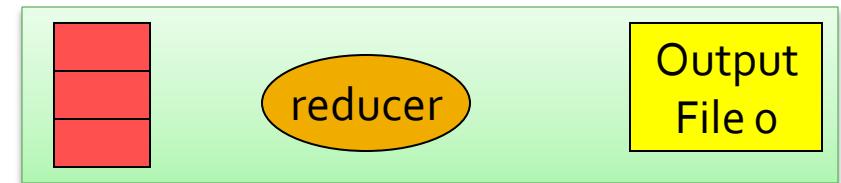
Hadoop implementation of the MapReduce phases

- In order to **parallelize** the work/the job, Hadoop executes a set of tasks in parallel
 - It instantiates one Mapper (Task) for each input split
 - And a user-specified number of Reducers
 - Each reducer is associated with a set of keys
 - It receives and processes all the key-value pairs associated with its set of keys
 - **Mappers and Reducers are executed on the nodes/servers of the clusters**

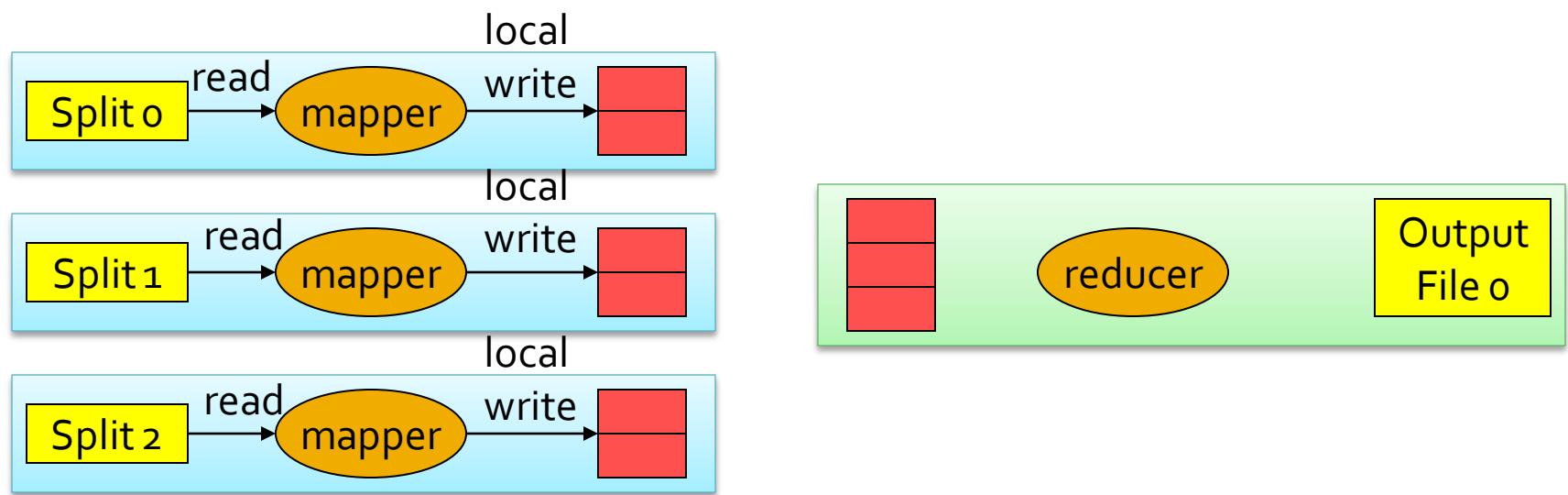
MapReduce data flow with a single reducer



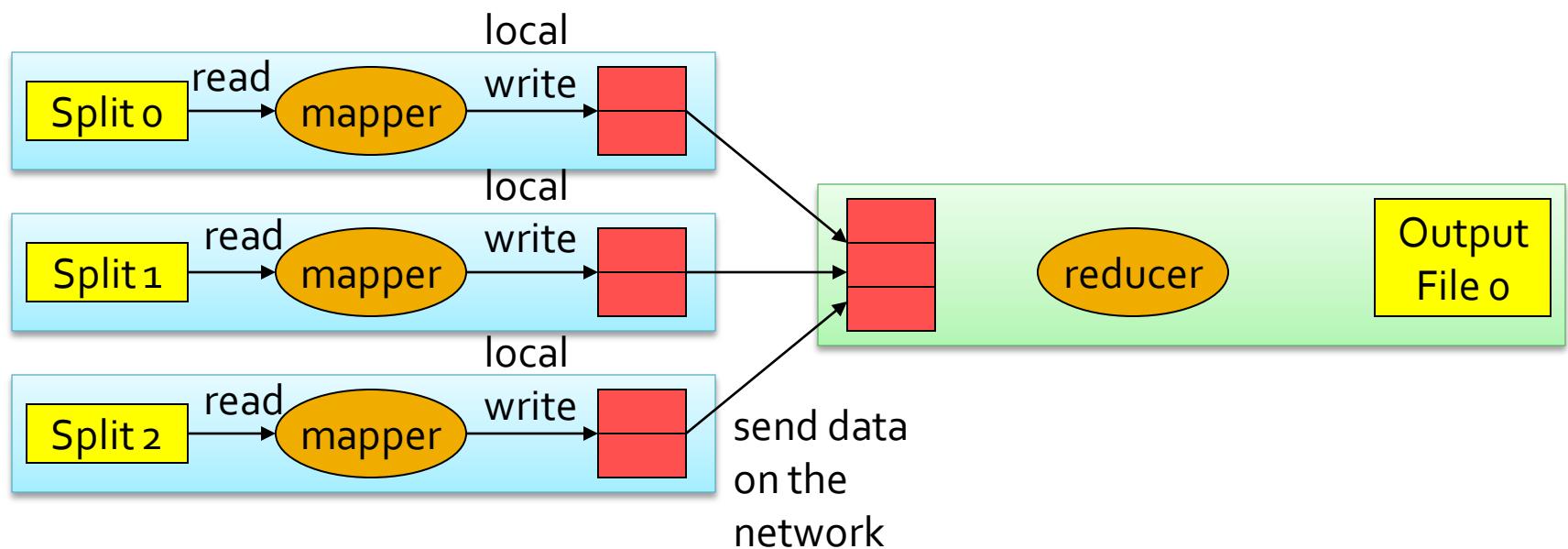
MapReduce data flow with a single reducer



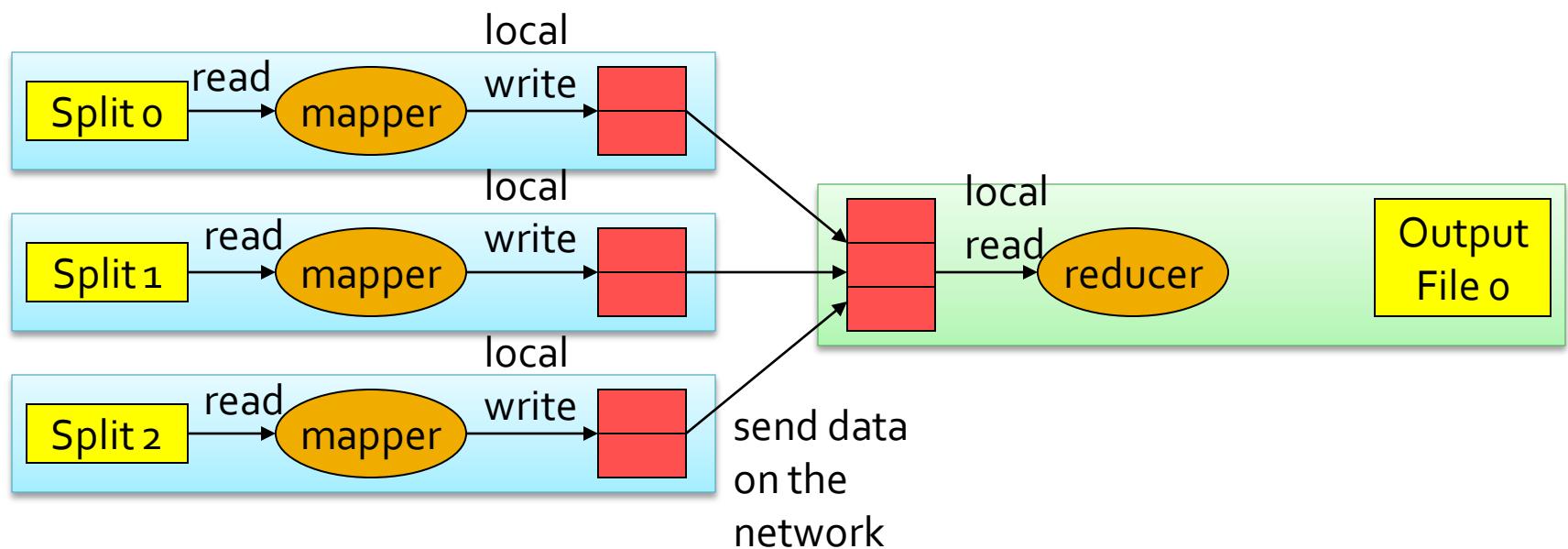
MapReduce data flow with a single reducer



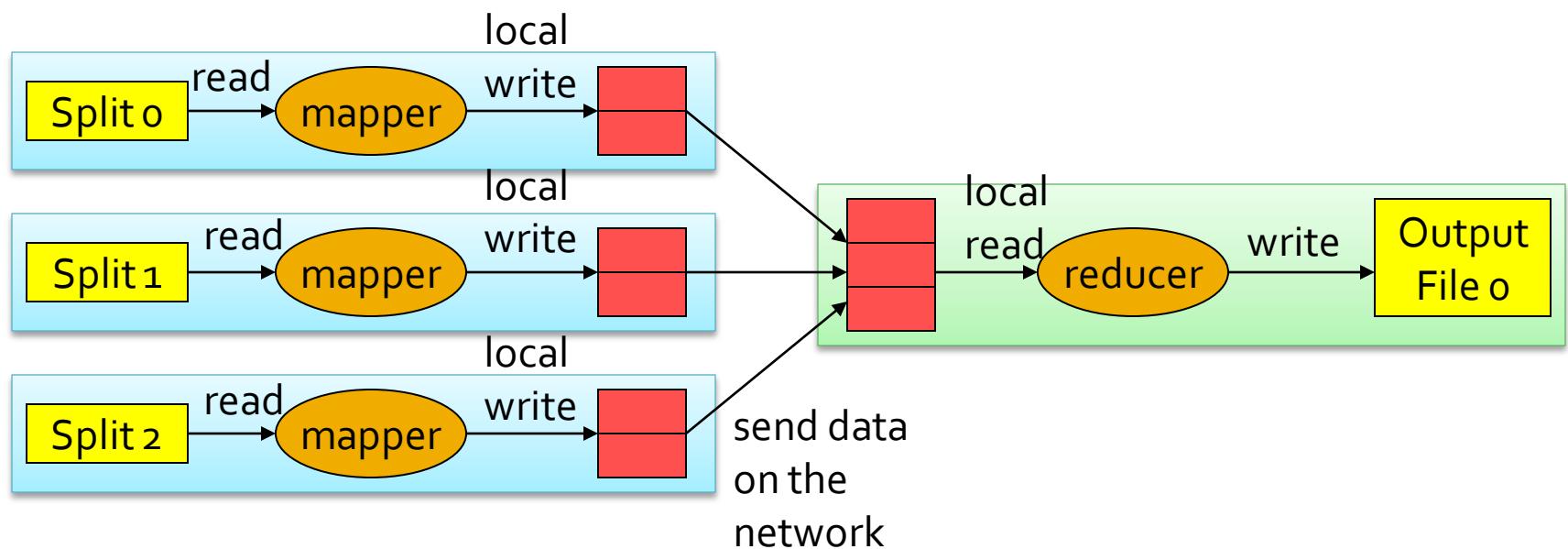
MapReduce data flow with a single reducer



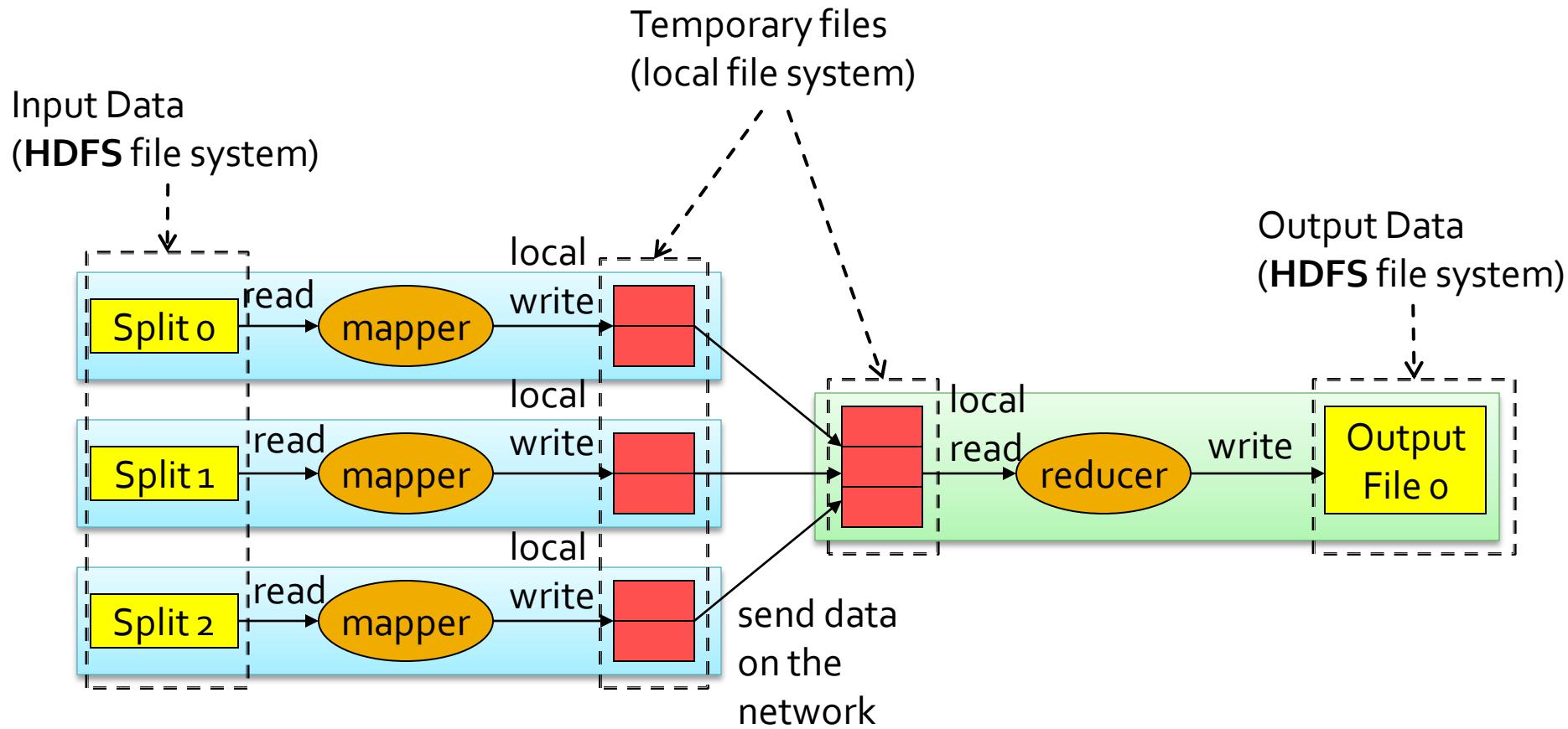
MapReduce data flow with a single reducer



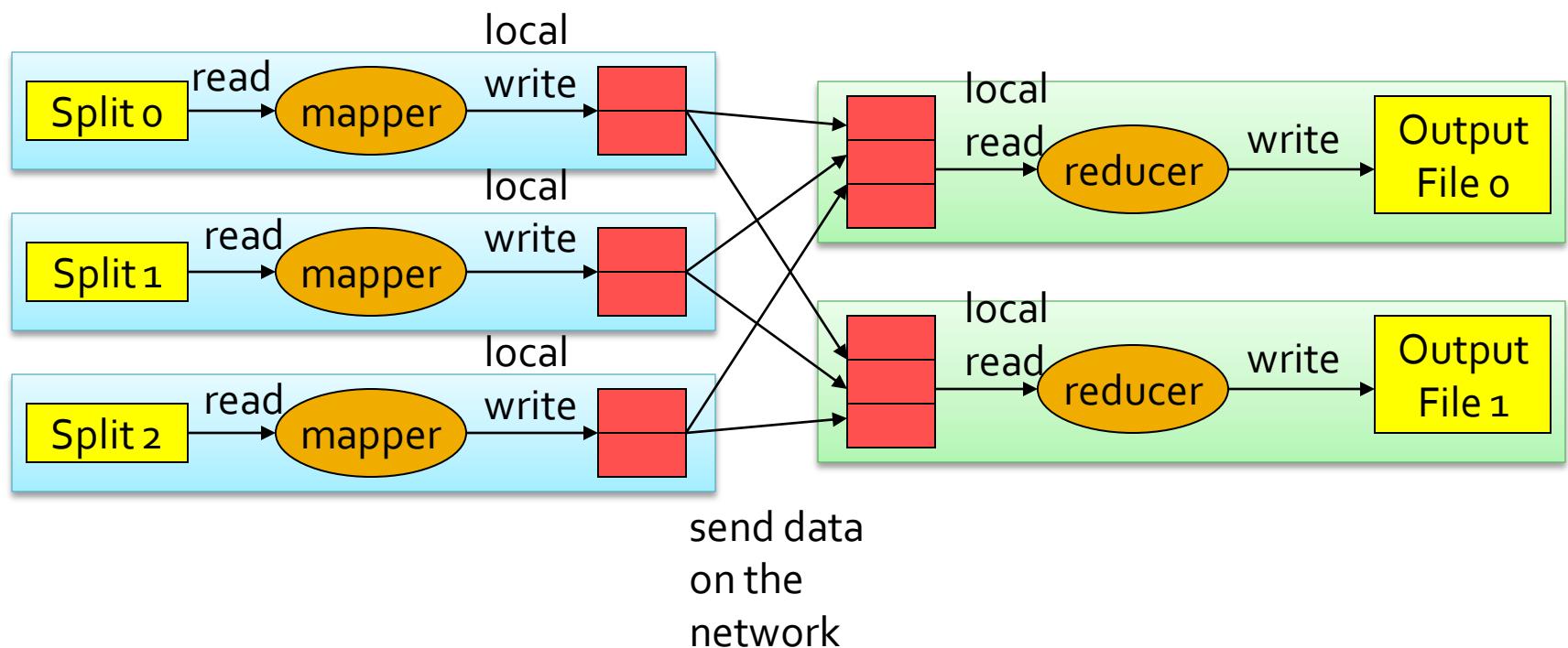
MapReduce data flow with a single reducer



MapReduce data flow with a single reducer

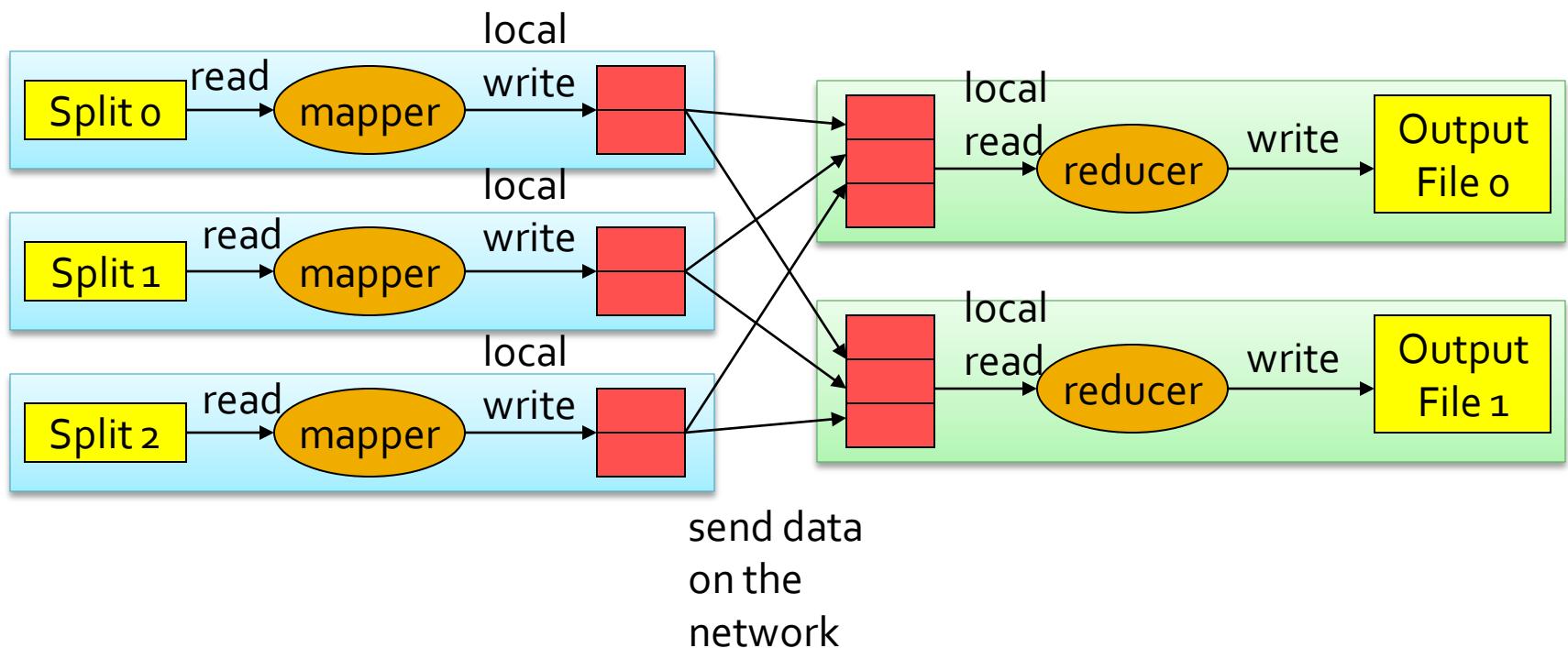


MapReduce data flow with multiple reducers



MapReduce data flow with multiple reducers

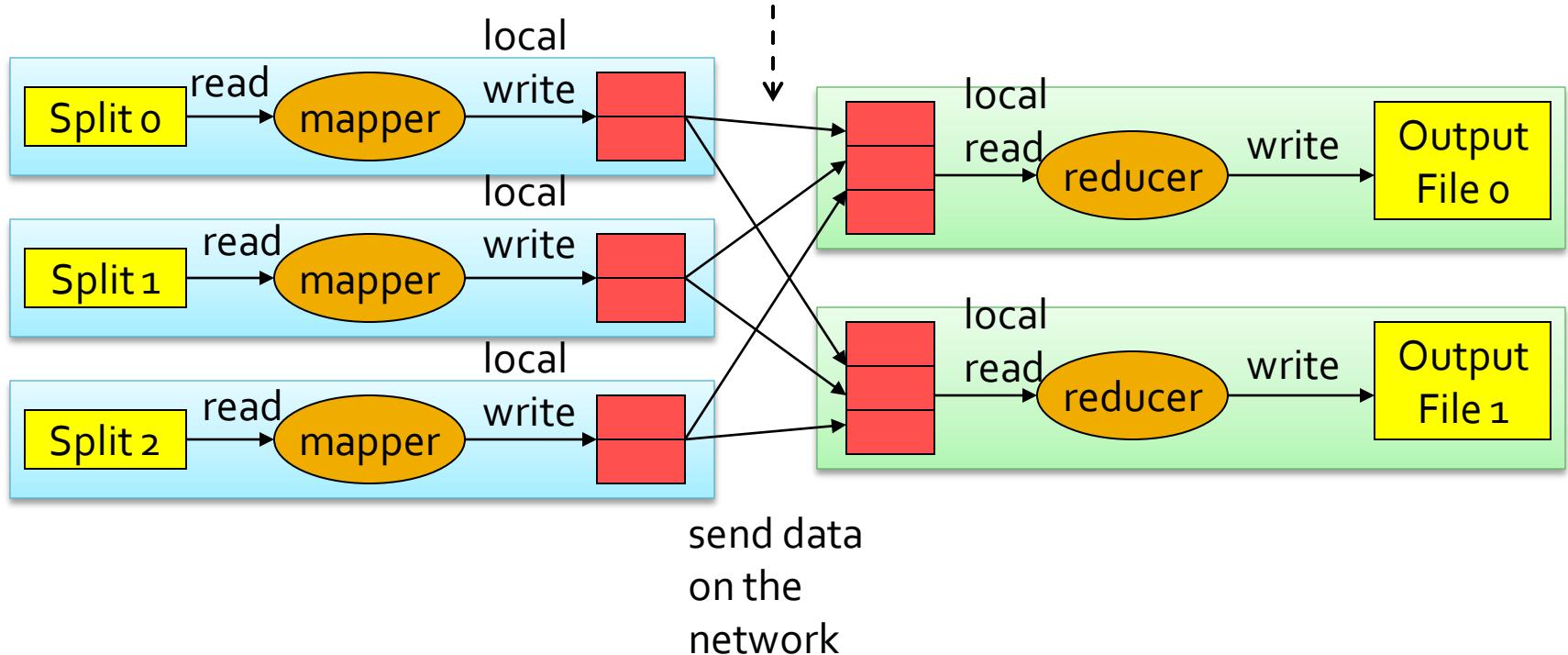
Each key is assigned
to/managed by one reducer



MapReduce data flow with multiple reducers

Each key is assigned
to/managed by one reducer

Potentially, all mappers send
data (a set of (key,value) pairs)
to all reducers



MapReduce programs - Driver

- The **Driver** class **extends** the `org.apache.hadoop.conf.Configured` **class** and **implements** the `org.apache.hadoop.util.Tool` **interface**
 - You can write a Driver class that does not extend Configured and does not implement Tool
 - However, you need to manage some low level details related to some command line parameters in that case
- The designer/developer implements the **main(...)** and **run(...)** methods

MapReduce programs - Driver

- The `run(...)` method
 - Configures the job
 - Name of the Job
 - Job Input format
 - Job Output format
 - Mapper class
 - Name of the class
 - Type of its input (key, value) pairs
 - Type of its output (key, value) pairs

MapReduce programs - Driver

- Reducer class
 - Name of the class
 - Type of its input (key, value) pairs
 - Type of its output (key, value) pairs
- Number of reducers

MapReduce programs - Mapper

- The Mapper class **extends** the org.apache.hadoop.mapreduce.Mapper **class**
 - The org.apache.hadoop.mapreduce.Mapper class
 - Is a generic type/generic class
 - **With four type parameters:** input key type, input value type, output key type, output value type
- The designer/developer implements the **map(...)** method
 - That is automatically called by the framework for each (key, value) pair of the input file

MapReduce programs - Mapper

- The **map**(...) method
 - Processes its input (key, value) pairs by using standard Java code
 - Emits (key, value) pairs by using the context.write(key, value) method

MapReduce programs - Reducer

- The Reducer class **extends** the `org.apache.hadoop.mapreduce.Reducer` **class**
 - The `org.apache.hadoop.mapreduce.Reducer` class
 - Is a generic type/generic class
 - **With four type parameters**: input key type, input value type, output key type, output value type
- The designer/developer implements the **reduce(...)** method
 - That is automatically called by the framework for each (key, [list of values]) pair obtained by aggregating the output of the mapper(s)

MapReduce programs - Reducer

- The **reduce**(...) method
 - Processes its input (key, [list of values]) pairs by using standard Java code
 - Emits (key, value) pairs by using the context.write(key, value) method

MapReduce Data Types

- Hadoop has its own basic data types
 - Optimized for network serialization
 - org.apache.hadoop.io.Text: like Java String
 - org.apache.hadoop.io.IntWritable: like Java Integer
 - org.apache.hadoop.io.LongWritable: like Java Long
 - org.apache.hadoop.io.FloatWritable : like Java Float
 - Etc

MapReduce Data Types

- The basic Hadoop data types implement the org.apache.hadoop.io.Writable and org.apache.hadoop.io.WritableComparable interfaces
- All classes (data types) used to represent keys are instances of WritableComparable
 - Keys must be “comparable” for supporting the sort and shuffle phase
- All classes (data types) used to represent values are instances of Writable
 - Usually, they are also instances of WritableComparable even if it is not indispensable

MapReduce Data Types

- Developers can define new data types by implementing the `org.apache.hadoop.io.Writable` and/or `org.apache.hadoop.io.WritableComparable` interfaces
 - It is useful for managing complex data types

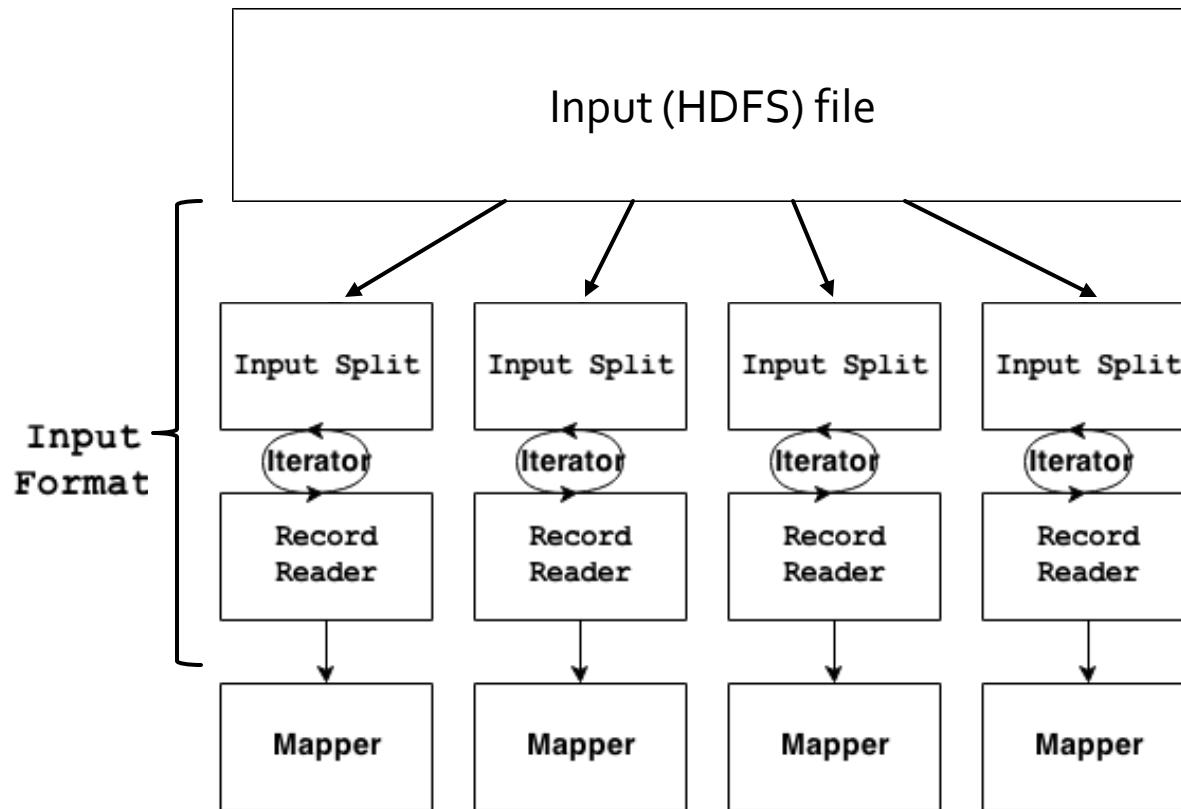
InputFormat

- The `input` of the `MapReduce program` is an `HDFS file` (or an HDFS folder)
- While the `input` of the `Mapper` is a `set` of (key, value) pairs
- The `classes` extending the `org.apache.hadoop.mapreduce.InputFormat abstract class` are used to `read` the `input data` and “`logically transform`” the input `HDFS` file in a `set` of (key, value) pairs

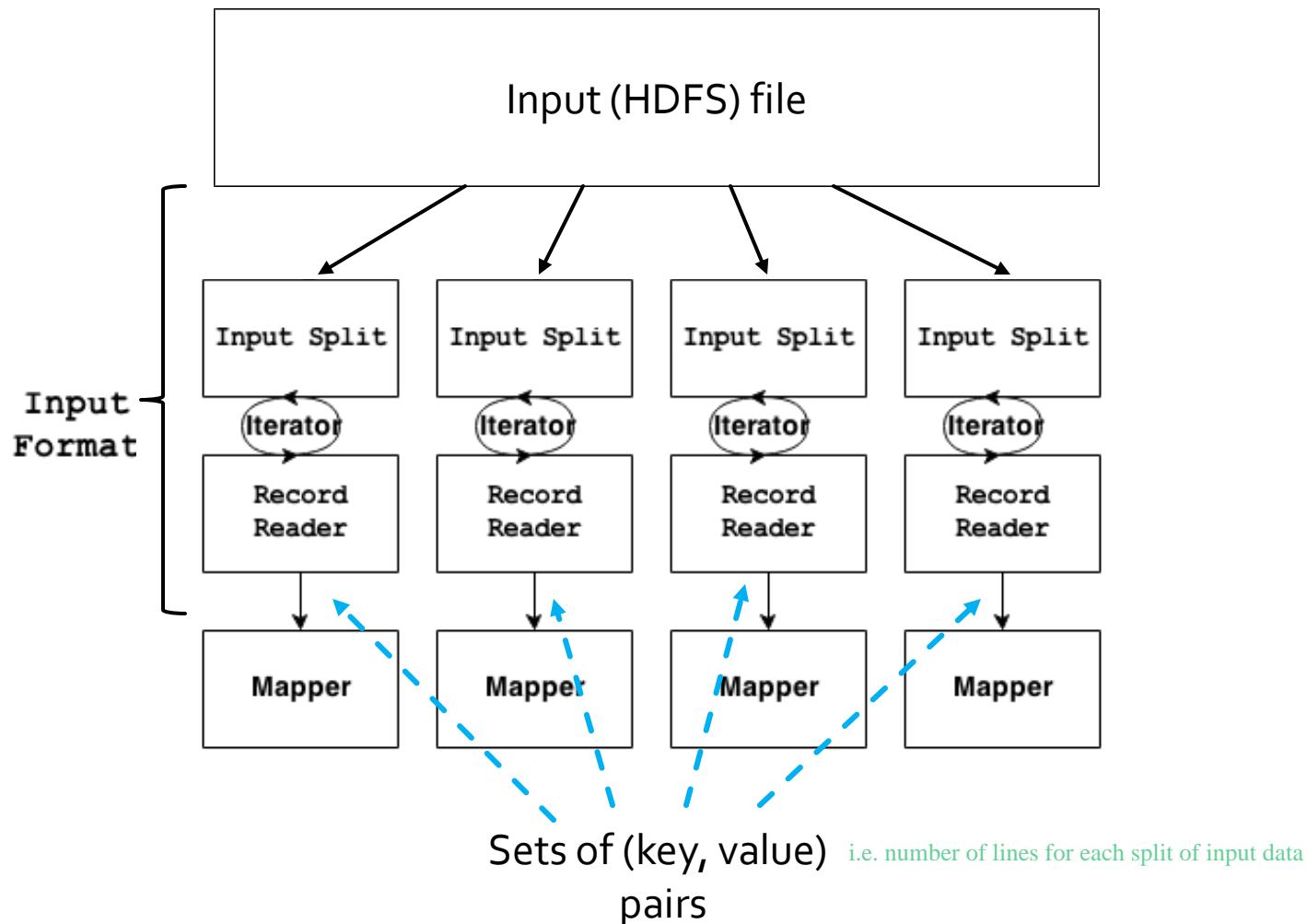
InputFormat

- InputFormat “describes” the input-format specification for a MapReduce application and processes the input file(s)
- The **InputFormat class** is **used** to
 - Read input data and validate the compliance of the input file with the expected input-format
 - Split the input file(s) into logical Input Splits
 - Each input split is then assigned to an individual Mapper
 - Provide the **RecordReader implementation** to be **used** to **divide the logical input split** in a **set** of (key,value) **pairs** (also called records) **for the mapper**

Getting Data to the Mapper



Getting Data to the Mapper



Reading Data

- InputFormat identifies partitions of the data that form an input split
 - Each input split is a (reference to a) part of the input file processed by a single mapper
 - Each split is divided into records, and the mapper processes one record (i.e., a (key,value) pair) at a time
- by map method

InputFormat

- A set of predefined classes extending the InputFormat abstract class are available for standard input file formats
 - **TextInputFormat**
 - An InputFormat for plain text files
 - **KeyValueTextInputFormat**
 - Another InputFormat for plain text files
 - **SequenceFileInputFormat**
 - An InputFormat for sequential/binary files
 -

TextInputFormat

- **TextInputFormat**
 - An InputFormat for plain text files
 - Files are broken into lines
 - Either linefeed or carriage-return are used to signal end of line
 - One pair (key, value) is emitted for each line of the file
 - Key is the position (offset) of the line in the file
 - Value is the content of the line

TextInputFormat example

Input HDFS file

Toy example file for Hadoop.\n

Hadoop running example.\n

TextInputFormat is used to split data.\n



(key, value) pairs generated by using TextInputFormat

(0, "Toy example file for Hadoop.")

(31, "Hadoop running example.")

(56, "TextInputFormat is used to split data.")

KeyValueTextInputFormat

- **KeyValueTextInputFormat**
 - An InputFormat for plain text files
 - Each line of the file must have the format
key<separator>value
 - The default separator is tab (\t)
 - Files are broken into lines
 - Either linefeed or carriage-return are used to signal end of line
 - Each line is split into key and value parts by considering the separator symbol/character
 - One pair (key, value) is emitted for each line of the file
 - Key is the text preceding the separator
 - Value is the text following the separator

KeyValueTextInputFormat

Input HDFS file

```
10125\tMister John\n
10236\tMiss Jenny\n
1\tMister Donald Duck\n
```



(key, value) pairs generated by using KeyValueTextInputFormat

(10125, "Mister John") both of them are string

(10236, "Miss Jenny")

(1, "Mister Donald Duck")

OutputFormat

- The classes extending the `org.apache.hadoop.mapreduce.OutputForm` abstract class are used to write the output of the MapReduce program in HDFS

OutputFormat

- A set of predefined classes extending the OutputFormat abstract class are available for standard output file formats
 - **TextOutputFormat**
 - An OutputFormat for plain text files
 - **SequenceFileOutputFormat**
 - An OutputFormat for sequential/binary files
 -

TextOutputFormat

- **TextOutputFormat**
 - An OutputFormat for plain text files
 - For each output (key, value) pair
TextOutputFormat writes one line in the output file
 - The format of each output line is
key\tvalue\n

Structure of a MapReduce program in Hadoop

Basic structure of a MapReduce program - Driver (1)

```
/* Set package */  
package it.polito.bigdata.hadoop.mypackage;  
  
/* Import libraries */  
import java.io.IOException;  
  
import org.apache.hadoop.mapreduce.Job;  
import org.apache.hadoop.util.Tool;  
import org.apache.hadoop.util.ToolRunner;  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.conf.Configured;  
import org.apache.hadoop.io.*;  
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;  
.....
```

Basic structure of a MapReduce program - Driver (2)

```
/* Driver class */
public class MapReduceAppDriver extends Configured implements
Tool {
    @Override
    public int run(String[] args) throws Exception {
        /* variables */
        int exitCode;
        ....
        // Parse parameters
        numberOfReducers = Integer.parseInt(args[0]);
        inputPath = new Path(args[1]);
        outputDir = new Path(args[2]);
        ....
```

Basic structure of a MapReduce program - Driver (3)

```
// Define and configure a new job
Configuration conf = this.getConf();
Job job = Job.getInstance(conf);

// Assign a name to the job
job.setJobName("My First MapReduce program");
```

Basic structure of a MapReduce program - Driver (4)

```
// Set path of the input file/folder (if it is a folder, the job reads all  
// the files in the specified folder) for this job  
FileInputFormat.addInputPath(job, inputPath);  
  
// Set path of the output folder for this job  
FileOutputFormat.setOutputPath(job, outputDir);  
  
// Set input format  
// TextInputFormat = textual files  
job.setInputFormatClass(TextInputFormat.class);  
  
// Set job output format  
job.setOutputFormatClass(TextOutputFormat.class);
```

Basic structure of a MapReduce program - Driver (5)

```
// Specify the class of the Driver for this job  
job.setJarByClass(MapReduceAppDriver.class);
```

```
// Set mapper class  
job.setMapperClass(MyMapperClass.class);
```

```
// Set map output key and value classes  
job.setMapOutputKeyClass(output key type.class);  
job.setMapOutputValueClass(output value type.class);
```

Basic structure of a MapReduce program - Driver (6)

```
// Set reduce class  
job.setReducerClass(MyReducerClass.class);
```

```
// Set reduce output key and value classes  
job.setOutputKeyClass(output key type.class);  
job.setOutputValueClass(output value type.class);
```

```
// Set number of reducers  
job.setNumReduceTasks(numberOfReducers);
```

Basic structure of a MapReduce program - Driver (7)

```
// Execute the job and wait for completion  
if (job.waitForCompletion(true)==true)  
    exitCode=0;  
else  
    exitCode=1;  
return exitCode;  
} // End of the run method
```

Basic structure of a MapReduce program - Driver (8)

```
/* main method of the driver class */
public static void main(String args[]) throws Exception {
    /* Exploit the ToolRunner class to "configure" and run the
     Hadoop application */

    int res = ToolRunner.run(new Configuration(),
                           new MapReduceAppDriver(), args);

    System.exit(res);
} // End of the main method

} // End of public class MapReduceAppDriver
```

Basic structure of a MapReduce program - Mapper (1)

```
/* Set package */  
package it.polito.bigdata.hadoop.mypackage;  
  
/* Import libraries */  
import java.io.IOException;  
  
import org.apache.hadoop.mapreduce.Mapper;  
import org.apache.hadoop.io.*;  
....
```

Basic structure of a MapReduce program - Mapper (2)

```
/* Mapper Class */  
class myMapperClass extends Mapper<  
    MapperInputKeyType, // Input key type (must be  
    consistent with the InputFormat class specified in the Driver)  
    MapperInputValueType, // Input value type (must be  
    consistent with the InputFormat class specified in the Driver)  
    MapperOutputKeyType, // Output key type  
    MapperOutputValueType> // Output value type  
{
```

Basic structure of a MapReduce program - Mapper (3)

```
/* Implementation of the map method */
protected void map(
    MapperInputKeyType key,          // Input key
    MapperInputValueType value,      // Input value
    Context context) throws IOException, InterruptedException {

    /* Process the input (key, value) pair and
       emit a set of (key,value) pairs.
       context.write(..) is used to emit (key, value) pairs
       context.write(new outputkey, new outputvalue); */

} // End of the map method

} // End of class myMapperClass
```

Basic structure of a MapReduce program - Reducer (1)

```
/* Set package */  
package it.polito.bigdata.hadoop.mypackage;  
  
/* Import libraries */  
import java.io.IOException;  
  
import org.apache.hadoop.mapreduce.Reducer;  
Import org.apache.hadoop.io.*;  
.....
```

Basic structure of a MapReduce program - Reducer (2)

```
/* Reducer Class */
class myReducerClass extends Reducer<
    ReducerInputKeyType,    // Input key type (must be
consistent with the OutputKeyType of the Mapper)
    ReducerInputValueType,   // Input value type (must be
consistent with the OutputValueType of the Mapper)
    ReducerOutputKeyType,   // Output key type (must be
consistent with the OutputFormat class specified in the Driver)
    ReducerOutputValueType> // Output value type (must
be consistent with the OutputFormat class specified in the Driver)
{
```

Basic structure of a MapReduce program - Reducer (3)

```
/* Implementation of the reduce method */
protected void reduce(
    ReducerInputKeyType key,                      // Input key
    Iterable<ReducerInputValueType> values, // Input values (list of
values)
    Context context) throws IOException, InterruptedException {

    /* Process the input (key, [list of values]) pair and
       emit a set of (key,value) pairs.
       context.write(..) is used to emit (key, value) pairs
       context.write(new outputkey, new outputvalue);
    */
}

// End of class myReducerClass
```

MapReduce programs in Hadoop: Word Count Example

Word Count Example

- Word count problem
 - Input: (unstructured) textual file
 - Each line of the input file can contains a set of words
 - Output: number of occurrences of each word appearing in the input file
- Parameters/arguments of the application:
 - args[0]: number of instances of the reducer
 - args[1]: path of the input file
 - args[2]: path of the output folder

Word Count Example

- Input file

Toy example
file for Hadoop.
Hadoop running
example.

- Output pairs
 - (toy, 1)
 - (example, 2)
 - (file, 1)
 - (for, 1)
 - (hadoop, 2)
 - (running, 1)

Word Count Example - Driver (1)

```
/* Set package */  
package it.polito.bigdata.hadoop.wordcount;  
  
/* Import libraries */  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.conf.Configured;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Job;  
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;  
import org.apache.hadoop.util.Tool;  
import org.apache.hadoop.util.ToolRunner;
```

Word Count Example - Driver (2)

```
/* Driver class */
public class WordCount extends Configured implements Tool {
    @Override
    public int run(String[] args) throws Exception {
        Path inputPath;
        Path outputDir;
        int numberOfReducers;
        int exitCode;

        // Parse input parameters
        numberOfReducers = Integer.parseInt(args[0]);
        inputPath = new Path(args[1]);
        outputDir = new Path(args[2]);
```

Word Count Example - Driver (3)

```
// Define and configure a new job
Configuration conf = this.getConf();
Job job = Job.getInstance(conf);

// Assign a name to the job
job.setJobName("WordCounter");
```

Word Count Example - Driver (4)

```
// Set path of the input file/folder (if it is a folder, the job reads all the files  
in the specified folder) for this job
```

```
FileInputFormat.addInputPath(job, inputPath);
```

```
// Set path of the output folder for this job
```

```
FileOutputFormat.setOutputPath(job, outputDir);
```

```
// Set input format
```

```
// TextInputFormat = textual files
```

```
job.setInputFormatClass(TextInputFormat.class);
```

```
// Set job output format
```

```
job.setOutputFormatClass(TextOutputFormat.class);
```

Word Count Example - Driver (5)

```
// Specify the class of the Driver for this job  
job.setJarByClass(WordCount.class);  
  
// Set mapper class  
job.setMapperClass(WordCountMapper.class);  
  
// Set map output key and value classes  
job.setMapOutputKeyClass(Text.class);  
job.setMapOutputValueClass(IntWritable.class);
```

Word Count Example - Driver (6)

```
// Set reduce class  
job.setReducerClass(WordCountReducer.class);  
  
// Set reduce output key and value classes  
job.setOutputKeyClass(Text.class);  
job.setOutputValueClass(IntWritable.class);  
  
// Set number of reducers  
job.setNumReduceTasks(numberOfReducers);
```

Word Count Example - Driver (7)

```
// Execute the job and wait for completion
if (job.waitForCompletion(true)==true)
    exitCode=0;
else
    exitCode=1;
return exitCode;
} // End of the run method
```

Word Count Example - Driver (8)

```
/* main method of the driver class */
public static void main(String args[]) throws Exception {
    /* Exploit the ToolRunner class to "configure" and run the
     Hadoop application */

    int res = ToolRunner.run(new Configuration(),
                           new WordCount(), args);

    System.exit(res);
} // End of the main method

} // End of public class WordCount
```

Word Count Example - Mapper (1)

```
/* Set package */  
package it.polito.bigdata.hadoop.wordcount;  
  
/* Import libraries */  
import java.io.IOException;  
  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.LongWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Mapper;
```

Word Count Example - Mapper (2)

```
/* Mapper Class */
class WordCountMapper extends Mapper<
    LongWritable, // Input key type
    Text,         // Input value type
    Text,         // Output key type
    IntWritable> // Output value type

{
    /* Implementation of the map method */
    protected void map(
        LongWritable key, // Input key type
        Text value,       // Input value type
        Context context) throws IOException, InterruptedException {

```

Word Count Example - Mapper (3)

```
// Split each sentence in words. Use whitespace(s) as delimiter
// The split method returns an array of strings
String[] words = value.toString().split("\\s+");

// Iterate over the set of words
for(String word : words) {
    // Transform word case
    String cleanedWord = word.toLowerCase();

    // emit one pair (word, 1) for each input word
    context.write(new Text(cleanedWord), new IntWritable(1));
}

} // End map method

} // End of class WordCountMapper
```

Word Count Example - Reducer (1)

```
/* Set package */  
package it.polito.bigdata.hadoop.wordcount;  
  
/* Import libraries */  
import java.io.IOException;  
  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Reducer;
```

Word Count Example - Reducer (2)

```
/* Reducer Class */
class WordCountReducer extends Reducer<
    Text,           // Input key type
    IntWritable,    // Input value type
    Text,           // Output key type
    IntWritable>   // Output value type
{
    /* Implementation of the reduce method */
    protected void reduce(
        Text key, // Input key type
        Iterable<IntWritable> values, // Input value type
        Context context) throws IOException, InterruptedException {
```

Word Count Example - Reducer (3)

```
/* Implementation of the reduce method */
protected void reduce(
    Text key, // Input key type
    Iterable<IntWritable> values, // Input value type
    Context context) throws IOException, InterruptedException {
    int occurrences = 0;

    // Iterate over the set of values and sum them
    for (IntWritable value : values) {
        occurrences = occurrences + value.get();
    }
    // Emit the total number of occurrences of the current word
    context.write(key, new IntWritable(occurrences));
} // End reduce method
} // End of class WordCountReducer
```

Combiner

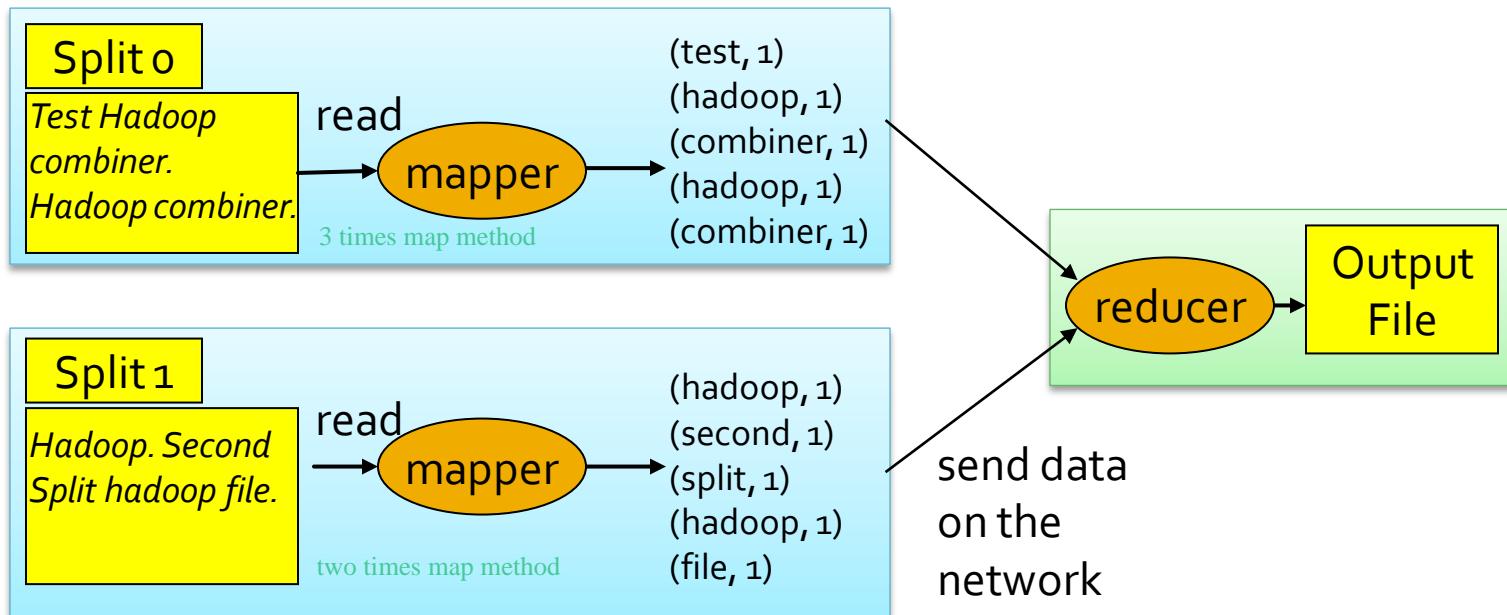
Combiner

- “Standard” MapReduce applications
 - The (key,value) pairs emitted by the Mappers are sent to the Reducers through the network
- Some “pre-aggregations” could be performed to limit the amount of network data by using Combiners (also called “mini-reducers”)

Combiner – Word count example

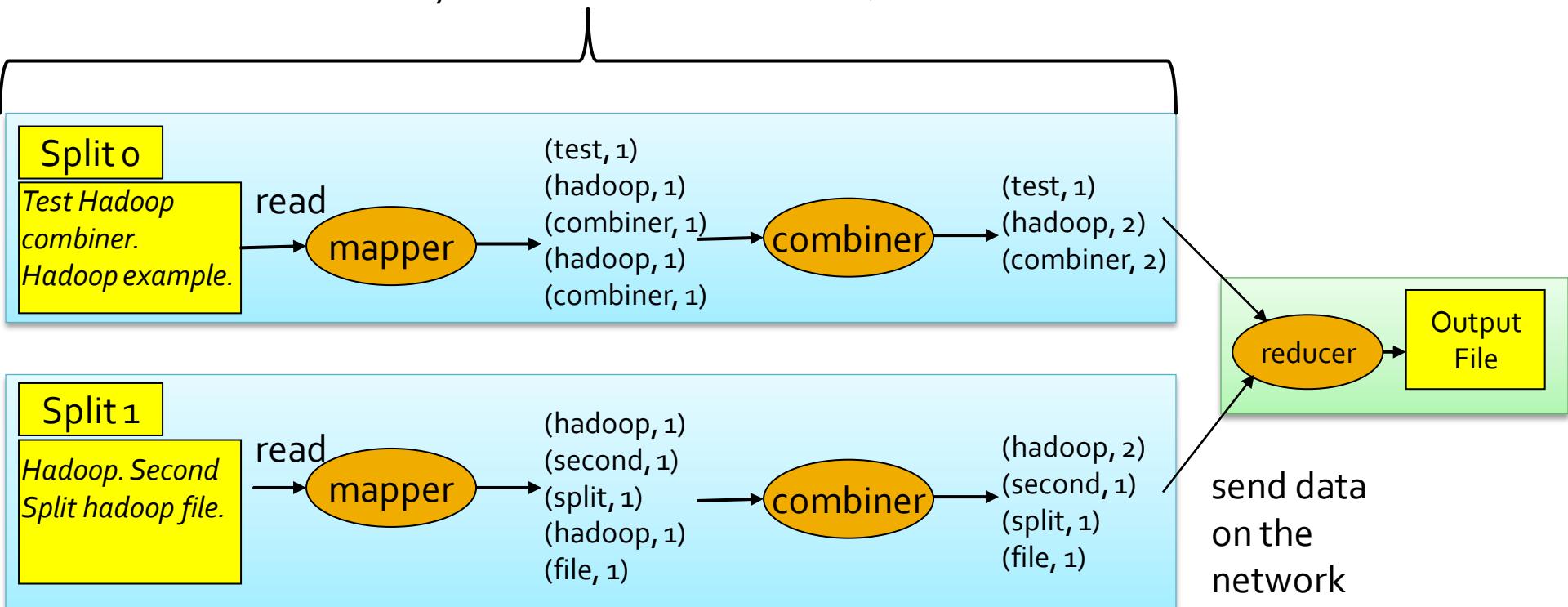
- Consider the standard word count problem
- Suppose the input file is split in two input splits
 - Hence, two Mappers are instantiated (one for each split)

Word count example without combiner



Word count example with combiner

The **combiner** is **locally** called on the output (key, value) pairs of the mapper (it works on data stored in the main-memory or on the local hard disks)



the number of mappers instances depends on the size of input file and the size of each mappers.
in this case the file is 100 MB which means one instance of 128 mapper. but the map method is invoked based on the input split lines?!!!

Combiner

- MapReduce applications with **Combiners**
 - The (key,value) **pairs** emitted by the Mappers are **analyzed in main-memory** (or on the local disk) and **aggregated** by the Combiners
 - Each Combiner pre-aggregates the values associated with the pairs emitted by the Mappers of a cluster node
 - Combiners perform “pre-aggregations” to limit the amount of network data generated by each cluster node

Combiner

- Combiners work only if the reduce function is commutative and associative
- The execution of combiners is not guaranteed
 - Hadoop may or may not execute a combiner
 - The decision is taken at runtime by Hadoop and you cannot check it in your code
 - Your MapReduce job should not depend on the Combiner execution

Combiner

- The Combiner
 - Is an **instance** of the `org.apache.hadoop.mapreduce.Reducer` class
 - There is not a specific combiner-template class
 - “Implements” a pre-reduce phase that aggregates the pairs emitted in each node by Mappers
 - Is characterized by the `reduce(...)` method
 - Processes `(key, [list of values])` pairs and emits `(key, value)` pairs
 - Runs on the cluster

MapReduce programs - Combiner

- The Combiner class extends the org.apache.hadoop.mapreduce.Reducer class
 - The org.apache.hadoop.mapreduce.Reducer class
 - Is a generic type/generic class
 - With four type parameters: input key type, input value type, output key type, output value type
 - i.e., Combiners and Reducers extend the same class
- The designer/developer implements the **reduce(...)** method
 - It is automatically called by Hadoop for each (key, [list of values]) pair obtained by aggregating the local output of a Mapper

MapReduce programs - Combiner

- The Combiner class is specified by using the `job.setCombinerClass()` method in the run method of the Driver
 - i.e., in the job configuration part of the code

Word Count Example with Combiner

- Word count problem
 - Input: (unstructured) textual file
 - Each line of the input file can contains a set of words
 - Output: number of occurrences of each word appearing in the input file
- Parameters/arguments of the application:
 - args[0]: number of instances of the reducer
 - args[1]: path of the input file
 - args[2]: path of the output folder

Word Count Example with Combiner

- Input file

Toy example
file for Hadoop.
Hadoop running
example.

- Output pairs
 - (toy, 1)
 - (example, 2)
 - (file, 1)
 - (for, 1)
 - (hadoop, 2)
 - (running, 1)

Word Count Example with Combiner

- Differences with the solution without combiner
 - Specify the combiner class in the Driver
 - Define the Combiner class
 - The reduce method of the combiner aggregates local pairs emitted by the mappers of a single cluster node
 - It emits partial results (local number of occurrences for each word) from each cluster node that is used to run our application

Word Count Example with Combiner - Driver (1)

```
/* Set package */  
package it.polito.bigdata.hadoop.wordcount;  
  
/* Import libraries */  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.conf.Configured;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Job;  
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;  
import org.apache.hadoop.util.Tool;  
import org.apache.hadoop.util.ToolRunner;
```

Word Count Example with Combiner - Driver (2)

```
/* Driver class */
public class WordCount extends Configured implements Tool {
    @Override
    public int run(String[] args) throws Exception {

        Path inputPath;
        Path outputDir;
        int numberOfReducers;
        int exitCode;

        // Parse input parameters
        numberOfReducers = Integer.parseInt(args[0]);
        inputPath = new Path(args[1]);
        outputDir = new Path(args[2]);
```

Word Count Example with Combiner - Driver (3)

```
// Define and configure a new job
Configuration conf = this.getConf();
Job job = Job.getInstance(conf);

// Assign a name to the job
job.setJobName("WordCounter");
```

Word Count Example with Combiner - Driver (4)

```
// Set path of the input file/folder (if it is a folder, the job reads all the files  
in the specified folder) for this job  
FileInputFormat.addInputPath(job, inputPath);  
  
// Set path of the output folder for this job  
FileOutputFormat.setOutputPath(job, outputDir);  
  
// Set input format  
// TextInputFormat = textual files  
job.setInputFormatClass(TextInputFormat.class);  
  
// Set job output format  
job.setOutputFormatClass(TextOutputFormat.class);
```

Word Count Example with Combiner - Driver (5)

```
// Specify the class of the Driver for this job  
job.setJarByClass(WordCount.class);
```

```
// Set mapper class  
job.setMapperClass(WordCountMapper.class);
```

```
// Set map output key and value classes  
job.setMapOutputKeyClass(Text.class);  
job.setMapOutputValueClass(IntWritable.class);
```

Word Count Example with Combiner - Driver (6)

```
// Set reduce class  
job.setReducerClass(WordCountReducer.class);
```

```
// Set reduce output key and value classes  
job.setOutputKeyClass(Text.class);  
job.setOutputValueClass(IntWritable.class);
```

```
// Set number of reducers  
job.setNumReduceTasks(numberOfReducers);
```

```
// Set combiner class  
job.setCombinerClass(WordCountCombiner.class);
```

Word Count Example with Combiner - Driver (7)

```
// Execute the job and wait for completion
if(job.waitForCompletion(true)==true)
    exitCode=0;
else
    exitCode=1;
return exitCode;
} // End of the run method
```

Word Count Example with Combiner - Driver (8)

```
/* main method of the driver class */
public static void main(String args[]) throws Exception {
    /* Exploit the ToolRunner class to "configure" and run the
     Hadoop application */

    int res = ToolRunner.run(new Configuration(),
                           new WordCount(), args);

    System.exit(res);
} // End of the main method

} // End of public class WordCount
```

Word Count Example with Combiner - Mapper (1)

```
/* Set package */  
package it.polito.bigdata.hadoop.wordcount;  
  
/* Import libraries */  
import java.io.IOException;  
  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.LongWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Mapper;
```

Word Count Example with Combiner - Mapper (2)

```
/* Mapper Class */
class WordCountMapper extends Mapper<
    LongWritable, // Input key type
    Text,         // Input value type
    Text,         // Output key type
    IntWritable> // Output value type

{
    /* Implementation of the map method */
    protected void map(
        LongWritable key, // Input key type
        Text value,       // Input value type
        Context context) throws IOException, InterruptedException {

```

Word Count Example with Combiner - Mapper (3)

```
// Split each sentence in words. Use whitespace(s) as delimiter
// The split method returns an array of strings
String[] words = value.toString().split("\\s+");

// Iterate over the set of words
for(String word : words) {
    // Transform word case
    String cleanedWord = word.toLowerCase();

    // emit one pair (word, 1) for each input word
    context.write(new Text(cleanedWord), new IntWritable(1));
}

} // End map method

} // End of class WordCountMapper
```

Word Count Example with Combiner - Combiner (1)

```
/* Set package */  
package it.polito.bigdata.hadoop.wordcount;  
  
/* Import libraries */  
import java.io.IOException;  
  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Reducer;
```

Word Count Example with Combiner - Combiner (2)

```
/* Combiner Class */
class WordCountCombiner extends Reducer<
    Text,           // Input key type
    IntWritable,    // Input value type
    Text,           // Output key type
    IntWritable>   // Output value type
{
    /* Implementation of the reduce method */
    protected void reduce(
        Text key, // Input key type
        Iterable<IntWritable> values, // Input value type
        Context context) throws IOException, InterruptedException {
```

Word Count Example with Combiner - Combiner (3)

```
/* Implementation of the reduce method */
protected void reduce(
    Text key, // Input key type
    Iterable<IntWritable> values, // Input value type
    Context context) throws IOException, InterruptedException {
    int occurrences = 0;

    // Iterate over the set of values and sum them
    for (IntWritable value : values) {
        occurrences = occurrences + value.get();
    }
    // Emit the total number of occurrences of the current word
    context.write(key, new IntWritable(occurrences));
} // End reduce method
} // End of class WordCountCombiner
```

Word Count Example with Combiner - Reducer (1)

```
/* Set package */  
package it.polito.bigdata.hadoop.wordcount;  
  
/* Import libraries */  
import java.io.IOException;  
  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Reducer;
```

Word Count Example with Combiner - Reducer (2)

```
/* Reducer Class */
class WordCountReducer extends Reducer<
    Text,           // Input key type
    IntWritable,    // Input value type
    Text,           // Output key type
    IntWritable>   // Output value type
{
    /* Implementation of the reduce method */
    protected void reduce(
        Text key, // Input key type
        Iterable<IntWritable> values, // Input value type
        Context context) throws IOException, InterruptedException {
```

Word Count Example with Combiner - Reducer (3)

```
/* Implementation of the reduce method */
protected void reduce(
    Text key, // Input key type
    Iterable<IntWritable> values, // Input value type
    Context context) throws IOException, InterruptedException {
    int occurrences = 0;

    // Iterate over the set of values and sum them
    for (IntWritable value : values) {
        occurrences = occurrences + value.get();
    }
    // Emit the total number of occurrences of the current word
    context.write(key, new IntWritable(occurrences));
} // End reduce method
} // End of class WordCountReducer
```

Word Count Example with Combiner

- The reducer and the combiner classes perform the same computation (the reduce method of the two classes is the same)
- We do not really need two different classes
 - We can simply specify that WordCountReducer is also the combiner class
 - In the driver

```
job.setCombinerClass(WordCountReducer.class);
```
 - In 99% of the Hadoop applications the same class can be used to implement both combiner and reducer

Personalized Data Types

Personalized Data Types and Values

- Personalized Data Types are useful when the value of a key-value pair is a complex data type
- Personalized Data Types are defined by implementing the org.apache.hadoop.io.Writable interface
 - The following methods must be implemented
 - public void readFields(DataInput in)
 - public void write(DataOutput out)
 - To properly format the output of the job usually also the following method is “redefined”
 - public String toString()

Personalized Data Types - Example

- Suppose to be interested in “complex” values composed of two parts:
 - a counter (int)
 - a sum (float)
- An ad-hoc Data Type can be used to implement this complex data type in Hadoop

Personalized Data Types - Example

(1)

```
package it.polito.bigdata.hadoop.combinerexample;
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

public class SumAndCountWritable implements
    org.apache.hadoop.io.Writable {
    /* Private variables */
    private float sum = 0;
    private int count = 0;
```

Personalized Data Types - Example

(2)

```
/* Methods to get and set private variables of the class */
public float getSum() {
    return sum;
}

public void setSum(float sumValue) {
    sum=sumValue;
}

public int getCount() {
    return count;
}

public void setCount(int countValue) {
    count=countValue;
}
```

Personalized Data Types - Example

(3)

```
/* Methods to serialize and deserialize the contents of the
instances of this class */
@Override /* Serialize the fields of this object to out */
public void write(DataOutput out) throws IOException {
    out.writeFloat(sum);
    out.writeInt(count);
}

@Override /* Deserialize the fields of this object from in */
public void readFields(DataInput in) throws IOException {
    sum=in.readFloat();
    count=in.readInt();
}
```

Personalized Data Types - Example

(4)

```
/* Specify how to convert the contents of the instances of this
class to a String
 * Useful to specify how to store/write the content of this class
 * in a textual file */
public String toString()
{
    String formattedString=
        new String("sum="+sum+",count="+count);

    return formattedString;
}

}
```

Personalized Data Types and Keys

- Personalized Data Types can be used also to manage complex keys
- In that case the Personalized Data Type must implement the org.apache.hadoop.io.WritableComparable interface
 - Because keys must be compared/sorted
 - Implement the compareTo() method
 - And split in groups
 - Implement the hashCode() method

Sharing parameters among Driver, Mappers, and Reducers

Sharing parameters among Driver, Mappers, and Reducers

- The configuration object is used to share the (basic) configuration of the Hadoop environment across the driver, the mappers and the reducers of the application/job
- It stores a list of (property-name, property-value) pairs
- Personalized (property-name, property-value) pairs can be specified in the driver
 - They can be used to share some parameters of the application with mappers and reducers

Sharing parameters among Driver, Mappers, and Reducers

- Personalized (property-name, property-value) pairs are useful to share small (**constant**) properties that are available only during the execution of the program
 - The driver set them
 - Mappers and Reducers can access them
 - Their values **cannot be modified by mappers and reducers**

Sharing parameters among Driver, Mappers, and Reducers

- In the driver
 - Configuration conf = this.getConf();
 - Retrieve the configuration object
 - conf.set("property-name", "value");
 - Set personalized properties
- In the Mapper and/or Reducer
 - context.getConfiguration().get("property-name")
 - This method returns a String containing the value of the specified property

Counters

Counters

- Hadoop provides a set of basic, built-in, counters to store some statistics about jobs, mappers, reducers
 - E.g., number of input and output records (i.e., pairs)
 - E.g., number of transmitted bytes
- Ad-hoc, user-defined, counters can be defined to compute global “statistics” associated with the goal of the application

User-defined Counters

initial value of the counter is zero automatically. since we have a lot of instances of mapper and reducer class, to avoid synchronization of between them, we should just increment the value of counter not to assign value to it

■ User-defined counters

- Are defined by means of Java enum
 - Each application can define an arbitrary number of enums
- Are **incremented** in the Mappers and Reducers
- The global/final **value** of each **counter** is available at the **end of the job**
 - **It is stored/printed by the Driver** (at the end of the execution of the job)

User-defined Counters

- The name of the enum is the group name
 - Each enum has a number of “fields”
- The enum’s fields are the counter names
- In mappers and/or reducers counters are incremented by using the increment()
method
 - `context.getCounter(countername).increment(value);`

User-defined Counters

- The `getCounters()` and `findCounter()` methods are used by the `Driver` to retrieve the final values of the counters

User-defined Dynamic Counters

- User-defined counters can be also defined **on the fly**
 - By using the method incrCounter("group name", "counter name", value)
- Dynamic counters are useful when the set of counters is unknown at design time

Example of user-defined counters

- In the driver

```
public static enum COUNTERS {  
    ERROR_COUNT,  
    MISSING_FIELDS_RECORD_COUNT  
}
```

- This enum defines two counters
 - COUNTERS.ERROR_COUNT
 - COUNTERS.MISSING_FIELDS_RECORD_COUNT

Example of user-defined counters

- This example increments the COUNTERS.ERROR_COUNT counter
 - In the mapper or the reducer

```
context.getCounter(COUNTERS.ERROR_COUNT).increment(1);
```

Example of user-defined counters

- This example retrieves the final value of the COUNTERS.ERROR_COUNT counter
- In the driver

```
Counter errorCounter =  
    job.get_counters().find_counter(COUNTERS.ERROR  
    _COUNT);
```

Map-only job

Map-only job

- In some applications all the work can be performed by the mapper(s)
 - E.g., record filtering applications
- Hadoop allows executing Map-only jobs
 - The **reduce phase is avoided**
 - Also the **shuffle and sort phase is not executed**
 - The **output of the map job** is **directly stored in HDFS**
 - i.e., the set of pairs emitted by the map phase is already the final output

Map-only job

- Implementation of a Map-only job
 - Implement the map method
 - Set the number of reducers to 0 during the configuration of the job (in the driver)
 - `job.setNumReduceTasks(0);`

Paolo Paolo
No Paolo
Luca

(0, Paolo Paolo)
(11, No Paolo)
(18, Luca)

they are the input for map phase

we need only the lines with Paolo without any key as output of map,
(key, value)pair must be defined so
(", Paolo Paolo) or (NullWritable, Paolo Paolo)

so we have just one distinct key ", so the reduce method is invoked just once and do nothing for us actually (also a combiner), so we dont use reducer and send any data through network and the map output is stored in the HDFS

In-Mapper combiner

Setup and cleanup method

- Mapper classes are also characterized by a setup and a cleanup method
 - They are empty if they are not overridden
- The setup method is called once for each mapper prior to the many calls of the map method
 - It can be used to set the values of in-mapper variables
 - In-mapper variables are used to maintain in-mapper statistics and preserve the state (locally for each mapper) within and across calls to the map method

Setup and cleanup method

- The **map method**, invoked many times, updates the value of the in-mapper variables
 - **Each mapper** (each instance of the mapper class) **has its own copy of the in-mapper variables**
- The **cleanup** method **is called once for each mapper** after the many calls to the map method
 - It can be used to **emit** (key,value) pairs based on the values of the in-mapper variables/statistics

Setup and cleanup method

- Also the **reducer** classes are characterized by
a setup and a cleanup method
- The **setup** method **is called once for each reducer** prior to the many calls of the reduce method
- The **cleanup** method **is called once for each reducer** after the many calls of the reduce method

In-Mapper Combiners

- In-Mapper Combiners, a **possible improvement** over “standard” Combiners
 - **Initialize** a set of in-mapper variables during the instance of the Mapper
 - Initialize them in the setup method of the mapper
 - Update the in-mapper variables/statistics in the map method
 - Usually, no (key,value) pairs are emitted in the map method of an in-mapper combiner

In-Mapper Combiners

- After all the input records (input (key, value) pairs) of a mapper have been analyzed by the map method, emit the output (key, value) pairs of the mapper
 - (key, value) pairs are emitted in the cleanup method of the mapper based on the values of the in-mapper variables

In-Mapper Combiners

- The in-mapper variables are used to perform the work of the combiner in the mapper
 - It can allow improving the overall performance of the application
 - But **pay attention** to the amount of **used main memory**
 - Each mapper can use a limited amount of main-memory
 - Hence, **in-mapper variables** should be “**small**” (at least smaller than the maximum amount of memory assigned to each mapper)

In-Mapper Combiner – Word count

Pseudo code

class MAPPER

method setup

$A \leftarrow \text{new AssociativeArray}$

this is specific for each instance of map class

potential problem is here, it may cause the running out o local main memory

method map(offset key, line l)

for all word $w \in \text{line } l$ do

$A\{w\} \leftarrow A\{w\} + 1$

method cleanup

for all word $w \in A$ do

EMIT(term w , count $A\{w\}$)

In-Mapper Combiner – Word count

Pseudo code

```
class MAPPER
```

```
    method setup
```

```
        A ← new AssociativeArray
```

} Invoked **one time**
for each mapper

```
    method map(offset key, line l)
```

```
        for all word w ∈ line l do
```

```
            A{w} ← A{w} + 1
```

```
    method cleanup
```

```
        for all word w ∈ A do
```

```
            EMIT(term w , count A{w})
```

} Invoked **one time**
for each mapper

In-Mapper Combiner – Word count

Pseudo code

class MAPPER

method setup

$A \leftarrow \text{new AssociativeArray}$

} Invoked **one time** for each mapper

method map(offset key, line l)

for all word $w \in \text{line } l$ do

$A\{w\} \leftarrow A\{w\} + 1$

} Invoked **multiple times** for each mapper

method cleanup

for all word $w \in A$ do

EMIT(term w , count $A\{w\}$)

} Invoked **one time** for each mapper

Part I

MapReduce patterns

Summarization Patterns

Summarization patterns

- Are used to implement applications that produce top-level/summarized view of the data
 - Numerical summarizations (Statistics)
 - Inverted index
 - Counting with counters

Summarization Patterns

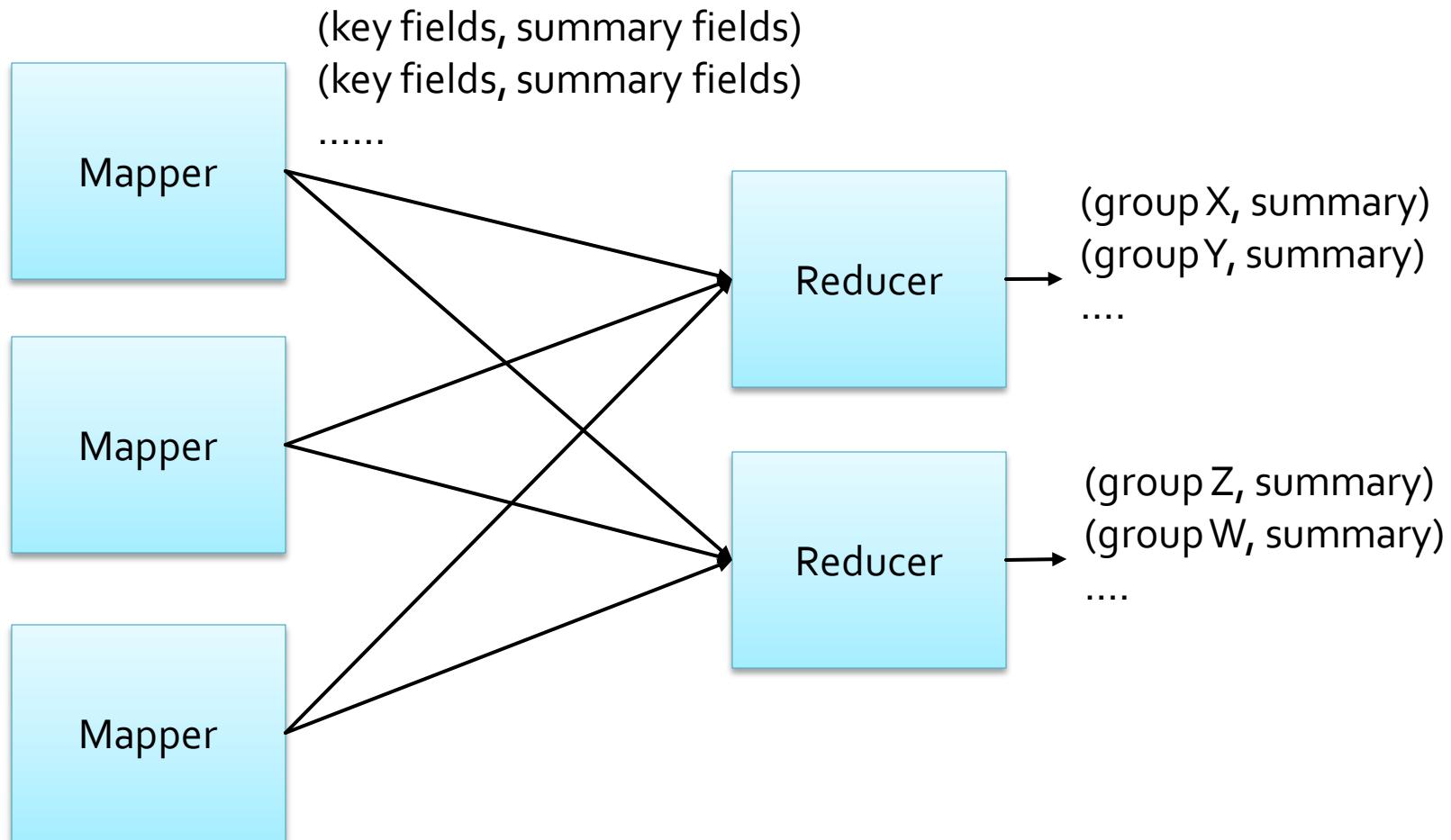
Numerical Summarizations

- Goal
 - Group records/objects by a key field(s) and calculate a numerical aggregate (average, max, min, standard deviation,...) per group
- Provide a top-level view of large input data sets
- Motivation
 - Few high-level statistics can be analyzed by domain experts to identify trends, anomalies, ...

Numerical Summarizations - structure

- Mappers
 - Output (key, value) pairs where
 - key is associated with the fields used to define groups
 - value is associated with the fields used to compute the aggregate statistics
- Reducers
 - Receive a set of numerical values for each “group-by” key and compute the final statistics for each “group”
- Combiners
 - If the computed statistic has specific properties (e.g., it is commutative and associative), combiners can be used to speed up performances

Numerical Summarizations - structure



Numerical Summarizations

- Known uses
 - Word count
 - Record count (per group)
 - Min/Max/Count (per group)
 - Average/Median/Standard deviation (per group)

Summarization Patterns

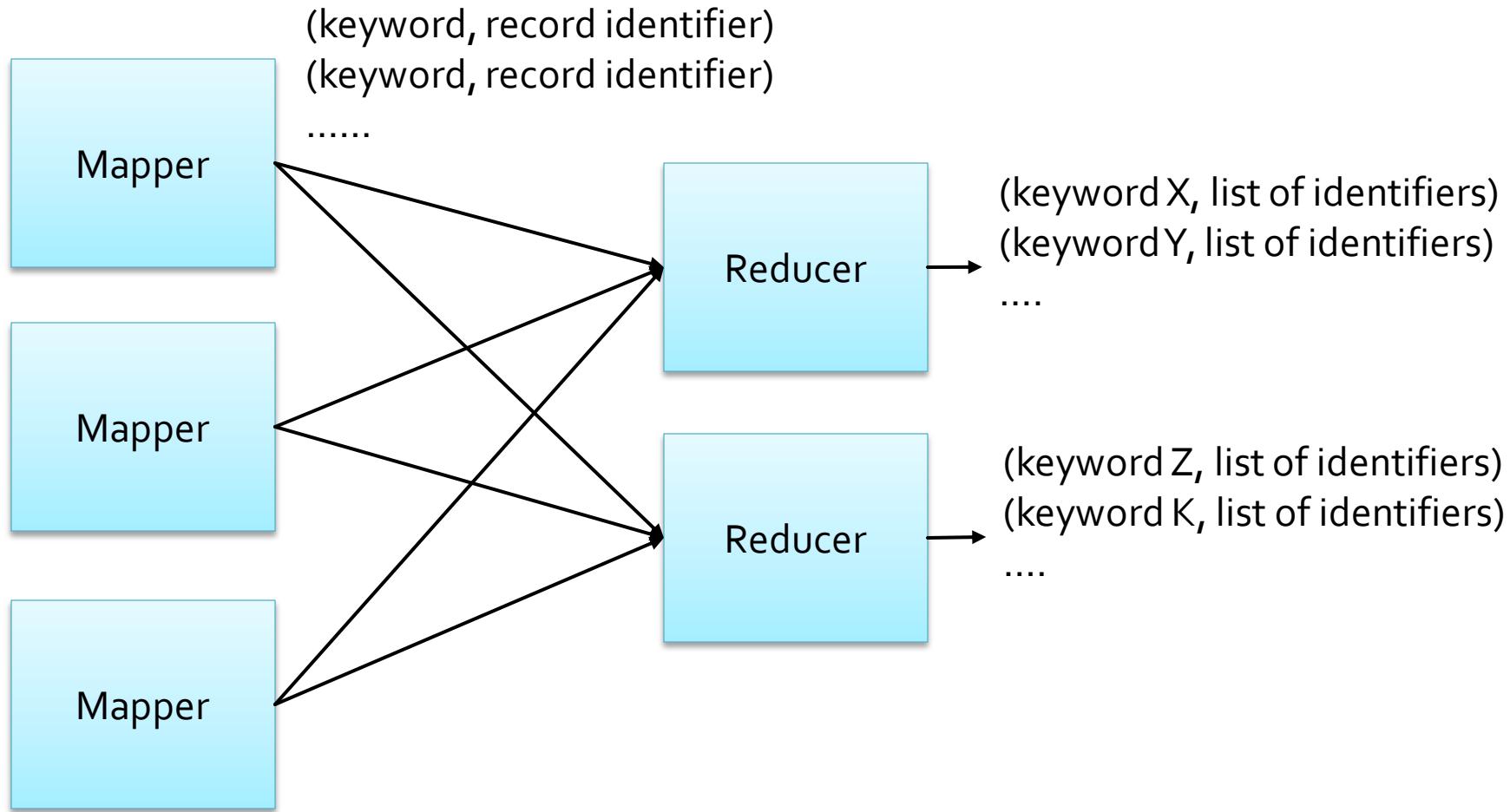
Inverted Index Summarizations

- Goal
 - Build an index from the input data to support faster searches or data enrichment
- Map terms to a list of identifiers
- Motivation
 - Improve search efficiency

Inverted Index Summarizations - structure

- Mappers
 - Output (key, value) pairs where
 - key is the set of fields to index (a keyword)
 - value is a unique identifier of the objects to associate with each “keyword”
- Reducers
 - Receive a set of identifiers for each keyword and simply concatenate them
- Combiners
 - Usually are not useful when using this pattern
 - Usually there are no values to aggregate

Inverted Index Summarizations - structure



Inverted Index Summarizations

- Most famous known use
 - Web search engine
 - Word – List of URLs (Inverted Index)

Summarization Patterns

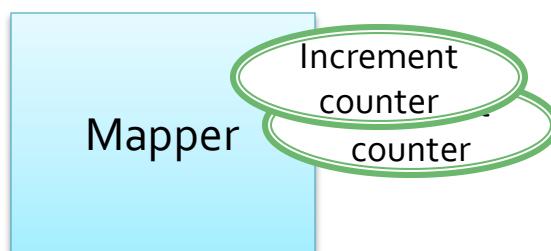
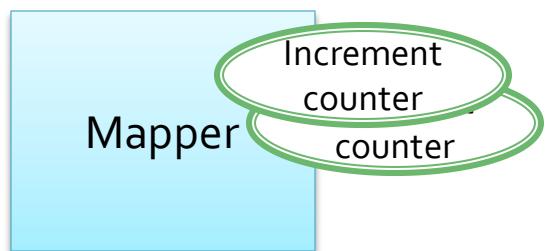
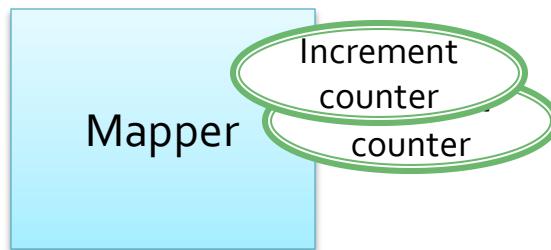
Counting with Counters

- Goal
 - Compute count summarizations of data sets
- Provide a top-level view of large data sets
- Motivation
 - Few high-level statistics can be analyzed by domain experts to identify trends, anomalies, ...

Counting with Counters - structure

- Mappers
 - Process each input record and increment a set of counters
- Map-only job
 - No reducers
 - No combiners
- The results are stored/printed by the Driver of the application

Counting with Counters - structure



Counting with Counters

- Known uses
 - Count number of records
 - Count a small number of unique instances
 - Summarizations

Filtering Patterns

Filtering Patterns

- Are used to select the subset of input records of interest
 - Filtering
 - Top K
 - Distinct

Filtering Patterns

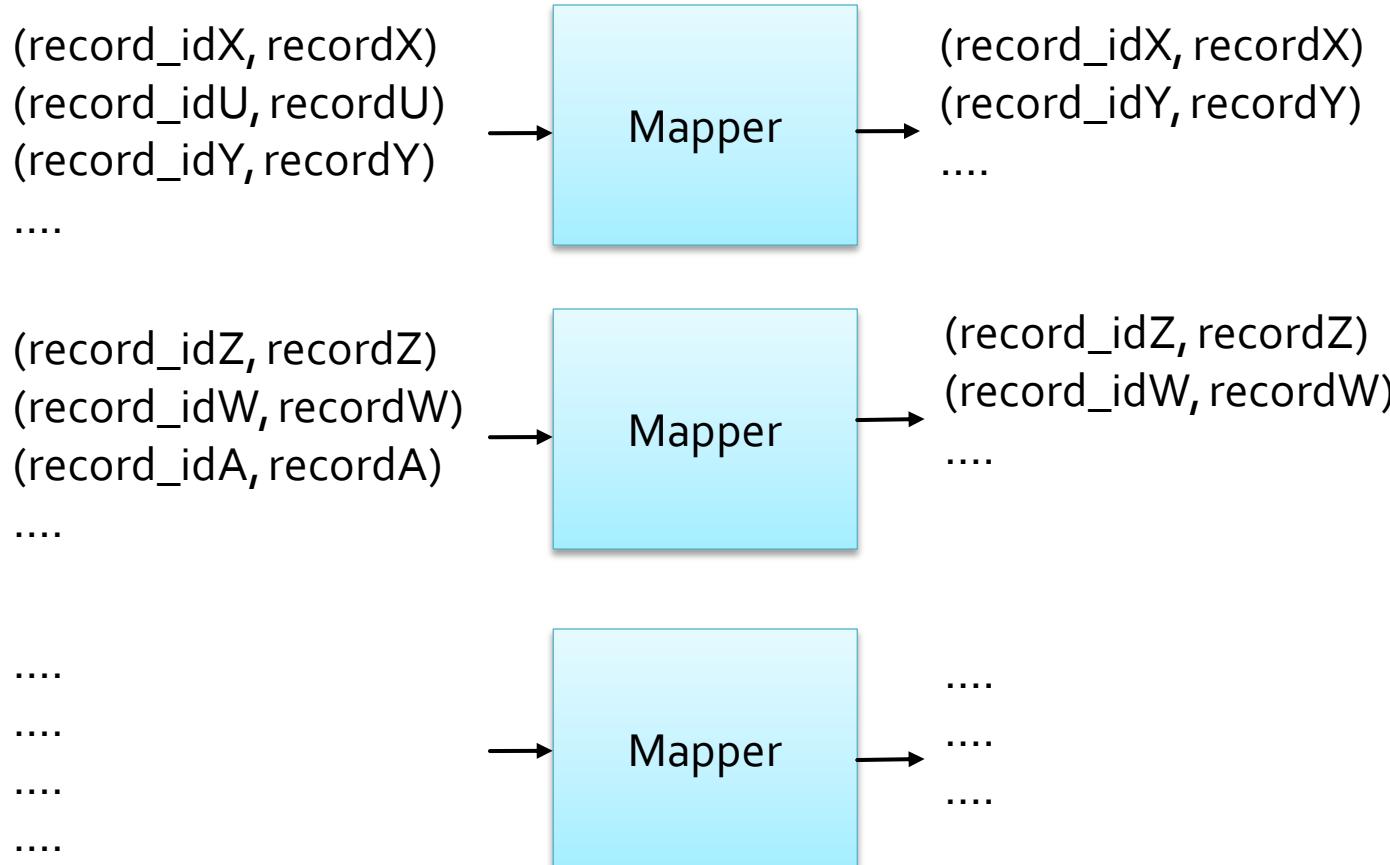
Filtering

- Goal
 - Filter out input records that are not of interest/keep only the ones that are of interest
- Focus the analysis of the records of interest
- Motivation
 - Depending on the goals of your application, frequently only a small subset of the input data is of interest for further analyses

Filtering - structure

- The input of the mapper is a set of records
 - Key = primary key
 - Value = record
- Mappers
 - Output one (key, value) pair for each record that satisfies the enforced filtering rule
 - Key is associated with the primary key of the record
 - Value is associated with the selected record
 - Reducers
 - The reducer is useless in this pattern
 - A **map-only job** is executed (number of reduce set to 0)

Filtering - structure



Filtering

- Known uses
 - Record filtering
 - Tracking events
 - Distributed grep
 - Data cleaning

Filtering Patterns

Top K

- Goal
 - Select a small set of top K records according to a ranking function
- Focus on the most important records of the input data set
- Motivation
 - Frequently the interesting records are those ranking first according to a ranking function
 - Most profitable items
 - Outliers

Top K - structure

- Mappers
 - Each mapper initializes an in-mapper (local) top k list
 - k is usually small (e.g., 10)
 - The current (local) top k-records of each mapper (i.e., instance of the mapper class) can be stored in main memory
 - Initialization performed in the setup method of the mapper
 - The map function updates the current in-mapper top k list

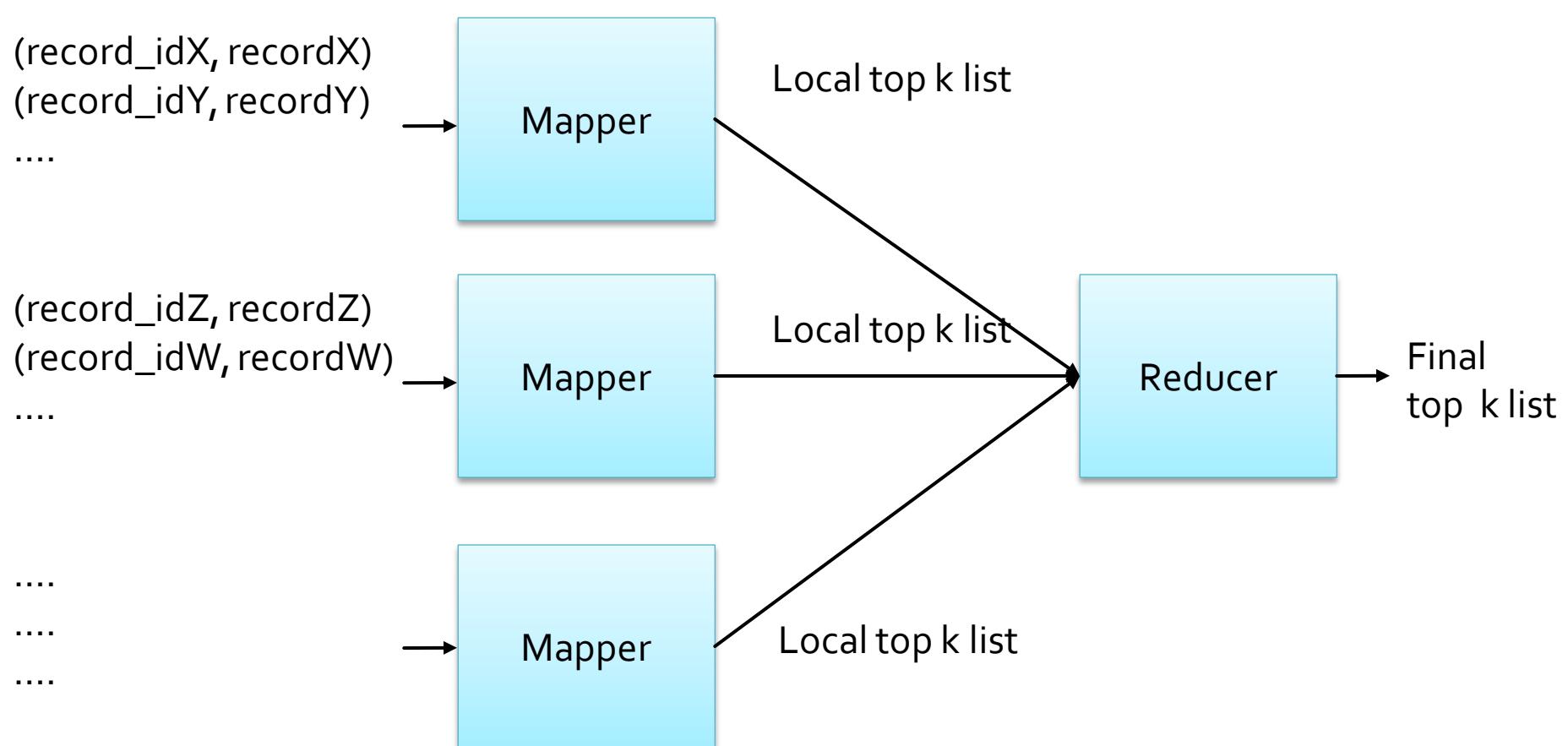
Top K - structure

- Mappers
 - The cleanup method emits the k (key, value) pairs associated with the in-mapper local top k records
 - Key is the “null key” to be sure that the reducer is invoked just one time since all keys are same
 - Value is a in-mapper top k record

Top K - structure

- Reducer
 - A single reducer must be instantiated (i.e., **one single instance** of the **reducer** class)
 - One single **global view** over the intermediate results emitted by the mappers to compute the final top k records
 - It computes the final top k list by merging the local lists emitted by the mappers
 - All input (key, value) pairs have the same key
 - Hence, the reduce method is called only once

Top K - structure



Top K

- Known uses
 - Outlier analysis (based on a ranking function)
 - Select interesting data (based on a ranking function)

Filtering Patterns

Distinct

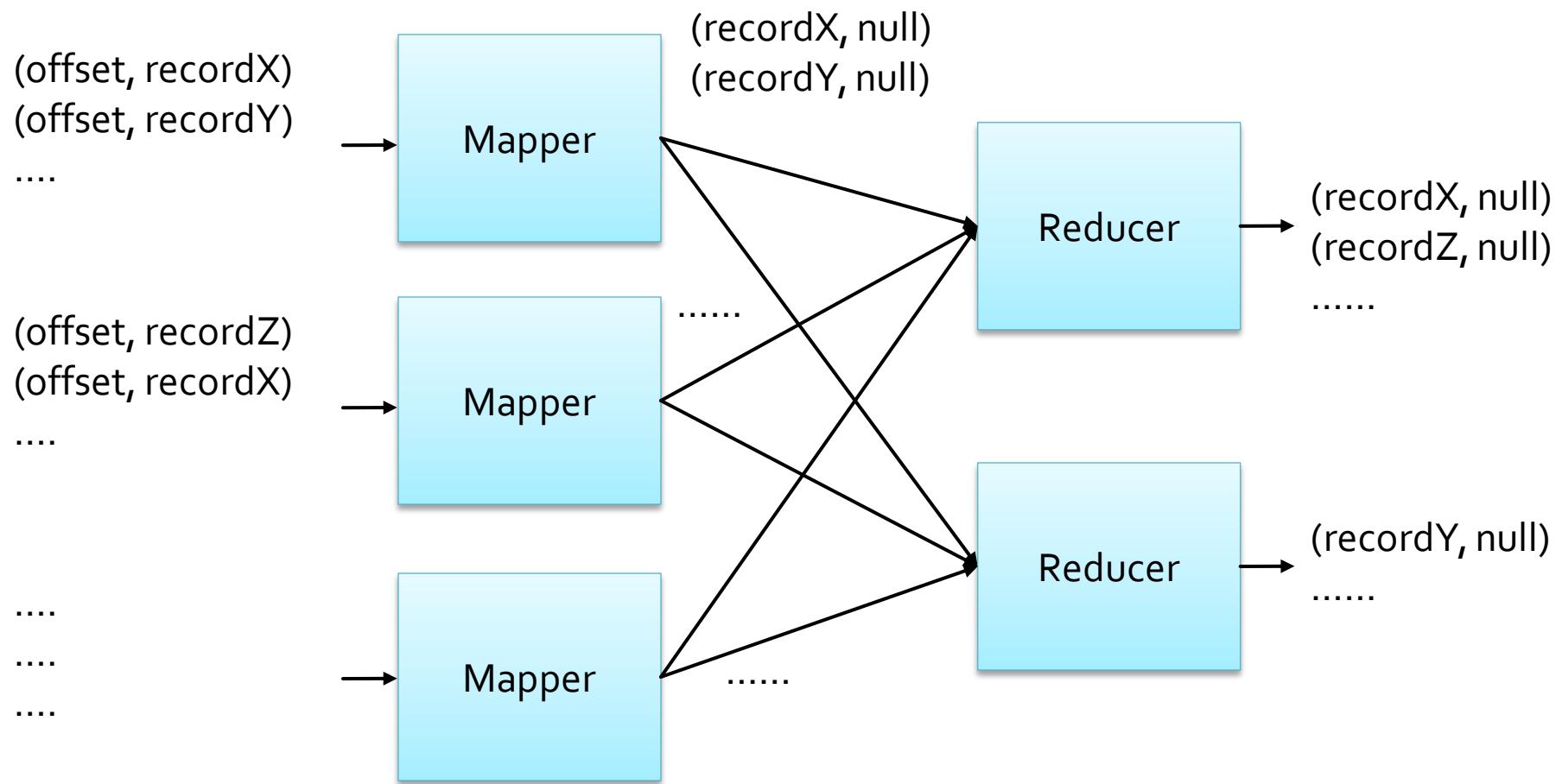
- Goal
 - Find a unique set of values/records
- In some applications duplicate records are useless
- Motivation
 - Duplicates records are frequently useless

Distinct - structure

- Mappers
 - Emit one (key, value) pair for each input record
 - Key = input record
 - Value = null value
- Reducers
 - Emit one (key, value) pair for each input (key, list of values) pair
 - Key = input key, i.e., input record
 - Value = null value

in this way, since reducer is invoked for each distinct key so no matter how many duplicated keys we have, reducer makes it to one output per each distinct key

Distinct - structure



Distinct

- Known uses
 - Duplicate data removal
 - Distinct value selection

MapReduce and Hadoop: Advanced topics

Multiple inputs

Multiple inputs

- In some applications data are read from two or more datasets
 - Datasets could have different formats
- Hadoop allows reading data from **multiple inputs** (multiple datasets) with different **formats**
 - One different mapper for each input dataset must be specified
 - However, the key-value **pairs emitted** by the mappers must be **consistent in terms of data types**

Multiple inputs

- Example of a use case
 - Input data collected from different sensors
 - All sensors measure the same “measure”
 - But sensors developed by different vendors use a different data format to store the gathered data/measurements

Multiple inputs

- In the driver
 - Use the `addInputPath` method of the `MultipleInputs` class multiple times to
 - Add one input path at a time
 - Specify the input format class for each input path
 - Specify the Mapper class associated with each input path

Multiple inputs

- E.g.,

```
MultipleInputs.addInputPath(job, new Path(args[1]),  
    TextInputFormat.class, Mapper1.class);
```

```
MultipleInputs.addInputPath(job, new Path(args[2]),  
    TextInputFormat.class, Mapper2.class);
```

- Specify two input paths (args[1] and args[2])
- The data of both paths are read by using the TextInputFormat class
- Mapper1 is the class used to manage the input key-value pairs associated with the first path
- Mapper2 is the class used to manage the input key-value pairs associated with the second path

Multiple outputs

Multiple outputs

- In some applications it could be useful to store the output key-value pairs of a MapReduce application in different files
 - Each file contains a specific subset of the emitted key-value pairs (based on some rules)
 - Usually this approach is useful for splitting and filtering operations
 - Each file name has a prefix that is used to specify the “content” of the file
- All the files are stored in one single output directory
 - i.e., there are no multiple output directories, but only multiple output files with different prefixes

Multiple outputs

- Hadoop allows specifying the prefix of the output files
 - The standard prefix is “part-” (see the content of the output directory of some of the previous applications)
 - The **MultipleOutputs** class is used to specify the prefixes of the output files
 - One different prefix for each “type” of output file
 - There will be one output file of each type for each reducer (for each mapper for map-only jobs)

Multiple outputs - Driver

- Use the method **MultipleOutputs.addNamedOutput** multiple times in the Driver to specify the prefixes of the output files
- The method has 4 parameter
 - The job object
 - The “name/prefix” of MultipleOutputs
 - The OutputFormat class
 - The key output data type class
 - The value output data type class
- Call this method one time for each “output file type”

Multiple outputs - Driver

- E.g.,
`MultipleOutputs.addNamedOutput(job, "hightemp",
TextOutputFormat.class, Text.class, NullWritable.class);`

`MultipleOutputs.addNamedOutput(job, "normaltemp",
TextOutputFormat.class, Text.class, NullWritable.class);`
- This example defines two types of output files
 - The first type of output files while have the prefix “hightemp”
 - The second type of output files while have the prefix “normaltemp”

Multiple outputs – Map-only example

- Define a private `MultipleOutputs` variable in the mapper if the job is a map-only job (in the reducer otherwise)
 - E.g.,
 - `private MultipleOutputs<Text, NullWritable> mos = null;`
- Create an instance of the `MultipleOutputs` class in the `setup` method of the mapper (or in the reducer)
 - E.g.,
 - `mos = new MultipleOutputs<Text, NullWritable>(context);`

Multiple outputs – Map-only example

- Use the write method of the MultipleOutputs object in the map method (or in the reduce method) to write the key-value pairs in the file of interest
 - E.g.,
 - `mos.write("hightemp", key, value);`
 - This example writes the current key-value pair in a file with the prefix “hightemp-”
 - `mos.write("normaltemp", key, value);`
 - This example writes the current key-value pair in a file with the prefix “normaltemp-”

Multiple outputs – Map-only example

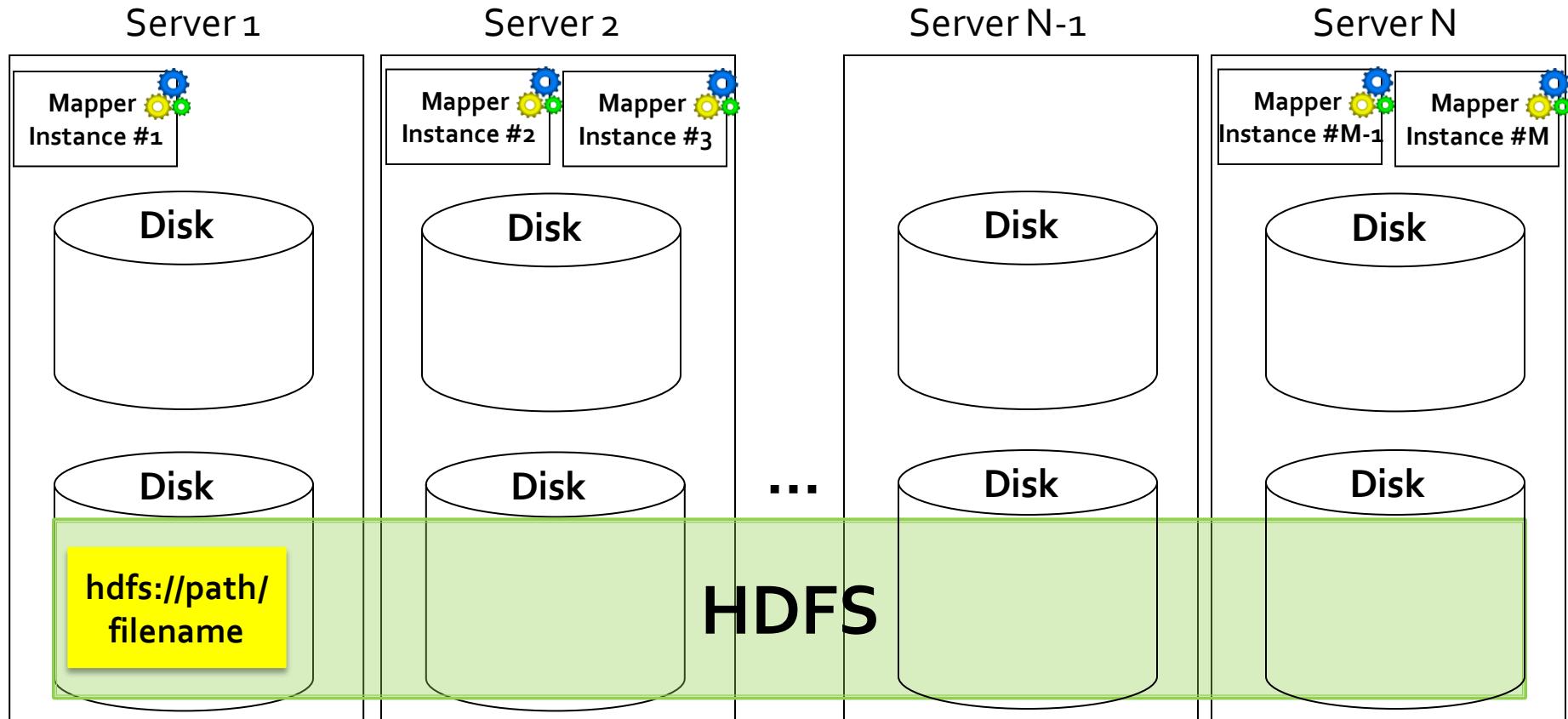
- Close the `MultipleOutputs` object in the cleanup method of the mapper (or of the reducer)
 - E.g.,
 - `mos.close();`

Distributed cache

Distributed cache

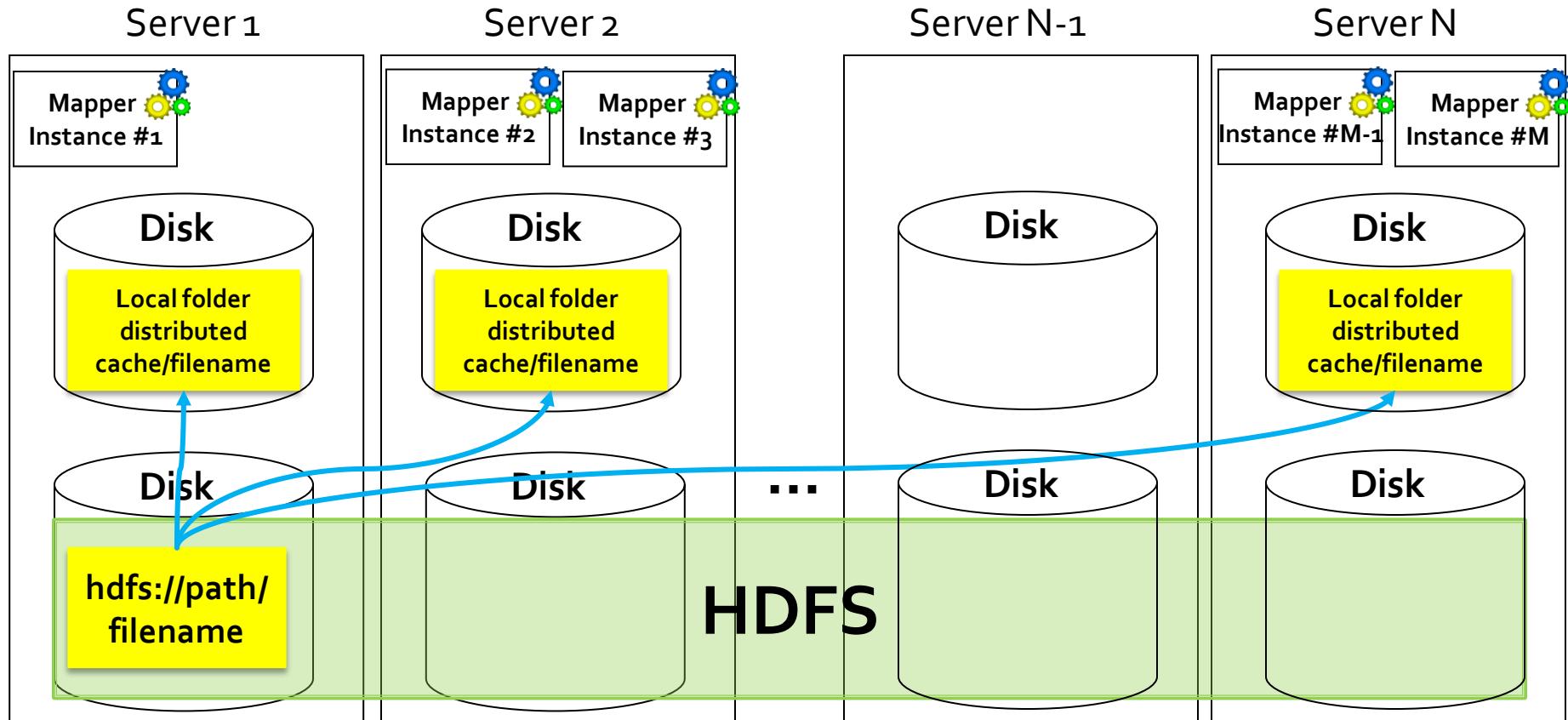
- Some applications need to share and cache (small) read-only files to perform efficiently their task
- These files should be accessible by all nodes of the cluster in an efficient way
 - Hence a copy of the shared/cached (HDFS) files should be available locally in all nodes used to run the application
- **DistributedCache** is a facility provided by the Hadoop-based MapReduce framework to cache files
 - E.g., text, archives, jars needed by applications

Distributed cache



like removing stop words from the big textual document

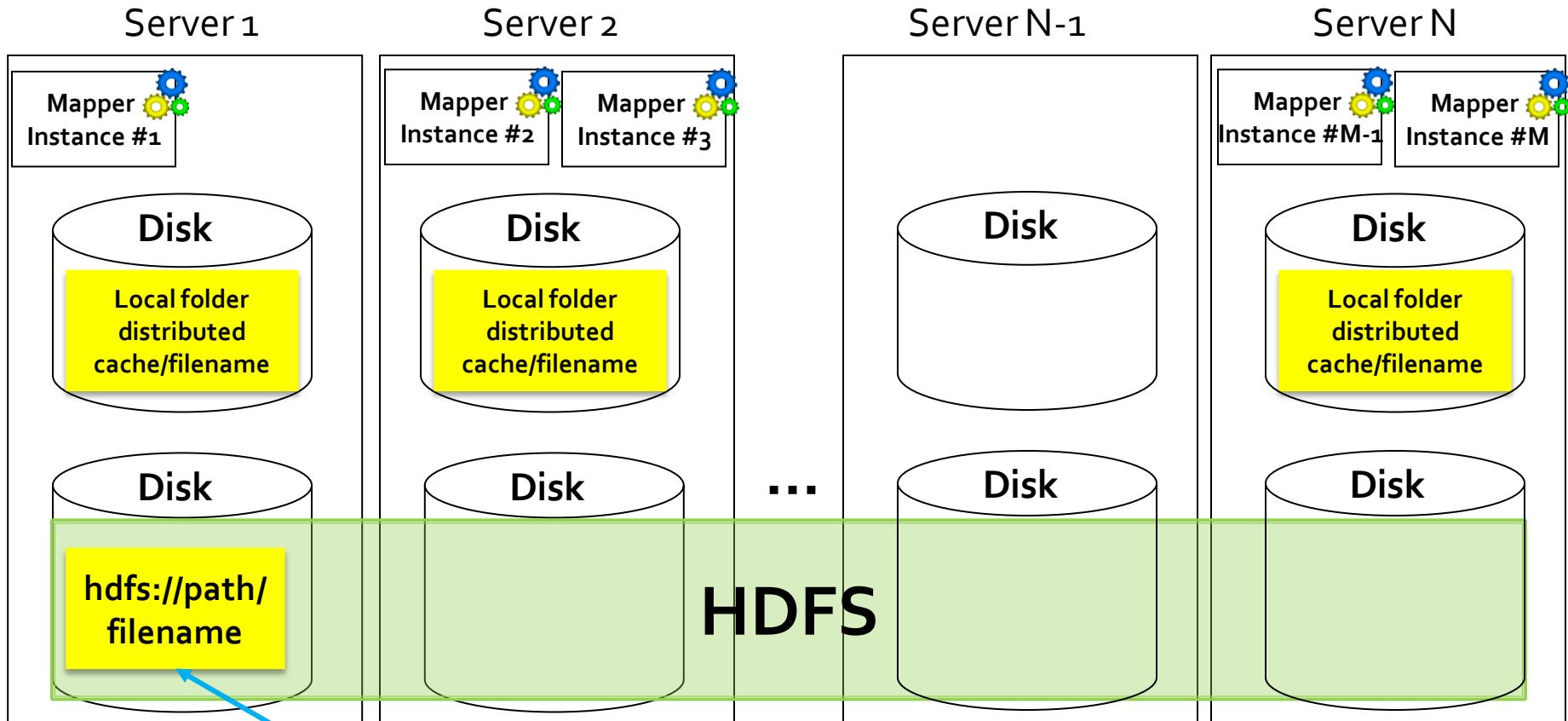
Distributed cache



the system sends the copy of small data to the servers through network and then servers use them in their local systems

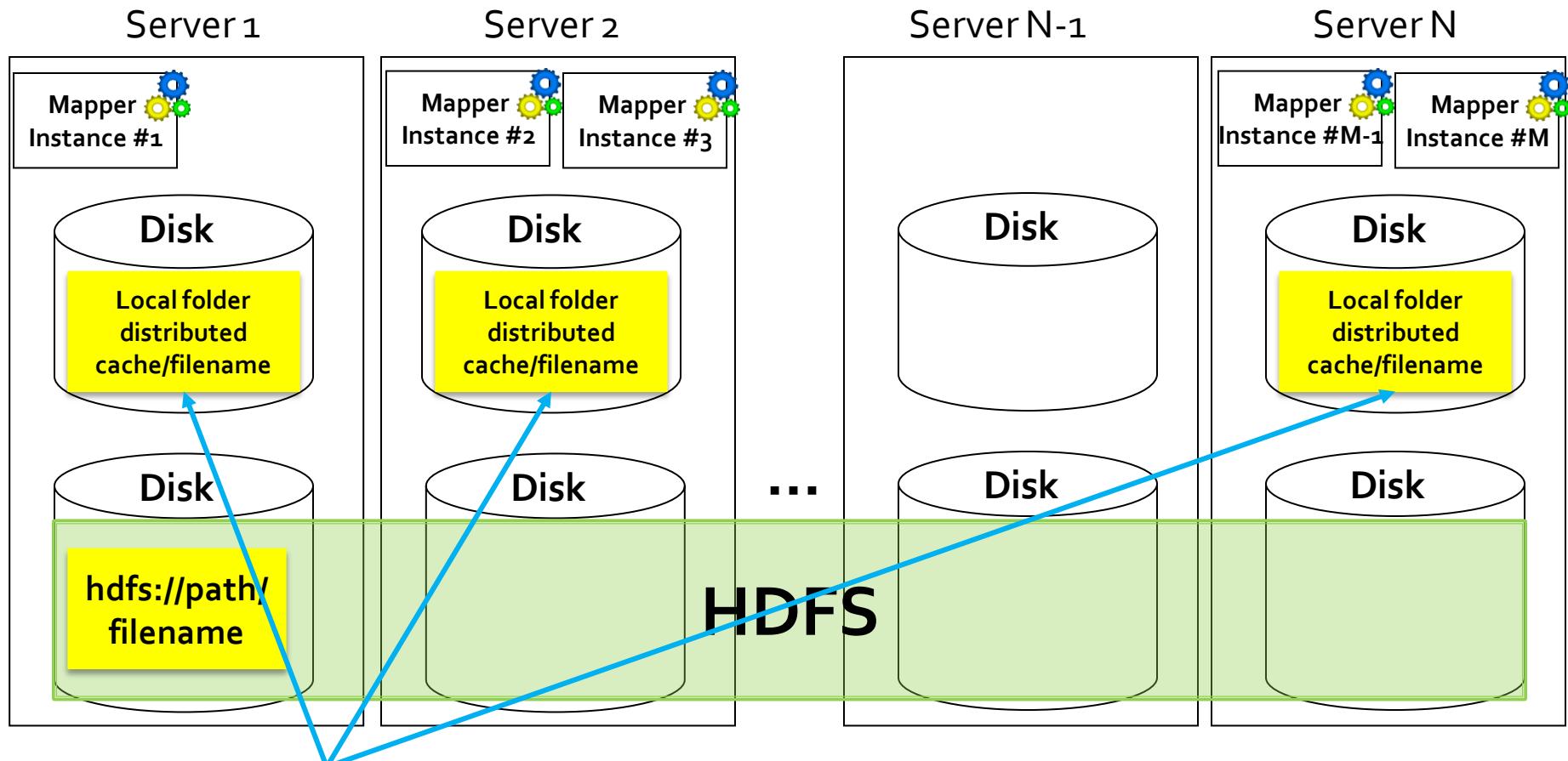
the reason is that we just send the data just once through the network not based on the number of instances of mappers or reducers, N.B. all systems are in one network

Distributed cache



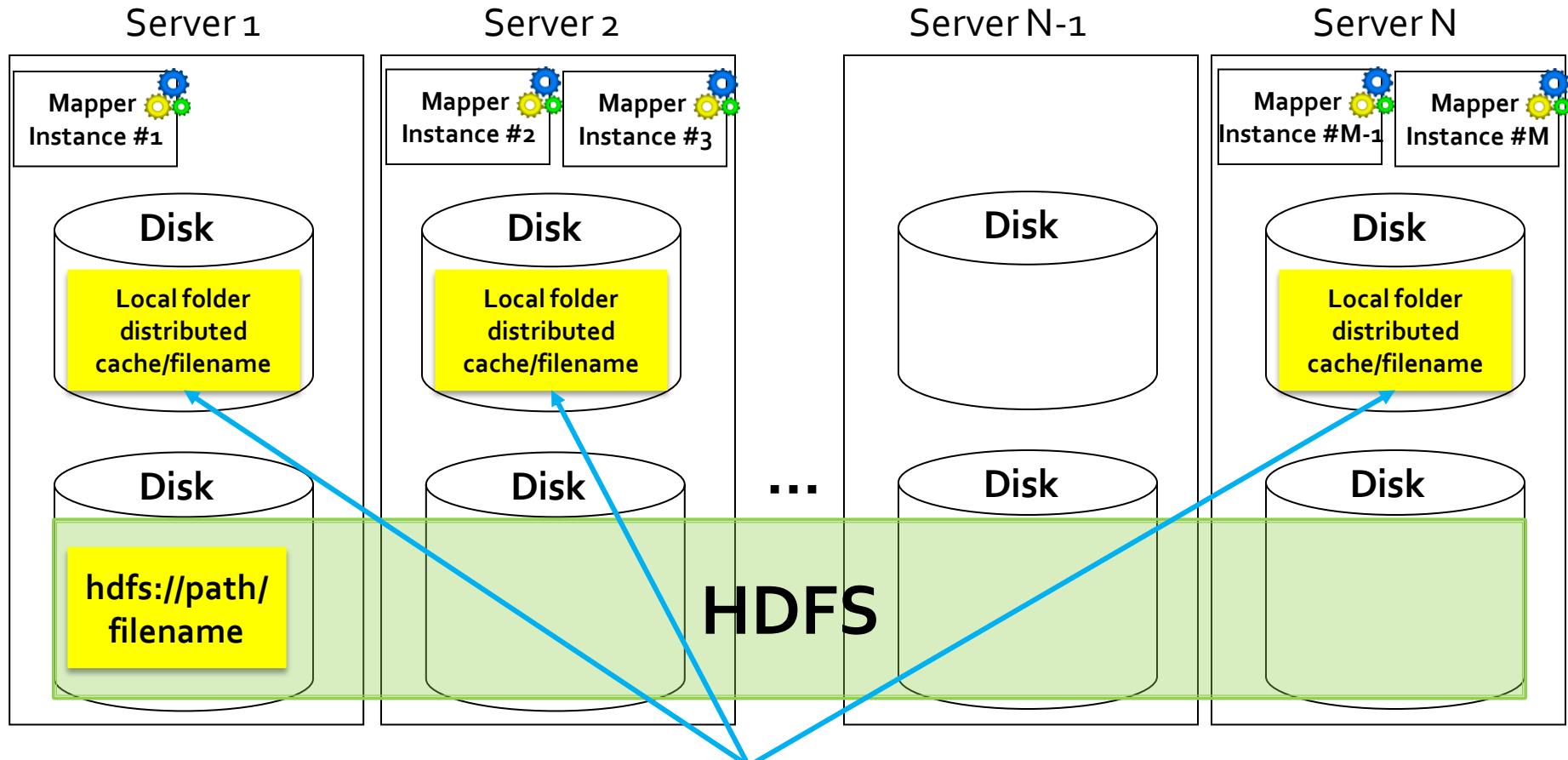
HDFS file(s) to be shared by means
of the distributed cache

Distributed cache



Local copies of the file(s) shared by means of the distributed cache

Distributed cache



A local copy of the file(s) shared by means of the distributed cache is created only in the servers running the application that uses the shared file(s)

Distributed cache

- In the Driver of the application, the set of shared/cached files are specified
 - By using the job.**addCacheFile**(path) method
 - During the initialization of the job, Hadoop creates a “local copy” of the shared/cached files in all nodes that are used to execute some tasks (mappers or reducers) of the job (i.e., of the running application)
 - The **shared/cache file** is read by the mapper (or the reducer), usually in its **setup method**
 - Since the shared/cached file is available **locally** in the used nodes/servers, its content can be read efficiently
- to be read just one time,
not as many as map
method invocation

Distributed cache

- The **efficiency** of the distributed cache depends on the **number of** multiple **mappers** (or reducers) running on the **same node/server**
 - For each node a local copy of the file is copied during the initialization of the job
 - The **local** copy of the **file** is **used by all mappers** (reducers) running on the **same node/server**
- **Without the distributed cache**, each mapper (reducer) should read, in the setup method, the shared HDFS file
 - Hence, **more time** is needed because reading **data from HDFS** is more inefficient than reading data from the local file system of the node running the mappers (reducers)

Distributed cache

Distributed cache: driver

```
public int run(String[] args) throws Exception {  
    ....  
  
    // Add the shared/cached HDFS file in the  
    // distributed cache  
    job.addCacheFile(new Path("hdfs  
path/filename").toUri());  
  
    ....  
}  
}
```

Distributed cache: mapper/reducer

```
protected void setup(Context context) throws IOException,  
InterruptedException {  
    .....  
    String line;  
  
    // Retrieve the (original) paths of the distributed files  
    URI[] urisCachedFiles = context.getCacheFiles();
```

Distributed cache: mapper/reducer

```
// Read the content of the cached file and process it.  
// In this example the content of the first shared file is opened.  
BufferedReader file = new BufferedReader(new FileReader(  
    new File(new Path(urisCachedFiles[0].getPath()).getName())));  
  
// Iterate over the lines of the file  
while ((line = file.readLine()) != null) {  
    // process the current line  
    ....  
}  
  
file.close();  
}
```

Distributed cache: mapper/reducer

```
// Read the content of the cached file and process it.  
// In this example the content of the first shared file is opened.  
BufferedReader file = new BufferedReader(new FileReader(  
    new File(new Path(urisCachedFiles[0].getPath()).  
        getName())));  
  
// Iterate over the lines of the file  
while ((line = file.readLine()) != null) {  
    // process the current line  
    ....  
}  
  
file.close();  
}
```

Retrieve the name of the file.
The shared file is stored in the root of a local temporary folder (one for each server that is used to run the application) associated with the distributed cache.
The path of the original folder is different from the one used to store the local copy of the shared file.

Part II

MapReduce patterns

Data Organization Patterns

Data Organization Patterns

- Are used to reorganize/split in subsets the input data
 - Binning
 - Shuffling
- The output of an application based on an organization pattern is usually the input of another application(s)

Data Organization Patterns

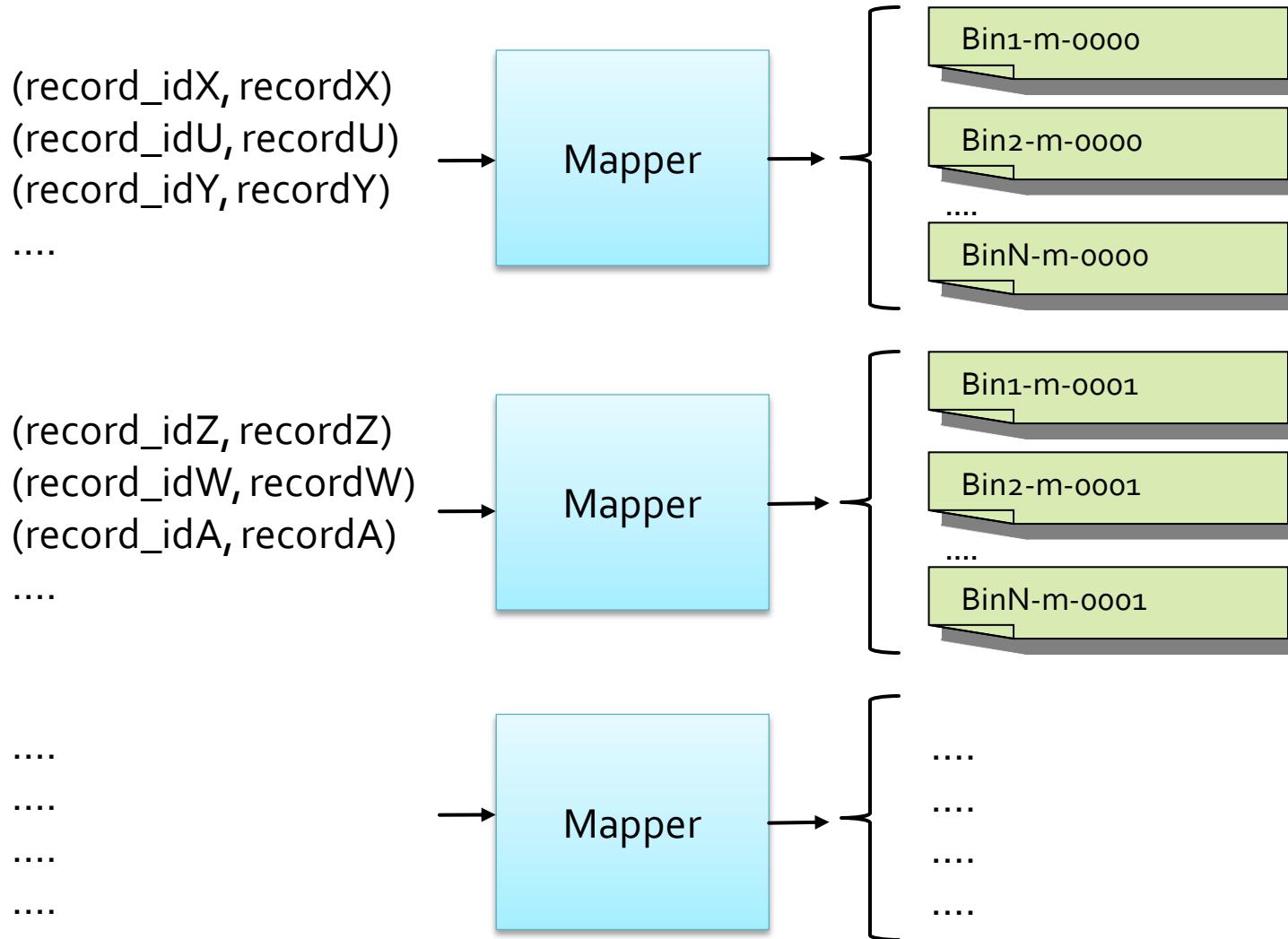
Binning

- Goal
 - Organize/move the input records into categories
- Intent
 - Partition a big data set into distinct, smaller data sets (“bins”) containing similar records
 - Each partition is usually the input of a following analysis
- Motivation
 - The input data set contains heterogeneous data, but each data analysis usually is focused only on a specific subsets of your data

Binning - structure

- Based on a Map-only job
- Driver
 - Sets the list of “bins/output files” by means of MultipleOutputs
- Mappers
 - For each input (key, value) pair, select the output bin/file associated with it and emit a (key,value) in that file
 - key of the emitted pair = key of the input pair
 - value of the emitted pair = value of the input pair
- No combiner or reducer is used in this pattern

Binning - structure



Data Organization Patterns

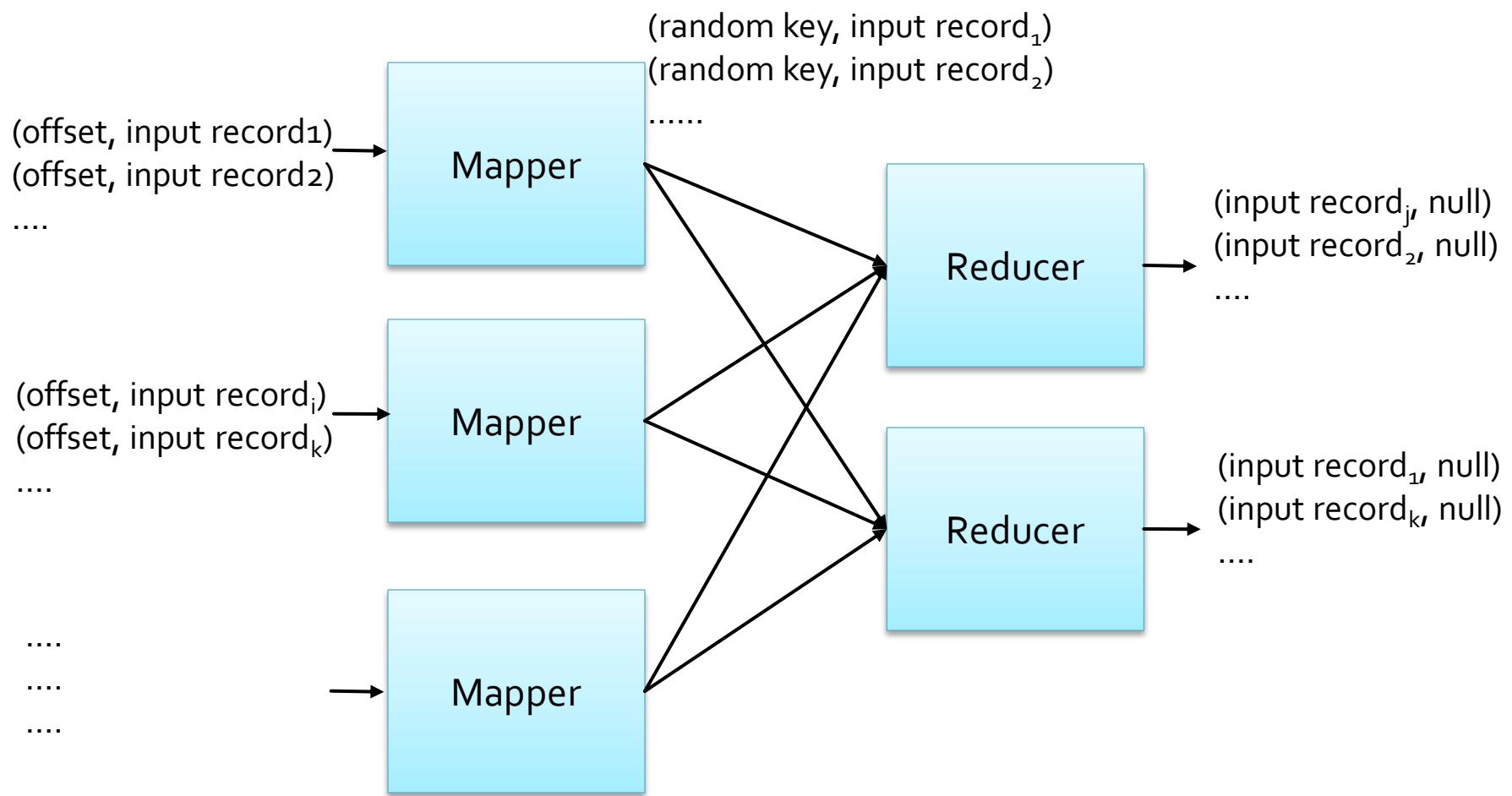
Shuffling

- Goal
 - Randomize the order of the data (records)
- Motivation
 - Randomize the order of the data
 - For anonymization reasons
 - For selecting a subset of random data (records)

Shuffling - structure

- Mappers
 - Emit one (key, value) for each input record
 - key is a random key (i.e., a random number)
 - value is the input record
- Reducers
 - Emit one (key, value) pair for each value in [list-of-values] of the input (key, [list-of-values]) pair

Shuffling - structure



Metapatterns

Metapatterns

- Are used to organize the workflow of a complex application executing many jobs
 - Job Chaining

Metapatterns

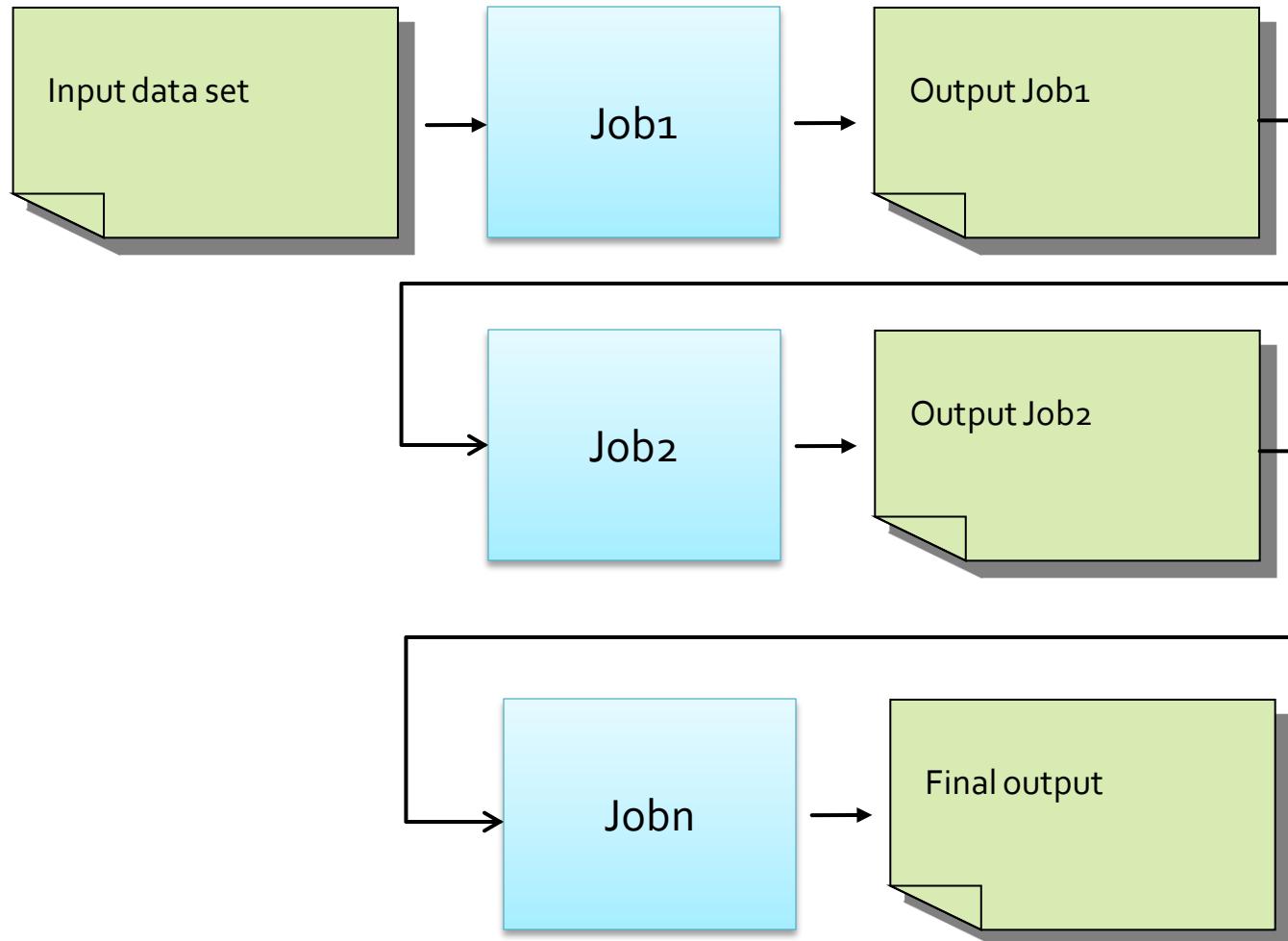
Job Chaining

- Goal
 - Execute a sequence of jobs (synchronizing them)
- Intent
 - Manage the workflow of complex applications based on many phases (iterations)
 - Each phase is associated with a different MapReduce Job (i.e., one sub-application)
 - The output of a phase is the input of the next one
- Motivation
 - Real application are usually based on many phases

Job Chaining - structure

- The (single) Driver
 - Contains the workflow of the application
 - Executes the jobs in the proper order
- Mappers, reducers, and combiners
 - Each phase of the complex application is implemented by a MapReduce Job
 - i.e., it is associated with a mapper, a reducer (and a combiner if it is useful)

Job Chaining - structure



Complex workflow

- More complex workflows, which execute jobs in parallel, can also be implemented
- However, the synchronization of the jobs become more complex

Join Patterns

Join Patterns

- Are used to implement the join operators of the relational algebra (i.e., the join operators of traditional relational databases)
 - Reduce side join
 - Map side join

Join Patterns

- We will focus on the natural join
- However, the pattern is analogous for the other types of joins (theta-, semi-, outer-join)

Join Patterns

Reduce side natural join

- Goal
 - Join the content of two relations (i.e., relational tables)
 - Both tables are large
- Motivation
 - The join operation is useful in many applications

Reduce side natural join - structure

- There are two mapper classes
 - One mapper class for each table
- Mappers
 - Emit one (key, value) pair for each input record
 - Key is the value of the common attribute(s)
 - Value is the concatenation of the name of the table of the current record and the content of the current record

Reduce side natural join - structure

- Suppose you want to join the following tables
 - *Users* with schema userid, name, surname
 - *Likes* with schema userid, movieGenre
- The record
 - userid=u₁, name=Paolo, surname=Garza of the Users table will generate the pair
 - (userid=u₁, "Users:name=Paolo,surname=Garza")
- While the record
 - userid=u₁, movieGenre=horror of the Likes table will generate the pair
 - (userid=u₁, "Likes:movieGenre=horror")

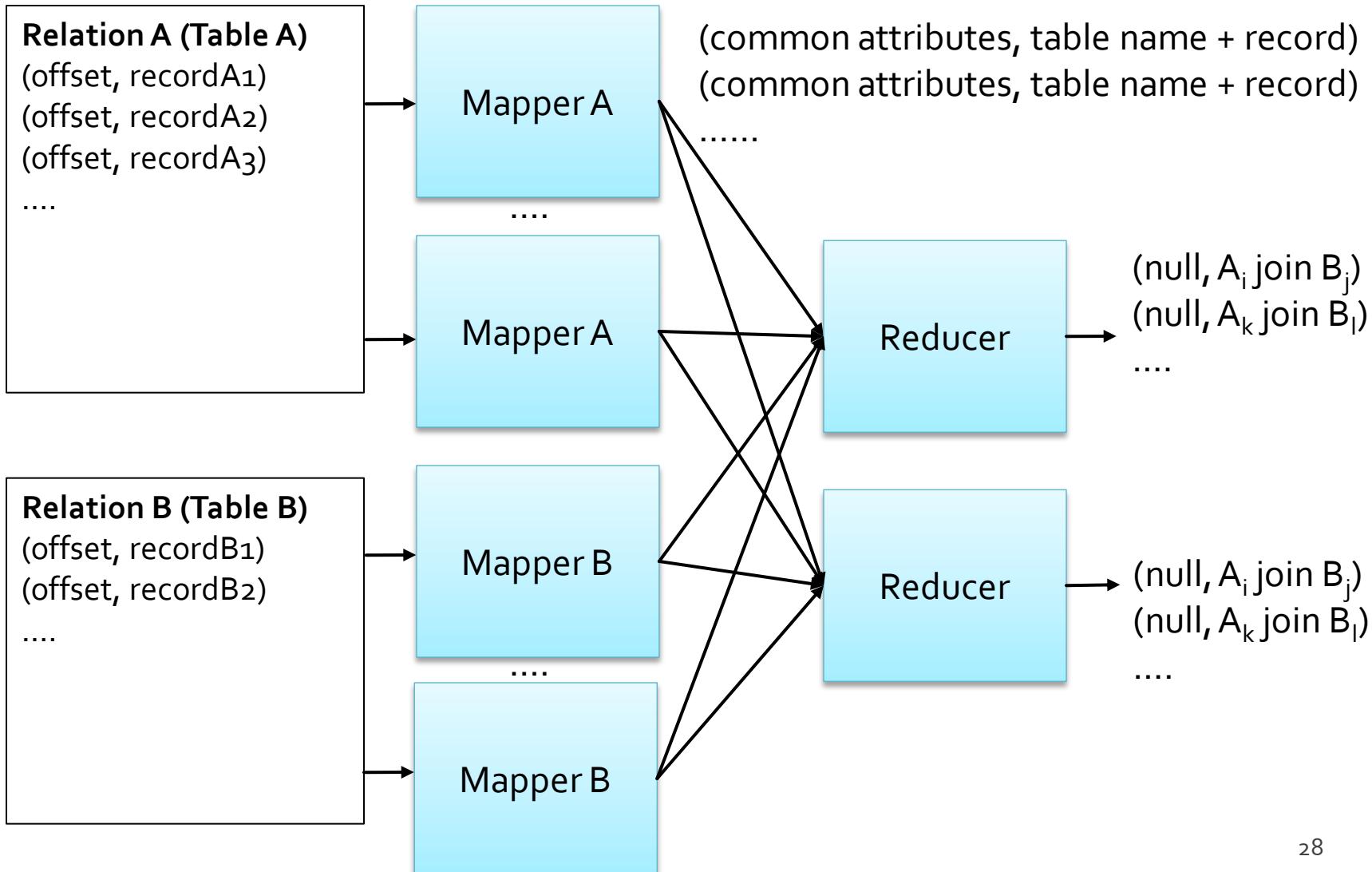
Reduce side natural join - structure

- Reducers
 - Iterate over the values associated with each key (value of the common attributes) and compute the “local natural join” for the current key
 - Generate a copy for each pair of values such that one record is a record of the first table and the other is the record of the other table

Reduce side natural join - structure

- For instance, the (key, [list of values]) pair
 - (`userid=u1`,`["User:name=Paolo,surname=Garza", "Likes:movieGenre=horror", "Likes:movieGenre=adventure"]`) will generate the following output (key,value) pairs
 - (`userid=u1`,`"name=Paolo,surname=Garza,genre=horror"`)
 - (`userid=u1`,`"name=Paolo,surname=Garza,genre=adventure"`)

Reduce side natural join - structure



Join Patterns

Map side natural join

- Goal
 - Join the content of two relations (i.e., relational tables)
 - One table is large
 - The other is small enough to be completely loaded in main memory
- Motivation
 - The join operation is useful in many applications and frequently one of the two tables is small

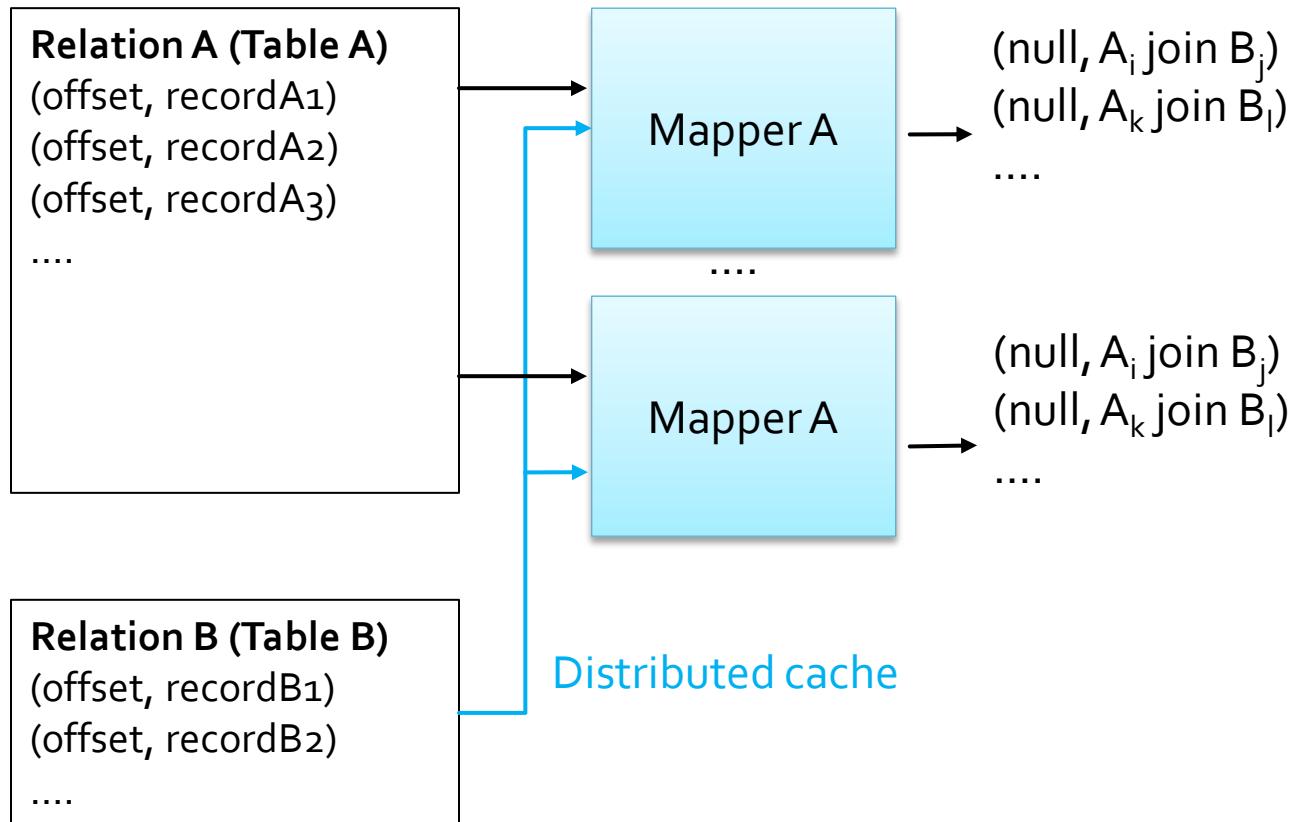
Map side natural join - structure

- Map-only job
- Mapper class
 - Processes the content of the large table
 - Receives one input (key,value) pair for each record of the large table and joins it with the “small” table
- The distributed cache approach is used to “provide” a copy of the small table to all mappers

Map side natural join - structure

- Each mapper
 - Performs the “local natural join” between the current record (of the large table) it is processing and the records of the small table (that is in the distributed cache)
 - The content of the small table (file) is loaded in the main memory of each mapper during the execution of its setup method

Map side natural join - structure



Join Patterns

Theta-join, Semi-join, Outer-join

- The SQL language is characterized by many types of joins
 - Theta-join
 - Semi-join
 - Outer-join
- The same patterns used for implementing the natural join can be used also for the other SQL joins
 - The “local join” in the reducer of the reduce side natural join (in the mapper of the map side natural join) is substituted with the type of join of interest (theta-, semi-, or outer-join)

Relational Algebra Operations and MapReduce

Relational Algebra Operators

- The relational algebra and the SQL language have many useful operators
 - Selection
 - Projection
 - Union, intersection, and difference
 - Join (see Join design patterns)
 - Aggregations and Group by (see the Summarization design patterns)

Relational Algebra Operators

- The MapReduce paradigm can be used to implement relational operators
 - However, the MapReduce implementation is efficient only when a full scan of the input table(s) is needed
 - i.e., when queries are not selective and process all data
 - Selective queries, which return few tuples/records of the input tables, are usually not efficient when implemented by using a MapReduce approach

Relational Algebra Operators

- Most preprocessing activities involve relational operators
 - E.g., ETL processes in the data warehousing application context

Relations/Tables

- Relations/Tables (also the big ones) can be stored in the HDFS distributed file system
 - They are broken in blocks and spread across the servers of the Hadoop cluster

Relations/Tables

- Note
 - In relational algebra, relations/tables do not contain duplicate records by definition
 - This constraint must be satisfied by both the input and the output relations/tables

Selection

- $\sigma_C(R)$
 - Applies predicate (condition) C to each record of table R
 - Produces a relation containing only the records that satisfy predicate C
- The selection operator can be implemented by using the filtering pattern

Selection

| Courses | <u>CCode</u> | CName | Semester | ProfID |
|---------|--------------|------------------|----------|--------|
| | M2170 | Computer science | 1 | D102 |
| | M4880 | Digital systems | 2 | D104 |
| | F1401 | Electronics | 1 | D104 |
| | F0410 | Databases | 2 | D102 |

- Find the courses held in the second semester
- $\sigma_{\text{Semester}=2}(\text{Courses})$

Selection

Courses

| <u>CCode</u> | CName | Semester | ProfID |
|--------------|------------------------|----------|-------------|
| M2170 | Computer science | 1 | D102 |
| <i>M4880</i> | <i>Digital systems</i> | 2 | <i>D104</i> |
| F1401 | Electronics | 1 | D104 |
| <i>F0410</i> | <i>Databases</i> | 2 | <i>D102</i> |



Result

| <u>CCode</u> | CName | Semester | ProfID |
|--------------|-----------------|----------|--------|
| M4880 | Digital systems | 2 | D104 |
| F0410 | Databases | 2 | D102 |

Selection

- Map-only job
- Each mapper
 - Analyzes one record at a time of its split
 - If the record satisfies C then it emits a (key,value) pair with key=record and value=null
 - Otherwise, it discards the record

Projection

- $\pi_S(R)$
 - For each record of table R, keeps only the attributes in S
 - Produces a relation with a schema equal to S (i.e., a relation containing only the attributes in S)
 - Removes duplicates, if any

Projection

Professors

| <u>ProfId</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |
| D105 | Jones | Computer engineering |
| D104 | Smith | Electronics |

- Find the surnames of all professors
- $\pi_{\text{PSurname}}(\text{Professors})$

Projection

Professors

| <u>ProfId</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |
| D105 | Jones | Computer engineering |
| D104 | Smith | Electronics |



Result

| PSurname |
|----------|
| Smith |
| Jones |

- Duplicated values are removed

Projection

- Each mapper
 - Analyzes one record at a time of its split
 - For each record r in R
 - It selects the values of the attributes in S and constructs a new record r'
 - It emits a $(key, value)$ pair with $key=r'$ and $value=null$
- Each reducer
 - Emits one $(key, value)$ pair for each input $(key, [list\ of\ values])$ pair with $key=r'$ and $value=null$

Union

- $R \cup S$
 - R and S have the same schema
 - Produces a relation with the same schema of R and S
 - There is a record t in the output of the union operator for each record t appearing in R or S
 - Duplicated records are removed

Union

DegreeCourseProf

| <u>ProfID</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |
| D105 | Jones | Computer engineering |
| D104 | White | Electronics |

MasterCourseProf

| <u>ProfID</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |
| D101 | Red | Electronics |

- Find information relative to the professors of degree courses or master's degrees
- $\text{DegreeCourseProf} \cup \text{MasterCourseProf}$

Union

DegreeCourseProf

| <u>ProfID</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |
| D105 | Jones | Computer engineering |
| D104 | White | Electronics |

MasterCourseProf

| <u>ProfID</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |
| D101 | Red | Electronics |



Result

| <u>ProfID</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |
| D105 | Jones | Computer engineering |
| D104 | White | Electronics |
| D101 | Red | Electronics |

Union

- Mappers
 - For each input record t in R , emit one (key, value) pair with key= t and value=null
 - For each input record t in S , emit one (key, value) pair with key= t and value=null
- Reducers
 - Emit one (key, value) pair for each input (key, [list of values]) pair with key= t and value=null
 - i.e., one single copy of each input record is emitted

Intersection

- $R \cap S$
 - R and S have the same schema
 - Produces a relation with the same schema of R and S
 - There is a record t in the output of the intersection operator if and only if t appears in both relations (R and S)

Intersection

DegreeCourseProf

| <u>ProfID</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |
| D105 | Jones | Computer engineering |
| D104 | White | Electronics |

MasterCourseProf

| <u>ProfID</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |
| D101 | Red | Electronics |

- Find information relative to professors teaching both degree courses and master's courses
- DegreeCourseProf \cap MasterCourseProf

Intersection

DegreeCourseProf

| <u>ProfID</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |
| D105 | Jones | Computer engineering |
| D104 | White | Electronics |

MasterCourseProf

| <u>ProfID</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |
| D101 | Red | Electronics |

Result

| <u>ProfID</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |



Intersection

- Mappers
 - For each input record t in R , emit one (key, value) pair with key= t and value="R"
 - For each input record t in S , emit one (key, value) pair with key= t and value="S"

Intersection

- Reducers
 - Emit one (key, value) pair with key= t and value=null for each input (key, [list of values]) pair with [list of values] containing two values
 - It happens if and only if both R and S contain t

Difference

- $R - S$
 - R and S have the same schema
 - Produces a relation with the same schema of R and S
 - There is a record t in the output of the difference operator if and only if t appears in R but not in S

Difference

DegreeCourseProf

| <u>ProfID</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |
| D105 | Jones | Computer engineering |
| D104 | White | Electronics |

MasterCourseProf

| <u>ProfID</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |
| D101 | Red | Electronics |

- Find the professors teaching degree courses but not master's courses
- DegreeCourseProf - MasterCourseProf

Difference

DegreeCourseProf

| <u>ProfID</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |
| D105 | Jones | Computer engineering |
| D104 | White | Electronics |

MasterCourseProf

| <u>ProfID</u> | PSurname | Department |
|---------------|----------|----------------------|
| D102 | Smith | Computer engineering |
| D101 | Red | Electronics |

Result

| <u>ProfID</u> | PSurname | Department |
|---------------|----------|----------------------|
| D105 | Jones | Computer engineering |
| D104 | White | Electronics |



Difference

- Mappers
 - For each input record t in R , emit one (key, value) pair with key= t and value=name of the relation (i.e., R)
 - For each input record t in S , emit one (key, value) pair with key= t and value=name of the relation (i.e., S)
- Two mapper classes are needed
 - One for each relation

Difference

- Reducers
 - Emit one (key, value) pair with key=t and value=null for each input (key, [list of values]) pair with [list of values] containing only the value R
 - It happens if and only if t appears in R but not in S

Join

- The join operators can be implemented by using the Join pattern
 - By using the reduce side or the map side pattern depending on the size of the input relations/tables

Aggregations and Group by

- Aggregations and Group by are implemented by using the Summarization pattern