

## MapReduce and Hadoop

### Corso di Sistemi e Architetture per Big Data

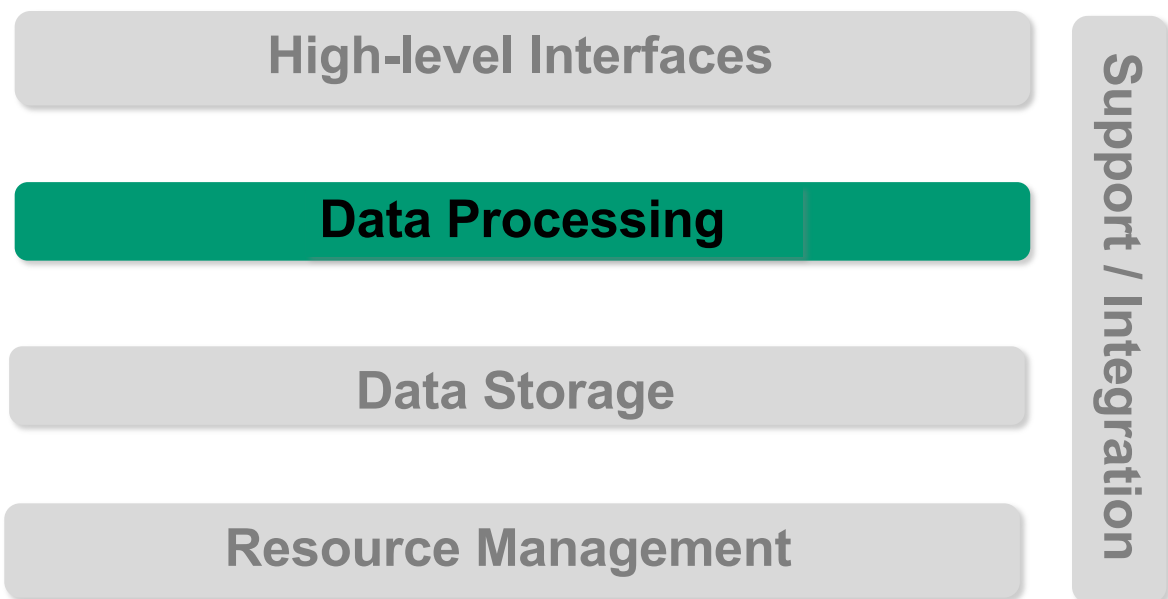
A.A. 2025/26

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

### The reference Big Data stack

---



---

# MapReduce

*The beauty of MapReduce is that any programmer can understand it, and its power comes from being able to harness thousands of computers behind that simple interface.*

David Patterson

---

## MapReduce

- **Programming model** for processing huge amounts of data sets over thousands of servers
  - Proposed by Google in 2004: [MapReduce: simplified data processing on large clusters](#)
  - Based on **shared nothing** approach
- Also associated **implementation** (framework) of the distributed system
  - Not released by Google
- Examples of applications in Google
  - Web indexing
  - Reverse Web-link graph
  - Distributed sort
  - Web access statistics

# MapReduce: programmer view

---

- MapReduce **hides system-level details**
  - Key idea: separate the *what* from the *how*
  - MapReduce abstracts away the “distributed” part of the system
  - Such details are handled by the framework
- Programmers get simple API
  - Don't have to worry about handling
    - Parallelization
    - Data distribution
    - Load balancing
    - Fault tolerance

## Typical Big Data problem

---

- Iterate over a large number of elements (e.g., tuples, documents)
- Extract something of interest from each element
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

**Map**

**Reduce**

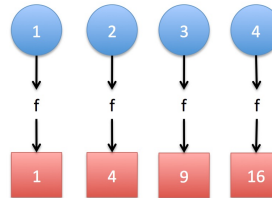
**Key idea: provide a functional abstraction of the two Map and Reduce operations**

# Your first MapReduce example (in Lisp)

- *Example*: sum-of-squares (sum the square of numbers from 1 to n) in MapReduce fashion

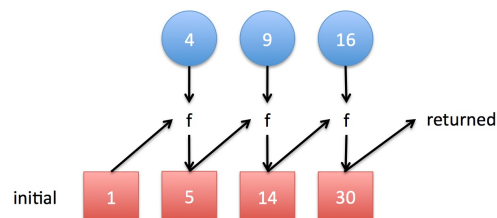
- Map function:

map square [1,2,3,4]  
returns [1,4,9,16]



- Reduce function:

reduce [1,4,9,16]  
returns 30 (sum of square elements)



## MapReduce: model

- Processing occurs in two phases: **Map** and **Reduce**
  - Functional programming roots (e.g., Lisp)
- Input and output: set of **key-value pairs**
- Programmers specify two functions: Map and Reduce

**map** $(k_1, v_1) \rightarrow [(k_2, v_2)]$

**reduce** $(k_2, [v_2]) \rightarrow [(k_3, v_3)]$

- $(k, v)$  denotes a (key, value) pair
- [...] denotes a list
- Keys do not have to be unique: different pairs can have the same key
- Keys of input elements ( $k_1$ ) are not relevant
- Output keys of reduce ( $k_3$ ) are often identical to input keys of reduce ( $k_2$ )

# Map

---

- Execute a function on a set of key-value pairs (input shard) to create a new list of key-value pairs

**map (input\_key, input\_value) →  
list(output\_key, intermediate\_value)**

- Map tasks are distributed across machines by automatically partitioning input data into *shards*
  - Parallelism is achieved as keys can be simultaneously processed by different map tasks
- MapReduce system groups together all intermediate values associated with the same intermediate key and passes them to Reduce tasks

# Reduce

---

- Combine values in sets to create a new value  
**reduce (output\_key, list(intermediate\_value)) →  
list(output\_key, output\_value)**
  - Parallelism is achieved as Reduce tasks operating on different keys can be executed simultaneously

# MapReduce program

---

- A MapReduce program, referred to as a *job*, consists of:
  - Code for Map and Reduce
  - Configuration parameters (where input lies, where output will be stored)
  - Input data set, stored on underlying distributed file system
    - Input does not fit on a single computer's disk
- Each MapReduce job is divided by system into smaller units called *tasks*
  - *Map tasks* or *mappers*
  - *Reduce tasks* or *reducers*
  - All mappers need to finish before reducers can begin
- Output of MapReduce job is also stored on distributed file system
- A MapReduce program may consist of *many rounds* of different map and reduce functions

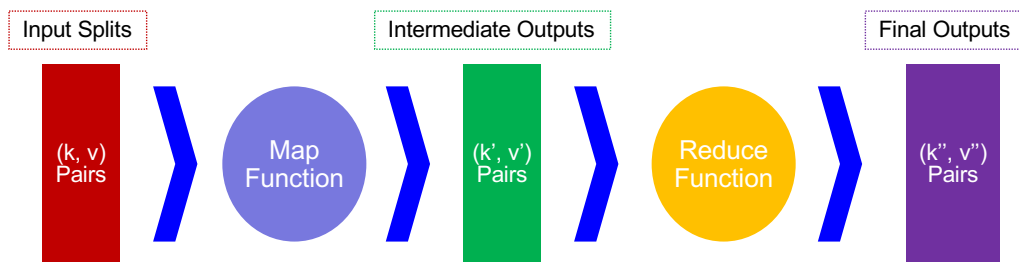
# MapReduce computation

---

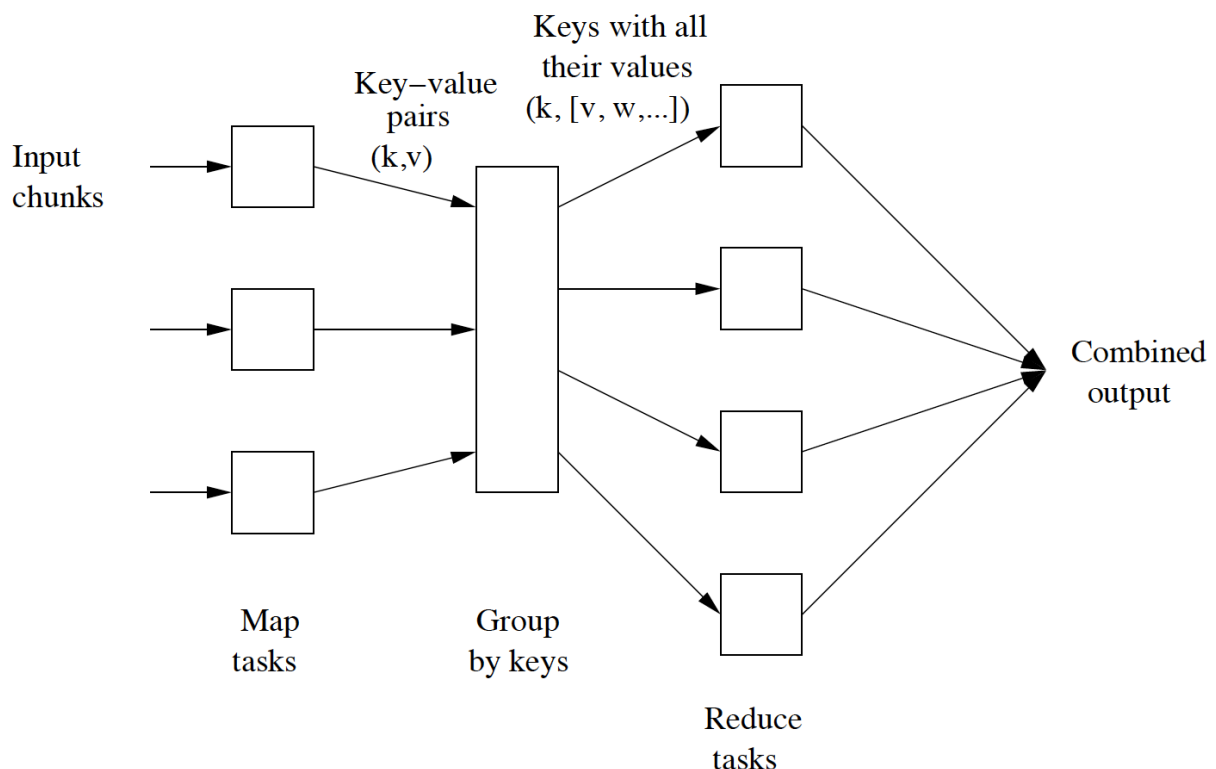
1. Some number of *Map tasks* each are given in input one or more *chunks of data* from *distributed file system*
2. These Map tasks turn the chunk into a sequence of *key-value pairs*
  - The way key-value pairs are produced from input data is determined by the code written by the programmer for the Map function
3. The key-value pairs from each Map task are collected by the *master controller* and sorted by key
4. The keys are divided among all the *Reduce tasks*, so *all key-value pairs with the same key* wind up at the same Reduce task
5. The Reduce tasks work on one key at a time, and combine all the values associated with that key in some way
  - The manner of combination of values is determined by the code written by the programmer for the Reduce function
6. Output key-value pairs from each reducer are written persistently back onto the distributed file system
7. Output ends up in  $r$  files, where  $r$  is the number of reducers
  - Such output may be the input to a subsequent MapReduce phase

# Where the magic happens

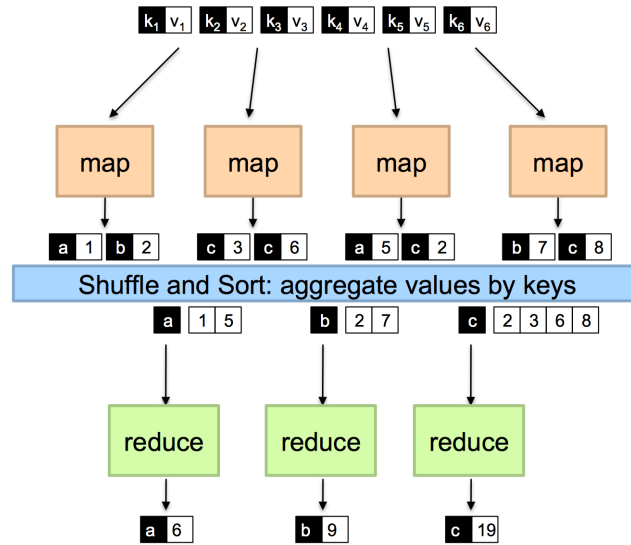
- Implicit between Map and Reduce phases is a distributed *group by* operation on intermediate keys, called **Shuffle and Sort**
  - Transfer data from mappers to reducers, sorting and merging mappers' intermediate output
  - Intermediate data arrives at each reducer *sorted by key*
- Intermediate data is transient
  - Not stored on distributed file system, rather “*spilled*” to local disk of each machine



# MapReduce computation: the complete picture



# Simplified view of MapReduce: example



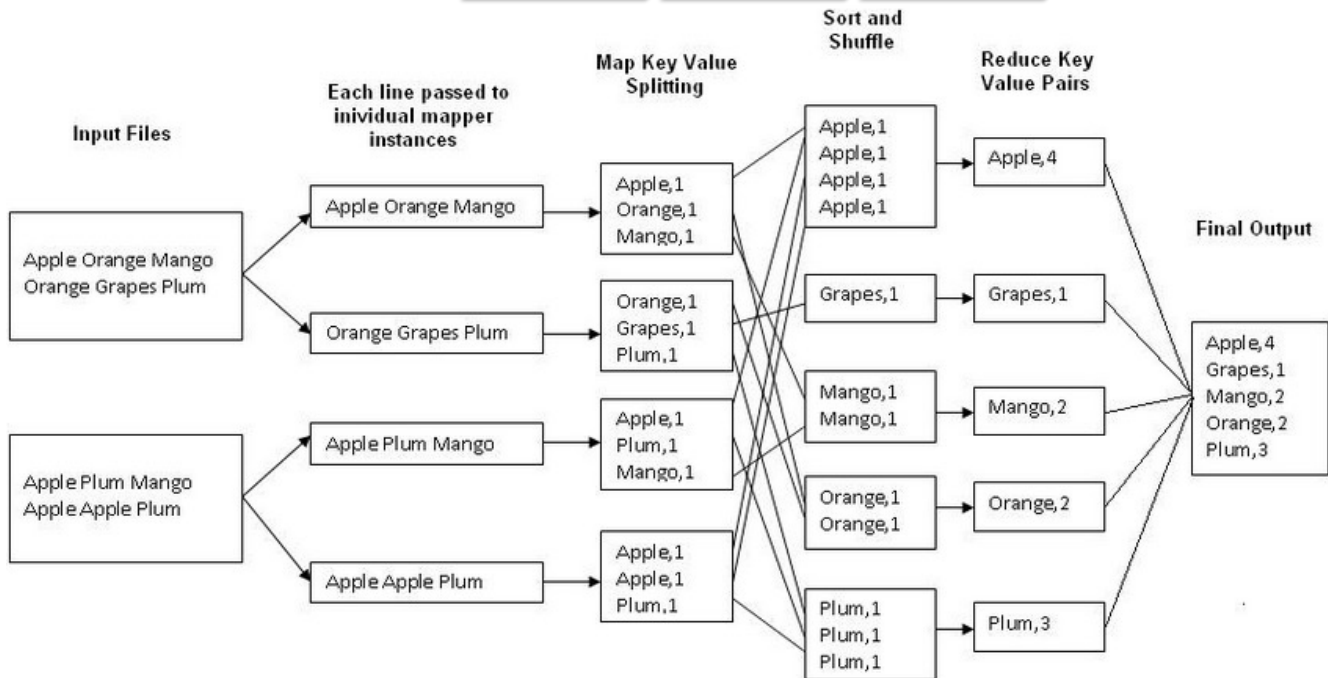
- Mappers are applied to input key-value pairs and generate an arbitrary number of intermediate key-value pairs
- Reducers are applied to all intermediate values associated with the same intermediate key
- Between Map and Reduce lies a barrier (Shuffle and Sort) that involves a large distributed sort and group by

## “Hello World” in MapReduce: WordCount

- **Problem:** count the number of occurrences for each word in a large collection of documents
- **Input:** repository of documents, each document is an element
- **Map:** read a document and emit a sequence of key-value pairs where:
  - Keys are words of the documents and values are equal to 1:  
 $(w_1, 1), (w_2, 1), \dots, (w_n, 1)$
- **Shuffle and sort:** group by key and generate pairs of the form  $(w_1, [1, 1, \dots, 1]), \dots, (w_n, [1, 1, \dots, 1])$
- **Reduce:** add up all the values and emit  $(w_1, k), \dots, (w_n, l)$
- **Output:**  $(w, m)$  pairs where:
  - $w$  is a word that appears at least once among all the input documents and  $m$  is the total number of occurrences of  $w$  among all those documents

# WordCount in practice

## Map Shuffle Reduce



## Example: WordLengthCount

- **Problem:** count how many words of certain lengths exist in a collection of documents
- **Input:** a repository of documents, each document is an element
- **Map:** read a document and emit a sequence of key-value pairs where the key is the word length and the value is the word itself:

$$(i, w_1), \dots, (j, w_n)$$

- **Shuffle and sort:** group by key and generate pairs of the form

$$(1, [w_1, \dots, w_k]), \dots, (n, [w_r, \dots, w_s])$$

- **Reduce:** count the number of words in each list and emit:

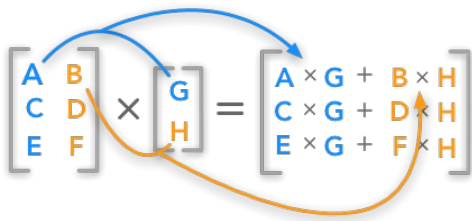
$$(1, l), \dots, (p, m)$$

- **Output:**  $(l, n)$  pairs, where  $l$  is a length and  $n$  is the total number of words of length  $l$  in the input documents

## Example: matrix-vector multiplication

---

- Sparse matrix  $A = [a_{ij}]$  size  $n \times n$
- Vector  $x = [x_j]$  size  $n \times 1$
- **Problem:** matrix-vector multiplication  $y = Ax$  where  $y_i = \sum_{j=1 \dots n} a_{ij}x_j$ 
  - Used in many algorithms, e.g., PageRank



$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \times \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \times 2 + 2 \times 4 \\ 3 \times 2 + 4 \times 4 \\ 5 \times 2 + 6 \times 4 \end{bmatrix} = \begin{bmatrix} 10 \\ 22 \\ 34 \end{bmatrix}$$

## Example: matrix-vector multiplication

---

- Assume that  $x$  can fit into main memory of each mapper
- **Map:** apply to  $((i, j), a_{ij})$  and produce key-value pair  $(i, a_{ij}x_j)$
- **Reduce:** receive  $(i, [a_{i1}x_1, \dots, a_{in}x_n])$  as input and sum all values of the list associated with a given key  $i$ , i.e.,  $y_i = \sum_{j=1 \dots n} a_{ij}x_j$ . The result will be a pair  $(i, y_i)$

## Example: matrix-vector multiplication

---

- What happens if  $x$  cannot fit in mapper's memory?
- *Solution:*
  - Split  $x$  in horizontal **stripes** fitting in memory
  - Split  $A$  accordingly in vertical stripes, stripes of  $A$  do not need to fit in memory

$$\begin{array}{c} \left[ \begin{array}{c} | \\ | \\ | \\ | \\ | \end{array} \right] \\ y \end{array} = \begin{array}{c} \left[ \begin{array}{c|c|c|c|c} | & | & \dots & | & | \\ \hline | & | & \dots & | & | \end{array} \right] \\ A \end{array} \begin{array}{c} \left[ \begin{array}{c} | \\ | \\ | \\ | \\ | \end{array} \right] \\ x \end{array}$$

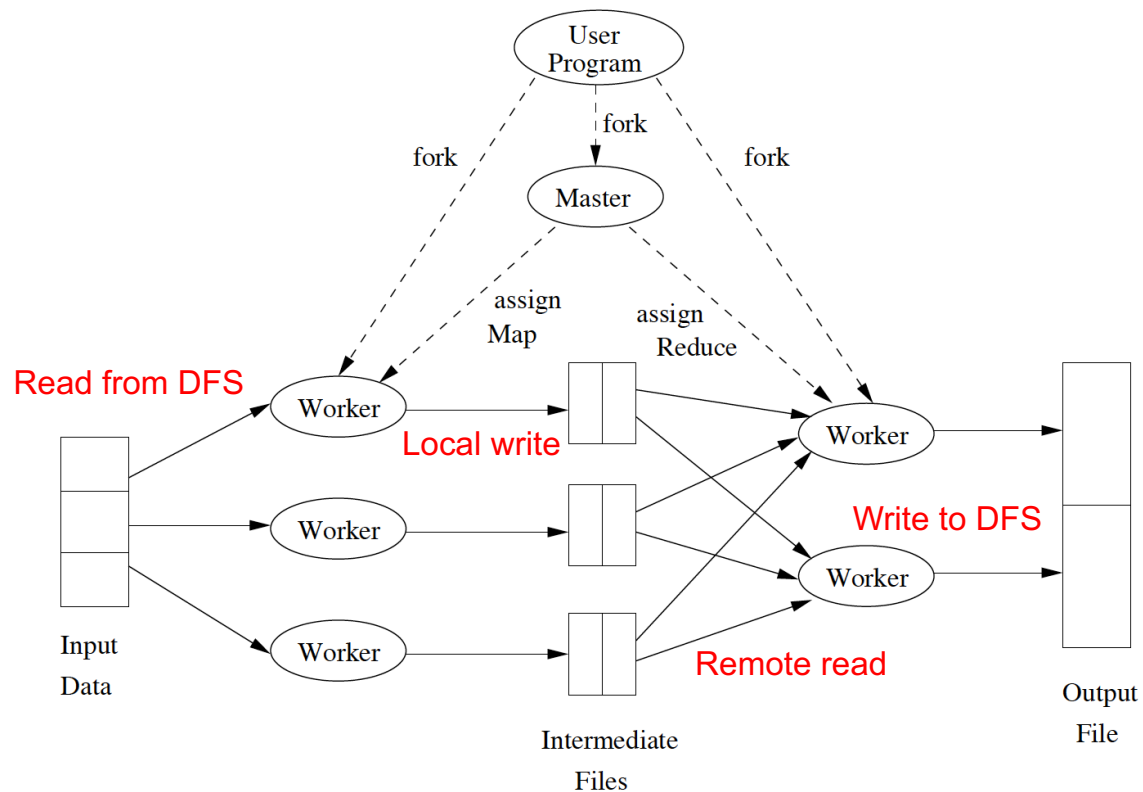
- Each mapper is assigned a matrix stripe; it also gets the corresponding vector stripe
  - Map and Reduce functions are as before
- A more efficient solution can be designed by partitioning  $A$  into **square blocks** rather than stripes

## MapReduce: execution overview

---

- Master-worker architecture
- Master coordinates map and reduce tasks controlling the flow of MapReduce job
  - Assigns (i.e., schedules) job tasks to workers, monitoring them and re-executing failed tasks
- Workers execute map and reduce tasks

# MapReduce: execution overview



## Coping with failures

- **Node hosting the master fails**
  - The entire MapReduce job must be restarted
  - The worst scenario
- **Worker node hosting a mapper fails**
  - All the map tasks that were assigned to this node will have to be redone on another node, even if they had completed, because the disk(s) of the failed node is inaccessible
- **Worker node hosting a reducer fails**
  - Reschedule reducer on another worker node

## Optimization: combining

---

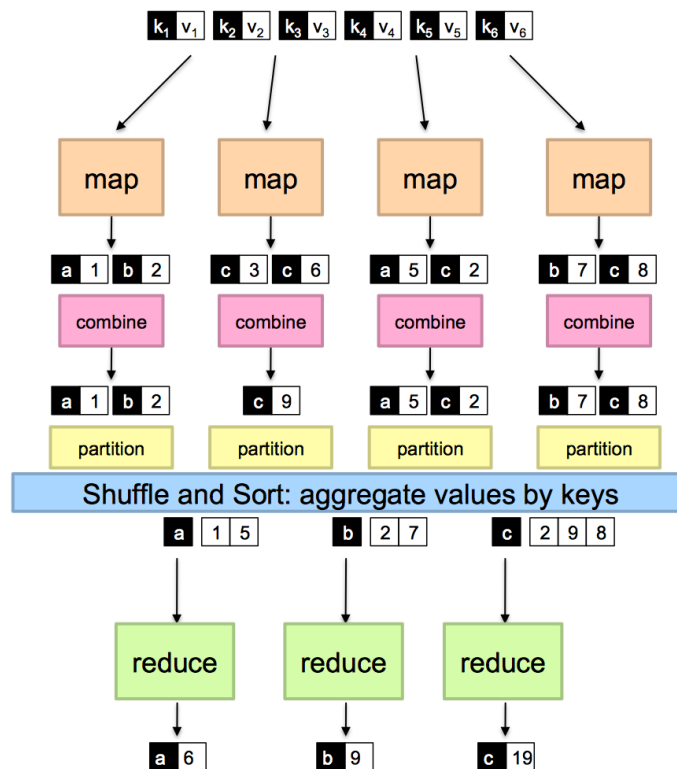
- How to improve performance?
- Run a **mini reduce phase on local map output**, thus pushing some of what the reducers do to the preceding mappers
- Let's apply a **combiner** to local Map output  
**combine**  $(k_2, [v_2]) \rightarrow [(k_3, v_3)]$
- But Reduce function needs to be associative and commutative
  - Values to be combined can be combined in any order, with the same result
  - E.g., addition in WordCount's Reduce

## Optimization: combining

---

- In many cases the same function can be used for combining as the final reduction
- But shuffle and sort is still necessary!
- Pros:
  - Reduce amount of intermediate data
  - Reduce network traffic

# WordCount with combiners



# WordCount with combiners

- **Problem:** count the number of occurrences for each word in a large collection of documents
- **Input:** repository of documents, each document is an element
- **Map:** read a document and emit a sequence of key-value pairs where:
  - Keys are words of the documents and values are equal to 1:  
 $(w_1, 1), (w_2, 1), \dots, (w_n, 1)$
- **Combiner:** group by key, add up all the values and emit:  
 $(w_1, i), \dots, (w_n, j)$
- **Shuffle and sort:** group by key and generate pairs of the form  $(w_1, [p, \dots, q]), \dots, (w_n, [r, \dots, s])$
- **Reduce:** add up all the values and emit  $(w_1, k), \dots, (w_n, l)$
- **Output:**  $(w, m)$  pairs where:
  - $w$  is a word that appears at least once among all the input documents and  $m$  is the total number of occurrences of  $w$  among all those documents

## Shuffle and sort

---

- Between Map(+combine) and Reduce phases
  - Data is **shuffled**: parallel-sorted and exchanged
  - Data is **moved** to the correct reducer
- Parallel sort: on mappers
  - Problem: key-value pairs must be sorted by key, but dataset is too large to be sorted on one machine
  - Solution: perform sorting in stages
  - Each mapper partitions its output by reducer, based on the hash of the key, and writes each partition to a file (sorted by key) on its local disk

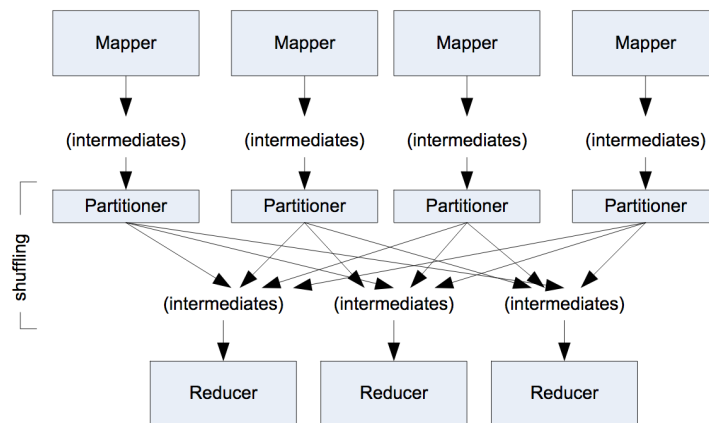
## Shuffle and sort

---

- Move data from mappers to reducers
  - Whenever a mapper finishes writing its files, it notifies the master; each reducer periodically asks the master which mappers to connect
  - Each reducer connects to its mappers and gets the files of sorted key-value pairs
  - For fault tolerance, mappers do not delete their files as soon as the reducer has retrieved them, but wait until the master tells them to delete the files
- Merge: on reducers
  - Each reducer merges the files from mappers together, preserving their sort ordering
  - Then it starts reducing on the merged input, invoking the reduce function for each key in its input

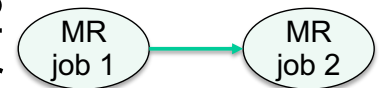
## Optimization: partitioning

- How to divide the intermediate key space in a custom way?
- Through a **partitioner**
  - Assigns intermediate key-value pairs to reducers



## MapReduce workflows

- Few problems can be solved using a single MapReduce job
- Example of job pipeline: to find the most visited URLs in a logfile we need 2 MapReduce jobs chained together
  - 1<sup>st</sup> MR job: count number of visits per URL
    - Like WordCount: mappers emit (URL, 1) key-value pairs; reducers aggregate URL counts
  - 2<sup>nd</sup> MR job: sort URL counts
    - Mappers of 2<sup>nd</sup> job swap keys and values, making counter as key and URL as value
    - Reducers of 2<sup>nd</sup> job run the identity function (i.e., do nothing), because they get in input URLs already sorted by frequency
    - We still need the reducers, why?



# MapReduce workflows

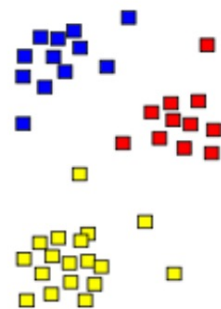
---

- MapReduce jobs can be **chained** together into **workflows**
  - Output of one job becomes input to next job
- Not only job pipeline (i.e., linear chain) but jobs can be also organized in more complex **directed acyclic graph (DAG)**
- Each job generates intermediate files on DFS (written to and read from)
  - Performance drops
- Apache Oozie: framework to manage Hadoop workflows

## Example: *k*-means in MapReduce

---

- Clustering: process of examining a collection of *points* and grouping them into *clusters* according to some distance measure
- Examples of cluster analysis
  - Customer segmentation
  - Stock market clustering
  - Dataset dimensionality reduction



## Distance between points

---

- Preliminary step: select distance metric between data points
- Most popular is **Euclidean distance**

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \dots + (p_n - q_n)^2}$$

where  $n$  is the number of independent variables in the space

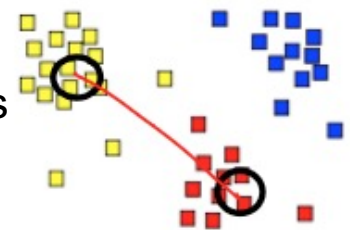
## Distance between clusters

---

- How to define the distance between clusters?

- **Centroid distance**

- Distance between centroids of clusters



- **Centroid** is the point that has the mean position of all data points in each coordinate

- Example: for points  $(-1, 10, 3)$ ,  $(0, 5, 2)$ , and  $(1, 20, 10)$ , the centroid is

$$((-1+0+1)/3, (10+5+20)/3, (3+2+10)/3) = (0, 35/3, 5)$$

## k-means clustering

- **k-means** is a well-known clustering algorithm belonging to **point assignment** class of clustering algorithms
  - Points are considered in some order, and each one is assigned to the cluster into which it best fits
  - Euclidian space
  - $k$  = number of clusters

## k-means clustering

- A variety of heuristic algorithms for  $k$ -means exist
- L'et's consider **Lloyd's algorithm**, the first and simplest
  - Goal: minimize the **within-cluster sum of squares**

Specify desired number of clusters  $k$ ;

Initially choose  $k$  data points that are likely to be in different clusters;

Make these data points the centroids of their clusters;

Repeat

For each remaining data point  $p$  do

Find the centroid to which  $p$  is nearest;

Add  $p$  to the cluster of that centroid;

Re-compute cluster centroids;

Until no improvement is made;

# MapReducing 1 iteration of $k$ -means

---

- **Classify:** assign each point to nearest cluster centroid

$$z_i \leftarrow \arg \min_j \|\boldsymbol{\mu}_j - \mathbf{x}_i\|_2^2$$

$\mathbf{x}_i$ : data point  
 $\boldsymbol{\mu}_j$ : centroid for cluster  $j$   
 $z_i$ : cluster  $i$  label

Map: given  $(\{\boldsymbol{\mu}_j\}, \mathbf{x}_i)$ , for each point  $\mathbf{x}_i$  emit  $(z_i, \mathbf{x}_i)$

Parallel over data points

- **Re-center:** update cluster centroids as mean of assigned points

$$\boldsymbol{\mu}_j = \frac{1}{n_j} \sum_{i:z_i=j} \mathbf{x}_i$$

$n_j$ : number of points in cluster  $j$

Reduce: average over all points in cluster  $j$  ( $z_i=j$ )

Parallel over cluster centroids

## Classification step as Map

---

- Classify: assign each point to nearest cluster centroid

$$z_i \leftarrow \arg \min_j \|\boldsymbol{\mu}_j - \mathbf{x}_i\|_2^2$$

map( $[\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \dots, \boldsymbol{\mu}_k], \mathbf{x}_i$ )

Map on data point and cluster centroids

$$z_i \leftarrow \arg \min_j \|\boldsymbol{\mu}_j - \mathbf{x}_i\|_2^2$$

emit  $(z_i, \mathbf{x}_i)$

Emit  $z_i$  (the cluster label) as key and data point  $\mathbf{x}_i$  as value

## Re-center step as Reduce

---

- Re-center: update cluster centroids as mean of assigned points

$$\mu_j = \frac{1}{n_j} \sum_{i:z_i=j} x_i$$

reduce(j, x\_in\_clusterj: [x<sub>i</sub>, ...])

sum = 0

count = 0

for x in x\_in\_clusterj

sum += x

count += 1

emit (j, sum/count)

Reduce on data points assigned to cluster j (having the cluster label j as key)

Emit cluster label j as key and new centroid for cluster j as value

## Multiple iterations for *k*-means

---

- *k*-means is an iterative algorithm: needs an iterative version of MapReduce
- Our implementation so far: each mapper gets a data point and all cluster centroids
  - ✗ Too many mappers!
- Better implementation: each mapper gets many data points
  - Anyway, at each iteration we must broadcast the new centroids across the MapReduce cluster and repeat multiple phases of Map and Reduce until convergence (or max number of steps)
- Any other optimization?

# Optimizing *k*-means for MapReduce

---

- **Combiners** can be used to optimize the distributed algorithm
  - Compute for each centroid local sums of points
  - Send to reducer: <centroid, partial sums>
- Can use a **single reducer**
  - ✓ Data to reducers is small
  - ✓ Single reducer can tell immediately if computation converges
  - ✓ One output file

---

## Apache Hadoop



# What is Apache Hadoop?

---

- Open-source software framework for reliable, scalable, distributed data-intensive computing
  - Originally developed by Yahoo!
- Goal: storage and processing of data-sets at massive scale
- Infrastructure: cluster of commodity hardware
- Core components:
  - **HDFS**: Hadoop Distributed File System
  - **Hadoop YARN**
  - **Hadoop MapReduce**
- Plus many related projects
  - Apache Pig, Apache Hive, Apache Hbase, ...

## Hadoop runs on clusters

---

- Compute nodes are stored on **racks**
  - 8-64 compute nodes on a rack
- Cluster composed by many racks of compute nodes
- How to assign tasks to compute nodes?
  - Take into account interconnection topology
    - Nodes on same rack are typically connected by 10 Gbit/s Ethernet
    - Racks are interconnected by another level of network or a switch
    - Bandwidth of **intra-rack communication** is **greater** than that of inter-rack communication
- How to deal with failures of compute nodes?
  - Files are stored redundantly
  - Computation is divided into tasks

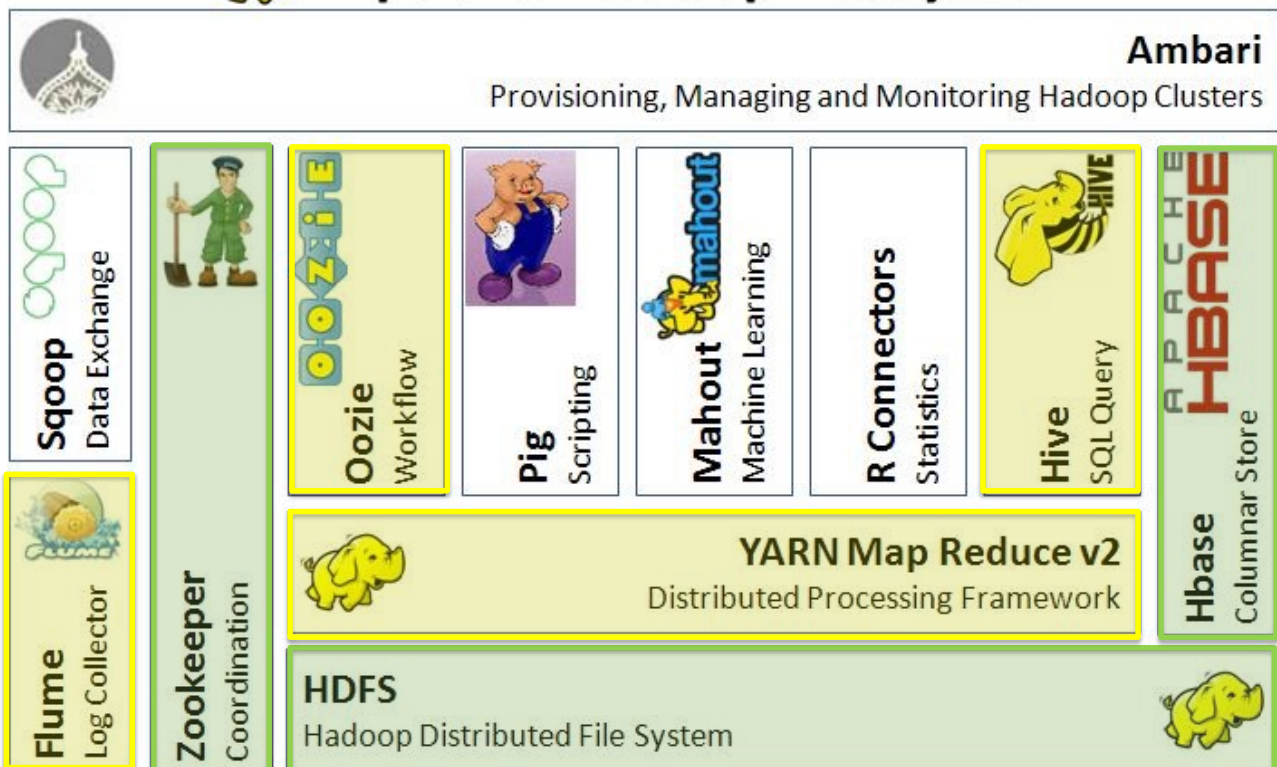


# Hadoop core

- **HDFS**✓
  - Distributed file system
  - Data is replicated across the cluster
  - Fault-tolerant
- **Hadoop YARN**
  - Cluster resource management
- **Hadoop MapReduce**
  - Distributed framework to run applications which process large datasets on large clusters (> 1000 nodes) of commodity hardware in a reliable, fault-tolerant manner



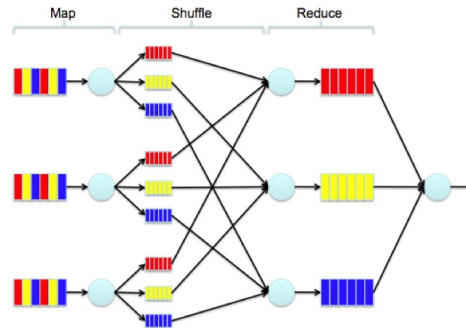
## Apache Hadoop Ecosystem



# Hadoop core: MapReduce

---

- We have already examined the MapReduce programming paradigm



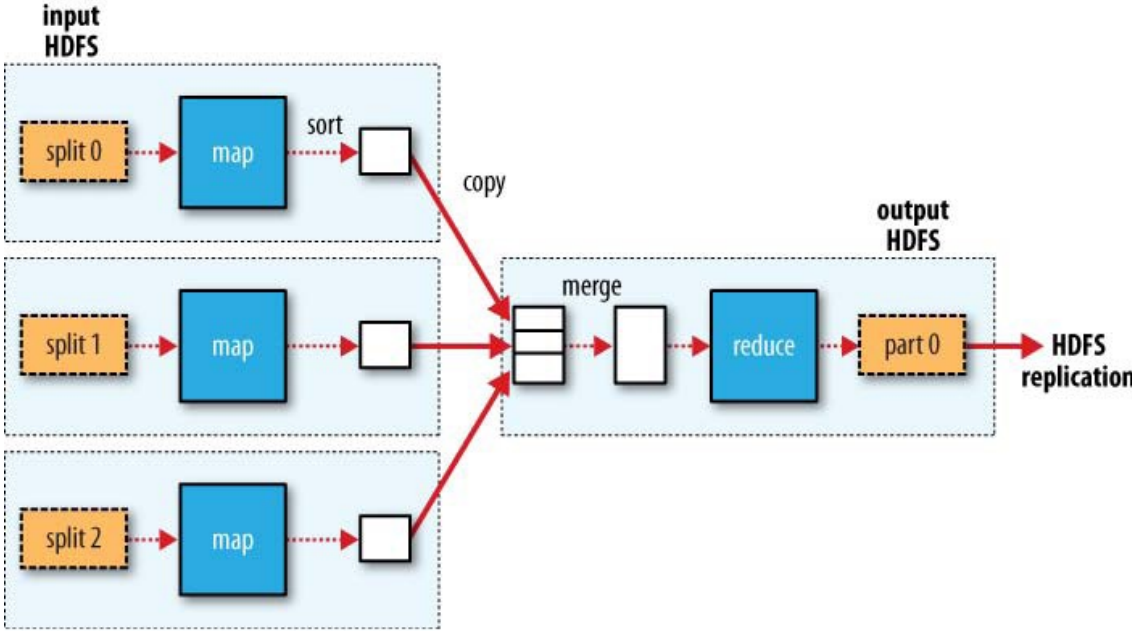
- Also used in other MPP environments and NoSQL databases (e.g., Vertica and MongoDB)

## Basic flow of how to use Hadoop

---

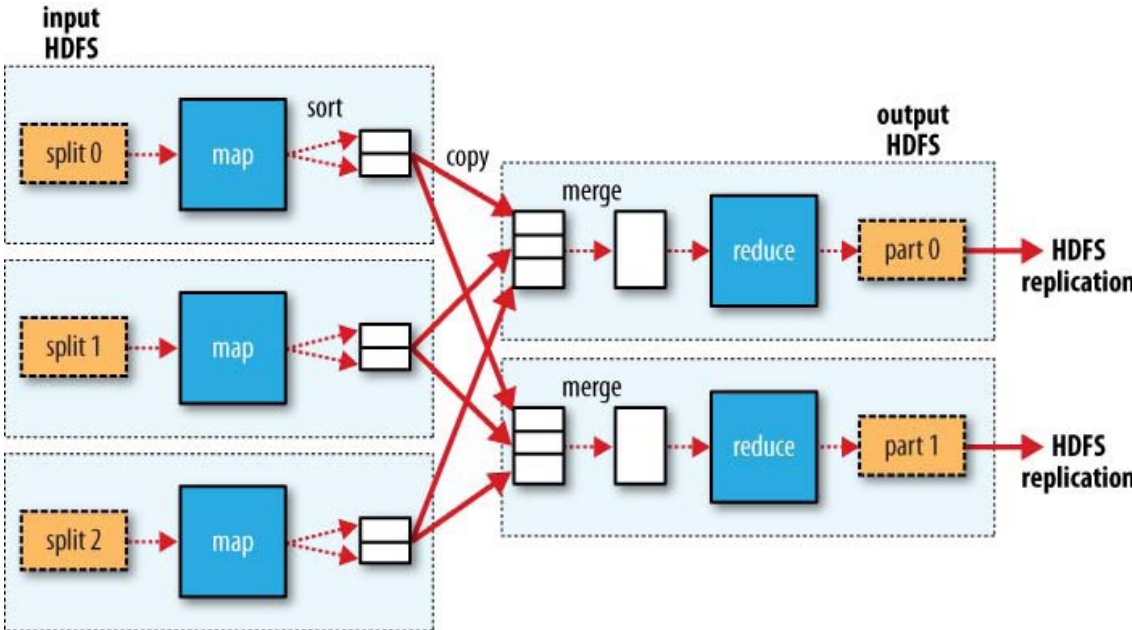
- Load data in HDFS
- Use MapReduce to analyze data
- Store results in HDFS
- Read results from HDFS

# MapReduce data flow: single Reduce task



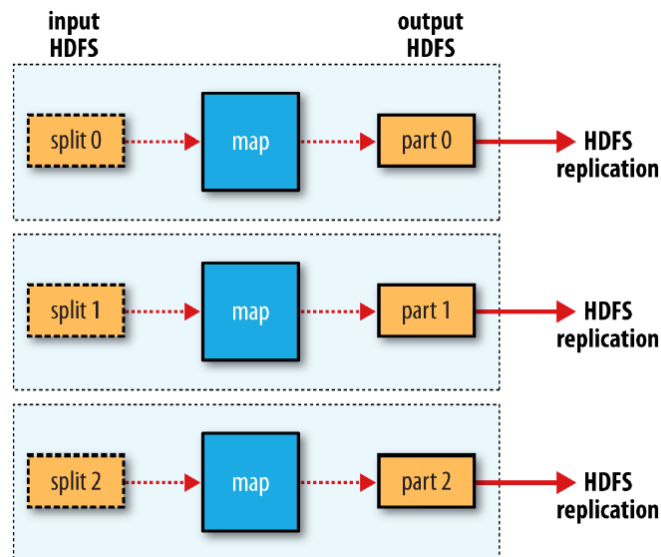
# MapReduce data flow: multiple Reduce tasks

- When there are multiple reducers, map tasks partition their output



# MapReduce data flow: no Reduce task

---



## Optional Combiner

---

- Many MapReduce jobs are limited by cluster bandwidth
  - How to minimize amount of data transferred between map and reduce tasks?
- Use Combine task
  - Combiner: optional **localized reducer** that applies a user-provided method to combine mapper output
  - Performs data aggregation on intermediate data of the same key for the Map task's output before transmitting the result to the Reduce task
    - Takes each key-value pair from the Map task, processes it, and produces the output as key-value collection pairs
- Reduce task is still needed to process records with the same key from different Map tasks

# Optional Partitioner

---

- When there are multiple reducers, map tasks partition their output
  - One partition for each Reduce task
- Goal: to determine which reducer will receive which intermediate keys and values
  - Records for any given key are all in a single partition
- Default partitioner uses a hash function on the key to determine which bucket (i.e., reducer)
- Partitioning can be also controlled by a [user-defined function](#)
  - Requires to implement a custom [partitioner](#)

# Programming languages for Hadoop

---

- Default programming language: Java
- Java program with at least 3 parts:
  1. [Main method](#) which configures job, and launches it
    - Set number of reducers
    - Set mapper and reducer classes
    - Set optional partitioner
    - Set other Hadoop configurations
  2. [Mapper class](#)
    - Takes (k,v) inputs, writes (k,v) outputs
  3. [Reducer class](#)
    - Takes k, Iterator[v] inputs, and writes (k,v) outputs

# WordCount in Java

---

- Let's analyze WordCount in Java

[https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example: WordCount\\_v1.0](https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example: WordCount_v1.0)

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
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.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

# WordCount in Java

---

map method processes one line at a time, splits the line into tokens separated by white spaces, via StringTokenizer, and emits a key-value pair <word, 1>

```
public class WordCount {

    public static class TokenizerMapper
        extends Mapper<Object, Text, Text, IntWritable>{

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context
            ) throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }
}
```

# WordCount in Java

---

reduce method sums up the values, which are the occurrence counts for each key

```
public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
        Context context
        ) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

# WordCount in Java

---

main method specifies various facets of the job

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

Output of each map is passed through local combiner (same as Reducer) for local aggregation, after being sorted on keys

## Other programming languages

---

- Use **Hadoop Streaming** utility to code Map and Reduce in programming languages different from Java (e.g., Python)  
<https://hadoop.apache.org/docs/stable/hadoop-streaming/HadoopStreaming.html>
  - Uses Unix standard streams as interface between the mapper/reducer and MapReduce framework
- Allows to use any language that can read standard input (*stdin*) and write to standard output (*stdout*)
  - Example in Python: use Hadoop Streaming for passing data between Map and Reduce code via *stdin* and *stdout*  
<http://bit.ly/19YutB1>
  - Reducer interface is different from Java: instead of receiving `reduce(k, Iterator[v])`, the script is sent one line per value, including the key

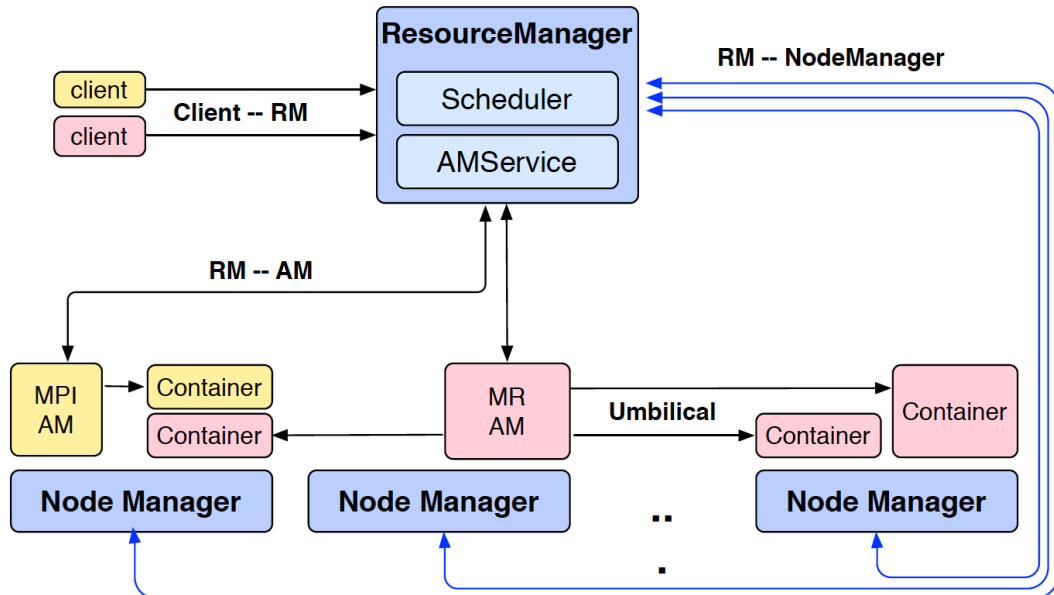
## YARN

---

- **YARN**: Yet Another Resource Negotiator
  - Distributed framework for cluster resource management and job scheduling
- Turns out Hadoop into an analytics platform in which resource management is separated from programming model
  - Not only MapReduce, but also other frameworks (e.g., Spark)

# YARN: architecture

- Global **ResourceManager** (RM)
- A set of per-application **ApplicationMasters** (AMs)
- A set of **NodeManagers** (NMs)



# YARN: data locality optimization

- Scheduling the job, YARN tries to run mappers on data-local nodes (*data locality optimization*)
  - So to not use cluster bandwidth
  - Otherwise rack-local
  - Off-rack as last choice



# Hadoop configuration

---

- Tuning Hadoop clusters for good performance is somehow magic
  - Disk I/O is usually the performance bottleneck
- Tuning hw and sw parameters, e.g.:
  - Find optimal number of disks so to maximize I/O bandwidth
  - Increase open file limit
  - Find optimal HDFS block size (related to number of mappers)
  - JVM settings for Java heap usage and garbage collection
- There are also Hadoop-specific parameters that can be tuned for performance
  - Number of mappers (not directly)
  - Number of reducers
  - Plus other map-side and reduce-side tuning parameters

# How many mappers

---

- Number of mappers
  - Driven by number of blocks in input files
  - You can adjust the HDFS block size to adjust the number of mappers
  - Good level of parallelism: 10-100 mappers per-node, but up to 300 mappers for very CPU-light map tasks
  - Task setup takes a while, so it is best if mappers take at least a minute to execute

# How many reducers

---

- Number of reducers
  - User-defined (default is 1): `Job.setNumReduceTasks(int)`
  - Right number is 0.95 or 1.75 multiplied by (*<no. of nodes> \* <no. of maximum containers per node>*)
    - 0.95: all reducers can launch immediately and start transferring map outputs as mappers finish
    - 1.75: faster nodes will finish their first round of reduce and launch a second wave of reduce doing a much better job of load balancing
  - Can be *zero* if no reduce is needed: no sorting of map-outputs before writing them to output file

<https://cwiki.apache.org/confluence/display/HADOOP2/HowManyMapsAndReducers>

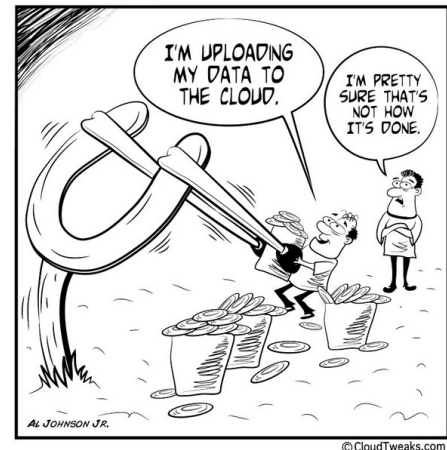
# Some tips for performance

---

- To handle massive I/O and save bandwidth
  - Compress input data
- To address massive I/O in partition and sort phases
  - Each mapper has a circular buffer memory to which it writes output; when buffer is full, its content is written (“**spilled**”) to disk. Avoid that records are spilled more than once
  - How? Adjust spill records and sorting buffer
- To address massive network traffic caused by large map output
  - Compress map output
  - Add combiner to reduce amount of data passing through shuffle and sort

# Hadoop in the Cloud

- Pros:
  - Gain cloud scalability and elasticity
  - Do not need to manage and provision the infrastructure and platform
- Main challenges:
  - Move data to cloud
    - Latency is not zero (because of speed of light)!
    - Minor issue: network bandwidth
  - Data security and privacy

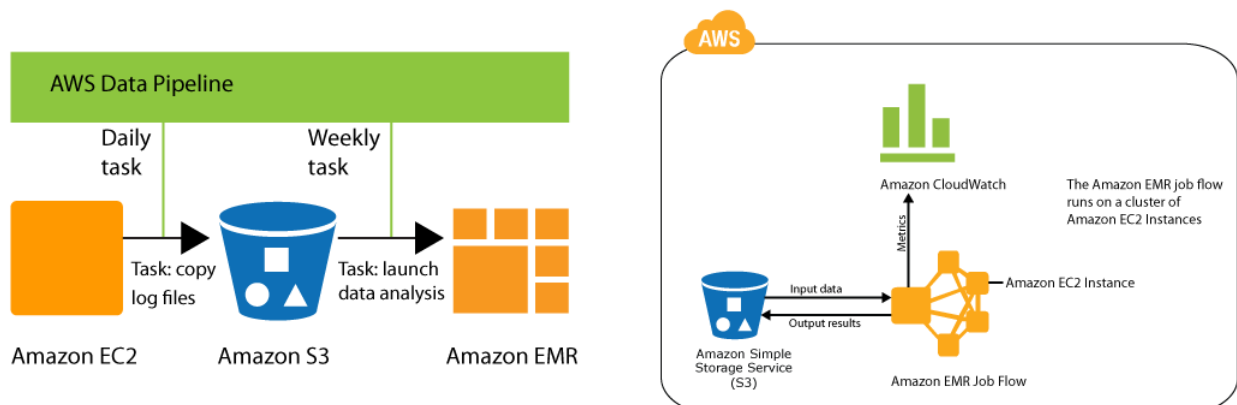


Valeria Cardellini - SABD 2025/26

68

## Amazon Elastic MapReduce (EMR)

- Distribute computational work across a cluster of virtual servers running in AWS cloud (EC2 instances)
- Not only Hadoop: also Hbase, Spark, Flink
  - Usually not the latest released version
- Input and output: Amazon S3, HDFS, DynamoDB ...
- Access through AWS Console, command line



Valeria Cardellini - SABD 2025/26

69

# Create EMR cluster

## Steps:

1. Provide cluster name
2. Select EMR release and applications to install
3. Select instance type (including spot instances)
4. Select number of instances
5. Select EC2 key-pair to connect securely to the cluster

See [AWS tutorial](#)

Running cluster can be automatically scaled or manually resized

Valeria Cardellini - SABD 2025/26

## Configurazione generale

Nome del cluster

Registrazione di log ⓘ

Cartella S3

Modalità di avvio  Cluster ⓘ  Esecuzione fase ⓘ

## Configurazione del software

Versione  ⓘ

Applicazioni  Core Hadoop: Hadoop 2.8.5 with Ganglia 3.7.2, Hive 2.3.4, Hue 4.3.0, Mahout 0.13.0, Pig 0.17.0, and Tez 0.9.1

HBase: HBase 1.4.9 with Ganglia 3.7.2, Hadoop 2.8.5, Hive 2.3.4, Hue 4.3.0, Phoenix 4.14.1, and ZooKeeper 3.4.13

Presto: Presto 0.214 with Hadoop 2.8.5 HDFS and Hive 2.3.4 Metastore

Spark: Spark 2.4.0 on Hadoop 2.8.5 YARN with Ganglia 3.7.2 and Zeppelin 0.8.1

Utilizza AWS Glue Data Catalog per i metadati delle tabelle ⓘ

## Configurazione hardware

Tipo di istanza  ⓘ Il tipo di istanza selezionato aggiunge un volume EBS GP2 a 32 GiB per istanza. [Ulteriori informazioni](#)

Numero di istanze  (1 nodo master e 2 nodi principali)

## Sicurezza e accesso

Coppia di chiavi EC2  ⓘ [Scopri come creare una coppia di chiavi EC2.](#)

Autorizzazioni  Predefinito  Personalizzato

Utilizza i ruoli IAM predefiniti. Se i ruoli non sono presenti, saranno automaticamente creati con policy gestite per aggiornamenti automatici delle policy.

Ruolo EMR [EMR\\_DefaultRole](#) ⓘ

Profilo dell'istanza EC2 [EMR\\_EC2\\_DefaultRole](#) ⓘ

70

# EMR cluster details

Ciona Termina Esportazione AWS CLI

Cluster: Il mio cluster Avvio in corso

Riepilogo Cronologia dell'applicazione Monitoraggio Hardware Configurazioni Eventi Fasi Operazioni di bootstrap

Connessioni: --

DNS pubblico master: --

Tag: -- [Visualizza tutto/Modifica](#)

Riepilogo	Dettagli di configurazione	Rete e hardware	Sicurezza e accesso
ID: j-2WVSI1MRK7YH8 Data di creazione: 2019-04-04 19:27 (UTC+2) Tempo trascorso: 31 secondi Terminazione automatica: No Protezione da cessazione: Disattivata <a href="#">Modifica</a>	Etichetta della versione emr-5.22.0 distribuita: Distribuzione Hadoop: Amazon 2.8.5 Applicazioni: Ganglia 3.7.2, Hive 2.3.4, Hue 4.3.0, Mahout 0.13.0, Pig 0.17.0, Tez 0.9.1 URI log: s3://aws-logs-738106480252-eu-central-1/elasticmapreduce/ ⓘ Visualizzazione EMRFS Disabilitato coerente: ID AMI personalizzata: --	Zona di disponibilità: eu-central-1a ID sottorete: subnet-52069c3b <a href="#"></a> Master: Provisioning 1 m4.large Core: Provisioning 2 m4.large Attività: --	Nome chiave: s3cc Profilo dell'istanza EC2 EMR_EC2_DefaultRole Ruolo EMR: EMR_DefaultRole Visibile a tutti gli utenti: Tutti <a href="#">Modifica</a> <a href="#">A breve questa caratteristica sarà impostata come deprecata.</a> Gruppi di sicurezza per sg-7f8c7a17 <a href="#"></a> (ElasticMapReduce-master) master: Gruppi di sicurezza per sg-728c7a1a <a href="#"></a> (ElasticMapReduce-slave) principale e attività:

- You can only tune some parameters for performance
  - Some EC2 parameters (e.g., heap size used by Hadoop and YARN)
  - Hadoop parameters (e.g., memory for map and reduce JVMs)

Valeria Cardellini - SABD 2025/26

71

# Google Cloud Dataproc

- Distribute the computational work across a cluster of virtual servers running in Google Cloud Platform
- Not only Hadoop, also Spark and Flink
- Input and output from other Google services, including Cloud Storage, Bigtable
- Access through REST API, Cloud SDK, and Cloud Dataproc UI
- Fine-grain pay-per-use (seconds)

## Create Cloud Dataproc cluster

The screenshot shows the 'Create a cluster' configuration page in the Google Cloud Dataproc console. The page is divided into two main sections: 'Master node' and 'Worker nodes'. The 'Master node' section includes fields for 'Name' (example-cluster), 'Region' (global), 'Zone' (us-central1-a), 'Machine type' (n1-standard-4), 'Cluster mode' (Standard), and 'Primary disk size' (500 GB). The 'Worker nodes' section includes fields for 'Machine type' (n1-standard-4), 'Nodes' (2), 'Primary disk size' (500 GB), and 'Local SSDs' (0 x 375 GB). Below these sections, there are fields for 'YARN cores' (8) and 'YARN memory' (24.0 GB). At the bottom, there is a 'Create' button and a 'Cancel' button. A link for 'Preemptible workers, bucket, network, version, initialization, & access options' is also visible.

Cloud Dataproc

← Create a cluster

Name <sup>?</sup>  
example-cluster

Region <sup>?</sup> global Zone <sup>?</sup> us-central1-a

Master node  
Contains the YARN Resource Manager, HDFS NameNode, and all job drivers

Machine type <sup>?</sup> n1-standard-4 (4 vCPU, 15.0 GB ... Cluster mode <sup>?</sup> Standard (1 master, N workers)

Primary disk size (minimum 10 GB) <sup>?</sup> 500 GB

Worker nodes  
Each contains a YARN NodeManager and a HDFS DataNode.  
The HDFS replication factor is 2.

Machine type <sup>?</sup> n1-standard-4 (4 vCPU, 15.0 GB ... Nodes (minimum 2) <sup>?</sup> 2

Primary disk size (minimum 10 GB) <sup>?</sup> 500 GB Local SSDs (0-8) <sup>?</sup> 0 x 375 GB

YARN cores <sup>?</sup> 8 YARN memory <sup>?</sup> 24.0 GB

[Preemptible workers, bucket, network, version, initialization, & access options](#)

Create Cancel

Equivalent [REST](#) or [command line](#)

# References

---

- Dean and Ghemawat, MapReduce: simplified data processing on large clusters, *OSDI 2004*  
<https://research.google/pubs/mapreduce-simplified-data-processing-on-large-clusters/>
- Leskovec, Rajaraman, and Ullman, Mining of Massive Datasets, 3<sup>rd</sup> ed., 2020 <http://www.mmds.org/>
  - Chapter 2, chapter 7 on clustering
- White, Hadoop: The Definitive Guide, 4<sup>th</sup> edition, O'Reilly, 2015
- Miner and Shook, MapReduce Design Patterns, O'Reilly, 2012