Panoramica – Parte 1

- Perché Spark
- Perché Scala
- Perché usare Spark con Scala

- **Differenze tra Programmazione Funzionale e Programmazione a Oggetti**

- **Principi di Programmazione Funzionale e Introduzione a Scala**

Perché Spark?

Spark è un framework open source per l'elaborazione distribuita di grandi quantità di dati.

Punti di forza:

- Velocità - dati elaborati in memoria senza scriverli su disco
- Scalabilità - parallelizzazione su diversi nodi di un cluster, aumentabili per una scalabilità orizzontale
- Fault-tolerance - se parte dell'elaborazione fallisce, Spark può ripetere l'operazione per recuperare i dati
- Semplicità - API di alto livello per Java, Python, R e Scala

Perché Scala?

Scala (**SCA**lable **LA**nguage) è un linguaggio di programmazione moderno di alto livello.

Unisce caratteristiche della programmazione orientata agli oggetti (OOP) a quelle della programmazione funzionale (FP).

Punti di forza:

- Espressività della sintassi
- Tipizzazione statica
- Immutabilità e Programmazione Funzionale
- Flessibilità con la Programmazione a Oggetti
- Compatibilità con Java
- Funzionalità avanzate non presenti in molti altri linguaggi

Perché Spark con Scala?

(invece di Java, Python, R)

- Spark è scritto in Scala: integrazione fluida e naturale senza problemi di compatiblità
- Migliore performance grazie ai tipi statici e alla velocità di Scala
- **Programmazione funzionale: testabile, modulare e scalabile**
- API Spark per Scala sono le più complete e ottimizzate
- Interoperabilità con Java

OOP vs FP

Oggetti vs Funzioni


```java
public class Quadrato {

    private int lato;

    public Quadrato(int lato) {
        this.lato = lato;
    }

    public int getLato() {
        return lato;
    }

    public void setLato(int lato) {
        this.lato = lato;
    }

    public int getArea() {
        return this.lato * this.lato;
    }

}
```

```java
public static void main(String[] args) {

    Quadrato q = new Quadrato(2);
    int area = q.getArea();
    System.out.println(area);

}
```

```scala
def areaDiUnQuadrato(lato: Int): Int = {
  lato * lato
}
```

```scala
val area = areaDiUnQuadrato(2)

println(area)
```

OOP vs FP

Stato Mutabile vs Funzioni Pure

```java
Quadrato q = new Quadrato(2);
System.out.println(q.getArea()); // 4

q.setLato(3);
System.out.println(q.getArea()); // 9
```

getArea non è una funzione pura, perchè chiamandola più volte con gli stessi argomenti (nessuno) si ottengono risultati diversi, in quanto esso si basa sullo stato interno mutabile della classe Quadrato

```scala
case class Quadrato(lato: Int)

def areaDiUnQuadrato(quadrato: Quadrato): Int = {
  quadrato.lato * quadrato.lato
}

val quadrato2 = Quadrato(2)
println(areaDiUnQuadrato(quadrato2)) // 4

val quadrato3 = Quadrato(3)
println(areaDiUnQuadrato(quadrato3)) // 9
```

AreaDiUnQuadrato è una funzione pura, perchè chiamandola più volte con lo stesso argomento (lo stesso quadrato) si otterrà sempre lo stesso risultato

OOP vs FP

Codice Imperativo vs Codice Dichiarativo

```java
List<Integer> listaDiNumeri = Arrays.asList(1, 2, 3, 4, 5);
int somma = 0;
for (Integer numero : listaDiNumeri) {
    somma += numero;
}

System.out.println("La somma è " + somma);
```

```scala
val listaDiNumeri = List(1, 2, 3, 4, 5)
val somma = listaDiNumeri.reduce(_ + _)

println(s"La somma è $somma")
```

Scala: variabili e tipizzazione

VALore (immutabile)
VARiabile (mutabile)

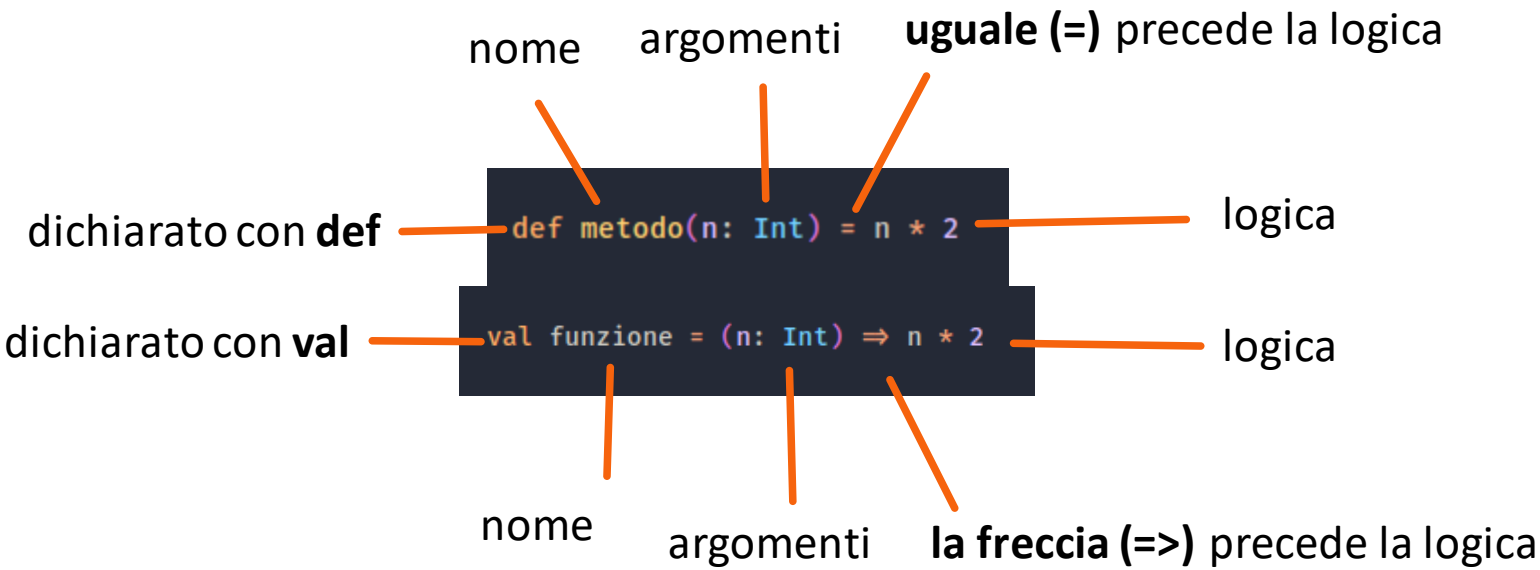
```
1    val x = 1
2    var y = 2
3
4 →  y = 3
5    x = 4
6
```

```
ms   /home/daniel/Projects/sca
     reassignment to val
     x = 4
```

Il tipo può essere esplicito o implicito (type inference)

```
1    val x = 1                    x: Int = 1
2    val y: Int = 1               y: Int = 1
3
4    println(x.getClass)          int
5 →  println(y.getClass)          int
6
```

Scala: metodi e funzioni




```
5   funzione(2)     res0: Int = 4
6   metodo(2)       res1: Int = 4
```

Panoramica – Parte 2

- Big Data
- Architettura del Cluster (Driver vs Esecutori)
- Visualizzazione degli algoritmi
- Esempi pratici in Spark

Il problema (Big Data, le 3 V)

Il concetto di Big Data è caratterizzato da 3 elementi principali:

- **Volume**
  I volumi di dati che vengono creati ogni minuto da determinati sistemi possono raggiungere dimensioni astronomiche. La lettura, analisi e salvataggio di questi dati è un'attività non banale
- **Velocity**
  La velocity è la velocità con cui i dati vengono generati, e con cui vengono fisicamente spostati da un punto all'altro, per arrivare alla loro destinazione finale di salvataggio
- **Variety**
  La tipologia di informazioni generata da sistemi Big Data può essere di diversi tipi:
  - Structured (SQL)
  - Semi-structured (XML, JSON, ecc)
  - Unstructured (testo, immagini, video, ecc)

Altri problemi (altre 2 V)

- Veracity
  La veridicità dei dati. In alcuni sistemi si potrebbero ottenere dei dati non attendibili.

- Value
  In alcuni contesti può risultare difficile estrarre del valore tangibile dai dati generati e raccolti.

La Soluzione (Spark)

Spark propone una soluzione open-source e general-purpose alla difficile gestione dei dati nel mondo Big Data.

- **Velocità**
- **Fault tolerance**
- **Scalabilità**
- **Ecosistema integrato**

Spark - Velocità

Spark esegue le operazioni in memoria.
Al costo di elevato utilizzo di RAM, si ottengono diversi vantaggi al confronto ad esempio con sistemi classici come un database relazionale.

- **Velocità**
  Operazioni in memoria sono più veloci che su disco
- **Ottimizzazione**
  Avere tutti i dati in memoria permette a Spark di ottimizzare le operazioni richieste
  - Catalyst Optimizer ad alto livello (trasformazione piano logico a fisico)
  - Tungsten Execution Engine a basso livello (ottimizzazione a basso livello dei vari step)
- **Meno Overhead I/O**
  Riducendo al minimo i momenti in cui Spark deve leggere fisicamente dei dati dal disco si diminuiscono le operazioni di I/O da disco che possono risultare lente.
  Idealmente un processo Spark dovrebbe leggere una volta tutti i dati, trasformarli, e riscriverli alla fine, tenendo tutto in memoria durante l'esecuzione del processo.

Spark – Fault Tolerance

Spark fornisce dei sistemi di fault-torance attraverso la distribuzione dei dati e la data lineage (genealogia dei dati)

- **RDD – Resilient Distributed Datasets**
  L'RDD è una collezione di dati distribuita sul cluster. Nell'evento di un qualche fallimento o problema, solo una parte dei dati viene impattata.
- **Data Lineage**
  Se dovesse verificarsi una perdita di dati su un nodo/esecutore, Spark è in grado di ricostruire i dati persi ri-eseguendo le operazioni che li hanno prodotti inizialmente
- **Recovery a runtime**
  Queste funzionalità consentono a Spark di recuperare a
  runtime dei dati persi, evitando che l'esecuzione del processo debba essere interrotta per risolvere il pr
  oblema che ha causato la perdita di dati

  Questo è essenziale in un sistema Big Data in cui il fallimento di un nodo non è
  un evento particolarmente raro, dato principalmente dalla grande quantità di macchine utilizzate

Spark – Scalabilità

Spark consente di scalare un sistema facilmente in base alle richiesta di potenza di calcolo

- **Calcolo Distribuito**
  Spark utilizza la distribuzione dei dati e dei calcoli per poter gestire grandi quantità di dati in maniera parallela, questo permette la scalabilità orizzontale di un sistema, a seconda delle richieste
- **Utilizzo ottimale delle risorse**
  Spark è in grado, grazie alle qualità viste in precedenza, di utilizzare in maniera ottimale le risorse disponibili, siano esse tante o poche
- **Versatilità**
  Spark può essere utilizzato per gestire enormi quantità di dati distribuiti su centinaia di nodi di un cluster, ma può anche essere usato per processare pochi dati su una singola macchina

Spark – Ecosistema Integrato

Spark esiste in un ricco ecosistema in grado di gestire diverse necessità

- **Librerie e API:**
  SparkSQL, Spark Streaming, MLlib, GraphX
- **Integrazione con fonti di dati esterne:**
  HDFS, Cassandra, HBase, S3
- **Lettura di tipi di dati diversi:**
  CSV, JSON, Parquet, Avro
- **Compatibilità con altri ecosistemi Big Data:**
  Apache Hadoop, Apache Hive, Kubernetes

Il Cluster

Un cluster spark ha due componenti principali: il **driver** e gli **esecutori**

- **Driver (Direttore d'orchestra), si trova su un nodo dedicato del cluster:**
  - Punto di ingresso del processo
  - Crea lo SparkContext/SparkSession
  - Schedula i task da eseguire
  - Monitora l'esecuzione da parte degli esecutori

- **Esecutori (Musicisti dell'orchestra), si trovano su diversi nodi del cluster:**
  - Eseguono i calcoli
  - Salvano i dati in memoria
  - Comunicano il proprio stato all'Esecutore

SparkContext/SparkSession

Lo SparkContext (Spark1) o lo SparkSession (Spark2) sono oggetti il cui scopo è quello di fornire punti di entrata all'API Spark, vengono usati per:

- **SparkContext:**
  - Creazione RDD (Resilient Distributed Dataset)
  - Offre connettività al cluster
  - Responsabile per la schedulazione ed esecuzione dei task

- **SparkSession (nato con Spark2):**
  - Unione di vecchi "Context" di Spark1, offre un unico punto di entrata per la lettura di dati: SparkSQL, lettura dati da file ecc
  - Configurazione applicazione
  - Contiene lo SparkContext

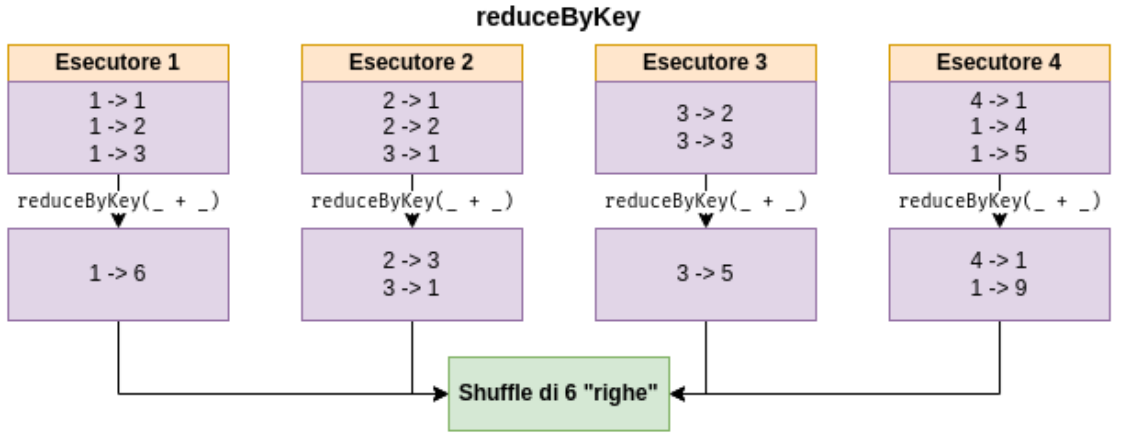
Parte 3
Considerazioni Spark / Big Data

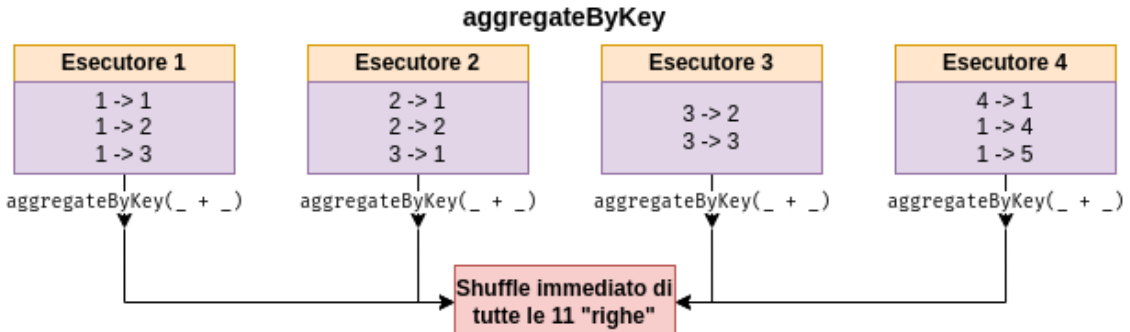
Sviluppando software Big Data in Spark ci sono delle considerazioni da tenere a mente

- **Distribuzione dei dati**
- **Spostamento dei dati tra diversi nodi (data shuffling)**
- **Lazy execution**
- **Serializzazione dei dati**
- **Formato dei dati**
- **Idempotenza**
- **Gestione degli errori**
- **Tuning**

Distirbuzione dei dati

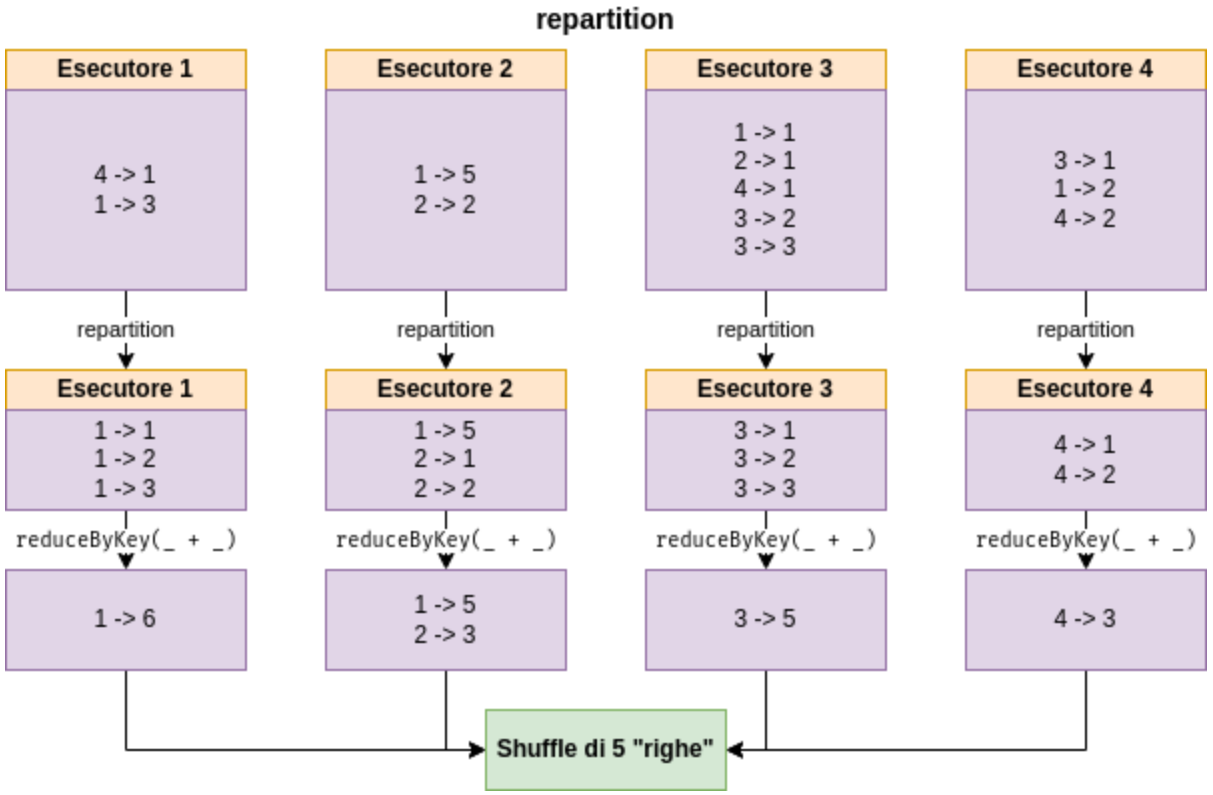
I dati processati da un'applicazione Spark sono distribuiti sul cluster, per questo bisogna tenere a mente che alcune operazioni necessitano potenzialmente di spostamenti di dati tra diversi nodi del cluster: queste operazioni sono molto costose e andrebbero minimizzate





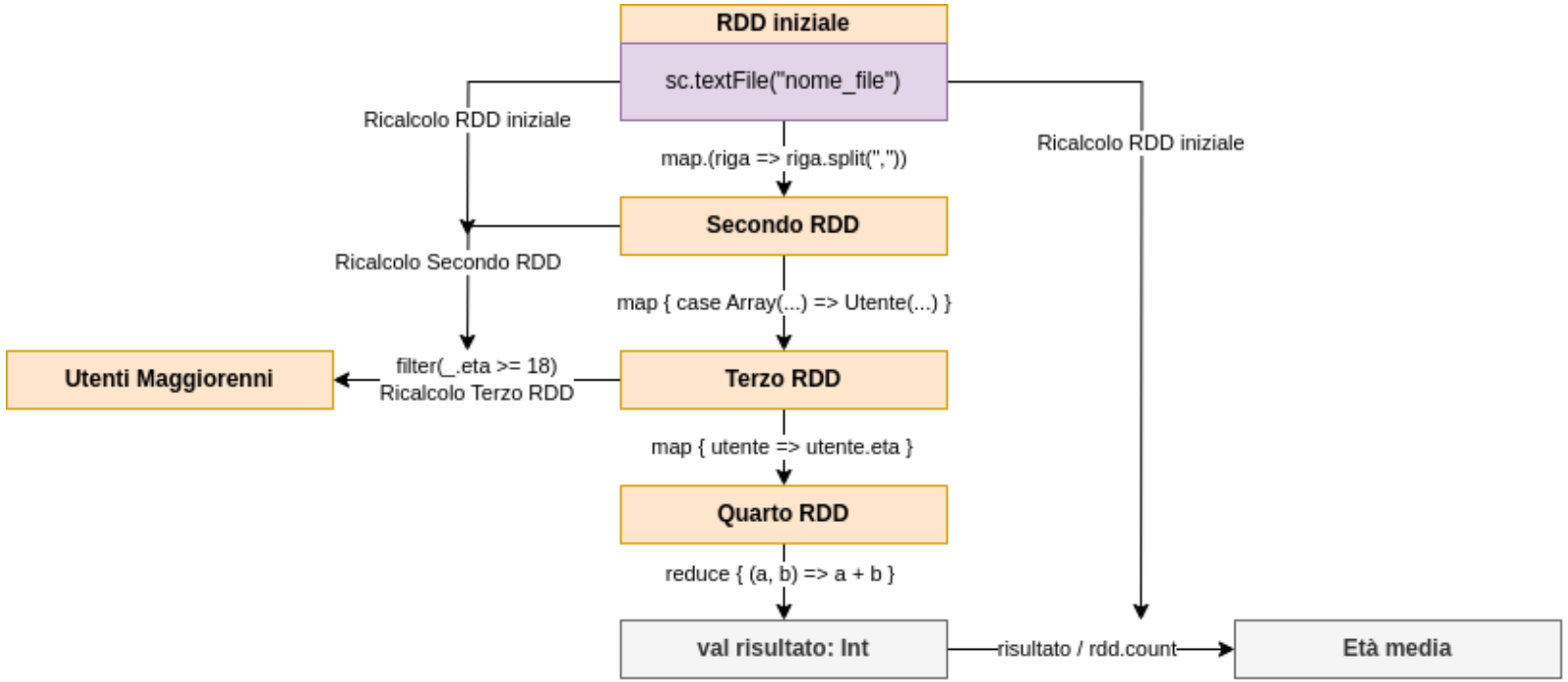
Distirbuzione dei dati v2

I dati distribuiti sul cluster potrebbero essere distribuiti in maniera sbilanciata.
Bisogna accorgersi o prevedere quando questo può succedere, e ribilanciare dati di conseguenza.



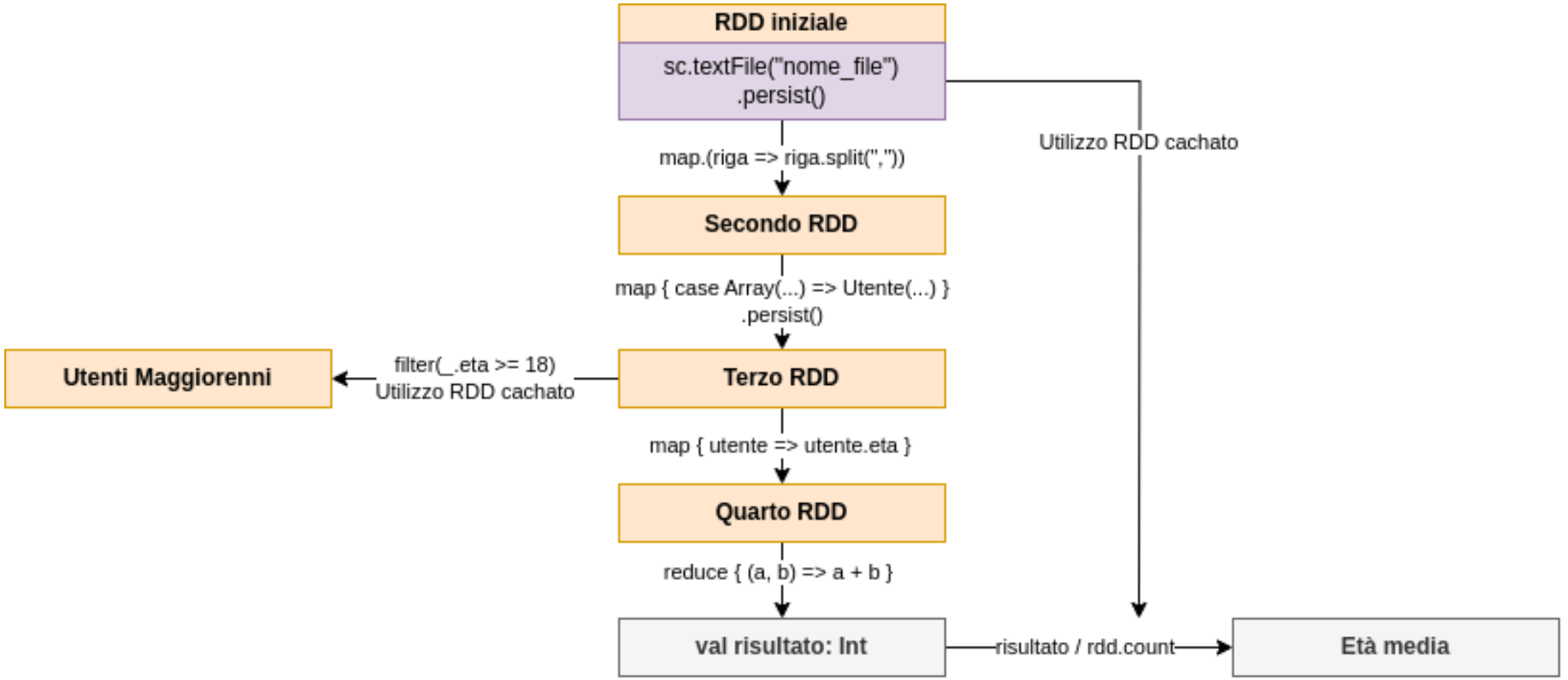
Lazy Execution

Gli oggetti Spark (RDD/DataFrame/DataSet) vengono elaborati in maniera Lazy.
Si può forzare la loro elaborazione quando può risultare utile farlo (count).
E' bene utilizzare qualche forma di caching (persist) quando i dati iniziali vengono elaborati più volte.



Lazy Execution - persist

Si può utilizzare il metodo persist per cachare il risultato del calcolo, in modo che questo non venga rieseguito più volte



Serializzazione dei dati

Inevitabilmente i dati devono essere spostati tra i vari nodi dell'applicazione Spark, quando questo avviene è bene che vengano serializzati efficientemente in modo da velocizzare il loro trasferimento, e minimizzare l'utilizzo di memoria.

Utilizzare il Kryo serializer invece del serializzatore base fornito da Java.

Assicurarsi che tutto ciò che deve spostarsi tra i vari nodi sia serializzabile.

```
spark-submit \
--conf "spark.serializer=org.apache.spark.serializer.KryoSerializer"

spark.conf.registerKryoClasses(Array(classOf[LaMiaClasse]))
```

Formato dei dati

Dovendo gestire grandi quantità di dati è importante scegliere un formato efficiente per salvarli.

E' bene evitare, quando possibile, formati poco performanti in termini di memoria e/o spazio come:

- CSV
- JSON
- XML

E prediligere invece formati nati appositamente per risolvere i problemi creati dal mondo Big Data, come:

- Parquet
- Avro
- ORC (Optimized Row Columnar)

Idempotenza

Definizione:
In informatica, in matematica, e in particolare in algebra, l'idempotenza è una proprietà delle funzioni per la quale applicando molteplici volte una funzione data, il risultato ottenuto è uguale a quello derivante dall'applicazione della funzione un'unica volta.

I processi Big Data, complessi per natura, sono suscettibili ad errori e fallimenti, Spark è nato per gestire queste situazioni grazie alla distribuzione dei dati e la data lineage.

La fault-tolerance di Spark però non basta se il nostro codice non tiene conto conto di eventuali fallimenti e possibili ricalcoli richiesti da Spark.

Gestione degli errori

Un processo Spark Big Data può essere complesso sia per la logica che viene eseguita, sia per l'architettura fisica sui cui avvengono i calcoli.

Risulta fondamentale una buona gestione degli errori, su più fronti:

- eventuali azioni da compiere in seguito a potenziali fallimenti
- logging degli errori, tenendo conto della natura distribuita e parallela dei processi Spark, e quindi anche dei log che ne risultano.

Tuning

Lo sviluppo del processo Spark è il primo passo, il successivo è quello di test e tuning tramite varie configurazioni rese disponibili da Spark. Le più importanti sono:

Nome	Descrizione
spark.executor.instances	Numero totale di esecutori
spark.executor.cores	Numero di core per ogni esecutore
spark.executor.memory	Memoria disponibile ad ogni esecutore
spark.driver.memory	Memoria disponibile al driver
spark.default.parallelism	Numero di partizioni di default