

BDLE Partie 2 : Programmation en Map-Reduce sous Spark

- 30-10-2020
 - Introduction à MapReduce - Spark et Scala
- 06-11-2020
 - Interrogation de données structurées en Spark
- 20-11-2020
 - Aperçu du modèle d'exécution Map-Reduce et Spark - Optimisation logique
- 27-11-2020
 - Ouverture : préparation de données pour le machine Learning

Canaux de communication

- Le [wiki](#) entrée principale
 - Diapos de cours, liens vers notebooks Databricks
- [Moodle](#) science
 - Envoi de message par équipe pédagogique
 - Dépôt rendu des devoirs
- [Mattermost](#)
 - Entraide, @all pour cibler tout les participants (@baazizi pour me cibler si besoin)
- [Zoom](#)
 - Lien permanent, sert pour les cours et les TME

Infos générales

- Modalités d'évaluation
 - Dans l'attente de précisions du Master info
 - Devoirs maisons notés (pondération à définir)
 - Examen final très probable
- Devoir maison partie 2
 - Sujet du TME 27-11-2020
 - Préparation de données pour la classification
 - Utilisation API Dataframe et ML de Spark
- Conseils
 - Travail personnel requis
 - Finir les TME_{*n*} avant de commencer le TME_{*n+1*}
- Documentation
 - Articles de recherches (conférences SIGMOD, VLDB, ICDE, EDBT)
 - Événements Spark (Spark AI Summit)
 - Medium, Stack overflow, ...

Introduction à Map Reduce, à Spark et à Scala

Master DAC – Bases de Données Large Echelle

Mohamed-Amine Baazizi

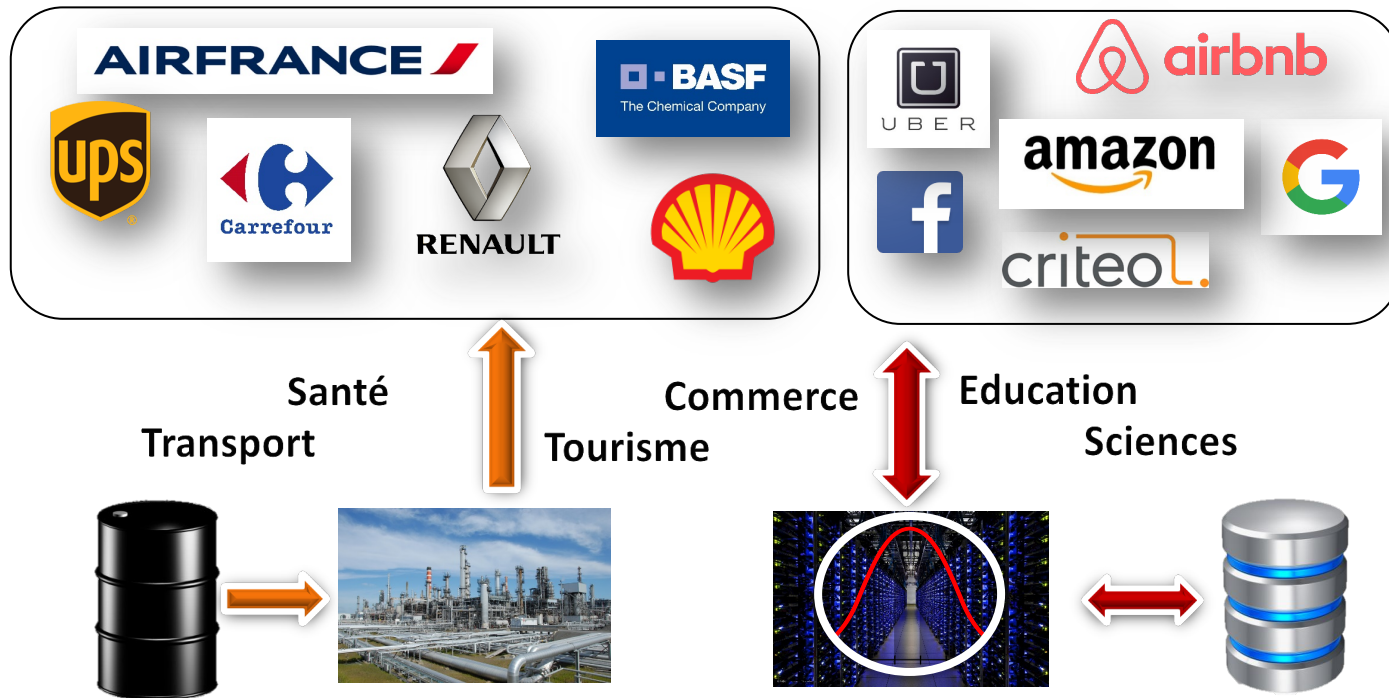
mohamed-amine.baazizi@lip6.fr

2020-2021

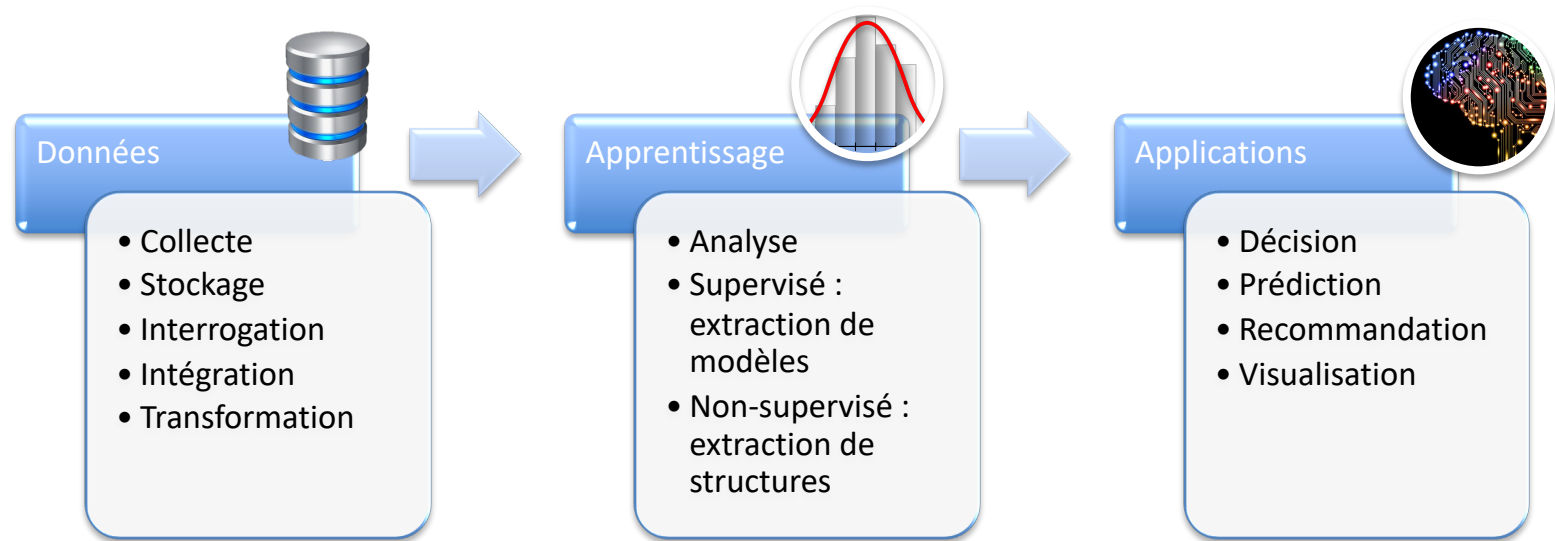
Plan

- Contexte et historique (30')
- Bases de Scala (40')
- L'algèbre RDD de Spark (30')

Données = Le pétrole du 21e siècle

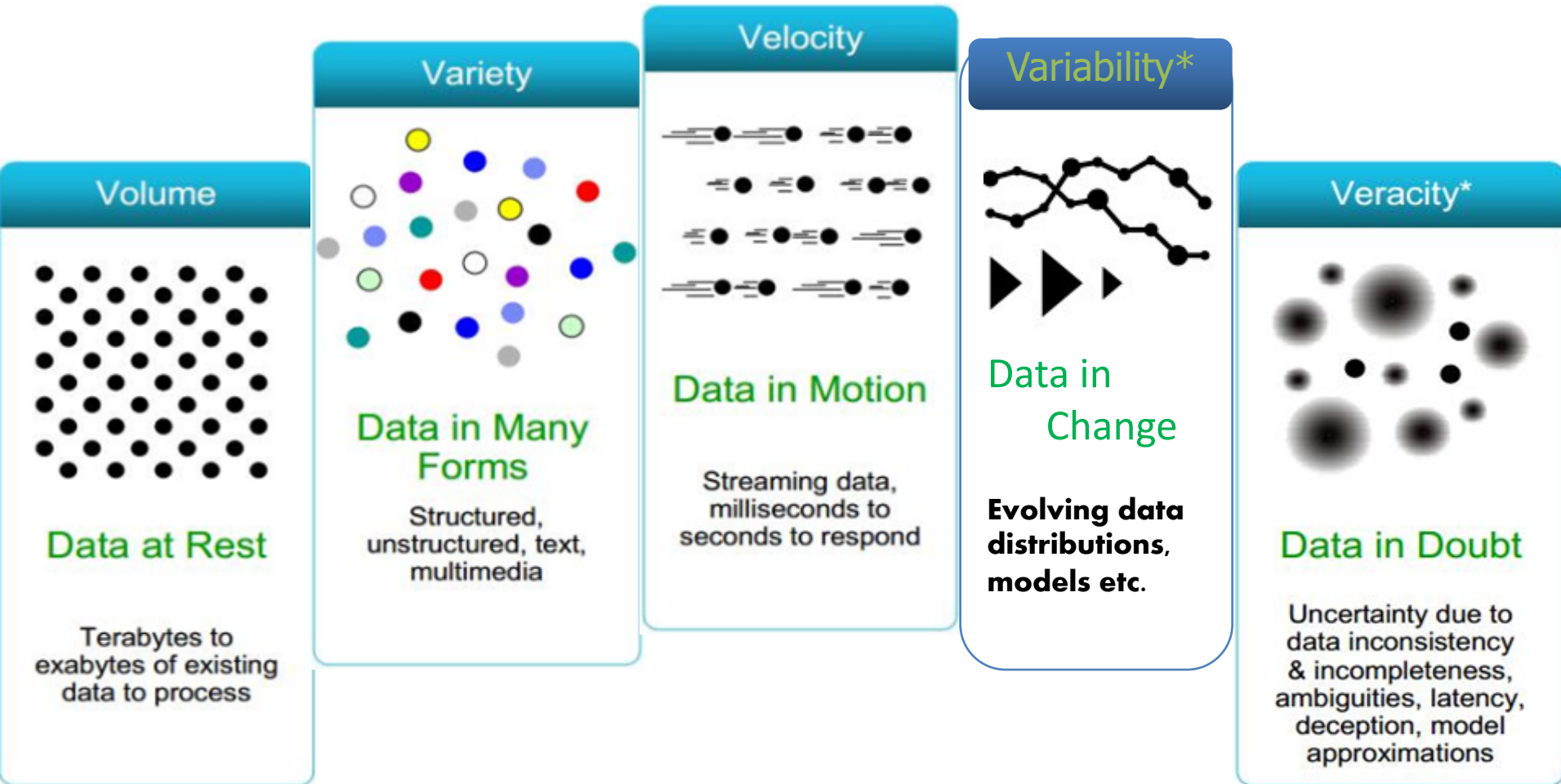


Big Data Pipeline



- Cycle more than a pipeline
- Data quality is a crucial issue: garbage in = garbage out

Caractéristiques du big data



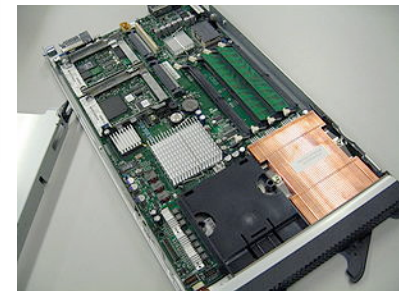
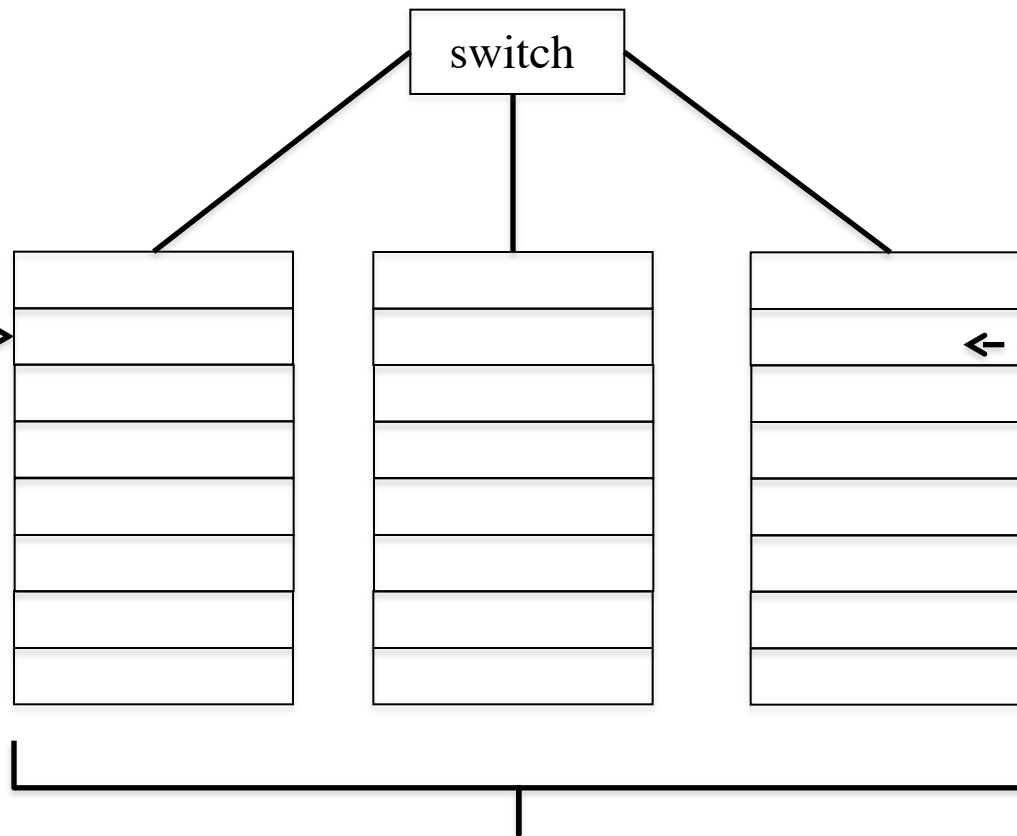
Relever le défi big data

- Systèmes distribués type cluster
 - à base de machines standard (*commodity machines*)
 - extensibles à volonté (architecture RAIN)
 - faciles à administrer et tolérants aux pannes
- Modèle de calcul distribué Map Reduce
 - calcul massivement parallèle, mode *shared nothing*
 - abstraction de la parallélisation (pas besoin de se soucier des détails sous-jacents)
 - plusieurs implantations (Hadoop, Spark, Flink...)

Architecture d'un cluster

Lame de calcul
= [8,64] unités

Unité de calcul
= CPU+RAM+DDur



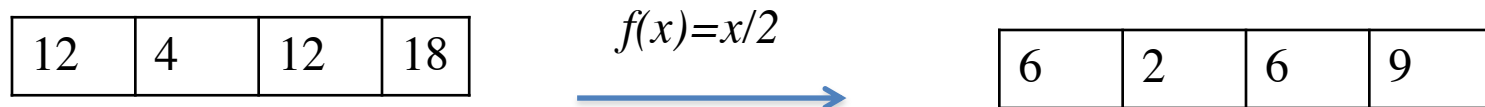
Cluster

Origine du modèle Map Reduce

- **Rappel** : calcul massivement parallèle, mode *shared nothing*
- Programmation fonctionnelle fonctions d'ordre supérieur
 - *Map* ($f: T \Rightarrow U$), f unaire : appliquer f à chaque élément de C
 - *Reduce* ($g: (T, T) \Rightarrow T$), g binaire

Illustration

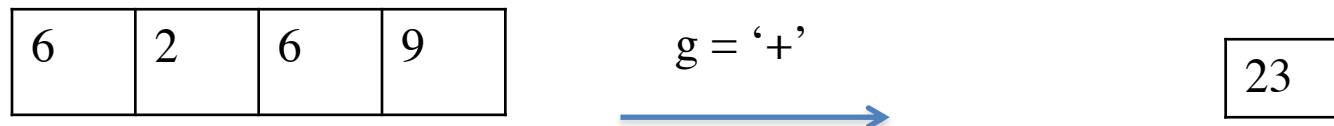
- *Map* ($f: T \Rightarrow U$), f unaire : appliquer f à chaque élément de C



la dimension de C est préservée le type en entrée peut changer

- *Reduce* ($g: (T,T) \Rightarrow T$), g binaire

- agréger les éléments de C deux à deux



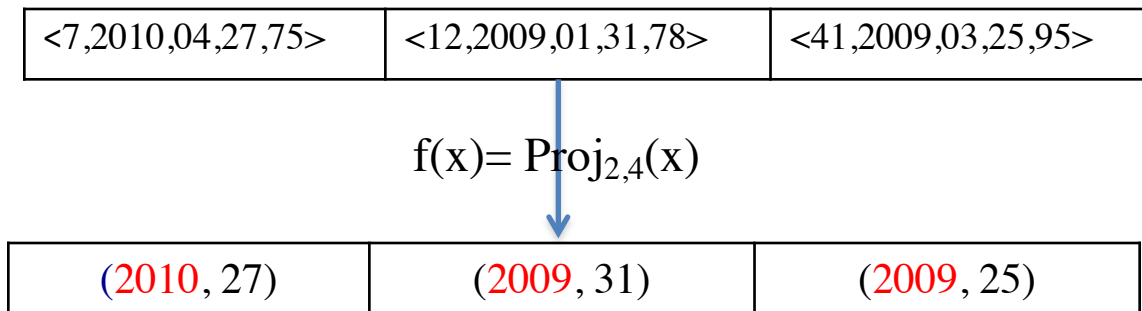
réduit la dimension de n à 1 le type en sortie identique à celui en entrée

Adaptation pour le big data

- Type de données
 - logs de connections, transactions, interactions utilisateurs, texte, images
 - structure homogène (schéma implicite)
- Type de traitements
 - Aggrégations (count, min, max, avg) → group by
 - Autres traitements (indexation, parcours graphes, Machine Learning)

Map Reduce pour le big data

- Les données en entrée sont des nuplets : identifier attribut de groupement (appelé clé)
- *Map* ($f: T \Rightarrow (k, U)$), f unaire
 - produire une paire (clé, val) pour chaque val de C



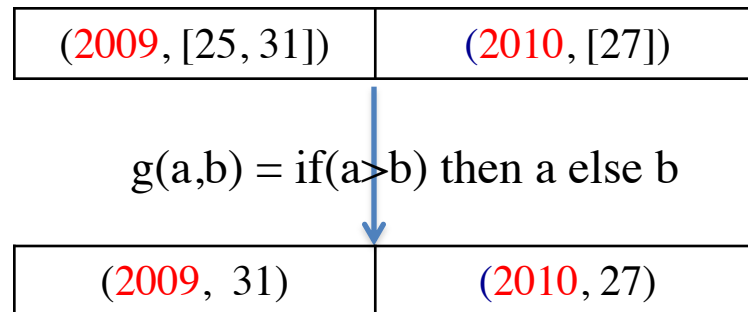
- Regrouper les paires ayant la même clé pour obtenir (clé, [list-val])

$(2009, [25, 31])$	$(2010, [27])$
--------------------	----------------

Map Reduce pour le big data

– *Reduce* ($g: (T,T) \Rightarrow T$), g binaire

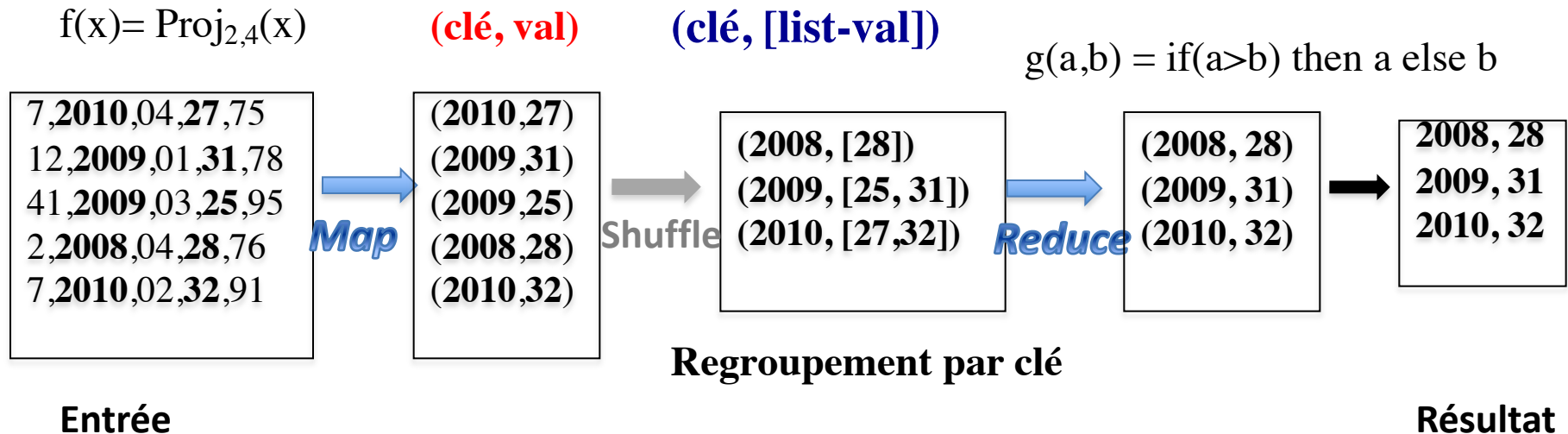
- pour chaque (clé, [list-val]) produit (clé, val) où $val = g([list-val])$



- **Important** : dans certains systemes, g doit être **associatif** car ordre de traitement des élément de C non présctit

Map Reduce : Exemple

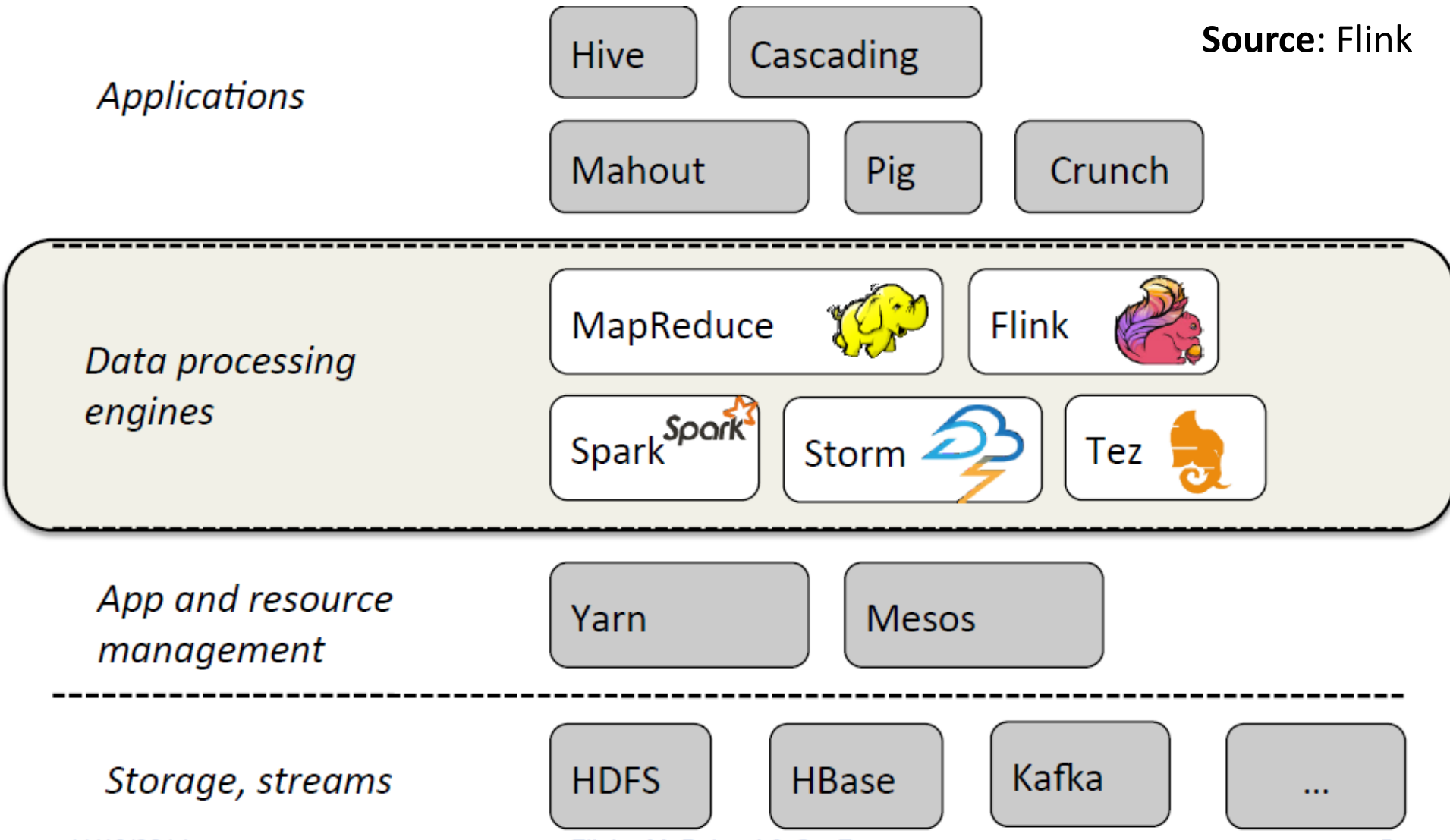
- Entrée : n-uplets (station, annee, mois, temp, dept)
- Résultat : select annee, Max(temp) group by annee



Plateformes Map Reduce

- **Traitement distribué**
 - Hadoop MapReduce (Google, 2004)
 - Spark (projet MPLab, U. Stanford, 2012)
 - Flink (projet Stratosphere, TU Berlin, 2009)
- **Stockage**
 - Hadoop FS, Hbase, Kafka
- **Scheduler**
 - Yarn, Mesos
- **Systèmes haut niveau**
 - Pig, Hive, Spark SQL

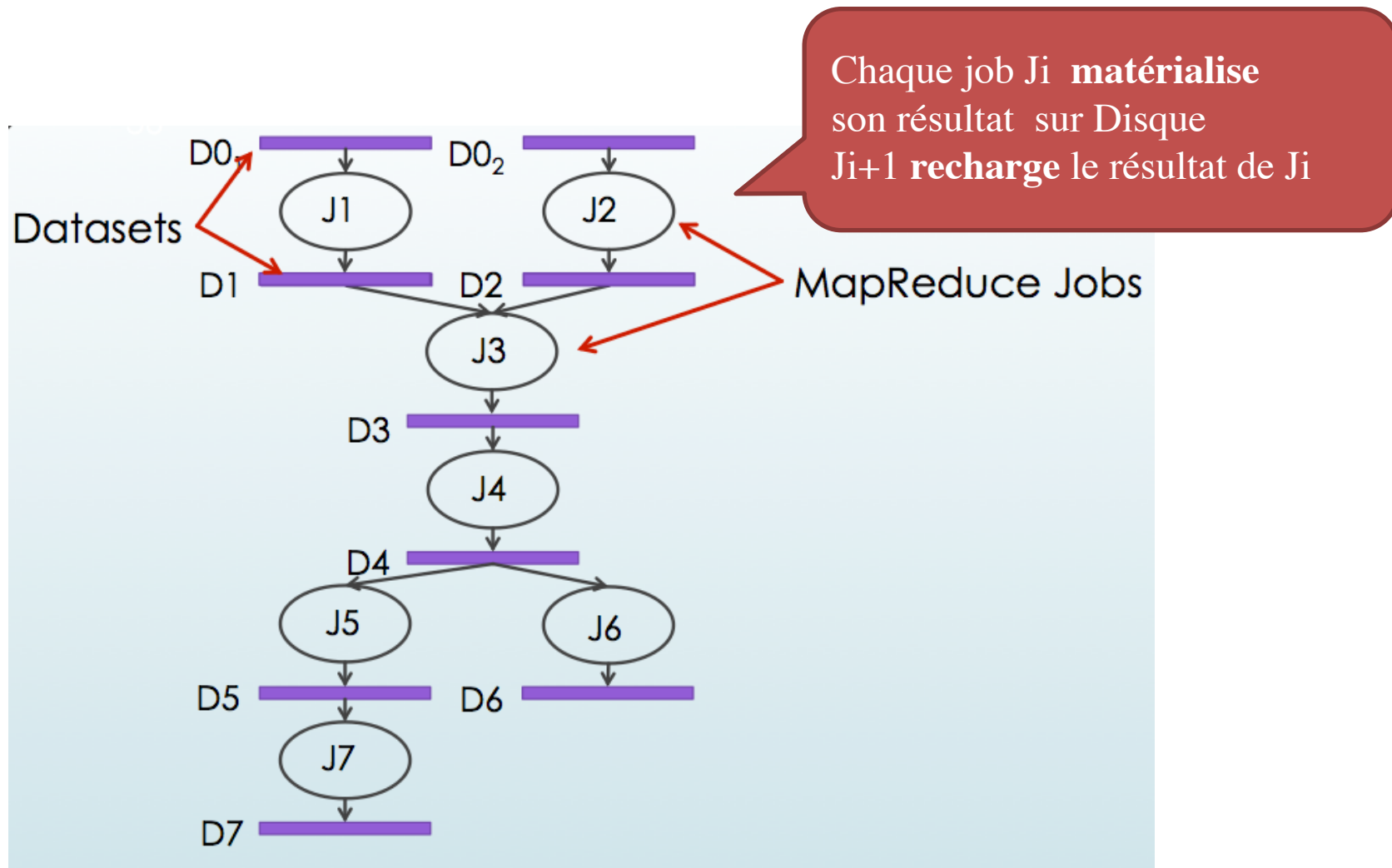
Open Source Big Data Landscape



Hadoop Map Reduce

- Introduit par Google en 2004
- Répondre à trois principales exigences
 - Utiliser cluster de machines standards
 - extensibles à volonté (architecture RAIN)
 - facilité d'administration, tolérance aux pannes
- Ecrit en Java. Utilisation autre langages possible
- Plusieurs extensions
 - Pig et Hive pour langage de haut niveau
 - HaLoop (traitement itératif), MRShare (optimisation)

Hadoop Map Reduce

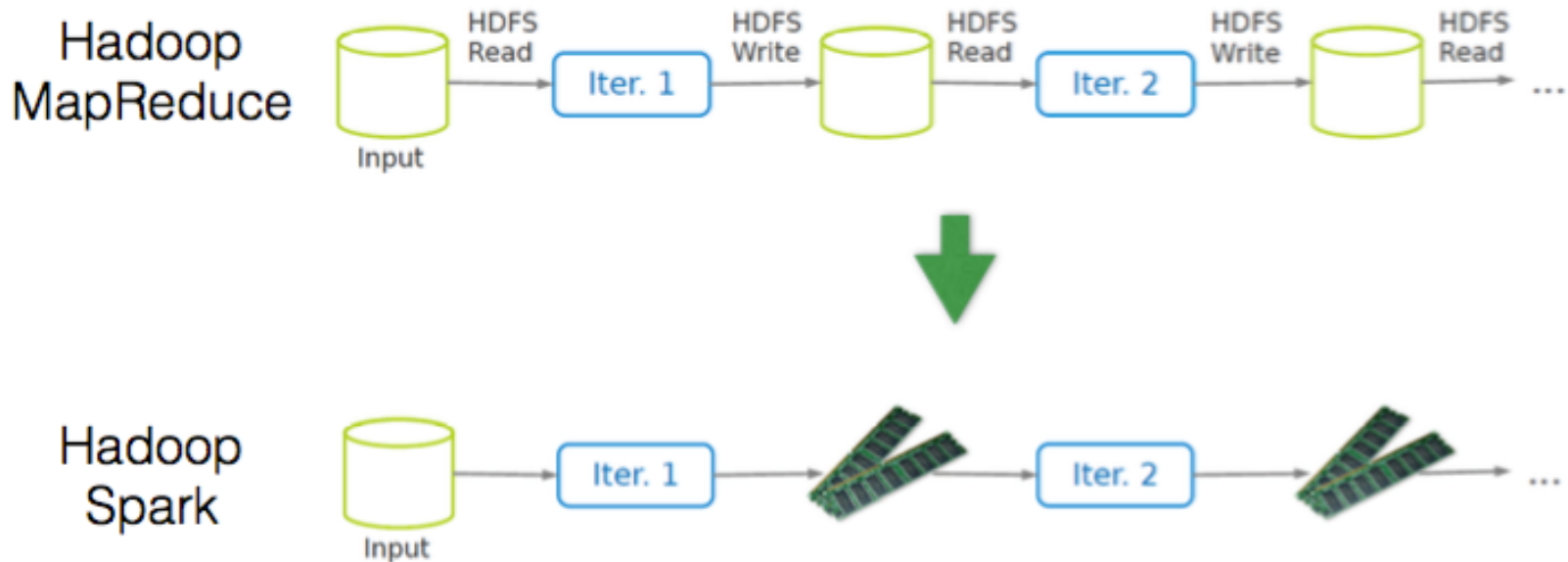


Limites de Hadoop Map Reduce

- Traitement complexes = performances dégradées
 - Traitement complexe = plusieurs étapes
 - Solution naïve : matérialiser résultat de chaque étape
 - avantages : reprise sur panne performante
 - inconvénients : coût élevé d'accès au disque
 - Solution optimisée : pipelining et partage
- Inadapté aux traitements itératifs (ML et analyse graphes)
- Pas d'interaction avec l'utilisateur

Spark

- Résoudre les limitations de Map Reduce
 - Matérialisation vs *persistance* en mémoire centrale
 - *Batch processing* vs interactivité (Read Execute Print Loop)



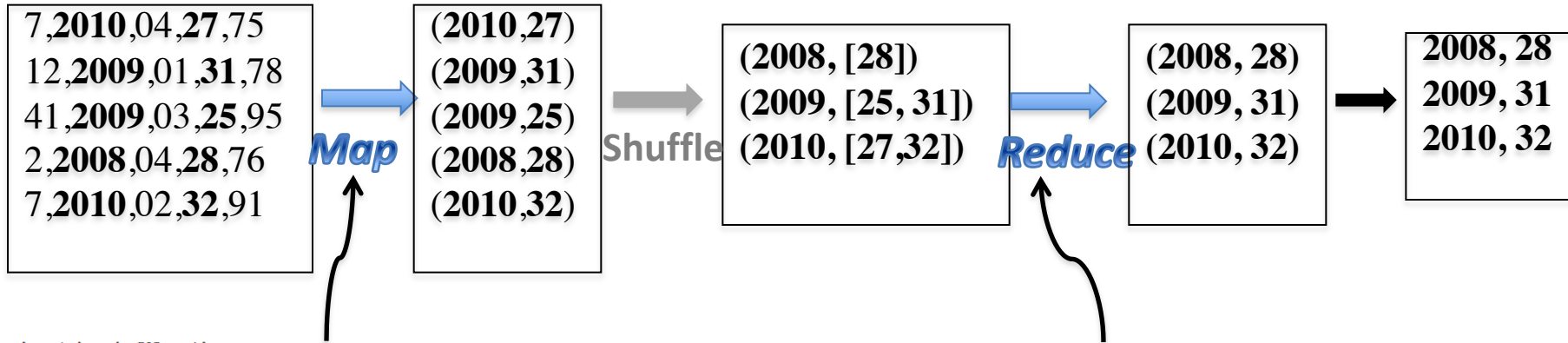
Spark

- *Resilient Distributed Dataset (RDD)*
 - collection logique de données distribuées sur plusieurs nœuds
 - traitement gros granule (pas de modification partielle)
 - tolérance aux pannes par réexécution d'une chaîne de traitement
- Réutilisation de certains mécanismes de Map Reduce
 - HDFS pour le stockage des données et résultat
 - Quelques similitude dans le modèle d'exécution

Pour ce cours

- Choix du système : Spark
 - *Framework* assez complet pour la préparation et l'analyse
 - Système interactif et de production à la fois
- Plusieurs langages hôtes
 - Scala (natif), Java, Python et R
 - API pour données structurées (relationnelles, JSON, graphes)
- Choix du langage hôte : Scala
 - langage natif de Spark
 - fonctionnel et orienté objet
 - concis, haut niveau
 - typage statique
 - détecter certains erreurs avant exécution

Programmer en Map Reduce



```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

    private static final int MISSING = 9999;

    @Override
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        String line = value.toString();
        String year = line.substring(15, 19);
        int airTemperature;
        if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
            airTemperature = Integer.parseInt(line.substring(88, 92));
        } else {
            airTemperature = Integer.parseInt(line.substring(87, 92));
        }
        String quality = line.substring(92, 93);
        if (airTemperature != MISSING && quality.matches("[01459]")) {
            context.write(new Text(year), new IntWritable(airTemperature));
        }
    }
}
```

Java

```
import java.io.IOException;

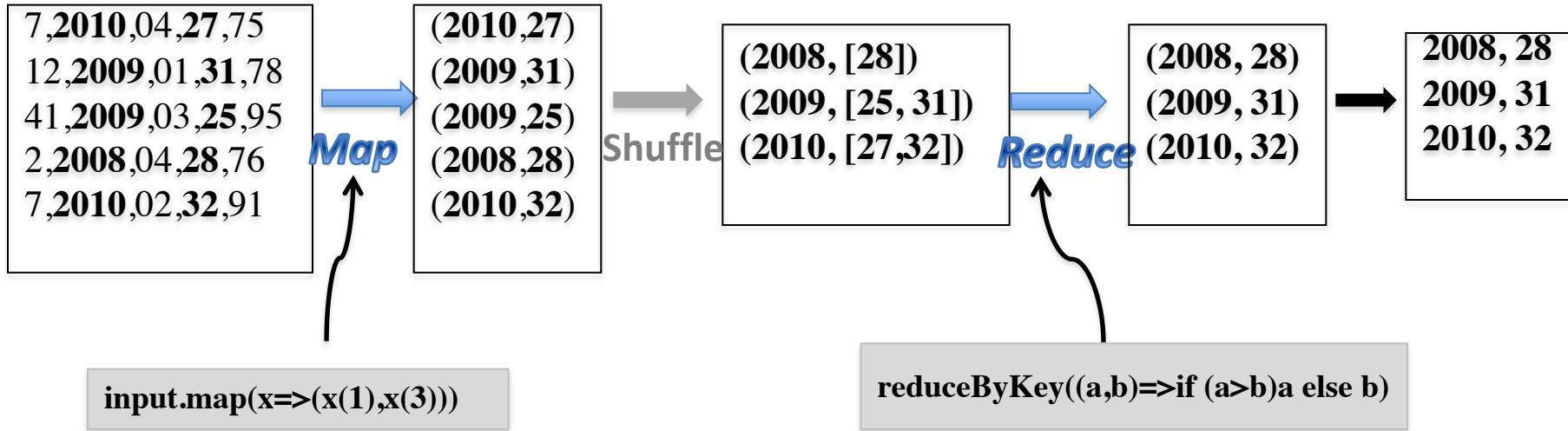
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MaxTemperatureReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

    @Override
    public void reduce(Text key, Iterable<IntWritable> values,
        Context context)
        throws IOException, InterruptedException {

        int maxValue = Integer.MIN_VALUE;
        for (IntWritable value : values) {
            maxValue = Math.max(maxValue, value.get());
        }
        context.write(key, new IntWritable(maxValue));
    }
}
```

Programmer en Spark



Scala

L'API RDD de Spark

- API de base
 - abstraction du parallélisme inter-machine
 - implantation du *Map* et *ReduceByKey* + autres opérateurs algébriques
 - surcouche au-dessus de Scala
 - gestion de la distribution des données (partitionnement)
 - persistance de données

Bases de Scala

Scala en quelques mots

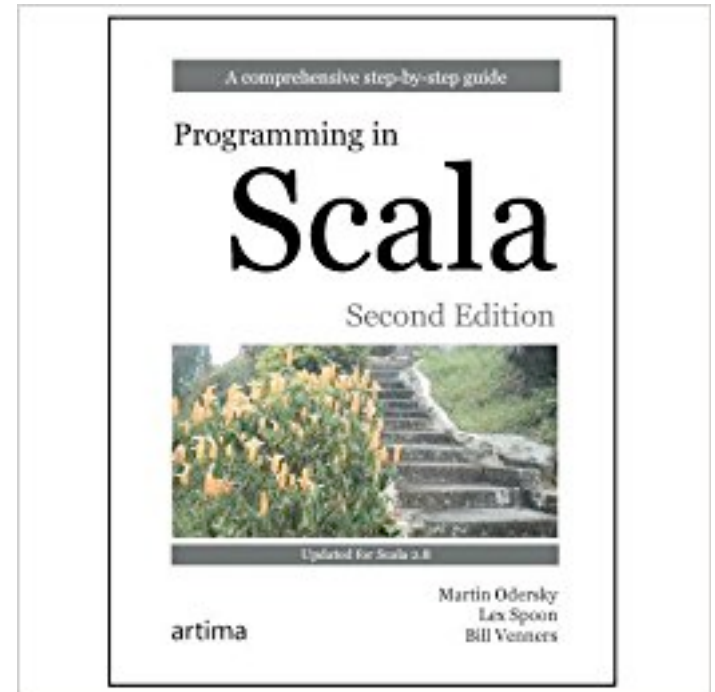
- Langage orienté-objet et fonctionnel à la fois
 - Orienté objet : valeur $\rightarrow$ objet, opération $\rightarrow$ méthode
Ex: l'expression `1+2` signifie l'invocation de la méthode `'+'` sur des objets de la classe `Int`
 - Fonctionnel
 - Les fonctions se comportent comme des valeurs : peuvent être retournées ou passées comme arguments
Ex: `Map(x=>f(x))` avec `f(x) = x/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
Ex: `c=[2, 4, 6]` `c.Map(x=>f(x))` produit une nouvelle liste `[1, 2, 3]`

Avantages de Scala

- Compatibilité avec Java
 - compilation pour JVM, types de base de Java (Int, float, ..)
- Syntaxe concise
 - Nb ligne Scala = 50% NB lignes Java (en moyenne)
- Haut niveau d'abstraction
 - possibilité de cacher des détails à l'aide d'interfaces
- Typage statique
 - éviter certaines erreurs pendant l'exécution
- Inférence de type
 - code plus concis que les langages avec typage statique

Plan

- Premiers pas
- Types et opérations de base
- Structures de contrôle
- Types complexes
- Fonctions d'ordre supérieur



Référence bibliographique

M. Odersky, L. Spoon, B. Venners. *Programming in Scala*. 2nd Edition. 2012

https://booksites.artima.com/programming_in_scala_2ed

Ligne de commande

- Mode interactif

```
$ spark-shell  
...  
scala>
```

Manipulations de base

```
scala> 1+2  
res0: Int = 3
```

```
scala> res0+3  
res1: Int = 6
```

res0	la valeur calculée
:Int	le type inféré
=3	la valeur calculée

```
scala> println("hello")  
hello  
  
scala>
```

Valeurs vs variables

- les valeurs sont immuables, i.e elles ne peuvent être modifiées

```
scala> val n=10
```

```
n: Int = 11
```

```
scala> n=n+1
```

```
<console>:12: error: reassignment  
to val
```

```
    n=n+1
```

```
    ^
```

```
scala> var m=10
```

```
m: Int = 10
```

```
scala> m=m+1
```

```
m: Int = 11
```

On ne peut réaffecter une nouvelle valeur à *n* car déclarée avec **val**

On peut incrémenter *m* car déclarée avec **var**

Définition des fonctions

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

```
max2: (x: Int, y: Int)Int
```

```
scala> max(1,3)
```

```
res3: Int = 3
```

```
scala> max(max(1,2),3)
```

```
res6: Int = 3
```

Diagram illustrating the syntax of a function definition:

- "def" starts a function definition
- function name
- parameter list in parentheses
- function's result type
- equals sign
- function body in curly braces

```
def max(x: Int, y: Int): Int = {  
  if (x > y)  
    x  
  else  
    y  
}
```

Le type retour inféré automatiquement sauf pour les fonctions récursives

Type par défaut Unit : correspond à void en Java

```
scala> def bonjour() = println ("bonjour")
```

```
bonjour: ()Unit
```

Types et opérations de base

Table 5.1 · Some basic types

Value type	Range
Byte	8-bit signed two's complement integer (-2^7 to $2^7 - 1$, inclusive)
Short	16-bit signed two's complement integer (-2^{15} to $2^{15} - 1$, inclusive)
Int	32-bit signed two's complement integer (-2^{31} to $2^{31} - 1$, inclusive)
Long	64-bit signed two's complement integer (-2^{63} to $2^{63} - 1$, inclusive)
Char	16-bit unsigned Unicode character (0 to $2^{16} - 1$, inclusive)
String	a sequence of Chars
Float	32-bit IEEE 754 single-precision float
Double	64-bit IEEE 754 double-precision float
Boolean	true or false

- Opérateurs arithmétiques : + - * / %
- Opérateurs logiques : && || !
- Opérateurs binaires ...
- Conversions : toInt toDouble toLowerCase toUpperCase
 - à explorer en mode interactif

Conversions de type

- Plusieurs possibilités, à explorer en mode interactif

```
scala> val v = 124
```

```
v: Int = 124
```

```
scala> v.to
```

Taper TAB

```
to
```

```
toChar
```

```
toFloat
```

```
toLong
```

```
toShort
```

```
toBinaryString
```

```
toDegrees
```

```
toHexString
```

```
toOctalString
```

```
toString
```

```
toByte
```

```
toDouble
```

```
toInt
```

```
toRadians
```

```
scala> v.toInt
```

Taper TAB

```
def toInt: Int
```


Structures de contrôle

Rappel: Paradigme fonctionnel, les structures retournent une valeur

- Conditions
- Boucles : *while* et *foreach*
 - à éviter car style impératif
 - On privilégiera les *map* (cf. fonctions d'ordre supérieur)
- Pattern matching

Structures de contrôle

- Conditions

if (*cond*) *val* **else** *val*

```
scala> val chaine = "abcde"
scala> val longueur =
    if (chaine.length %2 ==0) "pair"
    else "impair"
longueur: String = impair
```

à éviter car
style impératif

- Boucles

while (*cond*) {*val*}

```
scala> var i = 2
scala> while (i<3) { println(i); i+=1 }
1
2
```

Structures de contrôle

- Boucles

`val.foreach(action)`

```
scala> val txt = "hello"

scala> txt.foreach(print)
hello
scala> txt.foreach(println)
h
e
l
l
o

scala>
```

Structures de contrôle

- pattern matching
 - branchement conditionnel à n alternatives
 - exprimés par un *pattern* dans la clause *case*

```
var match {  
  case v0 => res0  
  case v1 => res1  
  ...  
  case _ => res_defaut  
}
```

```
scala> def verif (age : Int) = age match {  
    case 25 => "argent"  
    case 50 => "or"  
    case 60 => "diamond"  
    case _ => "inconnu"  
  }
```

```
verif: (age: Int)String
```

```
scala> verif(10)
```

```
res7: String = inconnu
```

Types complexes

- le type tableau (Array)
 - séquence d'éléments (souvent du même type)
 - construction directe ou à partir de certaines fonctions comme le *split()*
 - accès indexé pour lecture ou écriture, indice 1^{er} élément = 0

```
scala> val weekend = Array("sam", "dim")  
weekend: Array[String] = Array(sam, dim)
```

```
scala> weekend(0)  
res6: String = sam
```

```
scala> weekend(1)  
res7: String = dim
```

0	1
"sam"	"dim"

Types complexes

- le type liste (List)
 - collection d'éléments (souvent du même type)
 - construction :
 - instantiation d'un objet List avec valeurs fournies
 - de manière récursive avec l'opérateur *cons* noté *elem::liste*
 - conversion d'un tableau

```
scala> val fruits = List("pomme", "orange", "poire")
```

instantiation

```
fruits: List[String] = List(pomme, orange, poire)
```

```
scala> val unAtrois = 1 :: 2 :: 3 :: Nil
```

concaténation

```
unAtrois: List[Int] = List(1, 2, 3)
```

```
scala> val lweekend = weekend.toList
```

conversion

```
lweekend: List[String] = List(ven, sam)
```

Types complexes

- Manipulation de listes
 - ajout en tête seulement (immuabilité)

```
scala> 4 :: unAtrois
```

```
res13: List[Int] = List(4, 1, 2, 3)
```

```
scala> val quatreAun = 4 :: unAtrois.reverse
```

```
quatreAun: List[Int] = List(4, 3, 2, 1)
```

- concaténation à l'aide de `:::` - l'ordre interne est préservé

```
scala> val quatreAcinq = 4 :: 5 :: Nil
```

```
quatreAcinq: List[Int] = List(4, 5)
```

```
scala> val unAcinq = unAtrois ::: quatreAcinq
```

```
unAcinq: List[Int] = List(1, 2, 3, 4, 5)
```

Types complexes

- dés-imbrication de listes : la méthode *flatten*

```
scala> val nestd_unAinq = List(unAtrois, quatreAinq)
nestd_unAinq: List[List[Int]] = List(List(1, 2, 3), List(4, 5))
```

```
scala> nestd_unAinq.flatten
res19: List[Int] = List(1, 2, 3, 4, 5)
```

Pourquoi
le type
Any ?

```
unAhuit: List[List[Any]] = List(List(List(1, 2, 3), List(4, 5)), List(6, 7))
```

```
scala> unAhuit.flatten
res24: List[Any] = List(List(1, 2, 3), List(4, 5), 6, 7)
```


Types complexes

- Tuples
 - Collection d'attributs relatifs à un object (cf. modèle rel.)
 - Accès indexé avec `._index` où *index* commence par 1
 - structure immuable, construits souvent à partir de sources externes (ex. fichier csv)

```
scala> val tuple = (12, "text", List(1,2,3))  
tuple: (Int, String, List[Int]) = (12,text,List(1, 2, 3))
```

```
scala> tuple._1 = 13  
<console>:12: error: reassignment to val  
  tuple._1 = 13  
      ^
```

Types complexes

- Tuples et pattern matching
 - possibilité de reconnaître la forme des tuples et d'enclencher un traitement spécifique en utilisant des variables

```
scala> val listeTemp = List((7,2010,4,27,75), (12,2009,1,31,78))
listeTemp: List[(Int, Int, Int, Int, Int)] = List((7,2010,4,27,75),
(12,2009,1,31,78))
```

```
scala> listeTemp.map{
    case(sid,year,month,value,zip)=>(year,value)
}
res0: List[(Int, Int)] = List((2010,27), (2009,31))
```

Types complexes

- Tableaux associatifs (map)
 - ensemble de paires (clé, valeur) - unicité de clé – clé et valeur de type quelconque mais fixés une fois pour toute
 - possibilité d'insertion et de mise à jour de nouvelles paires

```
scala> var capital = Map("US" -> "Washington", "France" -> "Paris")  
capital...
```

```
scala> capital("US")  
res2: String = Washington
```

```
scala> capital += ("US" -> "DC", "Japan" -> "Tokyo")
```

```
Map(US -> DC, France -> Paris, Japan -> Tokyo)
```

Types complexes

- Classes
 - Conteneurs pour objets ayant les mêmes attributs
 - *class MaClasse (nom: String, num: Int)*
 { //attributs et méthodes – partie optionnelle }

```
scala> class Mesure(sid:Int, year:Int, value:Float)
defined class Mesure
```

```
scala> listeTemp.map{case(sid,year,month,value,zip)=>new
Mesure(sid,year,value)}
res2: List[Mesure] = List(Mesure@364c93e6, Mesure@66589252)
```

Types complexes

- **case** Classes

- classes pour instancier des objets immuables
- utiles pour le pattern matching!
- ***case class** MaClasse (nom: String, num: Int)*
 { //attributs et méthodes – partie optionnelle }

```
scala> case class cMeasure(sid:Int, year:Int, value:Float)
defined class cMeasure
```

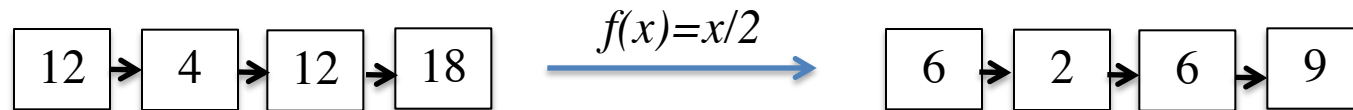
```
scala> listeTemp.map{case(sid,year,month,value,zip)=>
    cMeasure(sid,year,value)}
```

pas besoin de
new

Fonctions d'ordre supérieur

- *map*

Map ($f: T \Rightarrow U$), f unaire : applique f à chaque élément de C



la dimension de C est préservée mais le type en entrée peut changer

```
scala> def divide(n:Int) = n/2  
succ: (n: Int)Int
```

```
scala> val l= List(12,4,12,18)
```

```
scala> l.map(x=>divide(x))  
res1: List[Int] = List(6, 2, 6, 9)
```

Fonctions d'ordre supérieur

- *l.flatMap(f)* : équivalent de *l.map(f)* suivi de *flatten*

```
scala> def succ(n:Int) = n+1
```

```
succ: (n: Int)Int
```

```
scala> nestd_unAinq
```

```
res38: List[List[Int]] = List(List(1, 2, 3), List(4, 5))
```

```
scala> val deuxAsix = nestd_unAinq.flatMap(succ)
```

```
deuxAsix: List[Int] = List(2, 3, 4, 5, 6)
```

- *l.foreach(f)* : applique f à chaque élément sans retourner de valeur

```
scala> quatreAinq.foreach(println)
```

```
4
```

```
5
```

Fonctions d'ordre supérieur

- *filter(cond)*
 - *cond* retourne un booléen et permet de sélectionner les éléments de la liste sur laquelle *filter* est appliqué

```
deuxAsix: List[Int] = List(2, 3, 4, 5, 6)
```

```
scala> deuxAsix.filter(x=>x%2 ==0)
```

```
res46: List[Int] = List(2, 4, 6)
```

la condition s'exprime avec =>

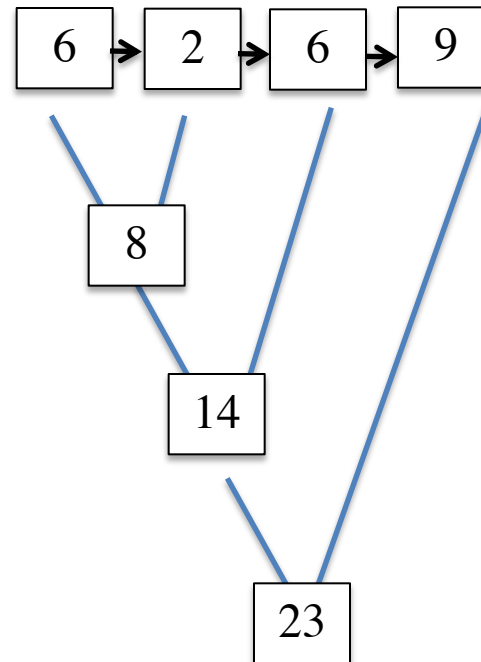
et signifie :

retourner x si x est multiple de 2

Fonctions d'ordre supérieur

- Réduction : *reduce*
 - $l.reduce(g: (T,T) \Rightarrow T)$: applique g sur les éléments de l

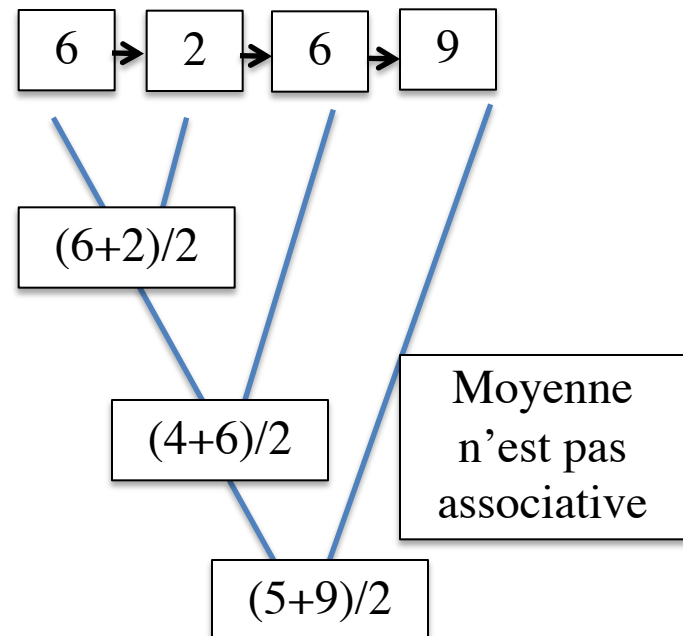
```
scala> def g(a:Int, b:Int) = {  
  println(a+"\t"+b)  
  a+b  
}  
  
scala> val l=List(6,2,6,9)  
  
scala> l.reduce((a,b)=>g(a,b))  
6    2  
8    6  
14   9  
res17: Int = 23
```



Fonctions d'ordre supérieur

- *reduce* ne marche que si *g* est associatif!
 - *l.reduce(g: (T,T)=>T)* : applique *g* sur les éléments de *l*

```
scala> def moyAB(a:Int, b:Int)=  
{  println(a+"\t"+b)  
  (a+b)/2  
}  
  
scala> val l=List(6,2,6,9)  
  
scala> l.reduce((a,b)=>moyAB(a,b))  
6    2  
4    6  
5    9  
res19: Int = 7
```



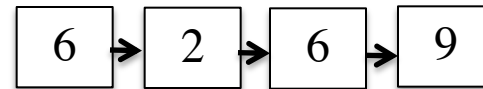
Fonctions d'ordre supérieur

- *reduce* n'a de sens que si *g* est associatif!
 - *l.reduce(g: (T,T)=>T)* : applique *g* sur les éléments de *l*

```
scala> val l=List(6,2,6,9)

scala> val sum = l.reduce((a,b)=>a+b)

scala> val moy = sum/l.count()
```



pour calculer le moyenne, il faut calculer la somme puis diviser sur le taille de la liste!

Bien entendu, on peut utiliser la fonction `avg()` prédéfinie.

L'algèbre RDD de Spark

L'abstraction RDD

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

→ *Resilient Distributed Datasets (RDDs)*

- Structure de données distribuées : séquence d'enregistrements de même type
- données distribuées → traitement parallèle
- immuabilité : chaque opérateur crée une nouvelles RDD
- évaluation *lazy* : plan pipeline vs matérialisation (M/R)

Exemple

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

1- Chargement depuis fichier

2- Application d'un filtre simple

3- Calcul de la cardinalité

Lazy evaluation : *count* déclenche le chargement de file.txt et le filter

Avantage : seules les lignes avec **"word"** sont gardées en mémoire

Deux types de traitements

A la base du modèle d'exécution de Spark

Transformations

opérations qui s'enchainent mais ne s'exécutent pas
opérations pouvant souvent être combinées

Ex. map, filter, join, reduceByKey

Actions

opérations qui lancent un calcul distribué
elles déclenchent toute la chaine de transformations qui la précède

Ex. count, save, collect

Opérateurs RDD

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

Opérateurs RDD

Map ($f:T \Rightarrow U$) : $\text{RDD}[T] \Rightarrow \text{RDD}[U]$

7,2010,04,27,75
12,2009,01,31,78
...
..
..

$\text{RDD}[\text{String}]$

$\text{map}(x \Rightarrow x.\text{split}(", "))$



7	2012	04	27	75
12	2009	01	31	78
...				
...				
...				

$\text{RDD}[\text{Array}[\text{String}]]$

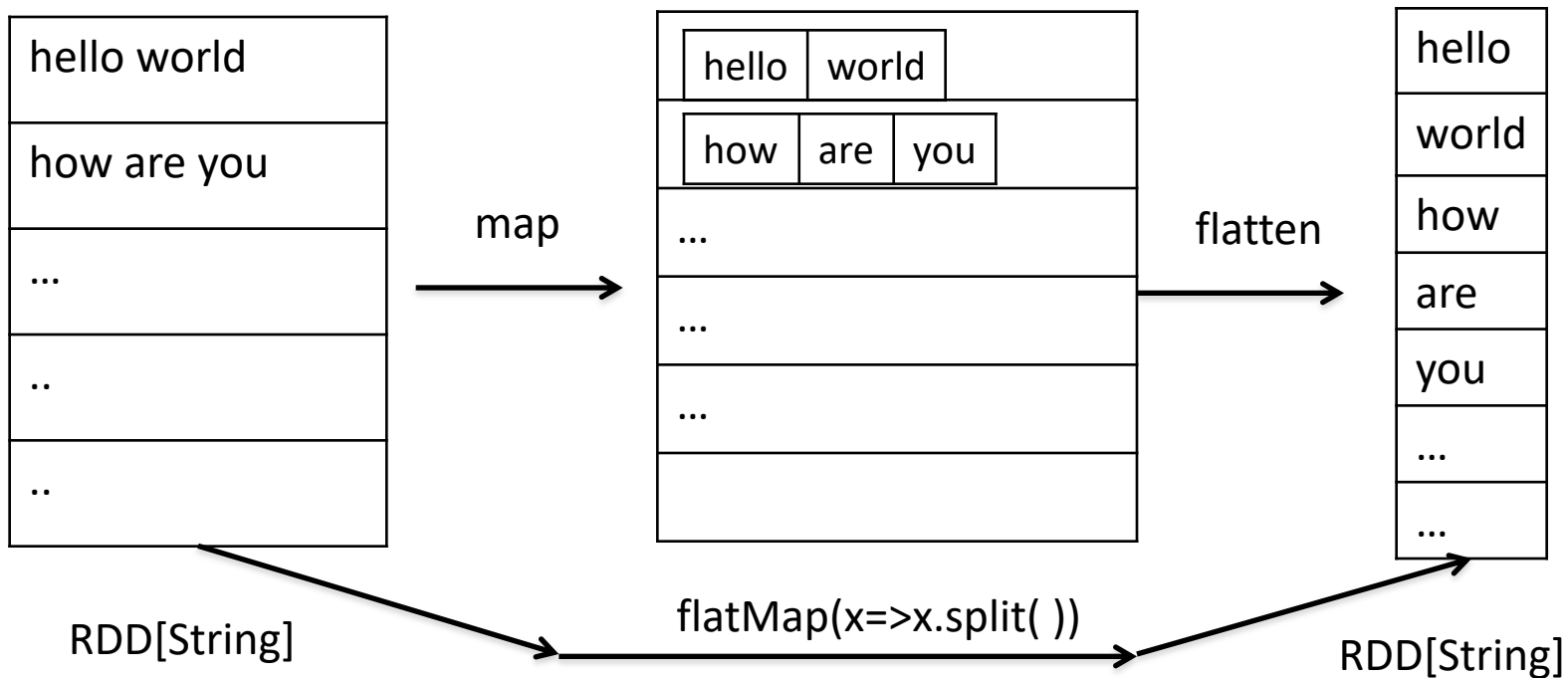
$\text{map}(x \Rightarrow (x(1).\text{toInt}, x(3)))$



?

Opérateurs RDD

flatMap ($f:T \Rightarrow \text{Seq}[U]$) : $\text{RDD}[T] \Rightarrow \text{RDD}[U]$



Opérateurs RDD

reduceByKey($f: (V, V) \Rightarrow V$) : $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$

(2010,27)
(2009,31)
(2008,28)
(2010,32)
(2009,25)

$\text{RDD}[(\text{Int}, \text{Double})]$

$\text{reduceByKey}((a,b) \Rightarrow \text{if } (a > b) a \text{ else } b)$



(2010,32)
(2009,31)
(2008,28)

$\text{RDD}[(\text{Int}, \text{Double})]$

Opérateurs RDD

join(): $(\text{RDD}[(K,V)], \text{RDD}[(K,W)]) \Rightarrow \text{RDD}[(K,(V,W))]$

films: $\text{RDD}[(\text{Int}, \text{String})]$

(1, ToyStory)
(2, Heat)
(3, Sabrina)

genres: $\text{RDD}[(\text{Int}, \text{String})]$

(1, Animation)
(2, Thriller)
(3, Comedy)

films.join(genres) →

(1, (ToyStory, Animation))
(2, (Heat, Thriller))
(3, (Sabrina, Comedy))

Opérateurs RDD

$reduce(f : (T,T) \Rightarrow T) : RDD[(T,T)] \Rightarrow T$

- Réduction de la dimension en utilisant une *User Defined Function (UDF)*
- Traitement distribué, aucun ordre prescrit
→ dans Spark f doit être commutative et associative!

Orange
Apple
Kiwi
...

$reduce((a,b) \Rightarrow a + '\t' + b)$



Kiwi Orange Apple

RDD et données structurées

- Constat sur l'utilisation des RDD
 - Pas d'exploitation du schéma par défaut
 - code peu lisible, programmation fastidieuse
 - Lorsque structure homogène, encapsuler chaque n-uplet dans un objet reflétant la structure
 - Performances dégradées (sérialisation d'objets, GC)
 - Absence d'optimisation logique (comme dans les SGBD)
- Pallier aux limites des RDD : API Dataset
 - Utiliser les schéma pour optimiser requêtes (SGBR)
 - et mieux organiser les données