

# Spark

# Da MapReduce a Spark

- Due le principali limitazioni di MapReduce
  - Difficile programmare direttamente con MR
    - Diversi problemi non sono semplici da descrivere con MR
  - Colli di bottiglia nelle prestazioni
    - Persistenza dei dischi è più lenta del lavoro in memoria principale
- Quindi MR non si adatta perfettamente a grosse applicazioni
  - Molte volte si ha l'esigenza di concatenare più passaggi di map-reduce

# Sistemi basati su Data-Flow

- MapReduce utilizza due "ranghi" di compiti: Uno per Map il secondo per Reduce
  - I dati passano dal primo al secondo.
- I sistemi di flusso di dati generalizzano questo in due modi:
  1. Consentire qualsiasi numero di attività / gradi;
  2. Consentono funzioni diverse da Map e Reduce;
  3. Finché il flusso di dati è solo in una direzione, possiamo avere la proprietà di blocco e consentire il recupero di compiti piuttosto che lavori completi.

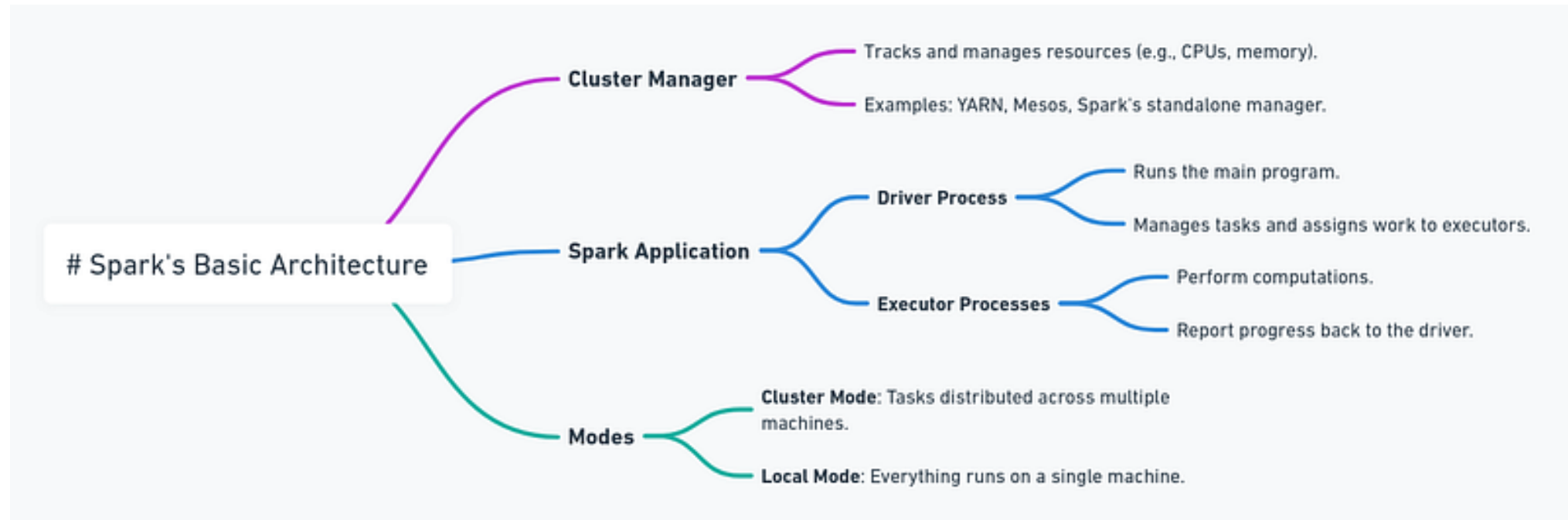
# Spark: popolare sistema data-flow

- Sistema di calcolo espressivo, non limitato al modello map-reduce
- Aggiunte al modello MapReduce:
  - Condivisione veloce dei dati
    - Evita di salvare risultati intermedi su disco;
    - Caching di dati per query ripetitive (ad es. machine learning).
  - Grafi di esecuzione generali (DAG)
  - Funzioni più ricche di Map e Reduce
  - Compatibile con Hadoop

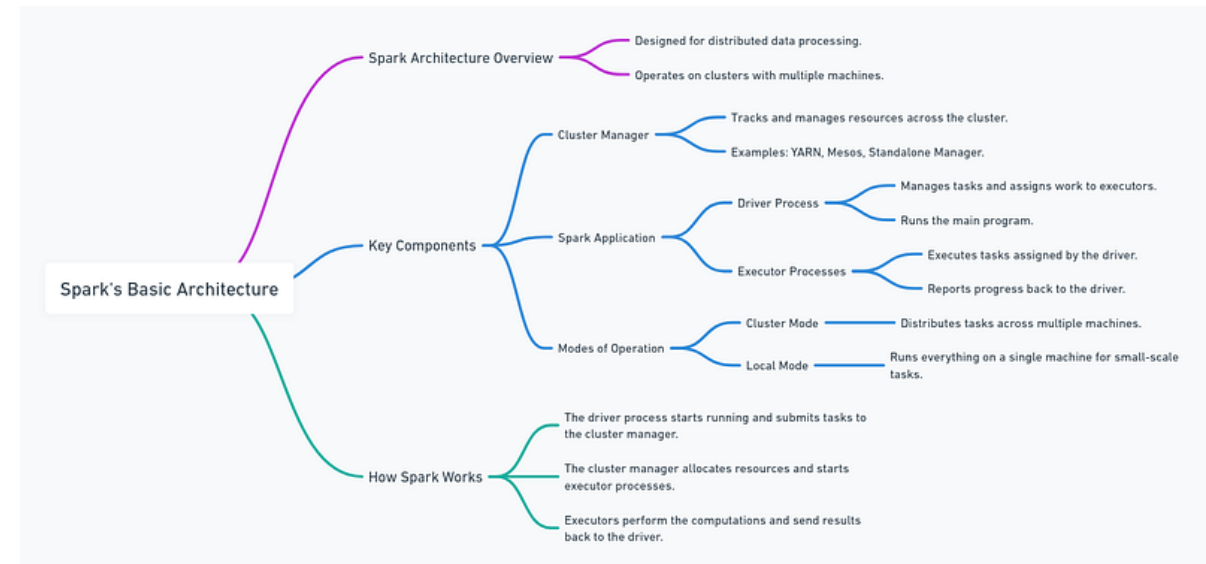
# Spark: Overview

- Software open source
- Supporta Java, Scala, Python e R
- Il costrutto chiave: Resilient Distributed Dataset (RDD)
- Api di alto livello: DataFrame & Dataset
  - Introdotte in una versione recente di Spark
  - Differenti API per aggregare i dati che supportano SQL

# Architettura SPARK



- Un cluster risolve un problema computazionalmente impossibile per una macchina utilizzando più computer (o nodi) per condividere il carico.
- Tuttavia, avere un gruppo di macchine non è sufficiente. Abbiamo bisogno di un framework come Apache Spark per gestire e assegnare compiti a queste macchine



# Componenti chiave di SPARK

## 1. Gestione cluster

- Uno strumento (come YARN, Mesos o il gestore autonomo di Spark) che tiene traccia e gestisce le risorse (ad esempio, CPU e memoria) nel cluster.

## 2. Applicazione Spark

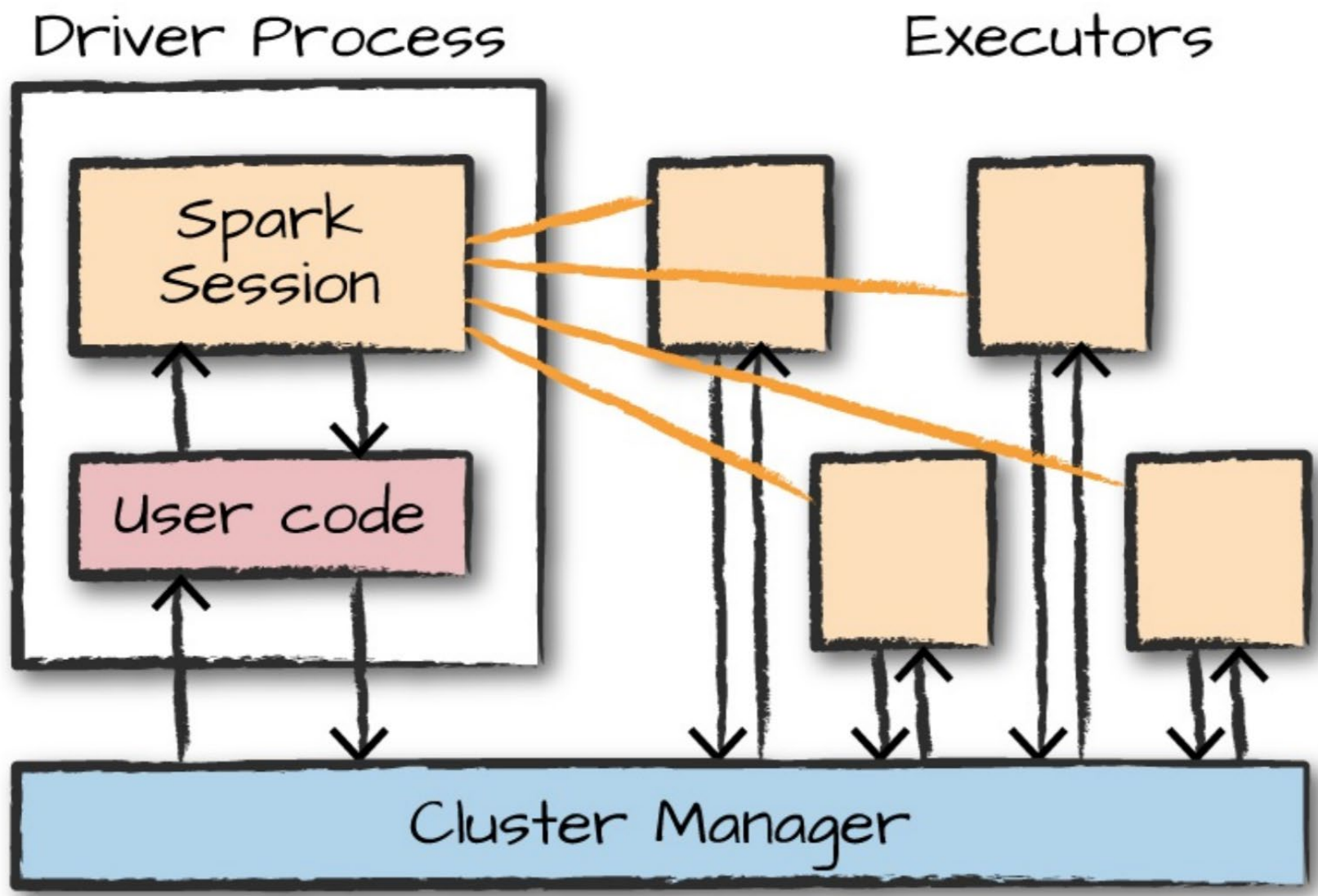
- Questo è il nostro programma che Spark eseguirà. Si compone di:
  - Driver Process: Il cervello dell'applicazione che: Esegue il nostro programma principale. Gestisce le attività e assegna il lavoro agli esecutori.
  - Processi esecutori: eseguono i calcoli effettivi e segnalano lo stato di avanzamento al driver.

## 3. Modalità

- Modalità cluster: le attività vengono distribuite su più computer.
- Modalità locale: tutto funziona su una singola macchina (ideale per l'apprendimento e le piccole attività).







# Linguaggi

- Scala
  - Spark è principalmente scritto in Scala, che quindi è il suo linguaggio di default
- Java
  - Si può programmare anche in Java
- Python
  - Python supporta tutti i costrutti che supporta Scala. C'è pure una API.
- SQL
  - Spark supporta un sottoinsieme dello standard ANSI SQL 2003.
- R
  - Spark ha due librerie in R: una come parte del core Spark (SparkR) e un pacchetto R (sparklyr).

# Spark:RDD

- Il costrutto chiave: Resilient Distributed Dataset (RDD)
  - Collezione partizionata di record
  - Generalizza le coppie (key-value)

(a , 10.3)	RDD[(String,Double)]
(b , 15,7)	
(b , 19.1)	
(c , 21.6)	

- Distribuiti nel cluster: read-only

# Spark:RDD

- Tecniche di partizionamento:
  - HashPartitioner: chiavi con lo stesso hashcode stanno nella stessa partizione
  - RangePartitioner: ordina i dati sulla base della chiave e divide i record nel numero di partizioni specificate.
  - CustomPartitioner
- Gli RDD possono essere creati da Spark o da trasformazione di altri RDD (è possibile impilare gli RDD)
- Gli RDD sono più adatti per le applicazioni che applicano la stessa operazione a tutti gli elementi di un set di dati.

# Spark:RDD

- Persistenza e Caching:
  - Salvataggio di dati intermedi da riutilizzare in stage successivi
  - `cache()` memorizza i risultati intermedi soltanto sulla memoria
  - `persistent()` invece mette a disposizione ulteriori opzioni:

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as <i>serialized</i> Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a <a href="#">fast serializer</a> , but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.

# Spark: Operazioni sugli RDD

- **Trasformazioni**: costruisce RDD attraverso operazioni deterministiche su altri RDD
  - Trasformazioni includono: **map, filter, join, union, intersection, distinct**
  - Valutazione pigra: nulla è calcolato fino a quando una azione non lo richiede
- **Azioni per restituire valori** o esportare dati
  - Azioni includono: **count, collect, reduce, save**
  - Azioni applicate agli RDD forzano il calcolo e la restituzione di un nuovo RDD
- **Variabili condivise**:
  - Broadcast: variabile in sola lettura cached su ogni macchina del cluster.
  - Accumulators: utilizzate per creare un counter condiviso tra le macchine del cluster.

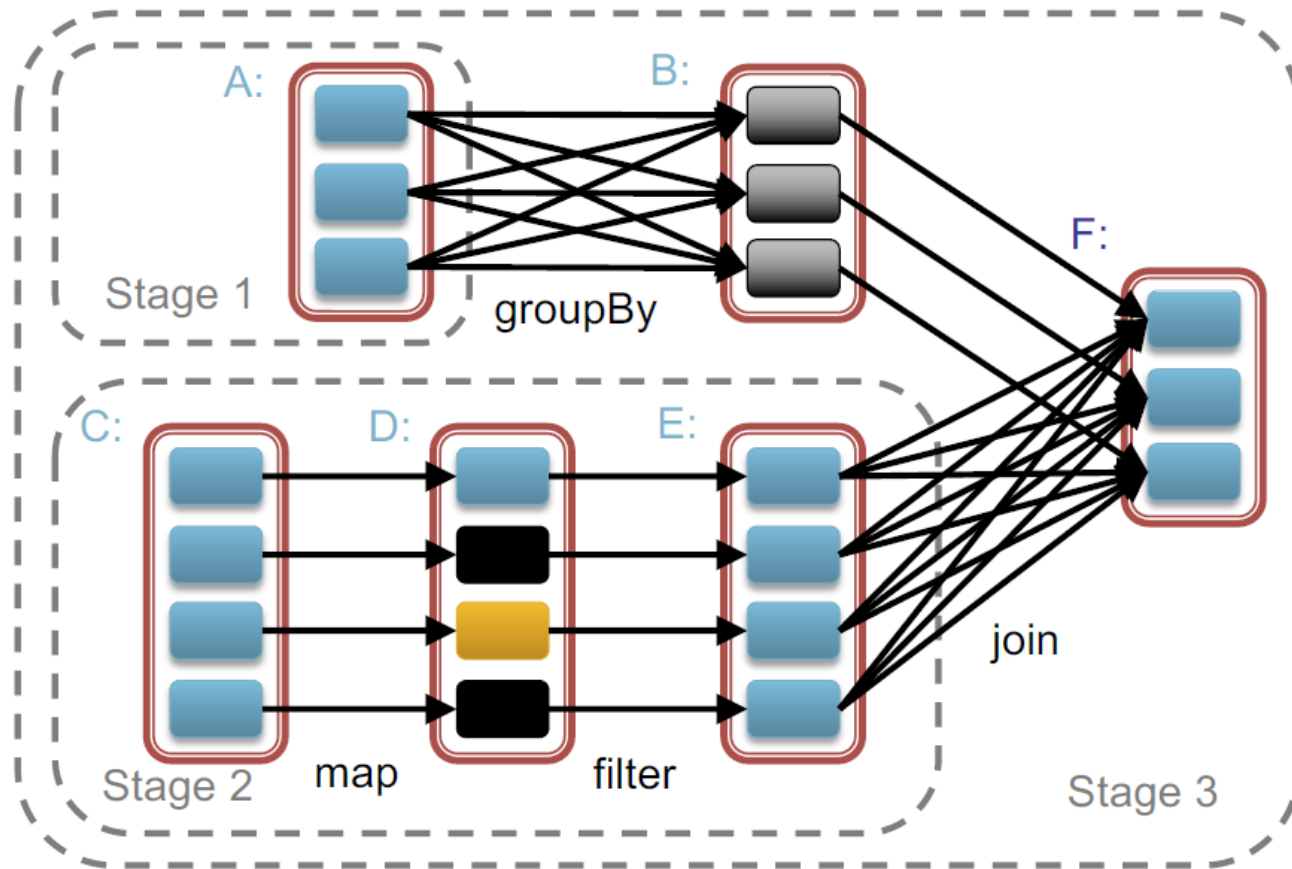
# Alcune operazioni da vicino

- **take(n)** – restituisce i primi n elementi di un RDD sottoforma di array.
- **collect()** – ritorna tutti gli elementi di un RDD come un array (usare con cautela);
- **count()** – ritorna il numero degli elementi di un RDD come int.
- **saveAsTextFile('path/to/dir')** – salva l'RDD in file in una directory. Crea la directory se non esiste e fallisce se questa già esiste.
- **foreach(func)** – esegue la funzione per ogni elemento dell'RDD ma non mantiene nessun risultato.



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

# Task scheduler: DAG generali



- Supporta grafi di attività generali
- Le pipeline funzionano dove possibile
- Cache-aware data reuse e locality dei dati
- Partitioning-aware per evitare shuffle

= RDD

= cached partition

# RDD Esempio:

- Creazione di un RDD a partire da un vettore di 5 interi:

- Scala

```
val data = Array(1, 2, 3, 4, 5)
```

```
val distData = sc.parallelize(data)
```

```
val conf = new SparkConf().setAppName(appName).setMaster(master)
new SparkContext(conf)
```

- Java:

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
```

```
JavaRDD<Integer> distData = sc.parallelize(data);
```

```
SparkConf conf = new SparkConf().setAppName(appName).setMaster(master);
JavaSparkContext sc = new JavaSparkContext(conf);
```

- Python:

```
data = [1, 2, 3, 4, 5]
```

```
distData = sc.parallelize(data)
```

```
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)
```

Demo

Esercitazione1 Spark

\$start-dfs.sh

start-master.sh

start-slave.sh spark://localhost:7077

localhost:50070

HadoopOverviewDatanodesDatanode Volume FailuresSnapshotStartup ProgressUtilities

Overview 'localhost:9000' (active)

Started:

Tue May 05 12:12:52 CEST 2020

Version:

2.7.6, r085099c66c728be31604560c376fa282e69282b8

Compiled:

2018-04-18T01:33Z by kshvachk from branch-2.7.6

Cluster ID:

CID-65666c6e-7dff-4a7b-9bd8-510c8daac3dd

Block Pool ID:

BP-72294067-127.0.1.1-1552503390688

Summary

Security is off.

Safemode is off.

275 files and directories, 207 blocks = 482 total filesystem object(s).

Heap Memory used 156.92 MB of 370 MB Heap Memory. Max Heap Memory is 889 MB.

Non Heap Memory used 48.01 MB of 49.25 MB Committed Non Heap Memory. Max Non Heap Memory is -1 B.

Configured Capacity:

460.69 GB

SPARK

Spark

2.4.1

Spark Master at spark://localhost:7077

URL: spark://localhost:7077  
**Alive Workers:** 1  
**Cores in use:** 4 Total, 0 Used  
**Memory in use:** 8.0 GB Total, 0.0 B Used  
**Applications:** 0 Running, 0 Completed  
**Drivers:** 0 Running, 0 Completed  
**Status:** ALIVE

Workers (1)

Worker Id	Address	State	Cores	Memory
worker-20200506121840-192.168.111.193-55351	192.168.111.193:55351	ALIVE	4 (0 Used)	8.0 GB (0.0 B Used)

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	----------------	------	-------	----------

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	----------------	------	-------	----------

```
pyspark --master spark://localhost:7077
```

```
alfredo@DESKTOP-CHEP0HH: ~
```

```
alfredo@DESKTOP-CHEP0HH:~$ pyspark --master spark://localhost:7077
Python 2.7.15rc1 (default, Nov 12 2018, 14:31:15)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
20/05/06 12:29:15 WARN util.Utils: Your hostname, DESKTOP-CHEP0HH resolves to a loopback address:
127.0.1.1; using 192.168.111.193 instead (on interface eth1)
20/05/06 12:29:15 WARN util.Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Welcome to
```

[illegible]

```
Using Python version 2.7.15rc1 (default, Nov 12 2018 14:31:15)
SparkSession available as 'spark'.
>>>
```

# map()

- Applica una operazione ad ogni elemento di un RDD e restituisce un nuovo RDD che contiene il risultato

```
>>> d1 = sc.textFile('/test.txt')
>>> d1.take(3)
[u'alfredo,pulvirenti', u'antonio,dimaria', u'salvatore,alaimo']
>>> d1.map(lambda line: line.split(',')).take(3)
[[u'alfredo', u'pulvirenti'], [u'antonio', u'dimaria'], [u'salvatore', u'alaimo']]
>>>
```

# flatMap()

- Applica l'operazione ad ogni elemento di un RDD e restituisce un nuovo RDD che contiene i risultati dopo che rimuove il container piu' esterno

```
>>> d1 = sc.textFile('/test1.txt')
>>> d1.flatMap(lambda line: line.split(',')).take(6)
[u'alfredo', u'pulvirenti', u'antonio', u'dimaria', u'salvatore', u'alaimo']
>>>
```



# mapValues()

- Applica l'operazione ad ogni elemento di un RDD e restituisce un nuovo RDD che contiene i risultati. Funziona solo su RDD a coppia

```
>>> d1 = sc.textFile('/test1.txt')
>>> d1 = d1.map(lambda line: line.split(','))
>>> d1 = d1.map(lambda pair: (pair[1],pair[0]))
>>> d1.take(3)
[(u'pulvirenti', u'alfredo'), (u'dimaria', u'antonio'), (u'alaimo', u'salvatore')]
>>> d1.mapValues(lambda name: name.upper()).take(3)
[(u'pulvirenti', u'ALFREDO'), (u'dimaria', u'ANTONIO'), (u'alaimo', u'SALVATORE')]
>>> _
```

# flatMapValues()

- Stringa come vettore di caratteri

```
>>> d1 = sc.textFile('/test1.txt')
>>> d1 = d1.map(lambda line: line.split(','))
>>> d1 = d1.map(lambda pair: (pair[0],pair[1]))
>>> d1.take(3)
[(u'alfredo', u'pulvirenti'), (u'antonio', u'dimaria'), (u'salvatore', u'alaimo')]
>>> d1.flatMapValues(lambda name: name.upper()).take(3)
[(u'alfredo', u'P'), (u'alfredo', u'U'), (u'alfredo', u'L')]
>>>
>>>
```

# filter()

- Applichiamo un filtro con una espressione regolare

```
>>> import re
>>> d1 = sc.textFile('/test1.txt')
>>> d1.take(6)
[u'alfredo,pulvirenti', u'antonio,dimaria', u'salvatore,alaimo', u'giovanni,pulvirenti', u'mario,pulvirenti', u'filippo,dimaria']
>>> d1.filter(lambda line: re.match(r'^[as]', line)).collect()
[u'alfredo,pulvirenti', u'antonio,dimaria', u'salvatore,alaimo']
>>>
```

# groupByKey()

```
>>> d1 = sc.textFile('/test1.txt')
>>> d1 = d1.map(lambda line: line.split(','))
>>> d1 = d1.map(lambda pair: (pair[1],pair[0]))
>>> d1.take(3)
[(u'pulvirenti', u'alfredo'), (u'dimaria', u'antonio'), (u'alaimo', u'salvatore')]
>>> d1.groupByKey().take(1)
[(u'dimaria', <pyspark.resultiterable.ResultIterable object at 0x7f8e02a74c90>)]
>>> for pair in d1.groupByKey().take(1):
..     print "%s:%s" % (pair[0],",".join([n for n in pair[1]]))
..
dimaria:antonio,filippo
>>>
```

# reduceByKey()

```
>> d1 = sc.textFile('/test1.txt')
>> d1 = d1.map(lambda line: line.split(','))
>> d1 = d1.map(lambda pair: (pair[1], pair[0]))
>> d1.take(3)
(u'pulvirenti', u'alfredo'), (u'dimaria', u'antonio'), (u'alaimo', u'salvatore')]
>> d1.reduceByKey(lambda v1,v2: v1+":"+v2).take(1)
(u'dimaria', u'antonio:filippo')]
```

# join()

```
>>> d1 = sc.textFile('/test.txt').map(lambda line: line.split(',')).map(lambda pair: (pair[0], pair[1]))
>>> d1.take(3)
[(u'alfredo', u'pulvirenti'), (u'antonio', u'dimaria'), (u'salvatore', u'alaimo')]
>>> d2 = sc.textFile('/test1.txt').map(lambda line: line.split(',')).map(lambda pair: (pair[0], pair[1]))
>>> d2.take(3)
[(u'alfredo', u'pulvirenti'), (u'antonio', u'dimaria'), (u'salvatore', u'alaimo')]
>>> d1.join(d2).collect()
[(u'alfredo', (u'pulvirenti', u'pulvirenti')), (u'salvatore', (u'alaimo', u'alaimo')), (u'antonio', (u'dimaria', u'dimaria'))]
>>> d1.fullOuterJoin(d2).take(2)
[(u'alfredo', (u'pulvirenti', u'pulvirenti')), (u'roberto', (None, u'alaimo'))]
>>> _
```

# subtract()

```
>>> d1 = sc.textFile('/test1.txt')
>>> d1.take(3)
[u'alfredo,pulvirenti', u'antonio,dimaria', u'salvatore,alaimo']
>>> d2 = sc.textFile('/test.txt')
>>> d2.take(3)
[u'alfredo,pulvirenti', u'antonio,dimaria', u'salvatore,alaimo']
>>> d1.subtract(d2).take(3)
[u'rosario,alaimo', u'filippo,alaimo', u'giovanni,alaimo']
>>>
```

sortByKey



# RDD Esempio:

- Conteggio caratteri di un file:

- Scala

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

- Java:

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

- Python:

```
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
totalLength = lineLengths.reduce(lambda a, b: a + b)
```

## RDD Esempio: Word Count



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

```
val textFile = sc.textFile("data.txt")
val counts = textFile.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
counts.saveAsTextFile("out.txt")
```



```
JavaRDD<String> textFile = sc.textFile("data.txt");
JavaPairRDD<String, Integer> counts = textFile .
    flatMap(s -> Arrays.asList(s.split(" ")).iterator()) .
    mapToPair(word -> new Tuple2<>(word, 1)) .
    reduceByKey((a, b) -> a + b);
counts.saveAsTextFile("out.txt");
```

# Spark: DataFrame e Dataset

- DataFrame

- Diversamente dagli RDD, dati organizzati in colonne con nome (es. Tabelle di un db relazionale)
- Impone una struttura su una collezione distribuita di dati, consentendo l'astrazione a un piu' alto livello
- Possono essere costruiti da diverse sorgenti: file parquet, json, cvs, database esterni, ecc.
- Lo Spark SQL Catalyst optimizer garantisce una notevole riduzione dei tempi di esecuzione delle query: il tempo impiegato per effettuare una join o groupBy è nettamente inferiore rispetto al tempo impiegato per effettuare le stesse operazioni utilizzando gli RDD.

# DataFrame e Dataset

- DataFrame: SparkSession
  - Per poter creare un DataFrame, bisogna inizialmente creare e configurare una SparkSession:

```
import org.apache.spark.sql.SparkSession
```



```
val spark = SparkSession  
  .builder()  
  .appName("Spark SQL basic example")  
  .config("spark.some.config.option", "some-value")  
  .getOrCreate()
```

```
from pyspark.sql import SparkSession
```

```
spark = SparkSession \  
  .builder \  
  .appName("Python spark SQL basic example") \  
  .config("spark.some.config.option", "some-value") \  
  .getOrCreate()
```



```
import org.apache.spark.sql.SparkSession;  
  
SparkSession spark = SparkSession  
  .builder()  
  .appName("Java Spark SQL basic example")  
  .config("spark.some.config.option", "some-value")  
  .getOrCreate();
```

# DataFrame e Dataset

- DataFrame: Esempio



```
val df = spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout
df.show()
// +----+-----+
// | age|  name|
// +----+-----+
// |null|Michael|
// | 30|   Andy|
// | 19|  Justin|
// +----+-----+
```

```
// Print the schema in a tree format
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)
```

```
// Select people older than 21
df.filter($"age" > 21).show()
```

```
// Count people by age
df.groupBy("age").count().show()
```



```
# spark is an existing SparkSession
df = spark.read.json("examples/src/main/resources/people.json")
# Displays the content of the DataFrame to stdout
df.show()
# +----+-----+
# | age|  name|
# +----+-----+
# |null|Michael|
# | 30|   Andy|
# | 19|  Justin|
# +----+-----+
```

```
# Print the schema in a tree format
df.printSchema()
# root
# |-- age: long (nullable = true)
# |-- name: string (nullable = true)
```

```
# Select people older than 21
df.filter(df['age'] > 21).show()
```

```
# Count people by age
df.groupBy("age").count().show()
```

# DataFrame e Dataset

- DataFrame: Esempio per chi desidera usare la sintassi SQL



```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()

// +-----+-----+
// | age|   name|
// +-----+-----+
// |null|Michael|
// | 30|   Andy|
// | 19|  Justin|
// +-----+-----+
```

```
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()

# +-----+-----+
# | age|   name|
# +-----+-----+
# |null|Michael|
# | 30|   Andy|
# | 19|  Justin|
# +-----+-----+
```



# DataFrame e Dataset

- Dataset
  - Estensione dell'API DataFrame che fornisce **type-safe, object-oriented programming interface** (compile-time error detection)
  - Eredita sia i punti di forza degli RDD (tipizzazione forte, utilizzo delle funzioni lambda) che i punti di forza dei DataFrame.
  - Sui Dataset è possibile invocare sia le funzioni degli RDD (map, flatMap, filter) che quelle fornite dai DataFrame (select, where, ecc)
  - Le Dataset API sono disponibili per Java e Scala ma non per Python.
  - Un DataFrame è un Dataset di oggetti Row.
- Entrambi sono basati sul motore Spark SQL.
- Entrambi possono essere convertiti in un RDD

# DataFrame e Dataset


- Dataset: Esempio

Creazione da lista di oggetti Persona

```
case class Person(name: String, age: Long)

// Encoders are created for case classes
val caseClassDS = Seq(Person("Andy", 32)).toDS()
caseClassDS.show()


// +-----+
// |name|age|
// +-----+
// |Andy| 32|
// +-----+
```



Creazione da Json file

```
// DataFrames can be converted to a Dataset by providing a class
val path = "examples/src/main/resources/people.json"
val peopleDS = spark.read.json(path).as[Person]
peopleDS.show()

// +-----+
// | age|  name|
// +-----+
// |null|Michael|
// | 30|   Andy|
// | 19|  Justin|
// +-----+
```



```
import java.util.Arrays;
import java.util.Collections;
import java.io.Serializable;

import org.apache.spark.api.java.function.MapFunction;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Encoder;
import org.apache.spark.sql.Encoders;

public static class Person implements Serializable {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```




Creazione da lista di oggetti Persona

```
// Create an instance of a Bean class
Person person = new Person();
person.setName("Andy");
person.setAge(32);

// Encoders are created for Java beans
Encoder<Person> personEncoder = Encoders.bean(Person.class);
Dataset<Person> javaBeanDS = spark.createDataset(
    Collections.singletonList(person),
    personEncoder
);
javaBeanDS.show();


// +-----+
// |age|name|
// +-----+
// | 32|Andy|
// +-----+
```



Creazione da Json file

```
// DataFrames can be converted to a Dataset by providing a class. Map
String path = "examples/src/main/resources/people.json";
Dataset<Person> peopleDS = spark.read().json(path).as(personEncoder);
peopleDS.show();

// +-----+
// | age|  name|
// +-----+
// |null|Michael|
// | 30|   Andy|
// | 19|  Justin|
// +-----+
```





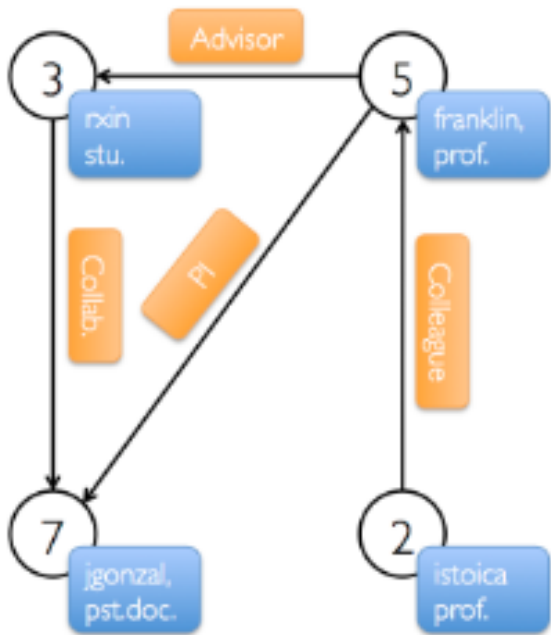
# Librerie utili per Spark

- Spark SQL
- Spark Streaming – stream processing di datastream live
- Mlib – Machine Learning
- GraphX – manipolazione grafi
  - Estende Spark RDD con Graph abstraction: multigrafo diretto con proprietà attaccate ad ogni vertice e ad ogni arco
  - Supporta archi paralleli per modellare relazioni multiple tra due stessi nodi
  - Definizione dei tipi "VD" ed "ED" per referenziare gli oggetti associati ai vertici ed archi del grafo
  - API disponibili solo in scala

# GraphX: Esempio

- Rete di collaboratori: le proprietà dei nodi contengono "username" ed occupazione, le proprietà degli archi contengono il tipo di relazione

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

```
// Assume the SparkContext has already been constructed
val sc: SparkContext

// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
    (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))

// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"), Edge(5L, 3L, "advisor"),
    Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))

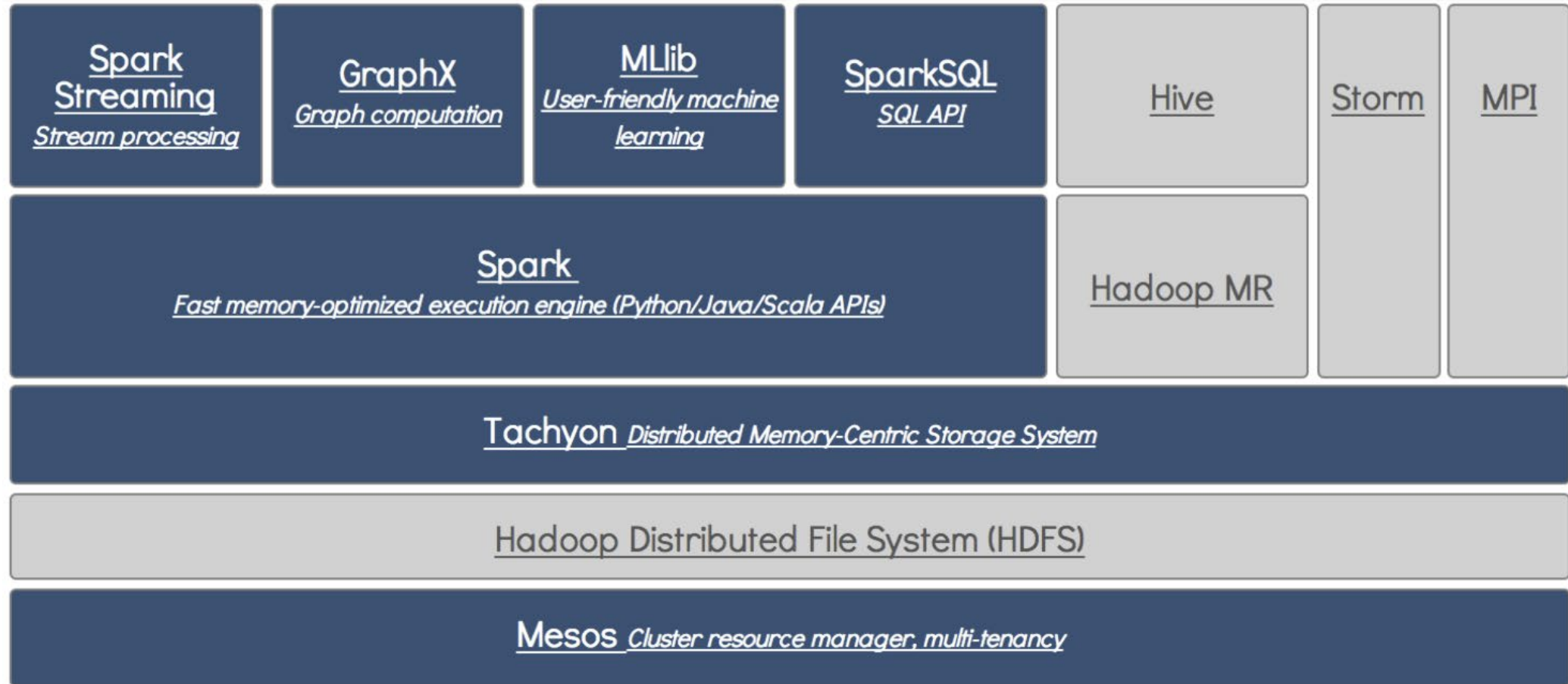
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")

// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)

// Count all users which are postdocs
graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count

// Count all the edges where src > dst
graph.edges.filter(e => e.srcId > e.dstId).count
```

# Data Analytics software stack

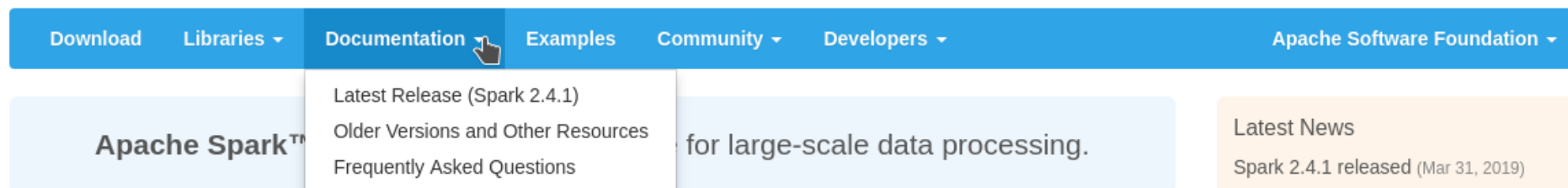


# Spark vs Hadoop MapReduce

- Performance: Spark normalmente più veloce ma:
  - Spark può elaborare i dati in memoria; Hadoop MapReduce scrive su disco dopo map reduce
  - Spark ha bisogno di molta memoria per funzionare bene; se ci sono altri servizi che richiedono risorse o che non possono essere integrati in memoria, Spark degrada
  - MapReduce funziona facilmente insieme ad altri servizi con differenze di rendimento minori e funziona bene con lavori a un passaggio per i quali è stato progettato;
- Facilità d'uso: Spark è più facile da programmare (livello superiore API)
- Elaborazione dati: Spark più generale

# Documentazione SPAK

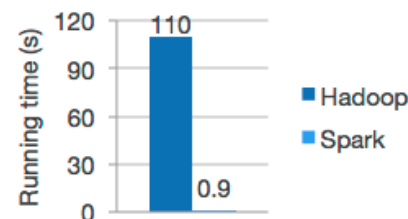
<https://spark.apache.org/>



## Speed

Run workloads 100x faster.

Apache Spark achieves high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine.



Logistic regression in Hadoop and Spark

## Latest News

Spark 2.4.1 released (Mar 31, 2019)  
Spark 2.3.3 released (Feb 15, 2019)  
Spark 2.2.3 released (Jan 11, 2019)  
Spark+AI Summit (April 23-25th, 2019, San Francisco) agenda posted (Dec 19, 2018)

[Archive](#)



[Download Spark](#)