Terzo giorno: mattina

Agenda 5/8

RDD

Introduzione
Caratteristiche principali

Azioni e trasformazioni principali

Coppie chiave-valore

Partizioni

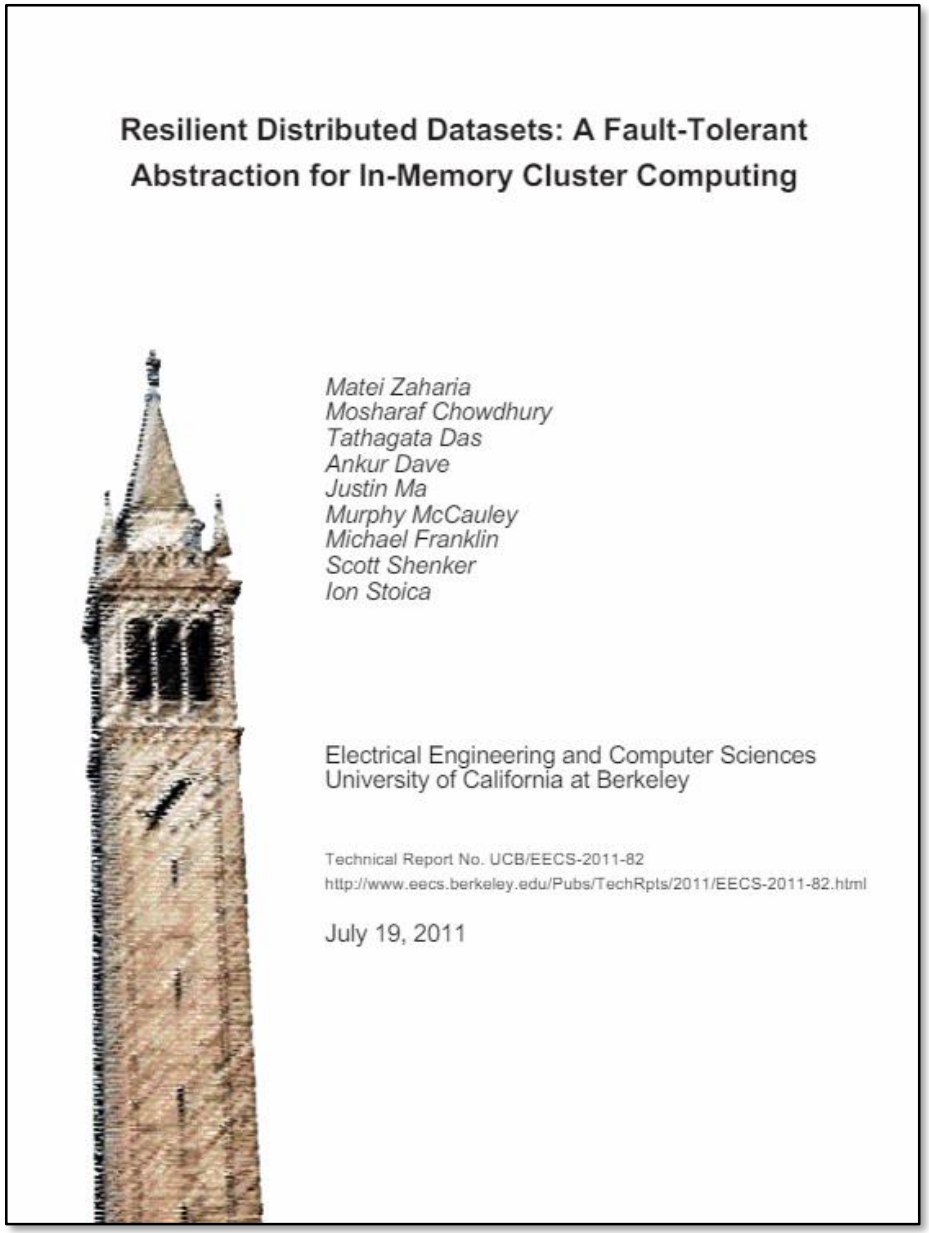
Accumulatori

Variabili broadcast

Esercizi di laboratorio
Esercizi di recapitolazione

RDD

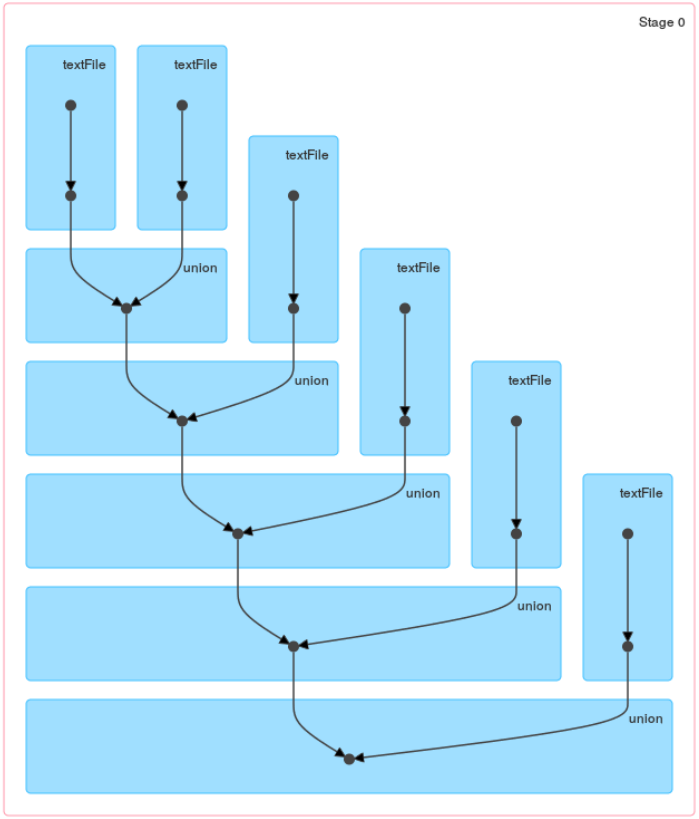
- Resilient
- Distributed
- Datasets



RDD : ***Resilient***

Gli RDD sono in grado di fronteggiare il fallimento di un nodo grazie al ***RDD lineage graph***.

In caso di guasto di un nodo e di perdita di una partizione dei dati Spark è in grado di ricostruire i dati mancanti rieseguendo le operazioni che li hanno prodotti

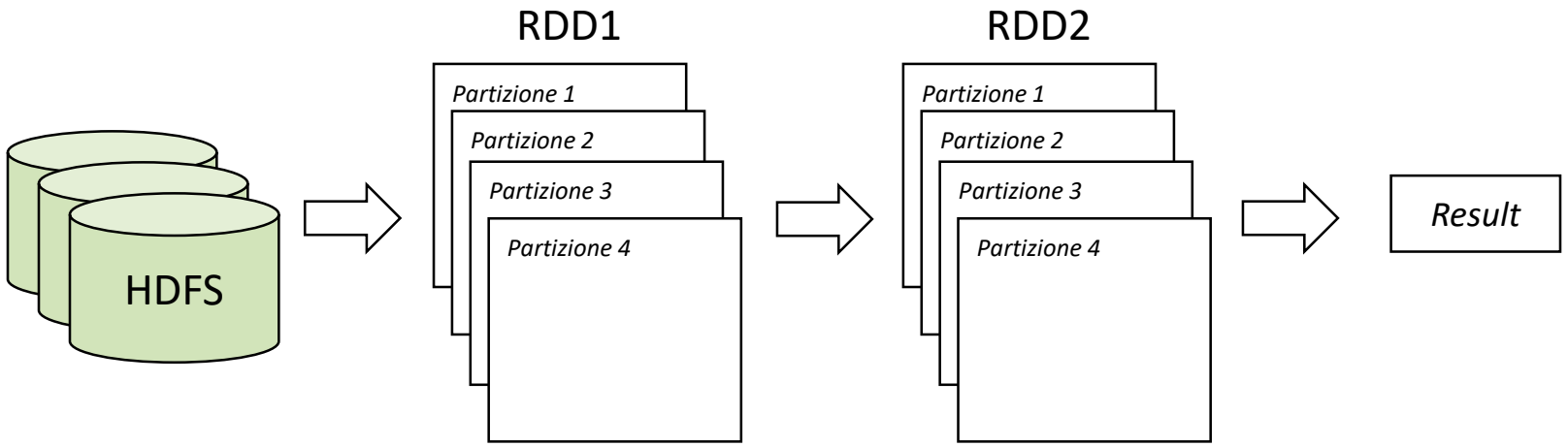


RDD : ***Distributed***

I dati contenuti in un RDD sono distribuiti («partizionati») sui nodi del cluster

- Il programmatore può scrivere un codice relativamente semplice in cui l'RDD è molto simile ad un container del linguaggio scelto (ad es. Scala).
- Il codice viene eseguito parallelamente sui nodi che ospitano le diverse partizioni

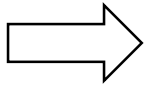
```
sc.textFile(filename, 4)
        .filter(s => s.toLowerCase().contains("duca"))
        .count()
```



RDD : ***Datasets***

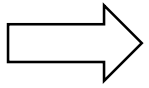
- L'RDD è una collezione di dati partizionati
- I valori che costituiscono la collezione sono:
  - Istanze di tipi primitivi (JVM) o
  - Combinazioni (es. ennuple o altri oggetti) di altri valori

```
val rdd1 = sc.textFile(filename)
```



RDD[String]

```
val rdd2 = rdd1.map(s=>(s,1))
```



RDD[(String, Int)]

RDD

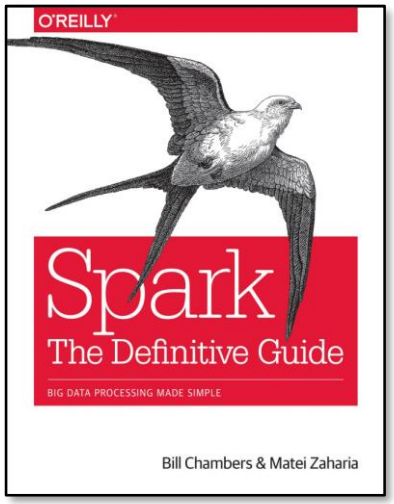
- API di basso livello:
  - Il cuore di Spark
  - Massimo controllo sul partizionamento
  - I dati sono oggetti della JVM: sottoposti a garbage collection e serializzazione
- Fortemente tipizzato
- Codice «legacy»

RDD

Chapter 12. Resilient Distributed Datasets (RDDs)

WARNING

If you are brand new to Spark, this is not the place to start. Start with the Structured APIs, you'll be more productive more quickly!



Apache Spark: 3 Reasons Why You Should Not Use RDDs

When it comes to building an enterprise-grade app in Spark, RDD isn't a good choice. Why? And if RDD is not a good choice, then what should we use?

by Himanshu Gupta Mar. 08, 18 · Big Data Zone

Quando usare gli RDD?

- Si ha bisogno del massimo controllo sulla distribuzione fisica dei dati sui nodi del cluster.
- Bisogna manutenere od utilizzare del codice che usa gli RDD
  (n.b. la maggior parte degli esempi che si trovano su web è basata sugli RDD)
- È necessario usare delle ***variabili condivise***

Inoltre:

Comprendere gli RDD è utile durante il debugging e la messa a punto:

- Tutti i workflow di Spark si riducono alla manipolazione di RDD
- Diversi strumenti di analisi e diagnostica (es. la *Spark UI*) fanno riferimento agli RDD anche se il codice sorgente usa *DataFrames* e *Datasets*

RDD in Python

Python ha prestazioni più basse usando gli RDD:

I valori dei record dell'RDD devono essere:

- Serializzati verso il processo Python
- Elaborati da Python
- Gli eventuali valori prodotti vanno serializzati e reintrodotti nella JVM

In Python si raccomanda di usare principalmente l'API di alto livello (*DataFrame*) e usare gli RDD solo se assolutamente necessario.

SparkContext

- Lo ***SparkContext*** è il punto di ingresso per le API di basso livello.
- È possibile ottenere lo SparkContext a partire da una SparkSession:

```
spark.sparkContext
```

- Negli ambienti interattivi (*spark-shell* o *Spark Notebook*) in genere è definita una variabile dedicata : ***sc***.

Come si crea/ottiene un RDD

- Leggendo un file

```
val rdd = sc.textFile("hdfs://path/to/my/file.txt")
```

- Usando il metodo *parallelize* dello *Spark Context*

```
val rdd = sc.parallelize(1 to 10000000)
```

- L'attributo *rdd* permette di accedere al RDD di un *DataFrame* o di un *Dataset*

```
var rdd = df.rdd
```

Gli RDD sono fortemente tipizzati

```
val rdd1 = sc.parallelize(1 to 10000000)
```

```
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[7] at parallelize at <console>:67
```

Took: 0.998s, at 2019-03-13 10:59

```
val rdd2 = sc.textFile("data/divina_commedia.txt")
```

```
rdd2: org.apache.spark.rdd.RDD[String] = data/divina_commedia.txt MapPartitionsRDD[9] at textF
ile at <console>:67
```

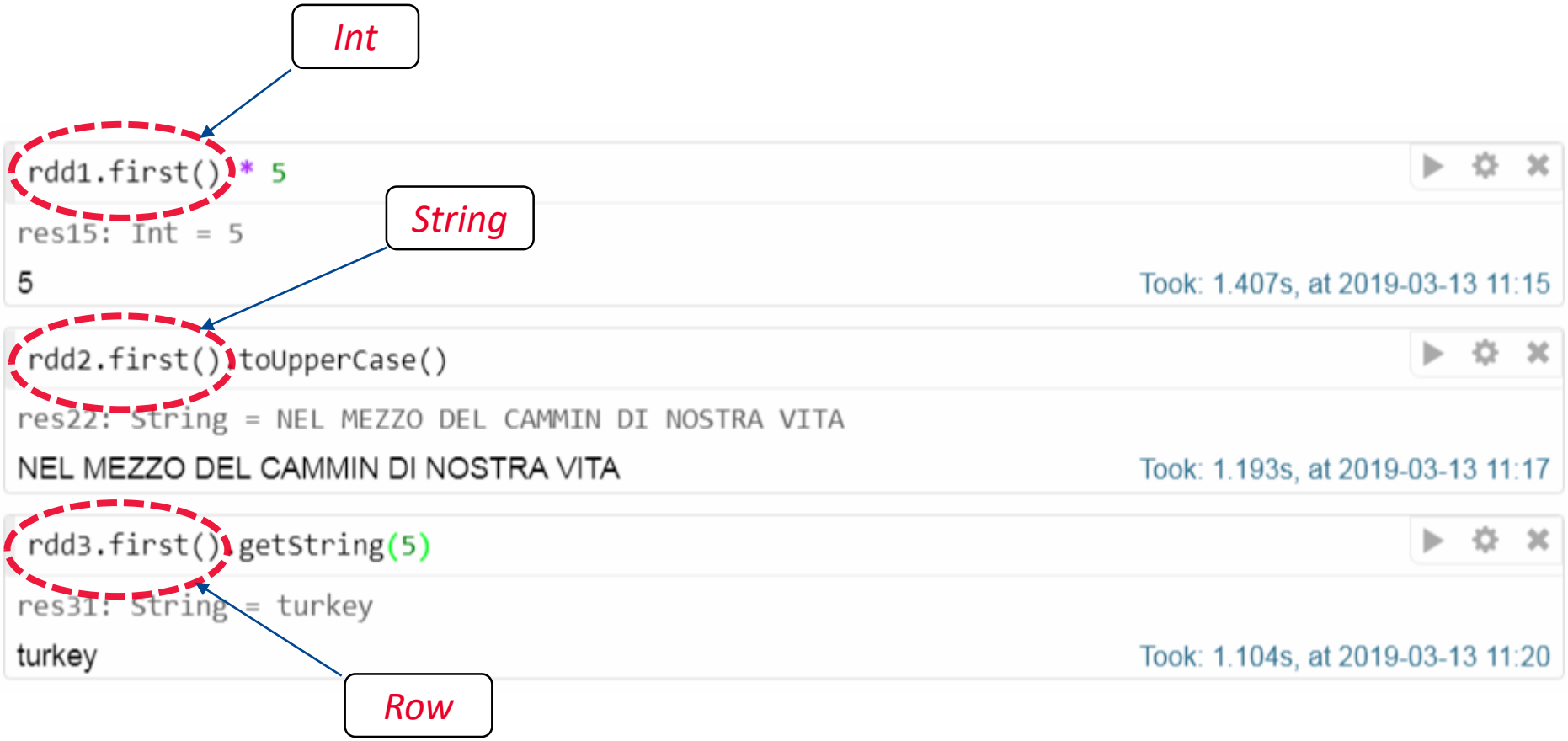
Took: 0.965s, at 2019-03-13 10:59

```
val rdd3 = df.rdd
```

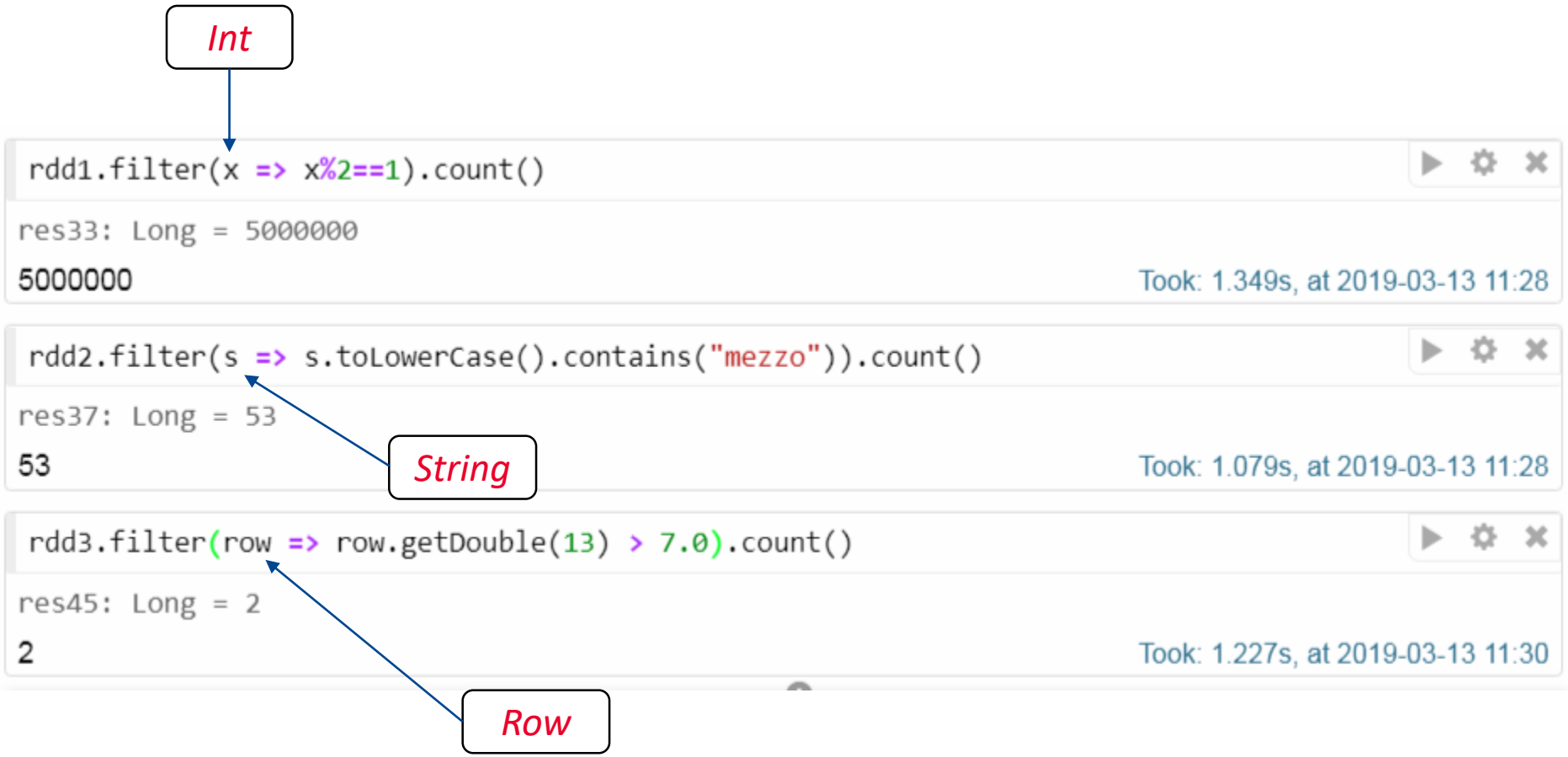
```
rdd3: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[13] at rdd at <con
sole>:71
```

Took: 2.042s, at 2019-03-13 11:00

Gli RDD sono fortemente tipizzati



Gli RDD sono fortemente tipizzati



Trasformazioni e azioni

- Per le azioni ci sono molte somiglianze con *DataFrame* e *Dataset*:

```
rdd.count()
rdd.take(20)
rdd.first()
rdd.collect()
rdd.reduce((a,b)=>a+b)
ecc.
```

- Anche per le trasformazioni:

```
rdd.map()
rdd.distinct()
ecc.
```

Trasformazioni e azioni

Un esempio con *map()* & *reduce*():

Vogliamo calcolare la somma dei cubi degli interi fra uno e dieci milioni.

```
val rdd = sc.parallelize(1 to 10000001)
val sum = rdd
      .map(x=>BigDecimal(x))
      .map(x=>x*x*x)
      .reduce((a,b)=>a+b)
```

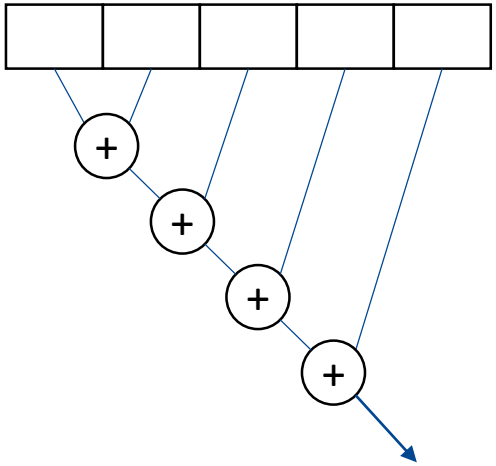
Reduce

```
val sum = rdd.reduce((a,b)=>a+b)
```

Funzione di riduzione
(a due parametri)

Formulazione alternativa

```
val sum = rdd.reduce( _ + _ )
```



Confronto fra RDD e *Dataset* di *case class*

`RDD[MyClass]` vs. `Dataset[MyClass]`

- Praticamente identici come API
- *Dataset* si avvantaggia delle ottimizzazioni della API strutturata (*Catalyst* & *Tungsten*)
- Gli RDD non usano queste ottimizzazioni

Struttura interna degli RDD

Internamente ogni RDD è caratterizzato da 5 proprietà:

- Una lista di partizioni
- Una lista di dipendenze da altre RDD
- Una funzione in grado di calcolare il valore di ogni partizione

E, opzionalmente:

- Un ***Partitioner***: l'oggetto che controlla il partizionamento
- Una lista di locazioni preferite dove calcolare i valori di ogni partizione

Tipi diversi di RDD implementano in modo diverso le cinque proprietà

Struttura interna degli RDD

```
val rdd = sc.parallelize(1 to 10000, 8)
```

```
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[12] at parallelize at <console>:67
```

Took: 0.779s, at 2019-03-13 21:10

```
val rdd2 = rdd.map(x=>x*3)
```

```
rdd2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[13] at map at <console>:69
```

Took: 0.801s, at 2019-03-13 21:10

```
rdd2.partitions.length
```

```
res46: Int = 8
```

8

Took: 0.973s, at 2019-03-13 21:10

```
rdd2.dependencies(0).rdd
```

```
res48: org.apache.spark.rdd.RDD[_] = ParallelCollectionRDD[12] at parallelize at <console>:67
```

ParallelCollectionRDD[12] at parallelize at <console>:67

Took: 0.774s, at 2019-03-13 21:11

```
rdd2.partitioner
```

```
res50: Option[org.apache.spark.Partitioner] = None
```

None

Took: 0.819s, at 2019-03-13 21:11

```
rdd2.preferredLocations(rdd2.partitions(0))
```

```
res60: Seq[String] = List()
```

empty seq

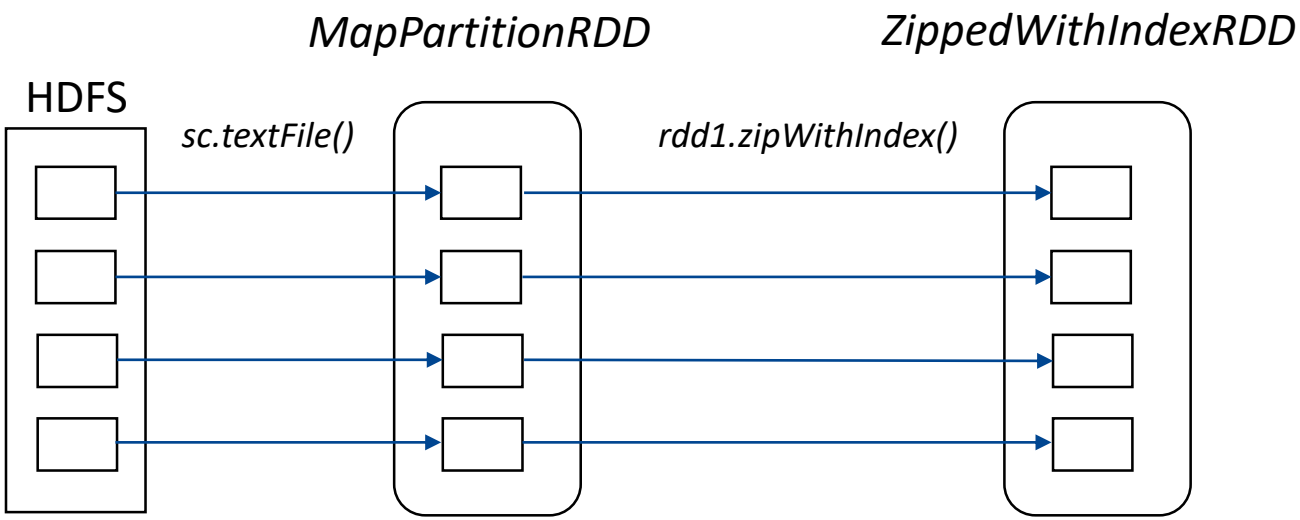
Took: 0.989s, at 2019-03-13 21:12

Tipi di RDD

Esistono tante sottoclassi di RDD elencati nella documentazione.

In gran parte servono all'API dei *DataFrame* per creare i piani di esecuzione ottimizzati.

Funzioni e metodi diversi creano RDD di tipo diverso.



Tipi di RDD

La stessa funzione può creare più di un RDD:

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

```
rdd: org.apache.spark.rdd.RDD[String] = data/divina_commedia.txt MapPartitionsRDD[15] at textF
ile at <console>:67
```

Took: 3.523s, at 2019-03-13 21:46

```
rdd.dependencies.length
```

```
res81: Int = 1
1
```

Took: 0.946s, at 2019-03-13 21:48

```
val rdd0 = rdd.dependencies(0).rdd
```

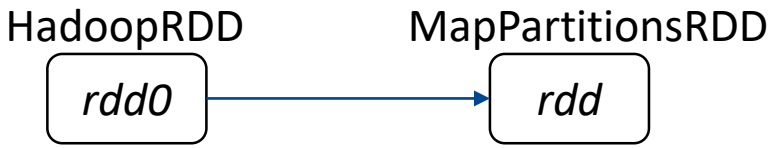
```
rdd0: org.apache.spark.rdd.RDD[_] = data/divina_commedia.txt HadoopRDD[14] at textFile at <con
sole>:67
```

Took: 0.800s, at 2019-03-13 21:48

```
rdd0.dependencies.length
```

```
res83: Int = 0
0
```

Took: 0.970s, at 2019-03-13 21:48



Tipi di RDD

Come utenti siamo interessati principalmente a due tipi di RDD:

- RDD «generici» (Es.: *RDD[Int], RDD[String], RDD[Row],* ecc.)
- RDD «chiave-valore».

Gli RDD *chiave-valore* possono essere creati con una semplice trasformazione:

```
val rdd2 = rdd1.map(s=>(s.toLowerCase,s))
```

Oppure si può usare un metodo dedicato:

```
val rdd2 = rdd1.keyBy(_.toLowerCase)
```

Non sono istanze di una classe particolare, ma offrono dei metodi che non sono normalmente accessibili.

I nomi di questi metodi sono in genere della forma: `rdd.qualcosaByKey(...)`

(ad es. *groupByKey(), reduceByKey(), aggregateByKey()*, ecc.)

RDD chiave-valore

In questo esempio *rdd* è un RDD generico, mentre *rdd2* è di tipo *chiave-valore*.

```
rdd.reduceByKey(_+_)
```

```
<console>:71: error: value reduceByKey is not a member of org.apache.spark.rdd.RDD[Int]
       rdd.reduceByKey(_+_)
           ^
```

```
val rdd2 = rdd.map(x=>(x,1))
```

```
rdd2: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[5] at map at <console>:69
```

Took: 1.102s, at 2019-03-13 19:03

```
rdd2.reduceByKey(_+_)
```

```
res9: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[6] at reduceByKey at <console>:73
```

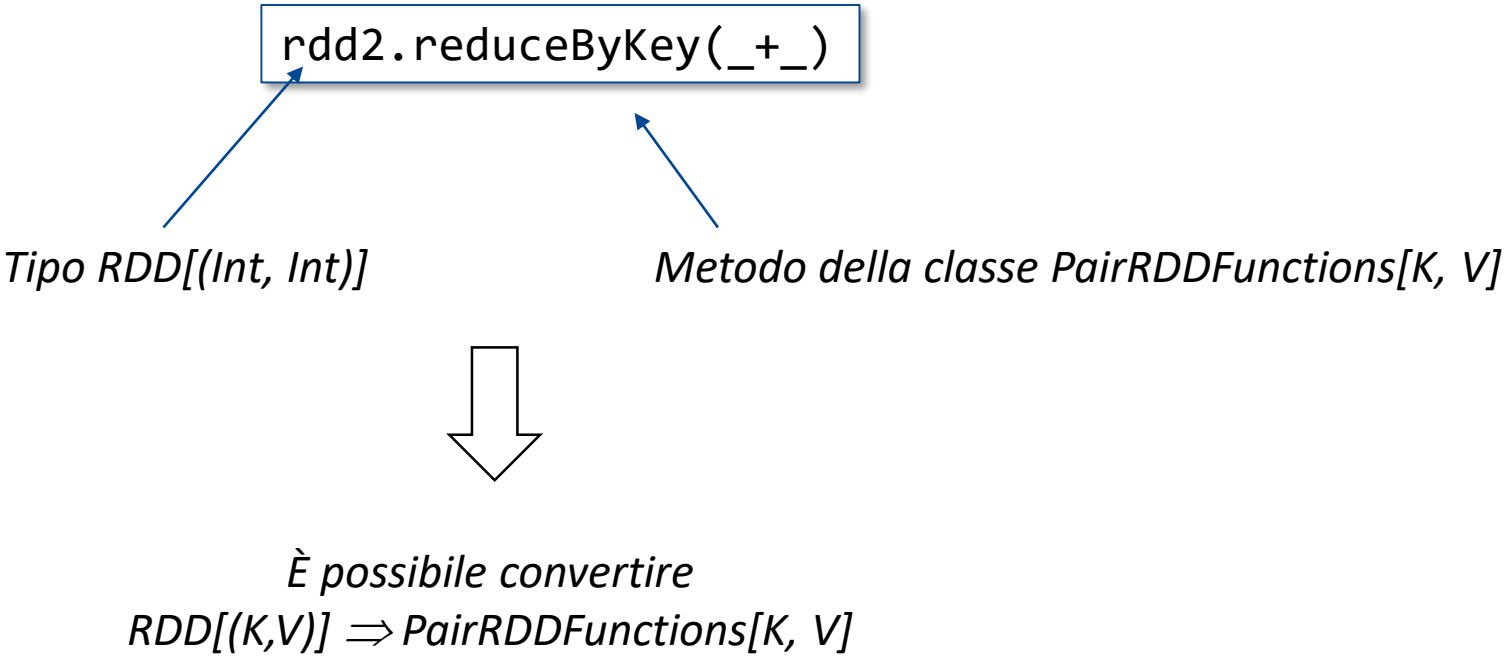
ShuffledRDD[6] at reduceByKey at <console>:73

Took: 1.626s, at 2019-03-13 19:03

RDD chiave-valore

Nella documentazione bisogna guardare la classe: *PairRDDFunctions*

Viene fatta una conversione implicita:



Le ennuple in Scala

Le ennuple di Scala sono usate molto spesso quando si lavora con gli RDD: in particolare le coppie *chiave-valore*

Crea una ennupla

```
val rdd2 = rdd1.map(name => (name, name.toUpperCase(), 1))
```

```
rdd2: org.apache.spark.rdd.RDD[(String, String, Int)] = MapPartitionsRDD[1] at map at <console
>:69
```

Took: 1.610s, at 2019-03-13 17:12

Accedi al primo elemento della ennupla

```
val rdd3 = rdd2.map(tuple => tuple._1)
```

```
rdd3: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at map at <console>:71
```

Took: 1.298s, at 2019-03-13 17:13

```
val rdd4 = rdd2.map(tuple => tuple._3)
```

```
rdd4: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[3] at map at <console>:71
```

Took: 1.192s, at 2019-03-13 17:13

Word Count con gli RDD

Esercizio:

- Leggere un file di testo e dividere le linee in parole (separate da spazi)
- Contare le occorrenze di ogni parola, considerando solo le parole di almeno 8 lettere
- Ordinare i risultati mettendo prima le parole più usate
- Scrivere il risultato su disco

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile
        .flatMap(line => line.toLowerCase().split(" "))
        .filter(word=>word.length>=8)
        .map(word=>(word,1))
        .reduceByKey(_+_)
        .sortBy(-_._2)
counts.saveAsTextFile("hdfs://...")
```

Confronto : *RDD* vs. *DataFrame*

DataFrame

```
val textFile = spark.read.text("hdfs://...")
val counts = textFile
        .select(explode(split(lower($"value")," ")) as "word")
        .where(length($"word")>=8)
        .groupBy("word").count()
        .orderBy(desc("count"))
counts.write.csv("hdfs://...")
```

RDD

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile
        .flatMap(line => line.toLowerCase().split(" "))
        .filter(word=>word.length>=8)
        .map(word=>(word,1))
        .reduceByKey(_+_)
        .sortBy(-_._2)
counts.saveAsTextFile("hdfs://...")
```

Confronto : *RDD* vs. *DataFrame*

DataFrame

```
val textFile = spark.read.text("hdfs://...")
val counts = textFile
       .select(explode(split(lower($"value")," ")) as "word")
       .where(length($"word")>=8)
       .groupBy("word").count()
       .orderBy(desc("count"))
counts.write.csv("hdfs://...")
```

RDD

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile
       .flatMap(line => line.toLowerCase().split(" "))
       .filter(word=>word.length>=8)
       .map(word=>(word,1))
       .reduceByKey(_+_)
       .sortBy(-_._2)
counts.saveAsTextFile("hdfs://...")
```

Confronto : *RDD* vs. *DataFrame*

DataFrame

```
val textFile = spark.read.text("hdfs://...")
val counts = textFile
       .select(explode(split(lower($"value")," ")) as "word")
       .where(length($"word")>=8)
       .groupBy("word").count()
       .orderBy(desc("count"))
counts.write.csv("hdfs://...")
```

RDD

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile
       .flatMap(line => line.toLowerCase().split(" "))
       .filter(word=>word.length>=8)
       .map(word=>(word,1))
       .reduceByKey(_+_)
       .sortBy(-_._2)
counts.saveAsTextFile("hdfs://...")
```

Confronto : *RDD* vs. *DataFrame*

DataFrame

```
val textFile = spark.read.text("hdfs://...")
val counts = textFile
       .select(explode(split(lower($"value")," ")) as "word")
       .where(length($"word")>=8)
       .groupBy("word").count()
       .orderBy(desc("count"))
counts.write.csv("hdfs://...")
```

RDD

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile
       .flatMap(line => line.toLowerCase().split(" "))
       .filter(word=>word.length>=8)
       .map(word=>(word,1))
       .reduceByKey(_+_)
       .sortBy(-_._2)
counts.saveAsTextFile("hdfs://...")
```

Partizioni

- Ogni RDD è distribuito su un certo numero di partizioni.
- Il numero di partizioni si ottiene con *getNumPartitions* o *partitions.length*
- E’ possibile generare un nuovo RDD con un numero prefissato di partizioni:
  - *coalesce(num)*
  - *repartition(num)*

Partitioner

- Spark ha due Partitioner già fatti:
  - *HashPartitioner* : per valori discreti
  - *RangePartitioner* : per valori continui

```
// rdd è un RDD chiave-valore
val rdd2 = rdd.partitionBy(new HashPartitioner(10))
```

- E' possibile creare nuovi Partitioner più raffinati

  (ad esempio per evitare che troppi record convergano sullo stesso nodo)

Partitioner

Per fare esperimenti con le partizioni: *glom()*

Ritorna un RDD contenente tanti array quante sono le partizioni: ogni array contiene gli elementi presenti nella partizione.

```
val rdd = sc.parallelize(1 to 61, 4)
rdd.glom().collect().foreach(x=>println(x.mkString(",")))
```

```
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
16,17,18,19,20,21,22,23,24,25,26,27,28,29,30
31,32,33,34,35,36,37,38,39,40,41,42,43,44,45
46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[11] at parallelize at <console>:73
```

Took: 1.231s, at 2019-03-14 08:40

Partitioner personalizzato

```
class MyPartitioner extends org.apache.spark.Partitioner {
  def numPartitions = 6
  def getPartition(key: Any): Int = {
    return key.asInstanceOf[Int] % 6
  }
}

val rdd2 = rdd.keyBy(v=>v).partitionBy(new MyPartitioner)
rdd2.values.glom().collect().foreach(x=>println(x.mkString(",")))
```

```
6,12,18,24,30,36,42,48,54,60
1,7,13,19,25,31,37,43,49,55,61
2,8,14,20,26,32,38,44,50,56
3,9,15,21,27,33,39,45,51,57
4,10,16,22,28,34,40,46,52,58
5,11,17,23,29,35,41,47,53,59
defined class MyPartitioner
rdd2: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[17] at partitionBy at <console>:41
```

Took: 0.941s, at 2019-03-14 08:46

Distributed shared variables

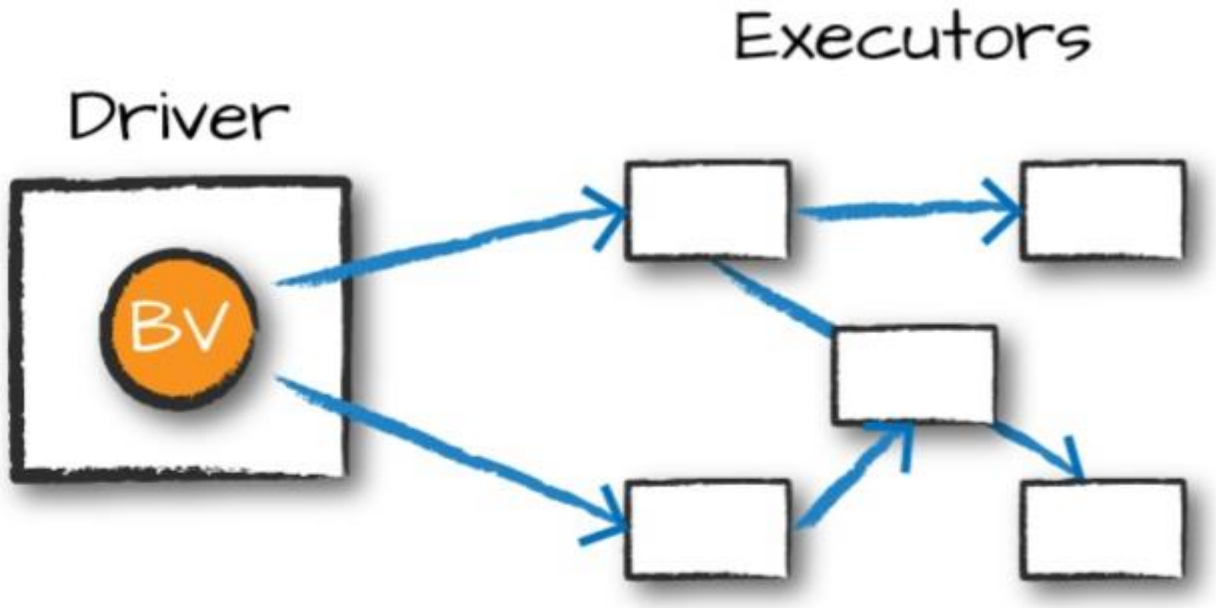
Oltre agli RDD l'API di basso livello comprende due tipi di Variabili Distribuite e Condivise:

- Variabili ***Broadcast***
- ***Accumulatori***

Possono essere utilizzate dentro le funzioni definite dall'utente. Ad es. in una trasformazione di tipo *map()* su un RDD o un DataFrame.

Variabili broadcast

- Permettono di distribuire dei valori (immutabili) su tutti i nodi in modo efficiente.
- Utile nel caso di look-up table grandi.
- Diverse trasformazioni possono utilizzare la stessa variabile più volte. Il contenuto viene trasferito sui nodi una sola volta.
- Il processo di trasferimento delle variabili broadcast è ottimizzato



Variabili broadcast

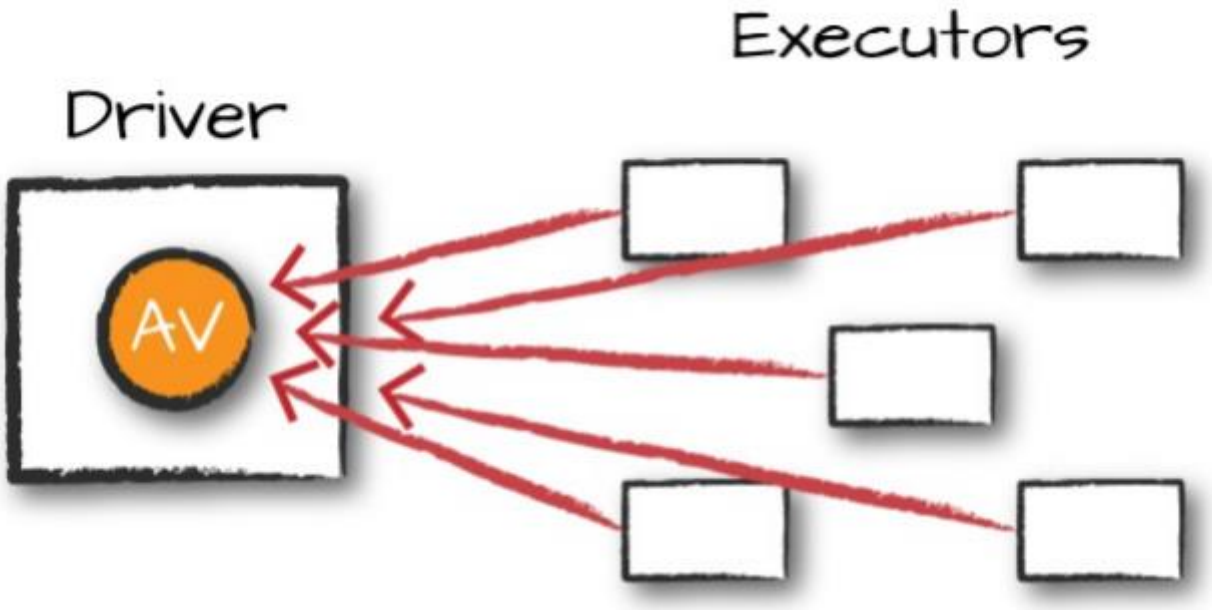
```
val myCollection =
  "Spark The Definitive Guide : Big Data Processing Made Simple"
  .split(" ")
val words = spark.sparkContext.parallelize(myCollection, 2)

val tableBv = spark.sparkContext.broadcast(Map(
  "Spark" -> 1000,
  "Definitive" -> 200,
  "Big" -> -300,
  "Simple" -> 100))

words.map(word => (word, tableBv.value.getOrElse(word, 0)))
  .sortBy(wordPair => wordPair._2)
  .collect()
```

Accumulatori

- Gli accumulatori permettono di modificare un valore all'interno di una trasformazione e di propagare il valore sul nodo *driver* in modo efficiente e resistente ai guasti.



Accumulatori

- Un'***accumulatore*** è una variabile che un cluster Spark può aggiornare ogni volta che una riga viene elaborata.

- Può essere usato per debugging o per creare un aggregatore a basso livello.

- Il valore dell'accumulatore viene modificato in base ad una operazione associativa e commutativa (es. somma o moltiplicazione) che quindi può essere calcolata efficientemente in parallelo

- Spark supporta in modo nativo solo accumulatori numerici, ma è possibile definire nuovi tipi di accumulatori

Accumulatori

- L’aggiornamento degli accumulatori avviene solo durante l’esecuzione delle azioni. Quindi se un task viene ricalcolato (per via del guasto di un nodo) questo non falsa il risultato finale.

- L’accumulatore viene aggiornato tenendo conto del modello di valutazione «pigra» di Spark. Se un’operazione su un RDD aggiorna l’accumulatore, questo aggiornamento avviene solo quando l’RDD viene effettivamente calcolato

- Gli accumulatori possono essere con nome o senza nome.

Accumulatori

```
val acc = spark.sparkContext.longAccumulator("AccName")

rdd.filter(x => x%13==0).foreach(x => acc.add(x))

println(acc.value)
```