Big Data Analytics

« Pilotage de la performance pour une bonne gouvernance des entreprises »

CHAPITRE 2 – Big Data Analytics avec Spark

Motivation de Spark

- Supporter des traitements itératifs efficacement
  - Applications émergentes tels que PageRank, clustering par nature itératives
  - Systèmes du style Hadoop matérialisent les résultats intermédiaires à performances dégradées
- Solution
  - Les données doivent résider en mémoire centrale et être partagées → à mémoire distribuée

Qu’est-ce que Spark ?

- Un **moteur d’exécution** basé sur des opérateurs de haut niveau.
- Comprend des opérateurs Map/Reduce, et d’autres opérateurs de second ordre.
- **Introduit un concept de collection résidente en mémoire** (RDD) qui améliore considérablement certains traitements, dont ceux basés sur des itérations.
- De nombreuses librairies pour la fouille de données (MLib), le traitement des graphes, le traitement de flux (*streaming*)…

Resilient Distributed Datasets (RDD)

- C’est le concept central : **Un RDD est une collection calculée à partir d’une source de données** (MongoDB, un flux, un autre RDD).

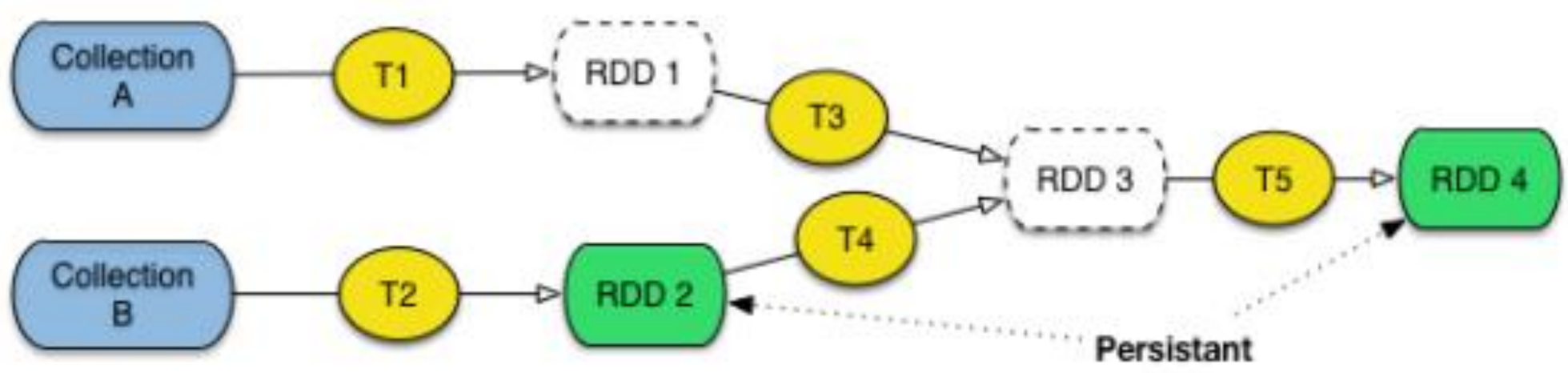
- Un RDD peut être marqué comme **persistant** : il est alors placé en mémoire RAM et conservé par Spark.

- Spark conserve **l’historique des opérations** qui a permis de constituer un RDD, et la reprise sur panne s’appuie sur cet historique afin de reconstituer le RDD en cas de panne.

- Un RDD est un "bloc" **non modifiable**. Si nécessaire il est **entièrement recalculé**.

Un workflow avec RDD dans Spark

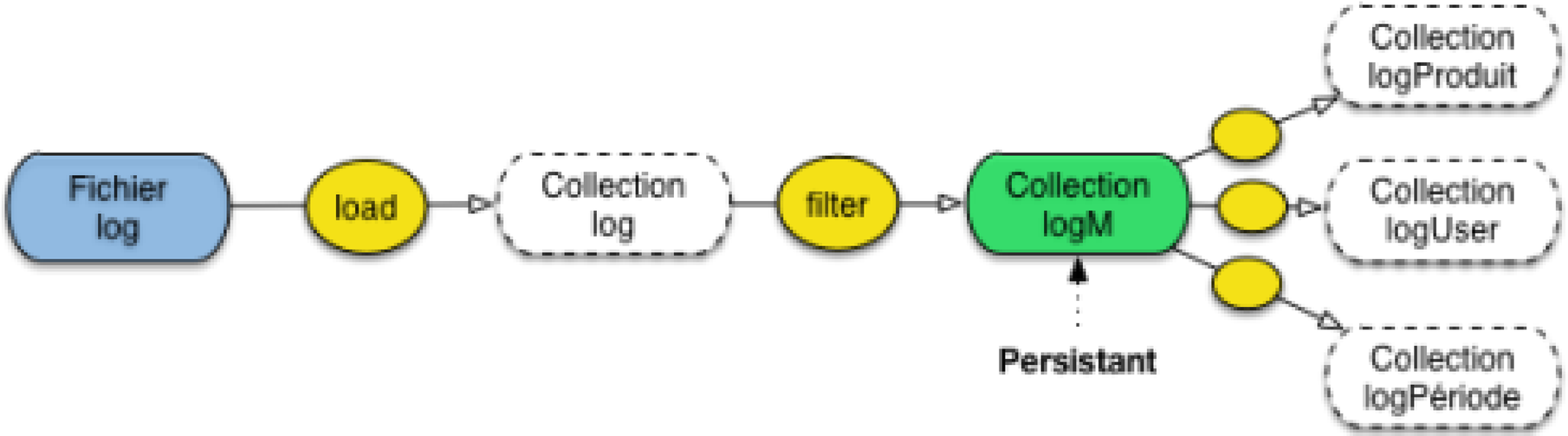
- Des **transformations** créent des RDD à partir d'une ou deux sources de données.



- Les RDD persistants sont en préservés en mémoire RAM, et peuvent être réutilisés par plusieurs traitements

Exemple : Analyse de fichiers log

- On veut analyser le fichier journal (log) d'une application dont un des modules (M) est suspect.
- On construit un programme qui charge le log, ne conserve que les messages produits par le module M et les analyse.
- On peut analyser par produit, par utilisateur, par période, etc.



esprit

Spécification avec Spark

- Première phase pour construire logM

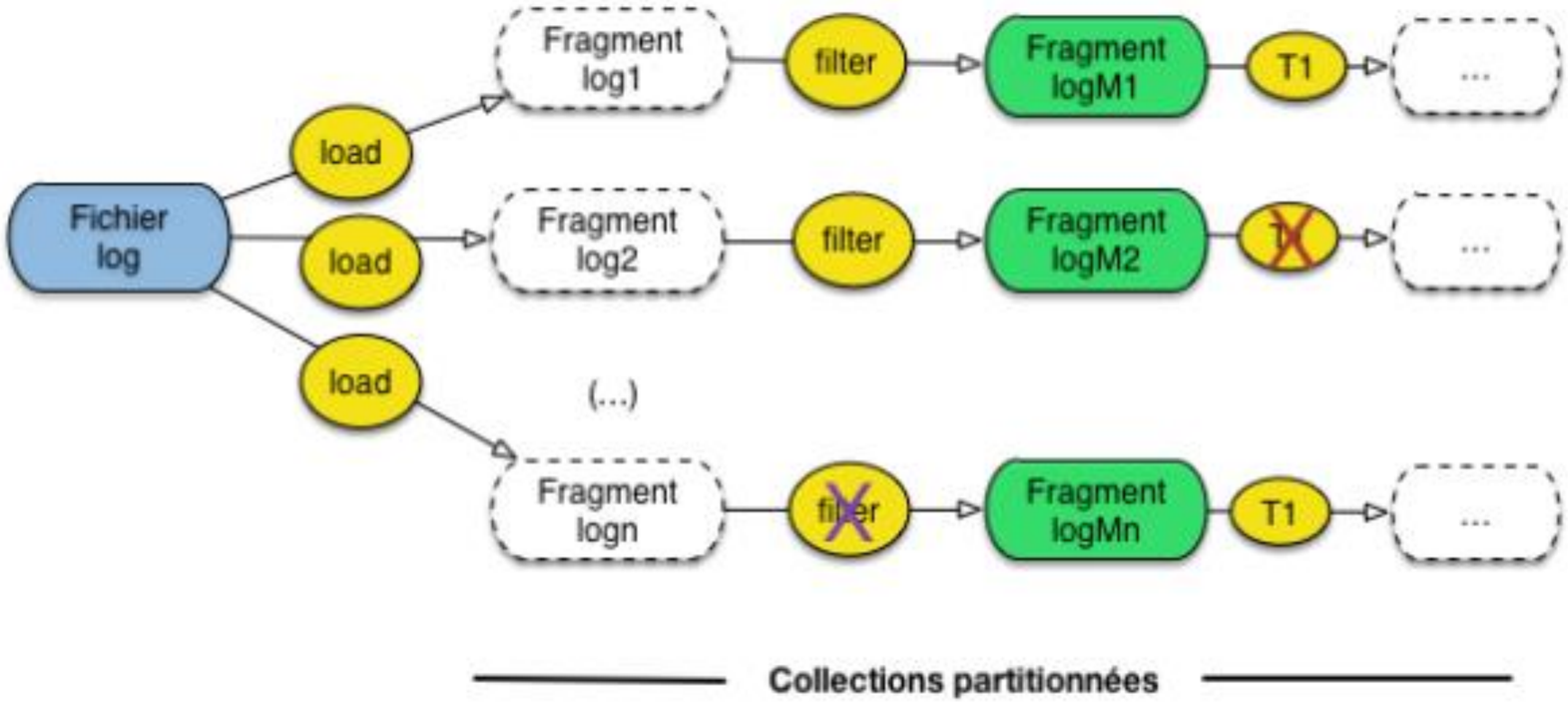
```
// Chargement de la collection
log = load ("app.log") as (...)
// Filtrage des messages du module M
logM = filter log with log.message.contains ("M")
// On rend logM persistant !
logM.persist();
```

- Analyse à partir de logM

```
// Filtrage par produit
logProduit = filter logM
with log.message.contains ("product P")
// .. analyse du contenu de logProduit
```

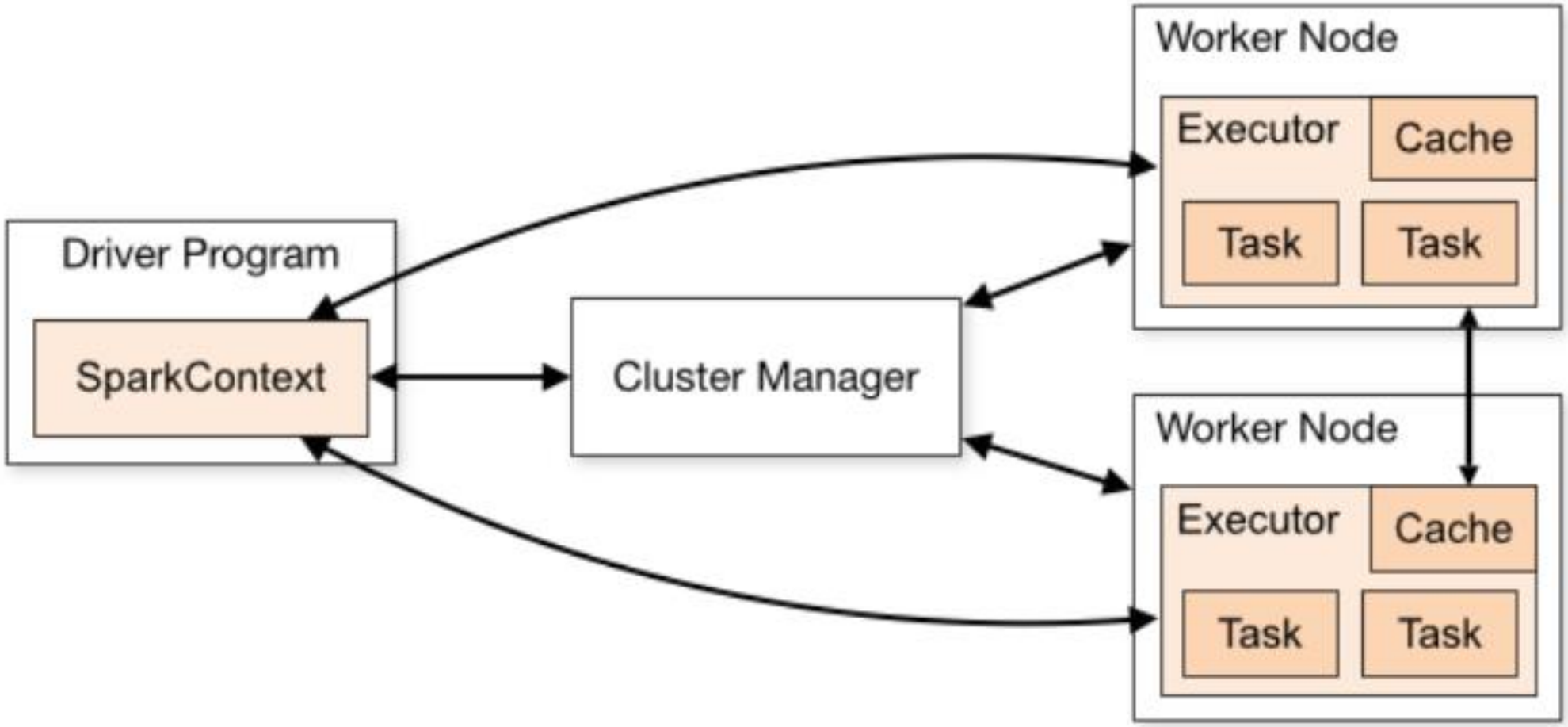
Reprise sur panne dans Spark

- Un RDD est une collection **partitionnée**.



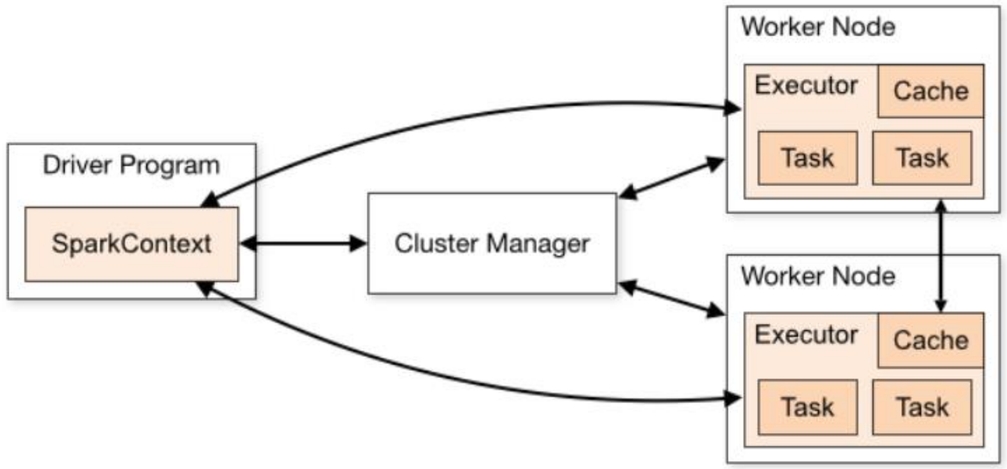
- Panne au niveau d'un nœud implique un recalcul basé sur le fragment F persistant qui précède ce nœud dans le workflow.

Architecture Spark



Architecture Spark

- Application = driver + exécuteurs
- Driver = programme qui lance et coordonne plusieurs tâches sur le cluster
- Exécuteurs = processus indépendants qui réalisent les tâches de calcul
- SparkContext
  - Objet Java qui permet de se connecter au cluster
  - Fournit des méthodes pour créer des RDD



esprit

Fonctionnement de Spark

- Resilient Distributed Datasets (RDDs)
  - Structures accessibles en lecture seule
  - Stockage distribué en mémoire centrale
  - Restriction aux opérations sur gros granules
    - Transformations de la structure en entier vs MAJ valeurs atomiques qui nécessite propagation replicats
  - Journalisation pour assurer la tolérance aux fautes
    - Possibilité de rejouer les transformations vs checkpointing

Fonctionnement des RDD

1. Création
   - Chargement données depuis SGF distribué/local
   - Transformation d'une RDD existante

Note : RDD est une séquence d'enregistrements

2. Transformations
   - map : applique une fonction à chaque élément
   - filter : restreint aux éléments selon condition
   - join : combine deux RDD sur la base des clés *

(*) Les RDD en entrée doivent être des séquences de paires (clé,valeur)

Fonctionnement des RDD

3. Actions
   - collect : retourne les éléments
   - count : comptes les éléments
   - save : écrit les données sur le SF

4. Paramétrage du stockage en mémoire
   - persist : force le maintien en mémoire
   - unpersist : force l'écriture sur disque

- Notes :
  - Par défaut, les RDD sont persistantes en mémoire
  - Si manque d'espace alors écriture sur disque
  - Possibilité d'attribuer des priorités

Illustration d’une RDD

- On considère une chaîne de traitements classique :

1. Chargement depuis stockage (local ou hdfs)
2. Application d’un filtre simple
3. Cardinalité du résultat de 2
4. Paramétrage de la persistance

```
1 lines=spark.textFile("hfds://file.txt")
2 data=lines.filter(_.contains( "word"))
3 data.count
4 data.persist()
```

Concepts Spark

- RDD - Resilient Distributed Dataset
  - L'abstraction de base de Spark est le RDD
    - 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 tout ce qui peut être chargé par Hadoop peut être chargé par Spark

Concepts Spark

- 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.

  - 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.

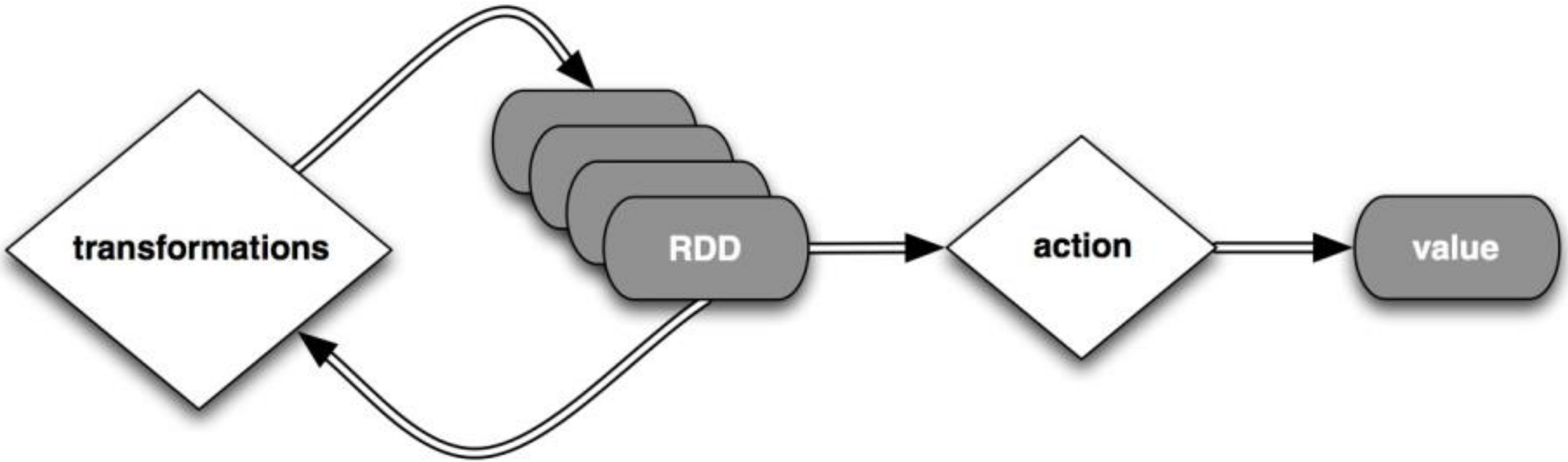
Concepts Spark

- Transformations et Actions
  - 2 concepts de base s'appuient et s'appliquent sur le RDD,
    - les Transformations
    - les Actions

  - Une fois toutes les transformations définies, il faut appliquer une Action pour lancer le calcul sur le Cluster ou les CPU locaux.

  - 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.

Concepts Spark



Concepts Spark

- Spark Context
  - 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 un CPU du code Spark qui va l'utiliser.

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

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

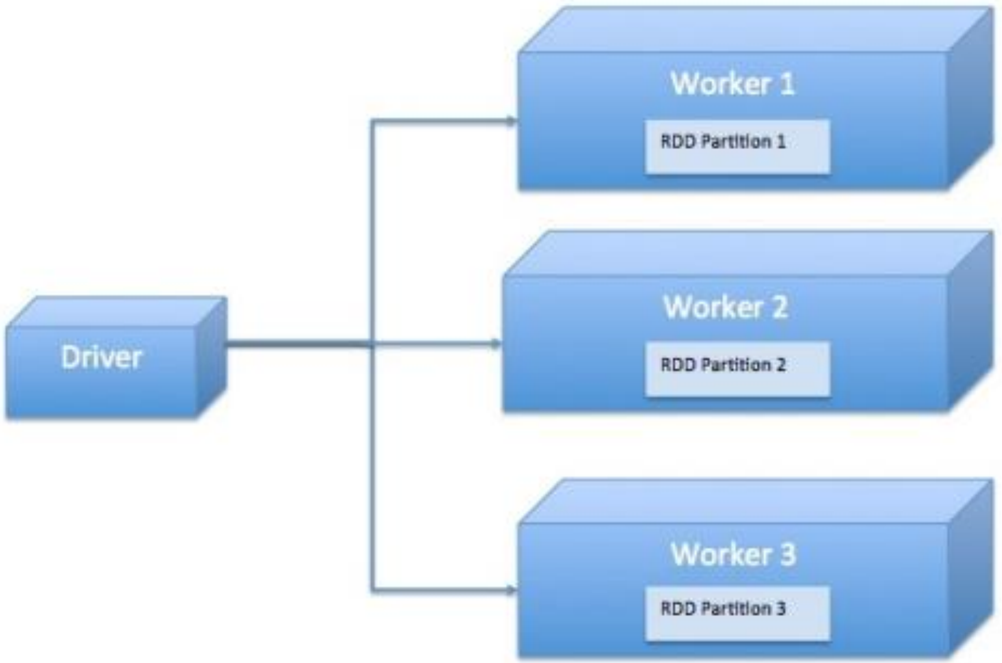
Instancier un SparkContext en Scala

RDD – Mise en œuvre

- 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 opérations disponibles sur un RDD sont :
  - La création
  - Les transformations
  - L'action



RDD – Mise en œuvre

- Le RDD peut subir des transformations successives au travers de fonctions similaires à celles présentes dans les collections classiques :
  - 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 nœuds 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.

RDD – Mise en œuvre

Cycle de vie d'un RDD

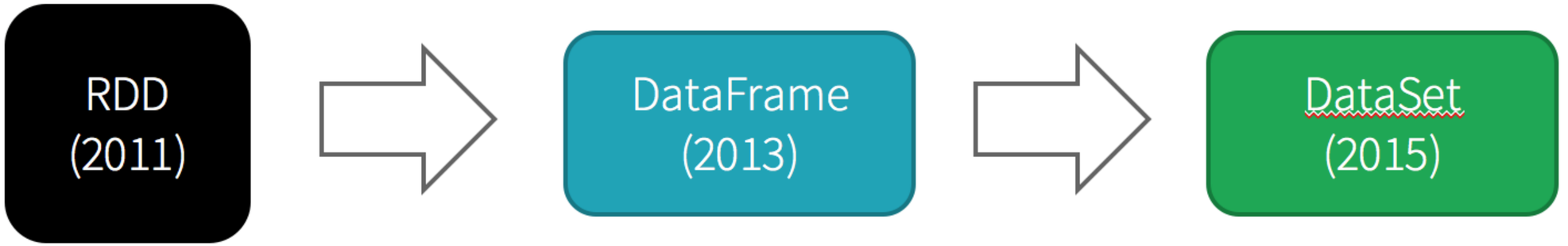
```
1. rdd1 = création du RDD à partir d'une source de données ou d'une collection
2. rdd2 = rdd1.tranform1(fonctionDeTransformationAAppliquerSurChaqueElement)
3. rdd3 = rdd2.tranform2(fonctionDeTransformationAAppliquerSurChaqueElement)
4. ...
5. ...
6. ...
7. rddn = rddm.tranform3(fonctionDeTransformationAAppliquerSurChaqueElement)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

- Une seule action peut être appliquée. Elle consiste à exécuter une opération sur tous les nœuds du cluster et à renvoyer le résultat au driver. L'action peut ne produire aucun résultat (action `foreach` par exemple) , ou produire un résultat qui soit un objet ou une collection comme :
  - `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)

Dataframes et Datasets

- Un RDD, du point de vue du programmeur, c'est un conteneur d'objets java.
- Le type précis de ces objets n'est pas connu par Spark. Du coup :
  - Tout ce que Spark peut faire, c'est appliquer la sérialisation/désérialisation java
  - Aucun accès aux objets grâce à un langage déclaratif n'est possible.
  - Et donc pas d'optimisation, et la nécessité de tout écrire sous forme de fonctions java.
- Depuis la version 1.6 : on dispose de RDD améliorés : les Datasets. On peut les traiter comme des tables relationnelles.
- Finalement, le schéma c'est utile, et le relationnel, c'est bien !

Dataframes et Datasets



Distribute collection of JVM objects

Functional Operators (map, filter, etc.)

Distribute collection of Row objects

Expression-based operations and UDFs

Logical plans and optimizer

Fast/efficient internal representations

Internally rows, externally JVM objects

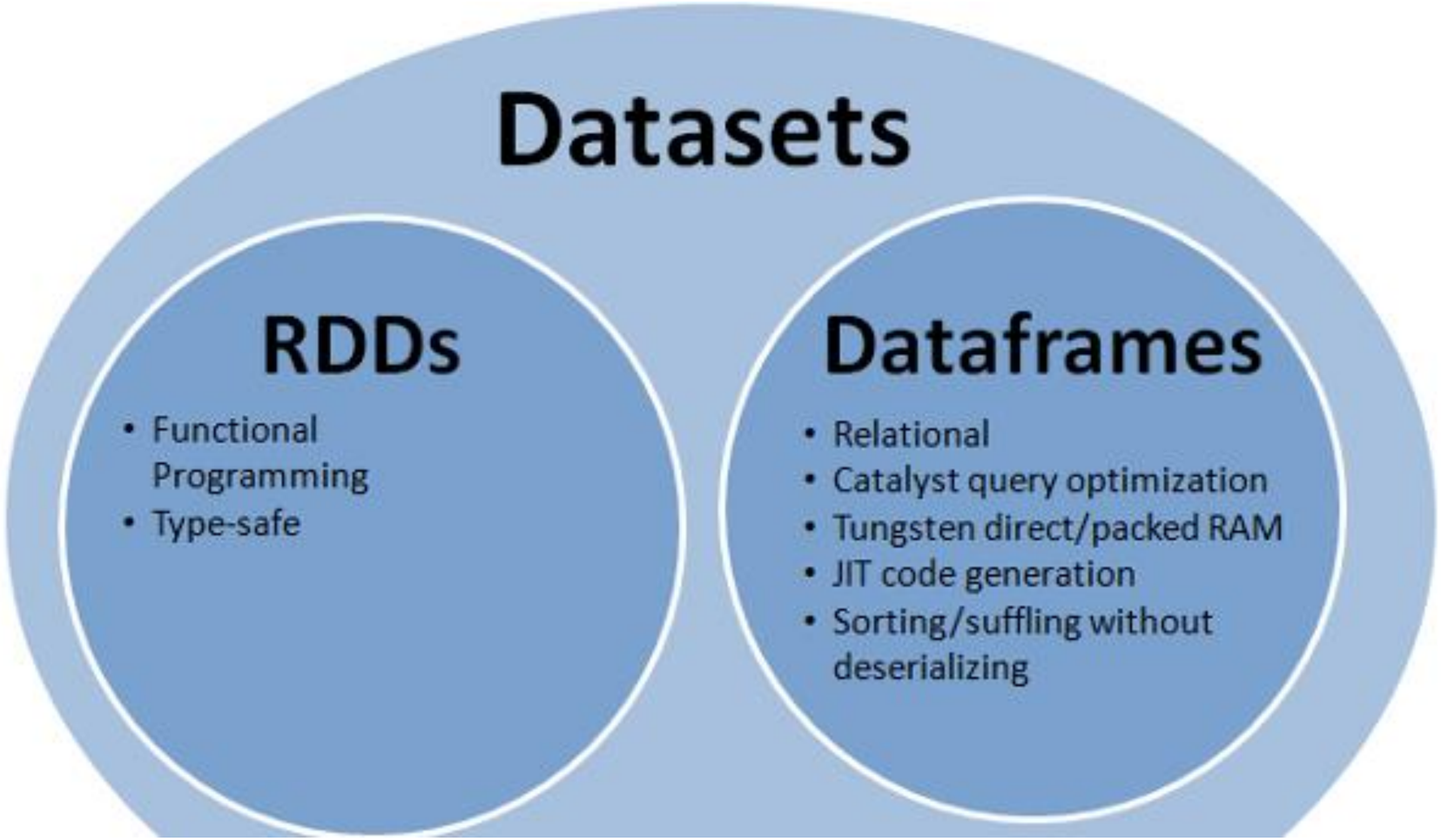
Almost the "Best of both worlds": **type safe + fast**

But slower than DF
Not as good for interactive analysis, especially Python

databricks

esprit

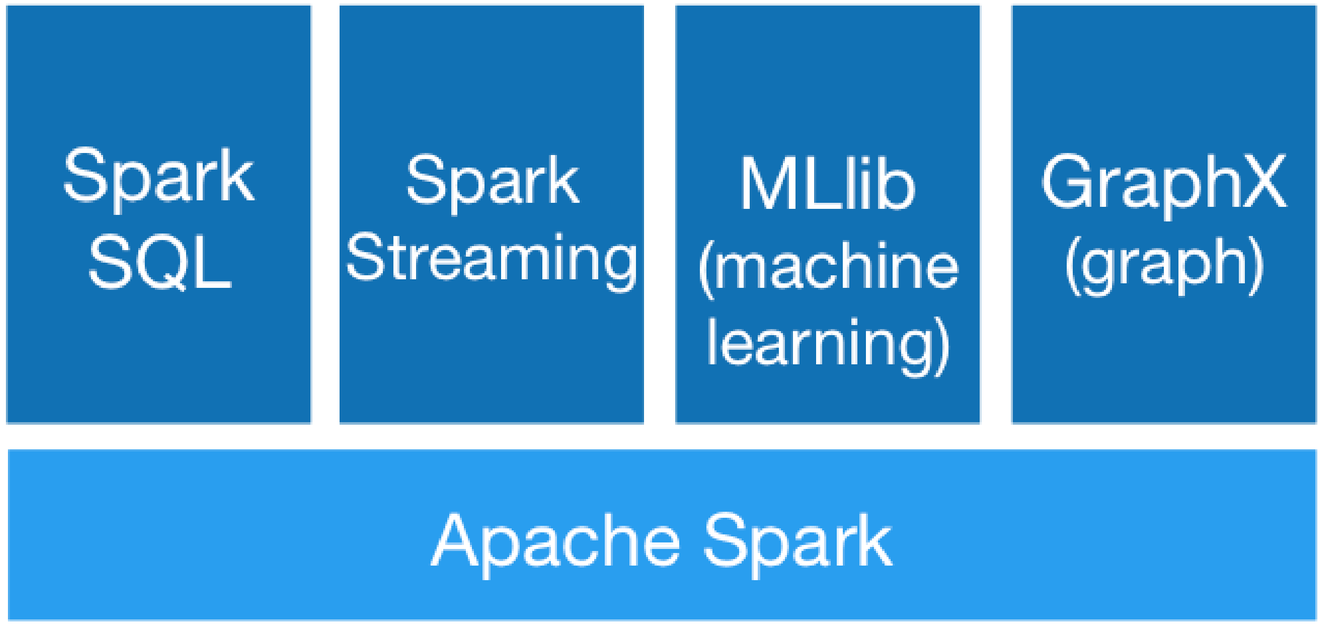
Dataframes et Datasets



API Spark

- Documentation https://spark.apache.org/docs/latest/
- Plusieurs langages hôtes
  - Java
  - Scala (langage fonctionnel sur JVM)
  - Python
- Documentation Scala : http://www.scala-lang.org/api/current/#package
- Tutoriel : http://docs.scala-lang.org/tutorials/

Spark Stack



- ***Spark SQL :*** pour le traitement de données (SQL et non structuré)
- ***Spark Streaming :*** traitement de flux de données en direct (live streaming)
- ***MLlib :*** Algorithmes Machine Learning
- ***GraphX :*** Traitement de graphes

Scala : vue d'ensemble

- Langage orienté-objet et fonctionnel :
  - Orienté objet : valeur à objet, opération à méthode
    - Par exemple : l'expression 1+2 signifie l'invocation de '+ ' sur des objets de la classe Int
  - Fonctionnel :
    1. Les fonctions se comportent comme des valeurs : peuvent être retournées ou passées comme arguments
    2. Les structures de données sont immuables (immutable) : les méthodes n'ont pas d'effet de bord, elles associent des valeurs résultats à des valeurs en entrée

Scala : vue d'ensemble

- Immuabilité des données

```
//1- déclarer une variable et lui associer une valeur
scala> var a=3
a: Int = 3
//2- vérifier la référence attribuée par Scala
scala> a
res0: Int = 3
//3- associer à a une nouvelle valeur, 5
scala> a=5
a: Int = 5
//même chose que 2
scala> a
res1: Int = 5
//manipuler l' ''ancienne'' valeur de a via res0
scala> res0*2
res2: Int = 6
```

Scala : vue d'ensemble

- Variable Vs. Valeur

```
//1- déclarons une valeur n
scala> val n=1+10
n: Int = 11
//2-essayons de la modifier
scala> n=n+1
<console>:12: error: reassignment to val
       n=n+1
         ^
//3- déclarons une variable
scala> var m=10
m: Int = 10
//4- idem que 2
scala> m=m+1
m: Int = 11
```

Scala : vue d'ensemble

- Inférence de types

```
scala> var a=1
a: Int = 1
scala> var a="abc"
a: String = abc
scala> var a=Set(1,2,3)
a: scala.collection.immutable.Set[Int] = Set(1, 2, 3)
scala> a+=4
res1: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)
scala> a+="a"
<console>:9: error: type mismatch;
 found   : String
 required: scala.collection.immutable.Set[Int]
              a+="a"
               ^
```

Scala : vue d'ensemble

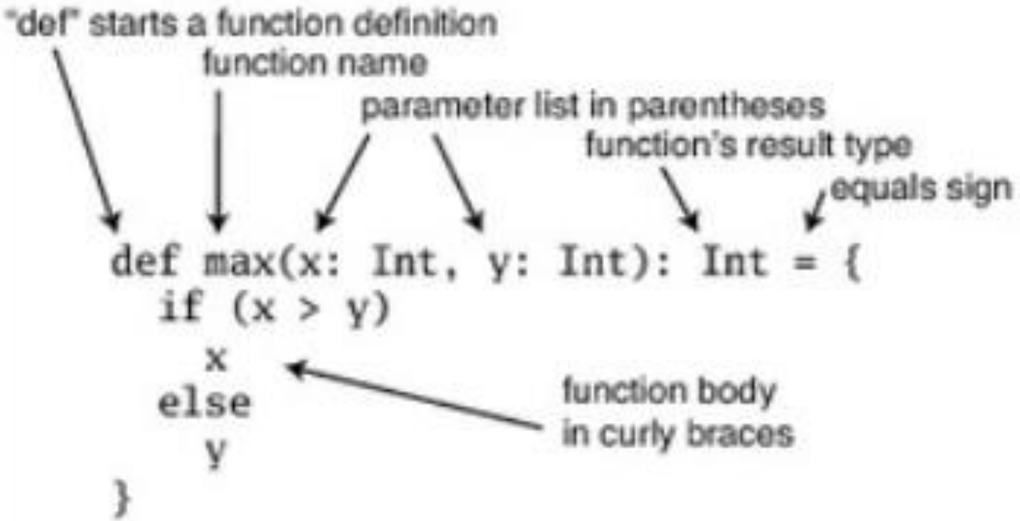
- Fonctions

```
scala> def max2(x: Int, y: Int) = if (x > y) x else y
max2: (x: Int, y: Int)Int

scala> max2(1,3)
res3: Int = 3

scala> max2(res3,0)
res5: Int = 3

scala> max2(max2(1,2),3)
res6: Int = 3
```



Scala : vue d'ensemble

- Itérations avec *foreach*
  - Style de programmation impérative
  - Méthode associé à un tableau (ou liste, ou ensemble)
  - Prend en entrée une fonction, souvent print

```
//déclarer une liste et l'initialiser
scala> var l=List(1,2,3)
l: List[Int] = List(1, 2, 3)
//imprimer chaque élément de la liste
scala> l.foreach(x=>print(x))
123
//syntaxe équivalente
scala> l.foreach(print)
123
```

Scala : vue d'ensemble

- Tableaux
  - Collections d'objets typés
  - Initialisation directe ou avec apply()
  - Mise à jour directe ou avec update()

```
scala> val b=Array.apply("1","2","3") //Initialisation avec apply
b: Array[String] = Array(1, 2, 3)
scala> b(0)="33" //mise à jour directe
scala> b.update(1,"22 ") //mise à jour avec en utilisant update
```

Scala : vue d'ensemble

- Listes et ensembles
  - Collections d'objets typés **immuables**
  - Initialisation directe
  - Mise à jour impossible

```
scala> val da=List(1,2,3) //initialisation directe
da: List[Int] = List(1, 2, 3)
scala> da(2) //accès indexé
res53: Int = 3
scala> da(0)=1 //tentative de mise à jour
<console>:9: error: value update is not a member of List[Int]
              da(0)=1
              ^
```

Scala : vue d'ensemble

- Opérations sur les listes
  - Concaténation avec :::, ajout en tête avec ::
  - Inverser l'ordre d'une liste reverse()
  - Et plein d'autres méthodes

```
scala> val l1=List(1,2,3)
l1: List[Int] = List(1, 2, 3)
scala> val l2=List(4,5)
l2: List[Int] = List(4, 5)
scala> l1:::l2
res44: List[Int] = List(1, 2, 3, 4, 5)
```

```
scala> (6::l2)
res47: List[Int] = List(6, 4, 5)
scala> val l1bis=1::2::3::Nil
l1bis: List[Int] = List(1, 2, 3)
//deviner la sorti de cette instruction
scala> l1:::(6::l2.reverse).reverse
```

Scala : vue d'ensemble

- Tableaux associatifs
  - Associer à chaque entrée un élément
  - Extension avec +

```
scala> var capital = Map("US" -> "Washington", "France" -> "Paris")
capital: scala.collection.immutable.Map[String,String] = Map(US ->
Washington, France -> Paris)
scala> capital("US")
res2: String = Washington
scala>   capital += ("Japan" -> "Tokyo")
```

Scala : vue d'ensemble

- Fonctions d'ordre supérieur Map et Reduce

```
scala> List(1, 2, 3) map (z=>z+1) //est équivalent à la ligne suivante
scala> List(1, 2, 3).map (_ + 1)
res71: List[Int] = List(2, 3, 4)
//rappel: capital désigne Map(US -> Washington, France -> Paris)
scala> capital.map(z=>(z._1.length))
res77: scala.collection.immutable.Iterable[Int] = List(2, 6)

scala> capital.reduce((a,b) =>  if(a._1.length>b._1.length) a else b)
res7: (String, String) = (France,Paris)
scala> capital+=("Algeria"->"Algiers")
scala> capital.reduce((a,b) =>  if(a._1.length>b._1.length) a else b)
res10: (String, String) = (Algeria,Algiers)
```

Scala sous Spark

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Scala sous Spark – Map Reduce

Préparation des données

```
scala> val lines=sc.textFile("/cours/mesures.txt")
lines: org.apache.spark.rdd.RDD[String] …
scala> lines.count
res3: Long = 5
scala> lines.collect
res4: Array[String] = Array(7,2010,04,27,75,
12,2009,01,31,7, ….
```

```
7,2010,04,27,75
12,2009,01,31,78
41,2009,03,25,95
2,2008,04,28,76
7,2010,02,32,91
```

/cours/mesures.txt

Map (f:T⇒U)

```
scala> lines.map(x=>x.split(",")).collect
res8: Array[Array[String]] = Array(Array(7, 2010, 04, 27, 75), Array(12, 2009, 01, 31,
7), …

scala> lines.map(x=>x.split(",")).map(x=>(x(1),x(3))).collect
res12: Array[(String, String)] = Array((2010,27), (2009,31), …
```

Scala sous Spark – Map Reduce

Map (f:T⇒U)

```
scala> lines.map(x=>x.split(",")).collect
res8: Array[Array[String]] = Array(Array(7, 2010, 04, 27, 75), Array(12, 2009, 01, 31,
7), ...

scala> lines.map(x=>x.split(",")).map(x=>(x(1),x(3))).collect
res12: Array[(String, String)] = Array((2010,27), (2009,31), ...
```

ReduceByKey (f:(V,V) ⇒ V)

```
//convertir l'entrée en entier pour pouvoir utiliser ReduceByKey!
scala> val v=lines.map(x=>x.split(",")).map(x=>(x(1).toInt,x(3)))
v: org.apache.spark.rdd.RDD[(Int, String)] = MappedRDD[7] at map at <console>:14
scala> val max=v.reduceByKey((a,b)=>if (a>b)a else b).take(10)
max: Array[(Int, String)] = Array((2010,32), (2008,28), (2009,31))
```

Scala sous Spark – Map Reduce

ReduceByKey (f:(V,V) ⇒ V)

```
//convertir l'entrée en entier pour pouvoir utiliser ReduceByKey!
scala> val v=lines.map(x=>x.split(",")).map(x=>(x(1).toInt,x(3)))
v: org.apache.spark.rdd.RDD[(Int, String)] = MappedRDD[7] at map at <console>:14
scala> val max=v.reduceByKey((a,b)=>if (a>b)a else b).take(10)
max: Array[(Int, String)] = Array((2010,32), (2008,28), (2009,31))
```

Comportement du *ReduceByKey*

```
scala> val max=v.reduceByKey((a,b)=>a).take(10)
max: Array[(Int, String)] = Array((2010,27), (2008,28), (2009,31))

scala> val max=v.reduceByKey((a,b)=>b).take(10)
max: Array[(Int, String)] = Array((2010,32), (2008,28), (2009,25))
```