Apache Spark - Introduzione

Limiti di MapReduce

- Eccessive operazioni di I/O (spesso collo di bottiglia per l'esecuzione).
- Rigidità del flusso di esecuzione (fase map, seguita da fase shuffle and sort, seguita da fase reduce).
- Overhead iniziale ogni volta che si avvia un job MapReduce.
- Non si adatta bene a flussi di lavoro iterativi, come PageRank o algoritmi di machine learning.
- Rende necessario gestire manualmente dettagli di basso livello (ad esempio, individuare manualmente quando una chiave cambia, nei reducer).
- Operazioni che dovrebbero essere semplici, come ordinamento o join di due dataset, diventano complicate.

Punti di forza di MapReduce

Alcune caratteristiche di MapReduce sono molto importanti, e sono alla base anche di Apache Spark.

- Scalabilità (lo stesso codice può essere eseguito su un cluster con un solo nodo o con 1000 nodi).
- Paradigma shared-nothing (evita problemi di sincronizzazione e condivisione della memoria che caratterizzano altri paradigmi).
- Fault tolerance (i dati e le computazioni intermedie sono sempre al sicuro, anche in caso di malfunzionamenti).
- Accesso semplice ai dati contenuti in un file system distribuito (HDFS).

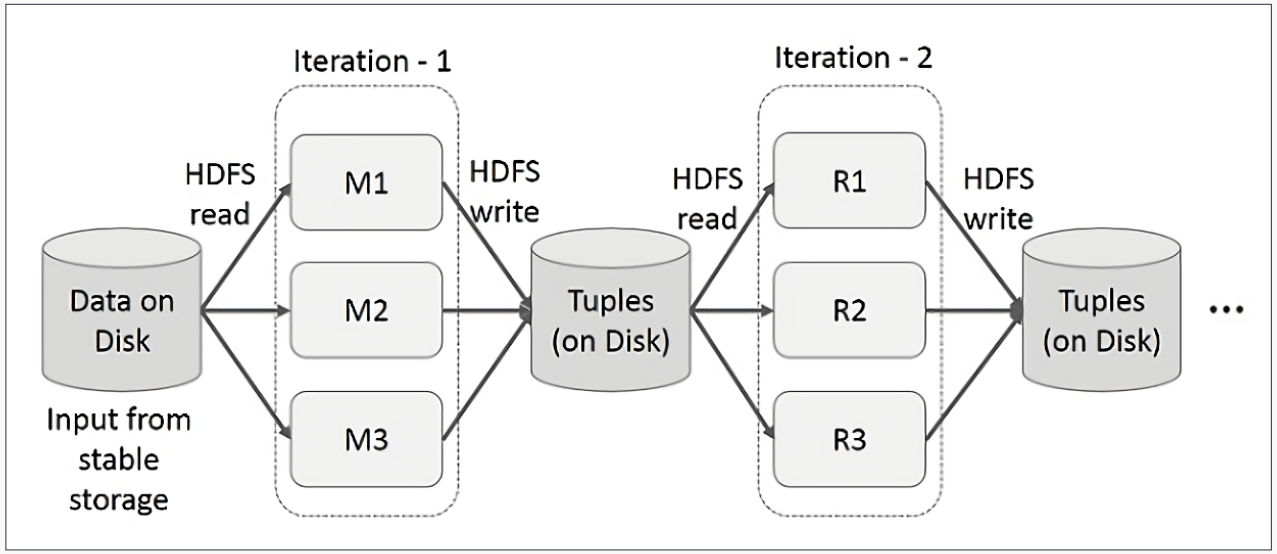
Osservazioni

- La memoria principale dei cluster è spesso poco utilizzata:
  - in certi casi i dati da analizzare possono essere contenuti completamente (o in gran parte) nella memoria RAM dei nodi del cluster;
  - il costo della memoria diminuisce sempre più.
- Alcune delle operazioni di I/O compiute da MapReduce sono ridondanti, specialmente nel caso di algoritmi iterativi:
  - i risultati intermedi spesso non sono importanti, e salvarli su HDFS comporta uno spreco di tempo.
- Sono necessarie API di alto livello per rendere semplice realizzare le operazioni più comuni.

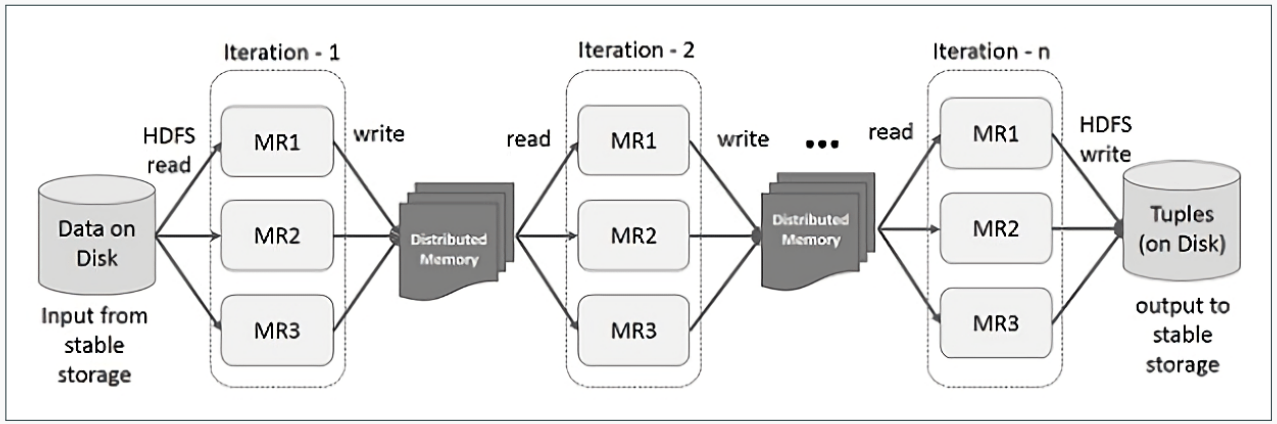
Apache Spark

- Spark è un framework open source per il calcolo distribuito proposto come alternativa a MapReduce.
- Sviluppato nel 2009 presso il laboratorio di ricerca AMPLab di Berkeley, e successivamente donato alla Apache Software Foundation.
- Il suo punto di forza principale è la possibilità di effettuare elaborazioni in-memory, e di salvare i risultati intermedi in memoria (caching).
- Adatto sia per flussi di lavoro iterativi (machine learning) che interattivi (ad esempio, query SQL).
- Fornisce API di alto livello per Python, Java e Scala.
- Permette di combinare in modo semplice ed efficiente diversi carichi di lavoro (query SQL, training di algoritmi di machine learning, operazioni su grafi, stream processing).

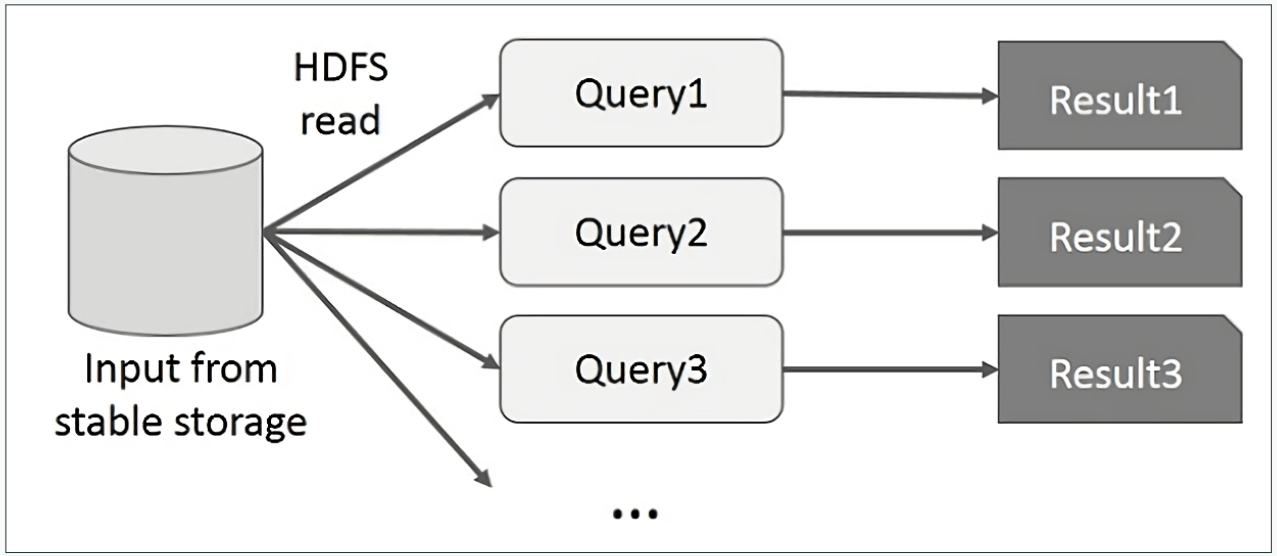
MapReduce



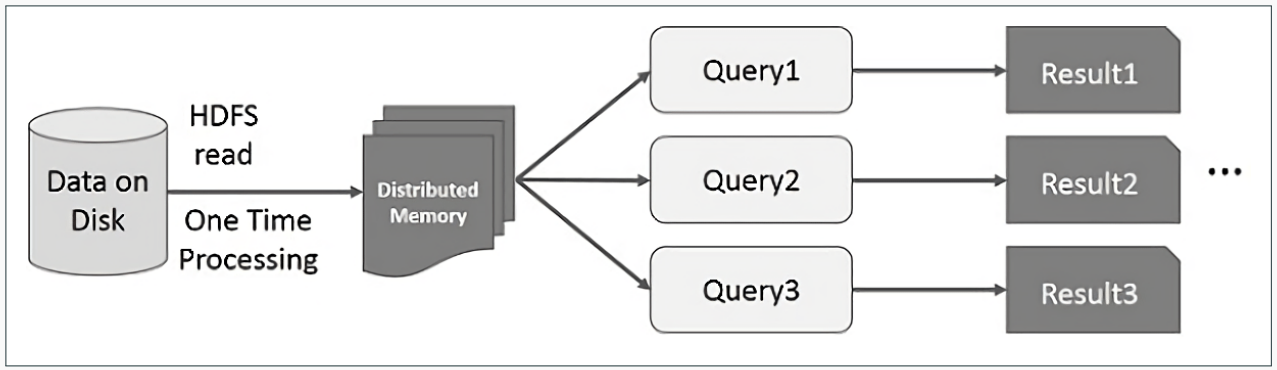
Spark



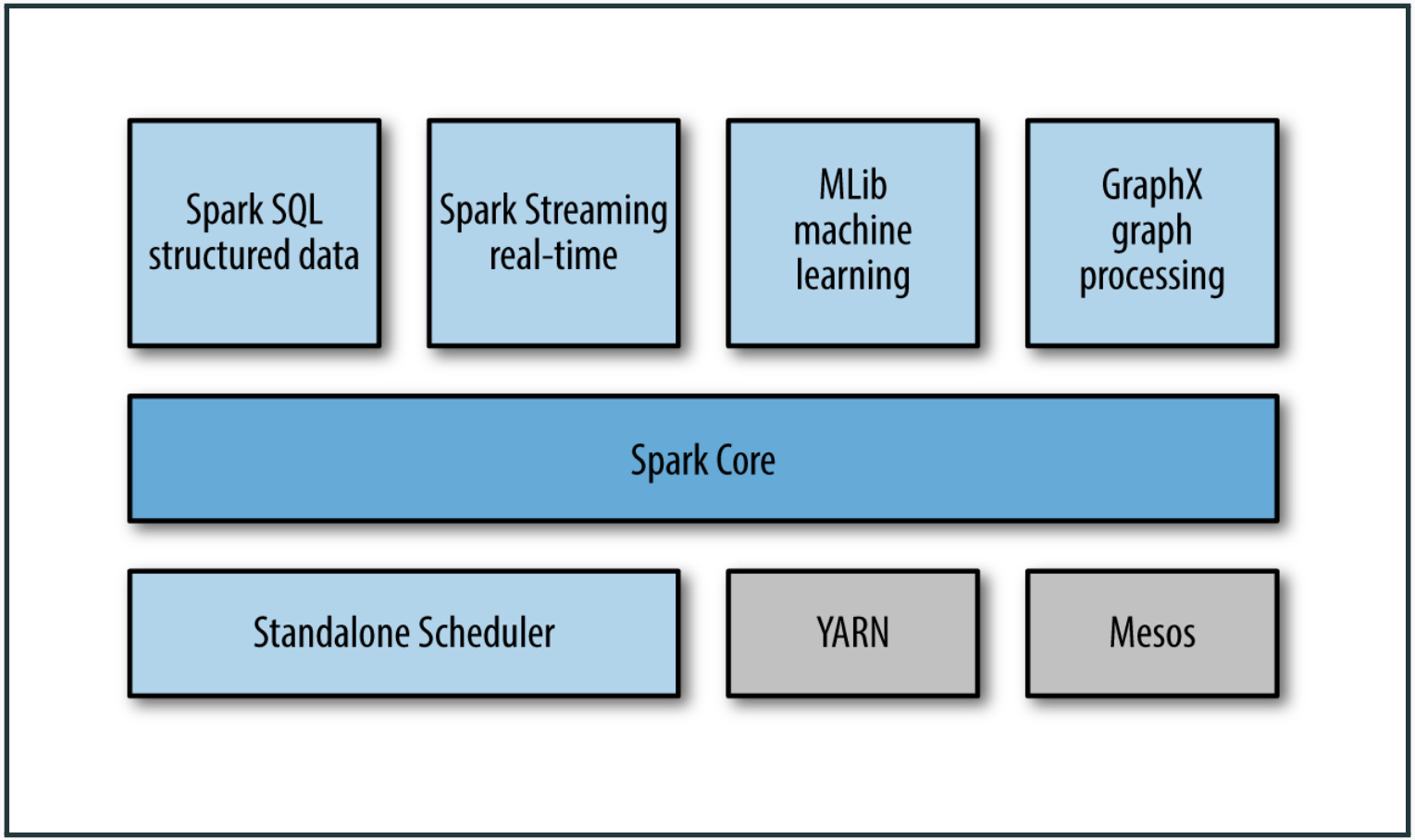
MapReduce



Spark



Architettura di Spark - 1



Architettura di Spark - 2

Spark è costituito da molteplici componenti strettamente integrati, che sono specializzati per diversi carichi di lavoro, come ad esempio query SQL o algoritmi di machine learning.

Spark Core

- Contiene le funzionalità di base di Spark, inclusi i componenti per la pianificazione dei task, la gestione della memoria, il ripristino dei guasti, l'interazione con i sistemi di storage e altro ancora.
- Fornisce le API per gestire i Resilient Distributed Dataset (RDD), che costituiscono la principale astrazione di programmazione di Spark.
- Gli RDD rappresentano un insieme di elementi distribuiti su molti nodi del cluster, che possono essere manipolati in parallelo.

Architettura di Spark - 3

Spark SQL

- È il pacchetto di Spark per lavorare con dati strutturati.
- Consente di interrogare i dati tramite SQL e la variante Apache Hive di SQL, chiamata Hive Query Language (HQL).
- Offre la possibilità di combinare le query SQL con le manipolazioni dei dati supportate dagli RDD in Python, Java e Scala, il tutto in un'unica applicazione, combinando così SQL con analisi complesse.

Spark Streaming

- Consente l'elaborazione di flussi di dati in tempo reale (ad esempio file di log generati dai server Web).
- Riceve flussi di dati in tempo reale e li divide in batch, che vengono poi elaborati dal motore Spark per generare il flusso finale dei risultati.

MLlib

- Fornisce diversi tipi di algoritmi di machine learning, tra cui la classificazione, la regressione, il clustering e il filtraggio collaborativo.
- Tutti questi algoritmi sono progettati per essere scalalabili all'interno di un cluster.

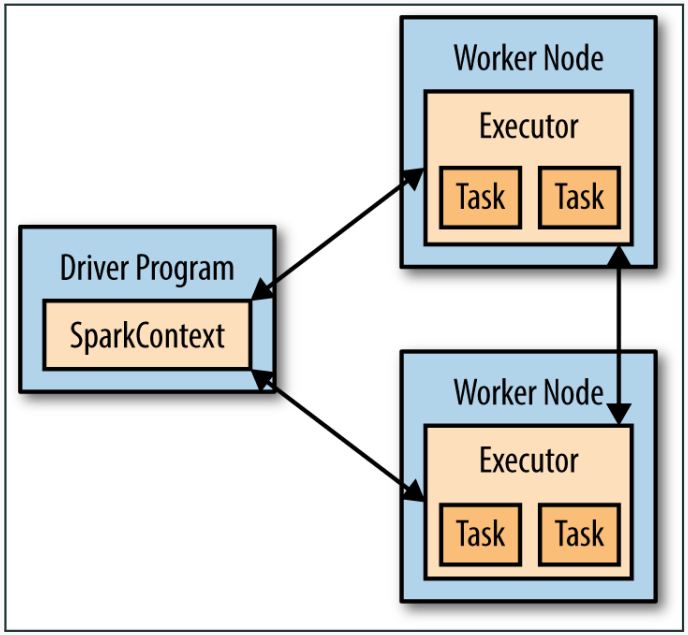
GraphX

- È una libreria per la manipolazione di grafi (ad esempio, grafo delle amicizie in un social network).
- Estende le API degli RDD, offrendo vari operatori per la manipolazione di grafi in modo parallelo e una raccolta degli algoritmi più comuni sui grafi.

Esecuzione di un programma Spark - 1

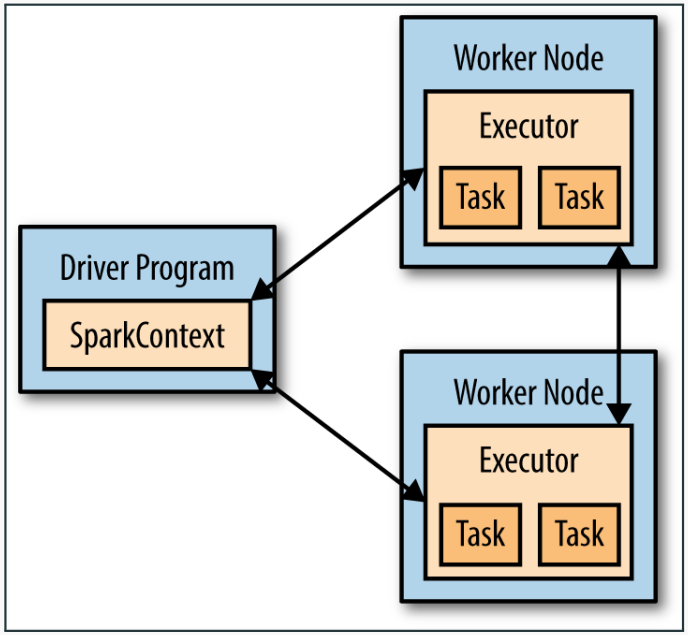
Spark può essere eseguito in locale o in modalità cluster.

- Le applicazioni Spark vengono eseguite come insiemi indipendenti di processi su un cluster, coordinati dall'oggetto SparkContext nel programma principale (chiamato programma driver).
- Lo SparkContext può connettersi a diversi tipi di cluster manager (YARN, Mesos o il proprio cluster manager standalone), che allocano le risorse tra le varie applicazioni.



Esecuzione di un programma Spark - 2

- Una volta connesso, Spark acquisisce esecutori (executors) sui nodi del cluster, che sono processi che eseguono calcoli e memorizzano dati per l'applicazione.
- Successivamente, SparkContext invia il codice dell'applicazione agli esecutori e, infine, assegna agli esecutori i task da svolgere.



Installazione e setup

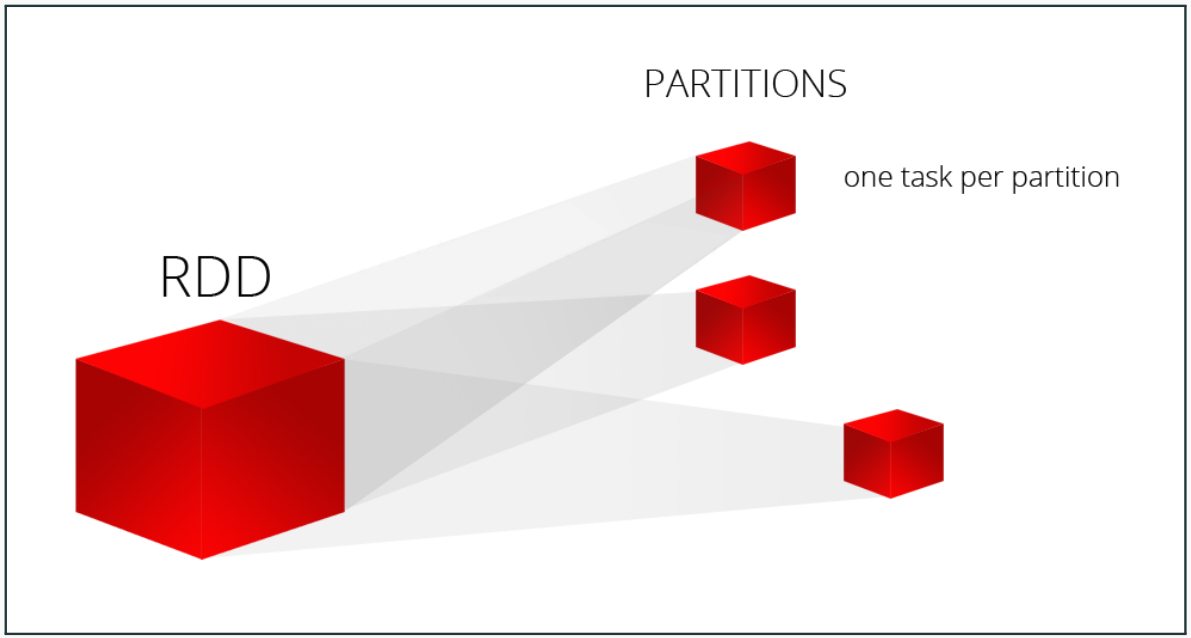
RDD

RDD - Resilient Distributed Dataset

- Il concetto alla base di una elaborazione Spark è l'RDD (Resilient Distributed Dataset), una collezione distribuita di oggetti immutabili che, durante l'esecuzione, risiedono tipicamente nelle memorie principali dei nodi del cluster.
- Gli RDD possono essere manipolati mediante diversi operatori che agiscono in parallelo e sono resistenti ai fallimenti.
- La fault tolerance è garantita da un meccanismo di replicazione e, quando necessario, di ricostruzione automatica.

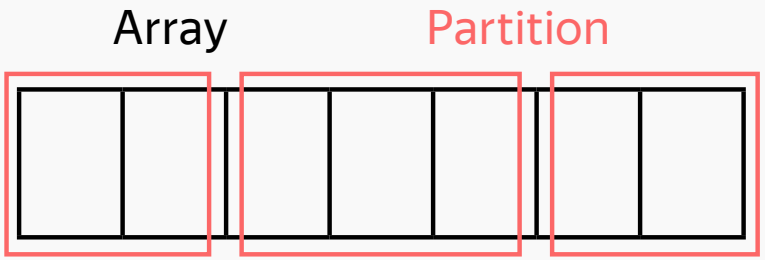
RDD - Resilient Distributed Dataset

- **Resilient:** la fault tolerance degli RDD è garantita automaticamente da Spark.
- **Distributed:** gli RDD sono divisi in partizioni, e ciascuna partizione può risiedere su un nodo diverso.
- **Dataset:** gli RDD rappresentano collezioni di dati.



RDD - Partizioni

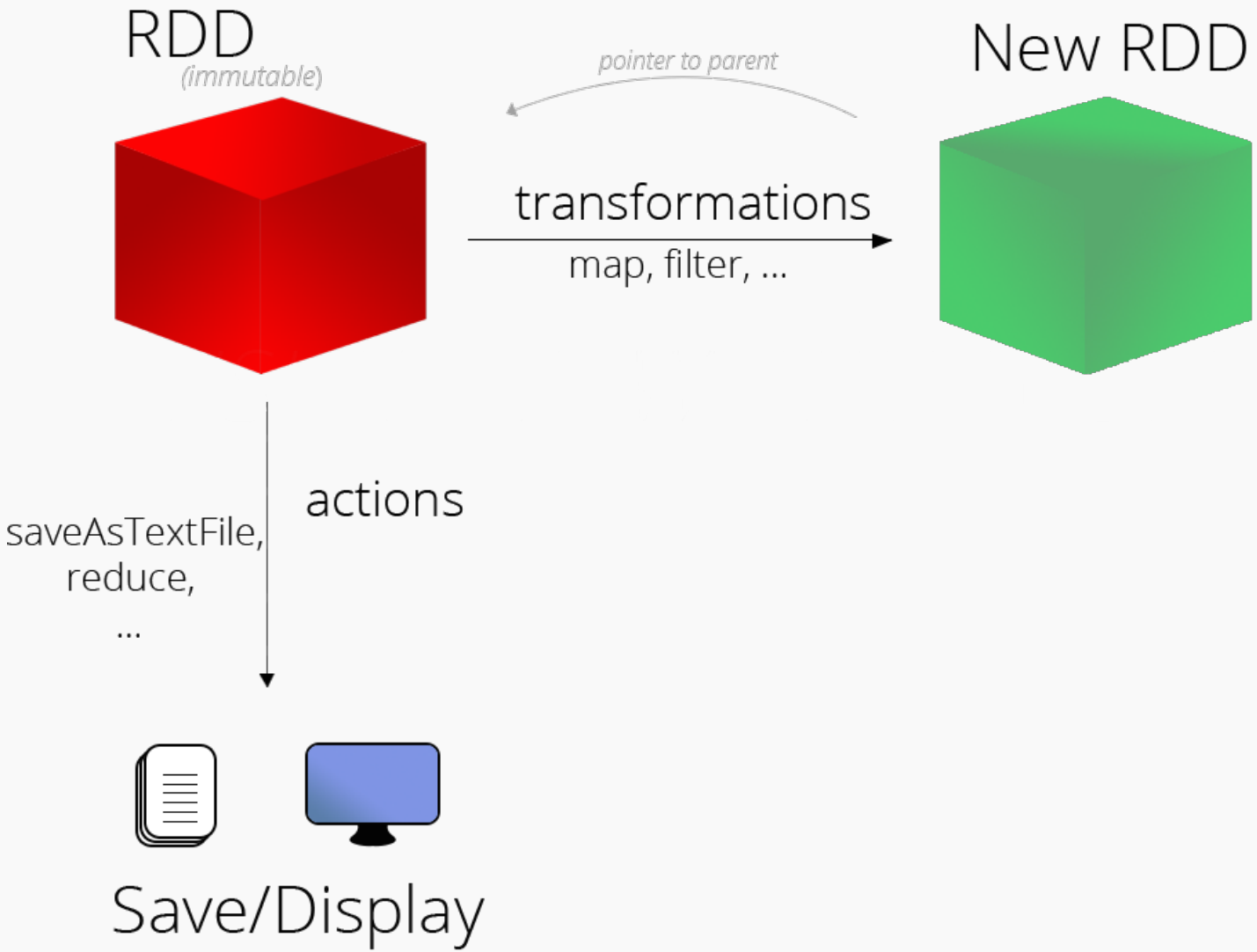
- Ogni set di dati in RDD è diviso in partizioni logiche, che possono essere calcolate su diversi nodi del cluster.
- Gli RDD possono contenere qualsiasi tipo di oggetto Python, Java o Scala, incluse classi definite dall'utente.
- Ad esempio, un semplice array può essere diviso in partizioni e utilizzato da Spark come RDD.



Ciclo di vita di un RDD - Creazione, trasformazioni, azioni

- Si può creare un RDD partizionando (e, quindi, parallelizzando) una collezione dati esistente nel programma del driver (ad esempio una lista in Python).
- In alternativa, si può caricare un set di dati in un sistema di archiviazione esterno, ad esempio un file system condiviso o HDFS.
- Infine, un RDD può essere costruito a partire da un altro RDD, applicando una o più trasformazioni, come `map` o `filter`.
- Dopo avere applicato le trasformazioni necessarie, è possibile eseguire delle azioni sugli RDD (ad esempio salvarli su disco, o visualizzarli a schermo).

Trasformazioni e azioni



Creare un RDD - parallelize

- È possibile creare un RDD a partire da una collezione di dati (lista, set, ecc.) utilizzando il metodo `parallelize` dell'oggetto `sc`:

```
values = range(10)
rdd = sc.parallelize(values)

rdd.getNumPartitions()  # ad esempio, 8
rdd.collect()           # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
rdd.glom().collect()    # [[0], [1], [2], [3, 4], [5], [6], [7], [8, 9]]
```

- `collect` è un'azione che restituisce una lista contenente tutti gli elementi di un RDD.

- `glom` è una trasformazione che raggruppa tutti gli elementi all'interno di ciascuna partizione in una lista.

- È possibile specificare il numero di partizioni da creare, quando si usa il metodo `parallelize`:

```
values = [1, 2, 4, 7, 8, 3, 6, 7]
rdd = sc.parallelize(values, 5)
rdd.getNumPartitions()  # 5
rdd.glom().collect()    # [[1], [2, 4], [7], [8, 3], [6, 7]]
```

Creare un RDD - textFile

- Il metodo `textFile` dell'oggetto `sc` legge un file di testo (o tutti i file presenti in una directory) dal file system locale o da HDFS:

```
rdd = sc.textFile('temperature_input/')

rdd.take(5)    # restituisce i primi 5 elementi dell'RDD

# ['2008-01-01,00:00:00,13.7',
#  '2008-01-01,01:00:00,13.0',
#  '2008-01-01,02:00:00,13.0',
#  '2008-01-01,03:00:00,13.0',
#  '2008-01-01,04:00:00,13.5']
```

- Nella VM Cloudera, il file system di default per Spark è HDFS, quindi la directory dell'esempio precedente viene cercata su HDFS.
  - Per indicare un percorso (assoluto) sul file system locale, si deve specificare il protocollo `file://`

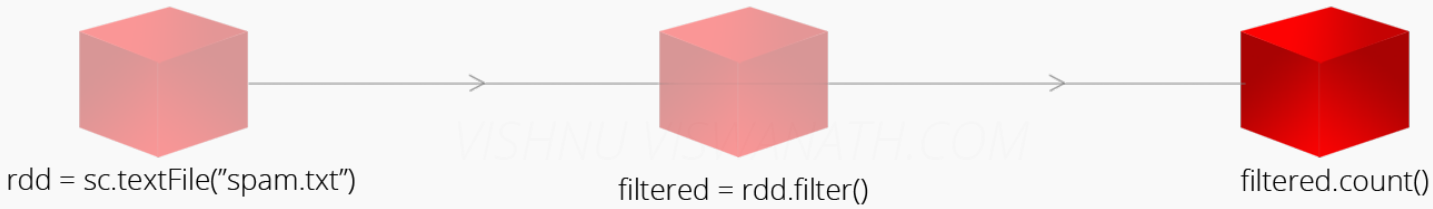
```
rdd = sc.textFile('file:///home/cloudera/temperature_input/')
```

- Se HDFS non è stato configurato, il percorso (relativo o assoluto) viene cercato sul file system locale.

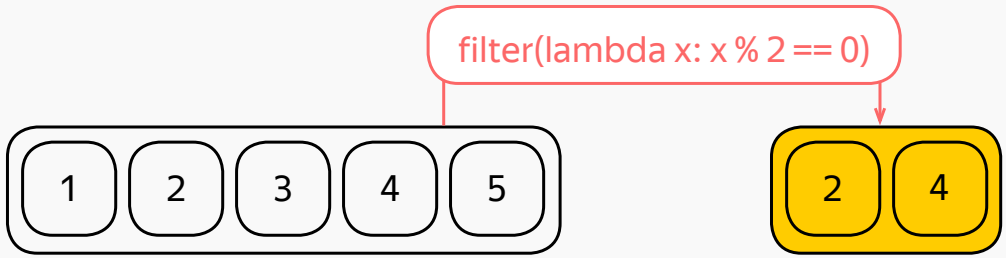
Trasformazioni

Lazy evaluation

- Le trasformazioni non vengono eseguite immediatamente. Al contrario, la loro esecuzione è posticipata fino a quando non viene effettuata un'azione (lazy evaluation).
- Spark tiene traccia della sequenza di trasformazioni da applicare all'RDD iniziale (lineage, o discendenza), compreso da dove leggere i dati.
- Nell'esempio illustrato in figura, le operazioni `textFile` e `filter` non sono eseguite immediatamente, ma solo dopo che viene richiesta l'azione `count`.



FILTER



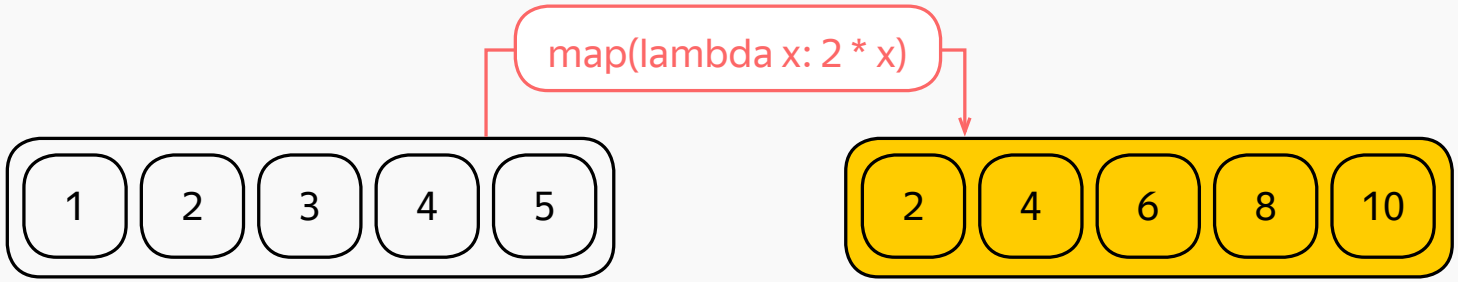
- Dato un RDD e una funzione `func`, crea un nuovo RDD costituito solo dagli elementi per i quali `func` restituisce `true`.

```
def is_even(x):
    return x % 2 == 0

rdd1 = sc.parallelize([1, 2, 3, 4, 5])
rdd2 = rdd1.filter(is_even)

rdd1.collect()    # [1, 2, 3, 4, 5]
rdd2.collect()    # [2, 4]
```

- Quando la funzione è molto semplice, è possibile utilizzare la sintassi più concisa delle `lambda` (funzioni usa e getta):

```
rdd2 = rdd1.filter(lambda x: x % 2 == 0)
```

MAP



- Dato un RDD e una funzione `func`, restituisce un nuovo RDD ottenuto applicando `func` ad ogni elemento dell'RDD originale.

```
rdd1 = sc.parallelize([1, 2, 3, 4, 5])
rdd2 = rdd1.map(lambda x: x * 2)

rdd1.collect()    # [1, 2, 3, 4, 5]
rdd2.collect()    # [2, 4, 6, 8, 10]
```

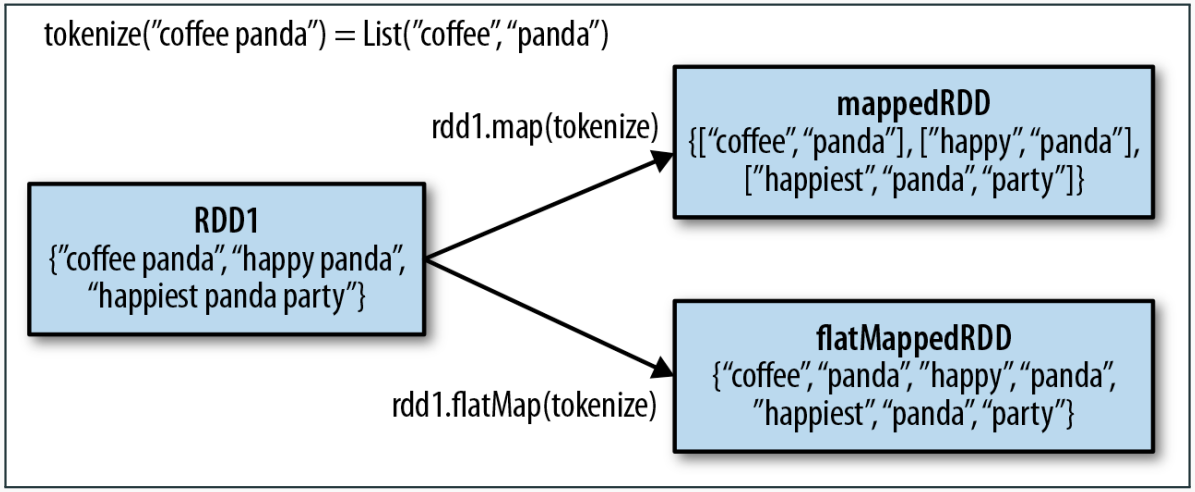
- A differenza dei mapper in MapReduce, il metodo `map` restituisce un RDD che ha **sempre** lo stesso numero di elementi dell'RDD originale (come la funzione `map` della programmazione funzionale).

FLATMAP

- Come `map`, ma appiattisce il risultato finale.

```
words = sc.parallelize(['uno due', 'tre quattro', 'cinque sei sette otto'])

words.map(lambda x: x.split()).collect()
# [['uno', 'due'], ['tre', 'quattro'], ['cinque', 'sei', 'sette', 'otto']]

words.flatMap(lambda x: x.split()).collect()
# ['uno', 'due', 'tre', 'quattro', 'cinque', 'sei', 'sette', 'otto']
```

- Può essere utilizzata per restituire un numero di elementi diverso rispetto all'RDD di partenza e, quindi, generalizza sia `map` che `filter`.



Parsing delle temperature - map + filter

```
def map_temp(line):
    date, time, temp = line.split(',')
    year, month, day = date.split('-')

    return int(year), float(temp)

def filter_temp(key_value):
    year, temp = key_value

    return temp <= 50.0


rdd1 = sc.textFile('temperature_input/')

rdd2 = rdd1.map(map_temp).filter(filter_temp)

rdd2.take(5)
# [(2008, 13.7), (2008, 13.0), (2008, 13.0), (2008, 13.0), (2008, 13.5)]
```

Parsing delle temperature - flatMap

```
def flatmap_temp(line):
    date, time, temp = line.split(',')
    year, month, day = date.split('-')

    result = []
    if float(temp) <= 50.0:
        value = int(year), float(temp)
        result.append(value)

    return result


rdd1 = sc.textFile('temperature_input/')

rdd2 = rdd1.flatMap(flatmap_temp)

rdd2.take(5)
# [(2008, 13.7), (2008, 13.0), (2008, 13.0), (2008, 13.0), (2008, 13.5)]
```

Trasformazioni di tipo insiemistico

- Gli RDD supportano le classiche operazioni tra set, come unione e intersezione:

```
r1 = sc.parallelize([1, 2, 3, 4, 5])
r2 = sc.parallelize([3, 1, 9, 2, 1, 2, 14])

r1.union(r2).collect()         # [1, 2, 3, 4, 5, 3, 1, 9, 2, 1, 2, 14]
r1.intersection(r2).collect()  # [1, 2, 3]
r1.subtract(r2).collect()      # [4, 5]
r2.distinct().collect()        # [1, 9, 2, 3, 14]
```

- `union` effettua l'unione (senza eliminare eventuali duplicati).
- `intersection` effettua l'intersezione.
- `subtract` effettua la sottrazione del secondo RDD dal primo.
- `distinct` elimina i duplicati.

Trasformazioni su Pair RDD

Pair RDD

- In molti casi è comodo considerare coppie chiave-valore.
- Gli RDD costituiti da coppie di questo tipo sono chiamati Pair RDD.
- In Python, un Pair RDD è formato da tuple `(chiave, valore)`.
- Alcune trasformazioni e azioni hanno senso solo se applicate a Pair RDD.
- A parte questo, in Python non ci sono differenze sostanziali tra i normali RDD e i Pair RDD.

```
key_values = sc.parallelize([('a', 7), ('b', 4), ('a', 1), ('b', 6), ('c', 3)])
key_values.collect()    # [('a', 7), ('b', 4), ('a', 1), ('b', 6), ('c', 3)]
```

KEYS E VALUES

- `keys` restituisce un nuovo RDD contenente solo le chiavi di un Pair RDD.
- `values` restituisce un nuovo RDD contenente solo i valori di un Pair RDD.

```
rdd = sc.parallelize([('a', 7), ('b', 4), ('a', 1), ('b', 6), ('c', 3)])
keys = rdd.keys()
values = rdd.values()

keys.collect()      # ['a', 'b', 'a', 'b', 'c']
values.collect()    # [7, 4, 1, 6, 3]
```

groupByKey

- Raggruppa tutti i valori con la stessa chiave e restituisce un nuovo RDD.

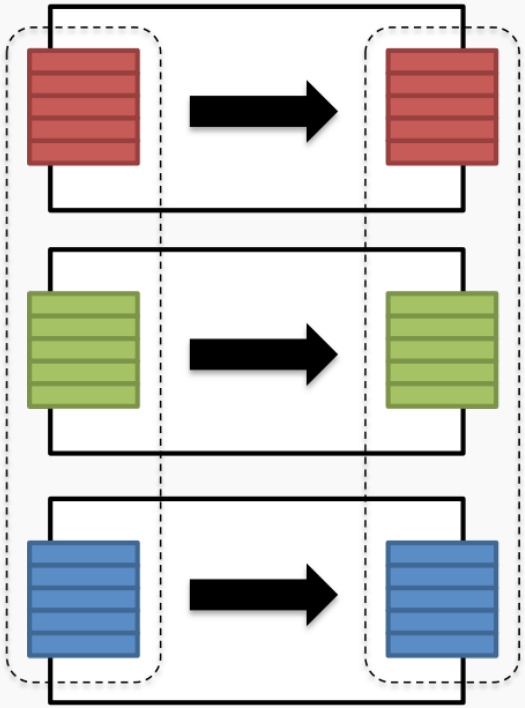
```
rdd = sc.textFile('temperature_input/')
parsed_temp = rdd.map(map_temp).filter(filter_temp)

grouped = parsed_temp.groupByKey()
grouped.take(3)
# [(2010, <pyspark.resultiterable.ResultIterable at 0x113db2bd0>),
#  (2011, <pyspark.resultiterable.ResultIterable at 0x113dabc50>),
#  (2012, <pyspark.resultiterable.ResultIterable at 0x113dab110>)]
```

- Il valore associato a ciascuna chiave è un oggetto iterabile (ad esempio in un ciclo for) che contiene tutti i valori associati a quella chiave.

Trasformazioni narrow e wide

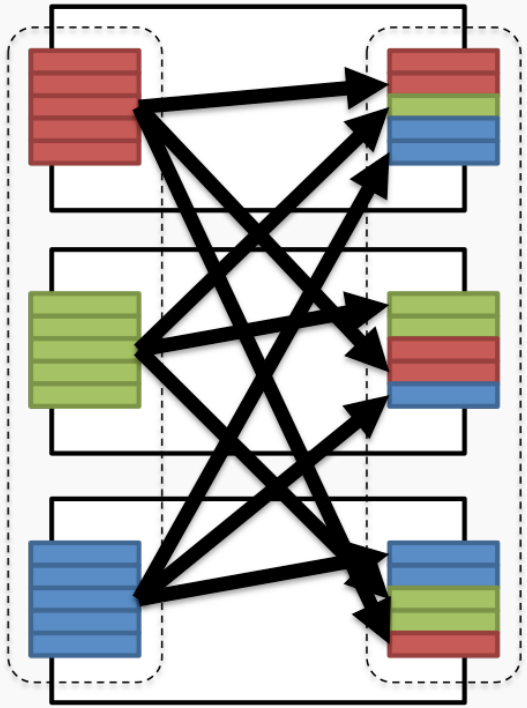
Narrow transformation

- Input and output stays in same partition
- No data movement is needed



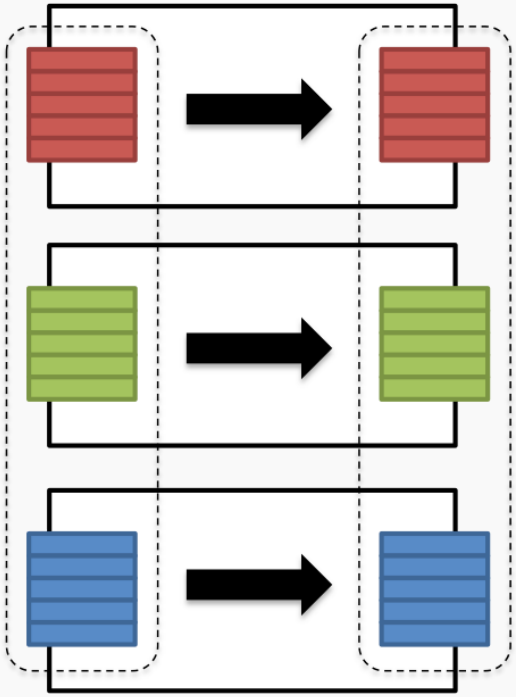
Wide transformation

- Input from other partitions are required
- Data shuffling is needed before processing

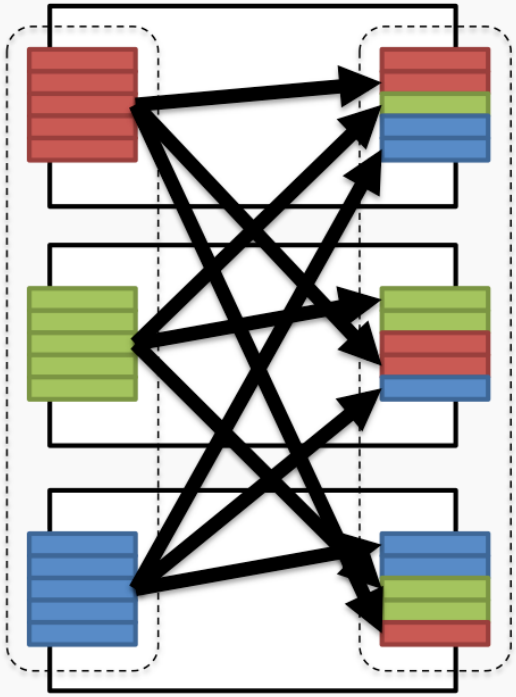


- Alcune trasformazioni sono semplici (ed efficienti) da effettuare, mentre altre richiedono una fase di shuffle, che può rallentare l'intero job.

Trasformazioni narrow e wide

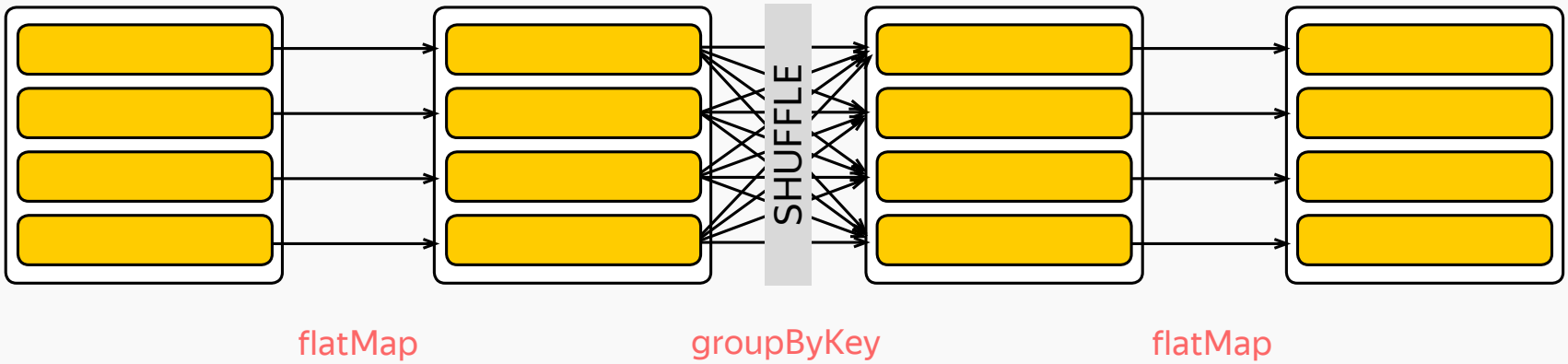


- `map`, `flatMap` e `filter` sono trasformazioni di tipo narrow.



- `groupByKey` è una trasformazione di tipo wide.

Workflow di MapReduce in Spark

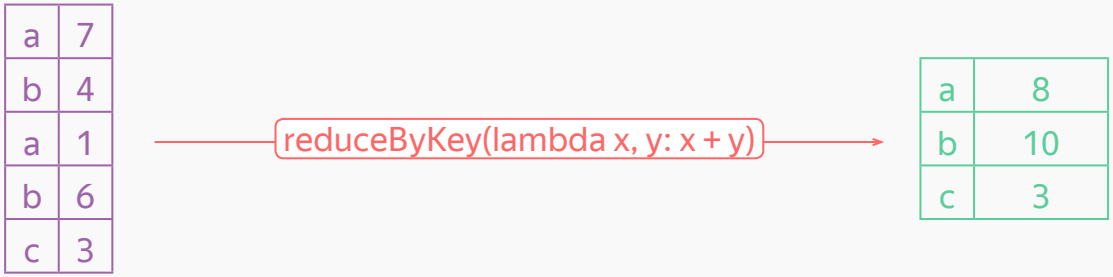


- Possiamo emulare il workflow di MapReduce (mapper, shuffle and sort, reducer) utilizzando tre trasformazioni in sequenza:
  - il compito del mapper è svolto da una `flatMap`;
  - il compito della fase shuffle and sort è svolto da una `groupByKey`;
  - il compito del reducer è svolto da un'altra `flatMap`.
- In Spark esistono altre trasformazioni più specifiche che permettono di ottenere lo stesso risultato in modo più semplice ed efficiente.

Temperatura massima - flatMap + groupByKey + flatMap

```
def mapper_temp(line):
    date, time, temp = line.split(',')
    year, month, day = date.split('-')

    result = []
    if float(temp) <= 50.0:
        value = int(year), float(temp)
        result.append(value)

    return result


def reducer_temp(key_val):
    year, temp_list = key_val
    max_temp = float('-inf')

    for temp in temp_list:
        max_temp = max(temp, max_temp)

    return [(year, max_temp)]


rdd = sc.textFile('temperature_input/')
result = rdd.flatMap(mapper_temp).groupByKey().flatMap(reducer_temp)

result.collect()
```

reduceByKey



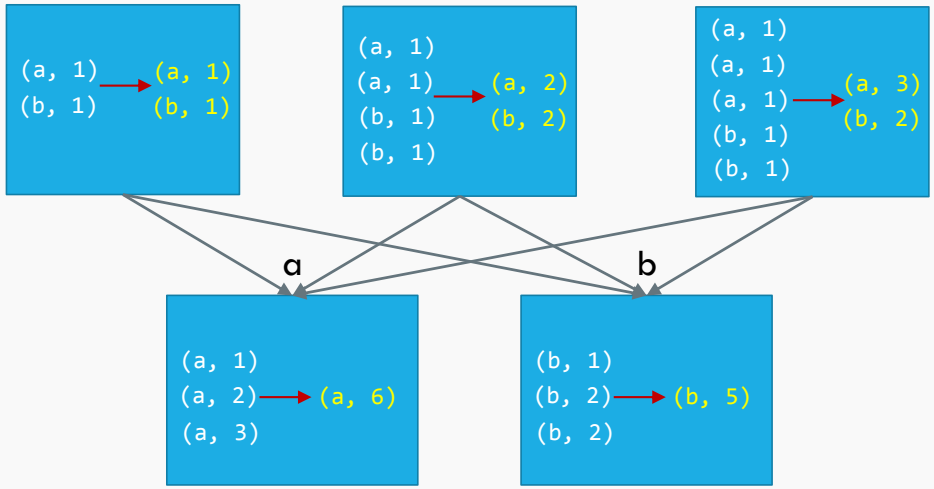
- Simile all'operazione `reduce` della programmazione funzionale, ma applica la funzione di riduzione per ogni chiave di un Pair RDD.
- La funzione passata come argomento ha il compito di ridurre tutti gli elementi con la stessa chiave a un solo valore.

```
key_values = sc.parallelize([('a', 7), ('b', 4), ('a', 1), ('b', 6), ('c', 3)])

reduce_sum = key_values.reduceByKey(lambda x, y: x + y)
reduce_sum.collect()    # [('a', 8), ('c', 3), ('b', 10)]

reduce_max = key_values.reduceByKey(max)
reduce_max.collect()    # [('a', 7), ('c', 3), ('b', 6)]
```

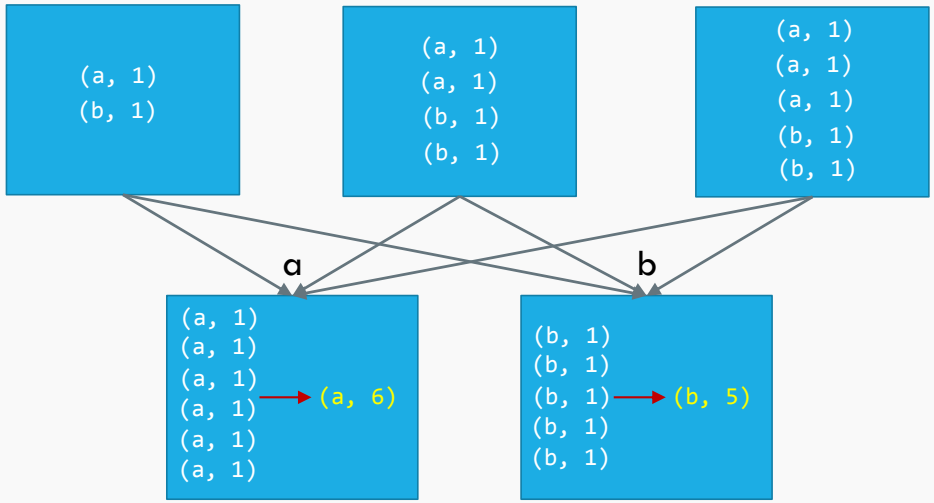
- Più efficiente di `groupByKey` (Spark può utilizzare dei combiner automaticamente).

Differenze tra reduceByKey e groupByKey

reduceByKey()



groupByKey()



Temperatura massima - map + filter + reduceByKey

```
def parse_temperatures(line):
    date, time, temp = line.split(',')
    year, month, day = date.split('-')

    return int(year), float(temp)


def filter_temperatures(key_value):
    year, temp = key_value

    return temp <= 50.0


temp_rdd = sc.textFile('temperature_input/')

parsed_temp = temp_rdd.map(parse_temperatures)
filtered_temp = parsed_temp.filter(filter_temperatures)
max_year_temp = filtered_temp.reduceByKey(max)

max_year_temp.collect()    # [(2010, 40.7), (2011, 35.0), ...]


# In alternativa, espressione unica
# temp_rdd.map(parse_temperatures).filter(filter_temperatures).reduceByKey(max)
```

mapValues e flatMapValues

- `mapValues` e `flatMapValues` funzionano come `map` e `flatMap`, rispettivamente, ma operano solo sui valori.
- Utili quando è necessario applicare una `map` (o `flatMap`) su un Pair RDD, ma si vogliono modificare solo i valori, lasciando invariate le chiavi.
- Più efficienti di `map` e `flatMap`, perché consentono a Spark di ottimizzare meglio la computazione (il partizionamento rimane invariato).

```
key_values = sc.parallelize([('a', 7), ('b', 4), ('a', 1), ('b', 6), ('c', 3)])

# Calcola il quadrato dei valori, senza modificare le chiavi
values_squared = key_values.mapValues(lambda x: x * x)

values_squared.collect()
# [('a', 49), ('b', 16), ('a', 1), ('b', 36), ('c', 9)]
```

Calcolare la media per ogni chiave

- Per calcolare la media dei valori associati ad ogni chiave, non basta utilizzare `reduceByKey`, perché siamo interessati a due cose (per ogni chiave): la somma dei valori e il numero dei valori.
- Una volta calcolati somma e numero dei valori, la media sarà semplicemente `sum_values / num_values`.

- Possiamo utilizzare `groupByKey` e operare in modo simile a quanto visto con MapReduce.
- `groupByKey` è però meno efficiente di `reduceByKey`.
- Combinando `mapValues` e `reduceByKey` possiamo ottenere lo stesso risultato.

Calcolare la media per ogni chiave - groupByKey

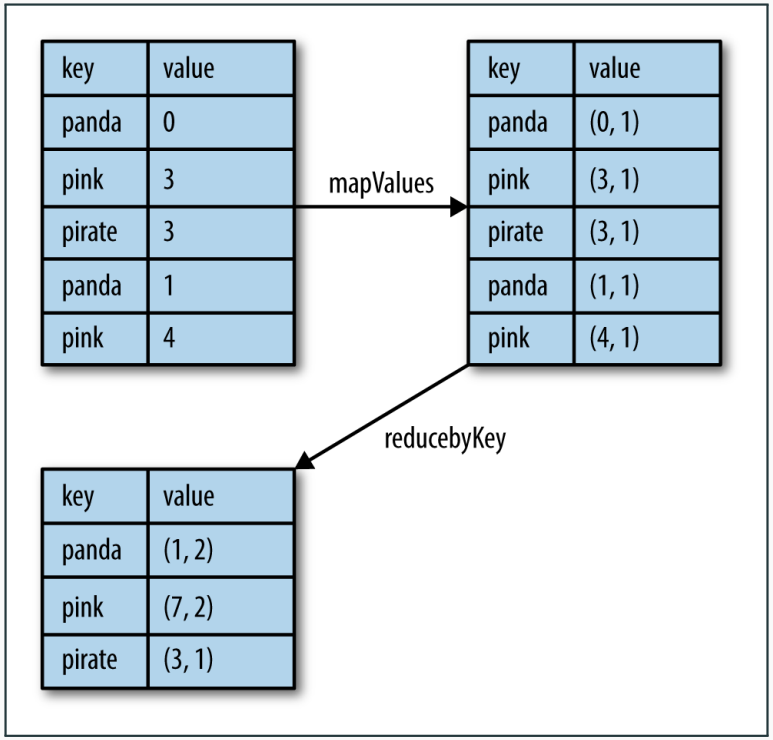
```
def calc_avg_temp(temp_list):
    sum_values = 0
    num_values = len(temp_list)

    for temp in temp_list:
        sum_values += temp

    return sum_values / num_values


# Supponiamo di avere a disposizione le temperature già parsate (v. slide precedenti)
parsed_temp.take(3)     # [(2008, 13.7), (2008, 13.0), (2008, 13.0)]

avg_temp = parsed_temp.groupByKey().mapValues(calc_avg_temp)
avg_temp.collect()
```

Calcolare la media per ogni chiave - mapValues + reduceByKey

1. Utilizziamo `mapValues` per associare un contatore (1) a ciascuna coppia chiave-valore, come nel word count.

   `Input: (pirate, 3).` `Output: (pirate, (3, 1))`

2. Utilizziamo `reduceByKey` per sommare sia i valori sia i contatori.
3. Utilizziamo `mapValues` per calcolare la media come `sum_values / num_values`



Calcolare la media per ogni chiave - mapValues + reduceByKey

```
def sum_values_and_counters(x, y):
    value1, counter1 = x
    value2, counter2 = y

    return value1 + value2, counter1 + counter2


def calc_avg(sum_counter):
    values_sum, counter = sum_counter
    return values_sum / counter


# Supponiamo di avere a disposizione le temperature già parsate (v. slide precedenti)
parsed_temp.take(3)     # [(2008, 13.7), (2008, 13.0), (2008, 13.0)]

temp_counters = parsed_temp.mapValues(lambda x: (x, 1))
sum_counters = temp_counters.reduceByKey(sum_values_and_counters)

avg_temp = sum_counters.mapValues(calc_avg)
avg_temp.collect()
```

Ordinamento in base alla chiave - sortByKey

- `sortByKey` ordina un Pair RDD in base alla chiave.

```
rdd = sc.parallelize([('a', 7), ('b', 4), ('a', 1), ('b', 6), ('c', 3)])
rdd.sortByKey().collect()
# [('a', 7), ('a', 1), ('b', 4), ('b', 6), ('c', 3)]
```

- Come parametro opzionale, è possibile specificare se l'ordinamento deve essere ascendente o discendente (`ascending`).

```
rdd.sortByKey(ascending=False).collect()
# [('c', 3), ('b', 4), ('b', 6), ('a', 7), ('a', 1)]
```

- È anche possibile specificare una funzione da applicare a ciascun elemento prima di effettuare i confronti (`keyfunc`).

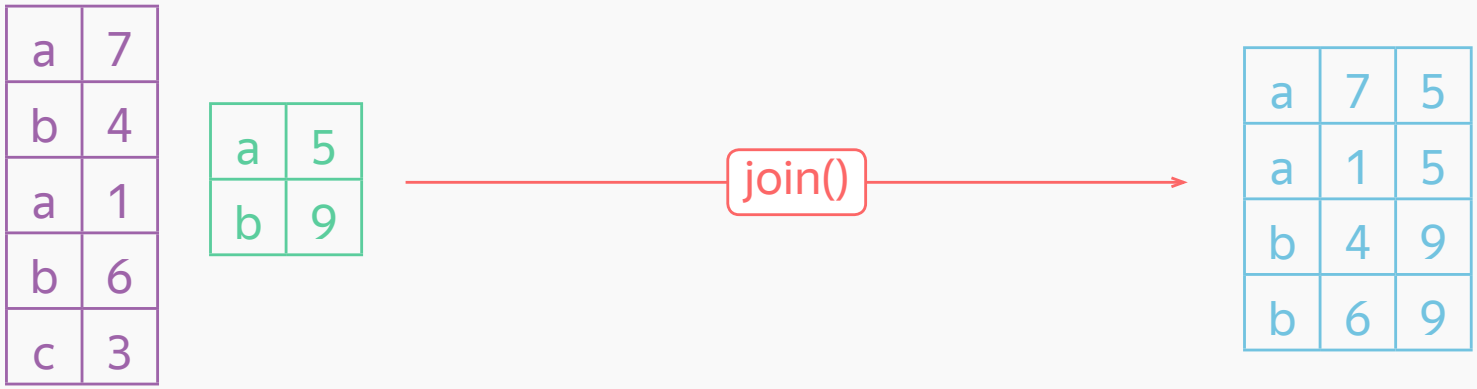
```
rdd = sc.parallelize([('a', 7), ('b', 4), ('A', 1), ('B', 6), ('C', 3)])
rdd.sortByKey().collect()
# [('A', 1), ('B', 6), ('C', 3), ('a', 7), ('b', 4)]

rdd.sortByKey(keyfunc=lambda x: x.lower()).collect()
# [('a', 7), ('A', 1), ('b', 4), ('B', 6), ('C', 3)]
```

Ordinamento generico - sortBy

- Ordina in base a una funzione passata come parametro.
- Più generale di `sortByKey`, può essere usata quando si vuole ordinare in base al valore, o quando non si ha a che fare con un Pair RDD.
- Come nel caso di `sortByKey`, i parametri da specificare sono `keyfunc` (richiesto) e `ascending` (opzionale).

```
rdd = sc.parallelize([('a', 7), ('b', 4), ('A', 1), ('B', 6), ('C', 3)])

rdd.sortBy(ascending=False, keyfunc=lambda key_value: key_value[1]).collect()
# [('a', 7), ('B', 6), ('b', 4), ('C', 3), ('A', 1)]
```

Join



- Dati due Pair RDD, `join` restituisce un nuovo RDD contenente tutte le coppie di elementi con chiavi corrispondenti.

```
rdd1 = sc.parallelize([('a', 7), ('b', 4), ('a', 1), ('b', 6), ('c', 3)])
rdd2 = sc.parallelize([('a', 5), ('b', 9)])

result = rdd1.join(rdd2)
result.collect()
# [('a', (7, 5)), ('a', (1, 5)), ('b', (4, 9)), ('b', (6, 9))]
```

- `join` considera solo chiavi presenti in entrambi gli RDD.

- Altri operatori, come `leftOuterJoin`, `rightOuterJoin` e `fullOuterJoin` considerano anche le chiavi presenti solo in uno dei due RDD.

leftOuterJoin, rightOuterJoin, fullOuterJoin

```
rdd1 = sc.parallelize([('a', 7), ('b', 4), ('a', 1), ('b', 6), ('c', 3)])
rdd2 = sc.parallelize([('a', 5), ('b', 9)])

# leftOuterJoin considera anche le chiavi presenti solo nel primo RDD (c)
rdd1.leftOuterJoin(rdd2).collect()
# [('a', (7, 5)), ('a', (1, 5)), ('c', (3, None)), ('b', (4, 9)), ('b', (6, 9))]

# rightOuterJoin considera anche le chiavi presenti solo nel secondo RDD (z)
rdd1.rightOuterJoin(rdd2).collect()
# [('a', (7, 5)), ('a', (1, 5)), ('b', (4, 9)), ('b', (6, 9)), ('z', (None, 4))]

# fullOuterJoin considera anche le chiavi presenti solo in uno dei due RDD (c, z)
rdd1.fullOuterJoin(rdd2).collect()
# [('a', (7, 5)),
#  ('a', (1, 5)),
#  ('c', (3, None)),
#  ('b', (4, 9)),
#  ('b', (6, 9)),
#  ('z', (None, 4))]
```

Azioni

Azioni applicabili a tutti gli RDD

- `collect()` restituisce una lista contenente tutti gli elementi di un RDD. Da utilizzare con attenzione se l'RDD è composto da molti elementi.
- `take(n)` restituisce una lista contenente i primi n elementi di un RDD.
- `saveAsTextFile(path)` salva l'RDD su file.

Azioni applicabili a Pair RDD

- `collectAsMap()` è simile a `collect`, ma può essere applicato solo a Pair RDD in cui ogni chiave compare al massimo una volta, e restituisce un dizionario invece di una lista di tuple.

```
rdd = sc.parallelize([('a', 5), ('b', 9), ('z', 4)])
rdd.collectAsMap()    # {'a': 5, 'b': 9, 'z': 4}
```

- `lookup(key)` restituisce una lista contenente i valori associati ad una particolare chiave in un Pair RDD.

```
rdd = sc.parallelize([('a', 7), ('b', 4), ('a', 1), ('b', 6), ('c', 3)])
rdd.lookup('a')       # [7, 1]
```

Azioni di aggregazione - min, max, mean

- `min()` restituisce il valore minimo contenuto in un RDD.
- `max()` restituisce il valore massimo contenuto in un RDD.

```
rdd = sc.parallelize([5, 8, 1, 6, 14, 9, 3])
rdd.min()    # 1
rdd.max()    # 14
```

- `mean()` restituisce il valore medio contenuto in un RDD.
- `min` e `max` consentono di indicare, in modo opzionale, una funzione da applicare a ciascun elemento prima di effettuare i confronti (`key`).

```
people = [{'name': 'Mario', 'age': 24},
          {'name': 'Alice', 'age': 21},
          {'name': 'Carlo', 'age': 28}]
rdd = sc.parallelize(people)

rdd.max(key=lambda x: x['age'])    # {'age': 28, 'name': 'Carlo'}
```

Contare gli elementi - count, countByValue e countByKey

- `count()` restituisce il numero di elementi contenuti in un RDD.
- `countByValue()` conta le occorrenze di ogni valore presente in un RDD.

```
rdd1 = sc.parallelize(['a', 'a', 'c', 'a', 'd', 'b', 'a', 'c'])
rdd1.countByValue()    # {'a': 4, 'b': 1, 'c': 2, 'd': 1}
```

- `countByKey()` conta le occorrenze di ogni chiave presente in un RDD.

```
rdd2 = sc.parallelize([('a', 7), ('b', 4), ('a', 1), ('b', 6), ('c', 3)])
rdd2.countByKey()      # {'a': 2, 'b': 2, 'c': 1}
```

Azioni con ordinamento - top e takeOrdered

- `top(n)` restituisce una lista contenente i primi `n` elementi di un RDD, ordinati in modo decrescente.
- `takeOrdered(n)` restituisce una lista contenente i primi `n` elementi di un RDD, ordinati in modo crescente.
- Entrambi i metodi consentono di indicare, in modo opzionale, una funzione da applicare a ciascun elemento prima di effettuare i confronti (`key`).

```
# Supponiamo che parsed_temp sia un Pair RDD contenente coppie chiave-valore
# di tipo anno-temperatura
parsed_temp.take(5)
# [(2008, 13.7), (2008, 13.0), (2008, 13.0), (2008, 13.0), (2008, 13.5)]

# Otteniamo le 5 temperature più basse
parsed_temp.takeOrdered(5, key=lambda year_temp: year_temp[1])
# [(2014, -1.5), (2014, 0.0), (2014, 0.0), (2014, 0.0), (2014, 0.0)]

# Otteniamo le 5 temperature più alte
parsed_temp.top(5, key=lambda year_temp: year_temp[1])
# [(2016, 41.0), (2010, 40.7), (2016, 40.0), (2016, 40.0), (2016, 40.0)]
```

Trasformazioni avanzate

aggregateByKey

- `aggregateByKey` è un metodo più generale, rispetto a `reduceByKey`.
- Per utilizzare `aggregateByKey` è necessario specificare tre parametri:
  1. un valore iniziale;
  2. una funzione per combinare, all'interno di ogni partizione, i valori corrispondenti a una particolare chiave;
  3. una funzione per combinare i risultati parziali provenienti da partizioni diverse.
- Esempio: calcolo della temperatura media, per ogni anno.
- L'obiettivo, come nel caso dell'utilizzo di `mapValues` + `reduceByKey`, è quello di ottenere delle tuple `(year, (sum_values, count_values))`, che consentiranno poi di calcolare facilmente le medie, anno per anno.
- In questo caso, vogliamo calcolare prima le tuple all'interno di una singola partizione, e poi combinare le tuple provenienti da partizioni diverse.

Primo passo - Calcolare le tuple all'interno di una partizione

- Il valore iniziale è una tupla che rappresenta la somma iniziale delle temperature e il numero di elementi considerati: `(0.0, 0)`
- La funzione che combina valori all'interno di ogni partizione accetterà due parametri (`sum_count`, `other_value`):
  - `sum_count` sarà una tupla contenente i valori correnti della somma di temperature e del numero di elementi considerati;
  - `other_value` sarà un nuovo valore di temperatura.
- Vogliamo sommare il nuovo valore di temperatura a quelli già considerati, e incrementare il contatore.

```
def combine_within_partition(sum_count, other_value):
    curr_sum, curr_count = sum_count

    return curr_sum + other_value, curr_count + 1
```

Secondo passo - Combinare risultati parziali di partizioni diverse

- La funzione che combina risultati parziali provenienti da partizioni diverse accetterà due parametri (`sum_count1`, `sum_count2`).
- Entrambi i parametri saranno tuple del tipo `(values_sum, values_count)`.
- Vogliamo sommare i valori tra loro e i contatori tra loro.

```
def combine_between_partitions(sum_count1, sum_count2):
    sum1, count1 = sum_count1
    sum2, count2 = sum_count2

    return sum1 + sum2, count1 + count2
```

- Alla fine otterremo ancora tuple del tipo `(values_sum, values_count)`.
- Per calcolare la media possiamo utilizzare una `mapValues` per calcolare `values_sum / values_count`.

Temperatura media - aggregateByKey

```
def combine_within_partition(sum_count, other_value):
    curr_sum, curr_count = sum_count

    return curr_sum + other_value, curr_count + 1


def combine_between_partitions(sum_count1, sum_count2):
    sum1, count1 = sum_count1
    sum2, count2 = sum_count2

    return sum1 + sum2, count1 + count2


def calc_avg(sum_count):
    values_sum, count = sum_count
    return values_sum / count


# Supponiamo di avere a disposizione le temperature già parsate (v. slide precedenti)
parsed_temp.take(3) # [(2008, 13.7), (2008, 13.0), (2008, 13.0)]

initial_val = (0.0, 0)
sum_counters = parsed_temp.aggregateByKey(initial_val, combine_within_partition,
                                          combine_between_partitions)

avg_temp = sum_counters.mapValues(calc_avg)
avg_temp.collect()
```

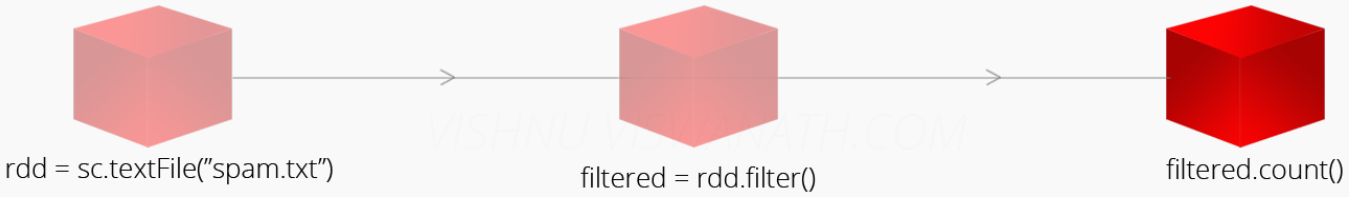
combineByKey

- `combineByKey` è un metodo ancora più generale.
- `groupByKey`, `reduceByKey` e `aggregateByKey` sono implementate in funzione di `combineByKey`.
- La differenza principale, rispetto a `aggregateByKey`, è che il valore iniziale è anch'esso una funzione.
- Per utilizzare `combineByKey` è quindi necessario specificare tre funzioni: una per calcolare il valore iniziale, una per combinare valori all'interno di ciascuna partizione, e una per combinare risultati parziali di partizioni diverse.

Persistenza, broadcast e contatori

Persistenza dei dati

- Se si accede più volte allo stesso RDD, Spark ricalcola tutte le trasformazioni necessarie per ricostruirlo, a partire dalla creazione dell'RDD originario.



- Una delle funzionalità più importanti di Spark è la capacità di mantenere parte dei dati in memoria durante l'elaborazione, per evitare questo problema e migliorare le prestazioni.

- Per specificare che un RDD deve essere mantenuto in memoria si usano i metodi `cache()` e `persist()`.

- Dopo aver utilizzato uno di questi metodi, la prima volta che viene eseguita un'azione su quell'RDD le sue partizioni saranno memorizzate in memoria sui nodi del cluster corrispondenti.

Persistenza dei dati

- La cache di Spark è fault-tolerant: se una qualsiasi partizione di un RDD viene persa, verrà automaticamente ricalcolata utilizzando le trasformazioni che l'hanno creata in origine.
- Ogni RDD può essere memorizzato utilizzando un diverso livello di archiviazione:
  - `StorageLevel.MEMORY_ONLY`
  - `StorageLevel.DISK_ONLY`
  - `StorageLevel.MEMORY_AND_DISK`
- Il metodo `persist()` consente di specificare il livello di archiviazione:

```
rdd_cached = rdd.persist(StorageLevel.DISK_ONLY)
```

- Il metodo `cache()` è equivalente a

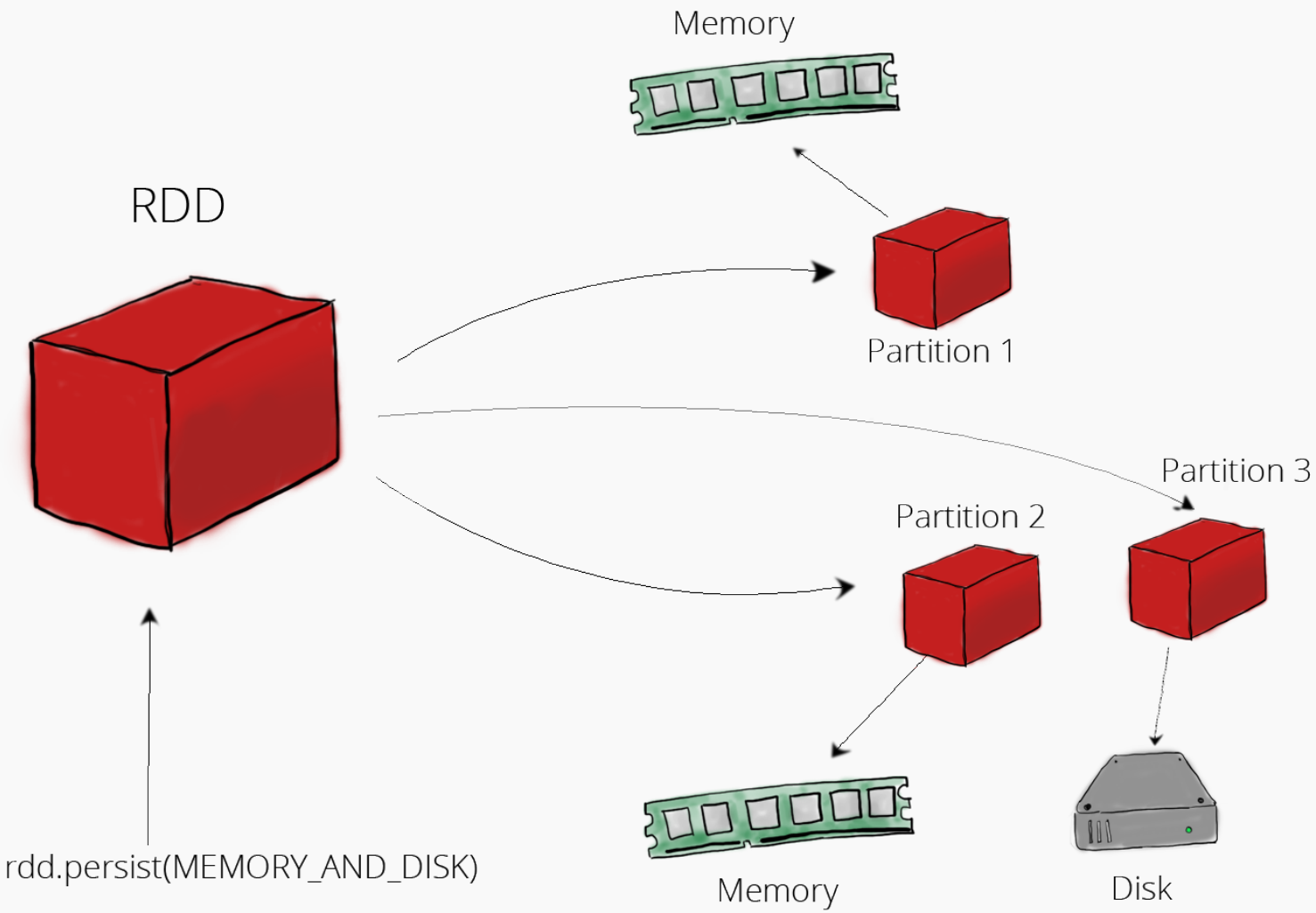
```
rdd_cached = rdd.persist(StorageLevel.MEMORY_ONLY)
```

Persistenza dei dati

- `MEMORY_ONLY`: memorizza l'RDD in memoria. Se non c'è abbastanza spazio, alcune partizioni non vengono salvate in cache e sono ricalcolate al volo quando necessario.
- `DISK_ONLY`: memorizza l'RDD su disco.
- `MEMORY_AND_DISK`: memorizza l'RDD in memoria. Se non c'è abbastanza spazio, alcune partizioni vengono salvate su disco.

Type	Memory used	CPU Time
`MEMORY_ONLY`	High	Low
`DISK_ONLY`	Low	High
`MEMORY_AND_DISK`	High	Medium

Persistenza dei dati



Variabili condivise

- Spark consente l'utilizzo di due tipi di variabili condivise, con precise limitazioni:
  - accumulatori
  - variabili broadcast
- Gli accumulatori consentono di mantenere uno o più contatori distribuiti, ad esempio per motivi di debug.
- Le variabili broadcast sono molto utili per rendere disponibili alcune variabili (in sola lettura) a tutti i nodi, in modo efficiente.

Accumulatori

- All'interno delle funzioni utilizzate nelle trasformazioni/azioni, è possibile fare riferimento a variabili definite al di fuori (ad esempio variabili globali).
- In questo caso, però, ogni task in esecuzione sul cluster riceve una nuova copia della variabile, e gli aggiornamenti da queste copie non vengono propagati al driver.
- Ad esempio, se proviamo a contare i valori di temperatura maggiori di $30\,^{\circ}\mathrm{C}$ con una variabile globale, senza utilizzare accumulatori, ciascun worker modificherà la propria copia locale della variabile, e la copia presente sul driver avrà sempre valore 0.

Tentativo di implementare un contatore senza usare accumulatori

```
def parse_temp(line):
    date, time, temp = line.split(',')
    year, month, day = date.split('-')

    global temp_over30    # Necessario per modificare la variabile globale

    if float(temp) > 30.0:
        temp_over30 += 1

    return int(year), float(temp)


temp_over30 = 0
rdd = sc.textFile('temperature_input/')
rdd_parsed = rdd.map(parse_temp)

print(temp_over30)        # Il valore stampato sarà 0
```

Accumulatori

- Per creare un accumulatore, si utilizza il metodo `sc.accumulator()` nel driver:

```
count = sc.accumulator(0)
```

- I worker potranno incrementare il valore dell'accumulatore utilizzando il metodo *add* (o l'operatore `+=`):

```
count += 1
```

- Infine, il driver potrà visualizzare il valore dell'accumulatore:

```
print(count.value)
```

- I worker non possono visualizzare il valore dell'accumulatore, possono solo incrementarlo.

- Attenzione: visto che le trasformazioni in Spark sono lazy, è necessario eseguire un'azione per ottenere il valore corretto.

Contatore delle temperature maggiori di una soglia - Accumulatori

```
def parse_temp(line):
    date, time, temp = line.split(',')
    year, month, day = date.split('-')

    global temp_over30        # Necessario per modificare la variabile globale

    if float(temp) > 30.0:
        temp_over30 += 1

    return int(year), float(temp)


temp_over30 = sc.accumulator(0)
rdd = sc.textFile('temperature_input/')
rdd_parsed = rdd.map(parse_temp)

print(temp_over30)            # Il valore stampato sarà 0

rdd_parsed.count()            # Eseguiamo un'azione qualsiasi sull'RDD

print(temp_over30)            # Il valore stampato sarà 1017
```

Variabili broadcast

- Le variabili broadcast consentono di inviare una variabile (in sola lettura) a tutti i worker, in modo efficiente.
- Sono utili se l'applicazione ha bisogno di inviare, ad esempio, grosse tabelle di lookup o vettori di feature in un algoritmo di machine learning.
- L'invio dei dati avviene in modo efficiente, con un meccanismo simile a quello di BitTorrent.

```
stopwords_list = load_stopwords()
stopwords = sc.broadcast(stopwords_list)
```

- I worker possono accedere al contenuto di una variabile condivisa attraverso la proprietà `value`:

```
stopwords.value
```

Word count con variabile broadcast - 1

```
import re

def load_stopwords():
    with open('stopwords-it.txt') as stopwords_file:
        stopwords = {word.strip() for word in stopwords_file}

    return stopwords


def split_words(line):
    line = line.lower()
    words = re.split('\W+', line)
    return words


def add_counter(word):
    word = word.lower()
    return word, 1


def filter_words(word):
    if word == '' or word.isdigit() or word in stopwords.value:
        return False

    return True
```

Continua →

Word count con variabile broadcast - 2

← Continua

```
stopwords = sc.broadcast(load_stopwords())
rdd = sc.textFile('wikipedia_input/')

words = rdd.flatMap(split_words).filter(filter_words).map(add_counter)

word_count = words.reduceByKey(lambda x, y: x + y)

top_words = word_count.top(100, key=lambda x: x[1])
print(top_words)
```

Lettura file CSV/JSON

- In Spark 1.6, per leggere un file CSV era necessario caricarlo in un RDD come un file di testo qualunque, splittare i campi (ad esempio usando la virgola come separatore) e modificarli come necessario (ad esempio convertendo alcuni campi da stringa a intero o float).

- Nel caso in cui fosse presente un header con i nomi delle colonne, nella prima riga del file, era necessario rimuoverlo prima di svolgere le altre operazioni.

- Da Spark 2 in poi, è possibile leggere file in formati standard (come CSV e JSON) in modo molto più semplice, utilizzando `spark.read`.

- Supponiamo di voler leggere un file CSV di noleggio biciclette, contenente informazioni sulla data e sul numero di utenti registrati, non registrati e totali che hanno noleggiato una bicicletta quel giorno:

```
date,casual,registered,total_users
2011-01-01,331,654,985
2011-01-02,131,670,801
2011-01-03,120,1229,1349
```

Lettura file CSV con sc.textFile()

```
def remove_header(line):
    if line[0].isdigit():
        return True
    else:
        return False

def parse_sharing(line):
    date, casual, registered, total_users = line.split(',')
    casual = int(casual)
    registered = int(registered)
    total_users = int(total_users)

    return {'date': date, 'casual': casual, 'registered': registered,
            'total_users': total_users}


# Lettura dei dati da file e parsing
sharing = sc.textFile('sharing.csv')
sharing = sharing.filter(remove_header).map(parse_sharing)

sharing.take(2)
# [{'casual': 331, 'date': '2011-01-01', 'registered': 654, 'total_users': 985},
#  {'casual': 131, 'date': '2011-01-02', 'registered': 670, 'total_users': 801}]
```

Spark 2 - Lettura file CSV/JSON

- Il metodo `spark.read.csv(file_path)` consente di leggere file di tipo CSV.
- È possibile specificare moltissime opzioni tramite parametri opzionali (fare riferimento all'help per maggiori informazioni). Tra le più importanti:
  - `header` (valore booleano) consente di specificare se la prima riga contiene un header con i nomi dei campi;
  - `inferSchema` (valore booleano) consente di specificare se Spark deve provare a inferire automaticamente il tipo di ciascun campo: utilizzare questa opzione rende l'operazione di lettura più lenta, ma è comodo se la struttura dell'RDD non è troppo complessa.
  - `sep` (stringa) consente di specificare il separatore (il carattere virgola, di default).
- Il metodo `csv` restituisce un `DataFrame` (struttura dati utilizzata in Spark SQL, come vedremo). I DataFrame possiedono una proprietà (chiamata `rdd`) che restituisce l'RDD corrispondente.

Lettura file CSV con spark.read.csv()

- Per leggere il dataset dell'esempio precedente è sufficiente una sola riga di codice:

```
sharing = spark.read.csv('sharing.csv', header=True, inferSchema=True).rdd
sharing.take(2)
# [Row(date=datetime.datetime(2011, 1, 1, 0, 0), casual=331, registered=654,
    total_users=985),
#  Row(date=datetime.datetime(2011, 1, 2, 0, 0), casual=131, registered=670,
    total_users=801)]
```

- L'RDD ottenuto è costituito da oggetti di tipo `Row`, che sono read-only.

- È possibile accedere alle proprietà di questi oggetti utilizzando la notazione del punto (`obj_name.property`) o la notazione delle parentesi quadre (`obj_name['property']`).

- Ad esempio, se si vuole ottenere un Pair RDD che contiene come chiave la data e come valore il numero totale di utenti:

```
date_users = sharing.map(lambda row: (row.date, row.total_users))
date_users.take(2)
# [(datetime.datetime(2011, 1, 1, 0, 0), 985),
#  (datetime.datetime(2011, 1, 2, 0, 0), 801)]
```