Primo giorno: pomeriggio

Agenda 2/8

Le fondamenta

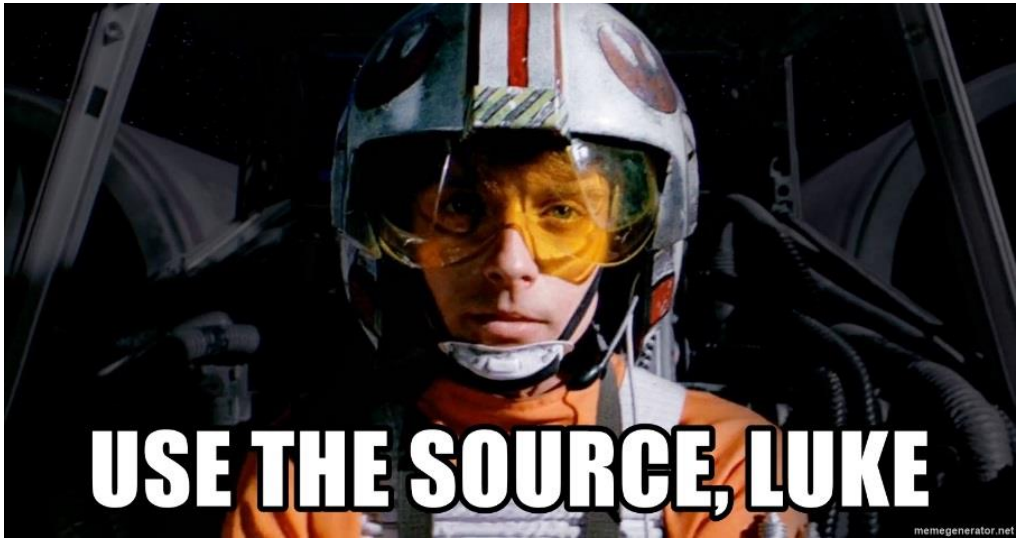
La programmazione funzionale
Un paradigma utile per il calcolo parallelo

Strutture dati
RDDs, DataSets e DataFrames

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

Punti di riferimento
Documentazione; Cloudera, Hortonworks; Databricks; sorgente su github

Esercizi di laboratorio
Mappe, filtri e qualche azione

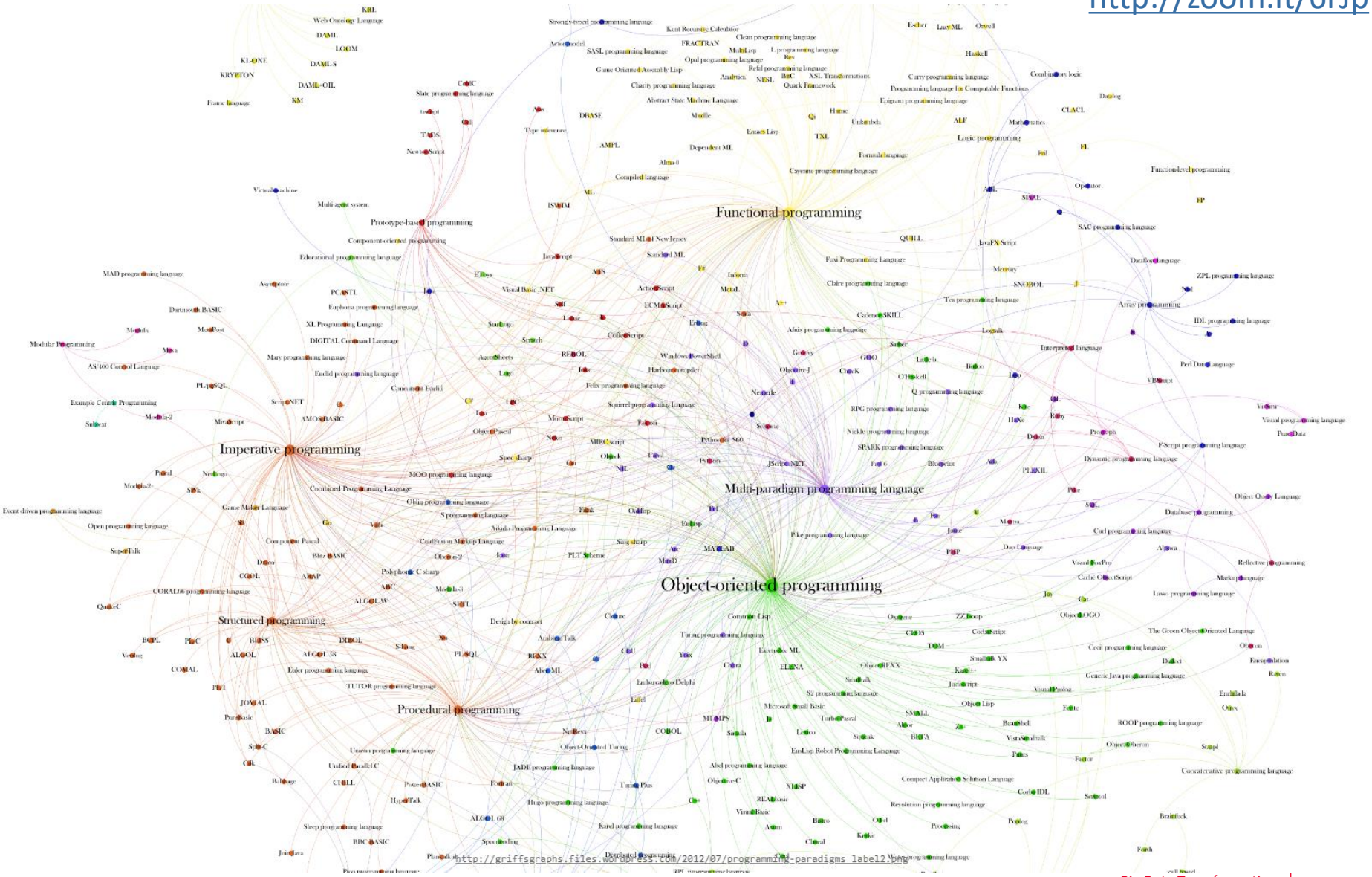


Cos'è la Programmazione Funzionale?

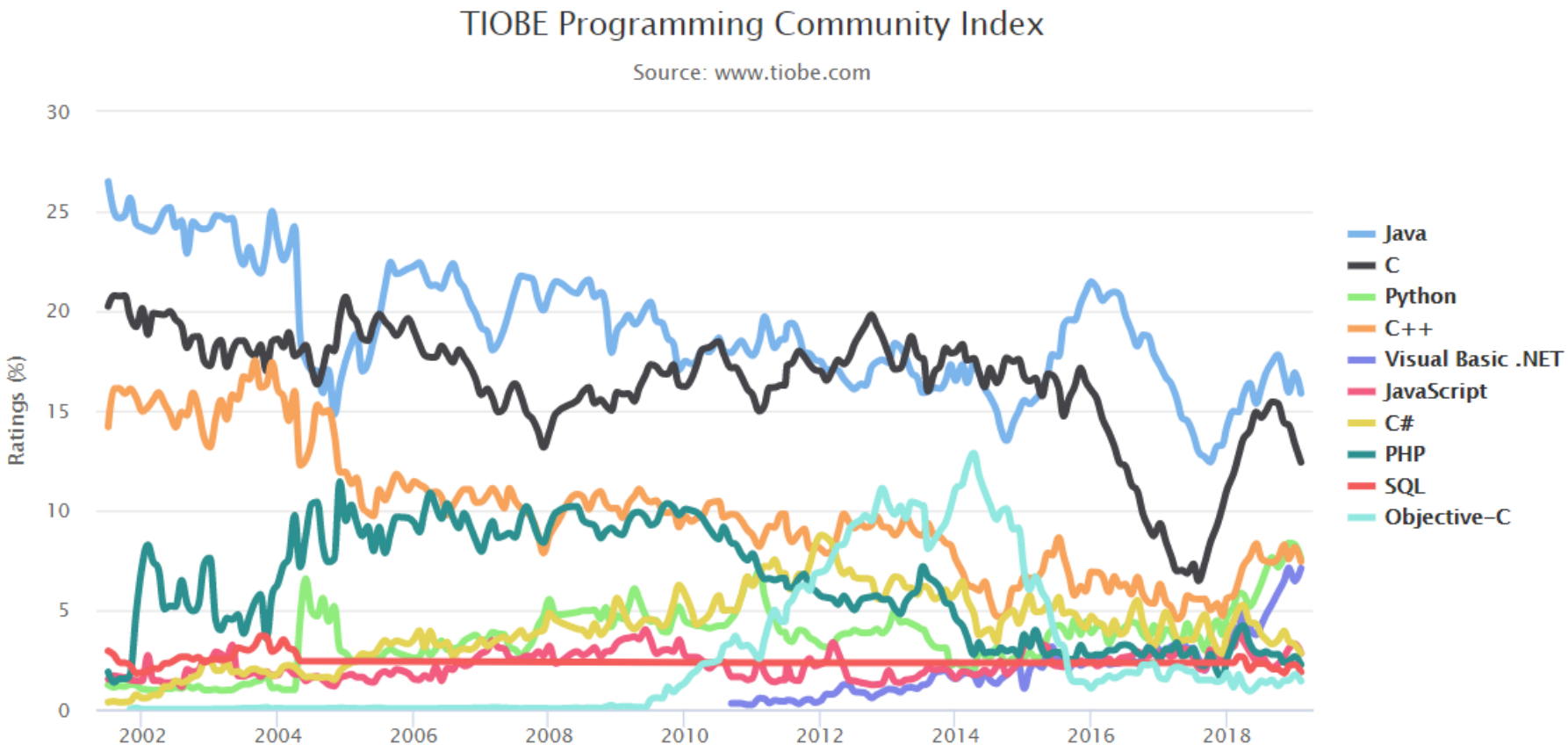
Esistono più di 8000 linguaggi di programmazione

The Graph of Programming Paradigms

http://zoom.it/6rJp



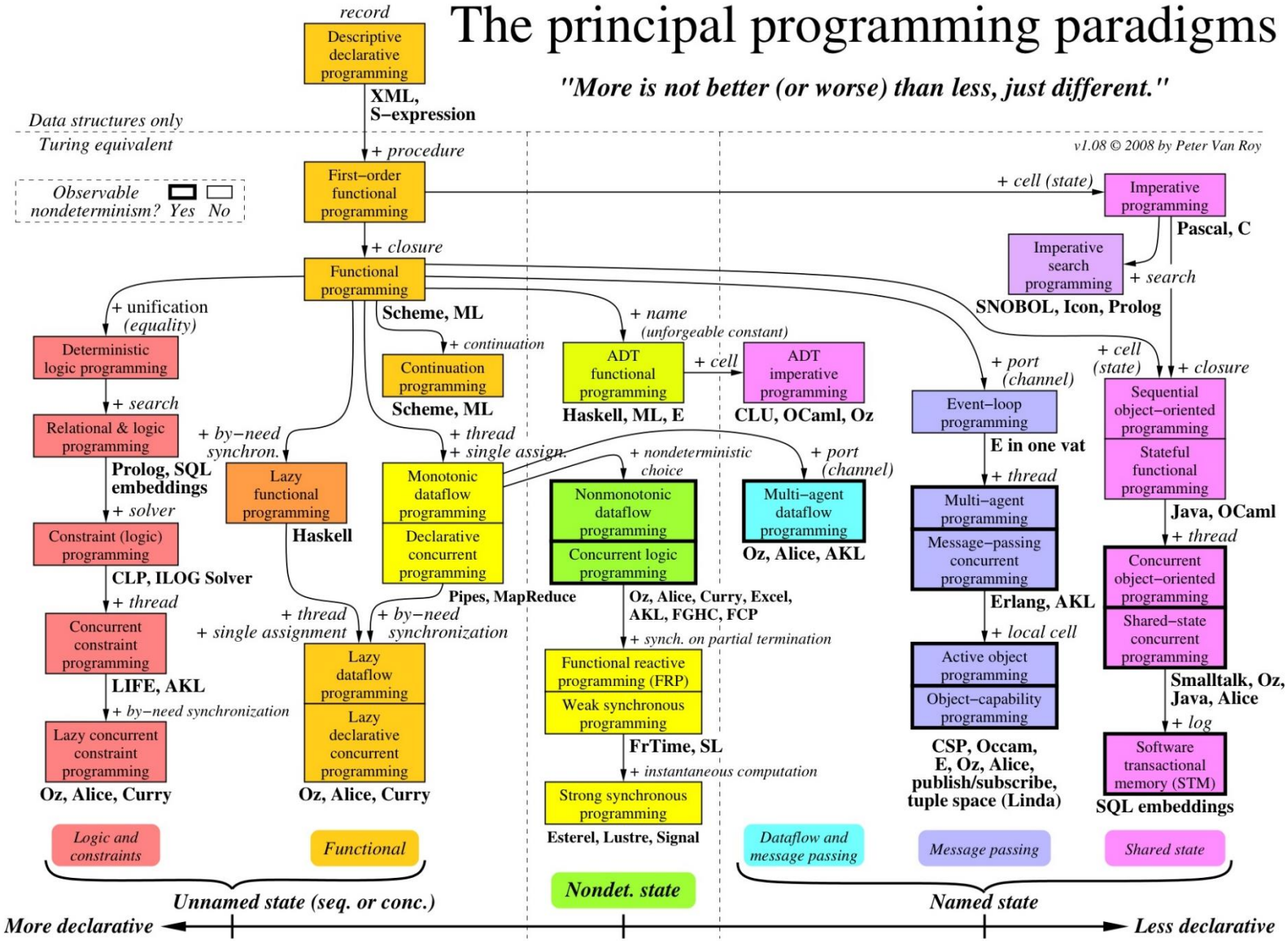
https://www.tiobe.com/tiobe-index/



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

— Alan Perlis

The principal programming paradigms



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

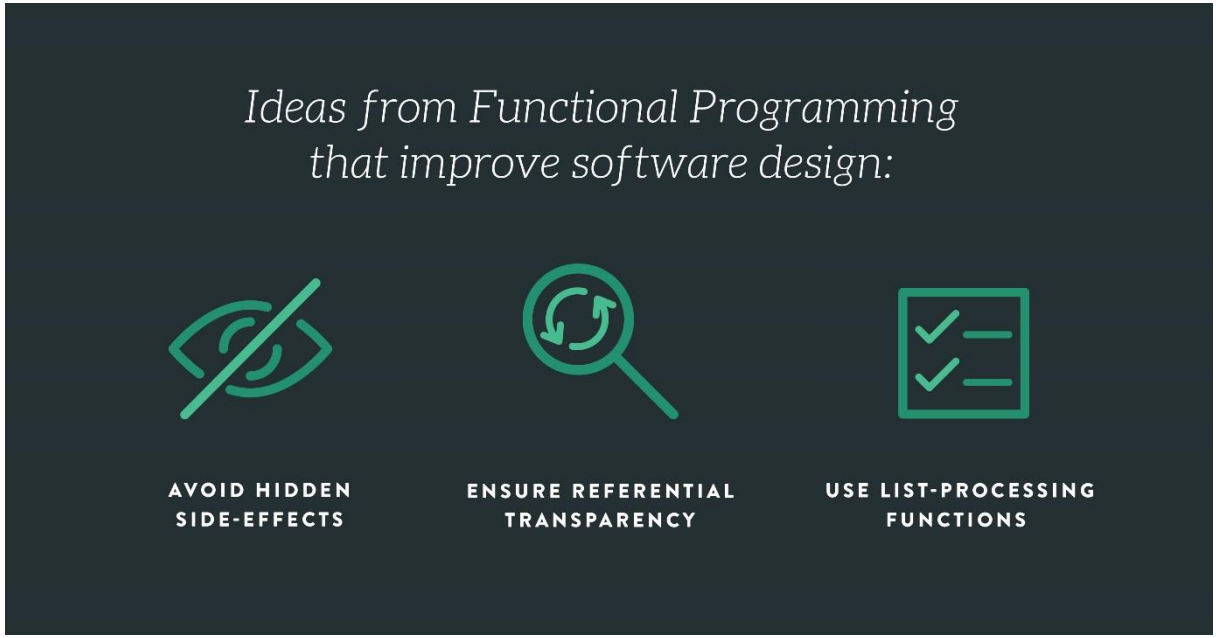
```
val arr = Seq(1, 2, 3, 5, 8, 13, 21)
var sum = 0
for ( x <- arr ) sum += x
```

Programmazione funzionale

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

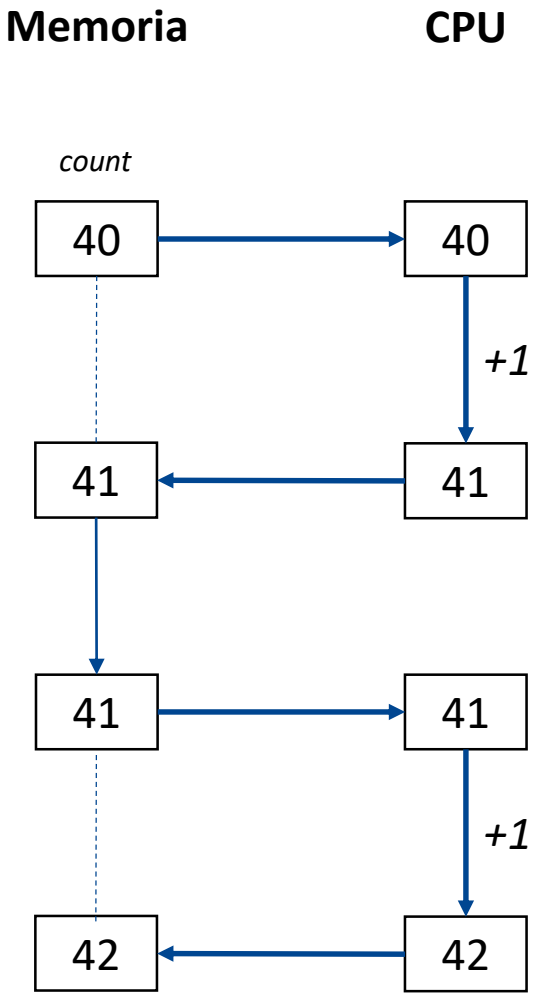
Caratteristiche del paradigma funzionale

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



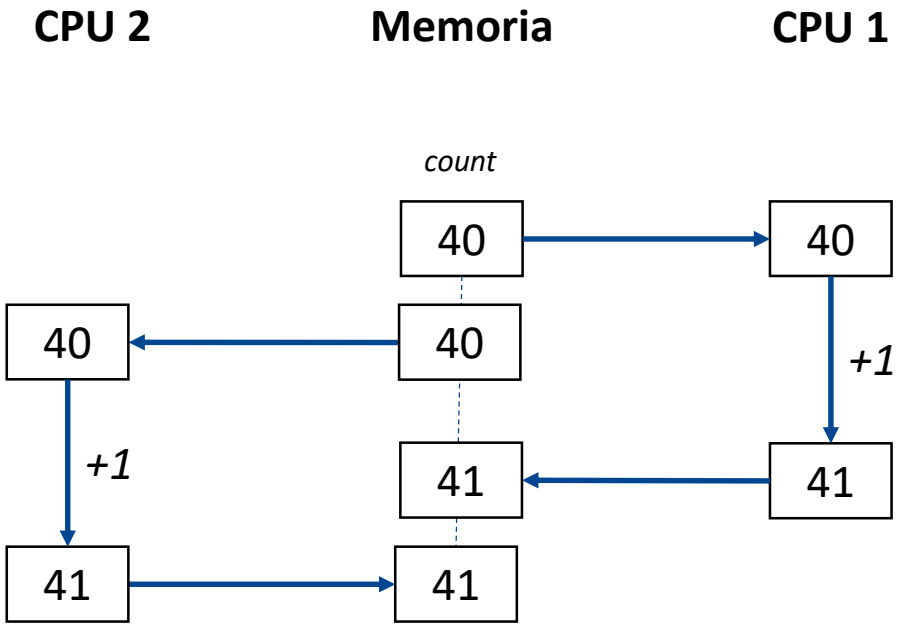
Gli effetti collaterali

```
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 valori diversi

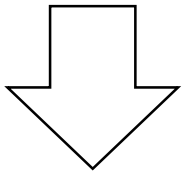
```
var i = 0

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

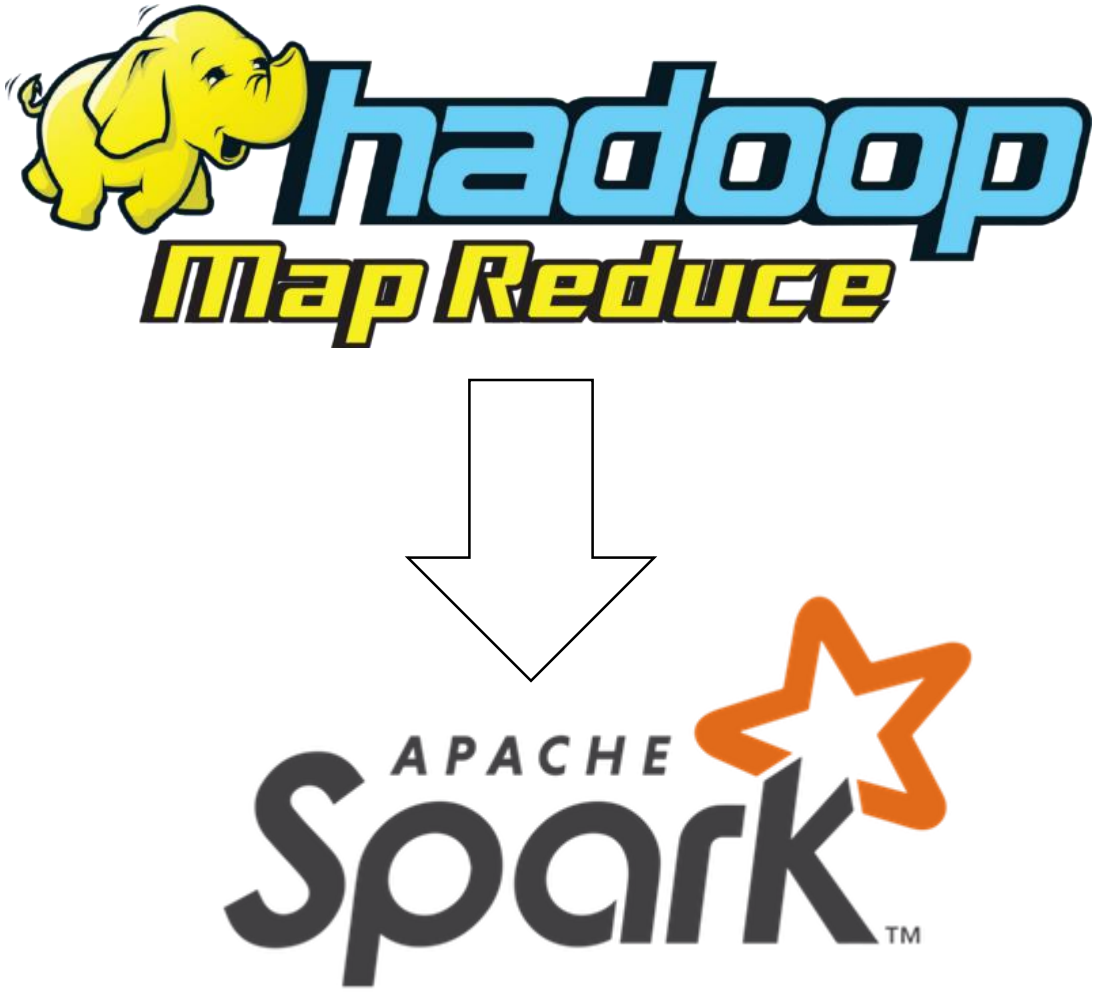


E' più facile:

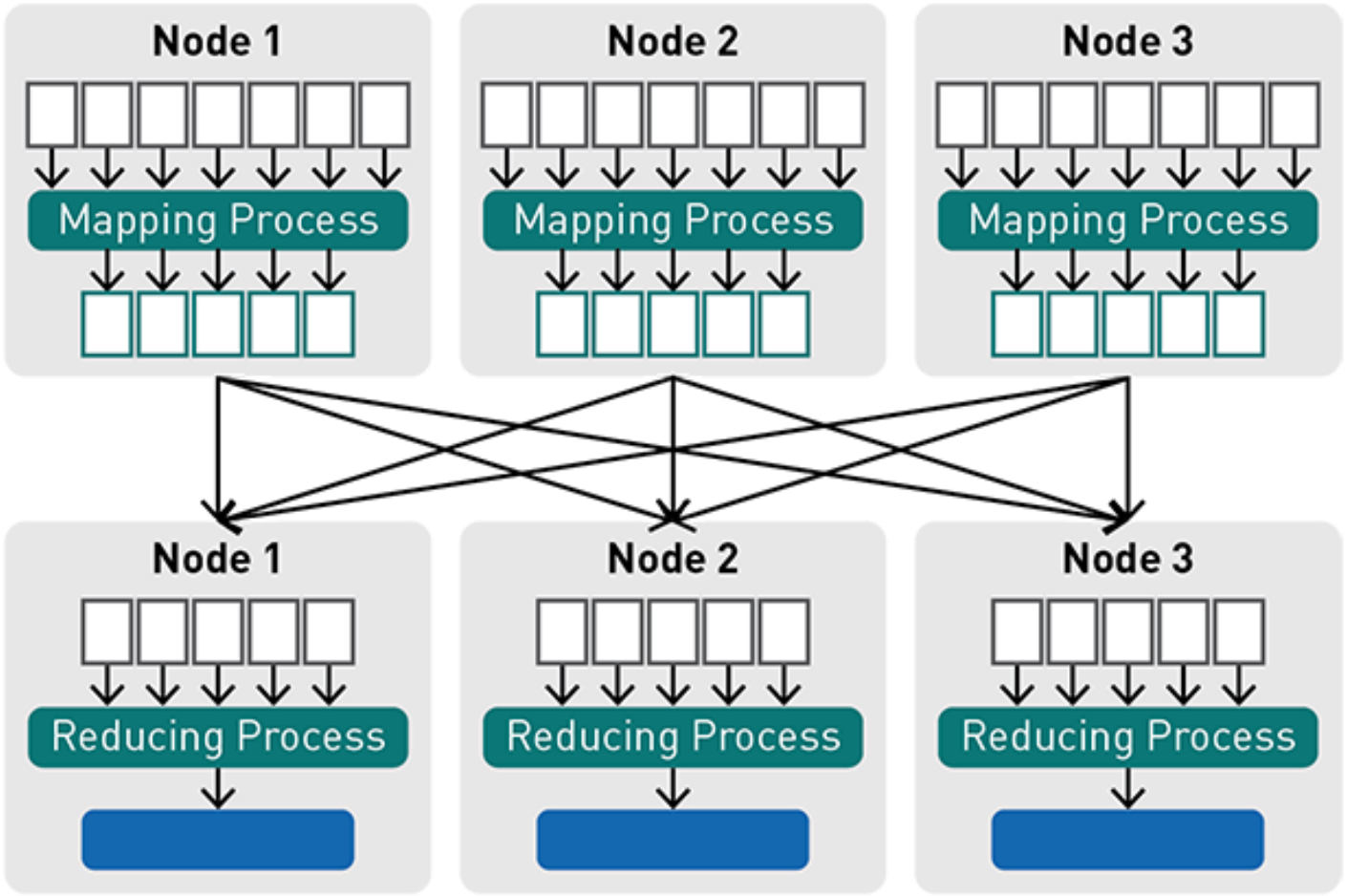
- Distribuire i dati fra i nodi
- Ricalcolare un dato intermedio in caso di fallimento di un nodo
- Ottimizzare il flusso dei calcoli

Torniamo a Spark!

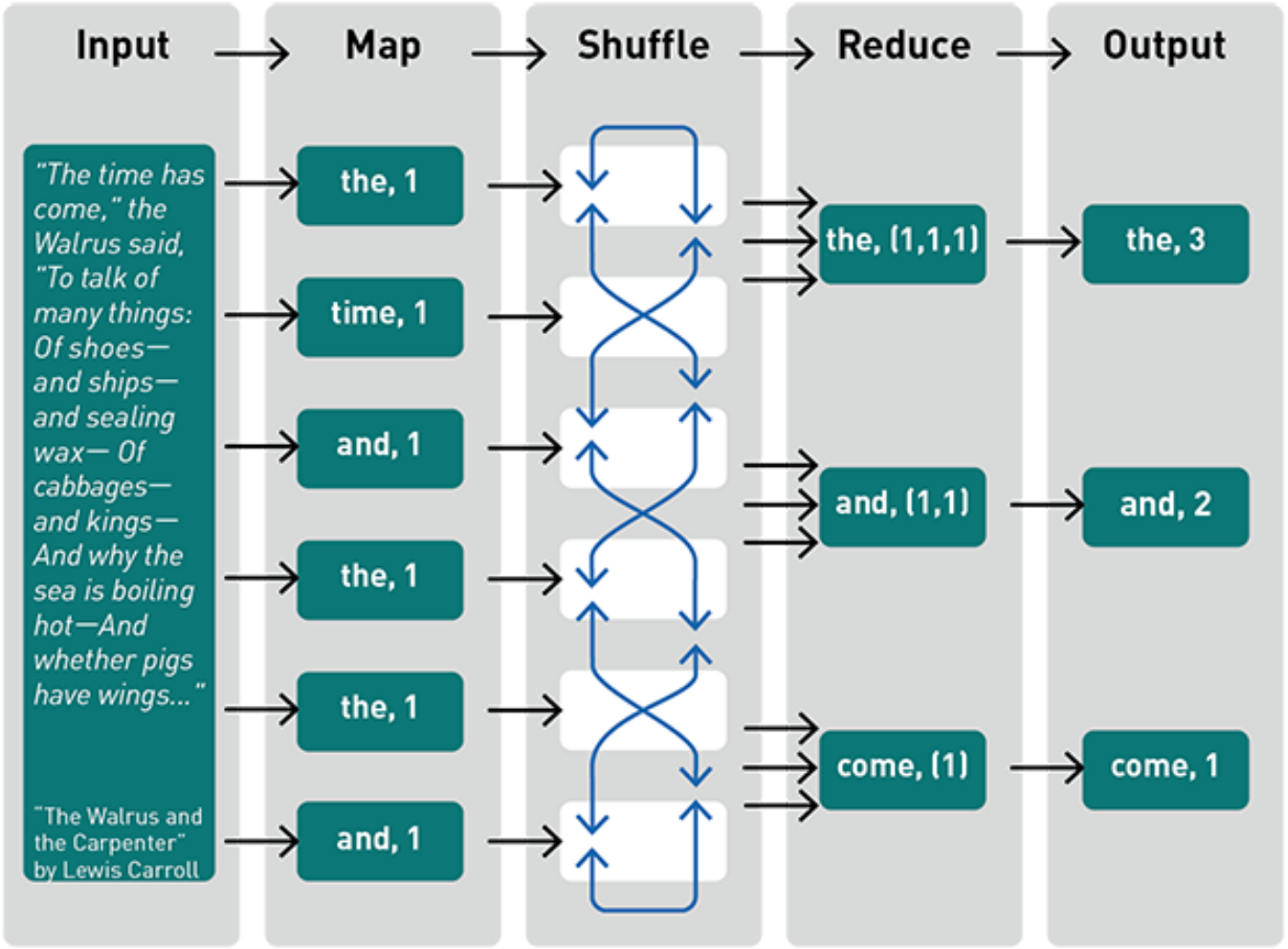
Spark è una diretta evoluzione di Hadoop Map Reduce



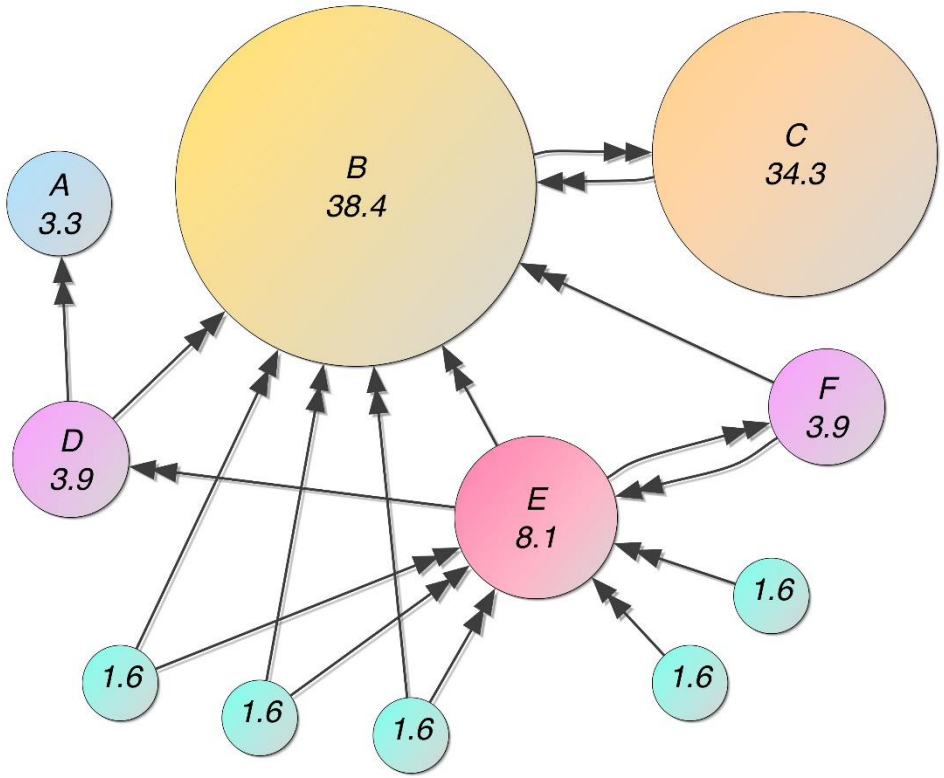
MapReduce


Node 1
Node 2
Node 3
Mapping Process
Mapping Process
Mapping Process
Node 1
Node 2
Node 3
Reducing Process
Reducing Process
Reducing Process

MapReduce


Input
Map
Shuffle
Reduce
Output
"The time has come," the Walrus said, "To talk of many things: Of shoes— and ships— and sealing wax— Of cabbages— and kings— And why the sea is boiling hot—And whether pigs have wings..."
"The Walrus and the Carpenter" by Lewis Carroll
the, 1
time, 1
and, 1
the, 1
the, 1
and, 1
the, (1,1,1)
and, (1,1)
come, (1)
the, 3
and, 2
come, 1

Un esempio di calcolo più complesso: PageRank



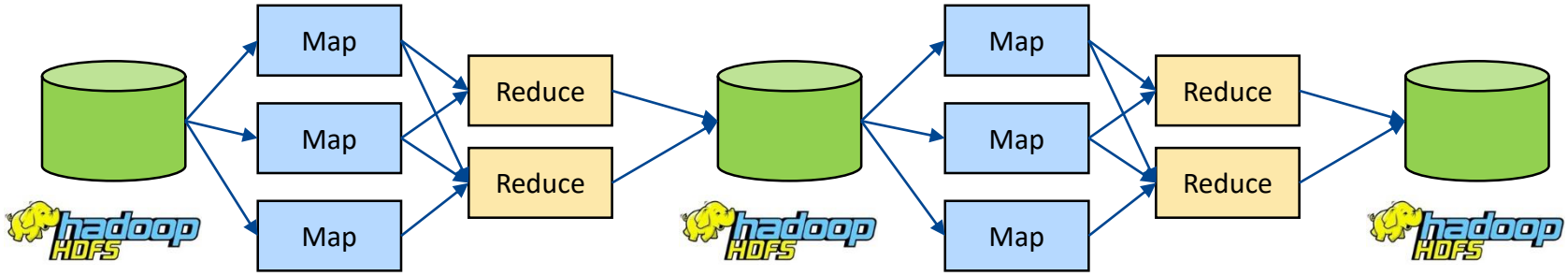
$$PageRank\ of\ site = \sum \frac{PageRank\ of\ inbound\ link}{Number\ of\ links\ on\ that\ page}$$

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

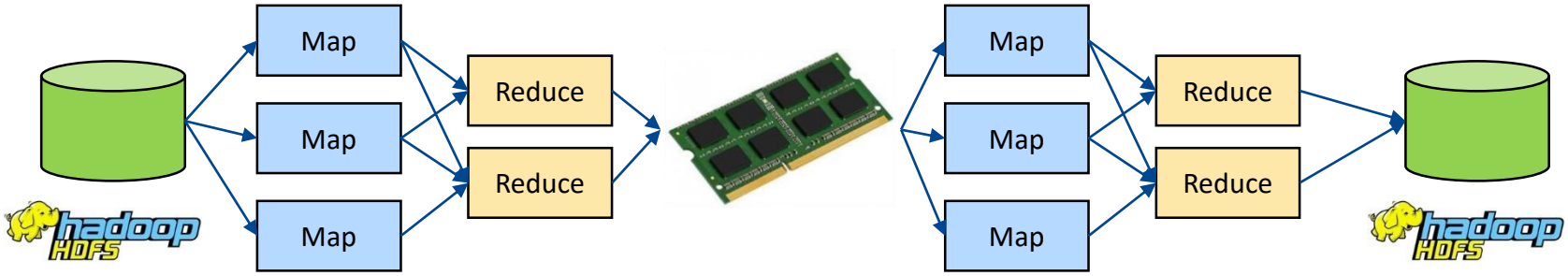
Spark è una diretta evoluzione di Hadoop Map Reduce

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

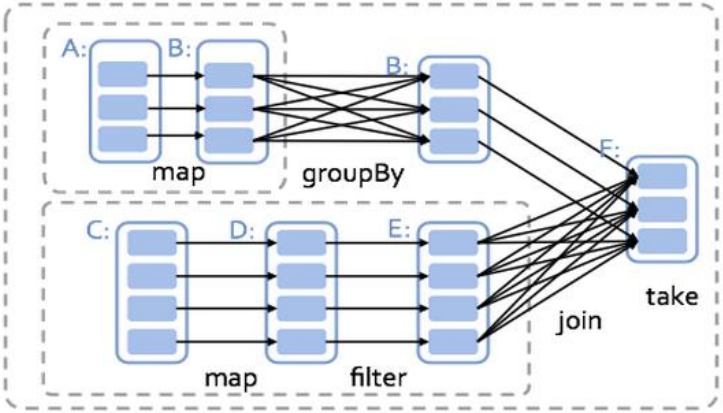
Le principali differenze sono:

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

I dati intermedi sono conservati in memoria



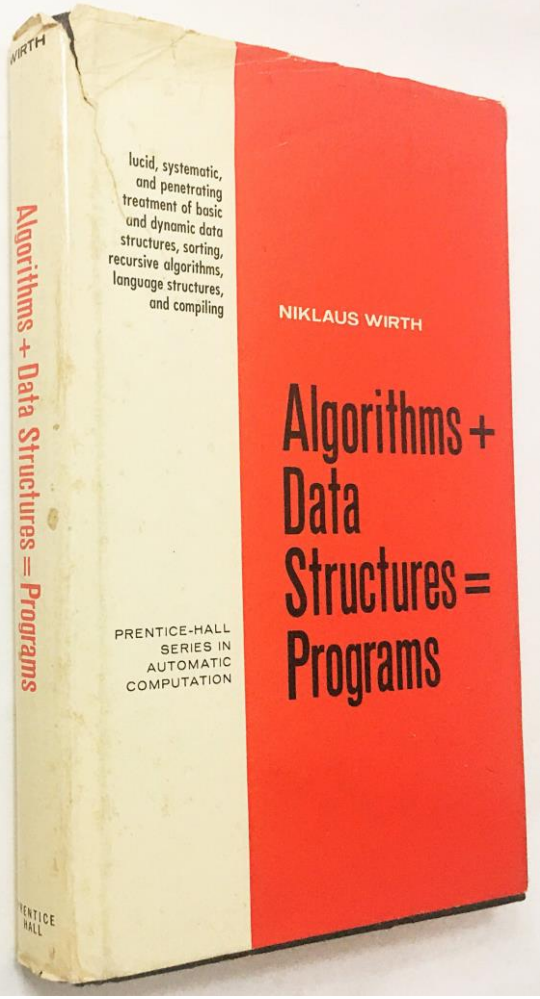
Il grafo delle operazioni (**DAG**= *Direct Acyclic Graph*) può essere più complicato

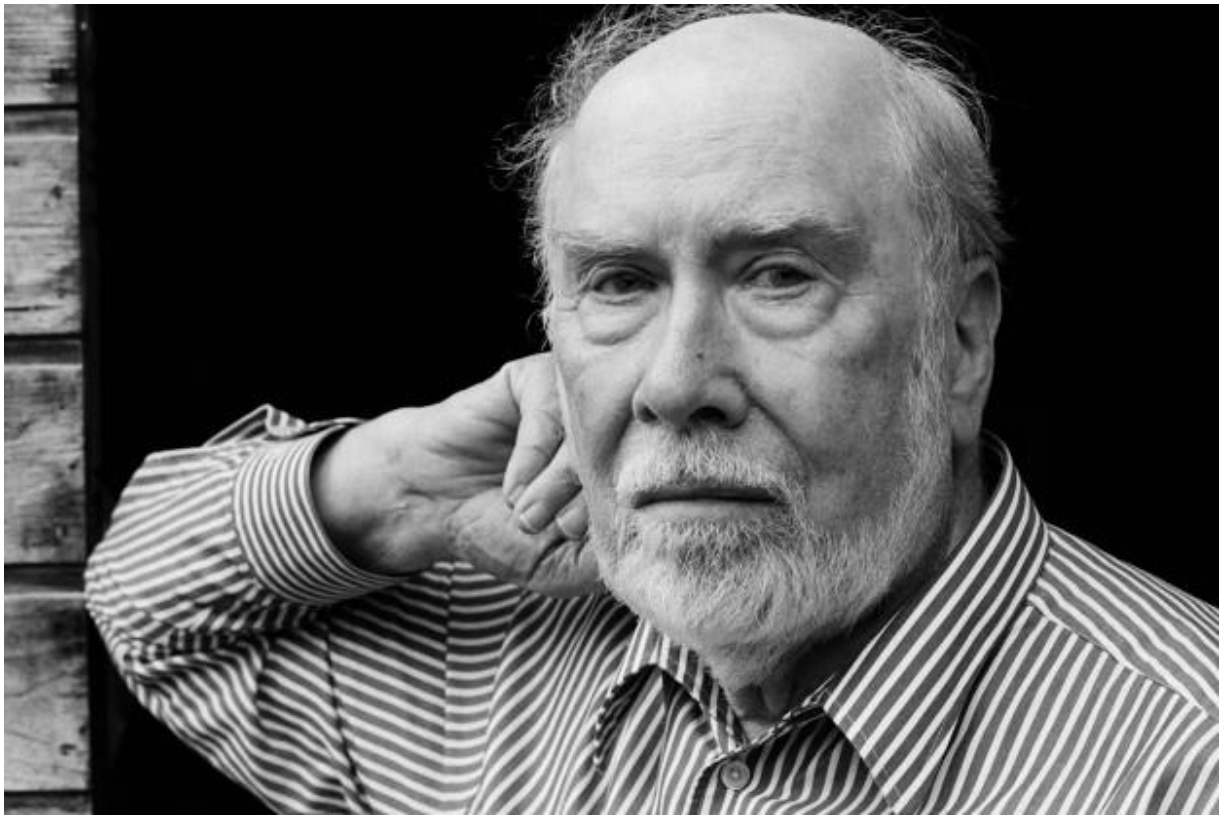


Come definisco il DAG?

Le strutture dati in Apache Spark

Le strutture dati condizionano il modo in cui creiamo i programmi





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 *Dataset* sono unificati

  « A *Dataset* is a strongly typed collection of domain-specific objects that can be transformed in parallel using functional or relational operations. Each Dataset also has an untyped view called a *DataFrame*, which is a Dataset of Row »

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

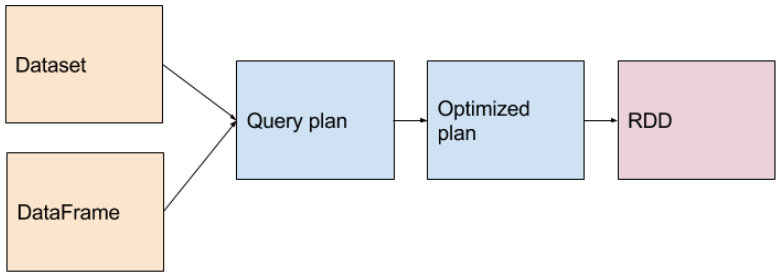
RDD sono considerate un'API a basso livello: permettono maggiore controllo, ma sono meno ottimizzabili da Spark. Inoltre sono un po' più complicate da usare.

Schema

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

Prestazioni

DataFrame e Dataset sono ottimizzate (*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

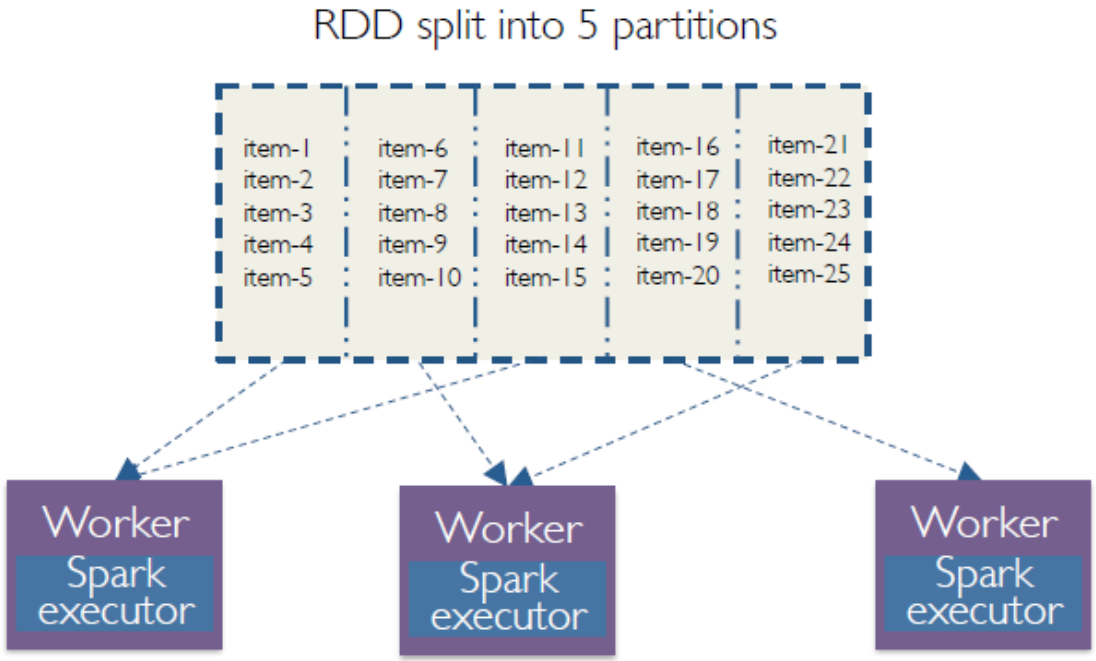
RDD e *DataFrame* sono disponibili per tutti i linguaggi

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()
```



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 le partizioni

```
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)
```

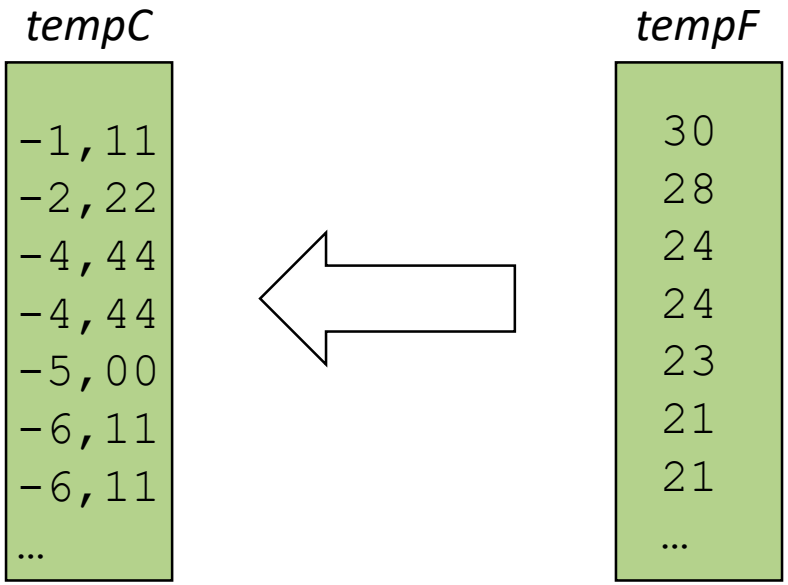
```
7
```

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»

Il 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
```

Non succede nulla

```
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
```

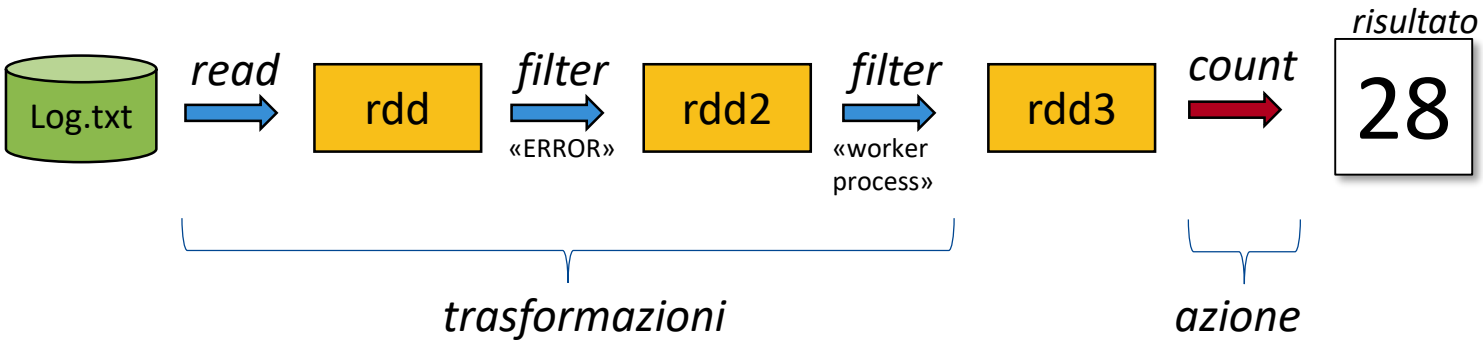
Non succede nulla

```
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
```

Non succede nulla

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

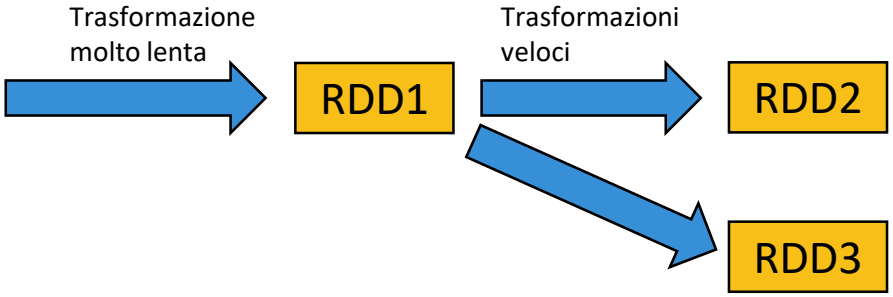
Parte il calcolo



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

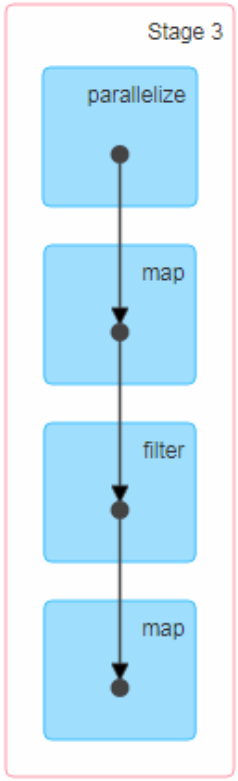
Lineage

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

Se una partizione di un RDD si perde (ad es. per il fallimento di un nodo) Spark deve essere in grado di ricostruirla.

Ogni RDD sa come ricostruire il suo contenuto in base agli RDD da cui dipende.

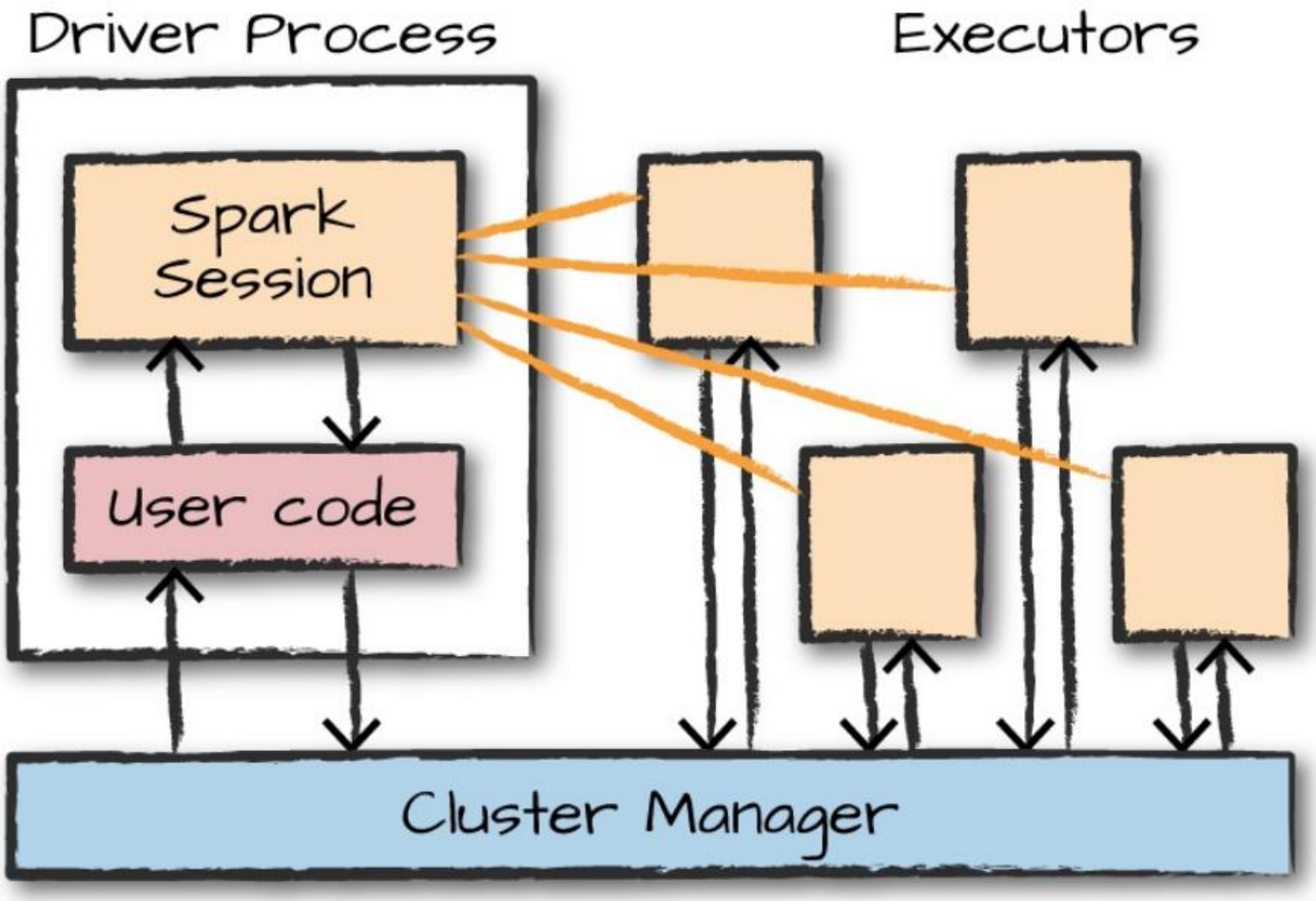
Questa informazione si chiama «lineage»



L’applicazione 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 di oggi

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()
```

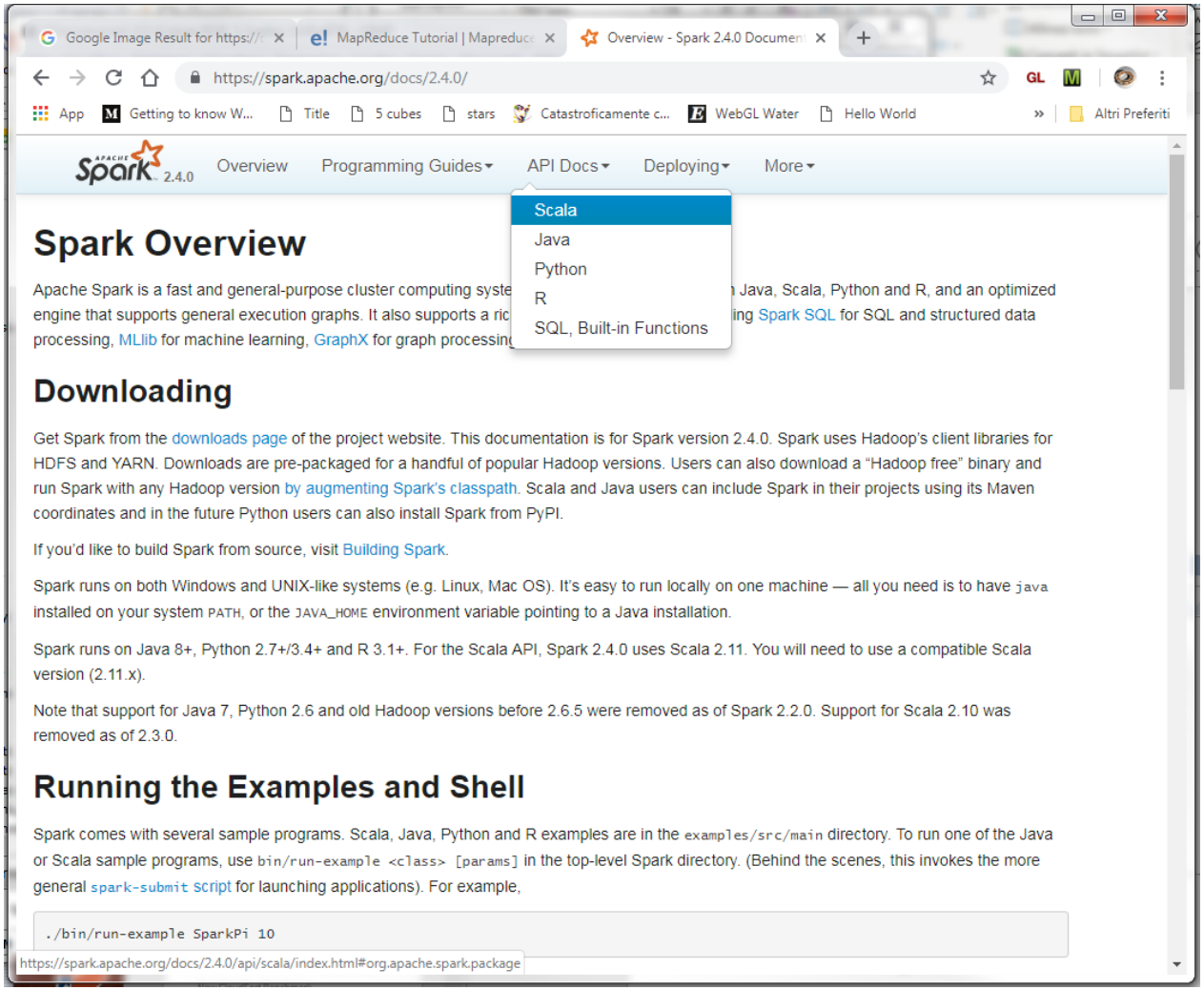
```
root
 |-- id: decimal(3,-11) (nullable = true)
 |-- date: string (nullable = true)
 |-- time: string (nullable = true)
 |-- lat: double (nullable = true)
 |-- long: double (nullable = true)
 |-- country: string (nullable = true)
 |-- city: string (nullable = true)
 |-- area: string (nullable = true)
 |-- 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

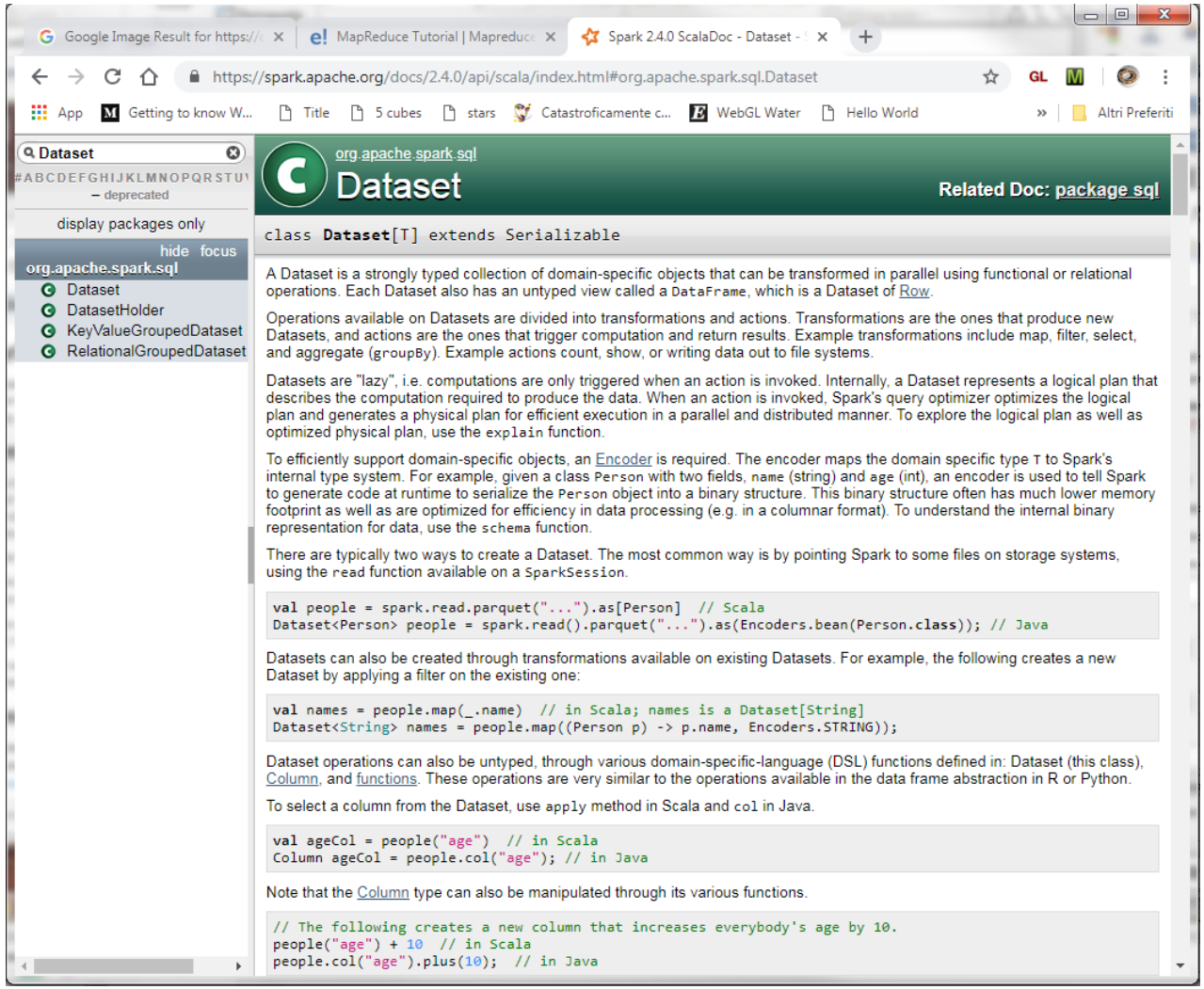
La documentazione

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



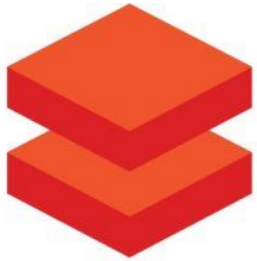
La documentazione

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



Un po’ di coordinate

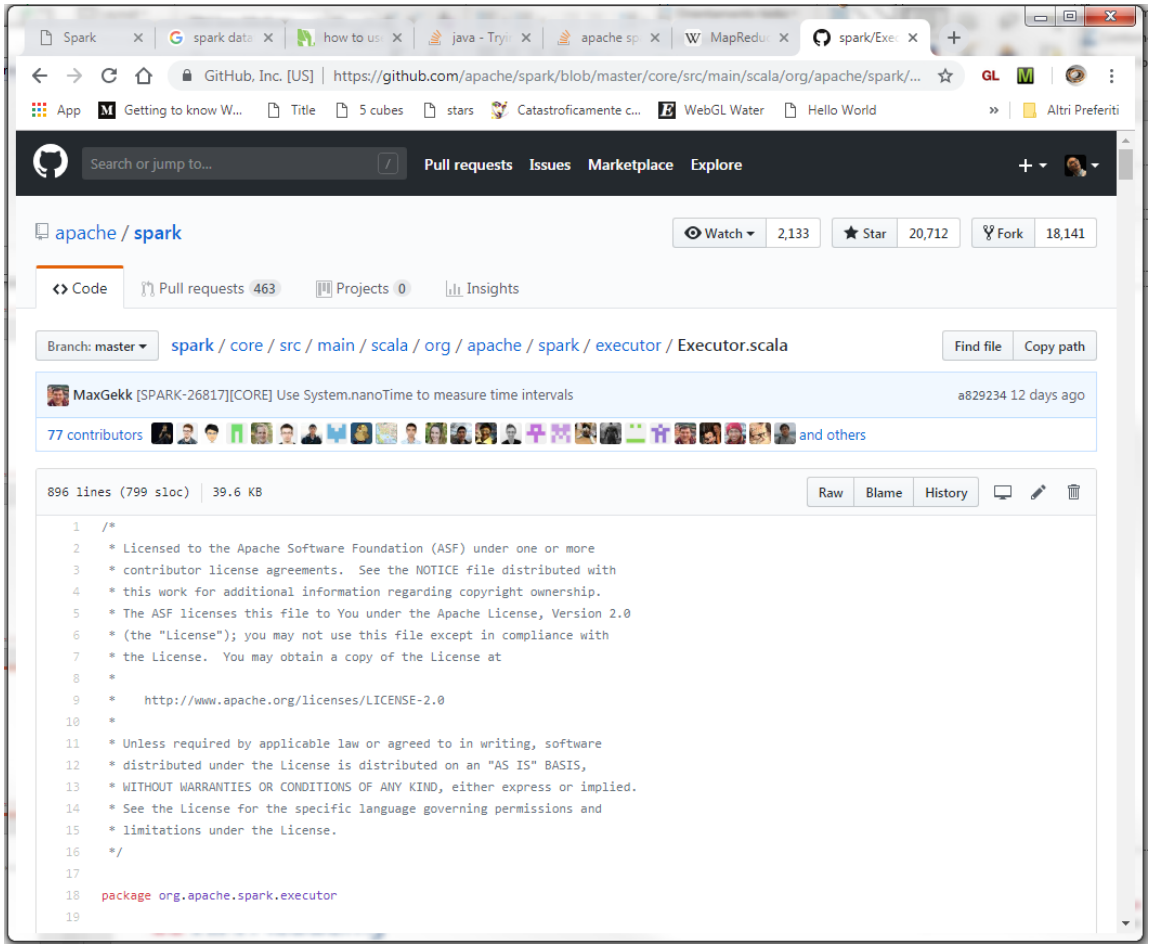
cloudera

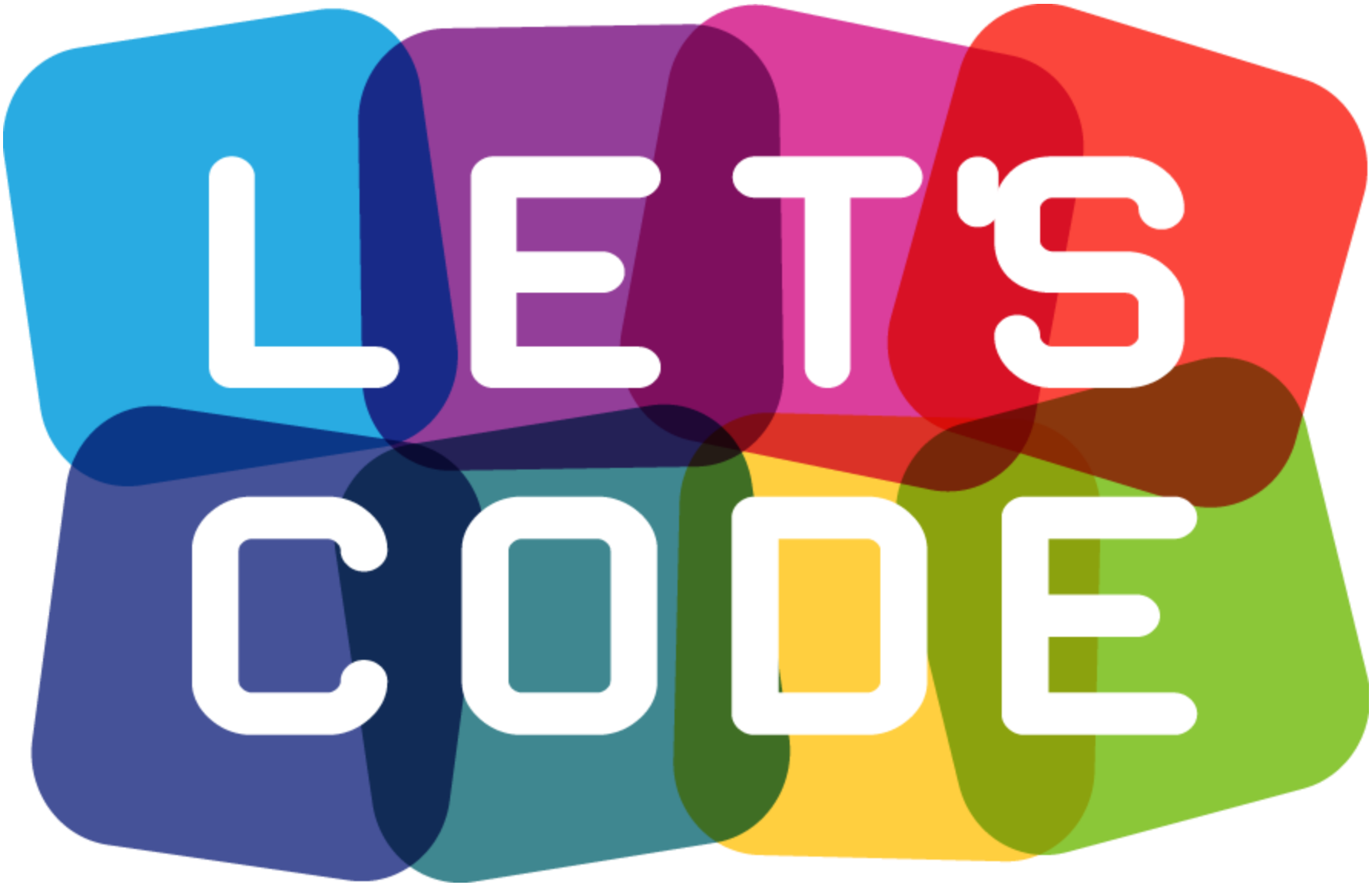


databricks®

Apache Spark è OpenSource

https://github.com/apache/spark/blob/master/core/src/...




LET'S
CODE