Cours
Big data Analytics

Cycle Ingénieur
INDIA, Semestre 5

Pr. Abderrahim El Qadi
Département Mathématique Appliquée et Génie Informatique
ENSAM, Université Mohammed V de Rabat

A.U. 2023/2024

Plan

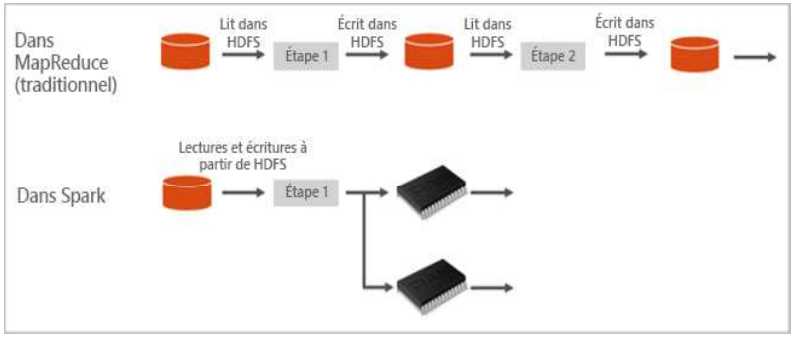
1ere partie

Ressources & Références 1ere partie

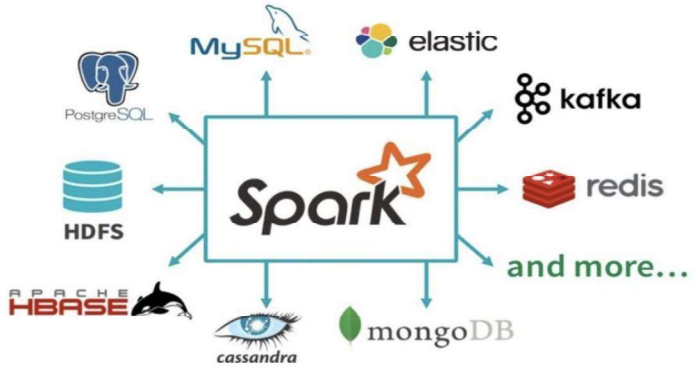
- Big Data Analytics with Spark A Practitioner's Guide to Using Spark for Large Scale Data Analysis. Mohammed Guller. 2015.
- Machine Learning with PySpark with Natural Language Processing and Recommender Systems. Pramod Singh. 2018
- Practical Apache Spark. Subhashini Chellappan, Dharanitharan Ganesan, Apress 2018
- Initiation a Spark avec Java et Scala. Olivier Girardot
- https://www.data-transitionnumerique.com
- https://sparkbyexamples.com/
- Apache Spark™ - Moteur unifié pour l'analyse de données à grande échelle.
- https://liliasfaxi.github.io/Atelier-Spark/p2-spark/
- https://spark.apache.org/docs/latest/cluster-overview.html
- https://data-flair.training/blogs/apache-spark-cluster-managers-tutorial
- https://learn.microsoft.com/fr-fr/azure/synapse-analytics/spark/apache-spark-overview

1. Apache Spark

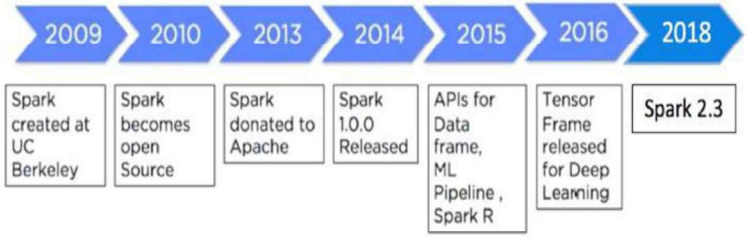
– Il est le projet open source le plus actif dans le monde du Big Data.
– Spark peut charger et mettre en cache des données en mémoire et les interroger à plusieurs reprises. Le calcul en mémoire est beaucoup plus rapide que les applications sur disque.



– Apache Spark est une plateforme de traitement sur cluster générique.
  - permet de réaliser des traitements par lot (*batch processing*) ou à la volée (*stream processing*) et est conçu de façon à pouvoir intégrer tous les outils et technologies Big Data.

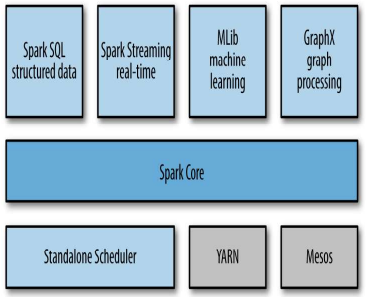


– Spark a commencé comme un projet de recherche à l'UC Berkeley AMPLab en 2009 et était open source au début de 2010
– Il offre des APIs de haut niveau en Java, Scala, Python et R.
– Il fournit une API de développement pour permettre un traitement en streaming, l'apprentissage automatique ou la gestion de requêtes SQL et demandant des accès répétés sur un grand volume de données.



– Il est né à la base pour pallier les problèmes posés par Hadoop Map Reduce, mais est devenu une entité à lui seul, offrant bien plus que le traitement par lot classique.

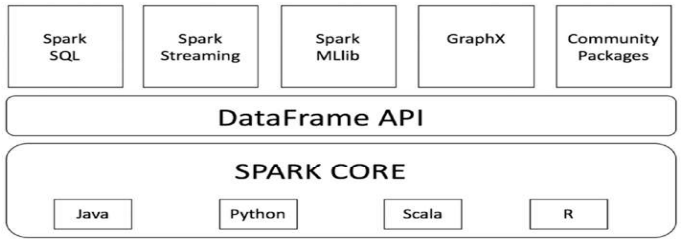
1.1 Apache Spark – Composants



1. **Spark Core** est le point central de Spark, fournit une plateforme d'exécution pour toutes les applications Spark.
2. **Spark SQL** se situe au-dessus de Spark, permettre aux utilisateurs d'utiliser des requêtes SQL/HQL.
3. **Spark Streaming** permet de créer des applications d'analyse de données interactives. Les flux de données sont transformés en micro-lots et traités par-dessus Spark Core.
4. **Spark MLlib** fournit des librairies de ML, très utiles pour les data scientists, autorisant de plus des traitements en mémoire améliorant de façon drastique la performance de ces algorithmes sur des données massives.
5. **Spark GraphX** est le moteur d'exécution permettant un traitement scalable utilisant les graphes, se basant sur Spark Core.

Spark Core

– C'est le bloc de construction le plus fondamental du Spark,
– Toutes les fonctionnalités de Spark sont construites sur Spark Core.
– Spark Core est responsable de la gestion des tâches, des opérations d'E/S, de la tolérance aux pannes et de la gestion de la mémoire, etc.



– Spark peut facilement être utilisé avec diverses sources de données telles que :
  - HBase, Cassandra, Amazon S3, HDFS, etc.
  - Fournit aux utilisateurs quatre options du langage à utiliser : Java, Python, Scala et R.

1.2 Principales caractéristiques

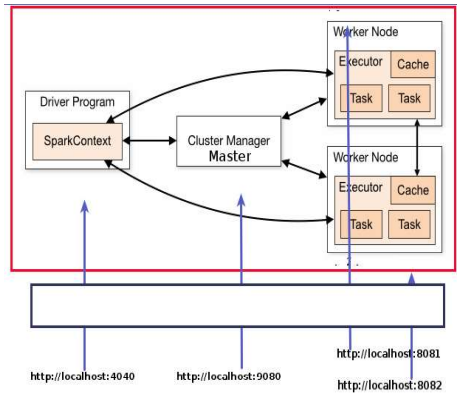
– Spark ressemble à Hadoop MapReduce pour le traitement de données volumineuses.
– Il propose de nombreux avantages par rapport à Hadoop MapReduce.
  - Offre des **API** riches pour le développement d'applications Big Data ; plus de 80 opérateurs de traitement de données.
  - Permet d'écrire **un code plus concis** par rapport à Hadoop MapReduce, qui nécessite beaucoup de code.
  - Est **plus rapide** que Hadoop MapReduce pour deux raisons :
    - Le calcul est en mémoire, les données peuvent être lues depuis la mémoire 100 fois plus rapidement que depuis le disque
    - Implémente un moteur d'exécution avancé (10 fois plus rapide lors de l'exécution sur le disque).
  - Spark est :
    - Evolutif (peut ajouter plus de nœuds si nécessaire)
    - Flexible (écosystème Hadoop facile à utiliser)
    - Tolérant aux pannes (si un nœud tombe en panne, le système continue de fonctionner)

– Limites du Spark

En tant que système de traitement en mémoire, le coût d'exécuter Spark sur un cluster peut être très élevé en terme de consommation mémoire.
Pas de système de gestion des fichiers : Spark est principalement un système de traitement, et ne fournit pas de solution pour le stockage des données. Il doit donc se baser sur d'autres systèmes de stockage tel que Hadoop HDFS ou Amazon S3.
Problèmes avec les fichiers de petite taille : Spark partitionne le traitement sur plusieurs exécuteurs, et est optimisé principalement pour les grands volumes de données. L'utiliser pour des fichiers de petite taille va rajouter un coût supplémentaire, il est donc plus judicieux dans ce cas d'utiliser un traitement séquentiel classique sur une seule machine

1.3 Fonctionnement du Spark

– Chaque application Spark utilise un programme *driver* qui sert à lancer des calculs distribués sur un cluster.
– Un nœud coordinateur (appelé aussi master) lancera ces opérations sur deux nœuds Worker qui hébergeront un/plusieurs exécuteur(s).

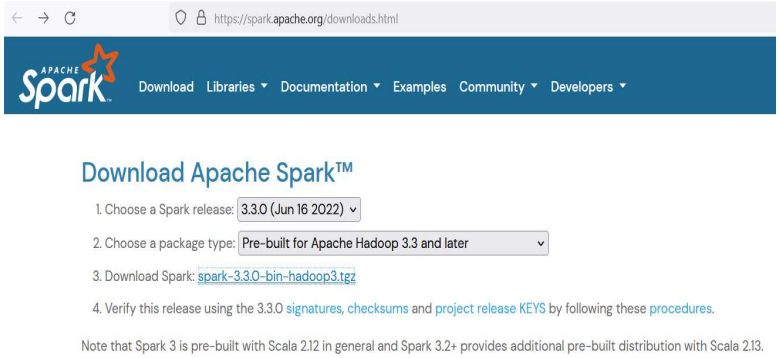


- Le code pour exécuter l'un de ces activités est d'abord écrit sur **Spark Driver,** et ensuite partagé entre les « **worker nodes** » où réside réellement les données.
- Chaque nœud de travail contient des exécuteurs qui exécutera réellement le code.
- **Cluster Manager** surveille la disponibilité de divers nœuds de travail pour la prochaine attribution de tâche.

1.4 Analyse de données interactive avec Spark Shell

– Le Shell Spark est fourni avec Spark.

– Télécharger Spark à partir du site Web suivant : http://spark.apache.org/downloads.html



– Exécution :

```
cd $SPARK_HOME
./bin/spark-shell
```

– Le shell Spark crée et met à disposition une instance de la classe **SparkContext** en tant que sc.

– Les applications Spark s'exécutent en tant qu'ensembles indépendants de processus sur un cluster, coordonnés par l'objet du programme principal (appelé *programme pilote*).SparkContext

2. Langage Scala

- est un excellent langage pour développer des applications Big Data.
- est un langage de programmation hybride orienté objet et fonctionnel, il met l'accent sur la programmation fonctionnelle.
- est un langage basé sur une machine virtuelle Java (JVM).
  - Le compilateur Scala compile une application en bytecode Java, qui s'exécutera sur n'importe quelle JVM.
  - Au niveau du bytecode, une application Scala est impossible à distinguer d'une application Java.
- Étant donné que Scala est basé sur JVM, il est parfaitement interopérable avec Java.
  - Une bibliothèque Scala peut être facilement utilisée depuis une application Java.
  - Une application Scala peut utiliser n'importe quelle bibliothèque Java sans aucun wrapper ou code de colle.

– Installation de Scala (www.scala-lang.org/download)

Plusieurs solutions sont alors à disposition pour utiliser Scala :

- Des éditeurs de texte, le compiler avec scalac et l'exécuter avec scala.
- Des IDE : comme Eclipsed-based Scala IDE, IntelliJ IDEA, ou NetBeans IDE

2.1 Variables et Types

– Scala a deux types de variables : **modifiables et non modifiables**.
  • L'utilisation de variables modifiables est fortement déconseillée.
  • Parfois l'utilisation de variables modifiables peuvent entraîner un code moins complexe, donc Scala prend également en charge les variables modifiables.

– Une variable modifiable est déclarée à l'aide du mot-clé var ;
– Une variable non modifiable est déclarée en utilisant le mot clé val.

– Le type **var** est similaire à une variable dans les langages impératifs tels que C/C++ et Java. Il peut être réaffecté après il a été créé.
– La syntaxe de création et de modification d'une variable est illustrée ci-après.

– Le type **Val** ne peut pas être réaffecté après avoir été initialisé. La syntaxe de création d'un val est illustrée ci-après.

```
scala> val y=10
val y: Int = 10

scala> y=20
-- [E052] Type Error: ----------------------------------------------------
1 |y=20
  |^^^^
  |Reassignment to val y
  |
  | longer explanation available when compiling with `-explain`
1 error found
```


Type de variable	Description
Byte	8-bit signed integer
Short	16-bit signed integer
Int	32-bit signed integer
Long	64-bit signed integer
Float	32-bit single precision float
Double	64-bit double precision float
Char	16-bit unsigned Unicode character
String	A sequence of Chars
Boolean	True or false

2.2 Opérateurs

– Scala fournit un ensemble d'opérateurs pour les types de base.
– Chaque type est une classe et chaque opérateur est une méthode.
– L'utilisation d'un opérateur équivaut à appeler une méthode.

```
scala> val x=1
val x: Int = 1

scala> val y=2
val y: Int = 2

scala> val z=x.+(y)
val z: Int = 3
```

+ n'est pas un opérateur intégré dans Scala. C'est une méthode définie dans la classe Int.

2.3 Fonctions

– Scala utilise le mot-clé **def**

def nomDeLaFonction(arg1, arg2, …, argN):
bloc d'instructions

```
scala> def add(x:Int, y:Int):Int={
     | val sum=x+y
     | return sum
     | }
def add(x: Int, y: Int): Int

scala> add(1,2)
val res1: Int = 3
```


– Fonction Literal

- est une fonction **sans nom ou anonyme** dans le code source.
- est défini avec des paramètres d'entrées entre parenthèses, suivis d'une flèche -> vers la droite et du corps de la fonction.
  - Le corps est entouré d'accolades facultatives.

Exemple :

```
scala> val add = (a:Int, b:Int) => a + b
val add: (Int, Int) => Int = Lambda$1338/188909616@62b0bf85

scala> add(1,2)
val res0: Int = 3
```


– Méthode d'ordre supérieur

- C'est une méthode qui prend une fonction comme paramètre d'entrée.
- De même, une fonction d'ordre supérieur est une fonction qui prend une autre fonction en entrée.
- Les méthodes et fonctions d'ordre supérieur aident à réduire dédoublement de codes.

```
scala> def f1(x:Int):Int={
     | return x+10
     | }
def f1(x: Int): Int
```

```
scala> def f2(n:Int, f1:(Int)=>Long):Long={
     | val x=n*10
     | f1(x)
     | }
def f2(n: Int, f1: Int => Long): Long
```

```
scala> f2(4,f1)
val res5: Long = 50
```


2.4 Pattern Matching (Correspondance du modèle)

– est un concept qui ressemble à une instruction switch dans d'autres langages.

– Une utilisation simple du Pattern Matching consiste à remplacer une instruction if-else à plusieurs niveaux.

Exemple :

```
scala> def f3(x:Int, y:Int, op:String):Double={
     | op match{
     |   case "+" => x+y
     |   case "-" => x-y
     |   case "*" => x*y
     |   case "/" => x/y.toDouble
     | }
     | }
def f3(x: Int, y: Int, op: String): Double

scala> val sum=f3(1,2,"+")
val sum: Double = 3.0

scala> val product=f3(1,2,"*")
val product: Double = 2.0
```

2.5 Classe et Objet

- Une classe dans Scala est similaire à celle des autres langages orientés objet. Elle se compose de champs et de méthodes

```
scala> class A(x:Int, ch:String){
     |   var code=x
     |   var nom=ch
     |   def newcode(y:Int)={
     |     code=y
     |   }
     | }
// defined class A

scala> val a1=new A(1,"Info")
val a1: A = A@3a1540be
```


2.6 Singleton classe

- est une classe qui ne peut être instanciée qu'une seule fois
- Scala fournit le mot-clé objet pour définir une classe singleton.

```
object DatabaseConnection {
    def open(name: String): Int = {
          ...
    }
    def read (streamId: Int): Array[Byte] = {
          ...
    }
    def close (): Unit = {
          ...
    }
}
```


2.7 Case classe

- Est une classe avec le modificateur « case »

```
case class Message (from: String, to: String, content: String)
```

- Scala fournit quelques commodités syntaxiques à cette classe
  - il crée une méthode avec le même nom.
  - Permet de créer une instance sans utiliser le mot-clé new.

```
val request = Message("obj1", "obj2", "scala")
```

- Tous les paramètres d'entrées spécifiés dans la définition de la classe reçoivent implicitement un préfixe val.

```
class Message(val from: String, val to: String, val content: String)
```


2.8 Type Option

- Une option est un type de données qui indique la présence ou l'absence de certaines données.
- Il représente des valeurs facultatives.
- Ce peut être une instance d'une classe case appelée ***Some*** ou d'un objet singleton appelé ***None***.
- Une instance d'une classe case peut stocker des données de tout type.
- L'objet None représente l'absence de données.
- Le type de données Option est utilisé avec une fonction ou une méthode qui renvoie éventuellement une valeur.
- Il revient soit Some(x), où x est la valeur réelle renvoyée, ou l'objet None, qui représente une valeur manquante.
- La valeur facultative renvoyée par une fonction peut être lue à l'aide du Pattern Matching.

Exemple :

```
scala> def colorCode(color: String): Option[Int]={
     |   color match{
     |     case "rouge" => Some(1)
     |     case "bleu" => Some(2)
     |     case _ => None
     |   }
     | }
def colorCode(color: String): Option[Int]

scala> val code = colorCode("vert")
val code: Option[Int] = None

scala> code match{
     | case Some(v) => println("code pour vert est:" + v)
     | case None -> println("code n'est pas defini pour vert")
     | }
code n'est pas defini pour vert
```


2.9 Traits

- Un trait représente **une interface** supportée par une hiérarchie de classes liées.
- C'est un mécanisme d'abstraction qui aide au développement de code modulaire, réutilisable et extensible.
- Les traits sont similaires aux interfaces Java.
- Contrairement à une interface Java, un trait Scala peut inclure le code d'une méthode.
- De plus, un trait Scala peut inclure des champs.


```
scala> trait Forme {
     | def surface():Int
     | }
// defined trait Forme
```

```
scala> class Carre(longueur:Int) extends Forme{
     |    def surface()=longueur*longueur
     | }
// defined class Carre

scala> class Rectangle(longueur:Int, largeur:Int) extends Forme{
     |    def surface()=longueur*largeur
     | }
// defined class Rectangle
```

```
scala> val r1=new Rectangle(3,4)
val r1: Rectangle = Rectangle@249fa2cb

scala> print (r1.surface())
12~

scala> val c1=new Carre(5)
val c1: Carre = Carre@795ce9b5

scala> print (c1.surface())
25~
```


2.10 Collections

- Array : les tableaux mutables de taille fixe homogènes
- List : les listes immuables homogènes (objets dont l'état ne peuvent pas être modifié après leur création)
- Set : les ensembles mutables ou immuables homogènes
- Map : les dictionnaires mutables ou immuables homogènes
- Option : le type optionnel immuable

Type	Description

Tuple	• est un conteneur pour stocker deux ou plusieurs éléments de type différent. Il est inchangeable ; il ne peut pas être modifié après sa création.
	• Un élément dans un tuple a un index.
	`scala> val pers=(1,"Alaoui","Rabat")`
	`val pers: (Int, String, String) = (1,Alaoui,Rabat)`
	`scala> val code=pers._1`
	`val code: Int = 1`
	`scala> val nom=pers._2`
	`val nom: String = Alaoui`

Array	• est une séquence indexée d'éléments. Tous les éléments d'un tableau sont du même type.
	`scala>`
	`val arr = Array(10, 20, 30, 40)`
	`val arr: Array[Int] = Array(10, 20, 30, 40)`
	`scala> arr(0)=50`
	`scala> val first=arr(0)`
	`val first: Int = 50`
	`scala> first`
	`val res6: Int = 50`
	`scala> arr`
	`val res7: Array[Int] = Array(50, 20, 30, 40)`

List	• est une séquence d'éléments linéaire du même type. C'est une structure de données récursive.
	La collection peut être stocké des objets en double
	`scala> val list1=List(10,20,30,40)`
	`val list1: List[Int] = List(10, 20, 30, 40)`
	`scala> val list2=(1 to 10).toList`
	`val list2: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`
	• Les opérations de base sur une liste incluent :
	  • Head : permet de récupérer du premier élément.
	  • Tail : pour récupérer tous les éléments sauf le premier élément.
	  • isEmpty : pour vérifier si une liste est vide.
	`scala> list1.head`
	`val res10: Int = 10`
	`scala> list2.tail`
	`val res11: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10)`

Vector	• La classe Vector est un hybride des classes List et Array, permet à la fois un accès aléatoire rapide et des mises à jour fonctionnelles rapides.
	`scala> val v1=Vector(0,10,20,30,40)`
	`val v1: Vector[Int] = Vector(0, 10, 20, 30, 40)`
	`scala> val v2=v1:+50`
	`val v2: Vector[Int] = Vector(0, 10, 20, 30, 40, 50)`
	`scala> val v3=v2:+60`
	`val v3: Vector[Int] = Vector(0, 10, 20, 30, 40, 50, 60)`
	`scala> val v4=v3(4)`
	`val v4: Int = 40`

Set	• est une collection non ordonnée d'éléments distincts.
	ne contient pas de doublons. pas d'accès à un élément par son index, puisqu'il n'en a pas.
	`scala> val color=Set("bleu", "rouge", "vert")`
	`val color: Set[String] = Set(bleu, rouge, vert)`
	`scala> color.contains("vert")`
	`val res12: Boolean = true`
	• Les sets prennent en charge deux opérations de base :
	  • contains : renvoie vrai si un ensemble contient l'élément passé en entrée à cette méthode.
	  • isEmpty : renvoie vrai si un ensemble est vide.

Map	• Est une collection de paires clé-valeur.
	En python, il s'agit d'un dictionnaire
	`scala> val listVille=Map(1->"Rabat", 2->"Casa", 3->"Fes")`
	`val listVille: Map[Int, String] = Map(1 -> Rabat, 2 -> Casa, 3 -> Fes)`
	`scala> val indiceFes=listVille(3)`
	`val indiceFes: String = Fes`

2.11 Méthodes de collections

Méthode	Fonctionnement

map	• Applique sa fonction d'entrée à tous les éléments de la collection et renvoie une autre collection
	`scala> val y=x.map((x:Int)=>x*10.0)`
	`val y: List[Double] = List(10.0, 20.0, 30.0, 40.0)`
	Ou :
	`scala> val y=x.map{(x:Int)=>x*10.0}`
	`val y: List[Double] = List(10.0, 20.0, 30.0, 40.0)`
	`scala> val y=x map{(x:Int)=>x*10.0}`
	`val y: List[Double] = List(10.0, 20.0, 30.0, 40.0)`

flatMap	• Prend une fonction en entrée, l'applique à chaque élément dans une collection et renvoie une autre collection en conséquence.
	`scala> val phrase="Scala est langage du Spark"`
	`val phrase: String = Scala est langage du Spark`
	`scala> val SimpleEspace=" "`
	`val SimpleEspace: String = " "`
	`scala> val mots=phrase.split(SimpleEspace)`
	`val mots: Array[String] = Array(Scala, est, langage, du, Spark)`
	`scala> val arryOfChars=mots flatMap {_.toList}`
	`val arryOfChars: Array[Char] = Array(S, c, a, l, a, e, s, t, l, a, n, g, a, g, e, d, u, S, p, a, r, k)`

filter	• Applique un prédicat à chaque élément d'une collection et renvoie une autre collection composée uniquement des éléments pour lesquels le prédicat a renvoyé true.
	Un prédicat est une fonction qui renvoie une valeur booléenne.
	`scala> val x=(1 to 10).toList`
	`val x: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`
	`scala> val y=x filter {_ %2 == 0}`
	`val y: List[Int] = List(2, 4, 6, 8, 10)`

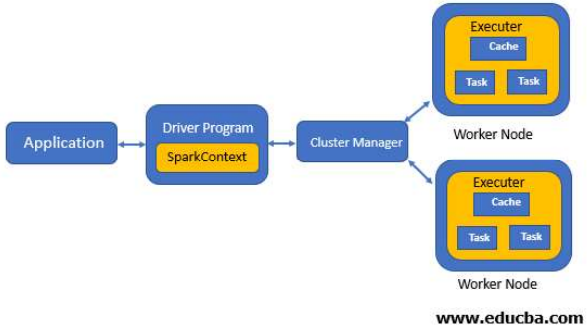
foreach	• Appelle sa fonction d'entrée sur chaque élément de la collection, mais ne retourne rien.
	`scala> val mots="Scala est langage du Spark".split(" ")`
	`val ts: Array[String] = Array(Scala, est, langage, du, Spark)`
	`scala> mots.foreach(println)`
	`Scala`
	`est`
	`langage`
	`du`
	`Spark`

reduce	• Réduit une collection à une seule valeur.
	La fonction d'entrée de la méthode reduce prend deux entrées à la fois et renvoie une valeur.
	`scala> val x=List(2,4,6,8,10)`
	`val x: List[Int] = List(2, 4, 6, 8, 10)`
	`scala> val sum=x reduce {(x,y) => x + y}`
	`val sum: Int = 30`
	`scala> val max = x reduce {(x,y) => if (x > y) x else y}`
	`val max: Int = 10`
	`scala> val min = x reduce {(x,y) => if (x < y) x else y}`
	`val min: Int = 2`

3. Interface de programmation d'applications (API) Spark

– L'API Spark se compose de deux abstractions importantes : SparkContext et Resilient Distributed Dataset (RDD).

– Ces abstractions permettent une application de se connecter à un cluster Spark et utiliser les ressources du cluster.



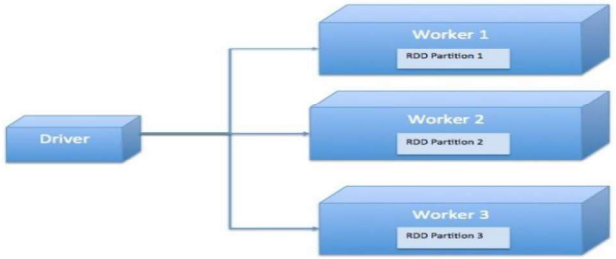
3.1 SparkContext

- est une classe définie dans la bibliothèque Spark.
- C'est le principal point d'entrée dans la bibliothèque Spark.
- Il représente une connexion à un cluster Spark.
- Il est également nécessaire pour créer d'autres objets importants fournis par l'API Spark.

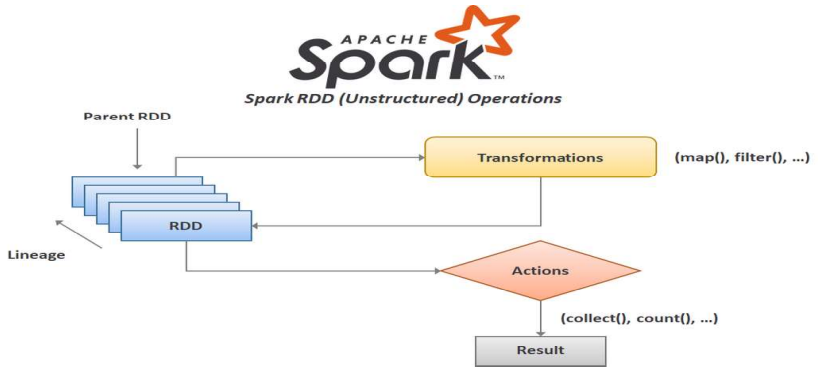
```
scala> import org.apache.spark.SparkContext
import org.apache.spark.SparkContext

scala> import org.apache.spark.SparkConf
import org.apache.spark.SparkConf

scala> val conf = new SparkConf().setAppName("Simple Application")
conf: org.apache.spark.SparkConf = org.apache.spark.SparkConf@73a91b68

scala> val sc = new SparkContext(conf)
org.apache.spark.SparkException: Only one SparkContext should be running in this JVM (see SPARK-2243)
ing SparkContext was created at:
org.apache.spark.sql.SparkSession$Builder.getOrCreate(SparkSession.scala:947)
org.apache.spark.repl.Main$.createSparkSession(Main.scala:106)
```


3.2 Resilient Distributed Dataset (RDD)

- Les RDDs sont une collection d'objets immuables répartis sur plusieurs nœuds d'un cluster.
- Un RDD est créé à partir d'une source de données ou d'une collection d'objets Scala / Python ou Java
- Les RDD sont résilient dans le sens où ils ont la capacité de recréer à n'importe quel moment du processus d'exécution.

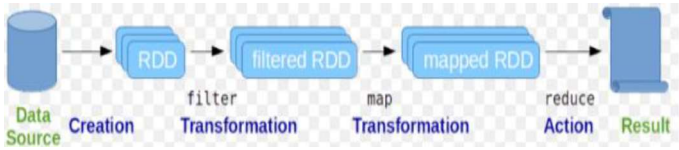


- Les opérations disponibles sur un RDD sont : la création, les transformations, et l'action.



3.2.1 Création d'un RDD

- Il est défini comme une classe abstraite dans la bibliothèque Spark.
- Conceptuellement, RDD est similaire à une collection Scala, sauf qu'il représente un ensemble de données distribué et qu'il prend en charge les opérations paresseuses.
- Les RDDs sont créés à partir d'un fichier dans HDFS par exemple, puis le transforment.
- Les utilisateurs peuvent demander à Spark de sauvegarder un RDD en mémoire, lui permettant ainsi d'être réutilisé efficacement à travers plusieurs opérations parallèles.



- Les méthodes couramment utilisées pour créer un RDD sont :

Méthode	Description
parallelize	• Crée un RDD à partir d'une collection Scala locale. • Il partitionne et distribue les éléments d'un Scala collection et renvoie un RDD représentant ces éléments `scala> val xs=(1 to 10000).toList` `xs: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177,...` `scala> val rdd=sc.parallelize(xs)` `rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>:26`
textFile	• Crée un RDD à partir d'un fichier texte. • Il peut lire un fichier ou plusieurs fichiers dans un répertoire stocké sur un système de fichiers local, HDFS, Amazon S3 ou tout autre système de stockage pris en charge par Hadoop. • Il renvoie un RDD de chaînes, où chaque élément représente une ligne dans le fichier d'entrée. Exemple : Création d'un RDD à partir d'un fichier ou d'un répertoire stocké sur HDFS.

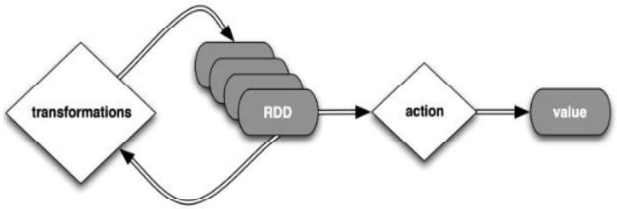
	`scala> val rdd = sc.textFile("hdfs://namenode:9000/path/to/file-or-directory")`
	`rdd: org.apache.spark.rdd.RDD[String] = hdfs://namenode:9000/path/to/file-or-directory MapPartitionsRDD[3] at textFile at <console>:25`
	• La méthode textFile prend un deuxième argument facultatif, qui peut être utilisé pour spécifier le nombre de partitions. Par défaut, Spark crée une partition RDD pour chaque bloc de fichiers.

WholeTextFilE	• Lit tous les fichiers texte d'un répertoire et renvoie un RDD de paires clé-valeur.
	• Chaque paire clé-valeur dans le RDD renvoyé correspond à un seul fichier.
	• La partie clé stocke le chemin d'un fichier et la partie valeur stocke le contenu d'un fichier.
	• Cette méthode peut également lire les fichiers stockés sur un système de fichiers local, HDFS, Amazon S3 ou tout autre système de stockage pris en charge par Hadoop.
	`scala> val rdd = sc.wholeTextFiles("path/to/my-data/*.txt")`
	`rdd: org.apache.spark.rdd.RDD[(String, String)] = path/to/my-data/*.txt MapPartitionsRDD[5] at wholeTextFiles at <console>:25`

sequenceFile	• Lit les paires clé-valeur à partir d'un fichier de séquence stocké sur un système de fichiers local, HDFS ou tout autre système de stockage pris en charge par Hadoop.
	• Il renvoie un RDD de paires clé-valeur.
	• En plus de fournir le nom d'un fichier d'entrée, il faut spécifier les types de données pour les clés et les valeurs en tant que paramètres.
	`scala> val rdd = sc.sequenceFile[String, String]("some-file")`
	`rdd: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD[7] at sequenceFile at <console>:25`

– Opérations de RDD

- Les opérations RDD peuvent être classées en deux types : **transformation et action** :
  - Une transformation crée un nouveau DDR.
  - Une action renvoie une valeur à un programme pilote.
- Les RDD étant immutables, une transformation appliquée à un RDD ne va pas le modifier mais plutôt en créer un nouveau enrichit de nouvelles informations correspondant à cette transformation.



3.2.2 Transformations

– Les transformations peuvent

- Produire un RDD à partir d'un autre RDD : map, filter, reduceByKey, etc.
- Produire un RDD à partir de deux RDD : union, join, cartesian, etc.
- Produire 0 ou plusieurs RDD à partir d'un RDD : flatMap.

Type	Description

map	• Prend une fonction en entrée et l'applique à chaque élément de RDD source pour créer un nouveau RDD.
	• La fonction d'entrée à mapper doit prendre un seul paramètre d'entrée et retourner une valeur.
	`scala> val lignes = sc.textFile("...")`
	`lignes: org.apache.spark.rdd.RDD[String] = ... MapPartitionsRDD[19] at textFile at <console>:25`
	`scala> val length=lignes map {l=>l.length}`
	`length: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[20] at map at <console>:25`

filter	- Prend une fonction booléenne en entrée et l'applique à chaque élément dans le RDD source pour créer un nouveau RDD
	`scala> val lignes = sc.textFile("...")`
	`lignes: org.apache.spark.rdd.RDD[String] = ... MapPartitionsRDD[22] at textFile at <console>:25`
	`scala> val longlignes=lignes filter {l=>l.length > 80}`
	`longlignes: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[23] at filter at <console>:25`

flatMap	- Prend une fonction d'entrée, et renvoie une séquence pour chaque élément d'entrée qui lui est passé.
	`scala> val lignes = sc.textFile("...")`
	`lignes: org.apache.spark.rdd.RDD[String] = ... MapPartitionsRDD[25] at textFile at <console>:25`
	`scala> val mots=lignes flatMap {l=>l.split(" ")}`
	`mots: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[26] at flatMap at <console>:25`

mapPartitions	- permet de traiter les données au niveau de la partition. - La fonction prend un itérateur en entrée et renvoie un autre itérateur en tant que production.

	- La méthode mapPartitions renvoie un nouveau RDD formé en appliquant une fonction spécifiée par l'utilisateur à chaque partition du RDD source.
	`scala> val lignes = sc.textFile("...")`
	`lignes: org.apache.spark.rdd.RDD[String] = ... MapPartitionsRDD[31] at textFile at <console>:25`
	`scala> val lengths=lignes mapPartitions {iter =>iter.map {l => l.length}}`
	`lengths: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[32] at mapPartitions at <console>:25`

union	- Prend un RDD en entrée et renvoie un nouveau RDD qui contient l'union des éléments dans le RDD source et le RDD qui lui sont transmis en tant qu'entrée.
	`scala> val lignesFile1 = sc.textFile("...")`
	`lignesFile1: org.apache.spark.rdd.RDD[String] = ... MapPartitionsRDD[34] at textFile at <console>:25`
	`scala> val lignesFile2 = sc.textFile("...")`
	`lignesFile2: org.apache.spark.rdd.RDD[String] = ... MapPartitionsRDD[36] at textFile at <console>:25`
	`scala> val lignesFromBothFiles = lignesFile1.union(lignesFile2)`
	`org.apache.hadoop.mapred.InvalidInputException: Input path does not exist: file:/C:/WINDOWS/system32/...`

intersection	- Prend un RDD en entrée et renvoie un nouveau RDD qui contient l'intersection des éléments du RDD source et le RDD qui lui sont transmis en entrée. `scala> val lignesInBothFiles = lignesFile1.intersection(lignesFile2)`
subtract	- Prend un RDD en entrée et renvoie un nouveau RDD qui contient des éléments dans la source RDD mais pas dans l'entrée RDD `scala> val lignesInFile1Only = lignesFile1.subtract(lignesFile2)`

distinct	- Renvoie un nouveau RDD contenant les éléments distincts du RDD source
	`scala> val nombres=sc.parallelize(List(1,2,3,4,3,2,1))`
	`nombres: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[39] at parallelize at <console>:25`
	`scala> val nombresUnique = nombres.distinct`
	`nombresUnique: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[42] at distinct at <console>:25`

cartésien	- prend un RDD en entrée et renvoie un RDD contenant le produit cartésien de tous les éléments des deux RDD.
	`scala>val nombres = sc.parallelize(List(1, 2, 3, 4))`
	`scala>val alphabets = sc.parallelize(List("a", "b", "c", "d"))`
	`scala>val cartesianProduct = nombres.cartesian(alphabets)`

zip	- Prend un RDD en entrée et renvoie paires de RDD, où le premier élément d'une paire est du RDD source et le deuxième élément provient du RDD d'entrée.
	- le RDD retourné par zip a le même nombre d'éléments que le RDD source.
	`scala>val nombres = sc.parallelize(List(1, 2, 3, 4))`
	`scala>val alphabets = sc.parallelize(List("a", "b", "c", "d"))`
	`scala>val zippedPairs = nombres.zip(alphabets)`

zipWithIndex	- Compresse les éléments du RDD source avec leurs indices et renvoie un RDD de paires
	`scala>val alphabets = sc.paralléliser(Liste("a", "b", "c", "d"))`
	`val alphabetsWithIndex = alphabets.zip`

groupBy	- regroupe les éléments d'un RDD selon un critère spécifié par l'utilisateur. - Il applique cette fonction à tous les éléments du RDD source et renvoie un RDD de paires. Dans chaque paire renvoyée, le premier élément est une clé et le deuxième élément est une collection des éléments mappés à cette clé par la fonction d'entrée

	Exemple : Ce code groupe les clients par zip code
	`case class Client(nom: String, age: Int, zip: String)`
	`val lignes = sc.textFile("...")`
	`val client = lignes map { l => { val a = l.split(",")`
	`Client(a(0), a(1).toInt, a(2))`
	`}`
	`}`
	`val groupByZip = client.groupBy { c => c.zip}`

keyBy	- Est similaire à la méthode groupBy
	- La différence entre groupBy et keyBy est que le deuxième élément d'une paire renvoyée est une collection de éléments dans le premier cas, alors qu'il s'agit d'un élément unique dans le second cas.
	`case class Personne(nom: String, age: Int, zip: String)`
	`val lignes = sc.textFile("...")`
	`val client = lignes map { l => { val a = l.split(",")`
	`Personne(a(0), a(1).toInt, a(2))`
	`}`
	`}`
	`val keyedByZip = client.keyBy { c => c.zip}`

sortBy	- Renvoie un RDD avec des éléments triés à partir du RDD source.
	- Définit avec deux paramètres d'entrée :
	- Le premier est une fonction qui génère une clé pour chaque élément du RDD source.
	- Le deuxième permet de spécifier l'ordre croissant ou décroissant pour le tri.
	`scala> val nombres = sc.parallelize(List(3,2,4,1,5))`
	`scala> val triNombres = nombres.sortBy(x => x, true)`
	`case class Personne(nom: String, age: Int, zip: String)`
	`val lignes = sc.textFile("...")`
	`val client = lignes map { l => { val a = l.split(",")`
	`Personne(a(0), a(1).toInt, a(2)) }`
	`}`
	`val triedByAge = client.sortBy ( c => c.age, true)`

pipe	- permet d'exécuter un programme externe dans un processus forké. Il capture la sortie du programme externe sous forme de chaîne et renvoie un RDD de chaînes.

randomSplit	- divise le RDD source en un tableau de RDD
	`scala> val nombres = sc.parallelize((1 à 100).toList)`
	`scala> val divise = nombres.randomSplit(Array(0.6, 0.2, 0.2))`

coalesce	- réduit le nombre de partitions dans un RDD
	`scala> val nombres = sc.parallelize((1 à 100).toList)`
	`scala> val nombresAvecUnePartition = nombres.coalesce(1)`

repartition	- prend un entier en entrée et renvoie un RDD avec un nombre spécifié de partitions.
	- C'est utile pour augmenter le parallélisme. Il redistribue les données, c'est donc une opération coûteuse.
	`scala> val nombres = sc.parallelize((1 à 100).toList)`
	`scala>val nombresAvecUnePartition = nombres.repartition(4)`

sample	- renvoie un sous-ensemble échantillonné du RDD source.
	- Il faut trois paramètres d'entrée :
	- Le premièr spécifie la stratégie de remplacement.
	- Le deuxième spécifie le rapport entre la taille de l'échantillon et taille RDD source.
	- Le troisième, qui est facultatif, spécifie des valeurs aléatoires pour l'échantillonnage.
	`scala> val nombres = sc.parallelize((1 à 100).toList)`
	`scala> val sampleNumbers = nombres. échantillon(vrai, 0.2)`

– Transformations sur RDD de paires clé-valeur

Type	Description
keys	- renvoie un RDD contenant uniquement les clés du RDD source scala> val cleRdd = sc.parallelize(List(("a", 1), ("b", 2), ("c", 3))) scala> val keysRdd=cleRdd.keys
values	- renvoie un RDD contenant uniquement les valeurs du RDD source scala> val cleRdd = sc.parallelize(List(("a", 1), ("b", 2), ("c", 3))) scala> val valuesRdd=cleRdd.values
mapValues	- est une méthode d'ordre supérieur qui prend une fonction en entrée et l'applique à chaque valeur dans le RDD source. - Il renvoie un RDD de paires clé-valeur. - Elle est similaire à la méthode map, sauf qu'elle s'applique la fonction d'entrée uniquement à chaque valeur dans le RDD source, de sorte que les clés ne sont pas modifiées. - Le RDD retourné a les mêmes clés que le RDD source. scala> val cleRdd = sc.parallelize(List(("a", 1), ("b", 2), ("c", 3))) scala> val valuesDoubled=cleRdd.mapValues { x => 2*x }

join	- Prend un RDD de paires clé-valeur en entrée et effectue une jointure interne sur la source et l'entrée RDD. - Il renvoie un RDD de paires, où le premier élément d'une paire est une clé trouvée à la fois dans le RDD source et d'entrée et le deuxième élément est un tuple contenant des valeurs mappées à cette clé dans le RDD source et d'entrée. scala> val pairRdd1 = sc.parallelize(List(("a", 1), ("b", 2), ("c", 3))) scala> val pairRdd2 = sc.parallelize(List(("b", "second"), ("c", "third"), ("d", "fourth"))) scala> val joinRdd=pairRdd1.join(pairRdd2)
leftOuterJoin	- Prend un RDD de paires clé-valeur en entrée et effectue une jointure externe gauche sur le source et entrée RDD. - Il renvoie un RDD de paires clé-valeur, où le premier élément d'une paire est une clé de source RDD et le deuxième élément est un tuple contenant la valeur de la source RDD et la valeur facultative de le RDD d'entrée. - Une valeur facultative du RDD d'entrée est représentée avec le type d'option.

	scala> val pairRdd1 = sc.parallelize(List(("a", 1), ("b", 2), ("c", 3))) scala> val pairRdd2 = sc.parallelize(List(("b", "second"), ("c", "third"), ("d", "fourth"))) scala> val leftOuterJoinRdd=pairRdd1.leftOuterJoin(pairRdd2)
rightOuterJoin	- Prend un RDD de paires clé-valeur en entrée et effectue une jointure externe droite sur le source et entrée RDD. - Il renvoie un RDD de paires clé-valeur, où le premier élément d'une paire est une clé de RDD d'entrée et le deuxième élément est un tuple contenant la valeur facultative de la source RDD et la valeur de l'entrée RDD. - Une valeur facultative du RDD source est représentée par le type d'option. scala> val pairRdd1 = sc.parallelize(List(("a", 1), ("b", 2), ("c", 3))) scala> val pairRdd2 = sc.parallelize(List(("b", "second"), ("c", "third"), ("d", "fourth"))) scala> val rightOuterJoinRdd=pairRdd1.rightOuterJoin(pairRdd2)

fullOuterJoin	- Prend un RDD de paires clé-valeur en entrée et effectue une jointure externe complète sur la source et entrée RDD. - Il renvoie un RDD de paires clé-valeur. scala> val pairRdd1 = sc.parallelize(List(("a", 1), ("b", 2), ("c", 3))) scala> val pairRdd2 = sc.parallelize(List(("b", "second"), ("c", "third"), ("d", "fourth"))) scala> val fullOuterJoinRdd=pairRdd1.fullOuterJoin(pairRdd2)
sampleByKey	- Renvoie un sous-ensemble du RDD source échantillonné par clé. - Il prend le taux d'échantillonnage pour chaque touche en entrée et renvoie un échantillon du RDD source. scala>val pairRdd = sc.parallelize(List(("a", 1), ("b",2), ("a", 11),("b",22),("a", 111), ("b",222))) scala>val sampleRdd = pairRdd.sampleByKey(true, Map("a"-> 0.1, "b"->0.2))

subtractByKey	- Prend un RDD de paires clé-valeur en entrée et renvoie un RDD de paires clé-valeur contenant uniquement les clés qui existent dans le RDD source, mais pas dans le RDD d'entrée.
	`val pairRdd1 = sc.parallelize(List(("a", 1), ("b",2), ("c",3)))`
	`val pairRdd2 = sc.parallelize(List(("b", "second"), ("c","third"), ("d","fourth")))`
	`val resultRdd = pairRdd1.subtractByKey(pairRdd2)`

groupByKey	- La méthode groupByKey fonctionne sur un RDD de paires clé-valeur, de sorte qu'une fonction de générateur de clé n'est pas requise en entrée.
	- C'est semblable au méthode groupBy que nous avons vue précédemment.
	- La différence est que groupBy est une méthode d'ordre supérieur qui prend comme entree une fonction qui renvoie une clé pour chaque élément du RDD source.
	`val pairRdd = sc.parallelize(List(("a", 1), ("b",2), ("c",3), ("a", 11), ("b",22), ("a",111)))`
	`val groupedRdd = pairRdd.groupByKey()`

reduceByKey	- prend un opérateur binaire associatif en entrée et réduit les valeurs avec la même clé en une seule valeur à l'aide de l'opérateur binaire spécifié.
	- Un opérateur binaire prend deux valeurs en entrée et renvoie une seule valeur en sortie.
	- Un opérateur associatif renvoie le même résultat quel que soit le groupement des opérandes.
	- La méthode reduceByKey peut être utilisée pour agréger des valeurs par clé.
	- Par exemple, il peut être utilisé pour calculer la somme, le produit, le minimum ou le maximum de toutes les valeurs mappées sur la même clé.
	`val pairRdd = sc.parallelize(List(("a", 1), ("b",2), ("c",3), ("a", 11), ("b",22), ("a",111)))`
	`val sumByKeyRdd = pairRdd.reduceByKey((x,y) => x+y)`
	`val minByKeyRdd = pairRdd.reduceByKey((x,y) => if (x < y) x else y)`

3.2.3 Actions

– Les actions sont des méthodes RDD qui renvoient une valeur à un programme pilote

Type	Description

collect	- renvoie les éléments du RDD source sous forme de tableau.
	`val rdd = sc.parallelize((1 to 10000).toList)`
	`val filteredRdd = rdd filter { x => (x % 1000) == 0 }`
	`val filterResult = filteredRdd.collect`

count	- renvoie un nombre d'éléments dans le RDD source
	`val rdd = sc.parallelize((1 to 10000).toList)`
	`val total = rdd.count`

countByValue	- envoie un décompte de chaque élément unique dans le RDD source. Il renvoie une instance de la classe Map contenant chaque élément unique et son nombre en tant que paire clé-valeur.
	`val rdd = sc.parallelize(List(1, 2, 3, 4, 1, 2, 3, 1, 2, 1))`
	`val counts = rdd.countByValue`

first	- envoie le premier élément dans le RDD source
	`val rdd = sc.parallelize(List(10, 5, 3, 1))`
	`val firstElement = rdd.first`

Max / min	- renvoie le plus grand/ plus petit élément d'un RDD
	`val rdd = sc.parallelize(List(2, 5, 3, 1))`
	`val maxElement = rdd.max`
	`val minElement = rdd.min`

take	- prend un entier N en entrée et renvoie un tableau contenant le premier élément N du RDD source
	`val rdd = sc.parallelize(List(2, 5, 3, 1, 50, 100))`
	`val first3 = rdd.take(3)`

takeOrdered	- prend un entier N en entrée et renvoie un tableau contenant les N plus petits éléments dans le RDD source
	`val rdd = sc.parallelize(List(2, 5, 3, 1, 50, 100))`
	`val smallest3 = rdd.takeOrdered(3)`

top	- prend un entier N en entrée et renvoie un tableau contenant les N plus grands éléments du RDD source
	`val rdd = sc.parallelize(List(2, 5, 3, 1, 50, 100))`
	`val largest3 = rdd.top(3)`

fold	- agrège les éléments dans le RDD source en utilisant le zéro neutre spécifié valeur et un opérateur binaire associatif.
	- Il agrège d'abord les éléments de chaque partition RDD, puis agrège les résultats de chaque partition.
	`val numbersRdd = sc.parallelize(List(2, 5, 3, 1))`
	`val sum = numbersRdd.fold(0) ((partialSum, x) => partialSum + x)`
	`val product = numbersRdd.fold(1) ((partialProduct, x) => partialProduct * x)`

reduce	- agrège les éléments du RDD source à l'aide d'une méthode associative et opérateur binaire commutatif qui lui est fourni.
	`val numbersRdd = sc.parallelize(List(2, 5, 3, 1))`
	`val sum = numbersRdd.reduce ((x, y) => x + y)`
	`val product = numbersRdd.reduce((x, y) => x * y)`

– **Actions sur RDD des paires clé-valeur**

Type	Description

countByKey	- Compte les occurrences de chaque clé unique dans le RDD source.
	- Il renvoie une carte de paires de nombre de clés
	`val pairRdd = sc.parallelize(List(("a", 1), ("b", 2), ("c", 3), ("a", 11), ("b", 22), ("a", 1)))`
	`val countOfEachKey = pairRdd.countByKey`

lookup	- Prend une clé en entrée et renvoie une séquence de toutes les valeurs mappées à cette clé dans le RDD source
	`val pairRdd = sc.parallelize(List(("a", 1), ("b", 2), ("c", 3), ("a", 11), ("b", 22), ("a", 1)))`
	`val values = pairRdd.lookup("a")`

– **Actions sur RDD de types numériques :** Mean, stdev, sum, variance

variance	`val numbersRdd = sc.parallelize(List(2, 5, 3, 1))` `val variance = numbersRdd.variance`

– **Sauvegarder un RDD :**

- Une fois les données traitées, les résultats sont enregistrés sur disque.
- Spark permet à un développeur d'applications d'économiser un RDD à n'importe quel système de stockage pris en charge par Hadoop.
- Un RDD enregistré sur le disque peut être utilisé par un autre Spark ou Application MapReduce.

saveAsTextFile	Enregistre les éléments du RDD source dans le répertoire spécifié sur n'importe quel Système de fichiers pris en charge par Hadoop. Chaque élément RDD est converti en sa représentation sous forme de chaîne et stocké sous forme de ligne de texte.
	`val numbersRdd = sc.parallelize((1 to 10000).toList)`
	`val filteredRdd = numbersRdd filter { x => x % 1000 == 0}`
	`filteredRdd.saveAsTextFile("numbers-as-text")`

saveAsObjectFile	enregistre les éléments du RDD source en tant qu'objets Java sérialisés dans le répertoire spécifié.
	`val numbersRdd = sc.parallelize((1 to 10000).toList)`
	`val filteredRdd = numbersRdd filter { x => x % 1000 == 0}`
	`filteredRdd.saveAsObjectFile("numbers-as-object")`

saveAsSequenceFile	Enregistre un RDD de paires clé-valeur au format SequenceFile.
	Un RDD de valeur-clé les paires peuvent également être enregistrées au format texte à l'aide de saveAsTextFile
	`val pairs = (1 to 10000).toList map {x => (x, x*2)}`
	`val pairsRdd = sc.parallelize(pairs)`
	`val filteredPairsRdd = pairsRdd filter { case (x, y) => x % 1000 ==0 }`
	`filteredPairsRdd.saveAsSequenceFile("pairs-as-sequence")`
	`filteredPairsRdd.saveAsTextFile("pairs-as-text")`