Introduction à Map Reduce et à Spark

Master DAC – Bases de Données Large Echelle

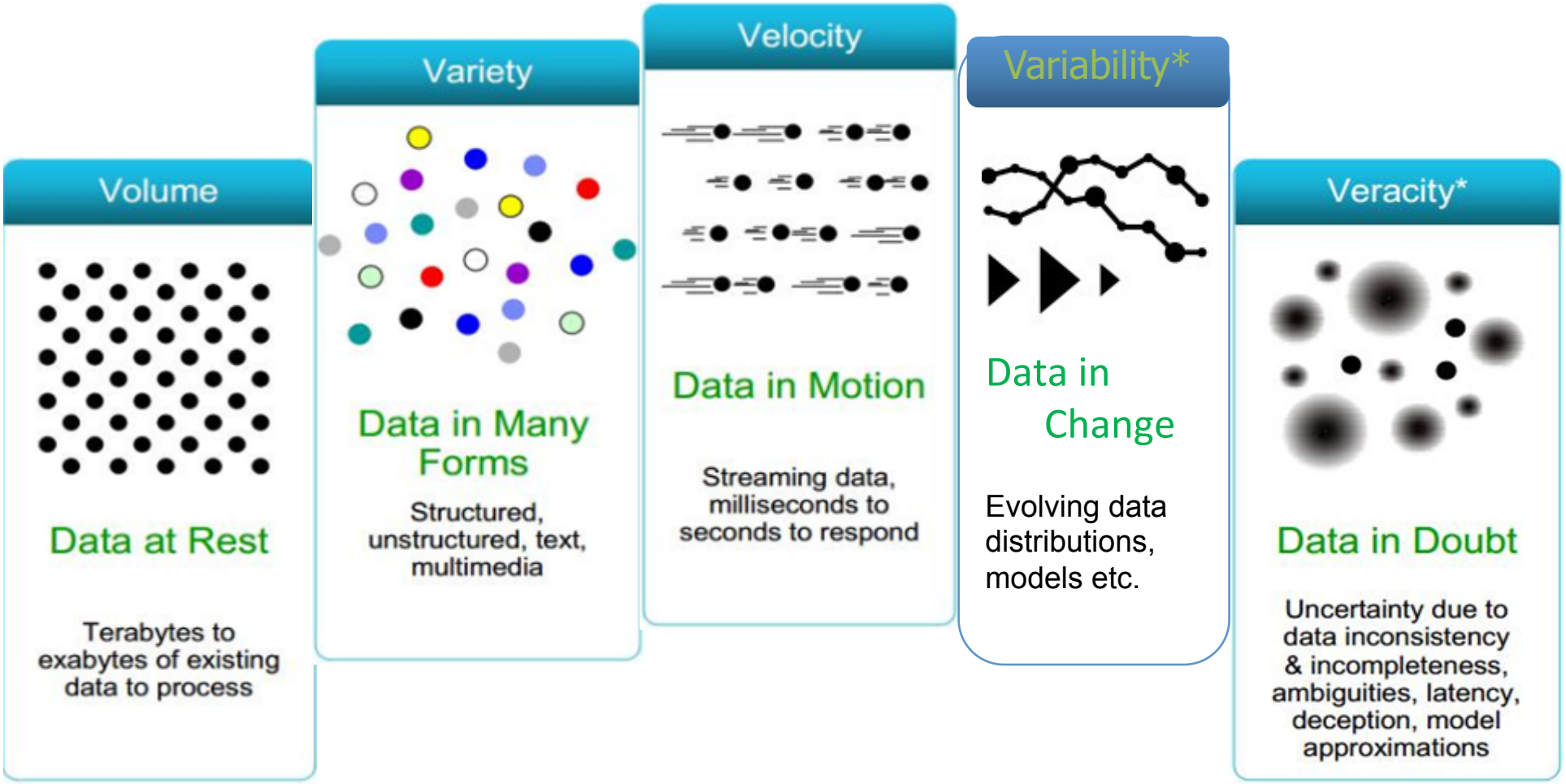
Mohamed-Amine Baazizi
baazizi@ia.lip6.fr

Octobre 2018

Contexte Big Data

- Nouvelles applications
  - Essor du web, indexation large volume de données
  - Réseaux sociaux, capteurs, GPS
  - Données scientifiques, séquençage ADN
- Analyses de plus en plus complexes
  - Moteurs de recherche
  - Comportement des utilisateurs
  - Analyse données médicales

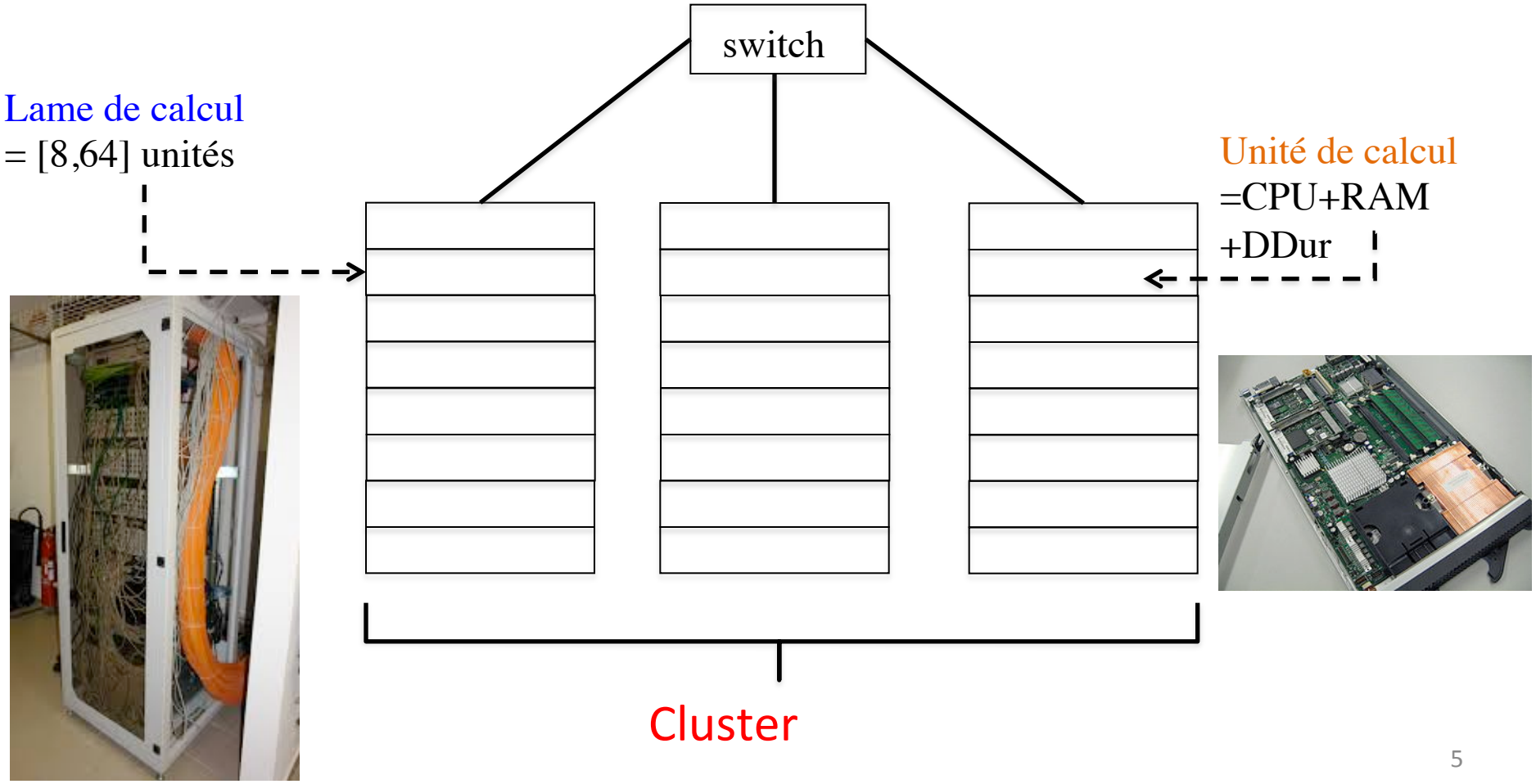
Caractérsitiques du big data



Relever le défi big data

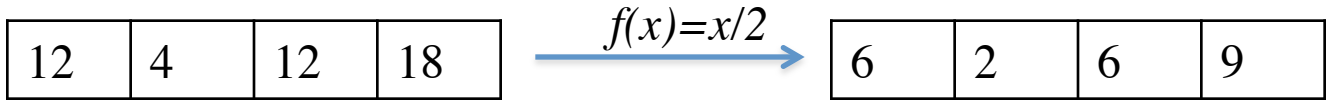
- Systèmes distribués type cluster
  - à base de machines standard (*commodity machine*s)
  - extensibles à volonté (architecture RAIN)
  - faciles à administrer et tolérants aux pannes
- Modèle de calcul distribué Map Reduce
  - calcul massivement parallèle
  - abstraction de la parallélisation (pas besoin de se soucier des détails sous-jacents)
  - plusieurs implantations (Hadoop, Spark, Flink…)

Architecture d'un cluster



Origine du modèle Map Reduce

- Programmation fonctionnelle : fonctions d'ordre supérieur appliquant une fonction aux objets d'une collection *C*
  - *Map (f:* T=>U), *f* unaire : appliquer *f* à chaque élément de *C*



Propriétés : la dimension de C est preservée

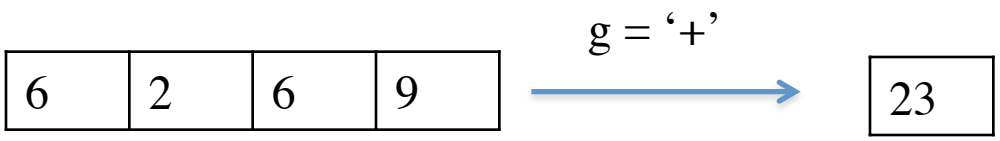
le type en entrée peut changer

Origine du modèle Map Reduce

- Programmation fonctionnelle : fonctions d'ordre supérieur appliquant une fonction aux objets d'une collection *C*

– *Reduce* (*g*: (T,T)=>T), *g* binaire

  - agréger les éléments de *C* deux à deux



Propriétés : réduit la dimension de *n* à 1

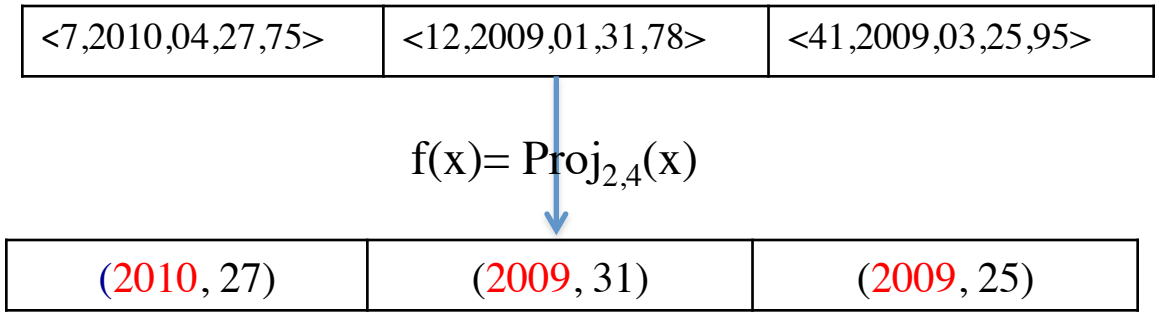
le type en sortie identique à celui en entrée

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, à ne pas confondre avec notion de clé)
- *Map (f:* T=>(*k*,U)), *f* unaire
  - produire une paire (clé, val) pour chaque val de *C*

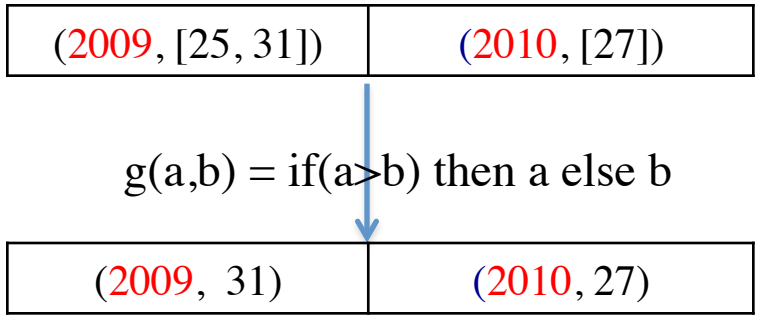


  - Regrouper les paires ayant la même clé pour obtenir (clé, [list-val])

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

Map Reduce pour le big data

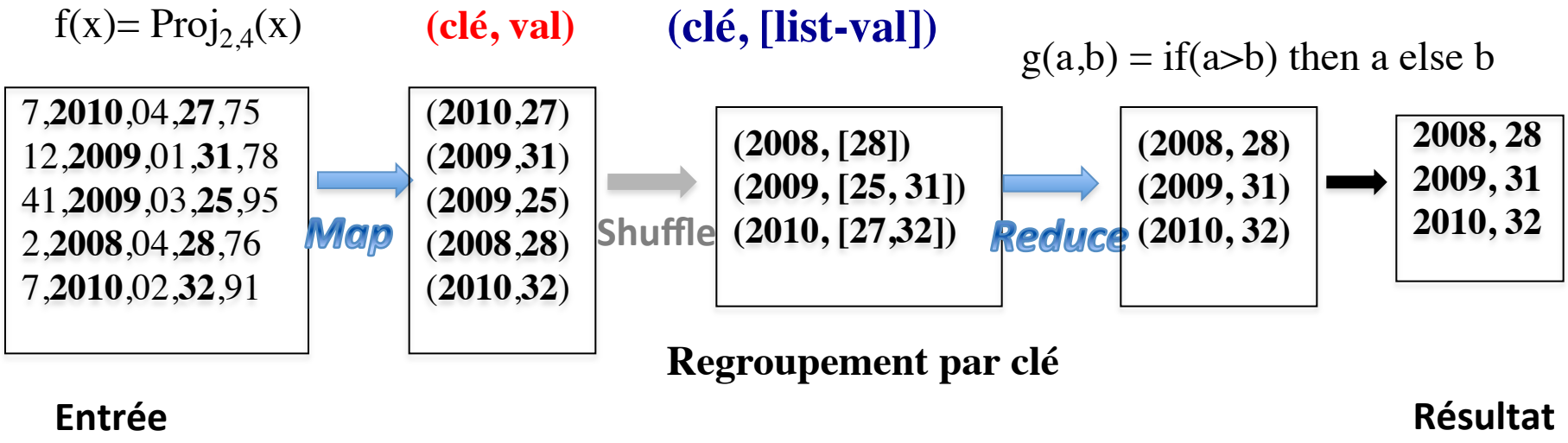
- *Reduce* (*g*: (T,T)=>T), *g* binaire
  - pour chaque (clé, [list-val]) produit (clé, val) où val = g([list-val])



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

Map Reduce : Exemple

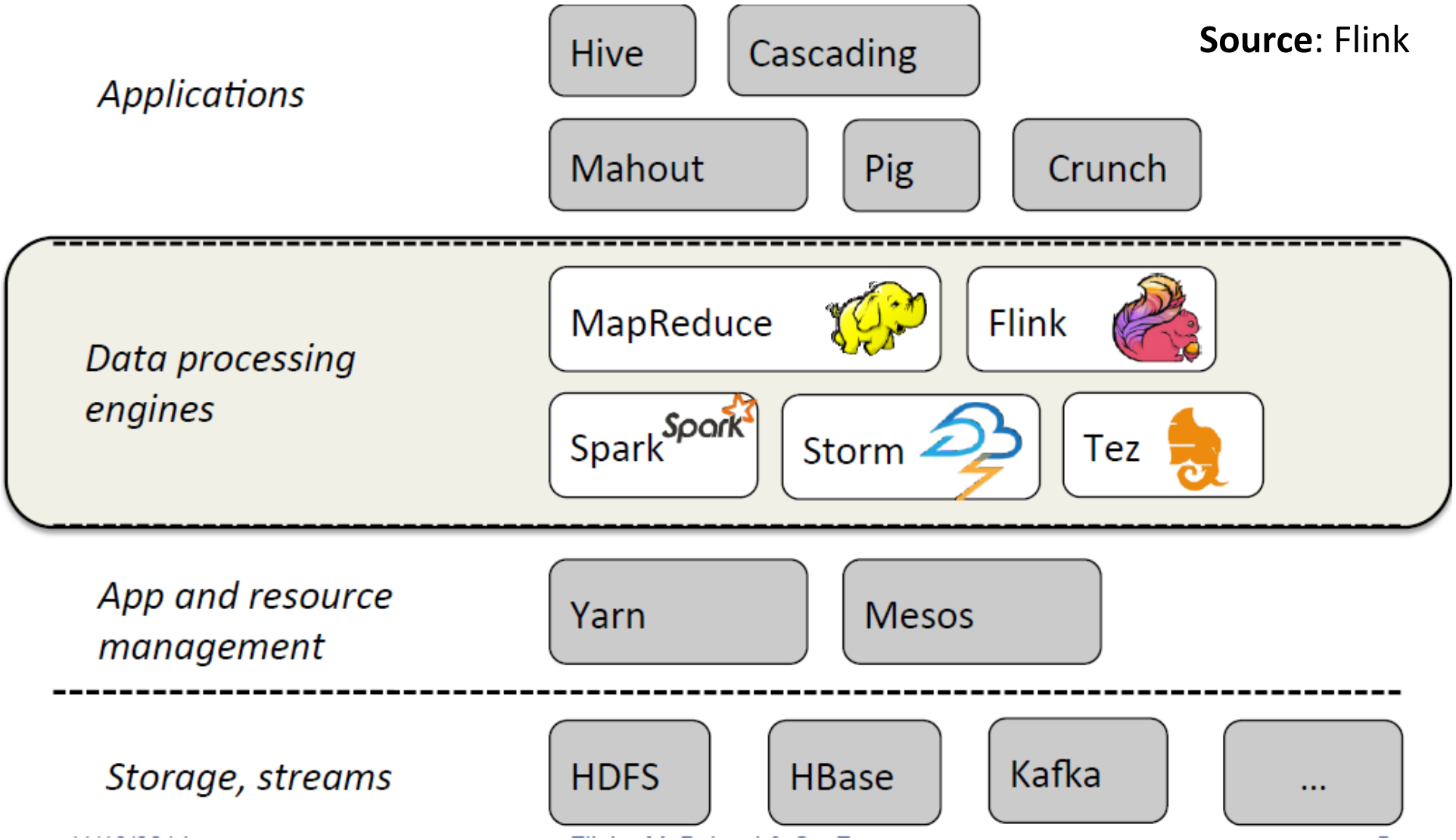
- Entrée : n-uplets (station, annee, mois, temp, dept)
- Résultat : select annee, Max(temp) group by annee



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



Source: Flink

Hadoop Map Reduce

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

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
- Quid des traitements itératifs - interaction avec l'utilisateur

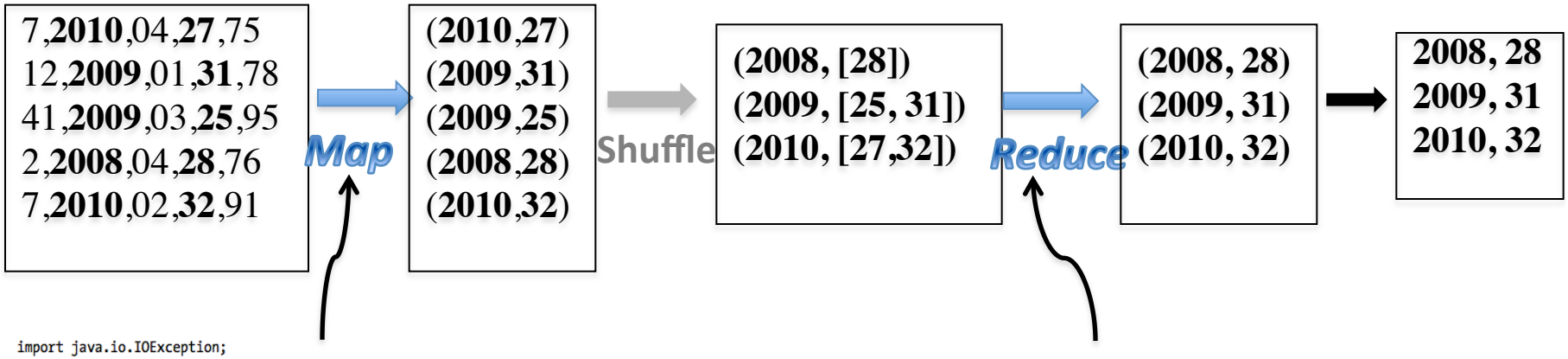
Spark

- Résoudre les limitations de Map Reduce
  - Matérialisation vs maintien en mémoire centrale
  - Traitement en lot vs interactivité (Read Execute Print Loop)
- *Resilient Distributed Dataset (RDD)*
  - collection logique de données distribuées sur plusieurs nœuds
  - 1 RDD référence données sur disque (HDFS), donnée en mémoire centrale ou autre RDD dont elle dépendent
  - traitement gros granule (pas de modification partielle)
  - tolérance aux pannes par réexécution d'une chaine de traitement
- Réutilisation de certains mécansimes 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
  - Un *framework* assez complet, en continuelle évolution
  - Système interactif et de production à la fois
- Choix du langage hôte : Scala
  - langage natif de Spark, élégant (car fonctionnel), en pleine expansion, concis

Programmer en Map Reduce

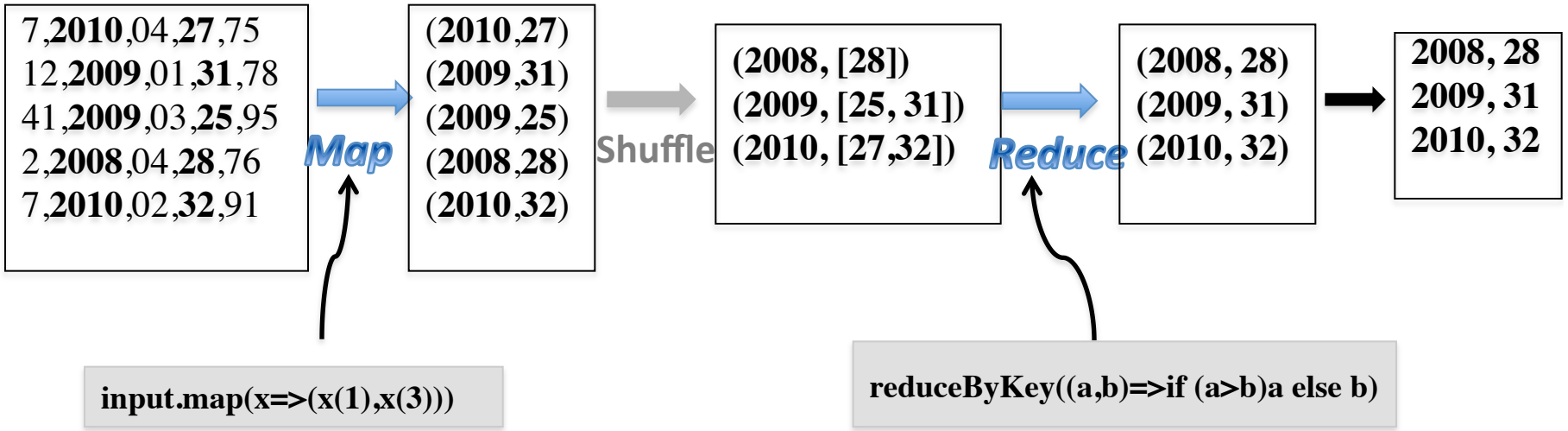



```java
import java.io.IOException;

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

```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 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

CORRECTION EXERCICE SCALA

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"))
3data.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

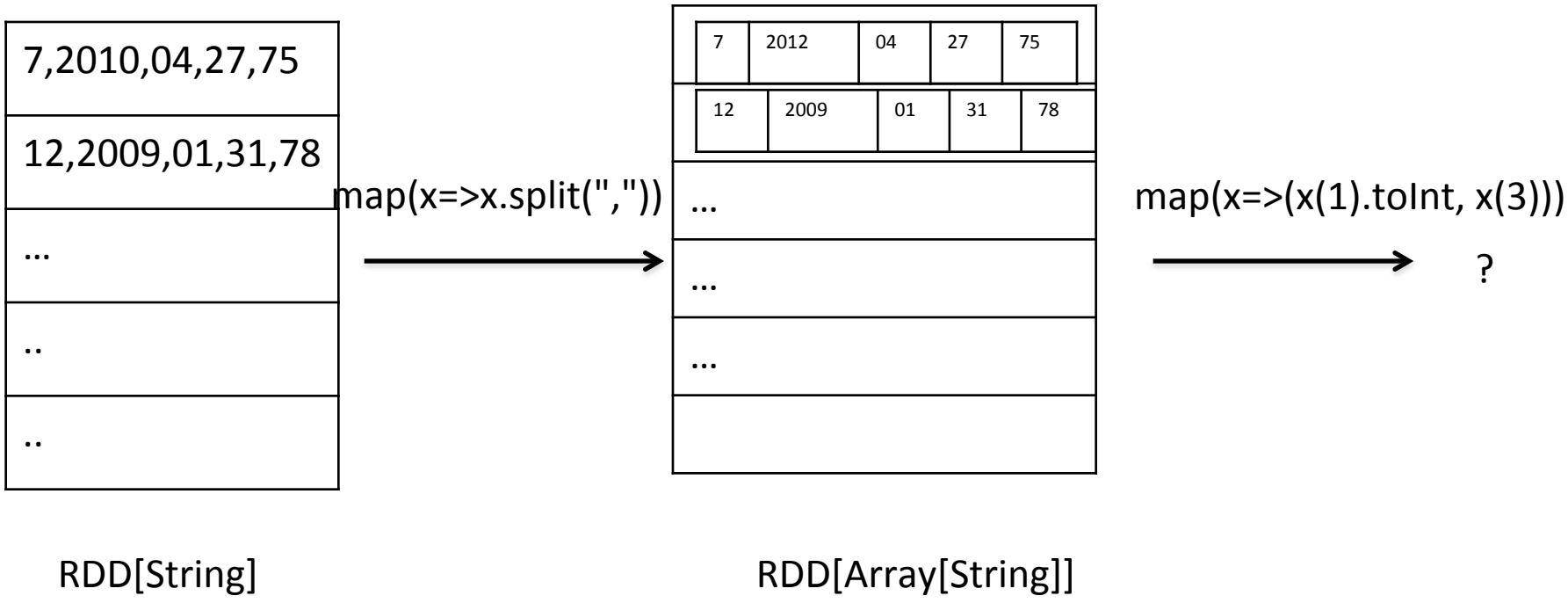
Ex. *count*, *save*, *collect*

Opérateurs RDD

Transformations	*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

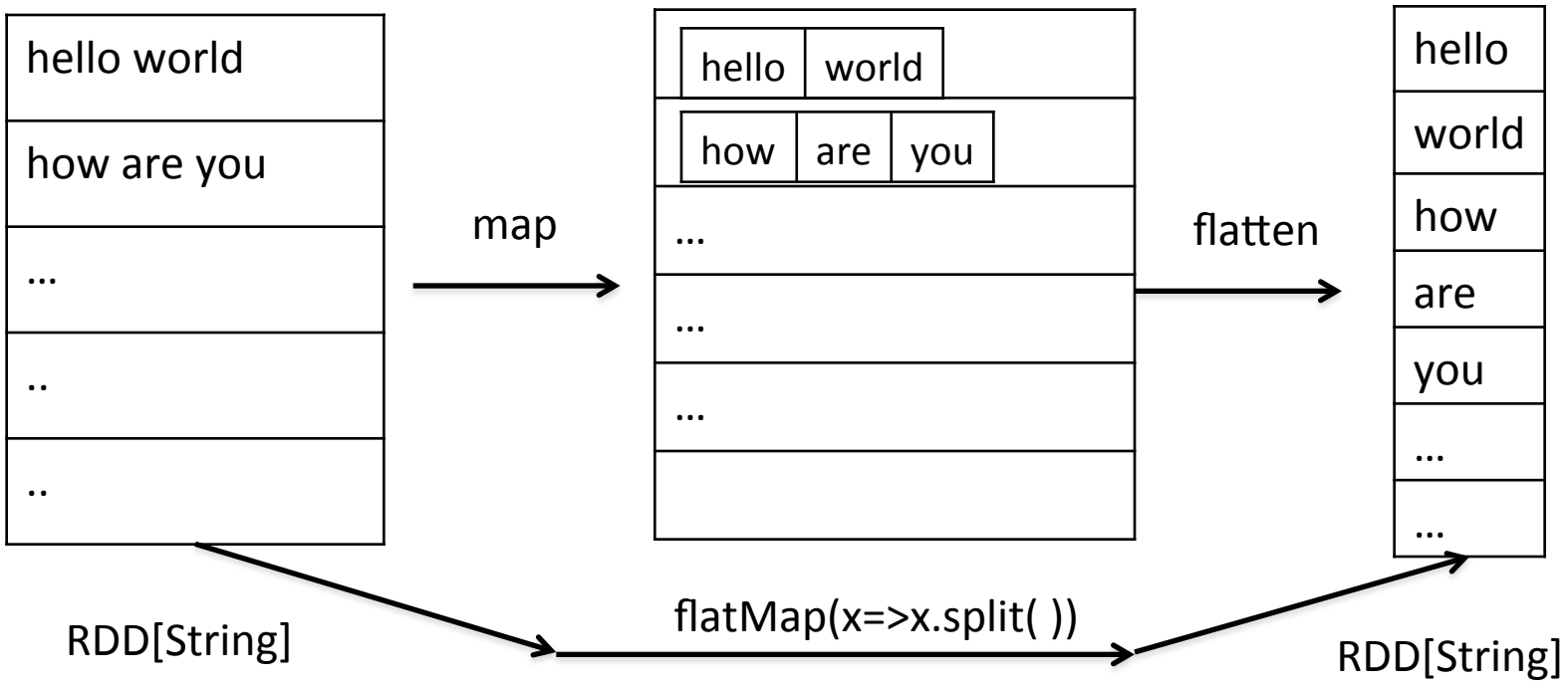
Opérateurs RDD

Map (*f:*T⇒U) : RDD[T] => RDD[U]



Opérateurs RDD

flatMap (*f:*T⇒Seq[U]) : RDD[T] => RDD[U]



Opérateurs RDD

reduceByKey(*f:* (V, V)⇒V) : RDD[(K,V)] => RDD[(K,V)]

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

RDD[(Int, Double)]

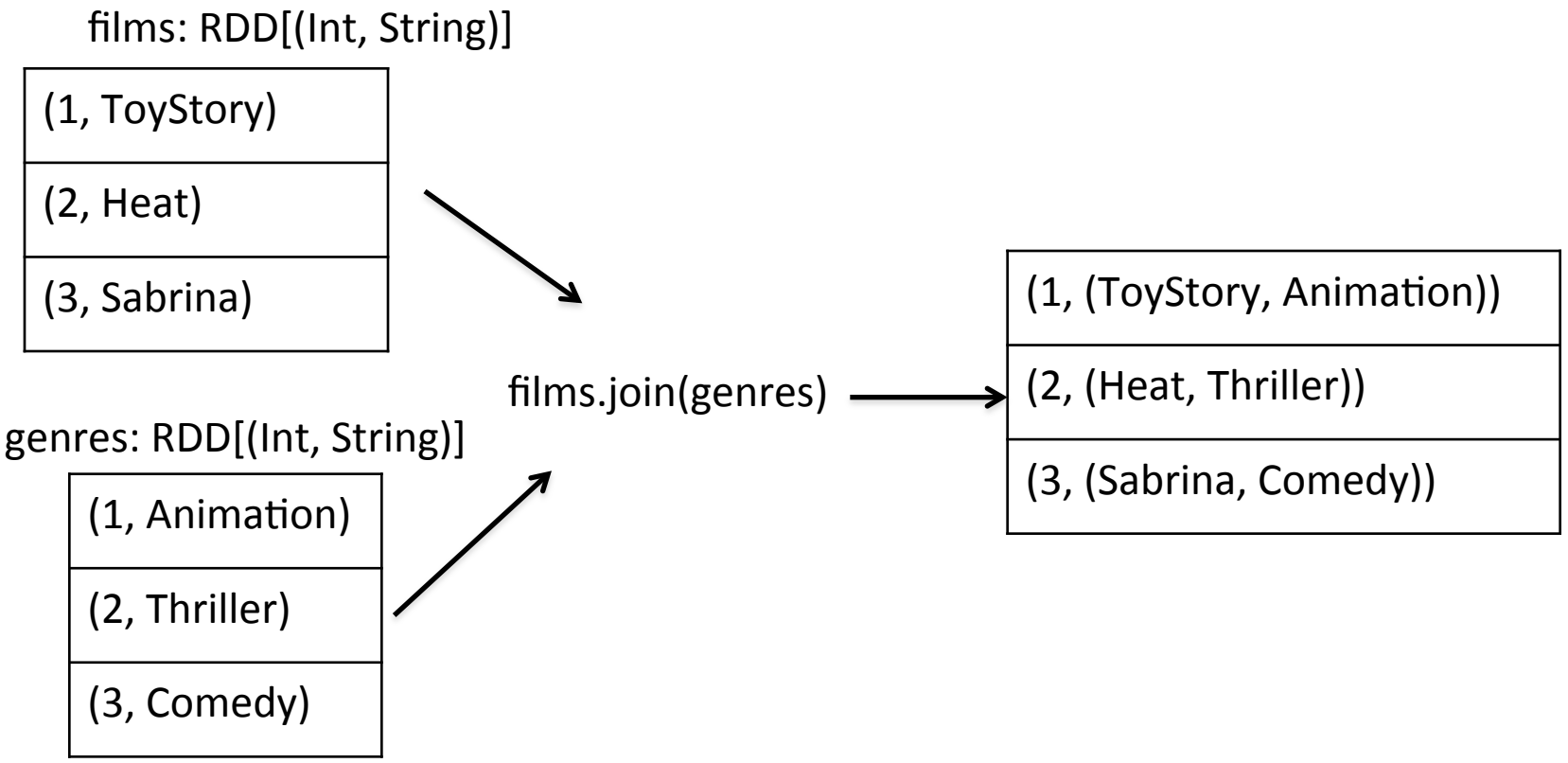
reduceByKey((a,b)=>if (a>b)a else b)

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

RDD[(Int, Double)]

Opérateurs RDD

join(): (RDD[(K,V)], RDD[(K,W)]) => RDD[(K,(V,W))]

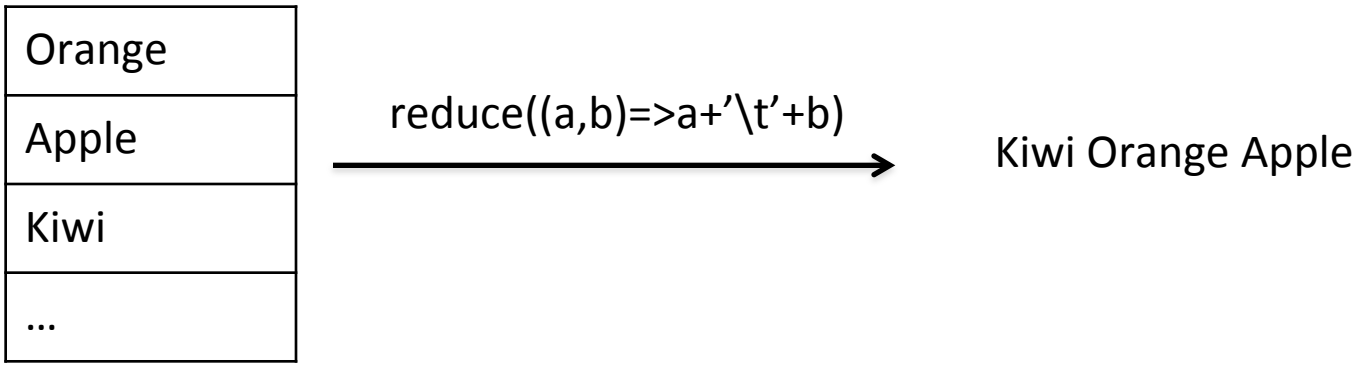


Opérateurs RDD

reduce(f : (T,T)=>T): RDD[(T,T)] => T

- Origine : prog. fonctionnelle
- Réduction de la dimension en utilisant une *User Defined Function (UDF)*
- Traitement distribué, aucun ordre prescris

  →dans Spark *f* doit être commutative et associative!



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 : Dataset
  - Utiliser les schéma pour exprimer requêtes (comme en SQL)
  - meilleure organisation des données : meilleures performances

Spark Dataset

- Description
  - Collection d'objets ayant un schéma homogène
  - Manipulées avec opérations *fonctionnelles* (ex. map) ou *relationnelles* (ex. join)
  - Distinction entre actions et transformations comme RDD
  - Possibilité d'optimisation logique (comme pour SQL)
- Création
  - Définir une *case class* avec schéma cible puis parcourir et encoder les données en utilisant cette classe
  - Possibilité de revenir vers RDD

RDD vs Dataset

```
val films =
sc.textFile().map(_.split(‘’,’’)).map(…)
films.filter(x=>x._1==2)
```

1	Toy Story (1995)	Animation|Children
2	Jumanji (1995)	Adventure|Children
	..	

```
case Class Film(MovieId:Str, Title:Str, Genres:Str)
val films = sc.textFile().map(_.split(‘’,’’)).map(…)
films.filter(x=>x.MovieID==2)
```

Film(1, Toy Story (1995), Animation|Children)
Film(2,Jumanji (1995),Adventure|Children)
…

RDD

```
val films = spark.read.format("csv"). …
films.filter(‘’MovieID=2’’)
```

MovieID	Title	Genres
1	Toy Story (1995)	Animation|Children
2	Jumanji (1995)	Adventure|Children

Dataset

Création d'un Dataset

MovieID,Title,Genres
1,Toy Story (1995),Animation|Children…
2,Jumanji (1995),Adventure|Children….
3,Grumpier Old Men (1995),Comedy….
…

movies.csv

```
scala> case class Movie(MovieID:String,Title:String,Genres:String)

scala> val films = spark.read.format("csv").
            option("header",true).
            load(path+ "movies.csv").as[Movie]
films: org.apache.spark.sql.Dataset[Movie] = [MovieID: string, Title: string ... 1 more field]
```

Quelques opérations Dataset

- Actions
  - count
  - describe
  - reduce
  - show
- Fonctions
  - rdd
  - dtypes
  - printSchema
- Transformations
  - distinct
  - except
  - filter
  - flatMap
  - groupByKey
  - map
  - orderBy
  - agg
  - groupBy
  - join
  - select

Dataset par l'exmple

- Actions et fonctions de base

scala> films.show

```
+-------+--------------------+--------------------+
|MovieID|               Title|              Genres|
+-------+--------------------+--------------------+
|      1|    Toy Story (1995)|Animation|Childre...|
|      2|      Jumanji (1995)|Adventure|Childre...|
|      3|Grumpier Old Men ...|      Comedy|Romance|
|      4|Waiting to Exhale...|        Comedy|Drama|
|      5|Father of the Bri...|              Comedy|
```

scala> films.printSchema

root

|-- MovieID: string (nullable = true)

|-- Title: string (nullable = true)

|-- Genres: string (nullable = true)

Dataset par l'exemple

- Transformations

```
scala> films.map(x=>x.Genres.split("\\|")).show
```

```
+--------------------+
|               value|
+--------------------+
|[Animation, Child...|
|[Adventure, Child...|
|   [Comedy, Romance]|
|     [Comedy, Drama]|
```

```
scala> films.orderBy("Title").show
```

```
+-------+--------------------+--------------------+
|MovieID|               Title|              Genres|
+-------+--------------------+--------------------+
|      5|Father of the Bri...|              Comedy|
|     10|    GoldenEye (1995)|Action|Adventure|...|
|      3|Grumpier Old Men ...|      Comedy|Romance|
|      6|         Heat (1995)|Action|Crime|Thri...|
|      2|      Jumanji (1995)|Adventure|Childre...|
|      7|      Sabrina (1995)|      Comedy|Romance|
|      9| Sudden Death (1995)|              Action|
```

Dataset par l'exmple

- Agrégation simple – agrégation avec groupement

```
scala>  case class Note(UserID: Int, MovieID: Int, Rating: Int, Timestamp: Int)

scala> val notes = sc.textFile(path+"sub-ratings.csv").map(x=>x.split(",")).
                    map(x=>Note(x(0).toInt,x(1).toInt, x(2).toInt, x(3).toInt)).toDS

scala> notes.agg(min("Rating"), max("Rating"), avg("Rating"))

scala> notes.describe("Rating").show //montre count, stddev en plus

scala> notes.groupBy("MovieID").agg(count("*")).sort("count(1)").show
```

Dataset par l'exmple

Sélection

```
scala> films.filter("MovieID=1")

scala> films.filter("MovieID=1 or Title='Toy Story (1995)'")
```

Projection

```
scala> films.select("MovieID","Genres")

scala> films("MovieID") // une colonne à la fois
```

Equi-jointure

```
scala> films.join(notes,"MovieID")
```