Terzo giorno: pomeriggio

Agenda 6/8

Il cluster

Architettura di un'applicazione Spark
Spark Driver, Spark executors,
cluster manager (Spark standalone, YARN o Mesos)

Modi di esecuzione
Cluster mode, client mode, local mode

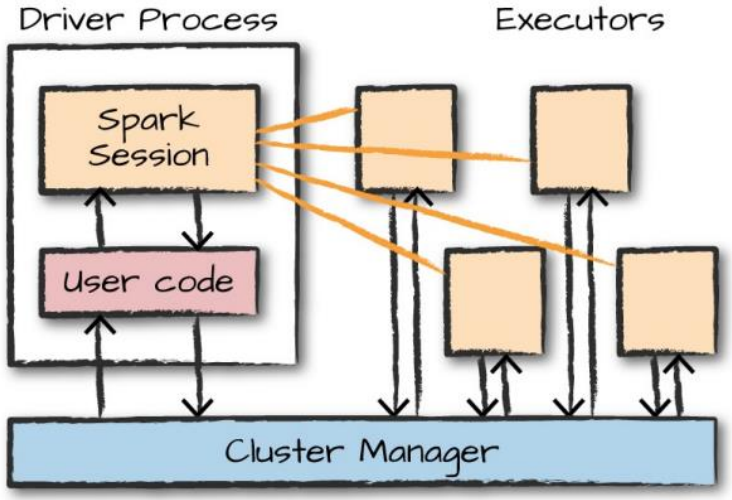
Ciclo di vita di un'applicazione Spark
Fuori e dentro Spark; tasks, jobs, stages

La sessione
SparkSession, SparkContext, SQLContext, HiveContext

Il programa
Struttura; compilazione (maven, sbt); spark-submit

Configurazione
SparkConf, argomenti di spark-submit;

Esercizi di laboratorio
Scrittura dei primi programmi in Spark; compilazione ed esecuzione.



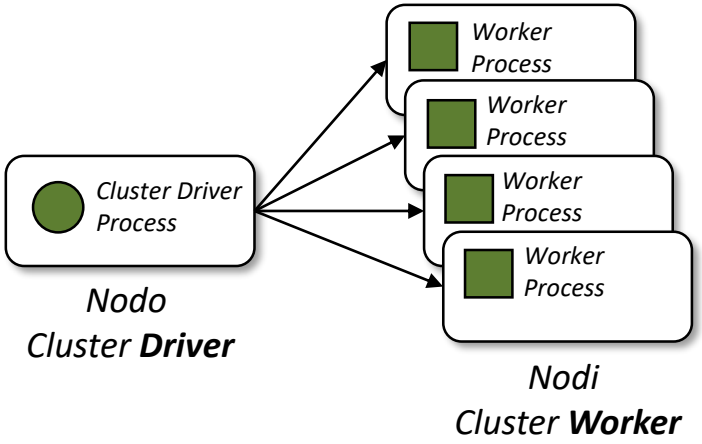
Il cluster

Il ***cluster*** è un insieme di computer (*nodi*) connessi fra loro.

Spark utilizza il cluster tramite un *cluster manager*.

Il cluster manager ha un'architettura distribuita e prevede:

- Un nodo *driver* (chiamato master)
- Un certo numero di nodi *worker*.



Architettura di un'applicazione Spark

Un'applicazione Spark ha due componenti principali:

- Lo ***Spark Driver***
  - È il processo che controlla l'esecuzione dell'applicazione.
  - Comunica con il cluster manager per ottenere risorse e lanciare gli esecutori.
  - Distribuisce il lavoro agli esecutori; controlla e raccoglie i risultati

- Gli ***Spark Executors***
  - Sono i processi che eseguono i task assegnati dallo Spark Driver.
  - Sono i componenti che eseguono materialmente l'elaborazione dei dati
  - Hanno come unica responsabilità:
    - Ricevere un task dallo Spark Driver
    - Eseguirlo
    - Comunicare lo stato finale (successo/fallimento) e gli eventuali risultati

Architettura di un’applicazione Spark

Un’applicazione Spark ha due componenti principali:

- Ogni *Esecutore* è un processo indipendente che gira su un nodo del cluster.
- Ogni *Applicazione Spark* ha i suoi propri *esecutori* indipendenti.
- Anche lo *Spark Driver* è un processo. A seconda della configurazione può girare su un nodo del cluster o su un computer esterno.
- Ogni processo Spark (il driver e tutti gli esecutori) usa una ***JVM*** (*Java Virtual Machine*) indipendente.

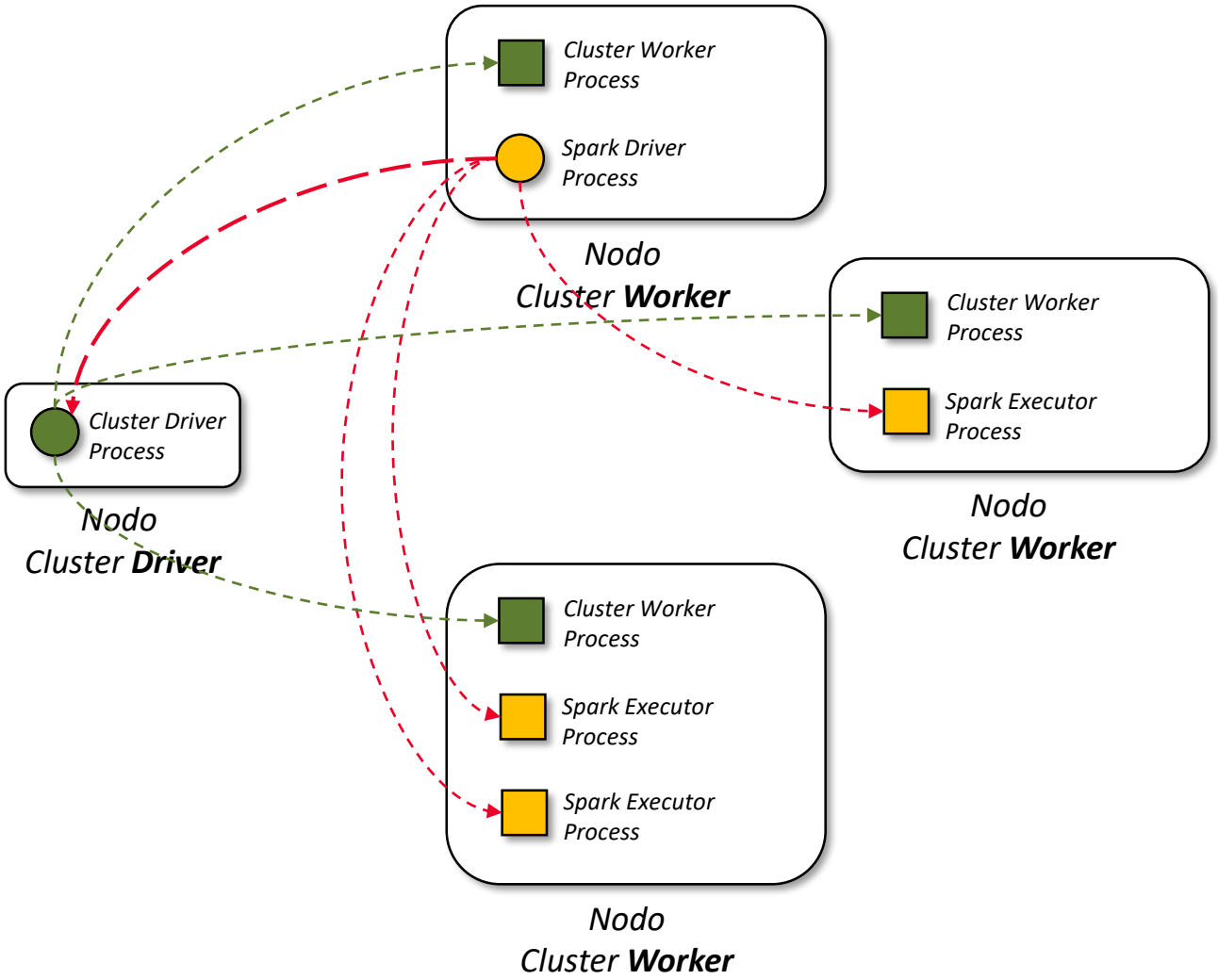
Attenzione a non confondere!

Cluster Manager ≠ Spark

Cluster Driver (o Master) e ***Cluster Worker***		***Spark Driver*** e ***Spark Executor***
Sono i nodi del cluster. I relativi processi formano il *Cluster Manager*	≠	Sono processi. Insieme formano il cosiddetto *Spark Cluster.* Nota: Ci possono essere più *esecutori* su uno stesso nodo.

Attenzione a non confondere!

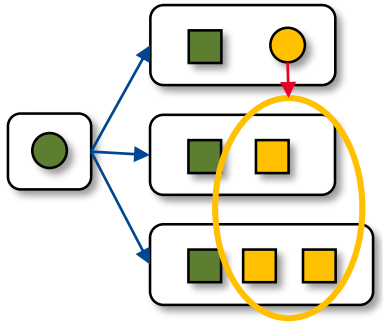
Cluster Manager ≠ Spark



Modalità di esecuzione

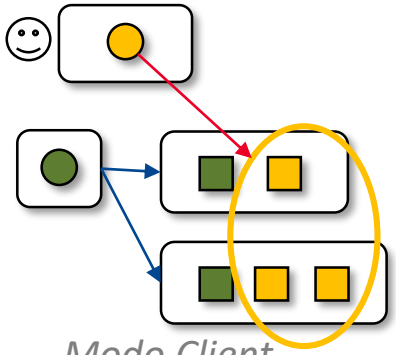
È possibile eseguire un'Applicazione Spark in tre modi diversi:

- ***Modo Cluster***
  - È il modo più comune
  - Tutti i processi (Driver ed Executors) girano sui nodi del cluster



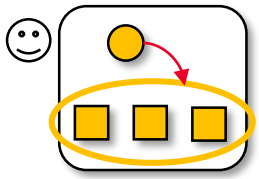
Modo Cluster

- ***Modo Client***
  - Lo *Spark Driver* rimane sul computer che ha fatto partire l'applicazione
  - Gli *Executor* girano sui nodi del cluster



Modo Client

- ***Modo Locale***
  - Non usa il cluster:
    - Utile per imparare Spark, fare test e mettere a punto le applicazioni
    - Non per la produzione!
  - *Spark Driver* ed *Executors* girano come processi diversi sullo stesso computer



Modo Locale

Ciclo di vita di un'applicazione Spark

Si avvia l'applicazione da un computer «locale», non necessariamente sul cluster.
(*Gateway Machines* o *Edge Nodes*)

L'applicazione è costituita da un JAR (Java Archive) precompilato oppure da uno script Python/R.

Si può usare il comando *spark-submit:*

```
spark-submit \
        --class <main-class> \
        --master <master-url> \
        --deploy-mode <mode> \
        --conf <key>=<value> \
        ... # altre opzioni
        <application-jar> \
        [eventuali-argomenti]
```

Un esempio concreto:

```
spark-submit \
  --class WordCount \
  --master local \
  target/test-1.0-SNAPSHOT.jar
```

Ciclo di vita di un'applicazione Spark

2

Il *master (*il *driver* del *cluster manager*) riceve la richiesta di creare lo ***Spark Driver***.

La richiesta viene accettata: il *master* crea il processo *Spark Driver* su un nodo del cluster e lo fa partire.

Il processo che ha lanciato l'applicazione termina. L'applicazione Spark comincia la sua esecuzione sul cluster.

Ciclo di vita di un'applicazione Spark

Comincia l'esecuzione del codice utente. Viene creata la ***Spark Session***.

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder()
        .appName("Tim Spark Example")
        .config("spark.sql.warehouse.dir", "/user/hive/warehouse")
        .getOrCreate()
```

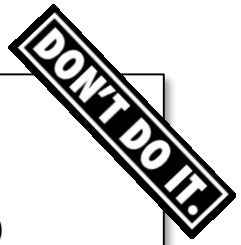
La *Spark Session* inizializza il *cluster Spark*: chiede al *cluster manager* di creare gli *Esecutori*.

Il *cluster manager* crea gli *esecutori* e comunica alla *Spark Session* gli indirizzi.

Nota: Il numero di esecutori e la quantità di memoria per esecutore sono specificati dall'utente tramite i parametri della *spark-submit*

Ciclo di vita di un'applicazione Spark

Attenzione: i «vecchi» programmi Spark creano la sessione usando il pattern:




```
val conf = new SparkConf()
        .setMaster("local")
        .setAppName("My App")
val sc = new SparkContext(conf)
```

Questo va evitato! Il nuovo sistema è più robusto e assicura che non ci siano conflitti se ci sono diverse librerie che cercano di creare una sessione nella stessa Applicazione Spark.

Ciclo di vita di un'applicazione Spark

4

Il *cluster Spark* è attivo e funzionante. Il codice utente viene eseguito.

Il driver e gli esecutori comunicano fra loro direttamente:

- Il driver assegna i task
- Gli esecutori svolgono i task assegnati
- Alla fine dell'esecuzione comunicano al driver lo stato (successo o fallimento)
- I dati vengono distribuiti fra i vari esecutori a seconda delle necessità

Ciclo di vita di un’applicazione Spark

5

Alla fine il processo Spark Driver termina (con successo o per un errore).

Il cluster manager termina gli esecutori.

L’utente può consultare il cluster manager per verificare se l’applicazione è terminata con successo.

Durante l'esecuzione

- L'esecuzione è determinata dalle ***azioni*** che vengono elaborate in modo sequenziale
- Spark compila un ***DAG*** (Direct Acyclic Graph) che rappresenta le relazioni fra le varie trasformazioni specificate.
- Sulla base dell'azione corrente e del DAG, Spark genera un «piano di esecuzione»: una serie di trasformazioni da eseguire in cascata per preparare gli argomenti per l'azione corrente.
- Il «piano di esecuzione» subisce successivi raffinamenti:
  - *Parsed Logical Plan*
  - *Analysed Logical Plan*
  - *Optimized Logical Plan*
  - *Physical Plan*

Durante l'esecuzione

```
df.filter("c1 = 0").filter("c2 = 0").explain(true)
```

```
== Parsed Logical Plan ==
'Filter ('c2 = 0)
+- Filter (c1#3L = cast(0 as bigint))
   +- Project [id#0L, c1#3L, (id#0L % cast(17 as bigint)) AS c2#7L]
      +- Project [id#0L, (id#0L % cast(13 as bigint)) AS c1#3L]
         +- Range (0, 10000, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint, c1: bigint, c2: bigint
Filter (c2#7L = cast(0 as bigint))
+- Filter (c1#3L = cast(0 as bigint))
   +- Project [id#0L, c1#3L, (id#0L % cast(17 as bigint)) AS c2#7L]
      +- Project [id#0L, (id#0L % cast(13 as bigint)) AS c1#3L]
         +- Range (0, 10000, step=1, splits=Some(8))

== Optimized Logical Plan ==
Project [id#0L, (id#0L % 13) AS c1#3L, (id#0L % 17) AS c2#7L]
+- Filter ((((id#0L % 13) = 0) && isnotnull((id#0L % 17))) && ((id#0L % 17) = 0))
   +- Range (0, 10000, step=1, splits=Some(8))

== Physical Plan ==
*Project [id#0L, (id#0L % 13) AS c1#3L, (id#0L % 17) AS c2#7L]
+- *Filter ((((id#0L % 13) = 0) && isnotnull((id#0L % 17))) && ((id#0L % 17) = 0))
   +- *Range (0, 10000, step=1, splits=8)
```

Took: 1.392s, at 2019-03-14 05:56

Durante l’esecuzione

L’esecuzione del «piano fisico» può essere divisa secondo la gerarchia seguente:

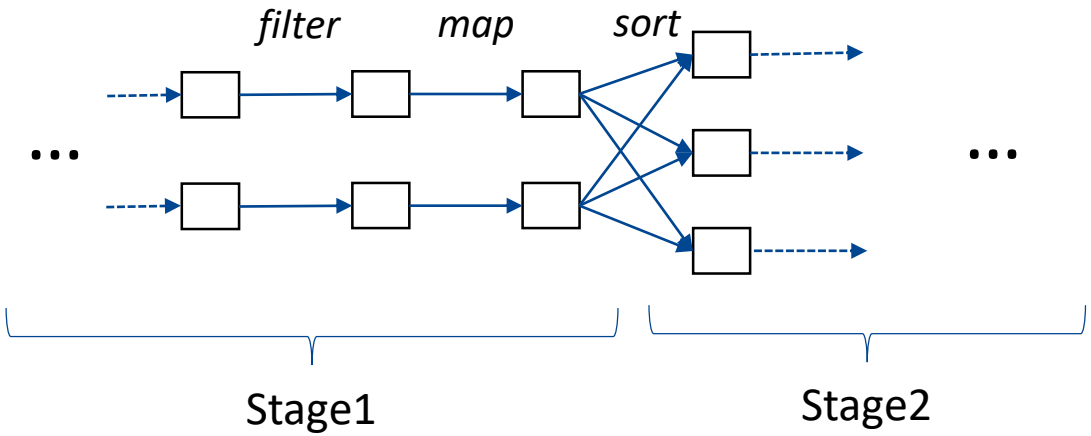
- ***Jobs***
- ***Stages***
- ***Tasks***

Job

- Il calcolo delle ***Trasformazioni*** («pigre») viene scatenato dalle ***Azioni***.
- Le azioni producono dei risultati
- All’esecuzione di ogni azione corrisponde uno ***Spark Job***
- In genere i *job* vengono eseguiti sequenzialmente

Stage

- Ogni *Job* si articola in ***Stages***; uno *Stage* è un'unità fisica di esecuzione.
- Spark cerca di fare quanto più lavoro possibile in un singolo *stage*
- Il numero di *stage* dipende da quanti ***shuffle*** sono necessari per eseguire l'azione
- Dopo ogni *shuffle* viene creato un nuovo *stage*.



Task

- Durante uno Stage gli esecutori lavorano in parallelo, ognuno su una diversa partizione dei dati da elaborare
- Il ***Task*** è un’unità di calcolo applicata ad un’unità di dati. Corrisponde ad:
  - un determinato insieme di trasformazioni
  - su una determinata partizione (cioè un determinato blocco di dati da elaborare)

Il numero di task corrisponde al numero di partizioni.

NON al numero di esecutori!

Durante l'esecuzione

Spark mette automaticamente in pratica due strategie:

- ***Pipelining***

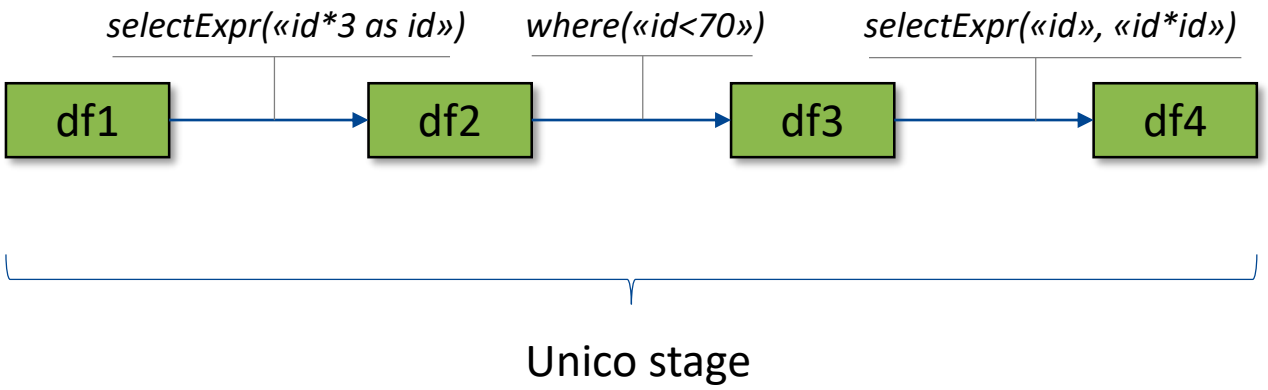
  Mette nello stesso *stage* tutte le trasformazioni che possono essere in sequenza da uno stesso Executor.

- ***Persistenza degli Shuffle***

  Salva su disco i dati che devono essere trasferiti con lo *shuffle*.

Pipelining

- Questa ottimizzazione avviene a basso livello (RDD e ancora più profondamente)
- Ogni sequenza di operazioni che possono essere eseguite di seguito senza necessità di trasferire i dati da un altro esecutore vengono eseguite nello stesso *stage*.

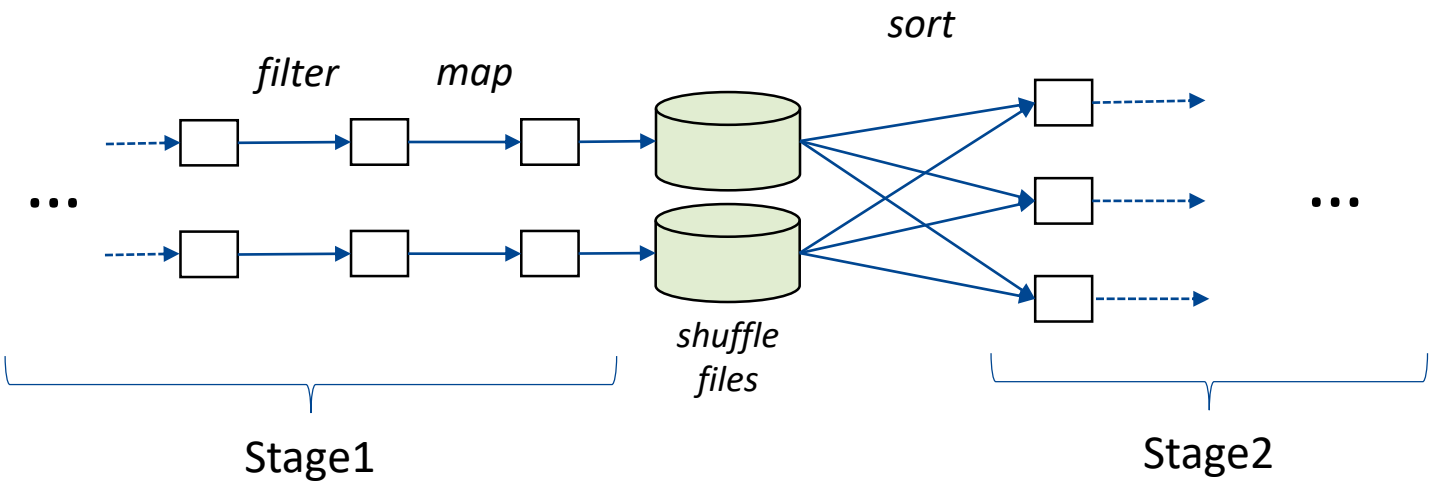



```
== Physical Plan ==
*Project [(id#0L * 3) AS id#3L, ((id#0L * id#0L) * 9) AS (id * id)#7L]
+- *Filter ((id#0L * 3) < 70)
   +- *Range (0, 10000, step=1, splits=4)
```

Persistenza degli Shuffle

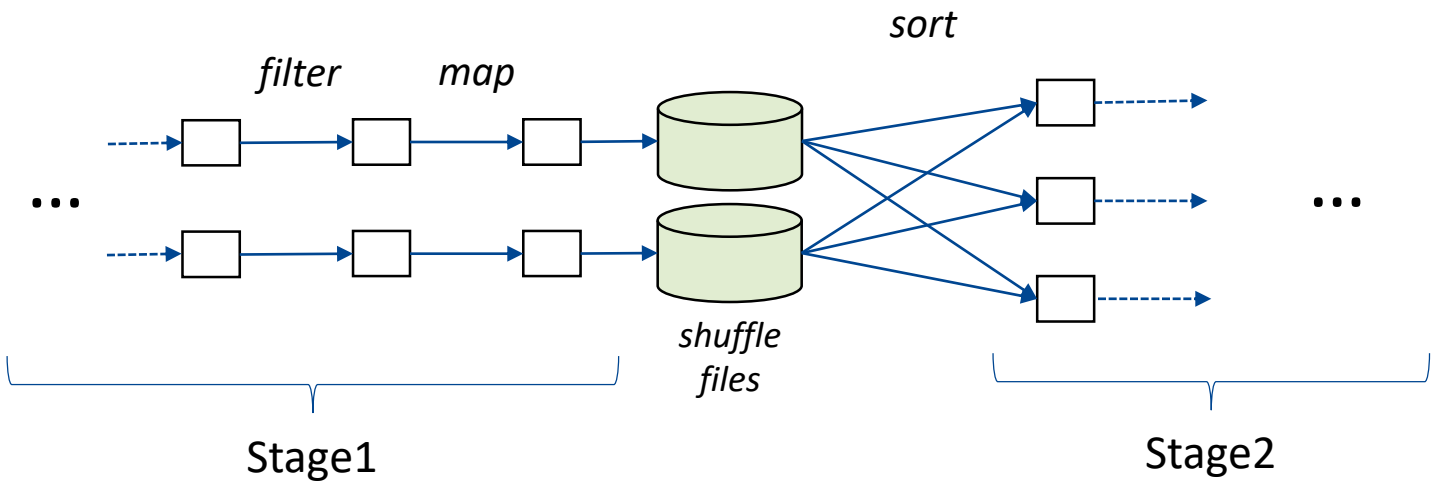
Quando Spark deve eseguire uno shuffle:

- I task logicamente precedenti scrivono i dati calcolati sui loro dischi locali (***shuffle files***)
- I task che eseguono lo shuffle prendono i record corrispondenti dagli *shuffle files* (trasferendo i dati da un altro nodo del cluster se necessario)



Persistenza degli Shuffle

- Salvare i dati sugli *shuffle file* permette a Spark di far ripartire l'esecuzione più volte senza dover ripetere il calcolo precedente
- Questo è indispensabile se ad es. nello stage successivo ci sono più partizioni che esecutori
- Inoltre in caso di fallimento di un nodo l'elaborazione può riprendere dai dati negli shuffle files



Un esempio concreto

- Mettiamo a fuoco i concetti visti fin qui analizzando in maniera dettagliata l'esecuzione di un programma di esempio

- Questo ci serve anche per introdurre alcuni strumenti di diagnostica

Un esempio concreto

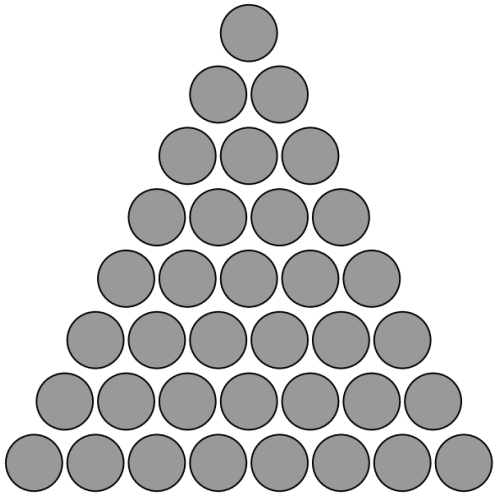
Vogliamo cercare i numeri sia triangolari che quadrati

Quanti ce ne sono da 1 a mille miliardi?

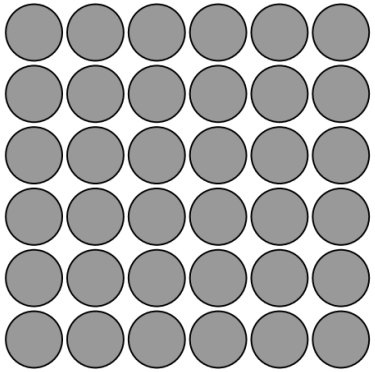
$$T_k = \frac{k(k+1)}{2}$$

$$Q_k = k^2$$

N = 36



$$T_8 = \frac{8(8+1)}{2} = 36$$



$$Q_6 = 6^2 = 36$$

Un esempio concreto

```
val t1 = spark.range(1,1500000)
   .toDF("x")
   .withColumn("2T",expr("x*(x+1)"))
val q1 = spark.range(1,1000000)
   .toDF("y")
   .withColumn("Q",expr("y*y"))
val limit = BigDecimal("1000000000000")
val t2 = t1.filter($"2T" < limit).repartition(5)
val q2 = q1.filter($"Q" < limit).repartition(6)
val df = q2.join(t2, expr("2T = 2*Q"))
   .selectExpr("Q as N", "x", "y")
   .orderBy("N")

df.show()
```

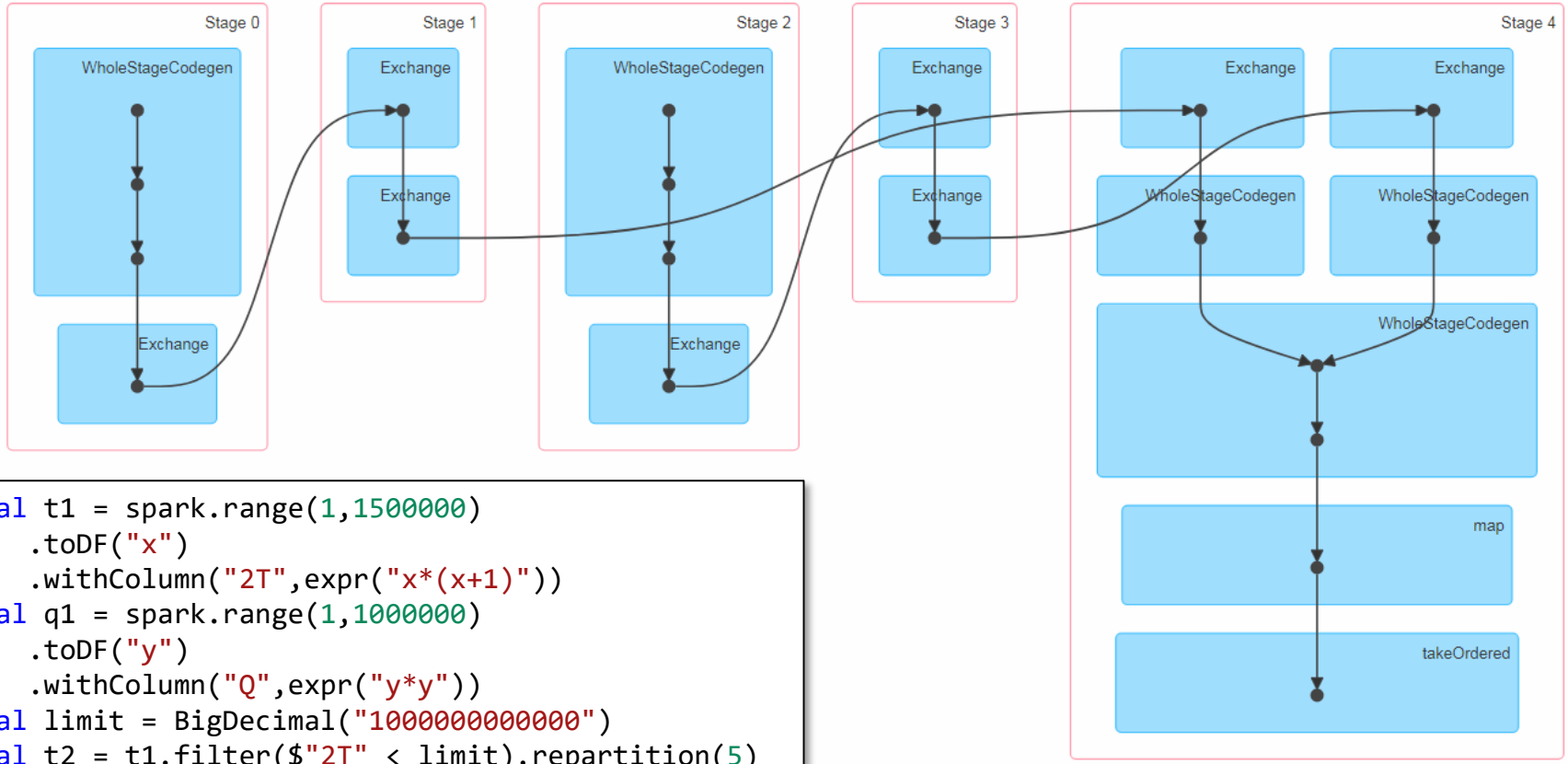
Esecuzione

- C’è un’unica azione
- Il piano di esecuzione

```
scala> df.explain()
== Physical Plan ==
*Sort [N#45L ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(N#45L ASC NULLS FIRST, 200)
   +- *Project [Q#21L AS N#45L, x#8L, y#18L]
      +- *BroadcastHashJoin [(2 * Q#21L)], [2T#11L], Inner, BuildRight
         :- Exchange RoundRobinPartitioning(6)
         :  +- *Project [id#15L AS y#18L, (id#15L * id#15L) AS Q#21L]
         :     +- *Filter ((id#15L * id#15L) < 10000000000)
         :        +- *Range (1, 100000, step=1, splits=8)
         +- BroadcastExchange HashedRelationBroadcastMode(List(input[1, bigint, false]))
            +- Exchange RoundRobinPartitioning(5)
               +- *Project [id#5L AS x#8L, (id#5L * (id#5L + 1)) AS 2T#11L]
                  +- *Filter ((id#5L * (id#5L + 1)) < 10000000000)
                     +- *Range (1, 150000, step=1, splits=8)
```

Esecuzione

La Spark UI : il DAG




```
val t1 = spark.range(1,1500000)
   .toDF("x")
   .withColumn("2T",expr("x*(x+1)"))
val q1 = spark.range(1,1000000)
   .toDF("y")
   .withColumn("Q",expr("y*y"))
val limit = BigDecimal("1000000000000")
val t2 = t1.filter($"2T" < limit).repartition(5)
val q2 = q1.filter($"Q" < limit).repartition(6)
val df = q2.join(t2, expr("2T = 2*Q"))
   .selectExpr("Q as N", "x", "y")
   .orderBy("N")

df.show()
```

Esecuzione

Abbiamo:

- Stage 0 : 8 task
- Stage 1 : 6 task
- Stage 2 : 8 task
- Stage 3 : 5 task
- Stage 4 : 200 task

```
val t1 = spark.range(1,1500000)
   .toDF("x")
   .withColumn("2T",expr("x*(x+1)"))
val q1 = spark.range(1,1000000)
   .toDF("y")
   .withColumn("Q",expr("y*y"))
val limit = BigDecimal("1000000000000")
val t2 = t1.filter($"2T" <
limit).repartition(5)
val q2 = q1.filter($"Q" <
limit).repartition(6)
val df = q2.join(t2, expr("2T = 2*Q"))
   .selectExpr("Q as N", "x", "y")
   .orderBy("N")

df.show()
```

Numero di task

Alcuni parametri di configurazione controllano il numero di task:

- *spark.default.parallelism* : sc.parallelize() con numero di partizioni non specificato

- *spark.sql.shuffle.partitions :* dopo un join

Come creare un'applicazione Spark

Per compilare

- Si può usare sbt o Apache Maven
- Sono entrambi sistemi di build per la JVM

La struttura del progetto

- Identica per sbt e maven

- Esiste un file che contiene
  - Metadati relativi al progetto (nome del package, informazioni sulla versione, ecc.)
  - Dove risolvere le dipendenze
  - Dipendenze specifiche del progetto (es. librerie)

- Questo file è ***pom.xml*** per *maven* e ***build.sbt*** per *sbt*.

La struttura del progetto

```
<file di configurazione>
src/
  main/
    resources/
      <files to include in main jar here>
    scala/
      <main Scala sources>
    java/
      <main Java sources>
  test/
    resources/
      <files to include in test jar here>
    scala/
      <test Scala sources>
    java/
      <test Java sources>
```

La struttura del progetto (semplificata)

```
<file di configurazione>
src/
  main/
    scala/
      <main Scala sources>
```

Il codice sorgente

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

object SquareTriangularNumbers {
  def main(args: Array[String]) {
    val spark = SparkSession.builder
      .appName("Square triangular numbers")
      .getOrCreate()
    import spark.implicits._

    val t1 = spark.range(1,1500000).toDF("x").withColumn("2T",expr("x*(x+1)"))
    val q1 = spark.range(1,1000000).toDF("y").withColumn("Q",expr("y*y"))
    val limit = BigDecimal("1000000000000")
    val t2 = t1.filter($"2T"<limit).repartition(5)
    val q2 = q1.filter($"Q"<limit).repartition(6)
    val df = q2.join(t2, expr("2T = 2*Q"))
      .selectExpr("Q as N", "x", "y")
      .orderBy("N")

    val results = df.collect()
    println("====================================================")
    results.foreach(println)
    println("====================================================")
  }
}
```

Il file di configurazione (pom.xml)

```
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>it.arakne</groupId>
  <artifactId>square-triangular-numbers</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>test</name>
  <url>http://maven.apache.org</url>

  <repositories>
    <repository>
      <id>scala-tools.org</id>
      <name>Scala-tools Maven2 Repository</name>
      <url>http://scala-tools.org/repo-releases</url>
    </repository>
    ...
    eccetera
```

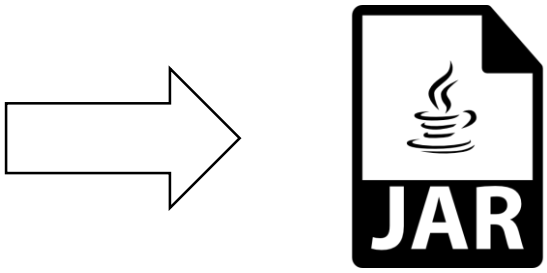
Comandi da shell

- Per compilare:

```
$ mvn package
```

oppure

```
$ mvn clean package
```



- Per lanciare l'applicazione

```
$ spark-submit \
  --class SquareTriangularNumbers \
  --master yarn \
  target/square-triangular-numbers-1.0-SNAPSHOT.jar
```

Cosa succede se un nodo si guasta durante l'esecuzione?

- **Se si guasta un nodo worker** (con uno o più Spark Executor)
  I dati persi vengono ricalcolati da un altro Executor

- **Se il nodo Master si guasta**
  Dipende dal cluster manager. Ci può essere uno Standby Master che subentra in caso di guasto

- **Se lo Spark Driver viene terminato**:
  Lo SparkContext è perduto. Tutti gli executor vengono terminati.
  Questo è un Single Point of Failure!!