

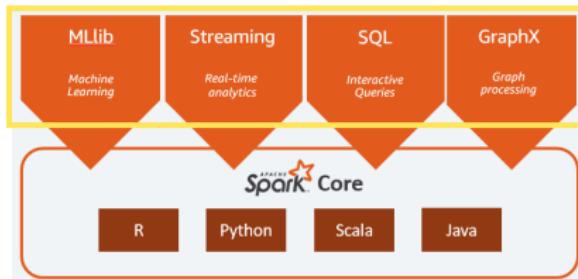
# Apache Spark Fundamentals

# OUTLINE

- ① Introduction
- ② Spark Application
- ③ Resilient Distributed Datasets (RDD)
- ④ Implementing of MapReduce Algorithms in Spark
- ⑤ DataFrames and Datasets

# Introduction

- Spark is an open-source **framework** for developing and running applications on clusters. (Devised at UC Berkeley in 2009 and later donated to Apache Software Foundation).
- Spark provides
  - Unified computing engine (Spark Core)
  - Programming interface usable with Scala, Java, Python, R
  - APIs for data analysis: Spark SQL (structured data) MLLib (machine learning), GraphX (graph analytics), Spark Streaming (streaming analytics). Spark is written in Scala.



⇒ *mapreduce  
core e  
streaming in  
corso*

- Spark runs on the Java Virtual Machine (JVM)

# Introduction

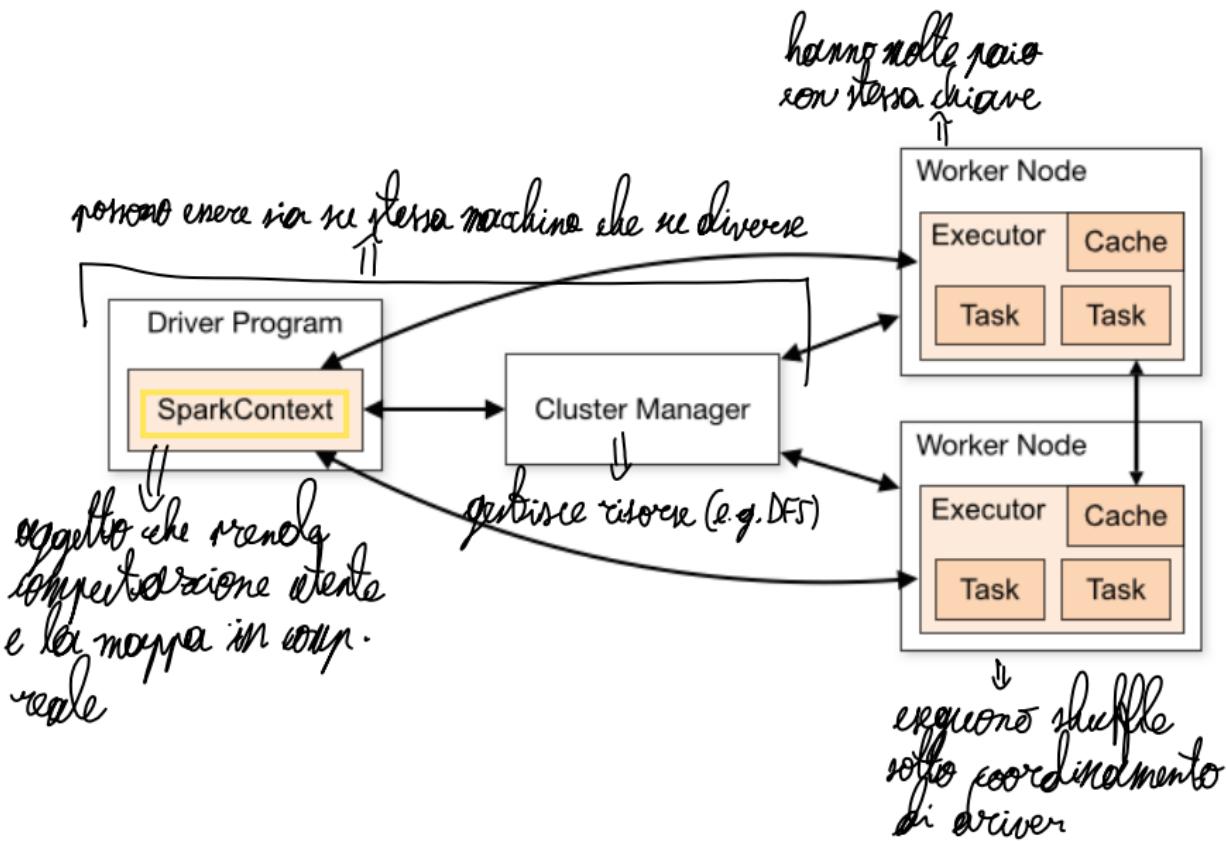
- Spark does not come with a storage system (unlike Hadoop) but can run on the Hadoop Distributed File System (HDFS) as well as on other systems (e.g., Amazon S3, Cassandra, HIVE, Relational DBMS).
- Spark's features:
  - Fault tolerance.
  - In-memory caching, which enables efficient execution of multiround algorithms, with a substantial performance improvement w.r.t. Hadoop.
- Spark can run:
  - On a single machine in the so called local mode.
  - On a cluster managed by a cluster manager such as Spark's Standalone, YARN, Mesos.

*In the homeworks, you will have the opportunity to use Spark on your machine as well as on a cluster through CloudVeneto (using YARN).*

## Why do we need a framework like Spark?

- nascondere dettagli di livello basso (fault tolerance, scheduling)
- + facile scrivere algoritmi BD
- approccio data-driven  $\Rightarrow$  approccio programmazione funzionale
- no bisogno di gestire cluster

# Spark Application



# Spark Application

- **Driver** (a.k.a. master): it is the heart of the application, and
  - Creates the **Spark Context**, an object which can be regarded as a channel to access all Spark functionalities.

**Obs.:** Spark 2.0.0 introduced a Spark Session object which encapsulates the Spark Context and provides more functionalities.

  - Distributes tasks to the executors. *controlla se esecutori*
  - Monitors the status of the execution. *→ si bloccano*

The driver runs Java/Python/Scala/R code through the Spark's API.

- **Executors** (a.k.a. workers): execute the tasks assigned (through Scala code) by the driver, and report their status to the driver.

# Spark Application

- **Cluster manager:** when the application is run on a distributed platform, the cluster manager controls the physical machines (i.e., compute nodes) and allocates resources to applications.

**EXECUTION:** when the application is run on a single machine (i.e., **local mode**), driver and executors run on that machine as threads. If instead the application is run on a distributed platform (i.e., **cluster mode**) driver and executors can run on different machines

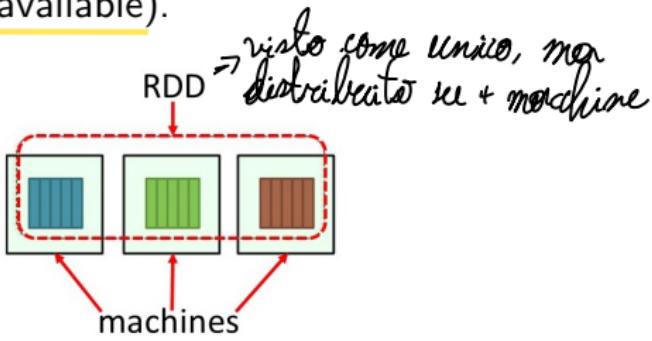
**MAP/REDUCE TASKS:** if the application implements a MapReduce computation, a task assigned to an executor typically comprises running a map function on several key-value pairs or a reduce function on several key-listOfValues pair.



# Resilient Distributed Dataset (RDD)

- Fundamental abstraction in Spark. An RDD is a collection of elements of the same type, partitioned and distributed across several machines (if available).

*quando vedi file distribuiti  
in DFS, Spark ci crea sopra  
un RDD*



- An RDD provides an interface based on coarse-grained transformations.
- RDDs ensure fault-tolerance.

## RDD: Main characteristics

- RDDs are created
  - from data in stable storage (e.g., HDFS), or
  - from other RDDs, through transformations.
- RDDs are immutable (i.e., read-only). *problema quando si valuta durata di trasformazioni*
- RDDs are materialized only when needed (lazy evaluation).
- Spark maintains the lineage of each RDD, namely the sequence of transformations that generate it, which enable to materialize it or reconstruct it after a failure, starting from data in stable storage.

*quando vogliamo lavorare RDD, dobbiamo estrarne un altro*

# RDD: Partitioning

- Key ingredient for efficiency: each RDD is broken into chunks called partitions which are distributed among the available machines.
- A program can specify the number of partitions for each RDD (if not, Spark will choose one).
- Partitions are created by default (using a HashPartitioner, based on objects' hash codes) or through a custom partitioner.
- Typical number of partitions: is 2x/3x the number of cores, which helps *balancing the work*.
- Partitioning is exploited to enhance performance:
  - Spark creates map tasks so to make each executor apply the map function on data from a locally stored partition (if possible).
  - To implement algorithms that require partitioning, the programmer can explicitly access RDD partitions (e.g., `mapPartition` method).
  - RDD partitions are exploited implicitly by some ready-made Spark aggregation primitives (e.g., `reduceByKey` method).



# RDD: Operations

The following types of operations can be performed on an RDD  $A$

- **TRANSFORMATIONS** A transformation generates a new RDD  $B$  starting from the data in  $A$ . We distinguish between:
  - **Narrow transformation.** Each partition of  $A$  contributes to (at most) one partition of  $B$ , which is stored in the same machine. No shuffling of data across machines is needed ( $\Rightarrow$  maximum parallelism). E.g., `map` method.
  - **Wide transformation.** Each partition of  $A$  may contribute to many partitions of  $B$ . Hence, shuffling of data across machines may be required. E.g., `groupByKey` method.

*reduce*

## RDD: Operations

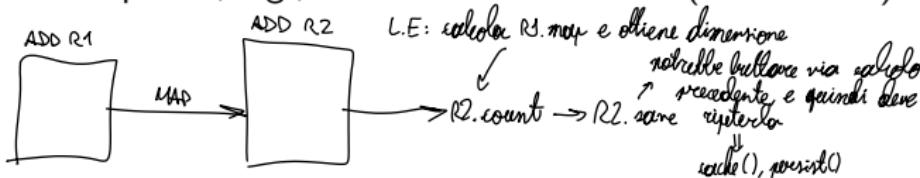
da RDD, calcola info necessarie a computazione

- ACTIONS** An action is a computation on the elements of A which returns a value to the application. E.g., count() method.

**LAZY EVALUATION:** RDD A is materialized only when an action is performed. This must be carefully taken into account when measuring running times

- Persistence.** Methods like cache() or persist() will save the RDD data in memory after the subsequent action.

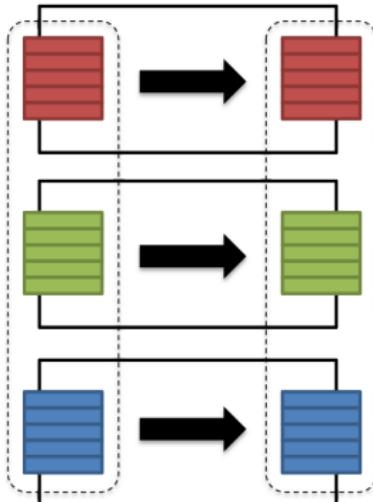
- **cache()**: data are stored asin RAM (if they fit). Data that do not fit are recomputed when needed.
- **persist()** data are stored as instructed. There are several options, e.g., MEMORY\_AND\_DISK (RAM+Disk).



# RDD: Operations

## Narrow transformation

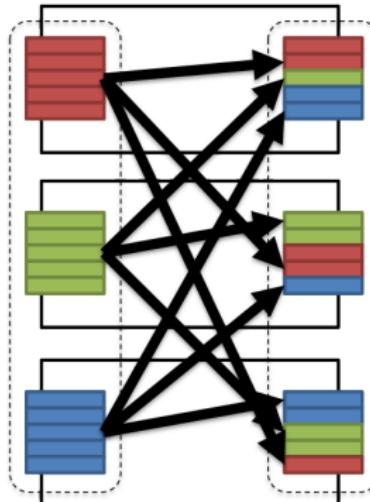
- Input and output stays in same partition
- No data movement is needed



simile a Map

## Wide transformation

- Input from other partitions are required
- Data shuffling is needed before processing



simile a Reduce  
(per fare un testo giurare, nono venire da  
varie parti diverse)

# Implementing MapReduce algorithms in Spark

The **homeworks** will provide you **first-hand experience** on how to implement MapReduce algorithms using Spark.

Summarized here are key points to keep in mind about this issue.

- Spark **enables the implementation of MapReduce algorithms** but offers a **much richer set of processing methods**.
- Spark (and MapReduce) exploits a **core idea of functional programming**: *functions can be argument to other functions*.
- A **program which implements a MapReduce algorithm** typically comprises
  - **Spark configuration** steps (e.g., **creation of the context**);
  - **Definitions of global variables and data structures** (typically, in the driver's memory);
  - **Reading of input data** into one or more RDDs;
  - **Code for the various rounds**.

# Implementing a MapReduce round in Spark

MAP-REDUCE

ROUND 1:

$(k_i, v_i) \rightarrow \text{emit } (\hat{k}_i, \hat{v}_i)$   
All have  $\hat{k}_i$ , reduce  $(L_{\hat{k}_i})$

SPARK

read  $(k_i, v_i)$  into RDD  $R_s$   
 $R_2 = R_s.\text{map}()$   
 $R_3 = R_2.\text{groupByKey}()$   
 $R_4 = R_3.\text{map}()$

# Implementing a MapReduce round in Spark

- **Map Phase:** on the input RDD  $X$  invoke one of map methods offered by Spark (narrow transformation) passing the desired map function as argument.
- **Reduce Phase:** on the RDD  $X'$  resulting from the Map Phase:
  - invoke one of grouping methods offered by Spark (wide transformation) to group the key-value pairs into key-ListOfValues pairs;
  - invoke one of map methods offered by Spark to apply the desired reduce function to each key-ListOfValues pair.

Alternatively, one can use one of the ready-made reduce primitives offered by Spark (e.g., `reduceByKey`).

# DataFrames and Datasets

- RDDs represent the most basic data model in Spark: it is low-level and schema-less.
- On top of RDDs, the Spark SQL module provides APIs to operate on the following structured data (similar to tables in Relational DB):
  - **DataFrame:** distributed collection of data organized into named columns
  - **Dataset:** extension of DataFrame (available only in Java and Scala) with type-safe, object-oriented programming interface.

In the course we use RDDs only!

# Summary

- Spark features
- Spark Application: driver process, executor processors, cluster manager.
- Resilient Distributed Dataset (RDD)
  - Main characteristics
  - Partitioning
  - Operations: transformations, actions, persistence.
- Implementing MapReduce algorithms in Spark.
- DataFrames and Datasets.

## References

- AS-1 Spark's Web Site: [spark.apache.org](http://spark.apache.org)
- AS-2 Spark's RDD Programming guide:  
[spark.apache.org/docs/latest/rdd-programming-guide.html](http://spark.apache.org/docs/latest/rdd-programming-guide.html)
- CZ18 A Gentle Introduction to Apache Spark. From B. Chambers, M. Zaharia. *Spark: The Definite Guide*, Databricks 2018.
- Z+12 M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI 2012: 15-28.