



Primo giorno: pomeriggio

Agenda 2/8

— Le fondamenta

La programmazione funzionale

Un paradigma utile per il calcolo parallelo

Strutture dati

RDDs, DataSets e DataFrames

Operazioni sui dati

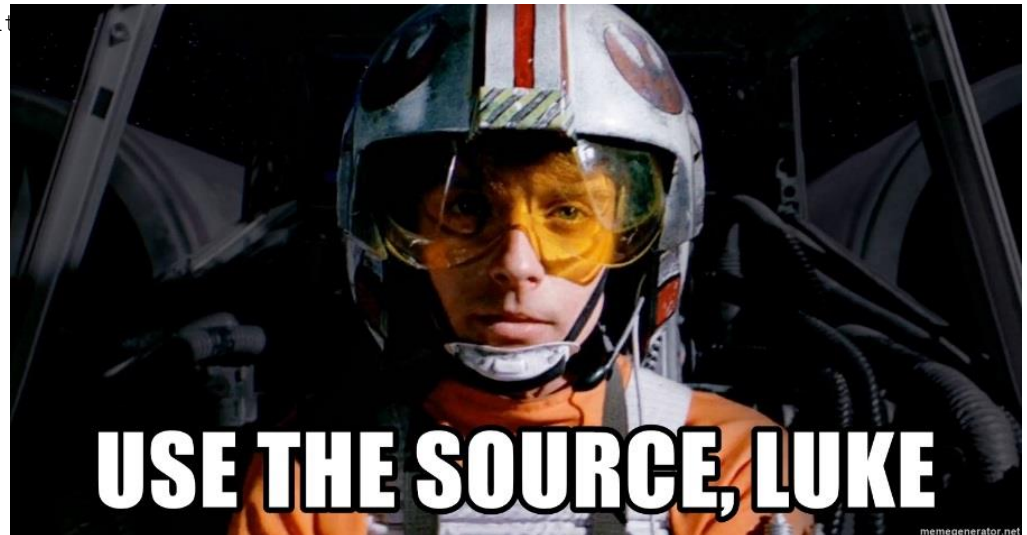
Transformazioni e Azioni; Valutazioni «pigre»; persistenza (cache)

Punti di riferimento

Documentazione; Cloudera, Hortonworks; Data Source su github

Esercizi di laboratorio

Mappe, filtri e qualche azione

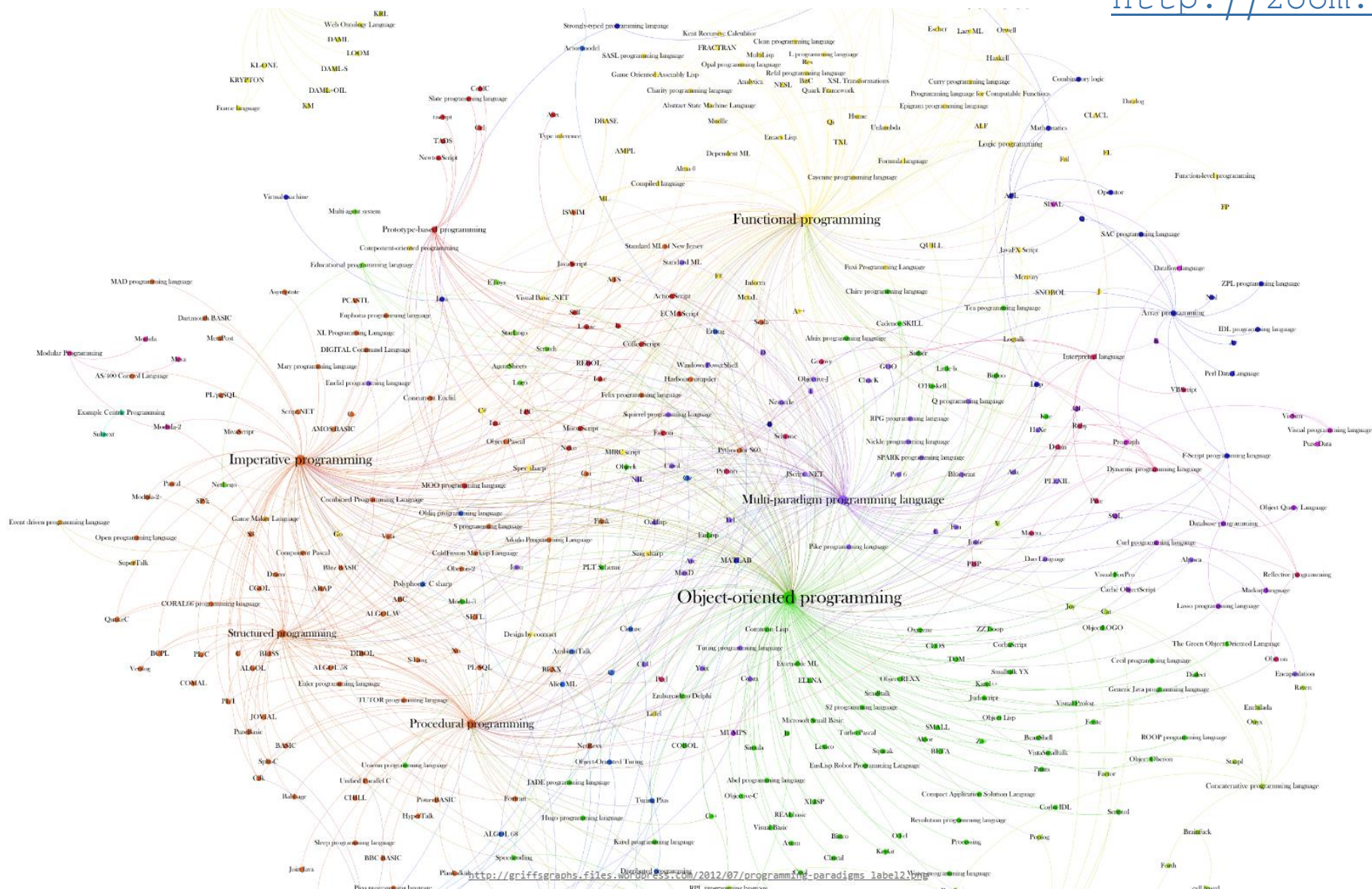


Cos'è la Programmazione e Funzionale?

Esistono più di 8000 linguaggi di programmazione

The Graph of Programming Pa

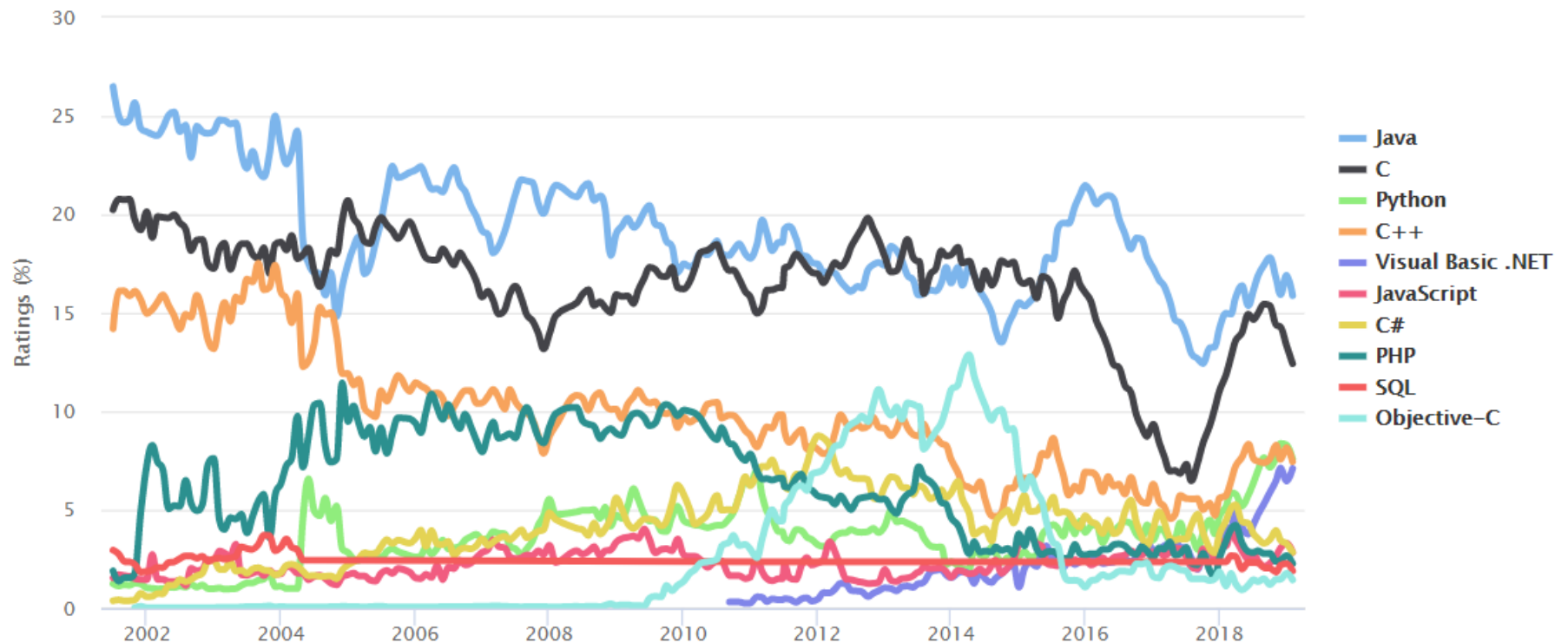
<http://zoom.it>



<https://www.tiobe.com/tiobe-index/>

TIOBE Programming Community Index

Source: www.tiobe.com



*«A language that
doesn't affect the
way you think about
programming is not
worth knowing».*

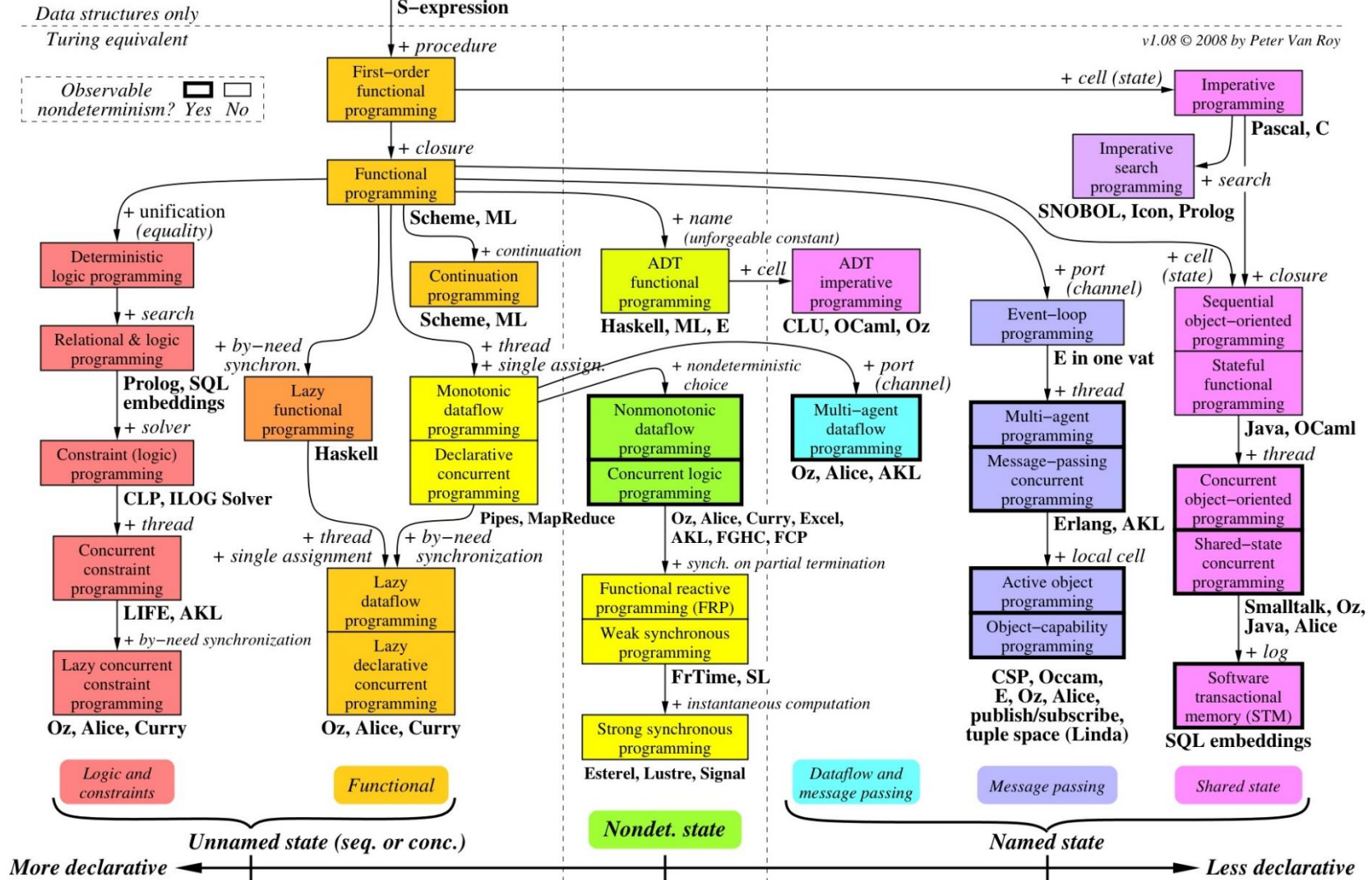


— Alan Perlis

The principal programming paradigms

"More is not better (or worse) than less, just different."

v1.08 © 2008 by Peter Van Roy



Paradigmi di programmazione : *procedurale* vs. *funzionale*

Programmazione procedurale o imperativa

Un programma è una serie di istruzioni che possono modificare la memoria.

Programmazione funzionale

Il programma è una funzione che calcola un output a partire da un certo numero di input.

Paradigmi di programmazione : *procedurale* vs. *funzionale*

Programmazione procedurale o imperativa

```
val arr = Seq(1, 2, 3, 5, 8, 13, 21)
var sum = 0
for ( x <- arr ) sum += x
```

Programmazione funzionale

```
val arr = Seq(1, 2, 3, 5, 8, 13, 21)
val sum = arr.reduce((a,b) => a+b)
```

Caratteristiche del paradigma funzionale

- Niente effetti collaterali (*side effects*)
- Non importa l'ordine di esecuzione
- Le funzioni vengono passate come argomento

*Ideas from Functional Programming
that improve software design:*



**AVOID HIDDEN
SIDE-EFFECTS**



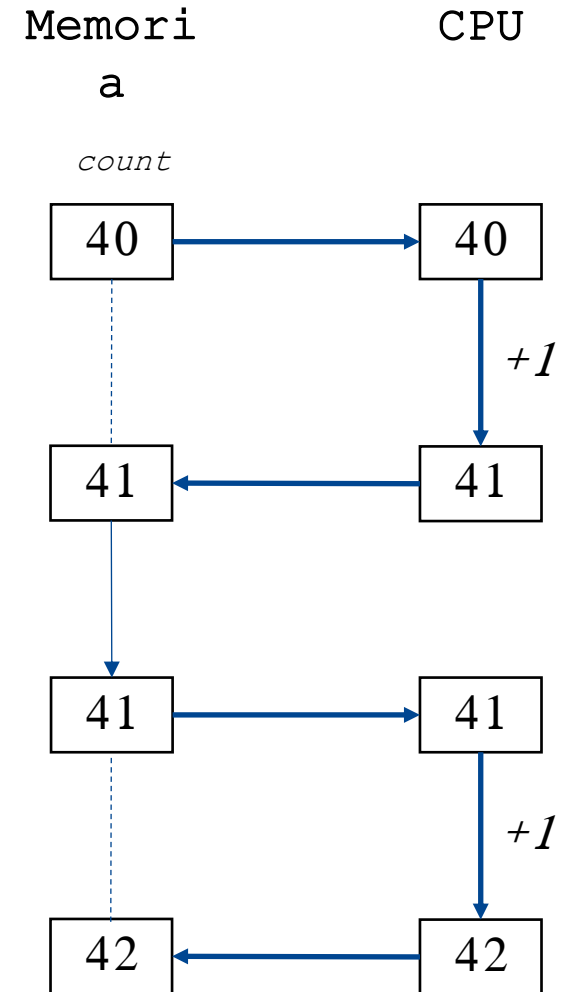
**ENSURE REFERENTIAL
TRANSPARENCY**



**USE LIST-PROCESSING
FUNCTIONS**

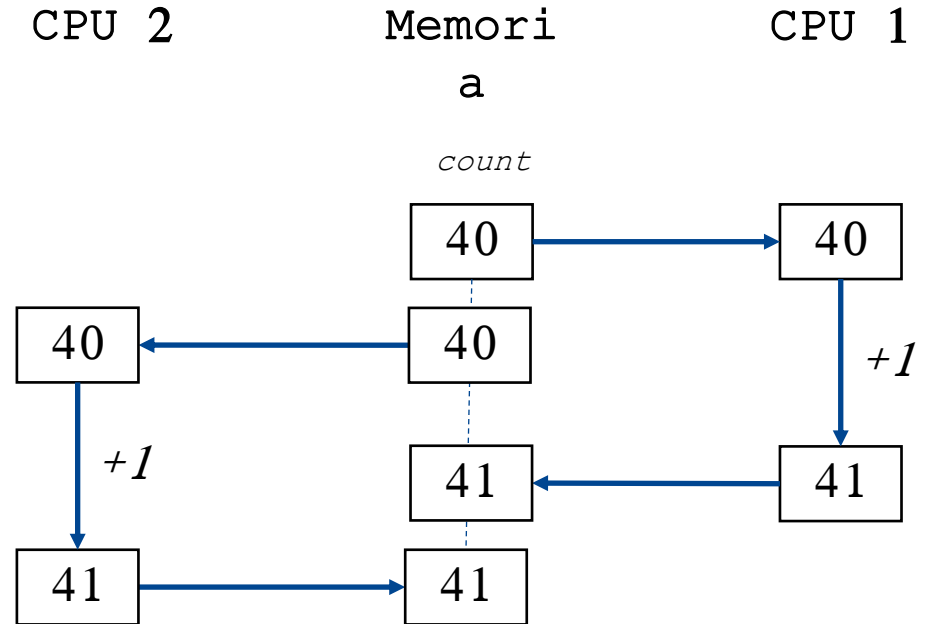
Gli effetti collaterali

```
for( item <- items ) {  
  ...  
  if ( condizione ) {  
    count = count + 1  
  }  
}
```



Gli effetti collaterali

```
for( item <- items ) {  
  ...  
  if ( condizione ) {  
    count = count + 1  
  }  
}
```



ERRORE !

Gli effetti collaterali

Se sono ammessi effetti collaterali una funzione invocata due volte può ritornare v

```
var i = 0  
def pippo() : Int = { i+=1; return i }
```


Conclusione dell'intermezzo

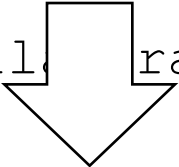
Il paradigma funzionale è utile nei sistemi di calcolo distribuito

Strutture dati

immutabili

Assenza di effetti

collaterali

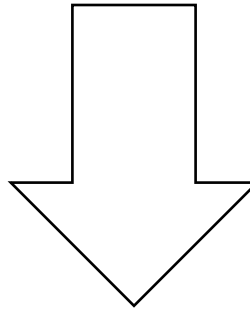


E' più facile:

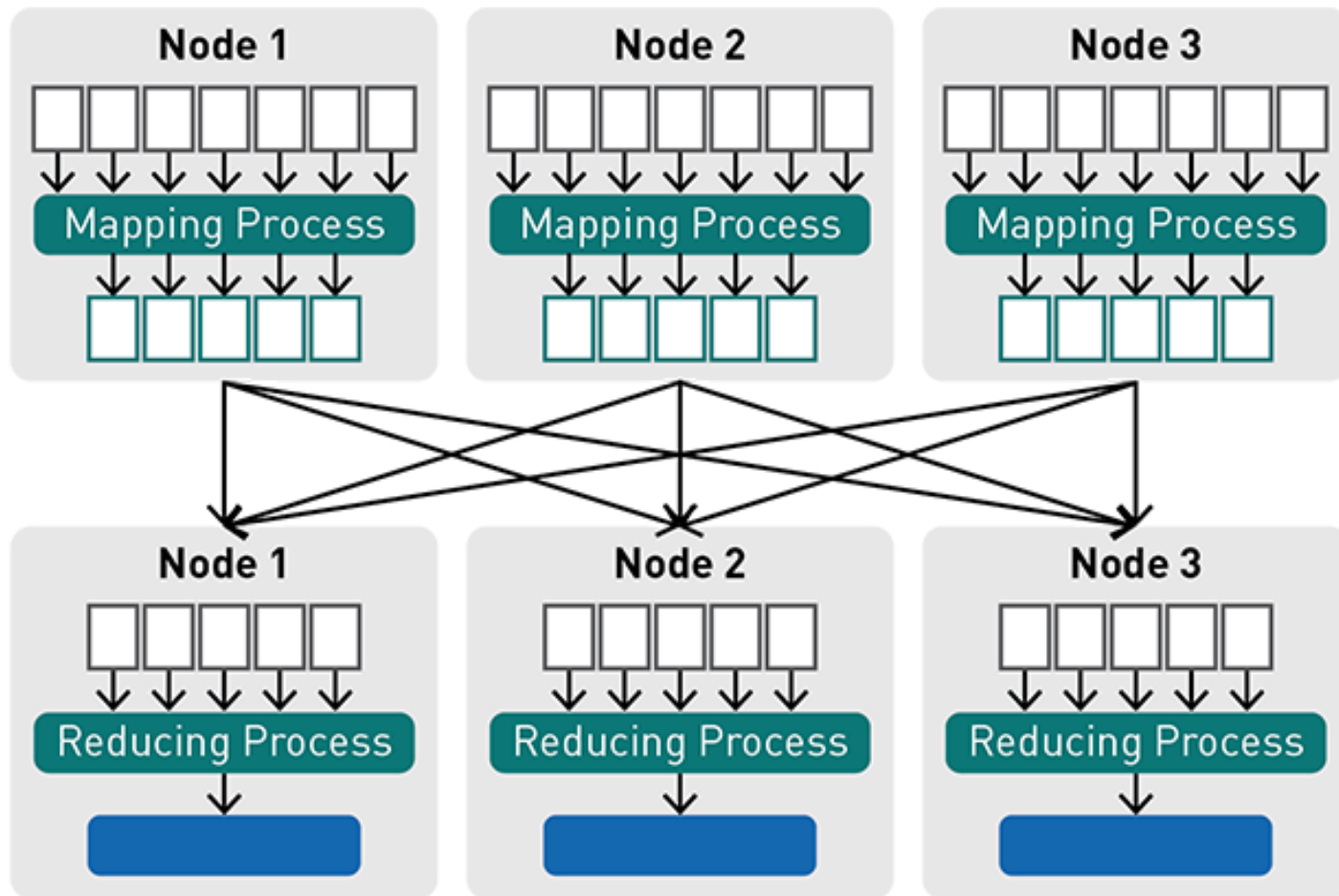
- Distribuire i dati fra i nodi
- Ricalcolare un dato intermedio in caso di fallimento di un nodo
- Ottimizzare il flusso dei calcoli

Torniamo a Spark!

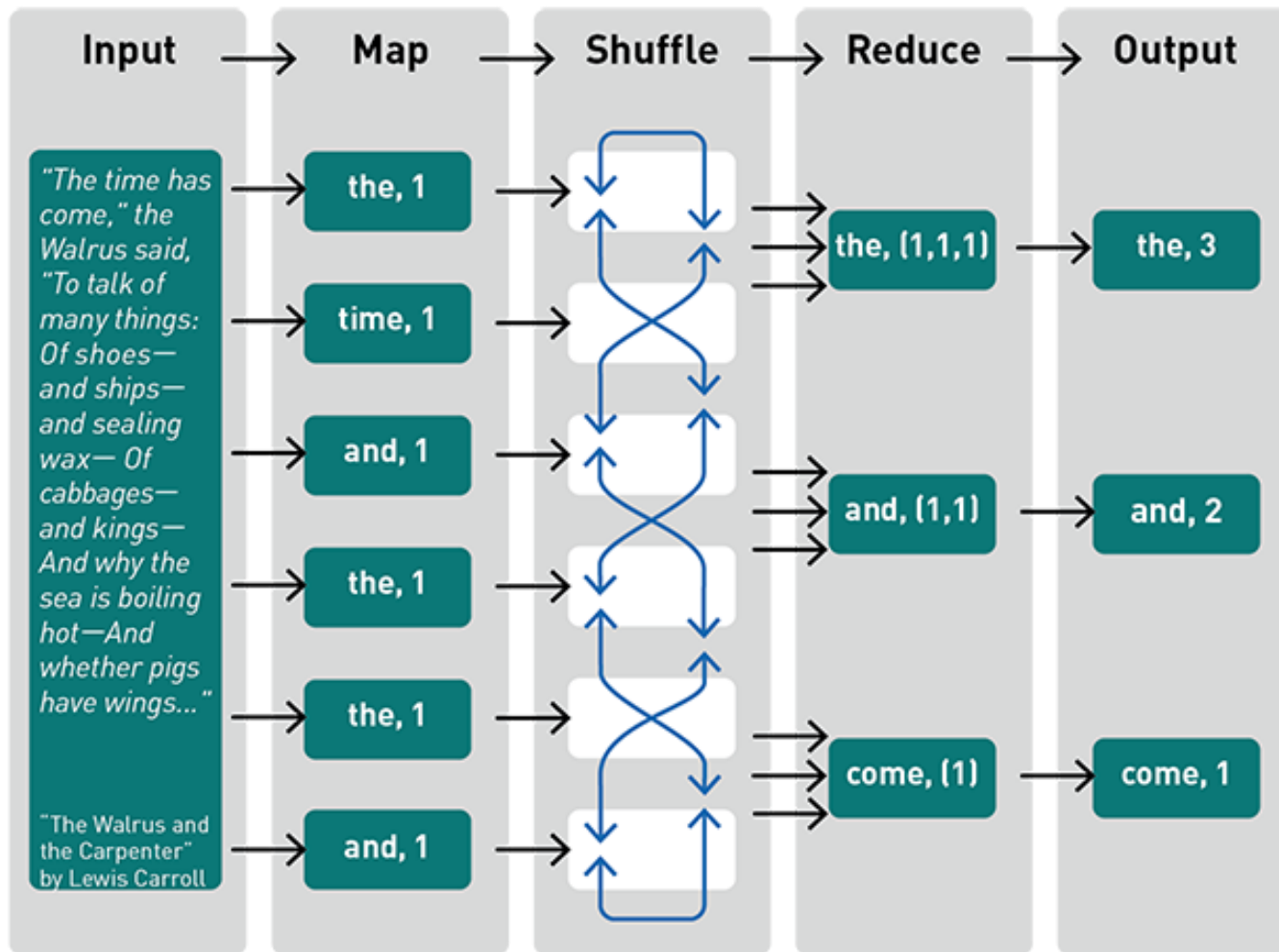
Spark è una diretta evoluzione di Hadoop Map Reduce



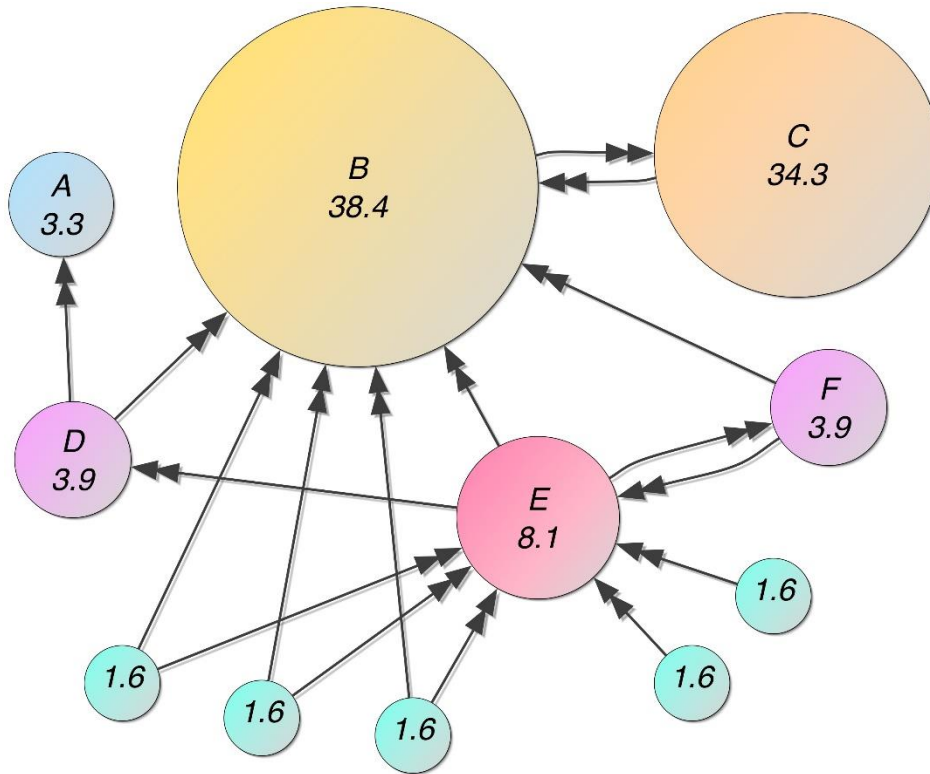
MapReduce



MapReduce



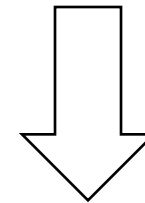
Un esempio di calcolo più complesso: PageRank



$$\text{PageRank of site} = \sum \frac{\text{PageRank of inbound link}}{\text{Number of links on that page}}$$

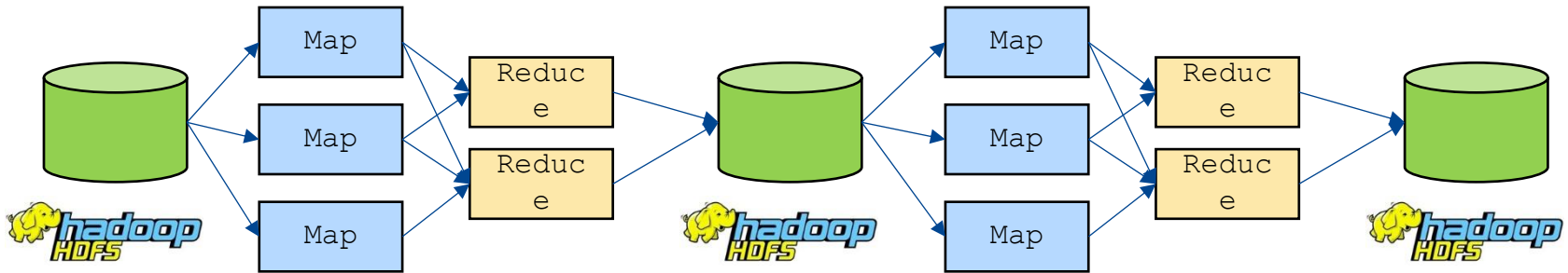
OR

$$PR(u) = (1 - d) + d \times \sum \frac{PR(v)}{N(v)}$$



Algoritmo iterativo

MapReduce



Se l'algoritmo prevede diverse fasi di MapReduce i dati intermedi devono essere scritti su disco ogni volta

Spark vs MapReduce

Spark è una diretta evoluzione di Hadoop Map Reduce

Mantiene il concetto base: Flusso di dati attraverso un grafo di operazioni

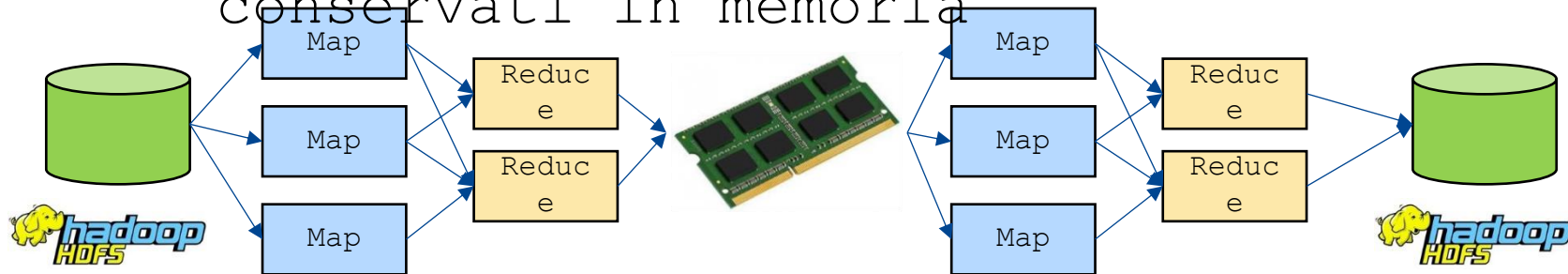
Le principali differenze sono:

- I dati intermedi sono conservati in memoria
- Il grafo delle operazioni (**DAG**= *Direct Acyclic Graph*) può essere più complesso

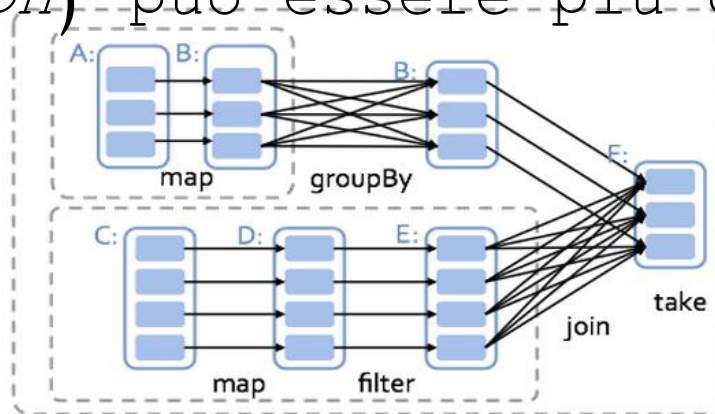


Spark

I dati intermedi sono
conservati in memoria



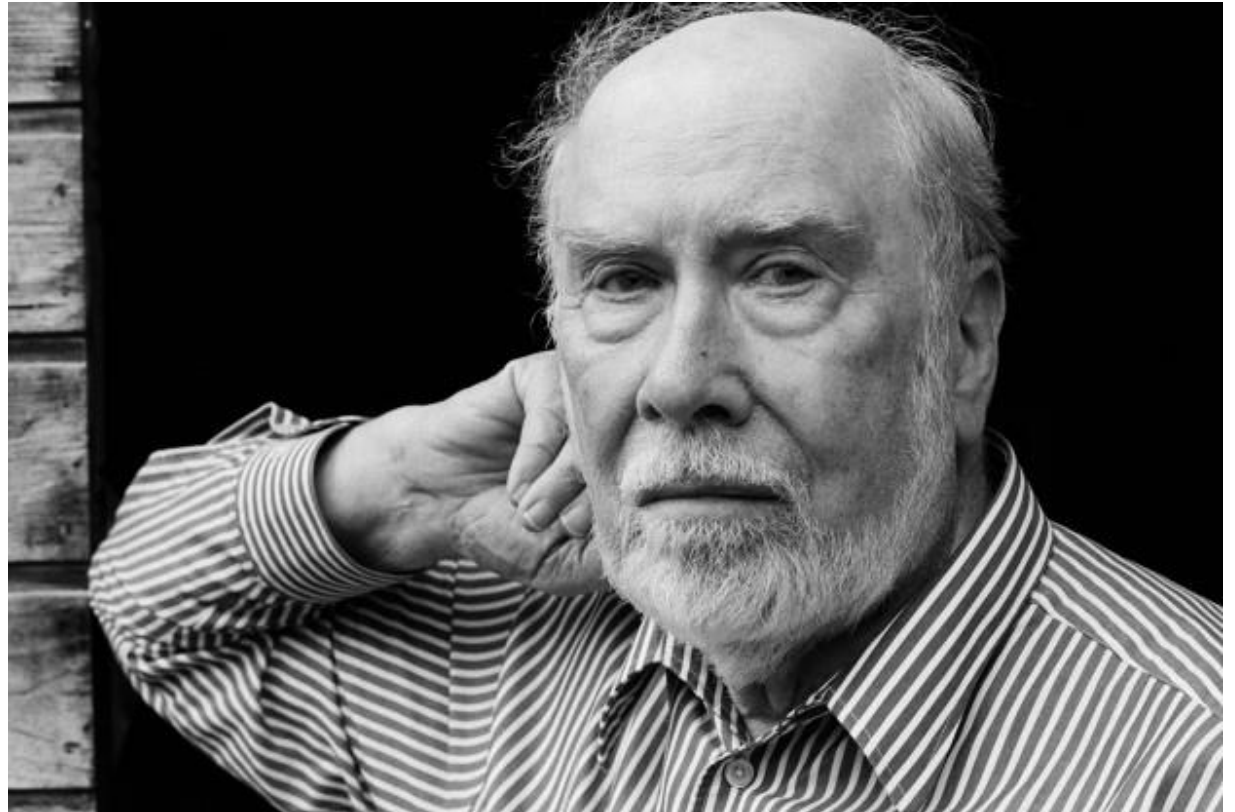
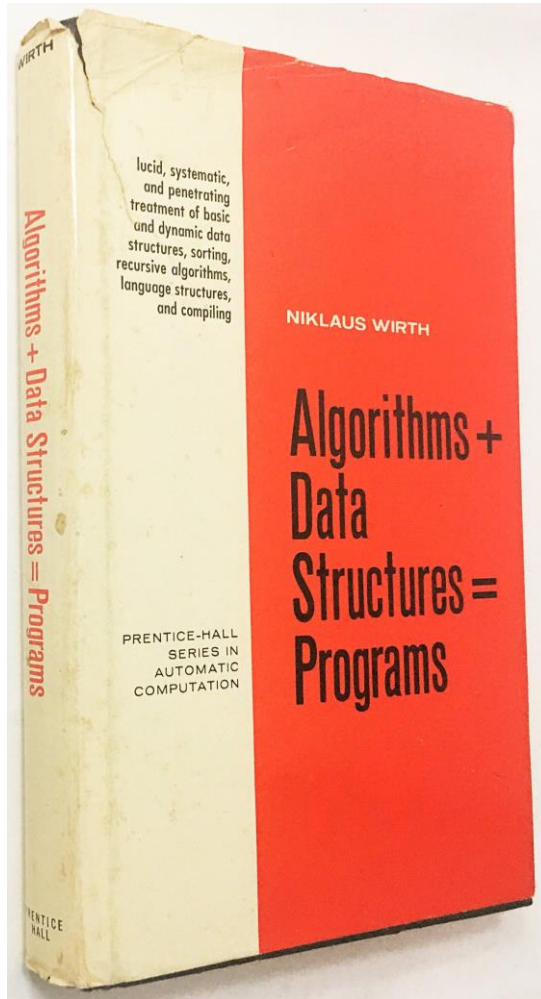
Il grafo delle operazioni (**DAG**= *Direct Acyclic Graph*) può essere più complicato



Come
definisco il
DAG?

Le strutture dati in Apache Spark

Le strutture dati condizionano il modo in cui creiamo i programmi



Le strutture dati in Apache Spark

API Strutturata (alto livello)

DataFrames & DataSet

API Non strutturata (basso livello)

RDD (*Resilient Distributed Datasets*)

Le strutture dati in Apache Spark

Un'analisi storica

- Le prime versioni di Spark sono fondate sugli *RDD* (almeno da *Spark0.6, 2012*)

`RDD` = « an immutable, partitioned collection of elements that can be operated on in parallel »

- In un secondo tempo vengono inseriti i *DataFrame* (*Spark1.3, 2015*)

`DataFrame` = « is equivalent to a relational table in Spark SQL »
(*experimental*)

- I *Dataset* sono un'estensione dei *DataFrame* (*Spark1.6, gennaio 2016*)

`Dataset` = « a strongly typed collection of objects that can be transformed in parallel using functional or relational operations » (*experimental*)

- A partire da *Spark2.0 (luglio 2016)* i *DataFrame* e i

Caratteristiche comuni di *RDD*, *DataFrame* e *Dataset*

- Sono collezioni di dati *distribuite* e *immutabili*.
- Supportano *trasformazioni* e *azioni* che vengono eseguite in parallelo
(le trasformazioni sono eseguite in maniera «pigra»).
- Sono i vertici del **DAG** (*Direct Acyclic Graph*) che rappresenta il calcolo da eseguire
(le trasformazioni sono gli archi)
DataFrame e *DataSet* sono implementati mediante *RDD*.
È possibile passare facilmente da una rappresentazione all'altra.

```
scala> val a = spark.range(1,1000000)
a: org.apache.spark.sql.Dataset[Long] = [id: bigint]

scala> val b = a.toDF()
b: org.apache.spark.sql.DataFrame = [id: bigint]

scala> b.rdd.partitions.length
res7: Int = 2
```

Differenze principali fra *RDD*, *DataFrame* e *Dataset*

Livello

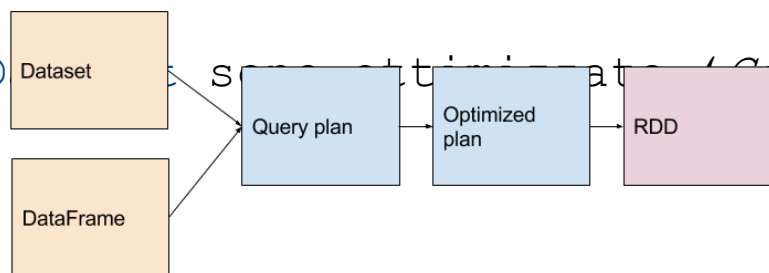
RDD sono considerate un'API a basso livello: permettono maggiore controllo, ma sono meno ottimizzabili da Spark. Inoltre sono un po' più complicate da usare.

Schema

DataFrame e *Dataset* hanno una definizione formale dei dati.
RDD no.

Prestazioni

DataFrame e *Dataset* sono ottimizzati (Catalyst e Tungsten)



Differenze principali fra *RDD*, *DataFrame* e *Dataset*

Sicurezza rispetto ai tipi

RDD e *Dataset* hanno un tipo ben definito in fase di compilazione.

Gli errori legati al tipo con i *DataFrame* possono essere individuati solo durante l'esecuzione.

Linguaggi

RDD e *DataFrame* sono disponibili per tutti i linguaggi

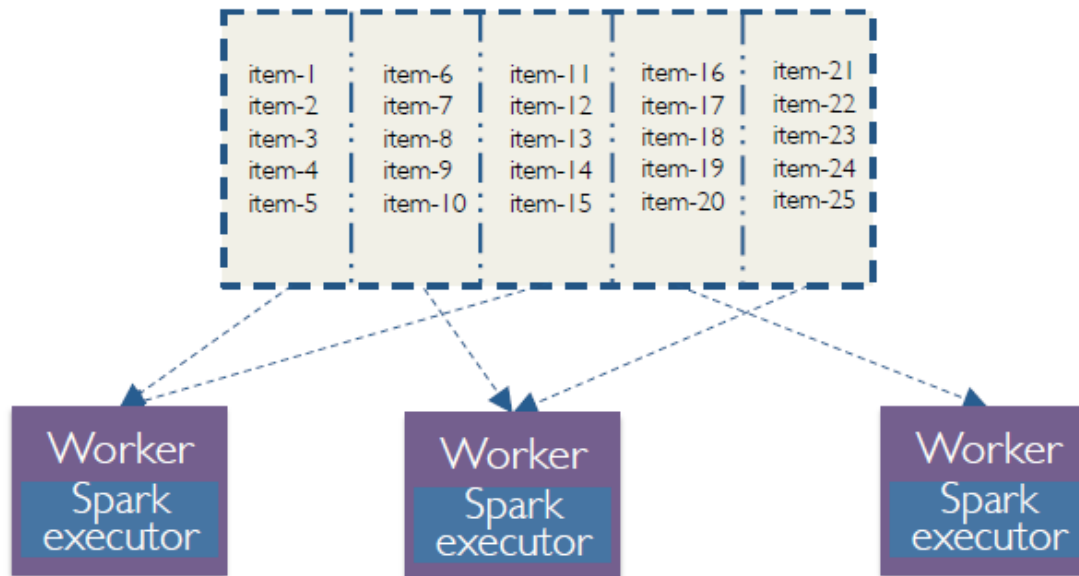
I *DataSet* solo in Java e Scala

Le partizioni

Gli RDD (e quindi i DataFrame e Dataset) sono dati distribuiti sui nodi del cluster

```
val rdd = sc.textFile("hdfs://localhost:9000/path/to/my/file")  
val errorCount = rdd.filter(_.startsWith("ERROR")).count()
```

RDD split into 5 partitions



Le partizioni

Gli RDD (e quindi i DataFrame e Dataset) sono dati distribuiti sui nodi del cluster

Leggo un file su HDFS, già diviso in partizioni

```
val rdd = sc.textFile("data/log.txt")
```

```
rdd: org.apache.spark.rdd.RDD[String] = data/log.txt MapPartitionsRDD[9] at textFile at <console>:67  
Took: 2.912s, at 2019-03-03 13:59
```

```
println(rdd.getNumPartitions)
```

```
2
```

```
Took: 1.464s, at 2019-03-03 14:02
```

Creo un RDD a partire da una sequenza numerica. Decido esplicitamente

```
val numeri = sc.parallelize(1 to 1000000, 7)
```

```
numeri: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[18] at parallelize at <console>:67
```

```
Took: 1.374s, at 2019-03-03 14:06
```

```
println(numeri.getNumPartitions)
```

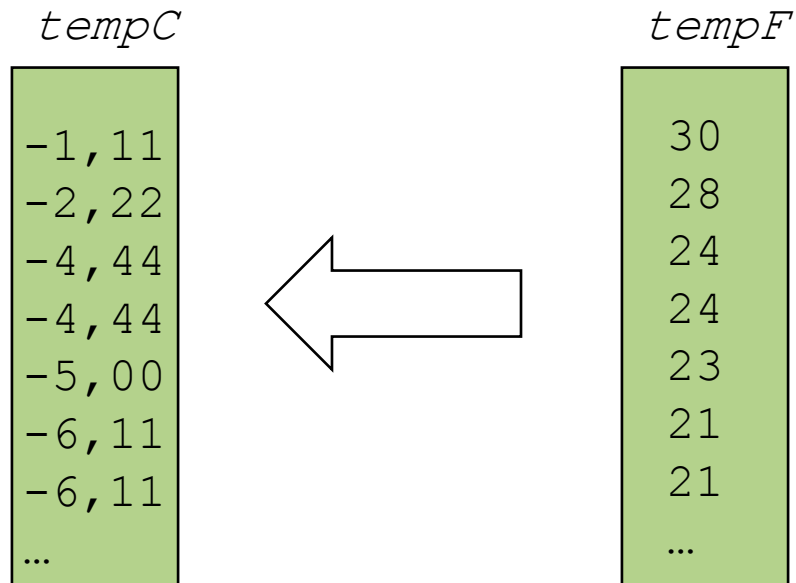
```
7
```

```
Took: 1.281s, at 2019-03-03 14:06
```

Immutabilità

Non posso modificare il contenuto. Devo creare una nuova entità modificata

```
val tempC = tempF.map(t => (t-32)*5/9)
```



Riassumendo

Le strutture dati in Spark:

- *DataFrame*, *Dataset* e *RDD*
- Immutabili e distribuite sui nodi del cluster
- Vengono manipolate con *trasformazioni* e *azioni*

Le Trasformazioni sono «pigre»

I calcoli effettivi vengono determinati dalle *Azioni*

```
val rdd = sc.textFile("data/log.txt")
rdd: org.apache.spark.rdd.RDD[String] = data/log.txt MapPartitionsRDD[34] at textFile at <console>:67
Took: 0.899s, at 2019-03-03 14:59

val rdd2 = rdd.filter(s=>s.contains("ERROR"))
rdd2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[35] at filter at <console>:69
Took: 0.827s, at 2019-03-03 14:59

val rdd3 = rdd2.filter(s=>s.contains("worker process"))
rdd3: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[36] at filter at <console>:71
Took: 0.939s, at 2019-03-03 14:59

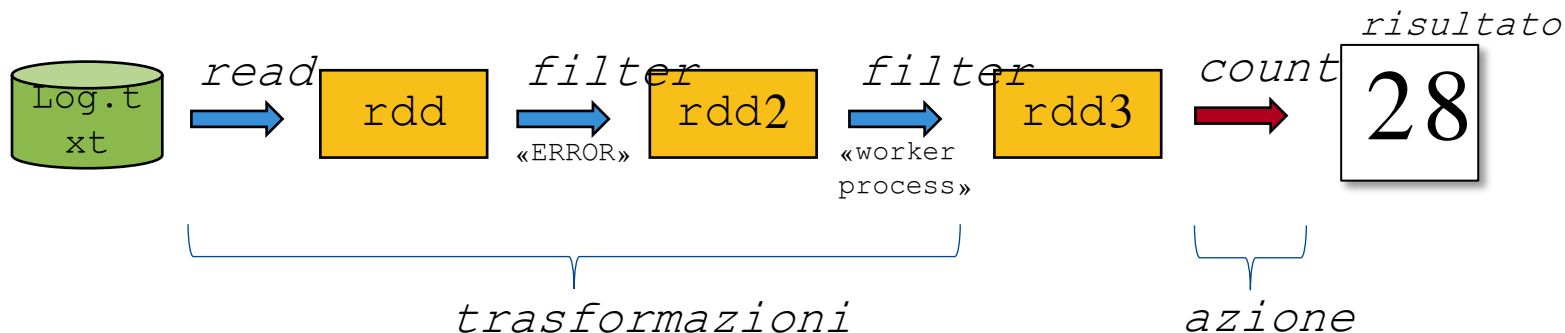
rdd3.count()
res55: Long = 28
28
Took: 1.275s, at 2019-03-03 14:59
```

Non succede nulla

Non succede nulla

Non succede nulla

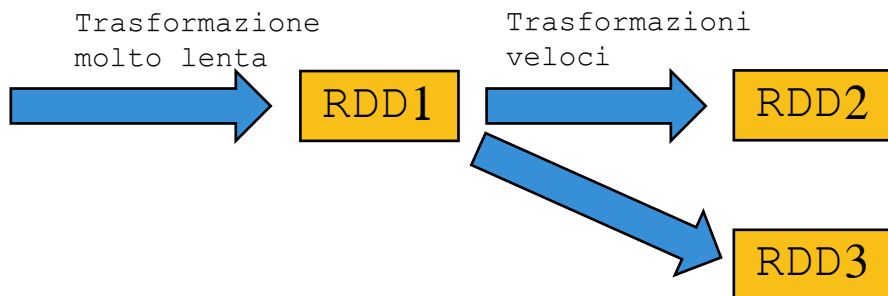
Parte il calcolo



Cache

Un'ottimizzazione importante

L'esecuzione di un'**azione** determina il calcolo di tutte le **trasformazioni** da cui l'azione dipende.



È possibile salvare il risultato in modo da evitare il ricalcolo.

I comandi relativi sono *cache()* e *persist()*

I dati possono essere salvati in memoria, su disco o nella memoria *off-heap*

Cache

Un'ottimizzazione importante

```
val df = spark.read
  .option("header", true)
  .option("inferSchema", true)
  .option("mode", "FAILFAST")
  .csv("data/earthquake.csv")
  .cache()
```

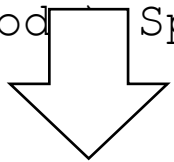
df: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [id: decimal(3,-11), date: string ... 15 more fields]

Took: 1.069s, at 2019-03-03 18:40

Lineage

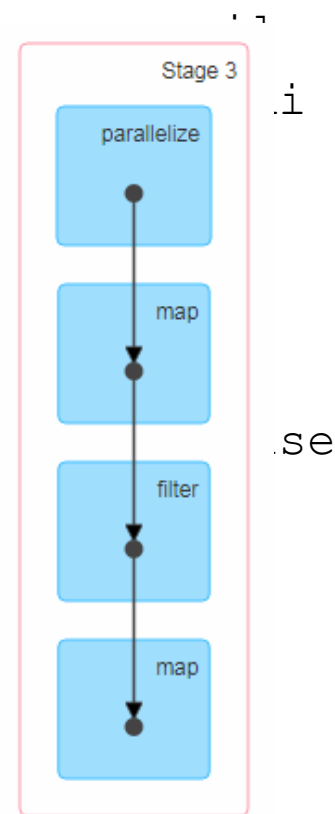
RDD (e quindi anche DataFrame e Dataset) sono «resilienti», cioè tolleranti agli errori.

Se una partizione di un RDD si perde (ad fallimento di un nodo Spark deve essere ricostruirla.



Ogni RDD sa come ricostruire il suo conte agli RDD da cui dipende.

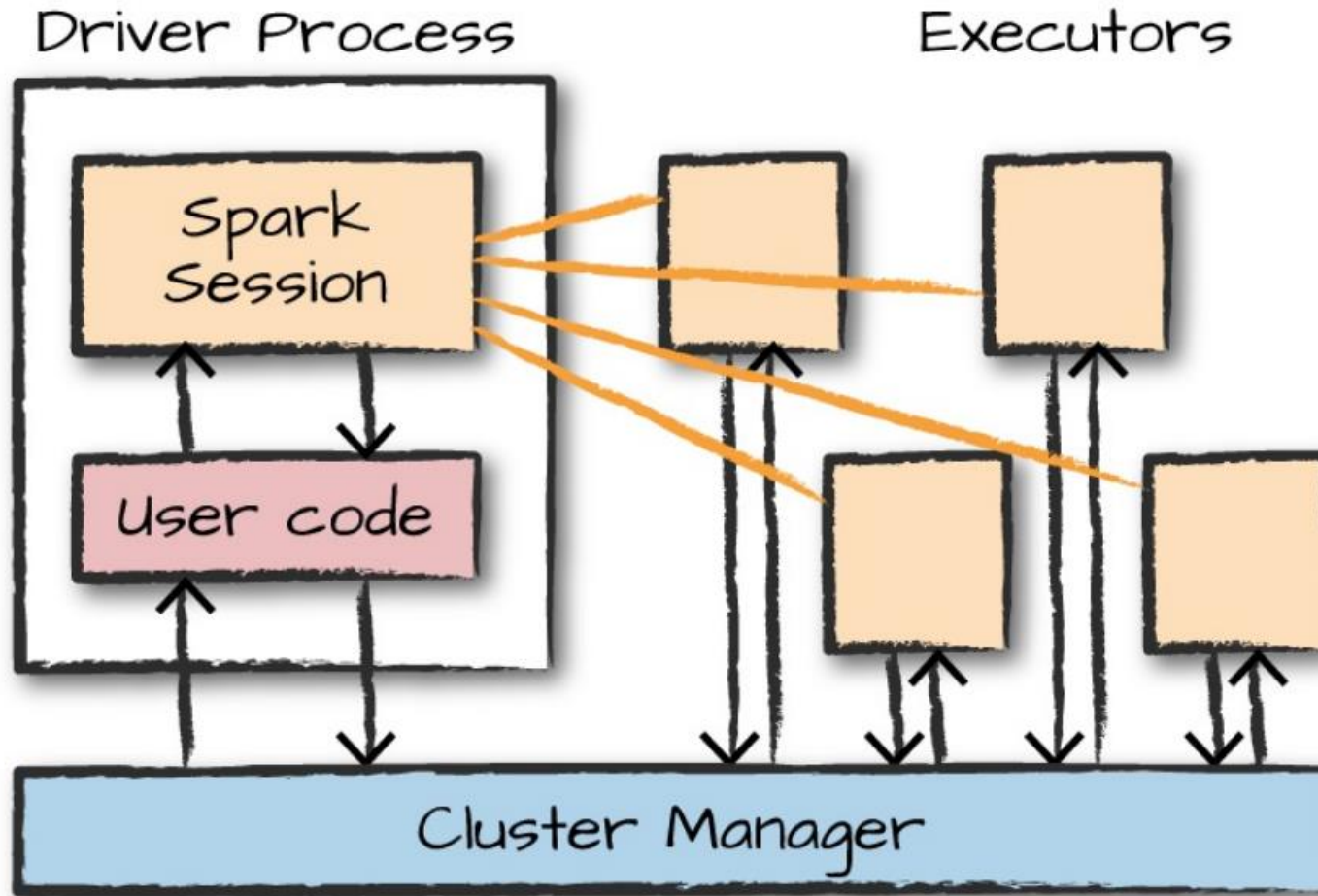
Questa informazione si chiama «lineage»



L' applicazione e Spark

Struttura di un'applicazione Spark

Un'applicazione Spark prevede un processo *Driver* ed un certo numero di *Esecutori*.



La Sessione

SparkSession è l'oggetto che permette di dare comandi all'applicazione Spark.

Quando si usa Spark in modalità interattiva (con il comando *spark-shell* o con un *Notebook*).

⇒ Viene automaticamente creato un oggetto (costante)
SparkSession di nome *spark*.

Quando si lancia un'applicazione precompilata con *spark-submit*

⇒ L'oggetto *SparkSession* deve essere creato esplicitamente

La Sessione

Il codice più antico (cioè di pochi anni fa!!) usa un altro oggetto: lo *SparkContext*.

Questo oggetto viene ancora utilizzato dall'API relativa agli RDD (ad es. per creare un RDD). Nelle sessioni interattive (spark-shell) la costante *sc* contiene lo *SparkContext*

```
val rdd = sc.parallelize(1 to 1000, 5)
```

Lo spark context è sempre contenuto nella *SparkSession*:

```
val rdd = spark.sparkContext.parallelize(1 to 1000, 5)
```

La Sessione

Per creare (o recuperare) l'istanza della *SparkSession* nei programmi si può fare:

```
val spark = SparkSession.builder().getOrCreate()
```

Ed è possibile passare dei parametri di configurazione:

```
val spark = SparkSession
  .builder()
  .appName("SparkSessionZipsExample")
  .config("spark.sql.warehouse.dir", warehouseLocation)
  .enableHiveSupport()
  .getOrCreate()
```

Qualche comando per il laboratorio

Qualche «ricetta» Spark

Per il laboratorio di oggi pomeriggio

Sommare i cubi del primo milione di interi

```
val numbers = spark.range(1,1000001)
numbers.map(x=>BigDecimal(x)).map(x=>x*x*x).reduce(_+_)
```

Leggere un file CSV

```
val earthquakes = spark.read
    .option("header",true)
    .option("inferSchema",true)
    .option("mode","failfast")
    .csv("data/earthquake.csv")
earthquakes.printSchema()
```

Qualche «ricetta» Spark

Per il laboratorio di oggi

```
val earthquake = spark.read
  .option("header", true)
  .option("inferSchema", true)
  .option("mode", "FAILFAST")
  .csv("data/earthquake.csv")
earthquake.printSchema()
```

```
root
|-- id: decimal(3,-11) (nullable = true)
|-- date: string (nullable = true)
|-- time: string (nullable = true)
|-- lat: double (nullable = true)
|-- long: double (nullable = true)
|-- country: string (nullable = true)
|-- city: string (nullable = true)
|-- area: string (nullable = true)
|-- direction: string (nullable = true)
|-- dist: double (nullable = true)
|-- depth: double (nullable = true)
|-- xm: double (nullable = true)
|-- md: double (nullable = true)
|-- richter: double (nullable = true)
|-- mw: double (nullable = true)
|-- ms: double (nullable = true)
|-- mb: double (nullable = true)
```

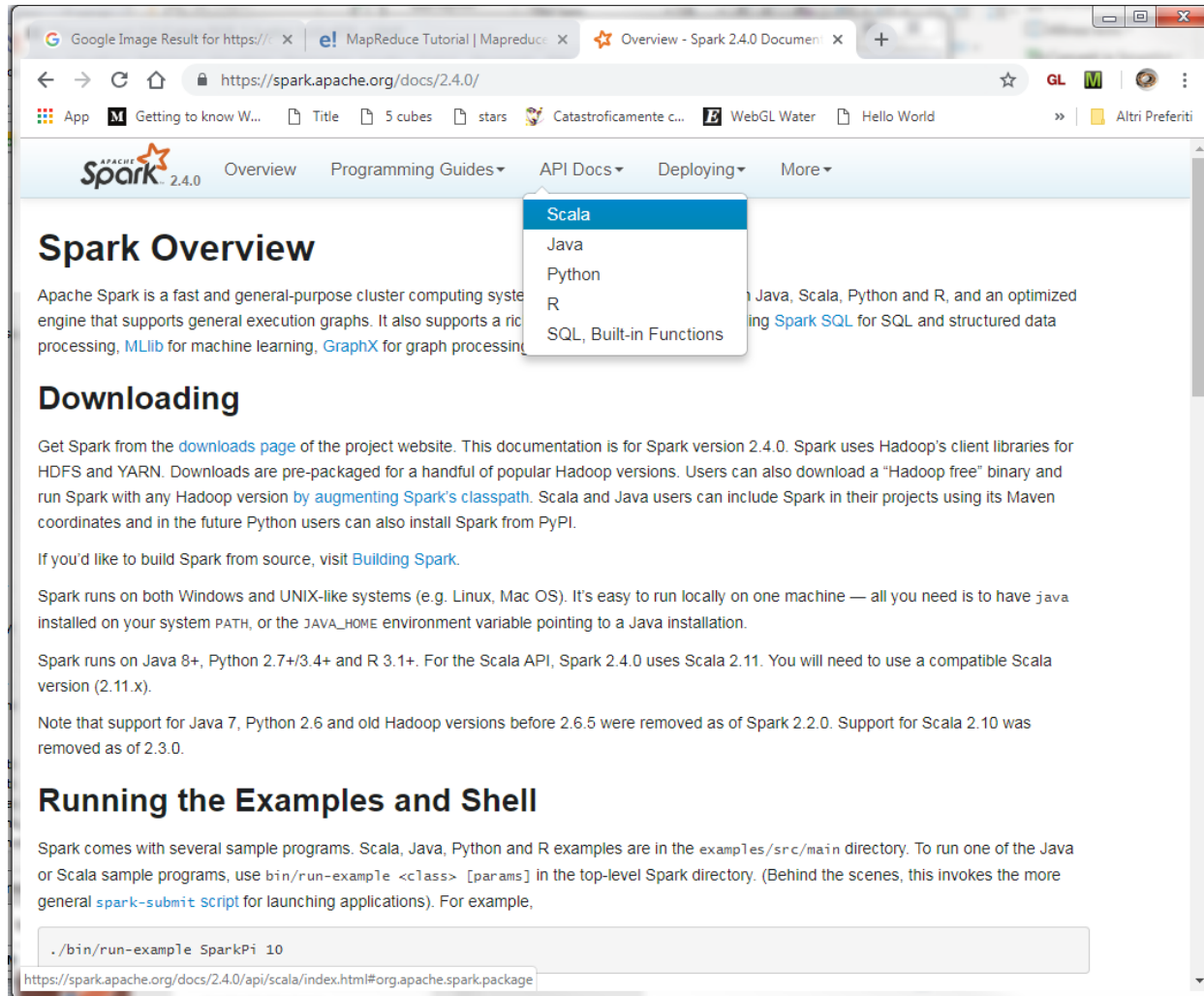
```
earthquake: org.apache.spark.sql.DataFrame = [id: decimal(3,-11), date: string ... 15 more fields]
```

Took: 1.732s, at 2019-02-28 20:02

La documentazione e

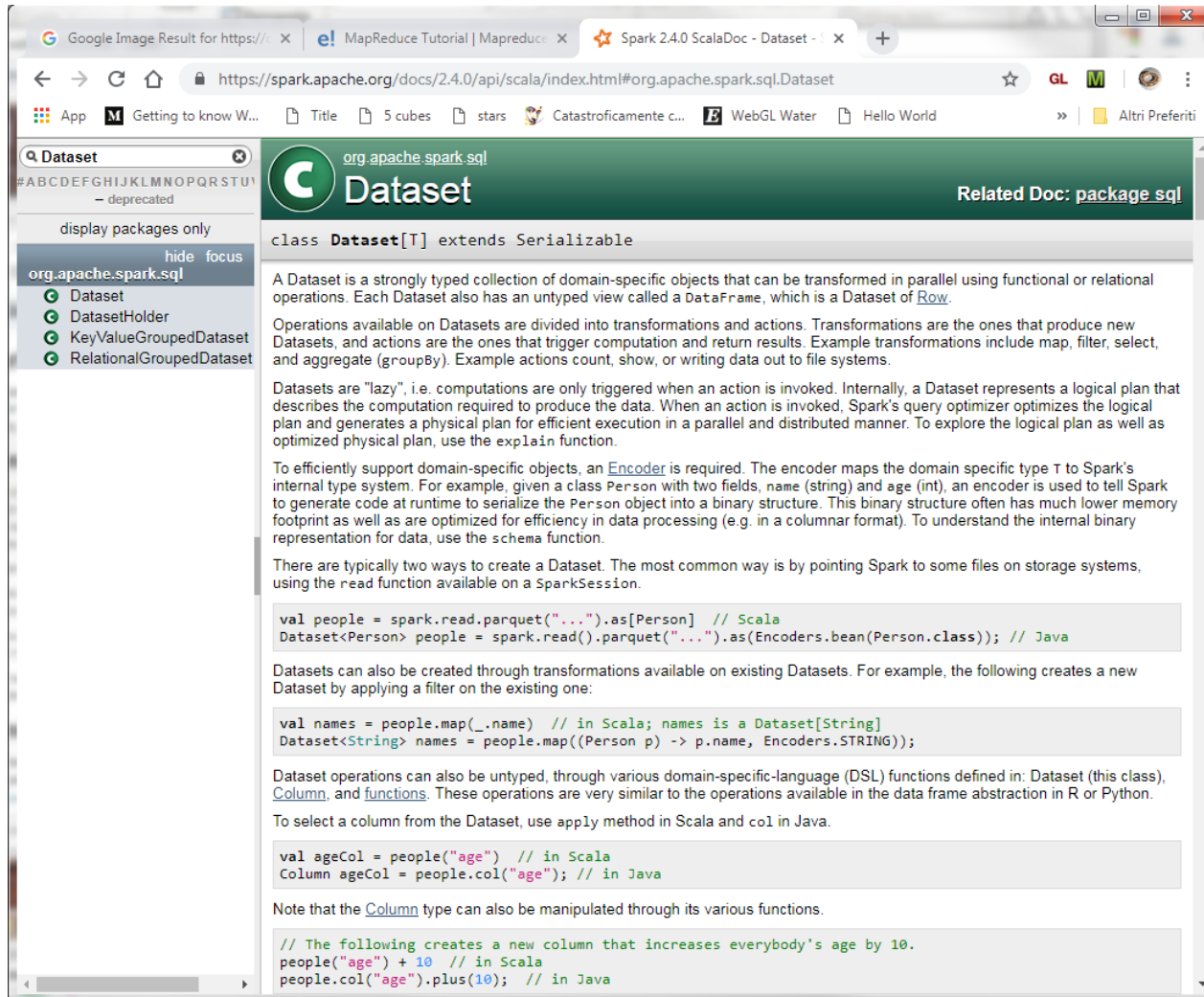
La documentazione

<https://spark.apache.org/docs/2.4.0/>



La documentazione

<https://spark.apache.org/docs/2.4.0/>



The screenshot shows a web browser displaying the Spark 2.4.0 ScalaDoc - Dataset page. The browser's address bar shows the URL <https://spark.apache.org/docs/2.4.0/api/scala/index.html#org.apache.spark.sql.Dataset>. The page features a green header with the Spark logo and the text "org.apache.spark.sql Dataset". A sidebar on the left lists the contents of the package, including "Dataset", "DatasetHolder", "KeyValueGroupedDataset", and "RelationalGroupedDataset". The main content area is titled "class Dataset[T] extends Serializable" and contains several paragraphs of text explaining the Dataset concept, its operations, and how to create and manipulate it. The text is accompanied by code snippets in Scala and Java. The code snippets show how to read data from a file, create a new Dataset from an existing one, and manipulate columns. The code is as follows:

```
val people = spark.read.parquet("...").as[Person] // Scala
Dataset<Person> people = spark.read().parquet("...").as(Encoders.bean(Person.class)); // Java

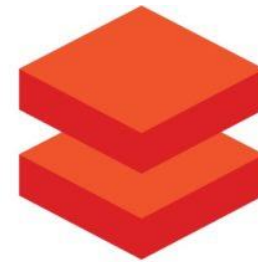
val names = people.map(_.name) // in Scala; names is a Dataset[String]
Dataset<String> names = people.map((Person p) -> p.name, Encoders.STRING));

val ageCol = people("age") // in Scala
Column ageCol = people.col("age"); // in Java

// The following creates a new column that increases everybody's age by 10.
people("age") + 10 // in Scala
people.col("age").plus(10); // in Java
```

Un po' di coordinate

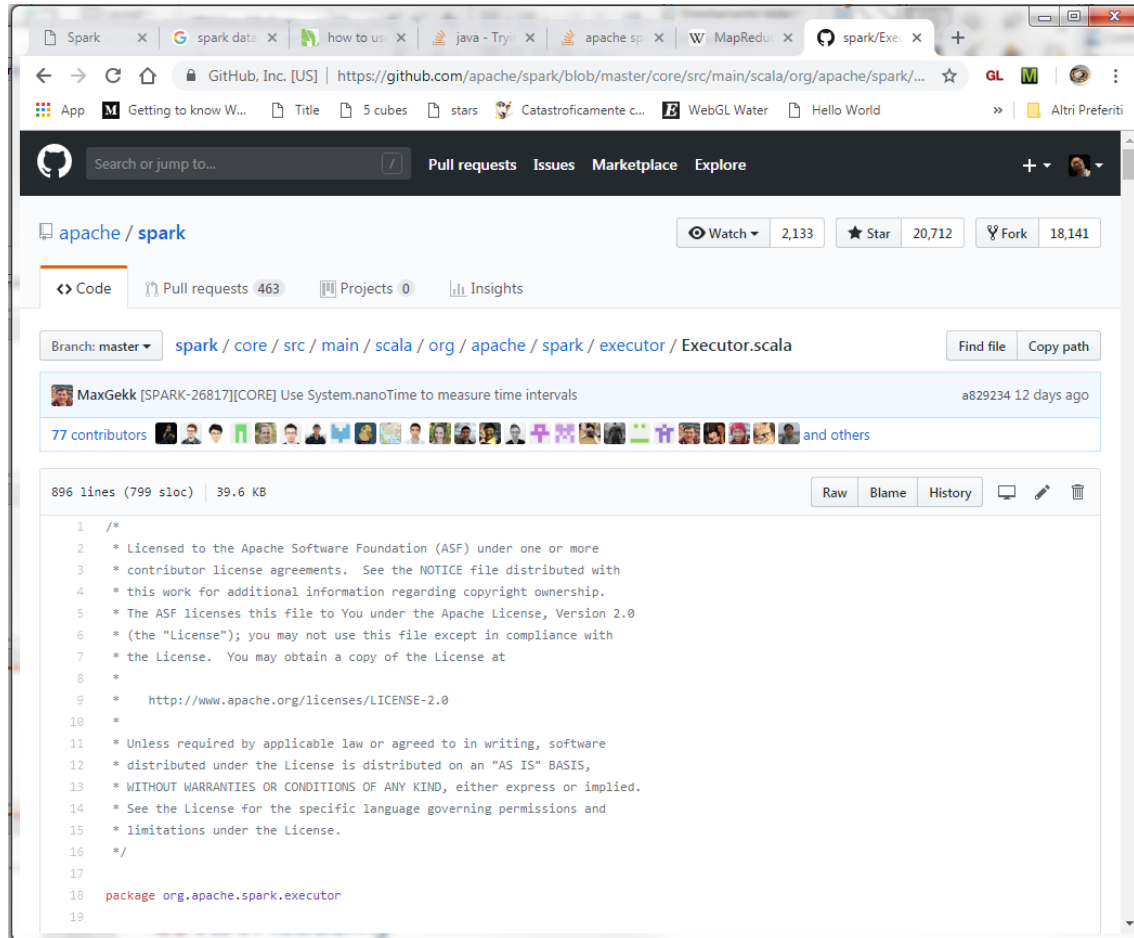
cloudera



databricks®

Apache Spark è OpenSource

<https://github.com/apache/spark/blob/master/core>



The screenshot shows the GitHub web interface for the Apache Spark repository. The browser's address bar displays the URL `https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/executor/Executor.scala`. The repository page for 'apache / spark' is visible, showing 2,133 watchers, 20,712 stars, and 18,141 forks. The 'Code' tab is selected, and the file path 'spark / core / src / main / scala / org / apache / spark / executor / Executor.scala' is shown. The commit history for this file is displayed, with the latest commit by MaxGekk [SPARK-26817][CORE] titled 'Use System.nanoTime to measure time intervals' from 12 days ago. Below the commit information, 77 contributors are listed. The file content is shown as 896 lines (799 sloc) and 39.6 KB. The code content is a Scala file with a multi-line comment block at the top, followed by a package declaration `package org.apache.spark.executor`.

```
1  /*
2   * Licensed to the Apache Software Foundation (ASF) under one or more
3   * contributor license agreements. See the NOTICE file distributed with
4   * this work for additional information regarding copyright ownership.
5   * The ASF licenses this file to You under the Apache License, Version 2.0
6   * (the "License"); you may not use this file except in compliance with
7   * the License. You may obtain a copy of the License at
8   *
9   * http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
18  package org.apache.spark.executor
19
```

