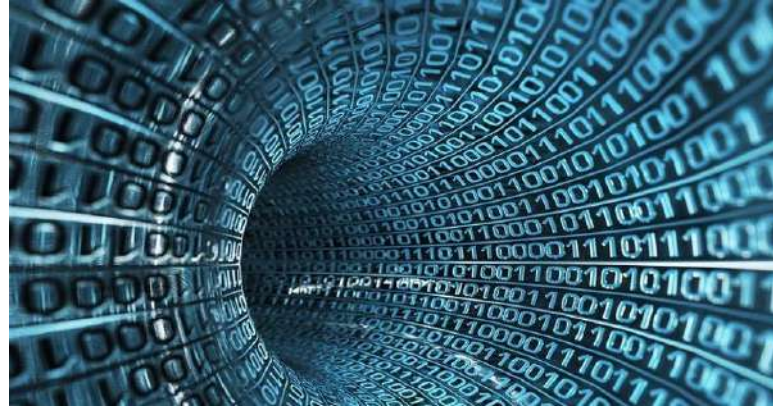


Sistemi Informativi Evoluti e Big Data



Tecnologie Big Data – Apache Spark

Prof. Devis Bianchini

Università degli Studi di Brescia

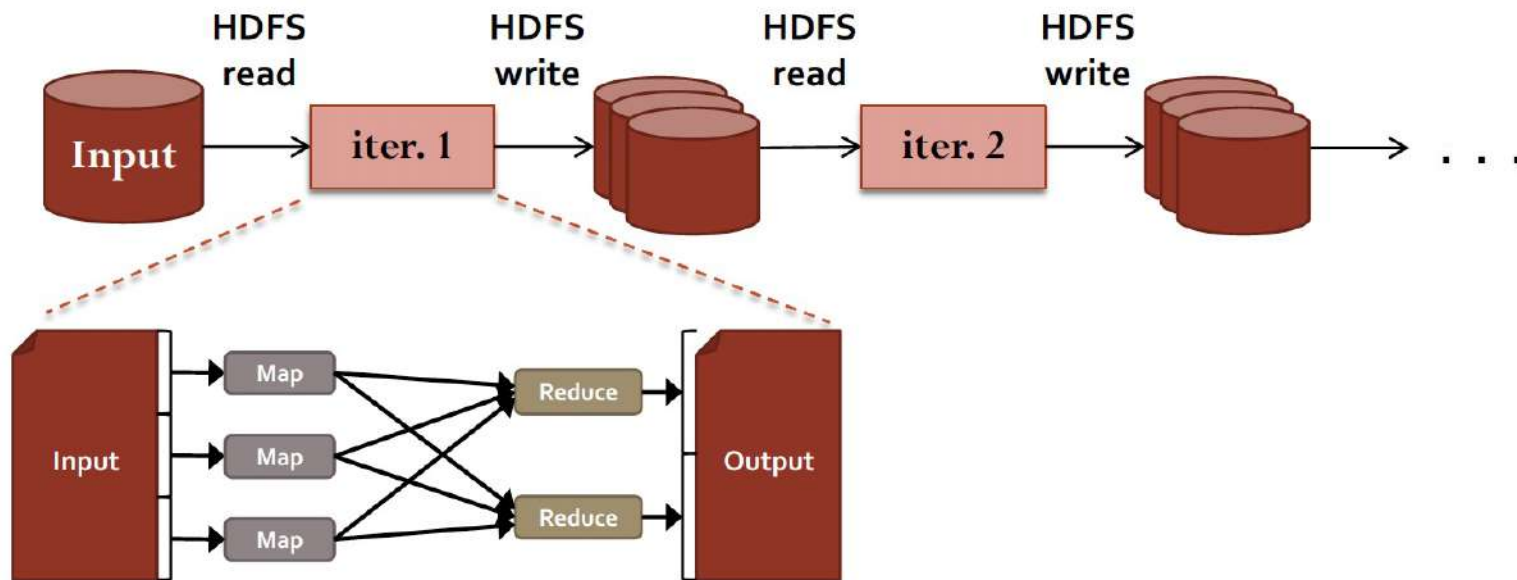
Dipartimento di Ingegneria dell'Informazione



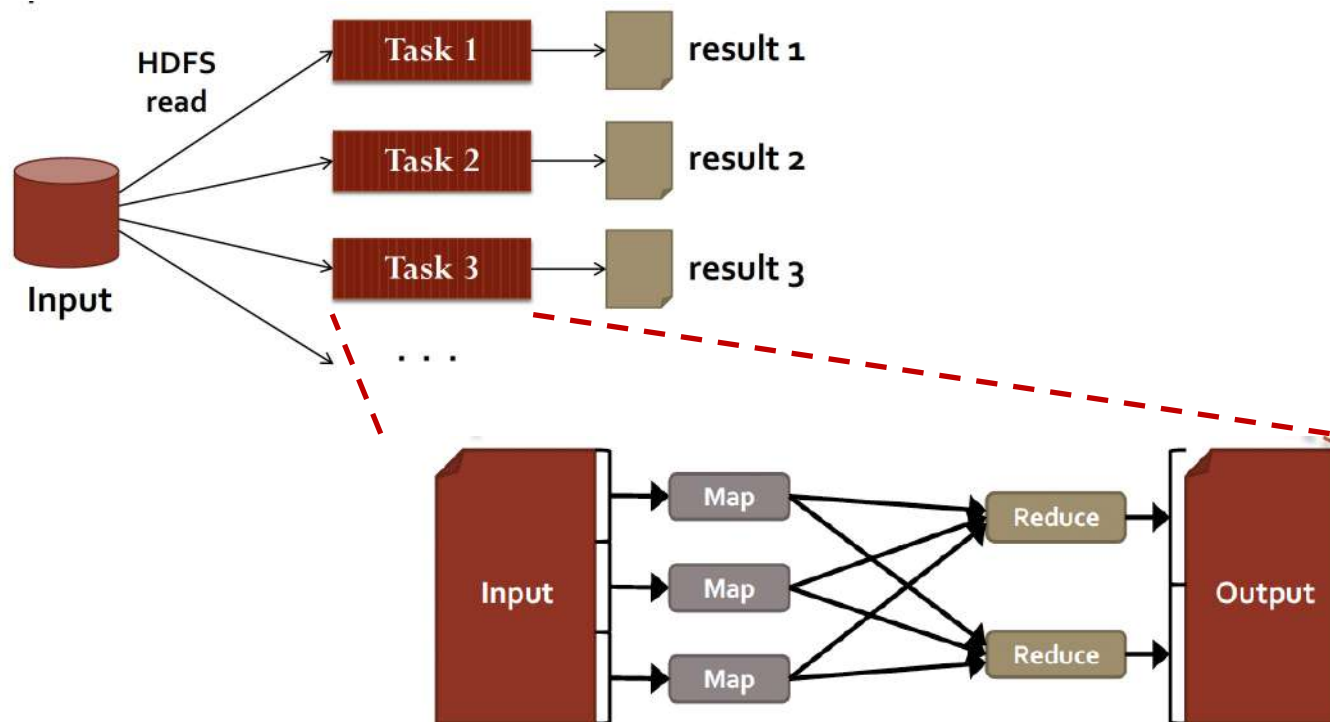
Limiti dei sistemi distribuiti map-reduce

- Il paradigma map-reduce potrebbe risultare **“lento”**, perché ad ogni job legge i dati da disco, esegue le funzioni di map e reduce, scrive il risultato delle elaborazioni ancora su disco
- Il paradigma map-reduce è **poco adatto per algoritmi iterativi o multi-stage** (e.g., Machine Learning, Graph & Network Analysis)
- Il paradigma map-reduce **non è adatto per effettuare Data Mining di tipo interattivo** (elaborazioni in R, ad-hoc reporting, searching)

Operazioni iterative con MapReduce



Operazioni interattive con MapReduce



Debolezze e limitazioni di MapReduce

- Problemi di efficienza
 - Accessi ai dati su disco frequenti
 - Ridotto sfruttamento della memoria principale
- Modello di programmazione
 - Difficile implementare tutto come un programma MR
 - Spesso anche per operazioni semplici sono richiesti diversi step MR
 - Costrutti di controllo e tipi di dato limitati
- Elaborazione real-time
 - Stream processing e random memory access praticamente impossibili

Un nuovo sistema di elaborazione distribuita

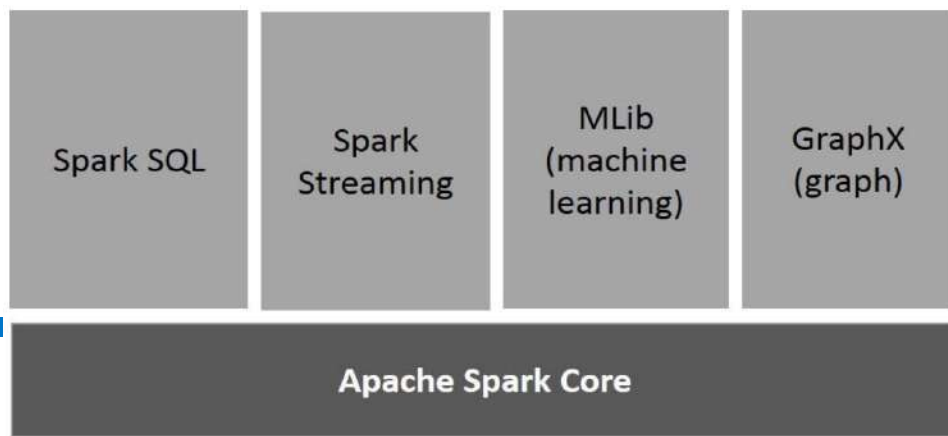
- Per consentire di specificare sequenze complete di operazioni, anche di tipo iterativo e interattivo, **in un solo job**, riducendo la complessità del codice dell'applicazione di elaborazione
- Per mantenere i dati **in memoria** fino al termine dell'intera sequenza di elaborazione, evitando di leggere o scrivere su disco (a meno che non sia lo sviluppatore stesso a richiederlo)
- Garantire in ogni caso la proprietà di **fault-tolerance** dei sistemi di elaborazione distribuita esistenti (e.g., Hadoop)

Apache Spark

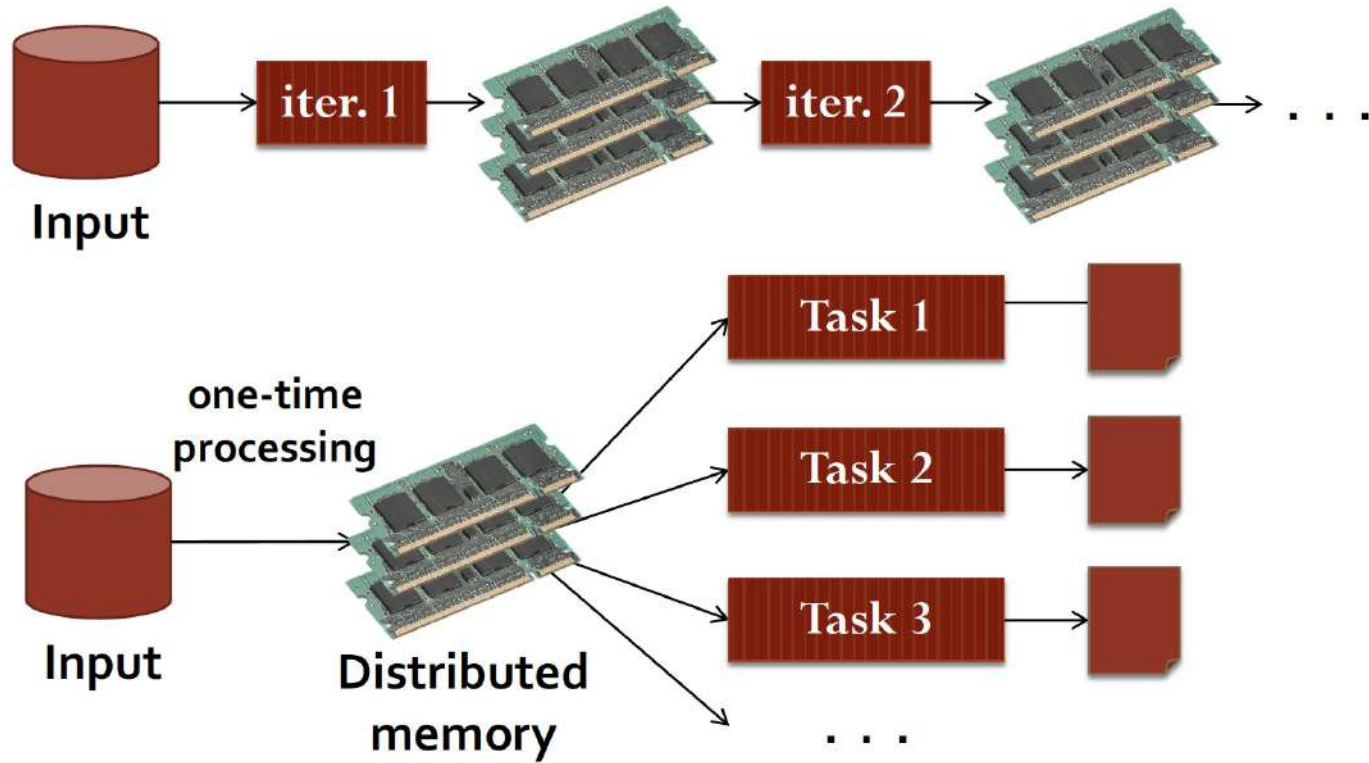
- Un **sistema di elaborazione distribuita** molto efficiente, per processare massive quantità di dati (versione corrente: 2.4.4 del 30 agosto 2019)
- Estende il paradigma MapReduce a operazioni di tipo interattivo e iterativo, e all'elaborazione di data streaming
- Creato nel 2009 da AMPLab (Berkeley), ora Databricks
- Scritto in **Scala**
- Open Source dal 2010, licenza Apache dal 2013
- Distribuito su GitHub
- Non è una versione modificata di Hadoop, né dipende da Hadoop per il suo funzionamento
 - Possiede un **proprio sistema di gestione dei cluster** (Mesos)
 - Hadoop è uno degli ecosistemi su cui installare Spark

Caratteristiche di Apache Spark

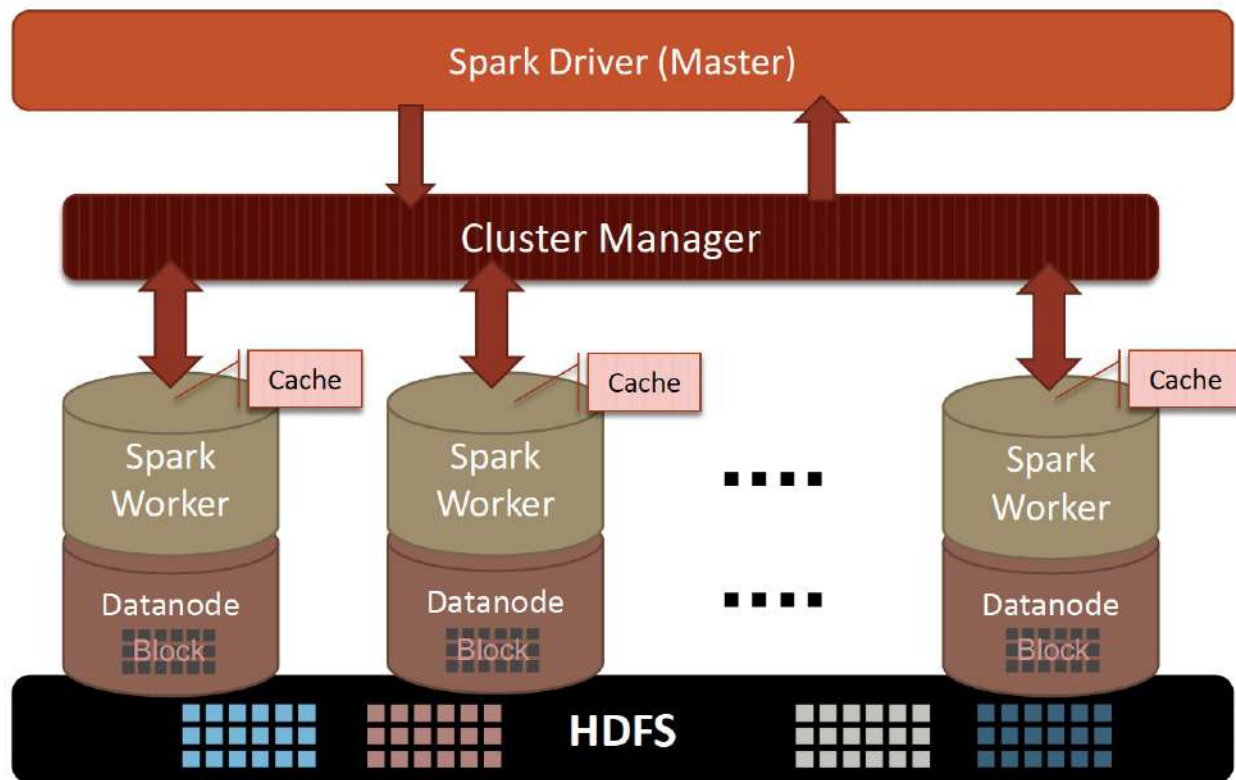
- **Velocità** - Spark permette di eseguire applicazioni fino a 100 volte più velocemente (se eseguite completamente in-memory) e 10 volte più velocemente (con scritture su disco), rispetto al sistema Hadoop, grazie alla limitazione di letture e scritture su disco
- **Supporto multi-language** – Spark fornisce API in Java, Scala e Python
- **Advanced analytics** – Spark non supporta soltanto il paradigma MapReduce, ma permette anche query SQL-like, fornisce soluzioni per la gestione di data streaming, oltre che algoritmi di Machine Learning e di graph processing



Operazioni iterative e interattive con Apache Spark

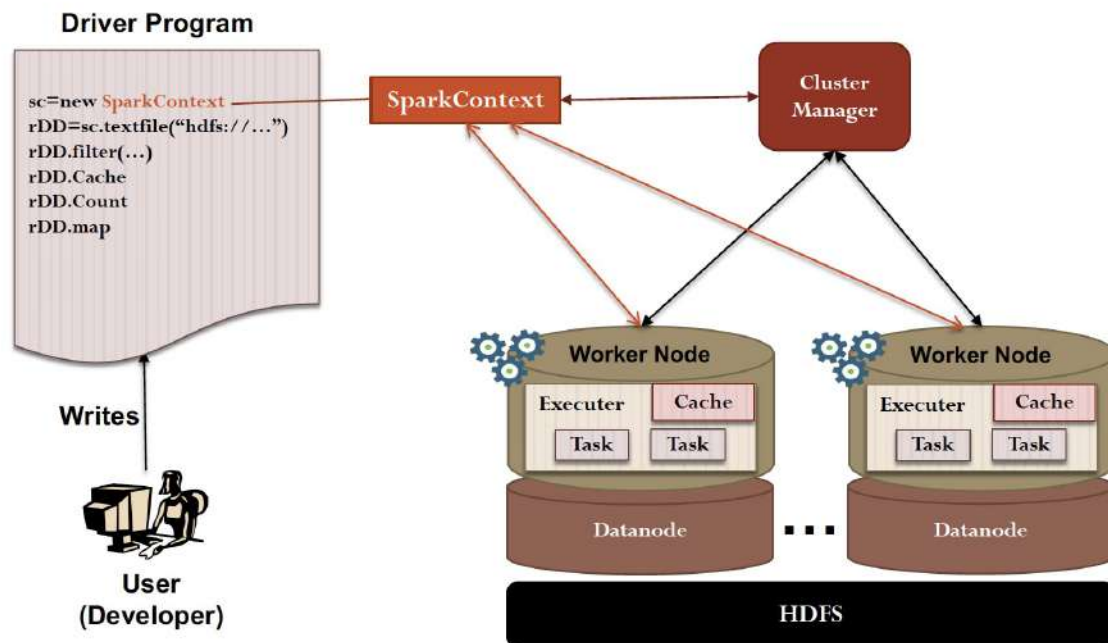


Architettura di Spark



Anatomia di un'applicazione in Spark

- L'applicazione viene eseguita sotto forma di un insieme di processi indipendenti
- I processi sono in esecuzione su diversi nodi di un cluster e sono coordinati da un oggetto di tipo **SparkContext** presente nel main program dell'applicazione (**driver program**)
- Per eseguire l'applicazione sul cluster, lo SparkContext si può connettere a diversi tipi di cluster manager (per esempio, Mesos o YARN)



Il “cuore” di Spark - RDD

- Un RDD (Resilient Distributed Dataset) è un set di dati diviso in tante partizioni (per esempio, una tabella chiave-valore spezzata in tante sotto-tabelle, un file spezzato in diverse linee)
- È **immutabile** (una volta creato non si può cambiare, ma solo sottoporre a trasformazione per creare un nuovo RDD)
- Le **trasformazioni possibili** per creare nuovi RDD sono poche, deterministiche e ripetibili (mapping, filtering, union)
- Può restare in memoria o essere materializzato su disco a scelta del programmatore
- È descritto dai **metadati** che ne consentono la ricostruzione in caso di fault (quali sono gli RDD padre e la sequenza di trasformazioni, o LINEAGE, che lo hanno generato)

RDD – Costrutti chiave

- Gli RDD rappresentano **dati o trasformazioni sui dati** opportunamente predisposti per essere parallelizzati su più nodi (**partizioni**)
- Gli RDD possono essere creati parallelizzando una collezione esistente tramite la funzione **parallelize** dello SparkContext (**sc.parallelize(data)**) oppure partendo da un dato che risiede in un sistema di storage esterno (e.g., HDFS, Hbase, S3) tramite il comando **sc.textFile(...)**
- Oltre alle trasformazioni, sugli RDD possono essere applicate delle **azioni**, che forzano l'esecuzione di operazioni sugli RDD e ritornano dei valori
- Gli RDD sono particolarmente indicati per applicazioni che effettuano le stesse operazioni su tutti gli elementi del dataset

Trasformazioni

- Rappresentano operazioni sugli RDD che ritornano altri RDD o collezioni di RDD
 - Per esempio, **map**, **filter**, **join**, etc.
- **Lazy evaluation**: nulla è mandato in esecuzione finché un'azione non lo richieda
- Pensati per rendere l'esecuzione di programmi Spark più efficienti (per esempio, l'output di una trasformazione di tipo **map** non è restituita come risultato, ma rappresentata come RDD in-memory e utilizzata per altre operazioni, come un'operazione di tipo **reduce**)

Esempi di trasformazioni Apache Spark (I)

- **map(func)**: ritorna un nuovo RDD ottenuto applicando ad ogni elemento dell'RDD di partenza la funzione **func**
- **filter(func)**: ritorna un nuovo RDD ottenuto selezionando quegli elementi dell'RDD di partenza sui quali la funzione **func** è **true**
- **union(otherDataset)**: ritorna un nuovo RDD che contiene l'unione degli elementi dell'RDD a cui la trasformazione è applicata e degli elementi dell'argomento
- **intersection(otherDataset)**: ritorna un nuovo RDD che contiene l'intersezione degli elementi dell'RDD a cui la trasformazione è applicata e degli elementi dell'argomento
- **join(otherDataset, [numTasks])**: quando invocato su due RDD di tipo chiave-valore (K, V) e (K, W), ritorna un RDD di coppie (K, (V, W)), contenente tutte le coppie di elementi per ogni chiave; può essere parallelizzato tramite **[numTasks]** in caso di partizioni



Esempi di trasformazioni Apache Spark (II)

- **distinct([numTasks])**: ritorna un nuovo RDD che contiene gli elementi distinti dell'RDD di partenza
- **groupByKey([numTasks])**: applicato su un RDD di coppie (K, V), ritorna un RDD di coppie (K, Iterable<V>)
- **reduceByKey(func,[numTasks])**: applicato su un RDD di coppie (K, V), ritorna un RDD di coppie <chiave, valore>, dove i valori associati ad una data chiave sono aggregati tramite una funzione **func**
- **sortByKey([ascending],[numTasks])**: ritorna un nuovo RDD che contiene coppie <chiave, valore> ordinate per chiave

Azioni

- Rappresentano operazioni sugli RDD che ritornano qualsiasi valore diverso da un RDD
- Per esempio, **reduce**, **count**, etc.
- Il risultato di un'azione sugli RDD sono salvati oppure resi disponibili all'analista
- Le azioni sono sincrone; fino a quando un'azione non è eseguita, non è nemmeno possibile accedere ai dati elaborati

Nota: le Azioni impongono l'esecuzione di un'operazione sull'RDD; le trasformazioni predispongono dei cambiamenti (**lazy evaluation**)



Esempi di azioni Apache Spark (I)

- **reduce(func)**: aggrega gli elementi dell'RDD applicando la funzione **func** (che accetta due argomenti e ne restituisce uno solo); la funzione **func** dovrebbe essere commutativa e associativa, cosicché possa essere eseguita correttamente in parallelo
- **collect()**: ritorna gli elementi di un RDD come un array; di solito utilizzata dopo una trasformazione di filtraggio o altra trasformazione che ritorna un numero piccolo di elementi
- **count()**: ritorna il numero di elementi in un RDD
- **first()**: ritorna il primo elemento in un RDD

Esempi di azioni Apache Spark (II)

- **saveAsTextFile(path)**: scrive gli elementi in un file di testo su file system locale, HDFS oppure un altro DFS. Spark invoca il metodo toString() per trasformare ogni elemento nell'RDD in una riga del file di testo
- **countByKey()**: possibile solo su RDD che contengono coppie (K,V), ritorna una hashmap (K, int) che riporta per ogni chiave K il numero di valori
- **foreach(func)**: esegue la funzione func per ogni elemento dell'RDD a cui è applicata l'azione

Spark examples (Scala)

```
// "sc" is a "Spark context" - this transforms the file into an RDD
val textFile = sc.textFile("README.md")
// Return number of items (lines) in this RDD; count() is an action
textFile.count()
// Filtering. Filter is a transformation
val linesWithSpark = textFile.filter(line => line.contains("Spark"))
// Chaining - how many lines contain "Spark"? count() is an action
textFile.filter(line => line.contains("Spark")).count()
// Length of line with most words. reduce() is an action
textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
// Word count - traditional map-reduce. collect() is an action
val wordCounts = textFile.flatMap(line
    => line.split(" ")).map(word => (word, 1)).reduceByKey((a, b) => a + b)
wordCounts.collect()
```

Source: <https://spark.apache.org/docs/latest/quick-start.html>

Spark – RDD Persistence (I)

- È possibile rendere esplicitamente persistente un RDD su disco
- Quando un RDD è reso persistente, ogni nodo salva le partizioni che sottopone ad elaborazione in-memory e che riutilizzerà in altre azioni
- Questo consente un aumento delle prestazioni fino a 10 volte, pur prevedendo un salvataggio di dati su disco
- Per rendere un RDD persistente, vengono utilizzati i metodi **`persist()`** o **`cache()`**; la prima volta che un'azione viene applicata su un RDD, quest'ultimo viene mantenuto in-memory sui nodi del cluster

Spark – RDD Persistence (II)

- Cache è fault-tolerant – se una partizione di un RDD viene persa, verrà automaticamente ricalcolata utilizzando le trasformazioni che l'hanno originata
- Usando `persist()` è possibile scegliere lo storage level (`MEMORY_ONLY`, `DISK_ONLY`, `MEMORY_AND_DISK`, etc.), il primo è di default usando `cache()`
- È possibile “liberare” spazio di storage utilizzando il metodo `unpersist()`



RDD, DataFrame e Dataset (I)

- Per semplicità, sono stati citati finora solo gli RDD, che sono gli elementi originali di Apache Spark
- Apache Spark in realtà attualmente presenta tre tipi di elementi — RDD, DataFrame e Datasets
 - **RDD** – Introdotti fin dalla prima release Spark 1.0
 - **DataFrame** – Introdotti nella release Spark 1.3
 - **Dataset** – Introdotti nella release Spark 1.6

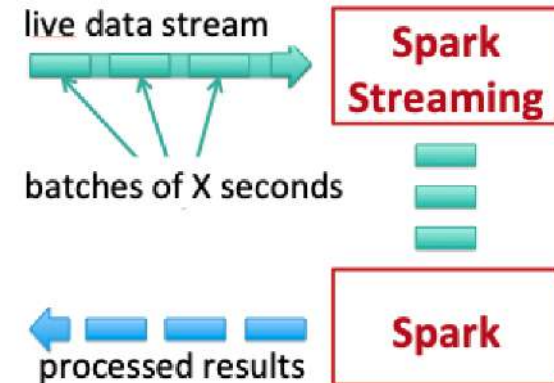


RDD, DataFrame e Dataset (II)

- **DataFrame:**
 - A differenza di un RDD, i dati sono organizzati in colonne con nome, come tabelle in un relazionale
 - Impongono una struttura su una collezione distribuita di dati, permettendo un alto livello di astrazione
- **Dataset:**
 - Estensione dei DataFrame, forniscono un'interfaccia object-oriented
 - Entrambi possono essere convertiti in RDD (tramite il metodo **rdd**), mentre il procedimento inverso è possibile solo se l'RDD è in formato tabulare (usando il metodo **toDF**)

Spark e data streaming

- Elabora lo streaming di dati suddividendolo in una serie di batch molto piccoli
 - I batch coprono tipicamente un lasso di tempo inferiore al mezzo secondo, con una latenza di circa 1 sec
 - Spark Core tratta ogni batch come un RDD e lo processa utilizzando le trasformazioni e le azioni viste per gli RDD
 - Alla fine, il risultato dell'elaborazione degli RDD in ingresso viene emesso a sua volta come una serie di RDD
 - Potenzialmente utile per unire in un unico sistema batch processing e streaming processing



Spark vs. Hadoop MapReduce

- **Performance:** Spark è normalmente più veloce, ma con qualche limitazione
 - Spark elabora dati in-memory; Hadoop MapReduce salva i dati su disco in maniera persistente dopo l'esecuzione di una funzione map o reduce
 - Spark generalmente richiede molta memoria per essere veramente performante; se sussistono altre applicazioni che sono memory-demanding, Spark degrada le proprie prestazioni
- **Ease of use:** Spark è più facile da programmare
- **Data processing:** Spark è un approccio più generale
- **Maturità:** Spark è in corso di maturazione, ma Hadoop MapReduce mantiene ancora un maggiore livello di maturazione

Avviare la Shell di Apache Spark

- Spark Shell [**Scala**]

```
$:~spark-*/bin/spark-shell
```

```
Welcome to
```

```
  ____
 /  __ \   _ __   ___
 \  __ <  / __ `\/ __ `\/
  \  __ < /  __/ /  __/
   \____/_/_____/_____/
                        version 1.3.1
```

```
Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_05)
```

```
Type in expressions to have them evaluated.
```

```
scala>
```

- Spark Shell [**Python**]

```
$:~spark-*/bin/pyspark
```



Esempio (Scala) – Word Count

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                      .map(word => (word, 1))
                      .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Per eseguire un programma [Scala] salvato in un file:

```
$:~spark-*/bin/spark-shell -i file.scala
```



Esempio (Python) – Word Count

```
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

Per eseguire un programma [Python] salvato in un file:

```
$:~spark-*/bin/spark-submit file.py
```

Il file deve contenere definizione esplicita dello **SparkContext**

```
from pyspark import SparkContext, SparkConf
conf = SparkConf().setAppName("AppName")
sc = SparkContext(conf=conf)
```



Java Spark API

```
package wordcount;

import java.util.Arrays;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.*;
import scala.Tuple2;

public class WordCount {

    public static void main(String[] args) {
        SparkConf conf = new SparkConf().setAppName("WordCountSpark");
        JavaSparkContext sc = new JavaSparkContext(conf);

        JavaRDD<String> textFile = sc.textFile("input/divina_commedia.txt");

        JavaPairRDD<String,Integer> counts = textFile
            .flatMap(s->Arrays.asList(s.split(" ")).iterator())
            .mapToPair(word->new Tuple2(word,1))
            .reduceByKey((a,b)->(int)a+(int)b);

        counts.saveAsTextFile("output/countData.txt");
    }
}
```



Navigare nel file system distribuito (Ambari)

http://host:8080

The screenshot displays the Ambari web interface. On the left is a dark sidebar with navigation links: Ambari, Dashboard, Services (HDFS, YARN, MapReduce2, Tez, Hive, HBase, Pig, Sqoop, Oozie, ZooKeeper, Storm), and a bottom logo for the University of Brescia. The main content area is titled 'Dashboard / Metrics' and includes tabs for METRICS, HEATMAPS, and CONFIG HISTORY. A red arrow points from the 'NameNode CPU WIO' widget to a 'Views' dropdown menu, which lists 'Files View', 'Workflow Manager', and 'DataNodes Live'. The 'Files View' is active, showing a file browser for the user 'amy_ds' with a search bar and a table of files and folders. The table has columns for Name, Size, Last Modified, Owner, Group, Permission, Erasure Coding, and Encrypted. The background shows various system metrics widgets like 'NameNode Heap', 'HDFS Disk Usage', 'NameNode CPU WIO', 'CPU Usage', and 'DataNodes Live'.

Ambari

Dashboard

Services

- HDFS
- YARN
- MapReduce2
- Tez
- Hive
- HBase
- Pig
- Sqoop
- Oozie
- ZooKeeper
- Storm

Dashboard / Metrics

METRICS HEATMAPS CONFIG HISTORY

Views

- Files View
- Workflow Manager
- DataNodes Live

1 HOUR

1/1

CPU Usage

No Data Available

Files View

Sandbox

/ > user > amy_ds

Total: 10 files or folders

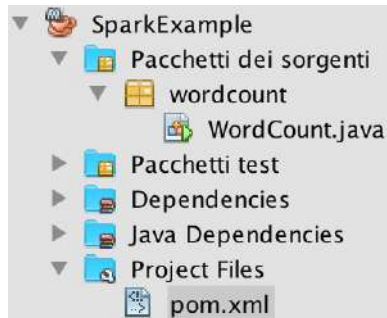
Select All New Folder Upload

Search in current directory...

Name >	Size >	Last Modified >	Owner >	Group >	Permission	Erasure Coding	Encrypted
Trash	--	2019-12-05 13:00	amy_ds	hdfs	drwx-----		No
.sparkStaging	--	2019-12-05 09:31	amy_ds	hdfs	drwxr-xr-x		No
.staging	--	2019-12-05 13:35	amy_ds	hdfs	drwx-----		No
WordCount\$IntSumReducer.class	1.7 kB	2019-12-02 13:48	amy_ds	hdfs	-rw-r--r--		No
WordCount\$TokenizerMapper.class	1.7 kB	2019-12-02 13:48	amy_ds	hdfs	-rw-r--r--		No
WordCount.java	2.0 kB	2019-12-02 13:48	amy_ds	hdfs	-rw-r--r--		No
files-view	--	2019-12-04 19:49	amy_ds	hdfs	drwxr-xr-x		No
input	--	2019-12-05 12:23	amy_ds	hdfs	drwxr-xr-x		No
output	--	2019-12-05 13:34	amy_ds	hdfs	drwxr-xr-x		No
wo.jar	3.0 kB	2019-12-02 10:48	amy_ds	hdfs	-rwxrwxr--		No

Compilare un'applicazione Spark Java

Come progetto MAVEN attraverso l'IDE NetBeans (dalla versione 6.9 in poi):



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany</groupId>
  <artifactId>SparkExample</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.12</artifactId>
      <version>2.4.0</version>
    </dependency>
  </dependencies>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
</project>
```



Eseguire l'applicazione Spark Java (I)

Esempio: **WordCount**

1. Generazione del file *jar* contenente la classe *WordCount* (per esempio, compilando il progetto MAVEN in NetBeans)
2. Spostamento del file *jar* sulla distribuzione Hadoop (file system locale)

```
scp -P 2222 <files-to-transfer> <hadoop_user>@<host>:/<destination-on-server>
```



Eseguire l'applicazione Spark Java (II)

Esempio: **WordCount**

1. Generazione del file *jar* contenente la classe *WordCount* (per esempio, compilando il progetto MAVEN in NetBeans)
2. Spostamento del file *jar* sulla distribuzione Hadoop (file system locale)
3. Creazione delle cartelle che conterranno gli input e il risultato dell'elaborazione (da prompt dei comandi)

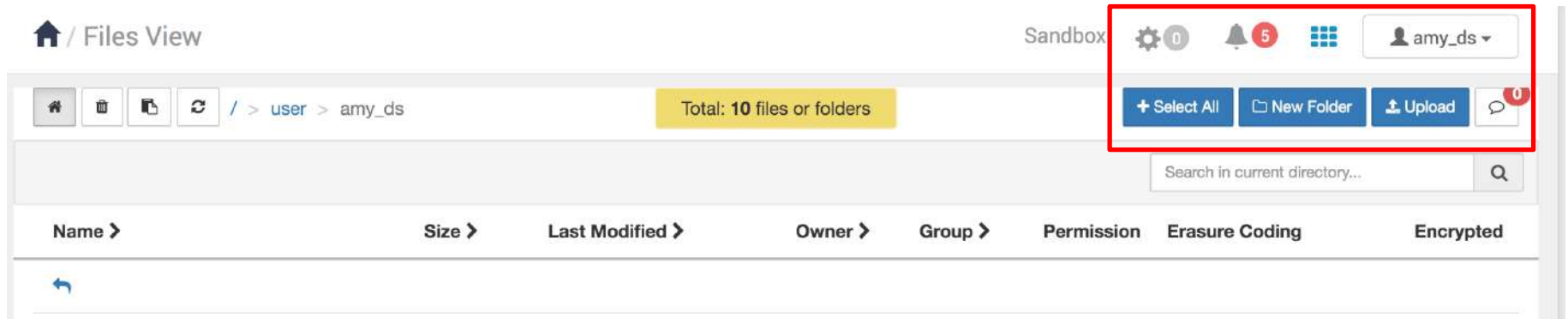
```
hdfs dfs -mkdir input
hdfs dfs -put divina_commedia.txt input
hdfs dfs -mkdir output
```



Eseguire l'applicazione Spark Java (III)

Esempio: **WordCount**

1. Generazione del file *jar* contenente la classe *WordCount* (per esempio, compilando il progetto MAVEN in NetBeans)
2. Spostamento del file *jar* sulla distribuzione Hadoop (file system locale)
3. Creazione delle cartelle che conterranno gli input e il risultato dell'elaborazione (da Ambari)



Eseguire l'applicazione Spark Java (IV)

```
$:~spark-*/bin/spark-submit
```

```
--class "SimpleApp"
```

```
--master local[4]
```

```
SparkProject-1.0.jar
```

**local to run locally
with one thread, or
local[N] to run locally
with N threads.**



Eseguire l'applicazione Spark Java (V)

```
$:~spark-*/bin/spark-submit
```

```
--class "SimpleApp"
```

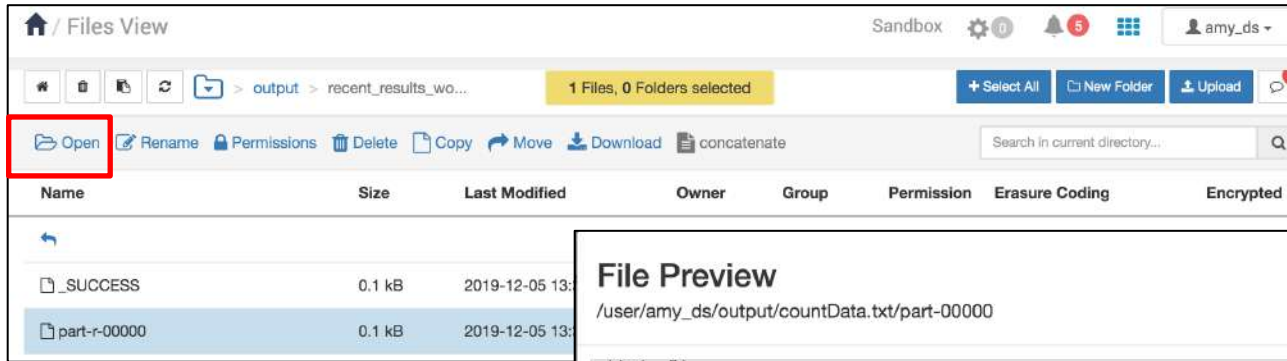
```
--master yarn
```

```
SparkProject-1.0.jar
```

**The --master option
allows to specify the
master URL for a
distributed cluster**

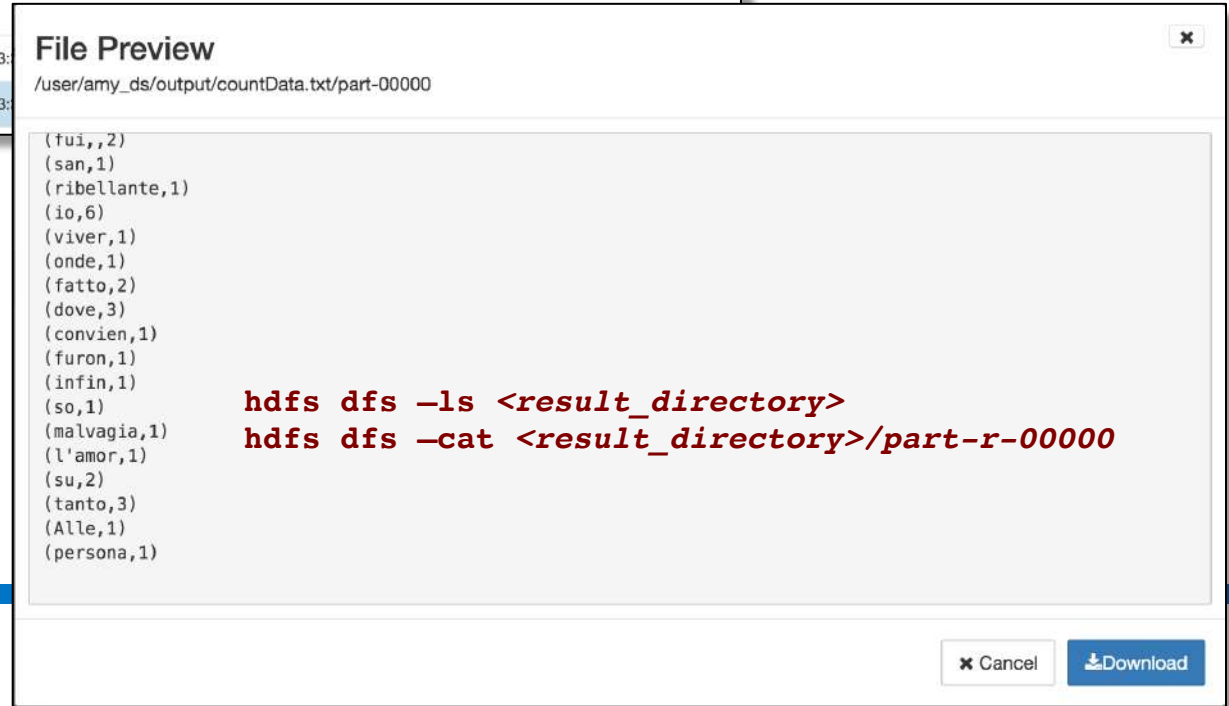


WordCount: visualizzazione dei risultati



The screenshot shows a 'Files View' window for a user named 'amy_ds' in a 'Sandbox' environment. The breadcrumb path is '> output > recent_results_wo...'. A status bar indicates '1 Files, 0 Folders selected'. The toolbar includes buttons for Open, Rename, Permissions, Delete, Copy, Move, Download, and concatenate. The 'Open' button is highlighted with a red box. Below the toolbar is a table with columns: Name, Size, Last Modified, Owner, Group, Permission, Erasure Coding, and Encrypted. The table contains two entries: '_SUCCESS' and 'part-r-00000', both 0.1 kB and dated 2019-12-05 13:30. The 'part-r-00000' file is selected.

Name	Size	Last Modified	Owner	Group	Permission	Erasure Coding	Encrypted
_SUCCESS	0.1 kB	2019-12-05 13:30					
part-r-00000	0.1 kB	2019-12-05 13:30					



The screenshot shows a 'File Preview' window for the file '/user/amy_ds/output/countData.txt/part-00000'. The preview displays a list of words and their counts in parentheses. The text is as follows:

```
(fui,,2)
(san,1)
(ribellante,1)
(io,6)
(viver,1)
(onde,1)
(fatto,2)
(dove,3)
(convien,1)
(furon,1)
(infin,1)
(so,1)
(malvagia,1)
(l'amor,1)
(su,2)
(tanto,3)
(Alle,1)
(persona,1)
```

Below the preview, there are two red text commands:

```
hdfs dfs -ls <result_directory>
hdfs dfs -cat <result_directory>/part-r-00000
```

At the bottom right of the window are 'Cancel' and 'Download' buttons.



Chi sta usando Apache Spark?



NTT DATA



Alcune letture interessanti

- Un tutorial introduttivo su Apache Spark: <https://data-flair.training/blogs/spark-tutorial/>
- Un tutorial sull'uso di Apache Spark in Python: <https://www.dezyre.com/apache-spark-tutorial/pyspark-tutorial>

