# INITIATION À SPARK AVEC JAVA 8 ET SCALA

Olivier Girardot

Published with GitBook

# Table des matières

1. Introduction
2. Partie 0 : Mise en place de l'environement
3. Partie 1 : Familiarisation avec les concepts de Spark
4. Partie 2 : Mise en oeuvre des RDDs
5. Partie 3 : Aspects avancés de Spark
6. Partie 4 : Spark Streaming
7. Partie 5 : SparkSQL
8. Partie 6 : Chargement et stockage des données
9. Conclusion

# A propos

Apache Spark se présente comme la nouvelle génération de moteur de calcul distribué qui remplace progressivement Hadoop/MapReduce.

L'objet de ce Hands-on Labs est de vous familiariser par la pratique au traitement massif et distribué dans le domaine du data crunching et du machine learning. A l'issue de cette session, vous serez familiers avec :

- Les Resilient Data Sets (RDD) qui désignent l'abstraction essentielle pour la manipulation distribuée des données.
- les patterns de transformations et d'actions offerts par l'API
- les API de chargement et de stockage de données - filesystem / hdfs / NoSQL(Elasticsearch et Cassandra)
- Les bonnes pratiques de programmation distribuée avec la mise en oeuvre du partitionnement sélectif et l'usage de variables partagées (accumulators et broadcast variables)
- l'analyse et le reporting via Spark SQL
- l'analytique temps-réel avec Spark Streaming

Les prérequis à installer :

JDK 8 Distribution Spark et contenu du Hands-on Lab

# Les animateurs

Hayssam Saleh Twitter FROM EBIZNEXT

Apache Spark Certified Developer, Hayssam Saleh is Senior Architect with a focus on fault tolerant distributed systems and Web applications. He is currently the technical lead at EBIZNEXT where he supervises Scala / NoSQL projects since early 2012. He holds a PhD in distributed computing from Université Pierre et Marie Curie (Paris VI).

Blog: http://blog.ebiznext.com

Olivier Girardot Twitter FROM LATERAL THOUGHTS

Consultant et Associé fondateur de Lateral-Thoughts.com. Je suis développeur Java, Scala, Python et je m'intéresse de près aux problématiques des Moteurs de Recherche, du BigData, du Machine Learning et du NLP.

Blog: http://ogirardot.wordpress.com

# Partie 0 : Mise en place de l'environement

## Pré-requis

Pour bien commencer, il vous faut installer sur la machine :

- un JDK >= 1.8.x
- Maven >= 3.x
- SBT
- un IDE ou éditeur de texte : IntelliJ IDEA, Sublime Text, Eclipse...

## Industrialisation en Java

Nous allons créer un nouveau projet maven avec Spark Core:

```
mkdir hands-on-spark-java/
cd hands-on-spark-java/
```

Dans le `pom.xml` :

```
<?xml version="1.0" encoding="UTF-8"?>
<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/xs
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.devoxx.spark.lab</groupId>
    <artifactId>devoxx2015</artifactId>
    <version>1.0-SNAPSHOT</version>

    <repositories>
        <repository>
            <id>Apache Spark temp - Release Candidate repo</id>
            <url>https://repository.apache.org/content/repositories/orgapachespark-1080/<
        </repository>
    </repositories>
    <dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_2.10</artifactId>
            <version>1.3.1</version>
            <!--<scope>provided</scope>--><!-- cette partie là a été omise dans notre pro
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!-- we want JDK 1.8 source and binary compatiblility -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
```

```
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.2</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

## Industrialisation en Scala

Pour créer un projet Scala, il vous faut :

```
mkdir hands-on-spark-scala/
cd hands-on-spark-scala/
```

```
name := "Devoxx2015"

version := "1.0"

scalaVersion := "2.11.2"

crossScalaVersions := Seq(scalaVersion.value)


resolvers += "Local Maven Repository" at "file://" + Path.userHome.absolutePath + "/.m2/r

resolvers += "sonatype-oss" at "http://oss.sonatype.org/content/repositories/snapshots"

resolvers += "Typesafe Releases" at "http://repo.typesafe.com/typesafe/releases/"

resolvers += "Conjars" at "http://conjars.org/repo"

resolvers += "Clojars" at "http://clojars.org/repo"


val sparkV = "1.3.0"

libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % sparkV,
)

Seq(Revolver.settings: _*)

 test in assembly := {}

 assemblyOption in assembly := (assemblyOption in assembly).value.copy(includeScala = fal
```

Avec dans un fichier `project/assembly.sbt` contenant :

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.12.0")
```

Pour utiliser le plugin **SBT-Assembly** qui permet de générer via la commande `sbt assembly` un jar livrable pour Spark ne contenant ni Spark, ni Scala, mais toutes les autres dépendances nécessaires au projet.

## Récupération de Spark en tant que projet

Pour pouvoir soumettre le projet à un cluster en local, il vous faut télécharger ou récupérer par clé USB le binaire pré-compilé de Spark.

Si vous ne l'avez pas récupérer par une clé autour de vous, vous pouvez télécharger Spark 1.3.0 en cliquant ici !

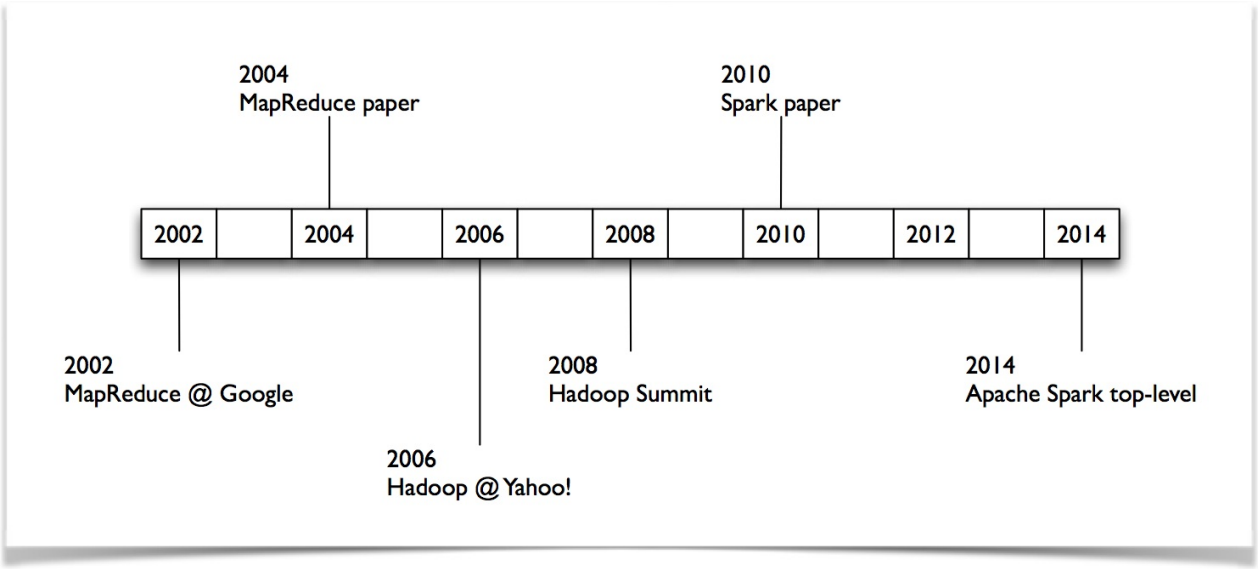
Pour installer à proprement dit Spark, il vous suffit de dézipper le fichier télécharger.

# Partie 1 : Familiarisation avec les concepts de Spark

Cette partie a pour but de vous familiariser avec les concepts principaux de Spark. Tout les exercices de cette partie seront en Java et s'appuieront sur Maven, mais vous pouvez aussi lancer un REPL Spark pour tester en live les concepts au fur et à mesure de la présentation.

## Historique

Pour comprendre Spark, il faut comprendre son contexte historique, Spark emerge près de 8 ans après les début du Map Reduce chez Google.




## Concepts

### RDD

L'abstraction de base de Spark est le RDD pour Resilient Distributed Dataset, c'est une structure de donnée immutable qui représente un Graph Acyclique Direct des différentes opérations à appliquer aux données chargées par Spark.

Un calcul distribué avec Spark commence toujours par un chargement de données via un **Base RDD**.

Plusieurs méthodes de chargements existent, mais pour résumé, tout ce qui peut être chargé par Hadoop peut être chargé par Spark, on s'appuie pour se faire sur un `SparkContext` ici nommé `sc` :

- En premier lieu, on peut charger dans Spark des données déjà chargées dans une JVM via `sc.paralellize(...)`, sinon on utilisera
- `sc.textFile("hdfs://...")` pour charger des données fichiers depuis HDFS ou le système de fichier local
- `sc.hadoopFile(...)` pour charger des données Hadoop
- `sc.sequenceFile(...)` pour charger des SequenceFile Hadoop

- Ou enfin `sc.objectFile(...)` pour charger des données sérialisées.

## Transformations et Actions

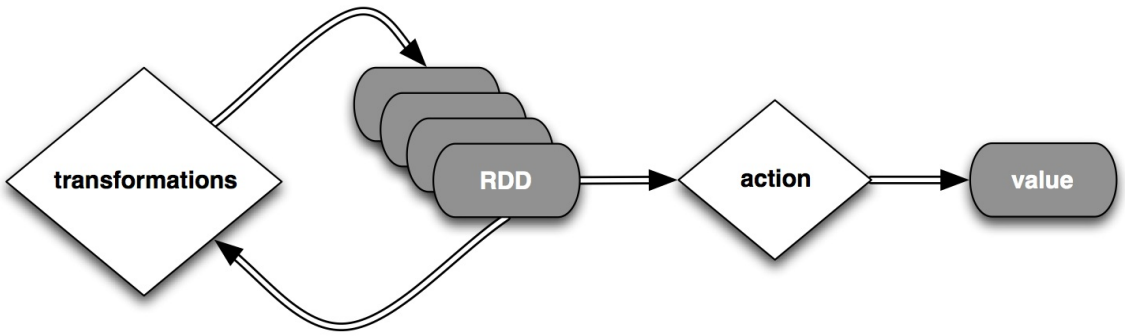
2 concepts de base s'appuient et s'appliquent sur le RDD,

- les Transformations
- les Actions

Les Transformations sont des actions **lazy** ou à évaluation paresseuse, elles ne vont lancer aucun calcul sur un Cluster.

De plus les RDD étant immutables, une transformations appliquée à un RDD ne va pas le modifier mais plutôt en **créer un nouveau enrichit de nouvelles informations correspondant à cette transformation**.

Une fois que vous avez définit toutes les transformations que vous voulez appliquer à votre donnée il suffira d'appliquer une Action pour lancer le calcul sur votre Cluster ou CPU locaux (selon le SparkContext utilisé *c.f. ci-après*)




Le RDD ne correspond en fait qu'à une sorte de plan d'exécution contenant toutes les informations de quelles opérations vont s'appliquer sur quelle bout ou partition de données.

## Spark Context

Le SparkContext est la couche d'abstraction qui permet à Spark de savoir où il va s'exécuter.

Un SparkContext standard sans paramètres correspond à l'exécution en local sur 1 CPU du code Spark qui va l'utiliser.

Voilà comment instantier un SparkContext en Scala :

```
val sc = SparkContext()
// on peut ensuite l'utiliser par exemple pour charger des fichiers :

val paralleLines : RDD[String] = sc.textFile("hdfs://mon-directory/*")
```

En Java pour plus de compatibilité, et même si on pourrait utiliser un `SparkContext` standard, on va privilégier l'utilisation d'un `JavaSparkContext` plus adapté pour manipuler des types Java.

```
public class SparkContextExample {
    public static void main(String[] args) {
        JavaSparkContext sc = new JavaSparkContext();

        JavaRDD<String> lines = sc.textFile("/home/devoxx/logs/201504*/*.txt");
    }
}
```

## Exercice : créer son premier RDD

En java :

```
public class FirstRDD {
    public static void main(String[] args) {
        JavaSparkContext sc = new JavaSparkContext();

        JavaRDD<String> lines = sc.textFile("/home/devoxx/logs/201504*/*.txt");
        // TODO rajouter une action sans quoi aucun calcul n'est lancé
    }
}
```

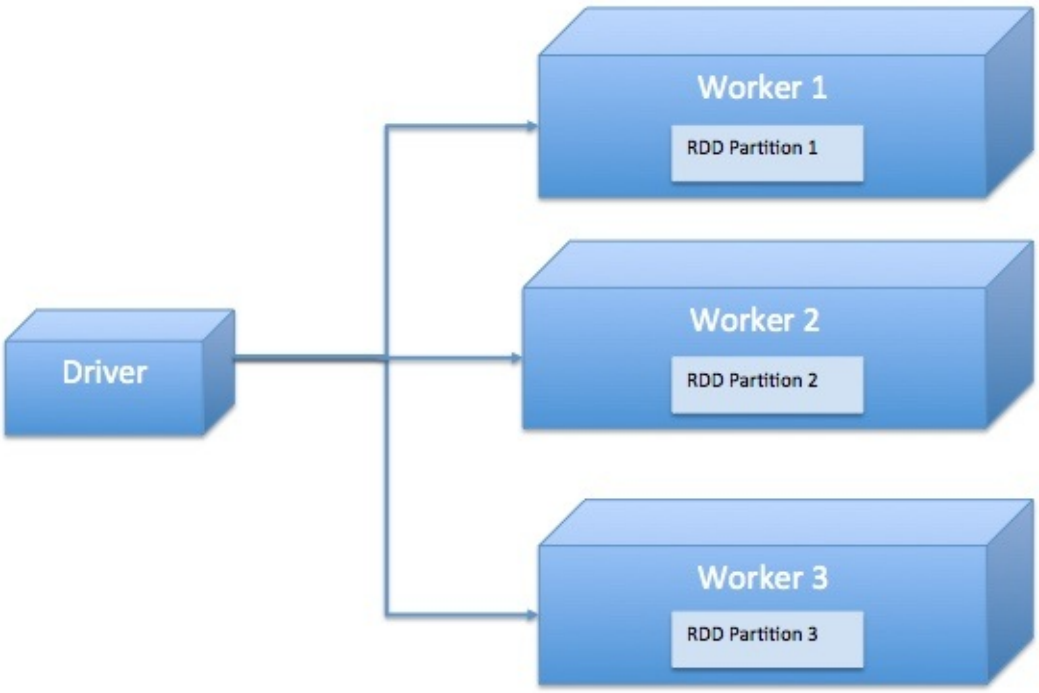
En Scala ou dans le REPL Spark vous pouvez aussi faire :

```
scala> sc.paralellize(1 to 1000).map( _ * 10)
// TODO rajouter une action sans quoi aucun calcul n'est lancé
```

## Exercice : soumettre un jar à spark-submit

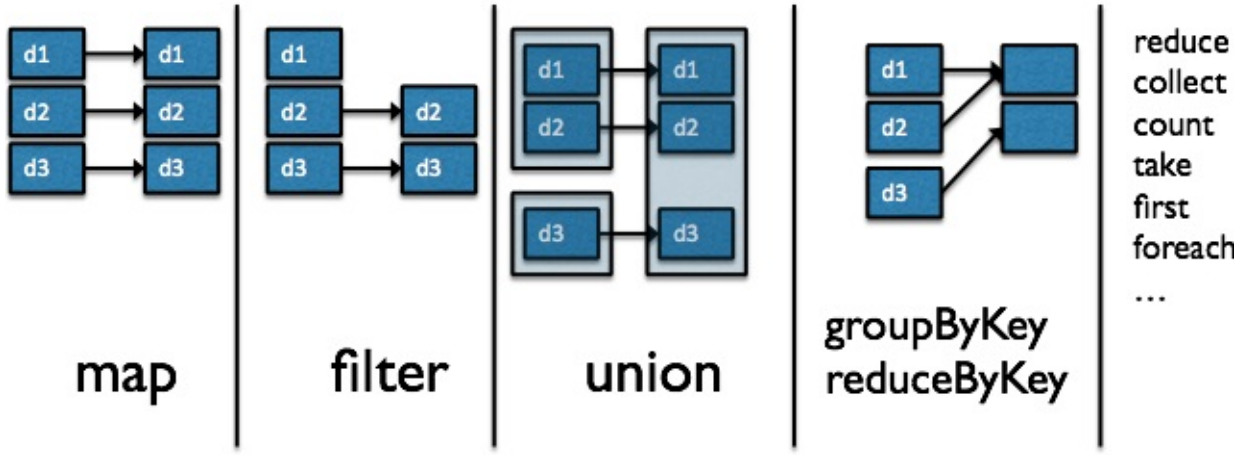
# Partie 2 : Mise en oeuvre des RDDs

Les RDDs sont une collection d'objets immuables répartis sur plusieurs noeuds d'un cluster. Un RDD est créé à partir d'un source de données ou d'une collection d'objets Scala / Python ou Java.




Les opérations disponibles sur un RDD sont :

- La création
- Les transformations
- L'action




Le RDD peut subir des transformations successives au travers de fonctions similaires à celles présentes dans les collections "classiques. on peut citer par exemple :

- `map` : renvoie un nouveau RDD avec application d'une fonction de transformation sur chacun des objets du RDD initial
- `filter` : renvoie un nouveau RDD qui contiendra un sous ensemble des données contenues dans le RDD initial.

Les opérations de création et de transformation de RDDs ne déclenchent aucun traitement sur les noeuds du cluster. Seul le driver est sollicité. Le driver va construire un graphe acyclique dirigé des opérations qui seront être exécutées sur le cluster au moment de l'application d'une action.

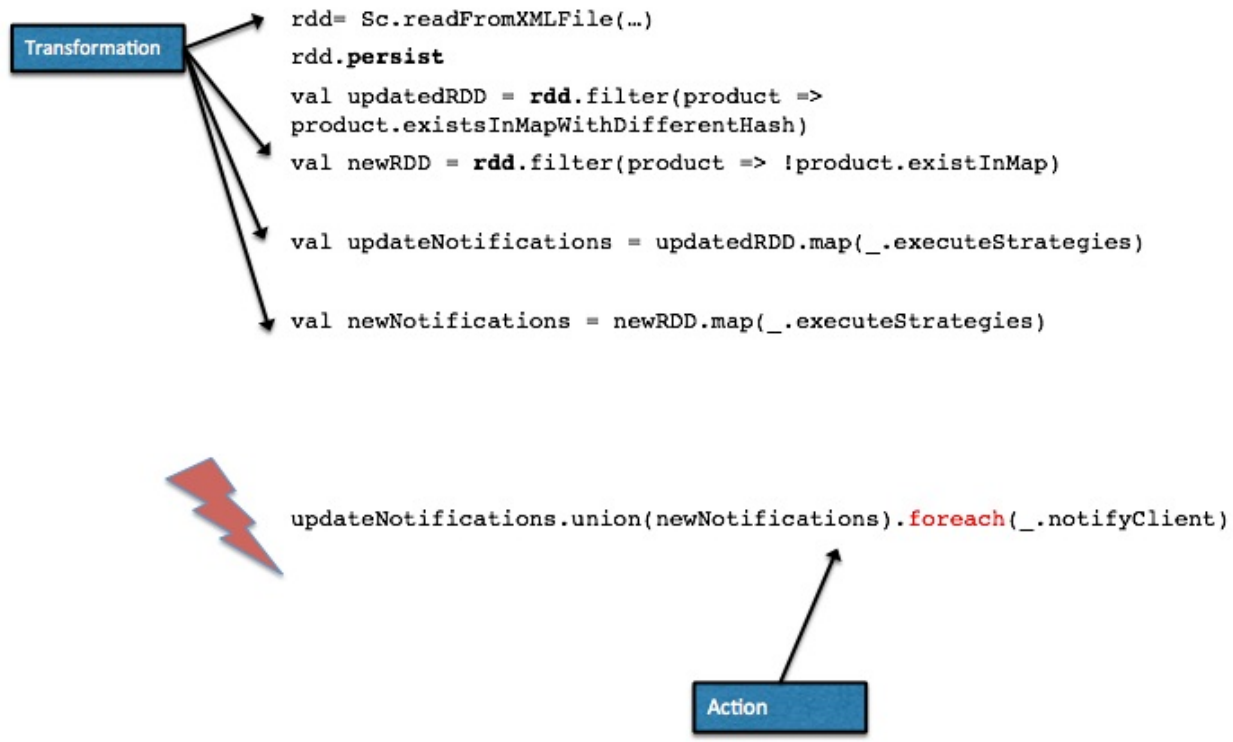
Le cycle de vie d'un RDD est donc le suivant :

```
1. rdd1 = création du RDD à partir d'une source de données ou d'une collection
2. rdd2 = rdd1.tranform1(fonctionDeTransformationAAppliquerSurChaqueElement)
3. rdd3 = rdd2.tranform2(fonctionDeTransformationAAppliquerSurChaqueElement)
4. ...
5. ...
6. ...
7. rddn = rddm.tranform3(fonctionDeTransformationAAppliquerSurChaqueElement)
8. objet = rddn.action
```

Une seule action peut être appliquée. Elle consiste à exécuter une opération sur tous les noeuds du cluster et à renvoyer le résultat au driver. L'action peut ne produire aucun résultat, l'action `foreach` par

exemple,ou produire un résultat qui soit un objet ou une collection. On peut citer en exemple :

- `reduce` qui renvoie un objet unique (Equivalent à Java reduce et Scala reduce)
- `take` qui renvoie les n premiers éléments d'un RDD (Equivalent à Java take et Scala take)




## Création d'un RDD

Un RDD peut être créé à partir d'une collection ou d'une source de données.

### Création à partir d'une collection

```
val rdd1 : RDD[Int] = sc.parallellize(List(1, 2, 3, 4))
val rdd2 : RDD[Int] = sc.parallellize(List("Hello", "Distributed", "World"))
val rdd3 : RDD[Int] = sc.parallellize(1 until 1000000)
```

### Création à partir d'une source de données

```
val lines1 : RDD[String] = sc.textFile("hdfs://....")
val lines2 : RDD[(String,String)] = sc.wholeTextFiles("file://path/")

// Création à partir d'une base de données JDBC
def connect() = {
  Class.forName("org.postgresql.Driver").newInstance()
  DriverManager.getConnection("jdbc:postgresql://localhost/mogobiz")
}
```

```
def readRecord(res : ResultSet) : (Long, String) = (res.getLong(1), res.getString(2))

val jdbcInputRDD = new JdbcRDD(sc, connect, "select age, name from Account offset ? limit
                               numPartitions = 10, mapRow = readRecord)
```

`textFile` et `wholeTextFile` permettent de créer un RDD à partir d'un fichier unique ou d'un répertoire. Dans ce deuxième cas, tous les fichiers présents dans le répertoire seront chargés.

- `textFile` va créer un RDD dont chaque élément correspond à une ligne du fichier
- `wholeTextFile` va créer un RDD dont chaque élément contiendra le nom du fichier (la clef: premier élément du tuple) et le contenu du fichier en entier (la valeur : deuxième élément du tuple). cette méthode est adaptée lorsqu'une ligne du fichier ne correspond pas à un enregistrement, cela concerne notamment les fichiers JSON et XML par exemple.

## Transformations d'un RDD

La ligne ci-dessous récupère exclusivement les noms des personnes majeures.

```
val namesRDD : RDD[String]= jdbcInputRDD.filter(tuple => tuple._1 >= 18).map(_._2)
```

Nous pouvons trier les utilisateurs par ordre croissant en utilisant la méthode `òrderBy``

```
val sortedNamesRDD = namesRDD.sortBy(x => x)
```

## Actions sur un RDD

Une fois l'action exécutée, l'ensemble des données résultant de cette action est rapatriée sur le driver. L'action est celle qui déclenche l'exécution sur les noeuds du cluster de la création, de la transformation, de l'action et du rapatriement des données issues des calculs sur le driver.

L'action ci-dessous récupère les 10 premiers noms

```
val firstNames : Array[String] = sortedNamesRDD.take(10)
```

L'exemple ci-dessus illustre bien que l'action conditionne le début et la fin de l'exécution du traitement sur les noeuds du cluster.

Un opération sur un RDD est une action lorsqu'elle ne renvoie pas de RDD mais qu'elle renvoie un objet ou une collection Scala standard.

Certaines actions peuvent ne rien renvoyer du tout, comme cela est le cas pour l'action `foreach`.

## Exercice 1 : Mise en oeuvre des RDDs

```
object Workshop1 {

  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("Workshop").setMaster("local[*]")
    // Fichier de rating au format userid\tmovie\trating\ttimestamp
    val url = Paths.get(getClass.getResource("/ratings.txt").toURI).toAbsolutePath.toStri
    val sc = new SparkContext(conf)

    // Charger le fichier de ratings dans un RDD
    // calculer la moyenne, le min, le max et le nombre d'éléments pour l'utilisateur ave
    // ...

    println( s"""
    count=$count
    min=$min
    mean=$mean
    max=$max
      """)
  }
}
```

## Le cache

Considérons l'exemple ci-dessous :

```
1. val lines = sc.textFile("...")
2. lines.count() // renvoie le nombre de lignes
3. lines.map(_.length).sortBy(x=> -x, ascending = false).first()
```

Dans l'exemple ci-dessus le fichier sera lu deux fois, une première fois à l'exécution de l'action `count` et une seconde fois à l'action `first` qui renvoie le premier élément du RDD (la ligne la plus longue).

```
// On rajoute l'appel à persist qui indique à Spark que le RDD ne doit pas être ``décharg
val lines = sc.textFile("...").persist(StorageLevel.MEMORY_AND_DISK)

lines.count() // renvoie le nombre de lignes
lines.map(_.length).sortBy(x=> -x, ascending = false)

// Le RDD est supprimé du cache.
lines.unpersist()
```

## Exercice 2 : Utilisation du cache

```
object Workshop2 {

  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("Workshop").setMaster("local[*]")
    val url = getClass.getResource("/ratings.txt").toString.replace("%20", " ")
```

```
    val sc = new SparkContext(conf)

    // Charger le fichier de ratings dans un RDD
    // Cacher le RDD de l'utilisateur avec l'id 200
    // calculer la moyenne, le min, le max et le nombre d'éléments pour l'utilisateur ave
    // Libérer du cache


    println( s"""
     count=$count
     min=$min
     mean=$mean
     max=$max
       """)

  }
}
```

## Les Pairs RDD

Spark offre des fonctionnalités spécifiques aux RDD clef-valeur - `RDD[(K,V)]`. Il s'agit notamment des fonctions `groupByKey, reduceByKey, mapValues, countByKey, cogroup`, https://spark.apache.org/docs/1.3.0/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions

### Exercice 3 : Utilisation des Pairs RDD

```
// Combien de fois chaque utilisateur a voté
 def main(args: Array[String]): Unit = {
   val conf = new SparkConf().setAppName("Workshop").setMaster("local[*]")
   val url = Paths.get(getClass.getResource("/ratings.txt").toURI).toAbsolutePath.toStri
   val sc = new SparkContext(conf)

   // Charger le fichier de ratings dans un RDD
   // ...
   // Créer un RDD clef/valeur [userid, rating]
   // ...
   // Mettre le RDD en cache
   // ...
   // Calculer le nombre de paires
   // ...
   println(s"Count=$count")
   // Calculer le nombre d'utilisateur distincts
   // ...
   println(s"UserCount=$userCount")

   // Calculer la somme des ratings / utilisateur
   // ...
   // Calculer le nombre de ratings par utilisateur
   // ...
   // Calculer la plus faible note donnée par chaque utilisateur
   // ...
   // Calculer la plus forte note donnée par chaque utilisateur
   // Mettre l'ensemble des RDDs dans un RDD unique (union) sans écraser les valeurs. (a
```

```
    // ...
    // Indice : Modifier les transformations précédentes pour avoir un tuple avec une seu
    // ...
    // Réaliser une variante en utilisant cogroup
    // ...


    // ----> Ne fonctionnera pas
    //    cachedRDD.map { userRatings =>
    //      val user = userRatings._1
    //      val ratings = sc.makeRDD(userRatings._2)
    //      ratings.map(_.rating).mean()
    //    }

    allRDDs.foreach { case (x, y) =>
      println( s"""
     user=$x
     count=${y._4}
     min=${y._1}
     mean=${y._2}
     max=${y._3}
       """)
    }
  }
}
```

# Partie 3 : Aspects avancés de Spark

Tous les programmes Spark ne sont pas égaux. Spark offre plusieurs mécanismes d'amélioration des performances dont :

- Les accumulateurs
- Le broadcast
- Le partitionnement

## La sérialisation

L'option `MEMORY_AND_DISK` indique que Spark peut persister les données sur le disque si la mémoire est insuffisante. D'autres options sont disponibles et permettent par exemple de réduire l'empreinte mémoire des objets (au détriment de la CPU toutefois).

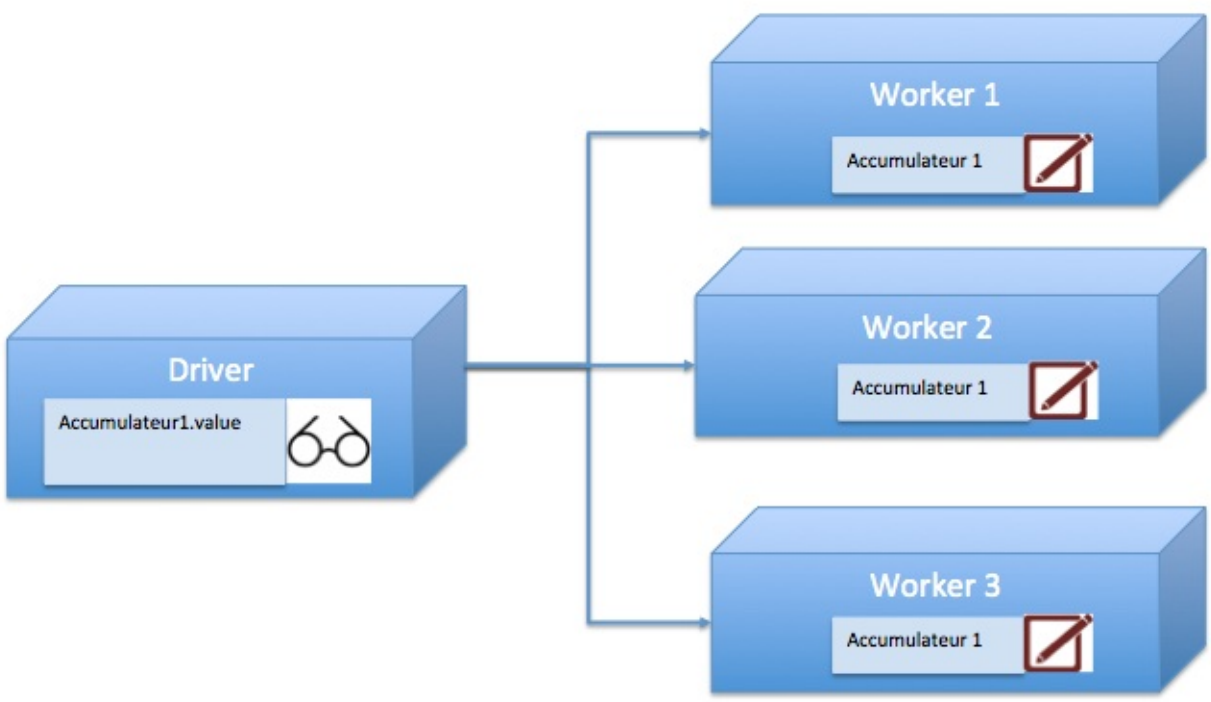
Si les opérations requièrent un nombre important d'échange entre les noeuds du cluster, un algorithme de sérialisation performant en temps d'exécution et capable de générer une représentation plus compacte peut avoir un impact positif sur la performance globale.

Spark permet de personnaliser le librairie de sérialisation. L'exemple ci-dessus illustre la mise en oeuvre de la dé/sérialisation avec Kryo.

```
val conf = new SparkConf()
conf.set("spark.serializer", ""org.apache.spark.serializer.KryoSerializer)
conf.set("spark.kryo.registrationRequired", "true")
conf.registerKryoClasses(Array(classOf[Class1], classOf[Class2], classOf[Class3]))
```

# Les accumulateurs

Consiste à définir une variable qui sera mise à jour par l'ensemble des noeuds du cluster lors de l'exécution de l'action. Cette variable pourra ensuite être lue sur le driver.




Un accumulateur ne doit jamais être mis à jour dans le cadre d'une transformation. Spark étant susceptible de rexécuter les transformations en cas de panne, cela résulterait à une double mise à jour de la valeur de l'accumulateur.

```
val accumSum: Accumulator[Int] = sc.accumulator(0)
val accumCount: Accumulator[Int] = sc.accumulator(0)
val rdd = sc.parallelize(1 until 10000)
rdd.foreach { x =>
  accumSum += x
  accumCount += 1
}
println("mean of the sum=" + accumSum.value / accumCount.value)
```

Un accumulateur est mis à jour par les noeuds du cluster et consulté au niveau du driver.

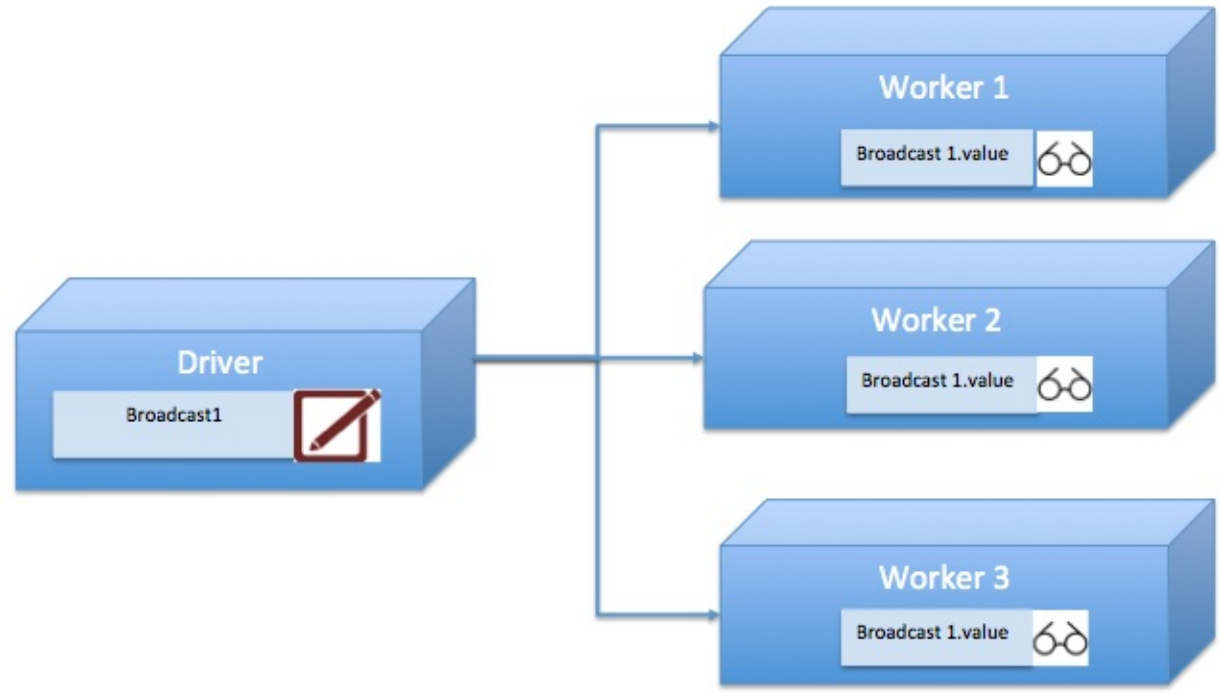
## Le broadcast

Le broadcast permet diffuser sur tous les neouds du cluster, des données de référence qui pourront être accédées par les transformations et l'action lors de l'exécution.

La variable de broadcast héberge en général des données de référence.

```
val labels: Broadcast[Map[Int, String]] = sc.broadcast(Map(1 -> "Un", 2->"2", 3 -> "3"))
val rdd = sc.parallelize(1 to 3)
```

```
rdd.foreach { x =>
    // Affiche les bales correspondant aux identifiants numériques
    // en s'appuyant sur les labels diffusés sur chacun des noeuds
    println(labels.value(x))
}
```




## Le partitionnement

Lorsqu'un RDD est utilisé plusieurs fois de suite, dans des actions différentes et donc dans des cycles distincts d'exécution du RDD, Spark est amené à recalculer le partitionnement des données sur le cluster. Cela peut avoir un effet négatif sur les performances.

Dans un système distribué les communications sont coûteuses, il est donc impératif de les limiter en renforçant la colocalisation des données. Spark permet d'indiquer la règle de répartition des données sur les partitions, par contre cela a un intérêt uniquement dans le cas des opérations sur les clefs dans le cadre des RDDs clefs/valeurs. Cela concerne notamment les opérations de type `join`, `cogroup`, `reduceByKey`

Le résultat du partitionnement doit systématiquement être mis en cache si l'on ne veut pas que les appels subséquents conduisent à un nouveau partitionnement du RDD.

Les transformations susceptibles de modifier la clef (tel que `map`) conduisent Spark à reconstruire le partitionnement du RDD parent.

L'exemple ci-dessous illustre l'utilisation du partitionnement

```
val lines : RDD[(Text, Int)] = sc.sequenceFile[String, Int]("/path").partitionBy(new Hash
val other = lines.mapValues(_ + 1)
lines.join(other)
```

Un fichier Hadoop est chargé dans un RDD en utilisant un partitionnement que l'on a pris le soin de mettre en cache.

Un nouvel RDD est créé mais en maintenant les clefs colocalisés sur les mêmes noeuds que le RDD parent.

Le `join` est réalisé avec une performance optimale.

## Repartitionnement

Suite à des opérations de transformation réduisant de manière significative la taille du RDD, il peut être utile de réduire le nombre de paritions. C'est ce que permet la transformation `coalesce`

```
lines.filter(x => x._2 < 3).coalesce(3)
```

## Exercice 4 : La colocalité

```
object Workshop4 {

  // Combien de fois chaque utilisateur a voté
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("Workshop").setMaster("local[*]")
    val url = Paths.get(getClass.getResource("/ratings.txt").toURI).toAbsolutePath.toStri
    val sc = new SparkContext(conf)

    val lines: RDD[Rating] = ...

    val cachedRDD: RDD[(Long, List[Int])] = ... // key = userid, value = list of rating v
    val count = cachedRDD.count()
    println(s"usercount=$count")

    val allRDDs: RDD[(Long, (Int, Int, Int, Int))] = ...
    // Utiliser mapPartition pour calculer par utilisateur la moyenne, le min, le max et
    // Noter que le sopérations sotn faites localement sans reshuffling.


    allRDDs.filter(_._1 == 315).foreach { case (x, y) =>
      println( s"""
     user=$x
     count=${y._4}
     min=${y._1}
     mean=${y._2}
     max=${y._3}
       """)
```

```
    }
  }
}
```

## Exercice 5 : Les accumulateurs

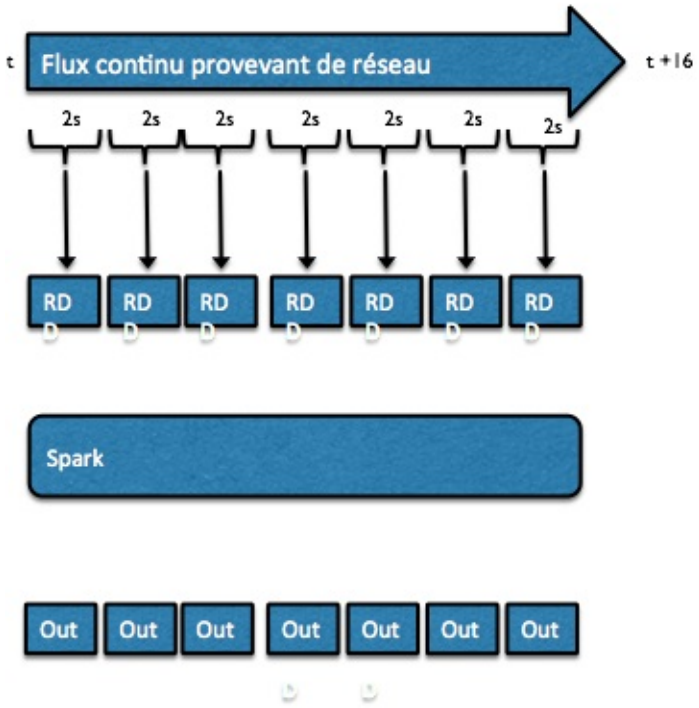
```
object Workshop5 {

  // Combien de fois chaque utilisateur a voté
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("Workshop").setMaster("local[*]")
    val url = Paths.get(getClass.getResource("/ratings.txt").toURI).toAbsolutePath.toStri
    val urlProducts = Paths.get(getClass.getResource("/products.txt").toURI).toAbsolutePa
    val sc = new SparkContext(conf)


    val lines: RDD[Rating] = ...
    val cachedRDD: RDD[(Long, List[Int])] = ...
    val count = cachedRDD.count()
    println(s"usercount=$count")

    val allRDDs: RDD[(Long, (Int, Int, Int, Int))] = ...

    // Calculer la moyenne globale
    val accumSum: Accumulator[Int] = sc.accumulator(0)
    val accumCount: Accumulator[Int] = sc.accumulator(0)
    //...
    println(s"global mean = ${globalMean}")
    allRDDs.filter(_._1 == 315).foreach { case (x, y) =>
      println( s"""
     user=$x
     count=${y._4}
     min=${y._1}
     mean=${y._2}
     max=${y._3}
       """)
    }
  }
}
```

## Exercice 6 : Le broadcast

```
object Workshop6 {

  // Combien de fois chaque utilisateur a voté
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("Workshop").setMaster("local[*]")
    val url = Paths.get(getClass.getResource("/ratings.txt").toURI).toAbsolutePath.toStri
    // Fichier de produits au format id\tLibellé
    val urlProducts = Paths.get(getClass.getResource("/products.txt").toURI).toAbsolutePa
    val sc = new SparkContext(conf)
```

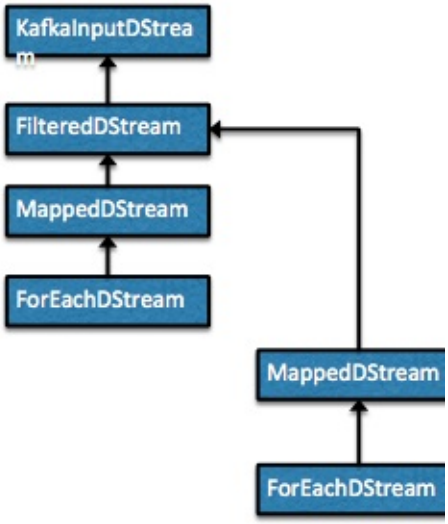
```
      // Récupérer la liste des produits sous forme de Map[idProduit:Long, Libelle:String]
      val products: Map[Long, String] = // ...
      val labels: Broadcast[Map[Long, String]] = // ...


      val lines: RDD[Rating] = // ...
      // Imprimer le libellé des films qui ont un rating de 5
      // ...
    }
  }
```

# Partie 4 : Spark Streaming




- Découpage du flux en un paquet de données (RDD) toutes les 2 secondes
- Un RDD est une collection de données
- Chaque paquet est traité comme un comme un RDD soumis au batch Spark classique
- Exécution de microbatchs
- Spark exécute les opérations sur le RDD comme dans un batch classique et renvoie le résultat en retour.

```
val tuples = kafka.initStream(context,"requests ", 2
seconds

rdd = tuples.filter( t => RulesHandler.match(t))

rdd2 = rdd.map( t => toDocument)

rdd2.foreachRDD(doc => es.insert doc) // via Kafka

rdd3 = rdd.flatMap(_.toRelations)

rdd3.foreachRDD(rel => neo4j.insert rel) // via Kafka
```




- Le code ci-dessus ne génère aucun calcul, juste un graphe d'objets
- Le Scheduler Spark va soumettre le graphe d'objets aux workers pour exécution
- ☺ En cas de panne, il suffit juste de soumettre à nouveau le graphe d'objets à un autre worker.
  - Pas besoin de réplication des résultats ou d'upstream backup

Spark Streaming offre la reprise sur erreur vie la méthode `checkpoint`. En cas d'erreur, Spark Streaming repartira du dernier checkpoint qui devra avoir été fait sur un filesystem fiable (tel que HDFS). Le checkpoint doit être fait périodiquement. La fréquence de sauvegarde a un impact direct sur la performance. En moyenne on sauvegardera tous les 10 microbatchs.

Les opérations sur les DStreams sont stateless, d'un batch à l'autre, le contexte est perdu. Spark Streaming offre deux méthodes qui permettent un traitement stateful : `reduceByWindow` et `reduceByKeyAndWindow` qui vont conserver les valeurs antérieures et permettent de travailler sur une fenêtre de temps multiples de la fréquence de batch paramétrée.

Dans l'exemple ci-dessous, on affiche toutes les 20s le nombre de mots enovyés dans la minute.

```
object Workshop {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("Workshop").setMaster("local[*]")
    val ssc = new StreamingContext(conf, Seconds(1))
    ssc.checkpoint("/tmp/spark")
    val lines: ReceiverInputDStream[String] = ssc.socketTextStream("localhost", 7777)
    val res: DStream[Int] = lines.flatMap(x => x.split(" ")).map(x => 1).reduceByWindow(_
    res.checkpoint(Seconds(100))
    res.foreachRDD(rdd => rdd.foreach(println))
    res.print()

//    lines.print()
    ssc.start()
    ssc.awaitTermination()
```

```
  }
}
```

Pour tester le streaming, il nous faut un serveur de test. Sur macos, il suffit de lancer `ncat` avec la commande `nc` et taper des caractères en ligne de commande qui seront alors retransmis sur le port d'écoute ncat :

```
$ nc -l localhost 7777
Hello World ++
Hello World ++
Hello World ++
Hello World ++
^C
```

Sur Windows, `ncat` peut être téléchargé sur nmap.org

## Exercice 7 : Opérations stateful avec Spark Streaming

Objectif : Compter le nombre d'occurrences de chaque mot par période de 30 secondes object Workshop7 { def main(args: Array[String]): Unit = { val conf = new SparkConf().setAppName("Workshop").setMaster("local[*]") val ssc = new StreamingContext(conf, Seconds(1)) ssc.checkpoint("/tmp/spark") val lines: ReceiverInputDStream[String] = ssc.socketTextStream("localhost", 7777)

```
// Compter le nombre d'occurrences de chaque mot par période de 30 secondes
val res: DStream[(String, Int)] = ...

res.checkpoint(Seconds(100))
res.foreachRDD(rdd => rdd.foreach(println))
res.print()
```

// lines.print() ssc.start() ssc.awaitTermination() } }

# Partie 5 : SparkSQL

Cette partie a pour but de vous faire prendre en main les APIs Spark SQL qui, à la différence de Spark Core, permet de gérer des flux de données **structurées** (avec un Schéma explicite ou définissable)

## Théorie

L'abstraction principale de Spark, les RDDs, n'ont aucune notion de schéma, on peut gérer des données sans aucune information de typage ou de noms de champs.

Pour permettre de gérer une notion de schéma (une liste de noms de champs avec leur type associé) Spark SQL a introduit dans les versions < 1.3, la notion de `SchemaRDD` qui est un `RDD` standard mais avec cette notion de schéma et donc des méthodes additionnelles.

Avec la nouvelle release de Spark 1.3.0, ce concept a été généralisé et renommé sous le nom de `DataFrame`.

## Exercice 8 : Appliquer un schéma à un RDD existant par *reflection*

Quand vous avez déjà un RDD avec des données structurées - une classe dédié comme ci-après la classe `Movie`, vous pouvez appliquer un schéma et transformer le flux en **DataFrame**

Dans cet exercice nous allons parser un flux non-structuré via un RDD standard, le transformer en un `RDD<Movie>` pour enfin lui appliquer un schéma par reflection !

Tout d'abord voilà la classe du modèle :

```
package org.devoxx.spark.lab.devoxx2015.model;

public class Movie implements Serializable {

    public Long movieId;
    public String name;

    public Product(Long movieId, String name) {
        this.movieId = movieId;
        this.name = name;
    }
}
```

Puis le code de travail Spark :

```
public class Workshop8_Dataframe {

    public static void main(String[] args) throws URISyntaxException {
        JavaSparkContext sc = new JavaSparkContext("local", "Dataframes");
```

```
        String path = Paths.get(Workshop8_Dataframe.class.getResource("/products.txt").to

        JavaRDD<Movie> moviesRdd = sc.textFile(path)
                .map(line -> line.split("\\t"))
                .map(fragments -> new Movie(Long.parseLong(fragments[0]), fragments[1]));

        SQLContext sqlContext = new SQLContext(sc);

        // reflection FTW !
        DataFrame df = // TODO utiliser Movie.class pour appliquer un schema

        df.printSchema();
        df.show(); // show statistics on the DataFrame
    }
}
```

En Scala dans **Workshop8.scala** :

```
case class Movie(movieId: Long, name: String)

val moviesRDD = sc.textFile("products.txt").map(line => {
    val fragments = line.split("\\t")
    Movie(fragments(0).toLong, fragments(1))
})

val df = ...;// TODO utiliser Movie.class pour appliquer un schema

df.show(); // show statistics on the DataFrame
```

## Exercice 9 : Charger un fichier Json

```
public class SchemaRddHandsOn {

    public static void main(String[] args) {
        JavaSparkContext sc = new JavaSparkContext();
        SQLContext sqlContext = new org.apache.spark.sql.SQLContext(sc);

        DataFrame df = ...// Charger le fichier products.json avec l'API sc.load(...)

        df.show(); // show statistics on the DataFrame
    }
}
```

En Scala dans **Workshop9.scala** :

```
object Workshop9 {

    def main(args: Array[String]): Unit = {
        val conf = new SparkConf().setAppName("Workshop").setMaster("local[*]")
        val url = Paths.get(getClass.getResource("/ratings.txt").toURI).toAbsolutePath.to
```

```
        val sc = new SparkContext(conf)

        val sqlContext: SQLContext = new org.apache.spark.sql.SQLContext(sc)
        import sqlContext.implicits._

        val df = ...// Charger un fichier json avec l'API sqlContext.load(...) --> DataFr
        df.show() // show statistics on the DataFrame
    }
```

## Quelques opérations sur une DataFrame

La nouvelle API des `DataFrame` a été calquée sur l'API de Pandas en Python. Sauf que bien sûr a la différence de Pandas, les `DataFrame` sont parallélisées.

```
df.printSchema(); // affiche le schéma en tant qu'arbre
df.select("name").show(); // n'affiche que la colonne name
df.select("name", df.col("id")).show(); // n'affiche que le nom et l'id
df.filter(df("id") > 900).show();
df.groupBy("product_id").count().show();

// chargement :
sqlContext.load("people.parquet"); // parquet par default sauf si overrider par spark.sql
sqlContext.load("people.json", "json");
```

## Exercice 10 : SQL et tempTable(s)

Dans cet exercice le but va être d'utiliser notre schéma ainsi définit dans le contexte Spark SQL pour executer via Spark des requêtes SQL

La première requête va être de : **Trouver tout les films dont le nom contient Hook**

```
public class SchemaRddHandsOn {

    public static void main(String[] args) {
        JavaSparkContext sc = new JavaSparkContext();
        SQLContext sqlContext = new org.apache.spark.sql.SQLContext(sc);

        DataFrame df = sqlContext.load("products.json", "json");

        // On met à disposition en tant que table la DataFrame
        df.registerTempTable("movies");

        // ceci est une transformation !
        DataFrame hooks = sqlContext.sql(...); // TODO ajouter la requête SQL

        hooks.show();
    }
}
```

```
object Workshop10 {

  // Combien de fois chaque utilisateur a voté
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("Workshop").setMaster("local[*]")
    val url = Paths.get(getClass.getResource("/ratings.txt").toURI).toAbsolutePath.toStri
    val sc = new SparkContext(conf)

    val sqlContext: SQLContext = new org.apache.spark.sql.SQLContext(sc)
    import sqlContext.implicits._

    val df = sqlContext.load("products.json", "json");
    df.registerTempTable("movies");
    // ceci est une transformation !
    DataFrame hooks = sqlContext.sql(...); // TODO ajouter la requête SQL

    hooks.show()
  }
}
```

## Exercice 11 : de RDD vers DataFrame

```
object Workshop11 {

  // Combien de fois chaque utilisateur a voté
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("Workshop").setMaster("local[*]")
    val url = Paths.get(getClass.getResource("/ratings.txt").toURI).toAbsolutePath.toStri
    val sc = new SparkContext(conf)

    val sqlContext: SQLContext = new org.apache.spark.sql.SQLContext(sc)
    import sqlContext.implicits._

    val lines: RDD[Rating] = // ...

    val df = // Convertir le RDD en DataFrame
    val maxRatings: DataFrame = // renvoyer les utilisateurs ayant noté 5
    val sqlCount = // Combien y a t-il eu de ratings ?
    println(sqlCount.head().getLong(0))

    // TOP 10 utilisateurs par nombre de votes et les imprimer
    val sqlTop10 = //
  }
}
```

# Partie 6 : Chargement et stockage des données

## Elasticsearch

ElasticSearch expose une interfe d'entrée sortie permettant de récupérer les données au format RDD et de sauvegarder les RDDs directement en JSON.

### Ajout d'Elasticsearch au contexte Spark

```
    import org.apache.spark.{SparkConf, SparkContext}
     val conf = new SparkConf().setAppName(Settings.Spark.AppName).setMaster(Settings.Spar
     conf.set("es.nodes", "localhost")
     conf.set("es.port", "9200")
     val sc = new SparkContext(conf)
```

### Lecture des données dans Elasticsearch

```
     val esWildcardQuery =  search in "myindex" -> "rating" query { matchall }

     // val ratings = sc.esRDD("myindex/rating")

     val allData: RDD[(String, Map[String, AnyRef])] = sc.esRDD("myindex/rating", "{'match
```

### Sauvegarde des données dans Elasticsearch

```
     allData.filter(_.rating < 5).saveToEs("myindex/rating")
```

## Cassandra - Connecteur

Cassandra possède maintenant son connecteur spécifique et avec l'introduction des `DataFrame` et du calcul de plan d'exécution qui vient avec a été codifié le **predictive pushdown**. TODO

```
<dependency>
    <groupId>com.datastax.spark</groupId>
    <artifactId>spark-cassandra-connector_2.10</artifactId>
    <version>1.2.0-rc3</version>
</dependency>
<dependency>
    <groupId>com.datastax.spark</groupId>
    <artifactId>spark-cassandra-connector-java_2.10</artifactId>
    <version>1.2.0-rc3</version>
</dependency>
```

ou en Scala en utilisant SBT :

```
libraryDependencies ++= Seq(
"com.datastax.spark" %% "spark-cassandra-connector" % "1.2.0-rc3"
)
```

Ensuite pour charger une table Cassandra dans Spark :

```
import static com.datastax.spark.connector.japi.CassandraJavaUtil.*;

public class CassandraTest{

    public static void main(String[] args) {
        JavaRDD<Double> pricesRDD = javaFunctions(sc).cassandraTable("ks", "tab", mapColu
    }
}
```

et en Scala

```
val conf = SparkConf(true)
    .set("spark.cassandra.connection.host", "127.0.0.1")

val sc = SparkContext("local[*]", "test", conf)

import com.datastax.spark.connector._

val rdd = sc.cassandraTable("test", "kv")

// ou encore :
val collection = sc.parallelize(Seq(("key3", 3), ("key4", 4)))

collection.saveToCassandra("test", "kv", SomeColumns("key", "value"))
```

# Conclusion

Voilà !