# Algèbre Spark

Master DAC – Bases de Données Large Echelle
Mohamed-Amine Baazizi
baazizi@ia.lip6.fr
Octobre 2017

# L'API Spark

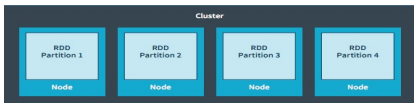
- Programme Scala : parallélisme intra-machine
- Contexte big data : parallélisme inter-machine
  - distribution de données et des traitements
- API Spark
  - surcouche au-dessus de Scala
  - gestion de la distribution des données
  - implantation du *Map* et *ReduceByKey* + autres opérateurs algébriques



# L'abstraction RDD

- Comment rendre la distribution des données et la gestion des pannes transparente?

→*Resilient Distributed Datasets (RDDs)*

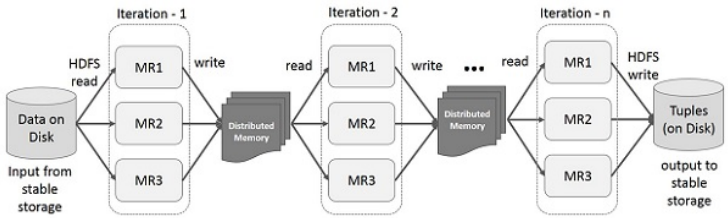



  - structures de données distribuées en mémoire centrale/ HDFS – représente une séquence d'enregistrements
  - accessibles en lecture seule – traitement parallèle
  - restriction aux opérations sur gros granules



# Traitement sur les RDD

- Chargement depuis HDFS → chaine de traitement → écriture résultat final



# Traitement sur les RDD : exemple

chaîne de traitements classique
1- Chargement depuis fichier
2- Application d'un filtre simple
3- Calcul de la cardinalité

```
1 val lines = spark.textFile("file.txt")
2 val data =
lines.filter(_.contains( "word"))
3  data.count
```

**Lazy evaluation**

Construire les RDDs seulement si action (la méthode *count*)
Avantage : chargement *sélectif* de file.txt



# Deux types de traitements

*A la base du modèle d'exécution de Spark*

Transformations

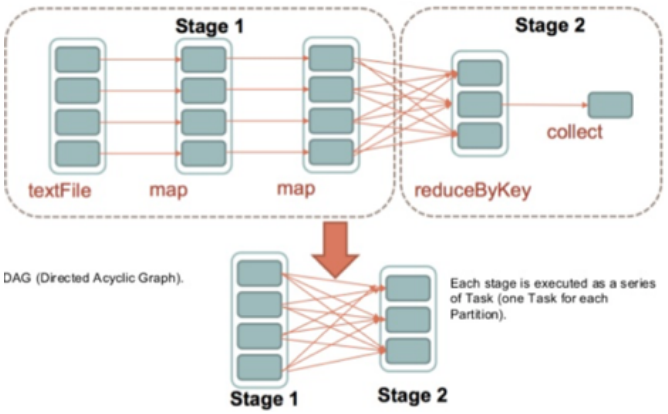
opérations qui s'enchainent mais qui ne s'exécutent pas
opérations pouvant souvent être combinées
Ex. *map, filter, join*

Actions

opérations qui lancent un calcul distribué
elles déclenchent toute la chaine de transformations qui la précède
Ex. *count, save, collect*



# Deux types de traitements






# API 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 |

# Illustration d'un programme Spark

```
scala> val lines=sc.textFile("/user/cours/mesures.txt")
lines: org.apache.spark.rdd.RDD[String] ...

scala> lines.count
res3: Long = 5

scala> lines.collect

scala> lines.first

scala> lines.take(10)
```

**Interdit** si *lines* volumineux!!

```
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
```

/user/cours/mesures.txt



# Illustration d'un programme Spark

- *Map* (*f:*T⇒U)

```
scala> lines.map(x=>x.split(","))
res0: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[1] ...

scala> val v= lines.map(x=>x.split(",")).map(x=>(x(1) .toInt,x(3)))
v: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[3]…
```

- *ReduceByKey* (*f:*(V,V) ⇒ V )

```
scala> val max=v.reduceByKey((a,b)=>if (a>b)a else b)
max: org.apache.spark.rdd.RDD[(Int, String)] = ShuffledRDD[4] …
```



# Pattern matching dans Scala

Condition exprimée dans map

*Map* (*f:*T⇒U) : RDD[T]⇒ RDD[U]

f peut s'exprimer avec case(exp) => U

```
scala> val zipcode = sc.parallelize(Array(("Paris", 75), ("Lyon", 69)))
zipcode: org.apache.spark.rdd.RDD[(String, Int)] = …

scala> val city = zipcode.map(x=>x._1)
city: org.apache.spark.rdd.RDD[String] = …

scala> val city = zipcode.map{case(ville,code)=>ville}
city: org.apache.spark.rdd.RDD[String] = …
```

Avantage : lisibilité du code!



# Pattern matching dans Scala

Condition exprimée dans filter

*Filter* (*f:*T⇒bool) : RDD[T]⇒ RDD[T]

f peut s'exprimer avec case(exp) => bool

```
scala> val zipcode = sc.parallelize(Array(("Paris", 75), ("Lyon",69), ("Cayenne", 973)))
zipcode: org.apache.spark.rdd.RDD[(String, Int)] =

scala> val dom = zipcode.filter{case(ville,code)=>code>100}
dom: org.apache.spark.rdd.RDD[(String, Int)] = ..
```

# Reduce

*Reduce* (f:(T,T)⇒T) : RDD[T]⇒ T

7,**2010**,04,**27**,75
12,**2009**,01,**31**,78
41,**2009**,03,**25**,95
.....

```
scala> val lines=sc.textFile("/user/cours/mesures.txt")
lines: org.apache.spark.rdd.RDD[String] = ...

scala> val tab = lines.map(x=>(x.split(",")(3).toInt))
tab: org.apache.spark.rdd.RDD[Int] = ...

scala> tab.collect
res1: Array[Int] = Array(27, 31, 25, 28, 32)

scala> val sum_all = tab.reduce(_+_)
moy_all: Int = 143

scala> val prod_all = tab.reduce(_*_)
moy_all: Int = 18748800

scala> val chose = tab.reduce(_-_)
-25

scala> val chose = tab.reduce(_-_)
-29
```

vérification

**La fonction f doit être associative**
→ à la base du parallélisme



# Reduce

***RDD.reduce* (f:(T,T)⇒T) : RDD[T]⇒ T**
f commutative et associative car aucun ordre des éléments dans RDD

```
scala> val range = ('a' to 'z').map(_.toString)
scala> val rddrange = sc.parallelize(range) // distribuer range

scala> println(range.reduce(_ + _))
abcdefghijklmnopqrstuvwxyz

scala> println(rddrange .reduce(_ + _))
 abcwxyzjklmnopqrsdefghituv
```

ordre perdu



# API Spark : RDD paires clé-valeur

| | | | |
|---|---|---|---|
| **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 |



# Transformations

**Une RDD paire clé/val**

- groupByKey
- reduceByKey
- combineByKey
- mapValues
- flatMapValues
- keys
- values
- sortByKey

**Deux RDD paire clé/val**

- substractByKey
- join
- rightOuterJoin
- leftOuterJoin
- cogroup

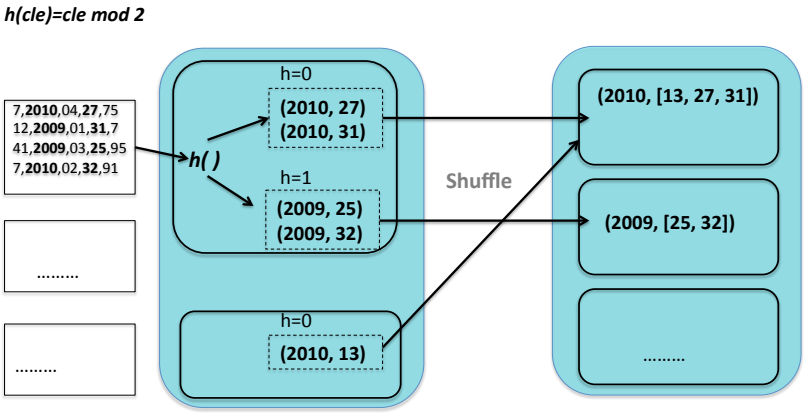
# groupByKey

Regroupe les valeurs ayant la même clé. Nécessite un shuffle
groupByKey(): RDD[(K, Iterable[V])]

```
scala> val lines=sc.textFile("user/cours/mesures.txt")
lines: org.apache.spark.rdd.RDD[String] = …

scala> val tab = lines.map(x=>x.split(","))
tab_s: org.apache.spark.rdd.RDD[Array[String]] = ...

scala> val tup = tab.map(x=>(x(1).toInt, x(3).toDouble))
tab: org.apache.spark.rdd.RDD[(Int, Double)] = ...

scala> tup.groupByKey()
res1: org.apache.spark.rdd.RDD[(Int, Iterable[Double])] = ShuffledRDD[10] ....
```



# Illustration de groupByKey



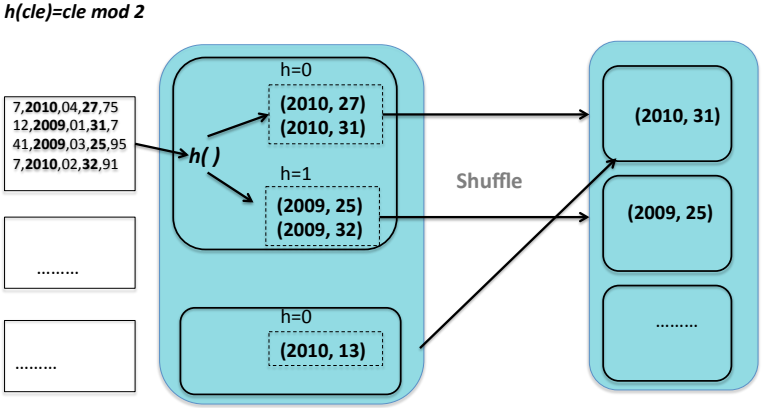

# reduceByKey

Le reduce au sens Map/Reduce – Shuffle nécessaire.
reduceByKey(func: (V, V) ⇒ V): RDD[(K, V)]

```
scala> tup // liste des paires (annee, temp)
res4: org.apache.spark.rdd.RDD[(Int, Double)] = MapPartitionsRDD[4] at map


scala> val chose = tup.reduceByKey((A,B)=>if (A>B) A else B)
chose: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[33]...
```



# Illustration résultat reduceByKey



# combineByKey

Analogue à aggregate sauf que traite des paires clé-valeur

1. createCombiner()
2. mergeValue
3. mergeCombiners

```
scala> tuple // liste des paires (annee, temp)
res37: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[21] ...

scala>  val struct_moy = tuple.combineByKey(
      (v) => (v, 1),                          ← createCombiner()
      (acc:(Int, Int), v) => (acc._1+v, acc._2+1), ← mergeValue ()
      (acc1:(Int, Int), acc2:(Int, Int)) => (acc1._1+acc2._1, acc2._1+acc2._2) ←
mergeCombiners()  )
struct_moy: org.apache.spark.rdd.RDD[(Int, (Int, Int))] = ShuffledRDD[23]
```

Exemple : (2010,4), (2009,1), (2009,3), (2008,4), (2010,2)



# Join

- S'applique à une RDD (table R), et prend une autre RDD en paramètre (table S)
- Chacune des RDD doit être partitionnée sur l'attribut/les attributs de jointure

```
scala> R // liste des tuples (A, B, C)
scala> S// liste des tuples (C, D, E)
//calculer R join S on R.C=S.C

Scala> R.map{case(a,b,c)=>(c, (a,b))}.join(S.map{case(c,d,e)=>(c,(d,e))})
//porduit une RDD de la forme (c, ((a,b), (d,e)))
```