

Map Reduce sur Spark

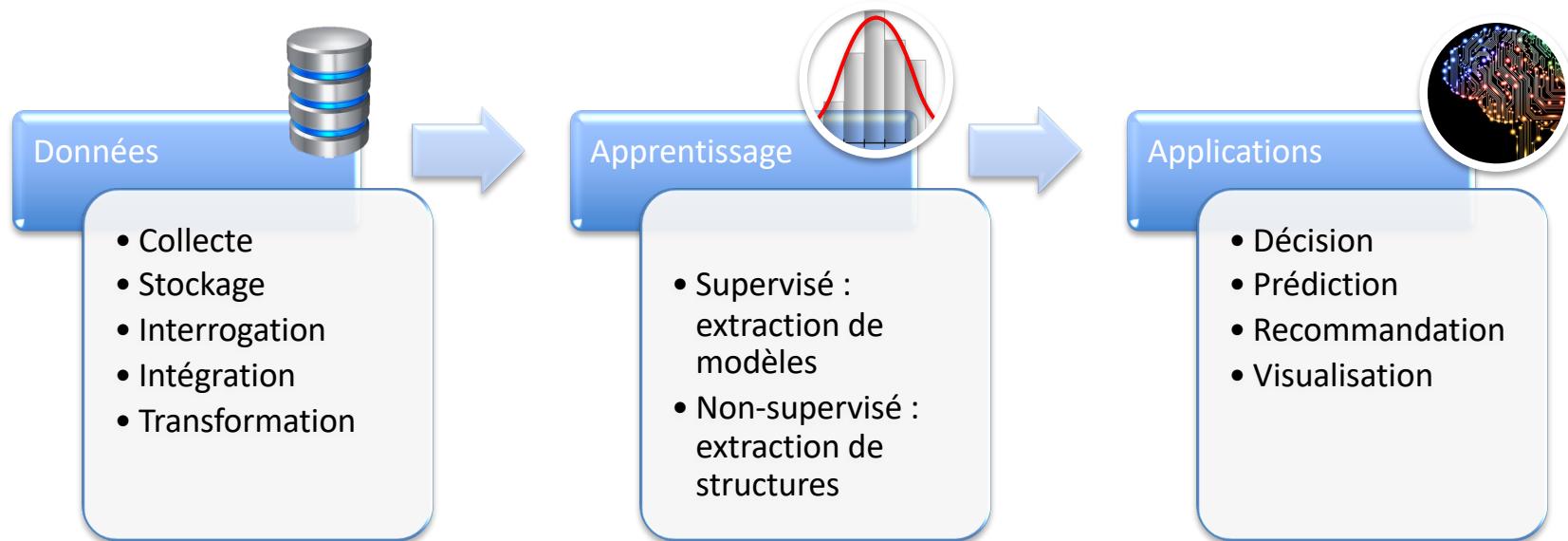
Master DAC – Bases de Données Large Echelle

Mohamed-Amine Baazizi

mohamed-amine.baazizi@lip6.fr

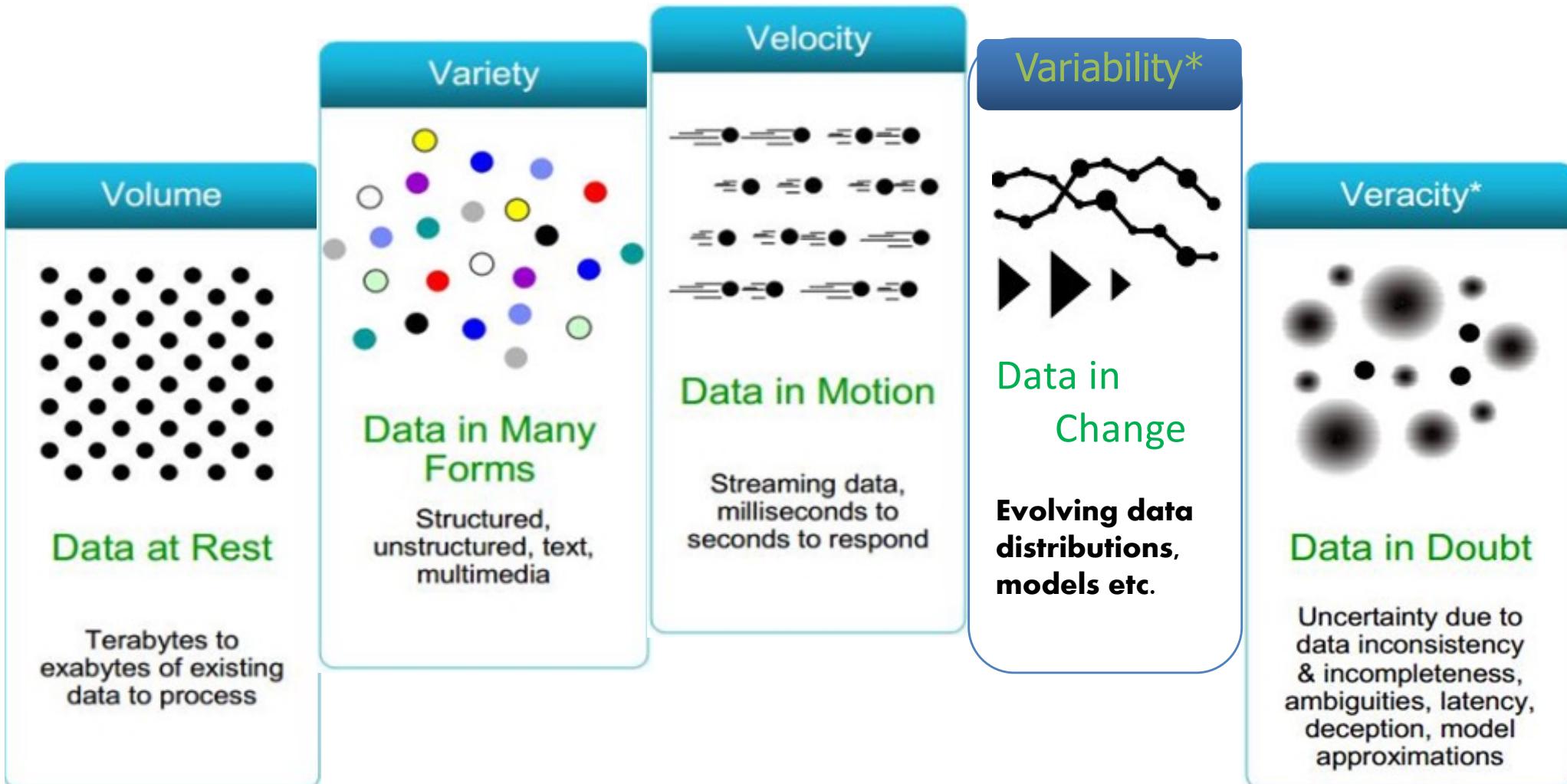
2021-2022

Big Data Pipeline



Data quality is a crucial issue: garbage in = garbage out

Caractéristiques du big data



Relever le défi du volume

- 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...)

Plateformes

- Traitement/stockage distribué
 - Hadoop (Google), Spark (MPLab, Apache), Flink (TU Berlin, Apache)
 - HBase, Kafka, HDFS, Amazon S3, ...
- Gestion ressources/déploiement
 - Yarn, Mesos, Docker, Kubernetes
- Concurrents
 - Dask (natif Python, traitement en parallèle single node ou cluster)

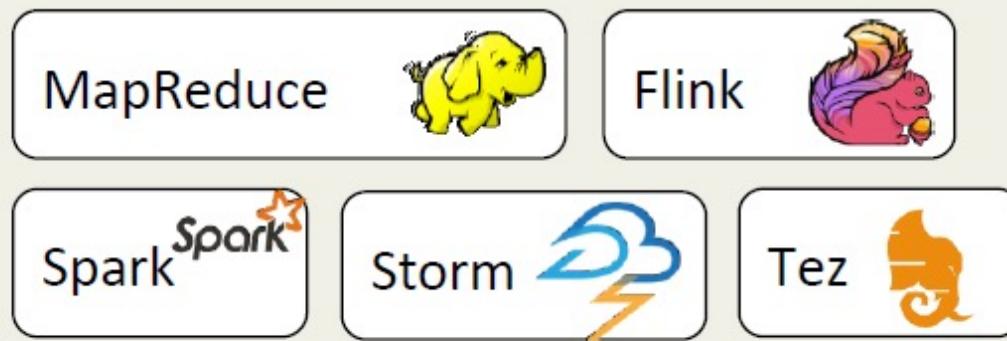
Pile logicielle

Applications



Source: Flink

Data processing engines



App and resource management



Storage, streams

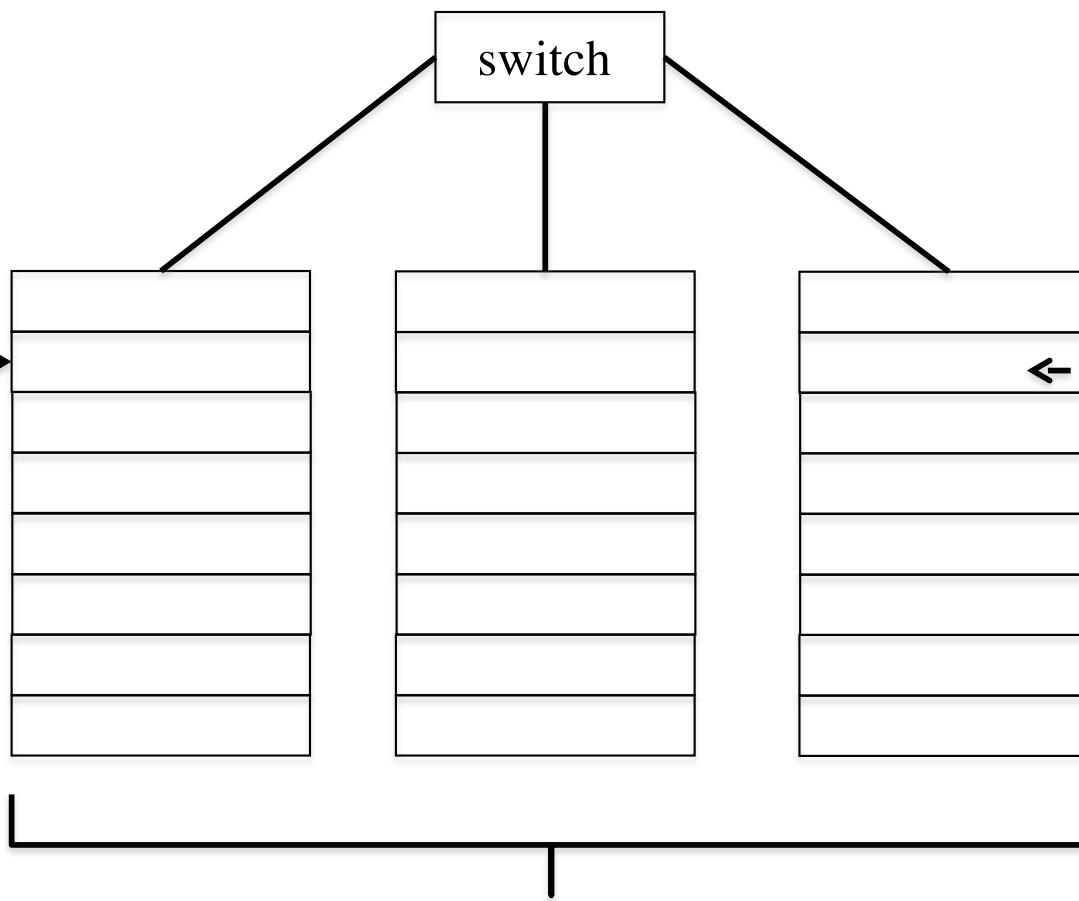


Architecture type d'un cluster

Lame de calcul
= [8,64] unités



Unité de calcul
=CPU+RAM+DDur



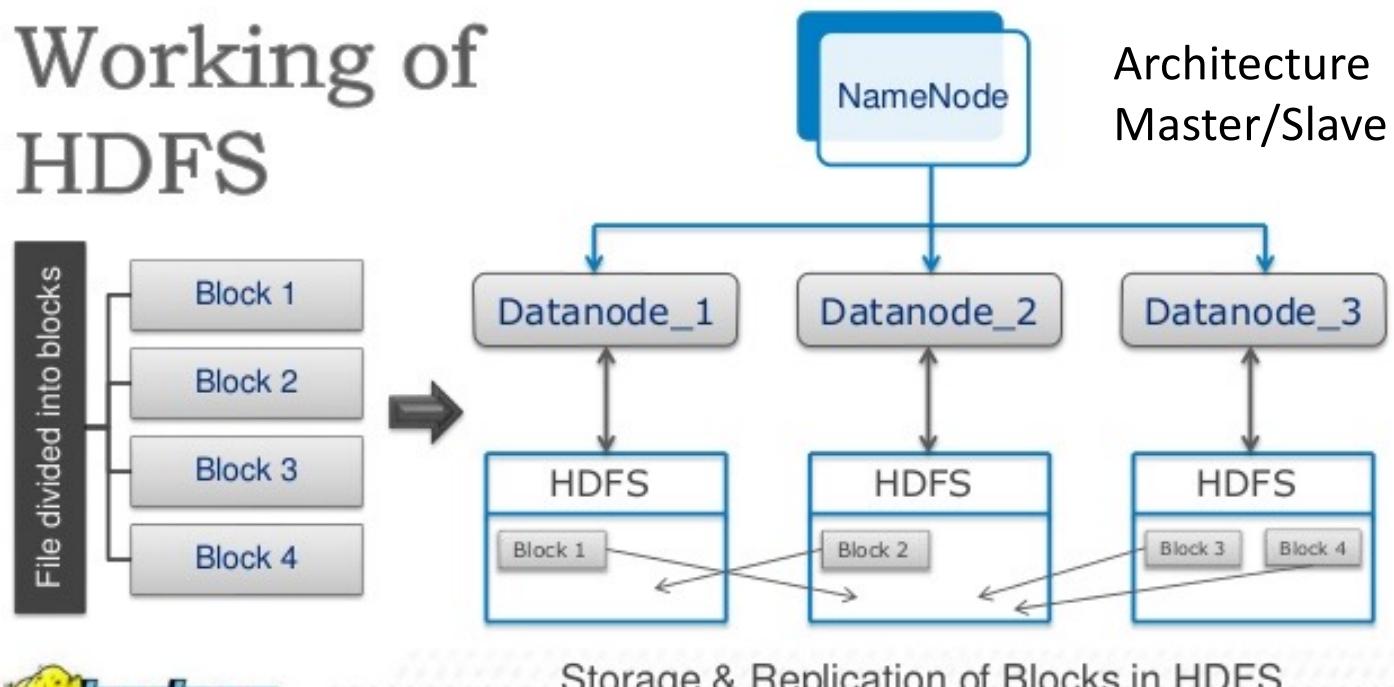
Cluster

Hadoop Map Reduce

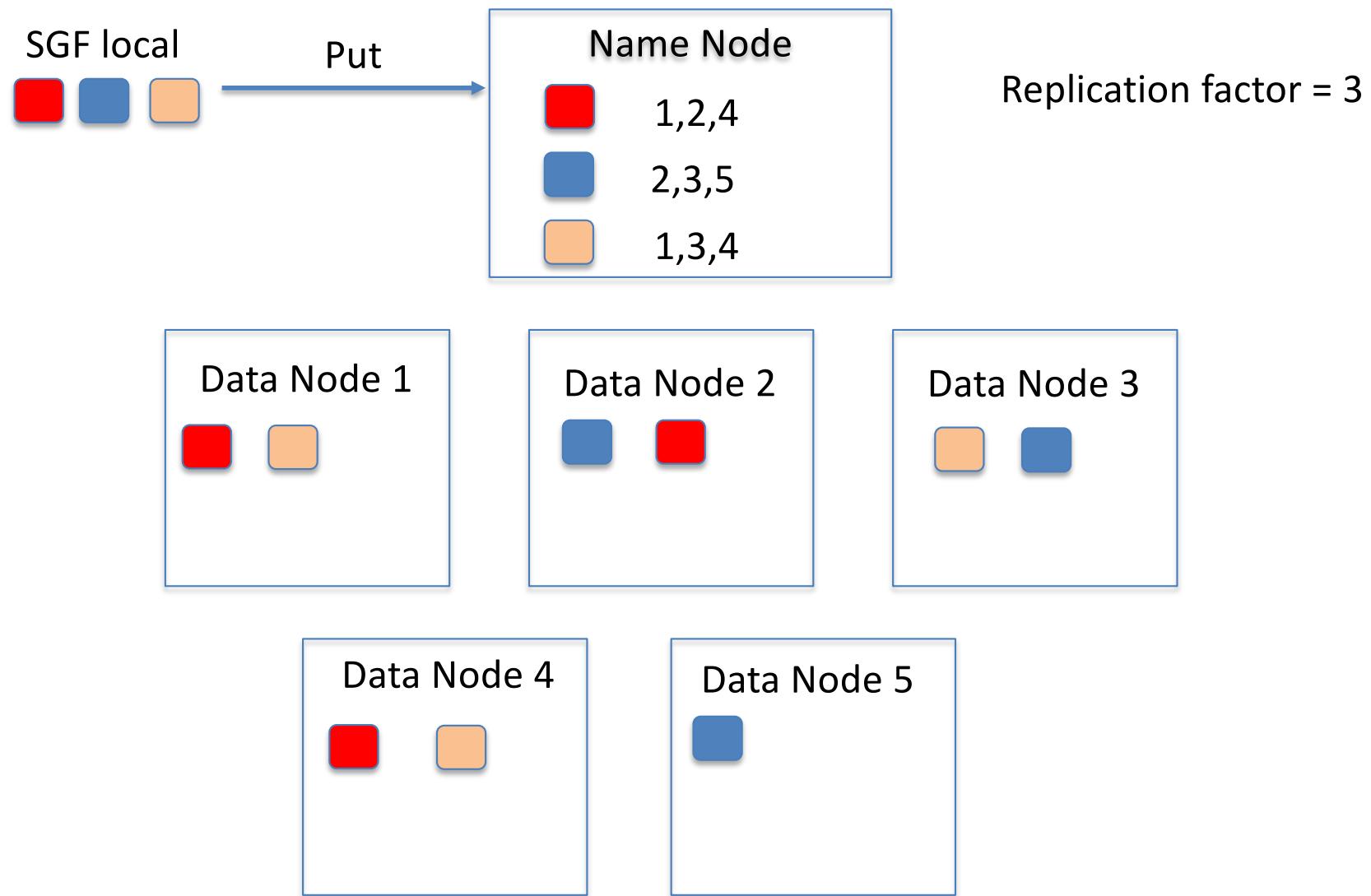
- Introduit par Google en 2004
- Répondre à trois principales exigences
 - Utiliser cluster de machines standards
 - extensibles à volonté (architecture RAIN)
 - facilité d'administration, tolérance aux pannes
- Ecrit en Java. Utilisation autre langages possible
- Plusieurs extensions
 - Pig et Hive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

HDFS : architecture

Working of HDFS



HDFS : replication

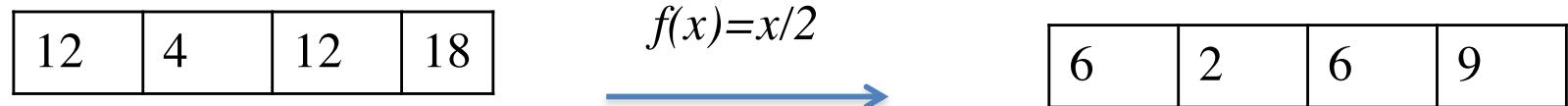


Origine du modèle Map Reduce

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

Illustration

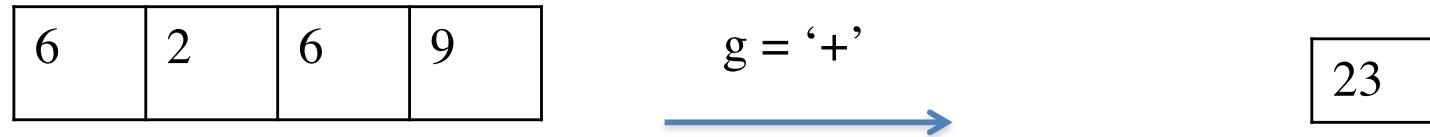
- *Map* ($f: T \Rightarrow U$), f unaire : appliquer f à chaque élément de C



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



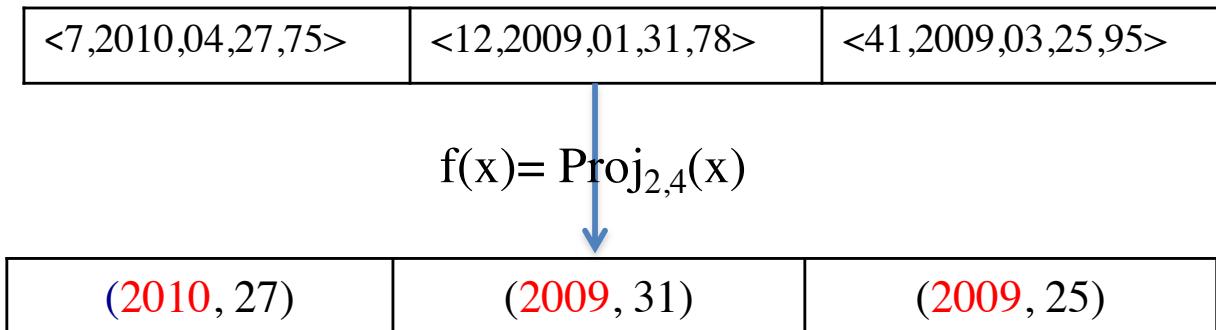
réduit la dimension de n à 1 le type en sortie identique à celui en entrée

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



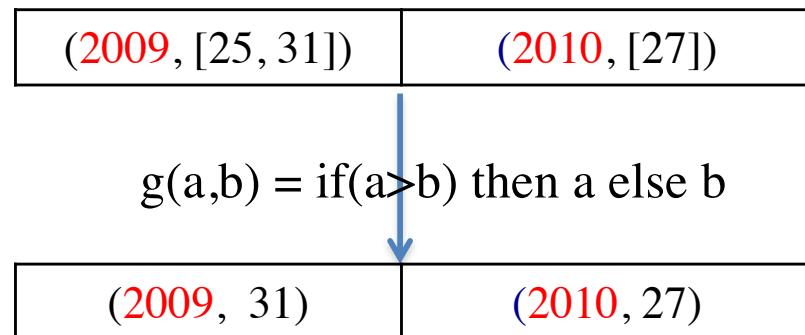
- Regrouper les paires ayant la même clé pour obtenir (clé, [list-val])

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

Map Reduce pour le big data

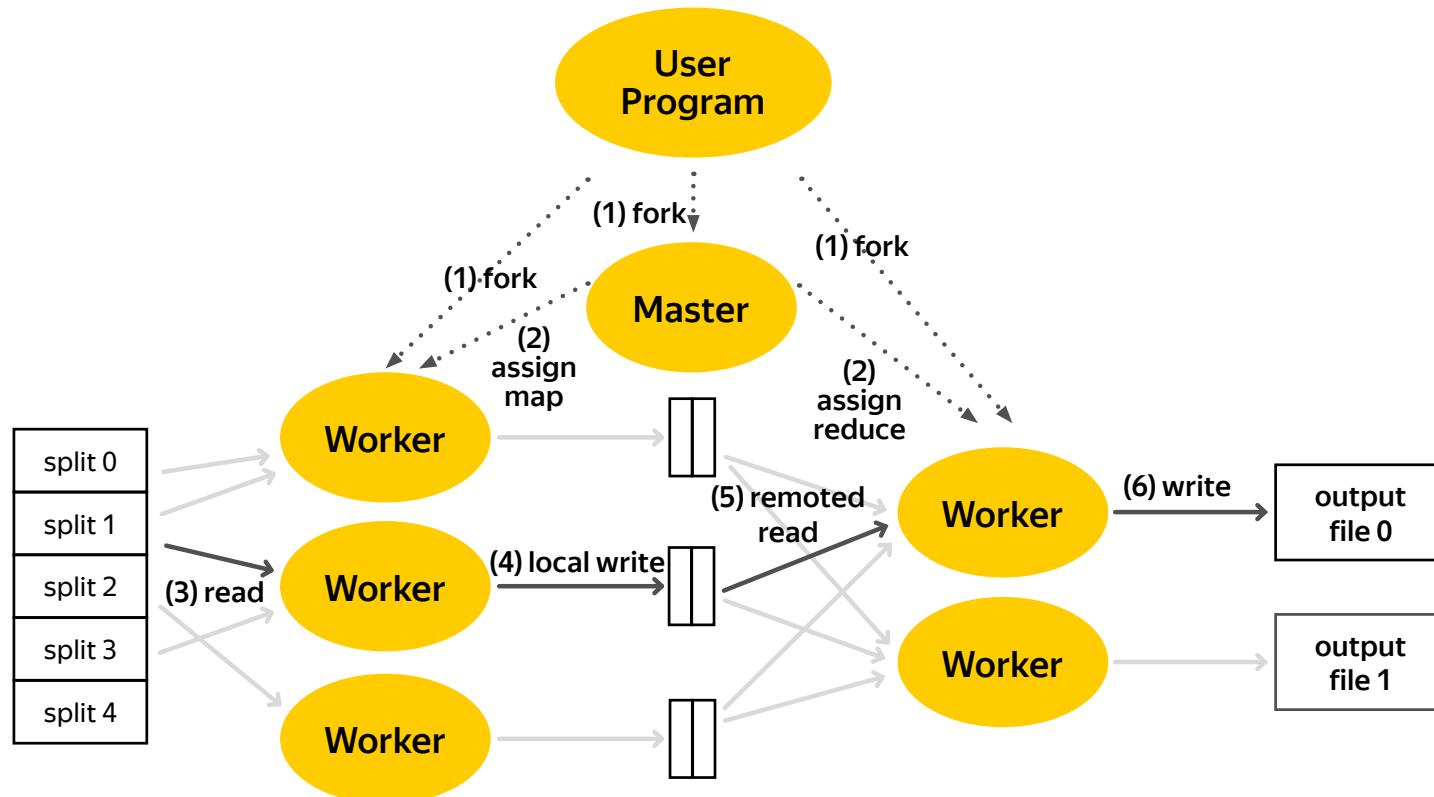
– *Reduce ($g: (T,T) \Rightarrow T$), g binaire*

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



- **Important** : dans certains systèmes, g doit être **associatif** car ordre de traitement des éléments de C non prescrit

Exécution Map-Reduce



Input
files

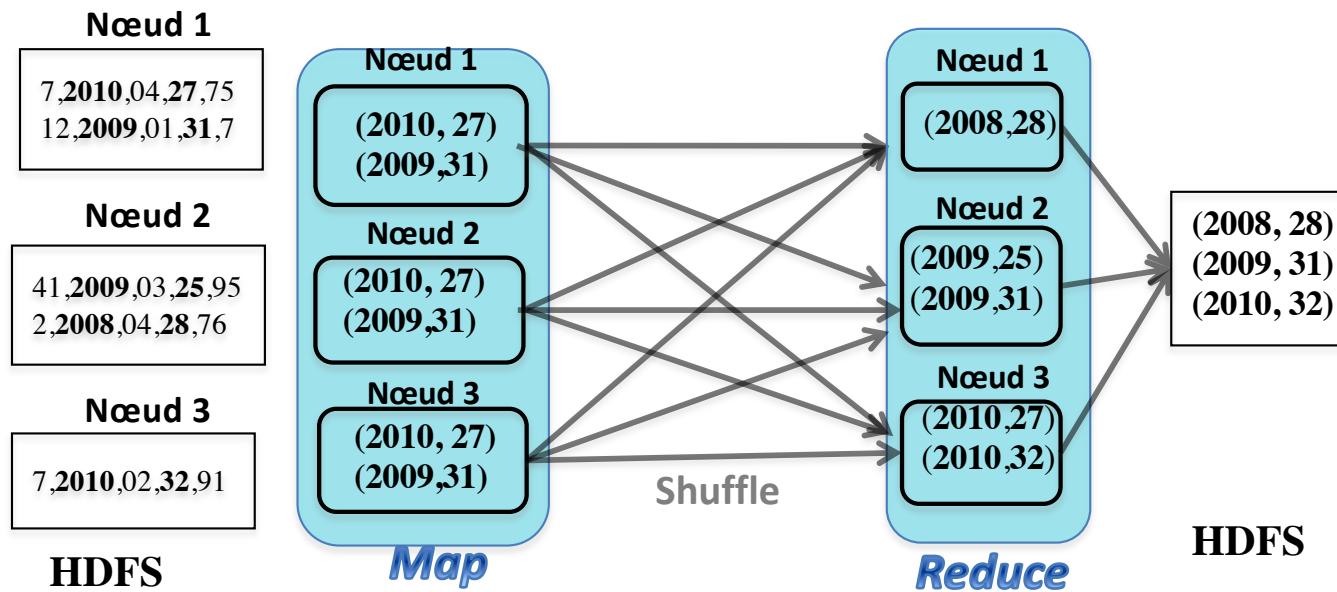
Map
phase

Intermediate files
(on local disks)

Reduce
phase

Output
files

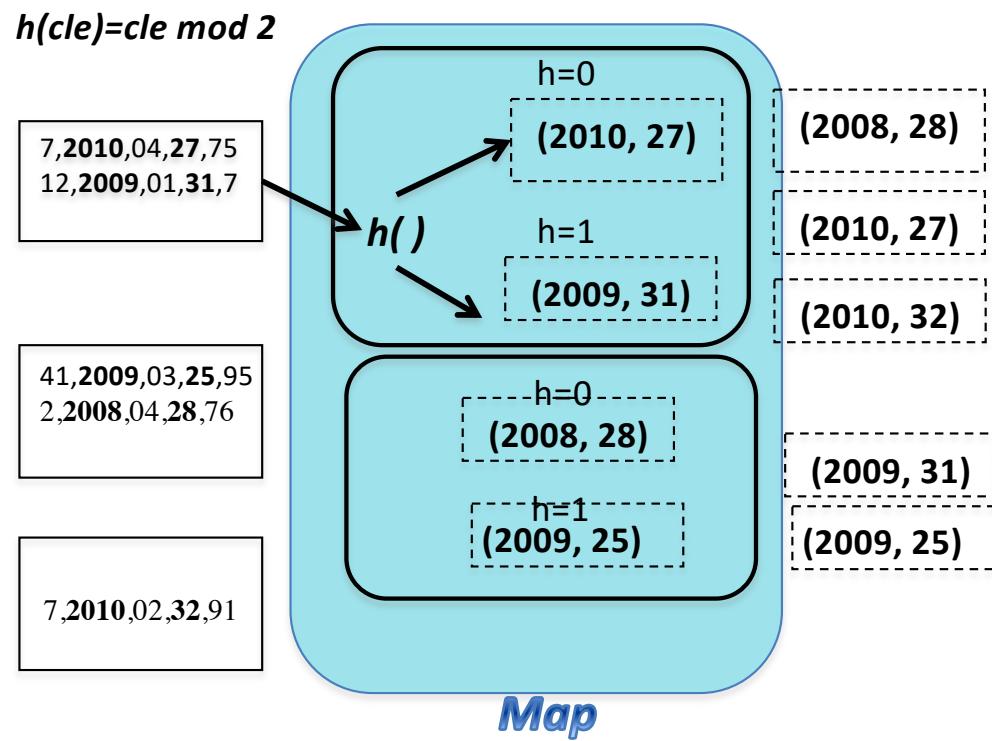
Exécution Hadoop Map Reduce



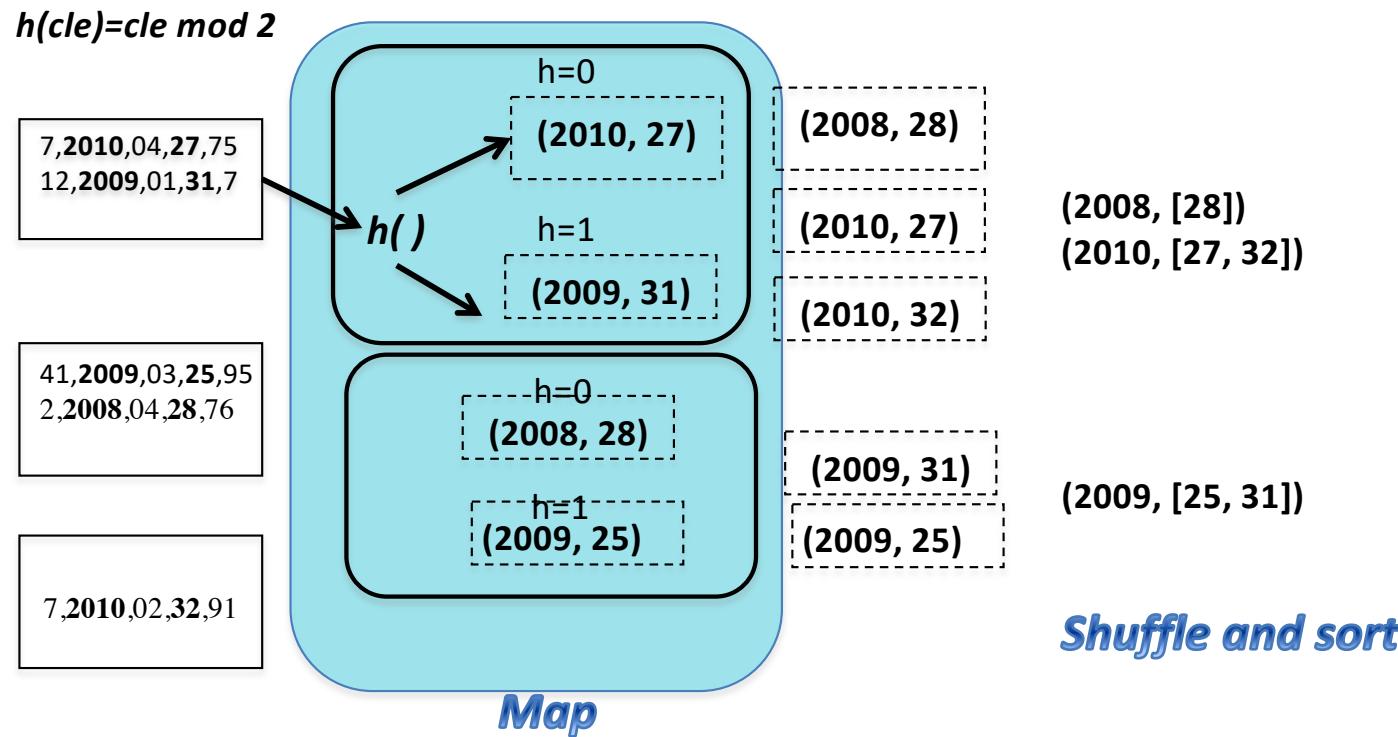
Entrée : n-uplets (station, **annee**, mois, **temp**, dept)

Résultat : select annee, Max(temp) group by annee

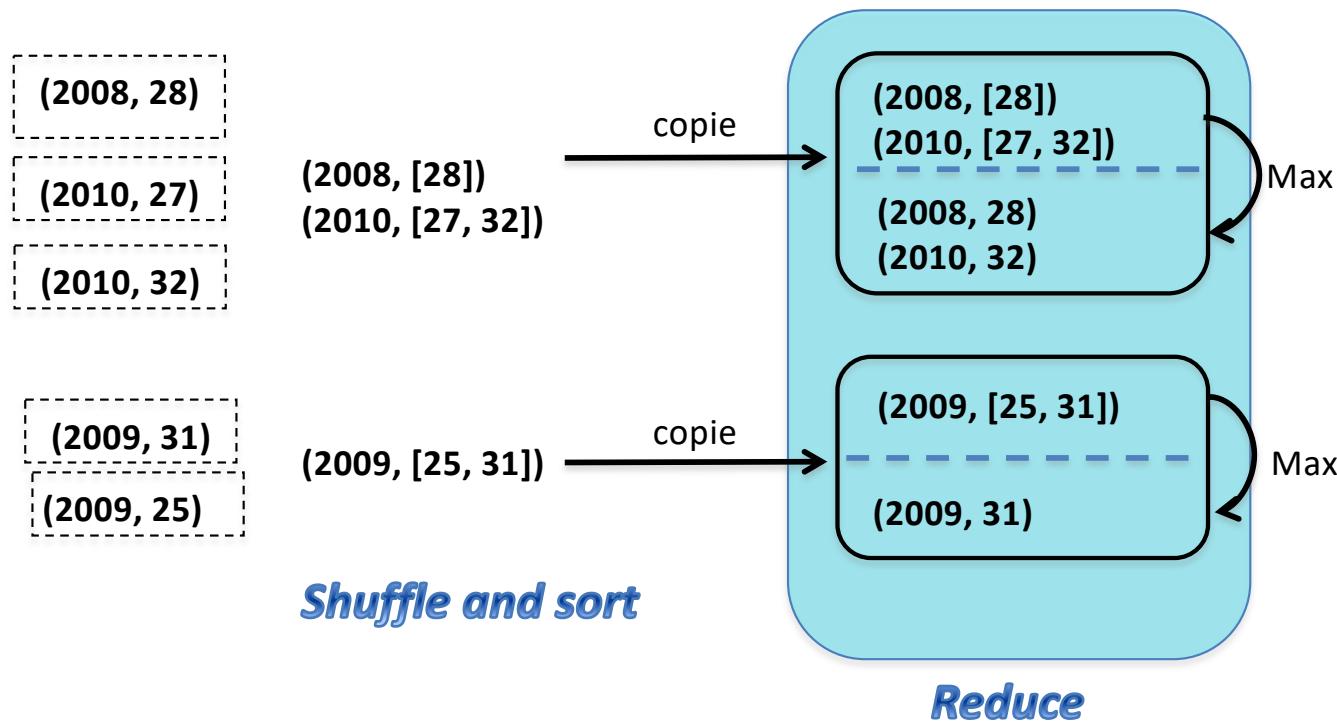
Phase Map



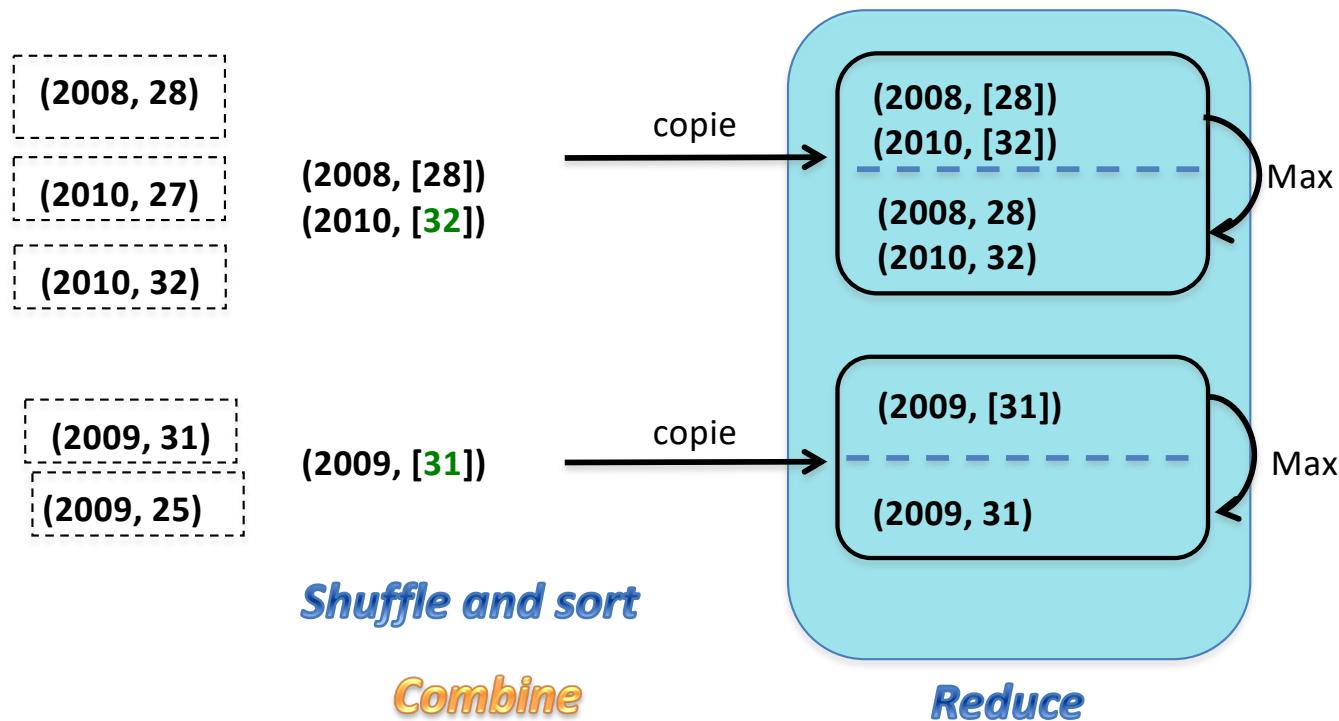
Phase Shuffle & Sort



Phase Reduce



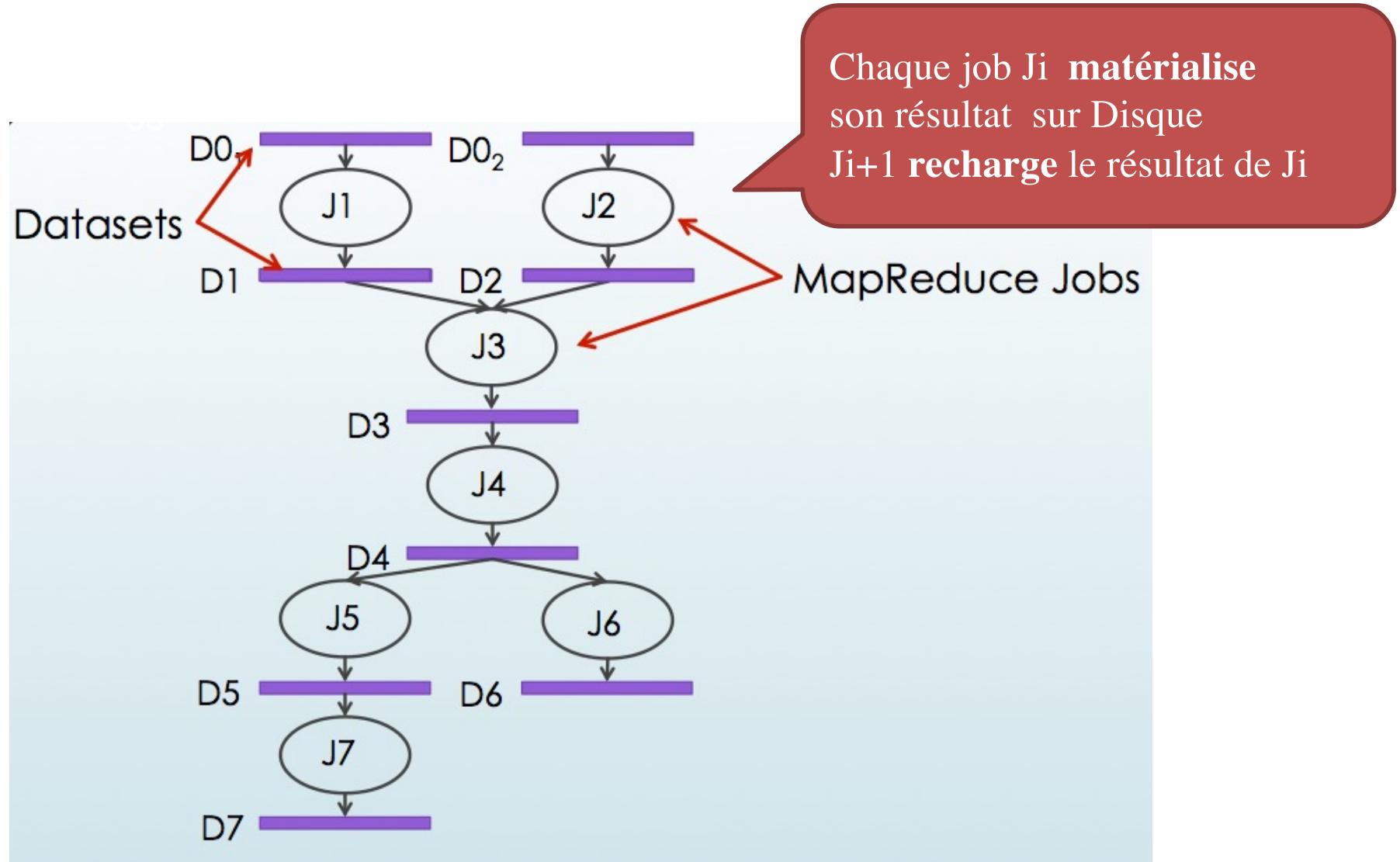
Combine



Limites de Hadoop Map Reduce

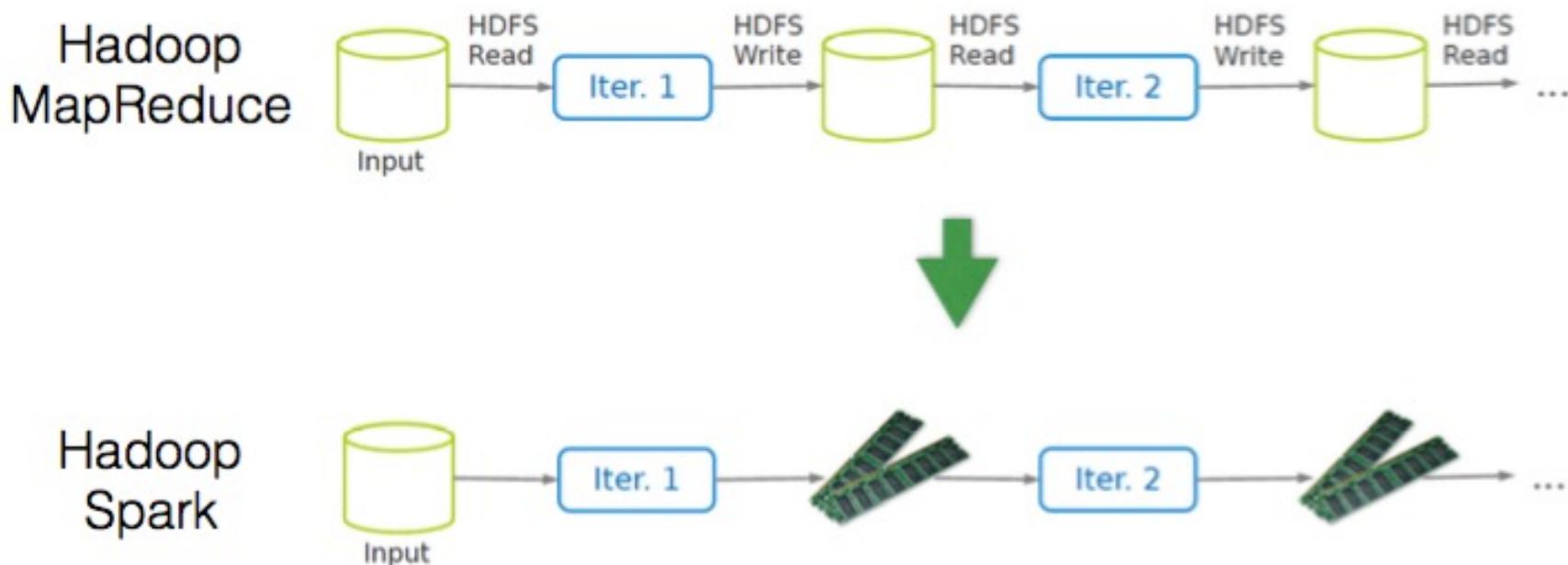
- Performances dégradées
 - Traitement complexe en plusieurs étapes
 - Raison : matérialisation résultat de chaque étape
 - avantages : reprise sur panne performante
 - inconvénients : accès fréquent au disque
 - Optimisation possible : pipelining et partage
 - Mais pas suffisante
- Inadapté aux traitements itératifs
 - ML et analyse graphes
- Pas d'interaction avec l'utilisateur

Hadoop Map Reduce

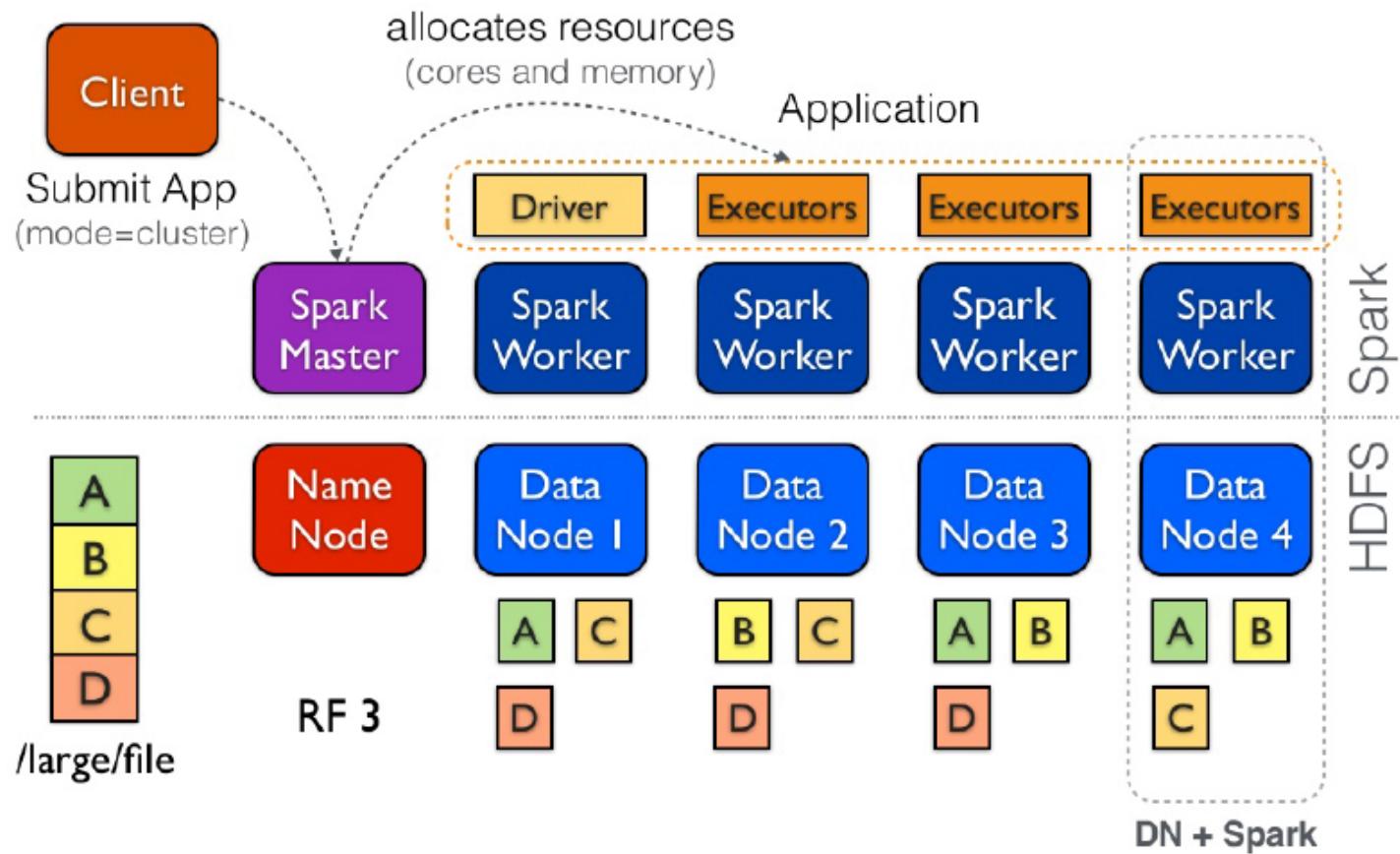


Hadoop Map Reduce vs Spark

- Résoudre les limitations de Map Reduce
 - *Persistance en mémoire centrale*
 - *Lazy evaluation*



Architecture type Spark



Caractéristiques Spark

- Framework assez complet, préparation et l'analyse des données massives
 - API de base, SQL pour CSV et JSON
 - ML : *feature extraction, model selection*
 - Graphs : modèle BSP
- Système interactif, utilisé en production
- Plusieurs langages hôtes
 - Scala (natif), Java, Python et R

RDD

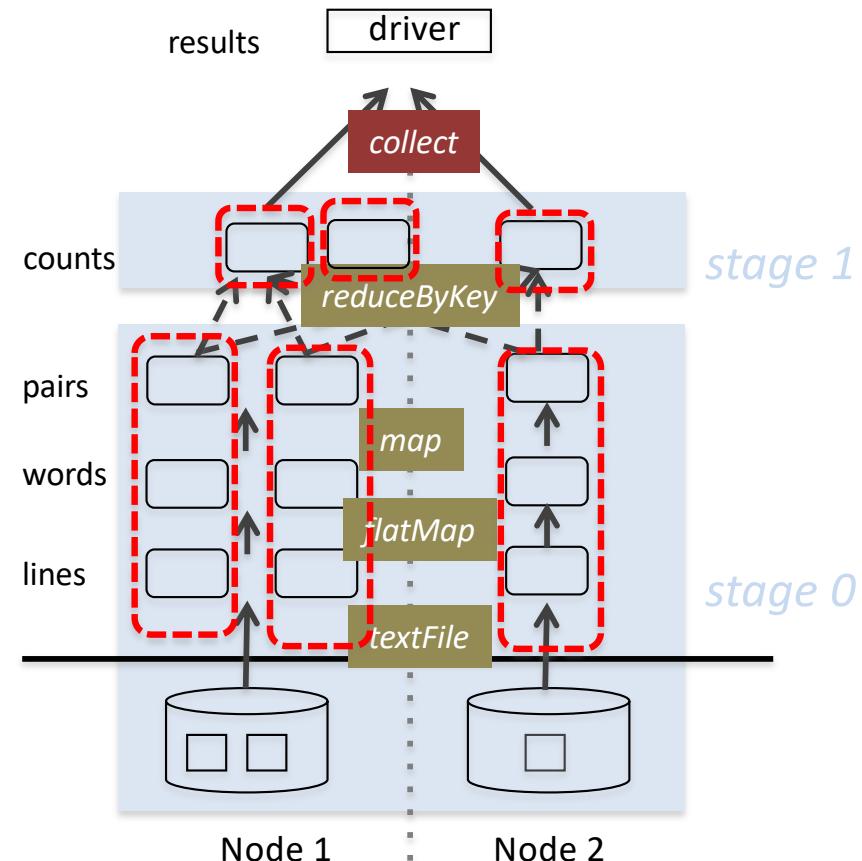
- Resilient Distributed Dataset (RDD)
 - collection logique de données distribuées sur plusieurs nœuds
 - traitement gros granule (pas de modification possible)
 - matérialisation optionnelle (lazy evaluation)
 - tolérance aux pannes par réexécution d'une chaîne de traitement
- Avantages:
 - pas de problème de cohérence (lecture seule)
 - gestion de pannes simplifiée : rejouer le *lineage*
 - gestion de charge simplifiée par copies
- Inconvénients:
 - inadapté aux transactions et aux flux de données

Wordcount en Spark RDD

```
lines = sc.textFile(filename)
words = lines.flatMap(lambda x:
x.split(" "))
pairs = words.map(lambda x: (x,1))
counts = pairs.reduceByKey(add)
results = counts.collect()
```

partition d'une RDD

action transformation



Deux types d'opérations

- Transformations :
 - opérations qui **définissent** une collection à partir d'une ou plusieurs autres collections
 - **unaires** : map, reduce, groupByKey, ...
 - **binaires** : union, subtract, join, ...
- Actions:
 - opérations qui **évaluent** des compositions de transformations (collections) et **génèrent des données dans le modèle du langage hôte** (Java, Scala, Python)

API RDD

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

map

Map (f:T⇒U) : RDD[T] => RDD[U]

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

`map(lambda x: x.split(","))`



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

RDD[String]

RDD[Array[String]]

Size (output) = size(input)

flatMap

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

hello world
how are you
...
..
..

Doit retourner une liste

flatMap(lambda x: x.split())

hello
world
how
are
you
...
...

RDD[String]

Size (output) >= size(input)
due to flattening

RDD[String]

filter

filter(f: T⇒Bool) : RDD[T] => RDD[T]

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

filter(lambda p : p[1]>30)

(2009,31)
(2010,32)

RDD[(Int, Double)]



RDD[(Int, Double)]

Size (output) <= Size (input)

reduceByKey

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

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

```
def f(a,b):  
    if a>b :  
        return a  
    else  
        return b
```

reduceByKey(lambda a,b : f(a,b))

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

RDD[(Int, Double)]



RDD[(Int, Double)]

f commutative et associative

Size (output) = |distinct keys|

groupByKey

groupByKey() : RDD[(K,V)] => RDD[(K,Seq[V])]

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

groupByKey()



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

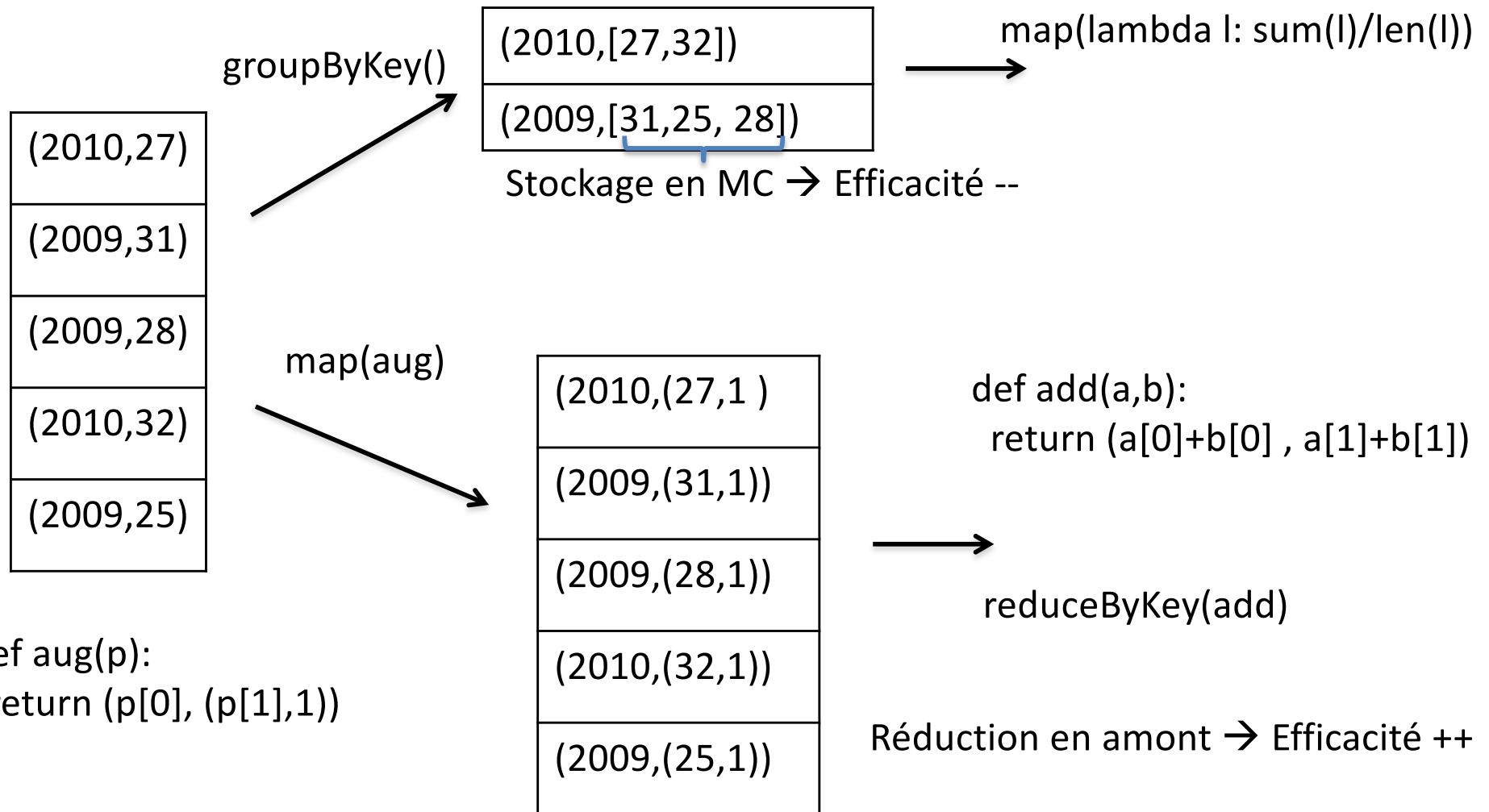
RDD[(Int, Seq[Double])]

RDD[(Int, Double)]

Possibilité d'appliquer
g algébrique (pas forcément associative)

Size (output) = |distinct keys|

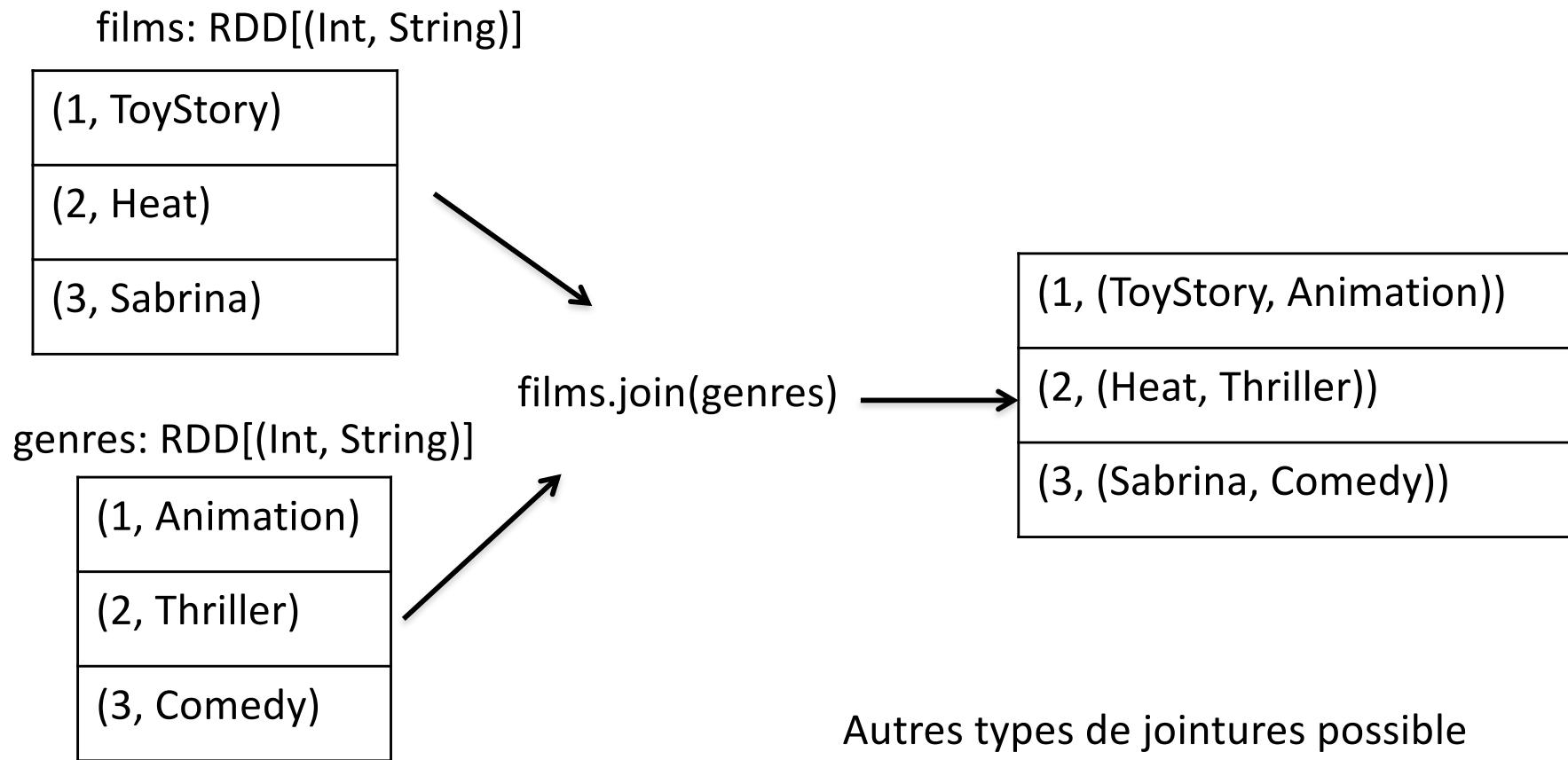
Calcul de la moyenne



```
def aug(p):  
    return (p[0], (p[1],1))
```

join

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



zipWithIndex

zipWithIndex() : rdd[T] ↗ rdd[(T,Long)]

hello
world
how
are
you
...
...

Augment une RDD avec un entier correspondant à l'indice de chaque élément

zipWithIndex()



(0, hello)
(1, world)
(2, how)
(3, are)
(4, you)
...
...

RDD[String]

RDD[(Int, String)]

Size (output) = Size (input)

RDD : Actions

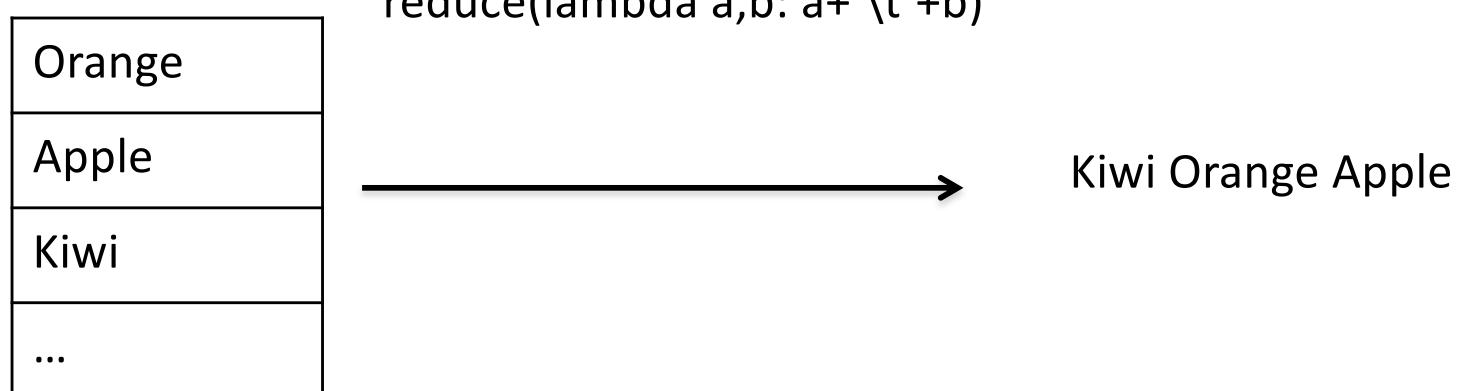
Transformation RDD vers un objet/une valeur
Python :

- `reduce()` : $\text{rdd}[\text{T}] \rightarrow \text{T}$
- `count()` : $\text{rdd}[\text{T}] \rightarrow \text{Long}$
- `countByKey()` : $\text{rdd}[(\text{K}, \text{V})] \rightarrow \text{dict}(\text{K:Long})$
- `collect()` : $\text{rdd}[\text{T}] \rightarrow [\text{T}]$
- `take(n)` : $\text{rdd}[\text{T}] \rightarrow [\text{T}]$

reduce

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

- Réduction de la dimension en utilisant une *User Defined Function (UDF)*
- Traitement distribué, aucun ordre prescrit
 - dans Spark f doit être commutative et associative car aucune garantie sur l'ordre d'application



Exercice : simplified TD-IDF

d1

one fish two fish

d2

red fish blue

$Tf * ifd(d_i, w_j)$
 $i = 1, 3$
 $w_j \in \{one, fish, \dots\}$

$TF(d, w) = |w \text{ in } d|$

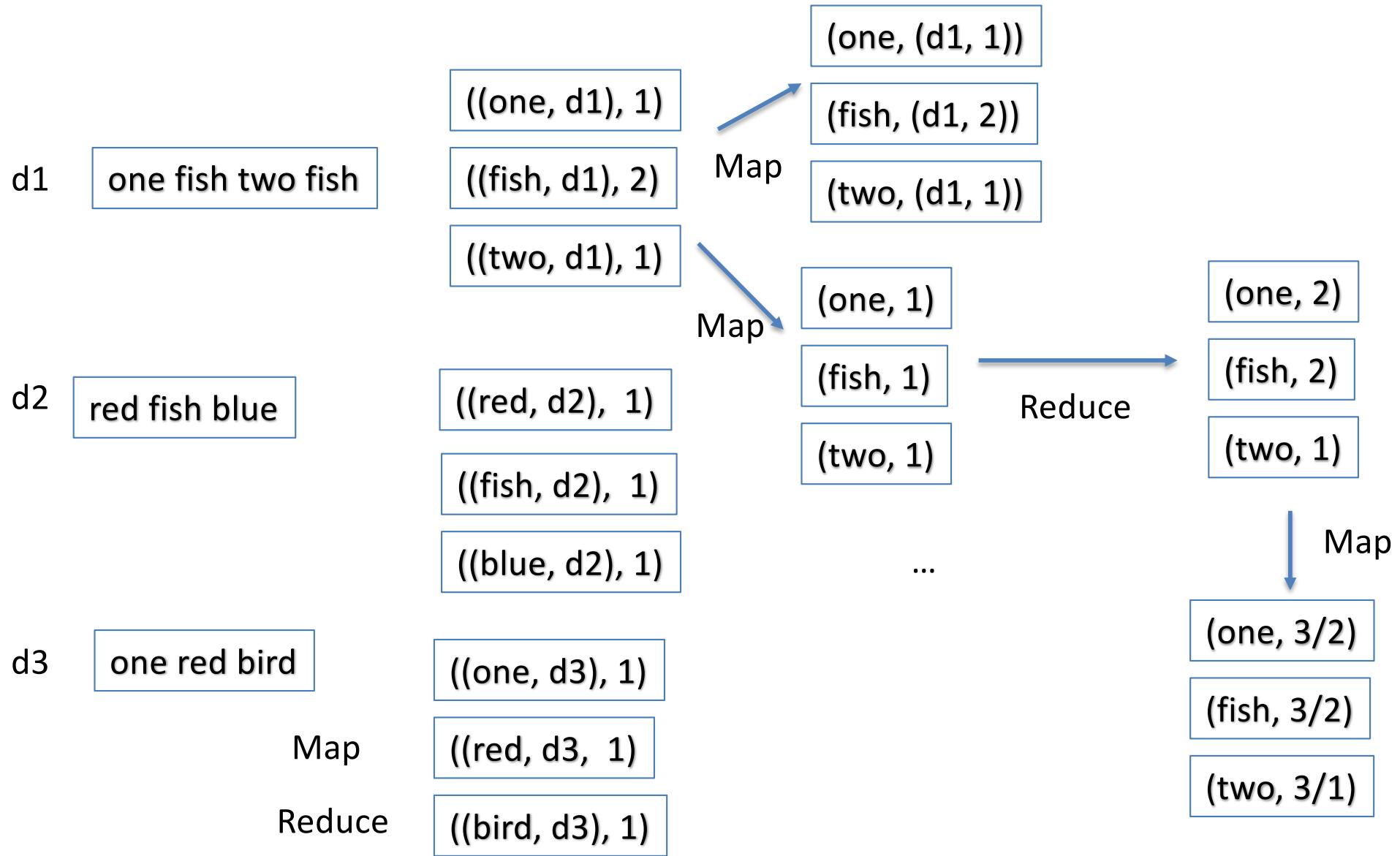
$IDF(w) = |\text{collection}| / |\text{d containing w}|$

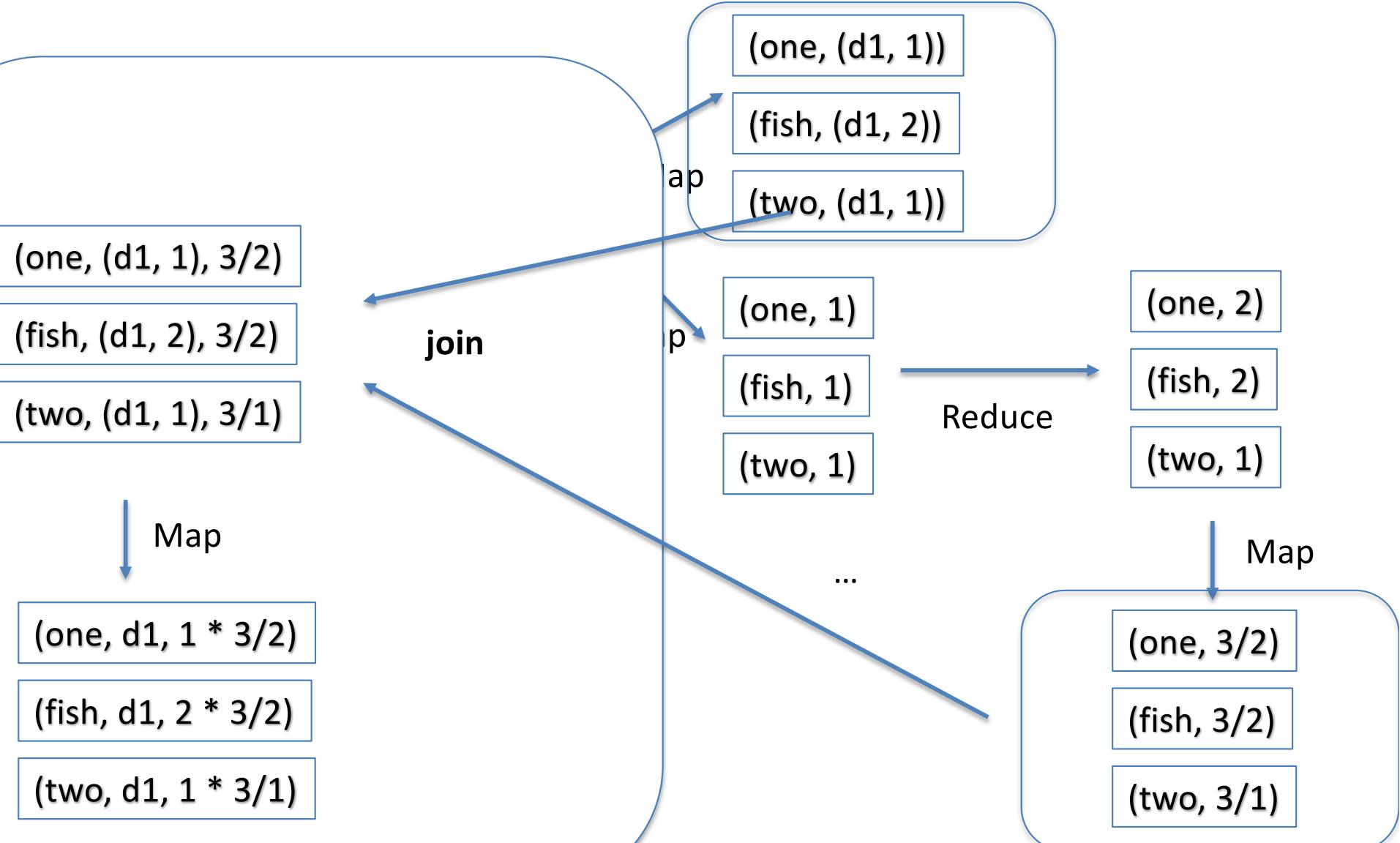
d3

one red bird

input

output





Bilan Spark RDD

- Limitations
 - Pas de schéma
 - code peu lisible, programmation fastidieuse
 - Possibilité de définir un type T : RDD[T]
 - Performances dégradées (sérialisation d'objets, GC)
 - Optimisation logique limitée
- Quand Utiliser RDD ?
 - si traitement pas exprimable avec API haut niveau
(Dataframe, ML, Graphe)

Avantages

- Maitrise couche physique
 - partitionnement (nombre, stratégie, ..)
 - persistance des données en MC (gestion buffer)
 - Optimisation bas niveau (compression, ...)

Perspectives

- Préparation de données, données complexes
 - Dataframes (opérateurs SQL décomposé), UDF
 - JSON avec structure variable

```
post_per_type = data\  
  .groupBy("event.event_type")\  
  .agg(countDistinct(data.event.event_id.post_id)  
  ).alias("count_objects"))
```

```
  ▼ _id: struct  
    $oid: string  
    code: long  
  ▼ event: struct  
    action: string  
    ▼ attachments: array  
      ▼ element: struct  
        ▶ album: struct  
        ▶ audio: struct  
        ▶ doc: struct  
        ▶ link: struct  
      ▼ note: struct  
        comments: long  
        date: long  
        id: long  
        owner_id: long  
        read_comments: long  
        title: string  
        view_url: string  
    ▼ page: struct  
      created: long  
      creator_id: long
```

Perspectives

- DeltaLake : qualité et validation des données
 - *Schema on read vs schema enforcement*

```
original_loans.printSchema()  
  
....  
root  
  |-- addr_state: string (nullable = true)  
  |-- count: integer (nullable = true)  
....
```

```
# Attempt to append new DataFrame (with new column) to existing table  
loans.write.format("delta") \  
    .mode("append") \  
    .save(DELTALAKE_PATH)
```

A schema mismatch detected when writing to the Delta table.

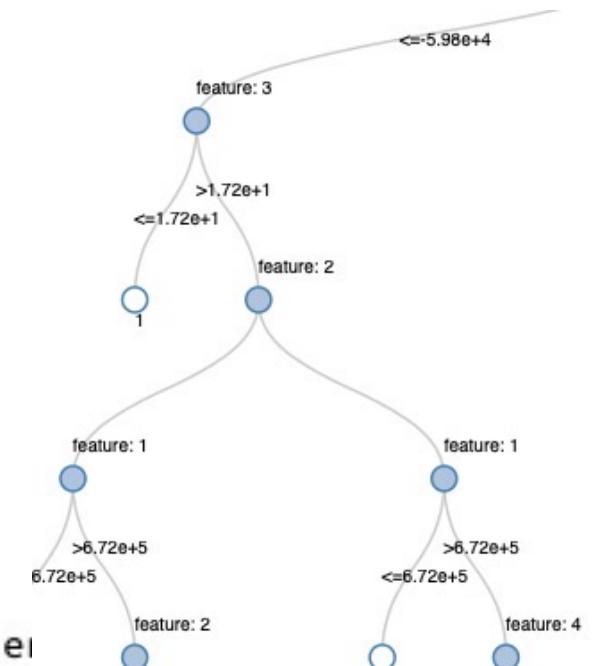


Perspectives

- Préparation des données pour le ML
 - Nettoyage des données
 - Création de pipeline ML
 - Sélection et encodage des *features*
 - Entrainement distribué

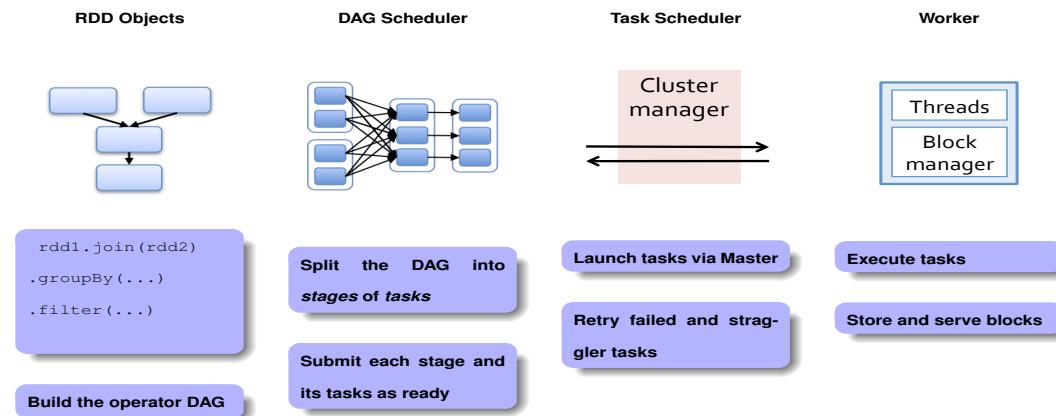
```
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import OneHotEncoderEstimator
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import DecisionTreeClassifier

# Encodes a string column of labels to a column of label indices
indexer = StringIndexer(inputCol = "type", outputCol = "typeIndexed")
```



Perspectives

- Exécution Spark



- Optimisation logique SQL

