Exécution Map Reduce et Algèbre Spark

Master DAC – Bases de Données Large Echelle

Mohamed-Amine Baazizi
baazizi@ia.lip6.fr

Octobre 2018

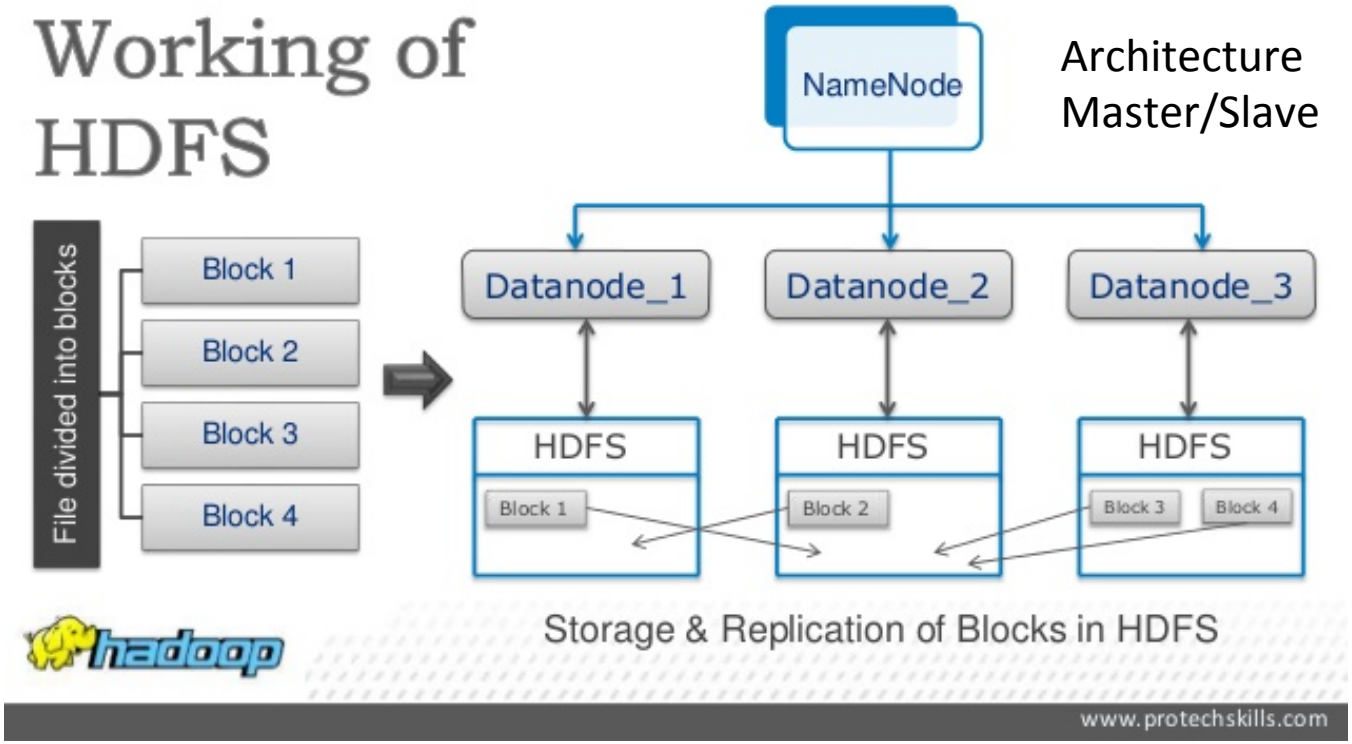
Plan

- Aperçu Hadoop file system et Map Reduce
- Exécution algèbre RDD dans Spark

HDFS (HaDoop File System)

- Système de gestion de données distribuées
- Passage à l'échelle (Peta octets, 4500 nœuds)
- Tolérance aux pannes grâce à la réplication
- Optimisé pour les lectures, écritures rares
- Fichier = plusieurs blocks
  - taille standard 128 MB
  - facteur de réplication 3, distribution sur différents nœuds

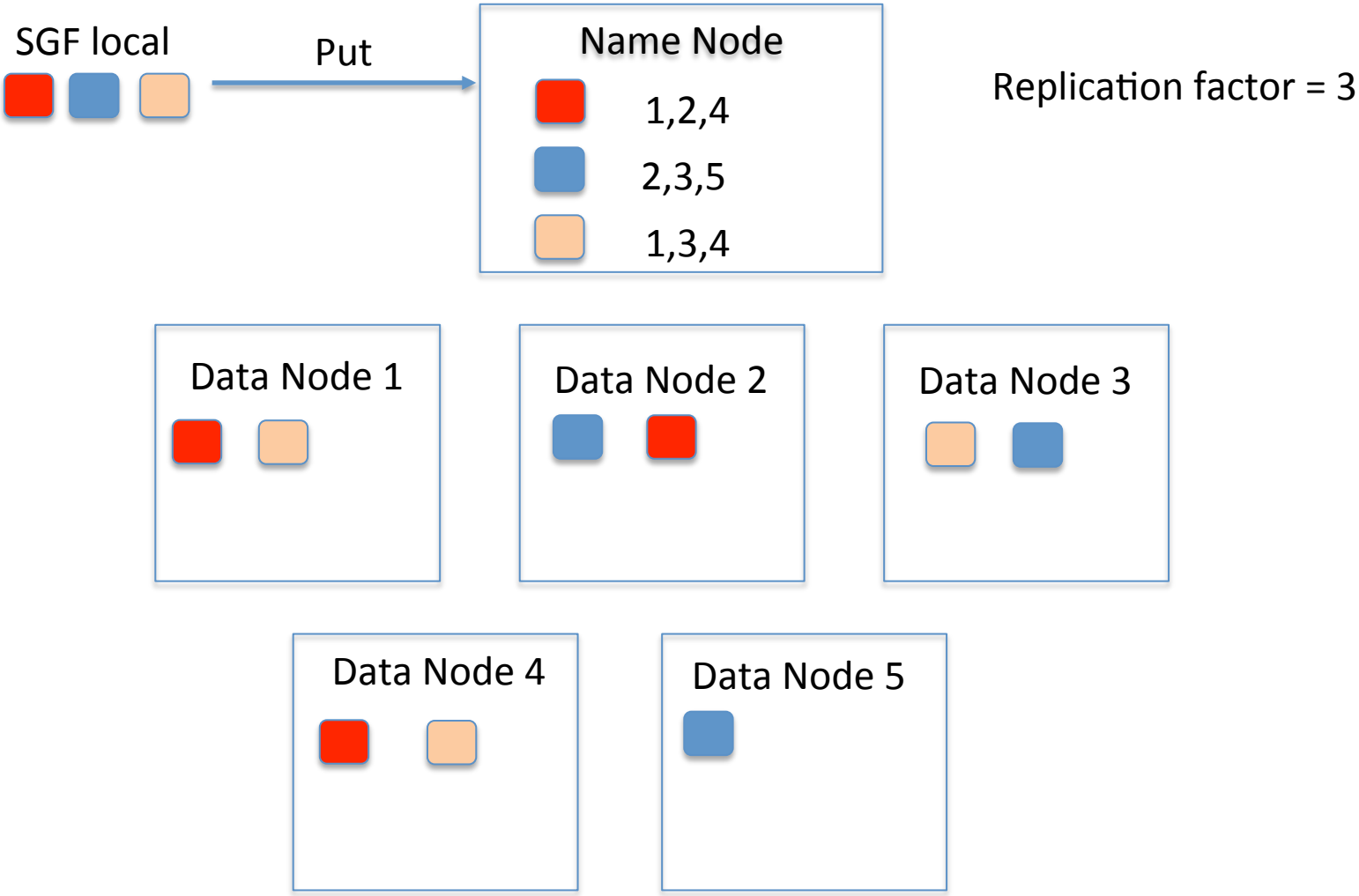
Architecture HDFS



Architecture HDFS : composants

- Un NameNode par cluster :
  - Contient métadonnées pour localiser les blocs
- Un DataNode par nœud
  - création, suppression, réplication, lecture et écriture de blocks sous l'ordre du NameNode
- Etapes de création d'un fichier
  - consulter NameNode pour disponibilité
  - découpage en blocs et envoi aux DataNode
  - demande de réplication

Illustration de HDFS



Démo HDFS

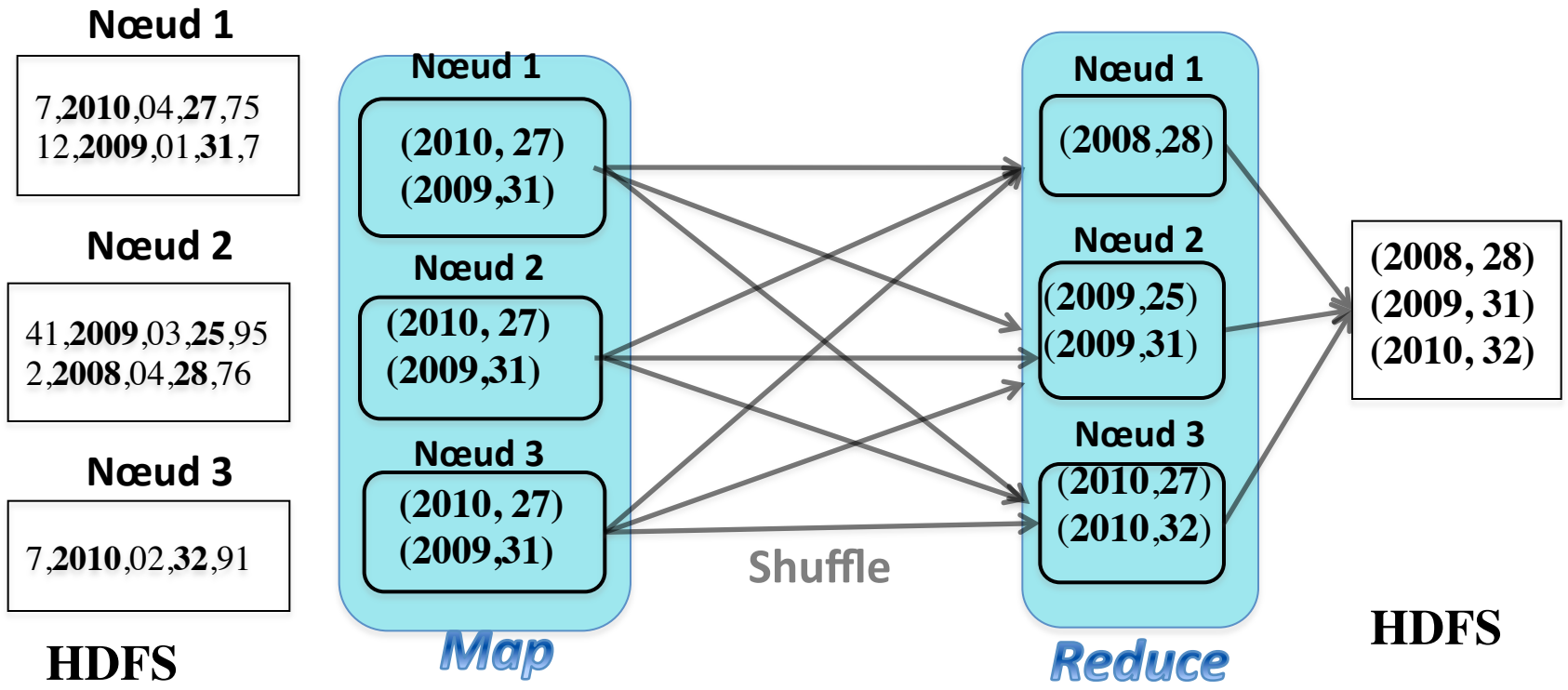
```
home$ hadoop fs -ls –h /tpch/lineitem.tbl
-rw-r--r--   3 bdle supergroup    718.9 M ...

home$ hdfs fsck /tpch/lineitem.tbl

Total size:        753849433 B
 Total dirs:       0
 Total files:      1
 Total symlinks:               0
 Total blocks (validated):6 (avg. block size 125641572 B)
 Minimally replicated blocks:  6 (100.0 %)
 Over-replicated blocks:  0 (0.0 %)
 …
 Default replication factor:      3
 ..
 Number of data-nodes:         5
 Number of racks:              1
```

Démo HDFS

```
home$ hdfs fsck /tpch/lineitem.tbl  -blocks -locations -files

0. BP-Number-IPAddr-Number: len=134217728 repl=3 [Datanode1,
Datanode2, Datanode2]

1. BP-Number-IPAddr-Number: len=134217728 repl=3 [Datanode1,
Datanode2, Datanode2]

…

5. BP-Number-IPAddr-Number: len=134217728 repl=3 [Datanode1,
Datanode2, Datanode2]
```

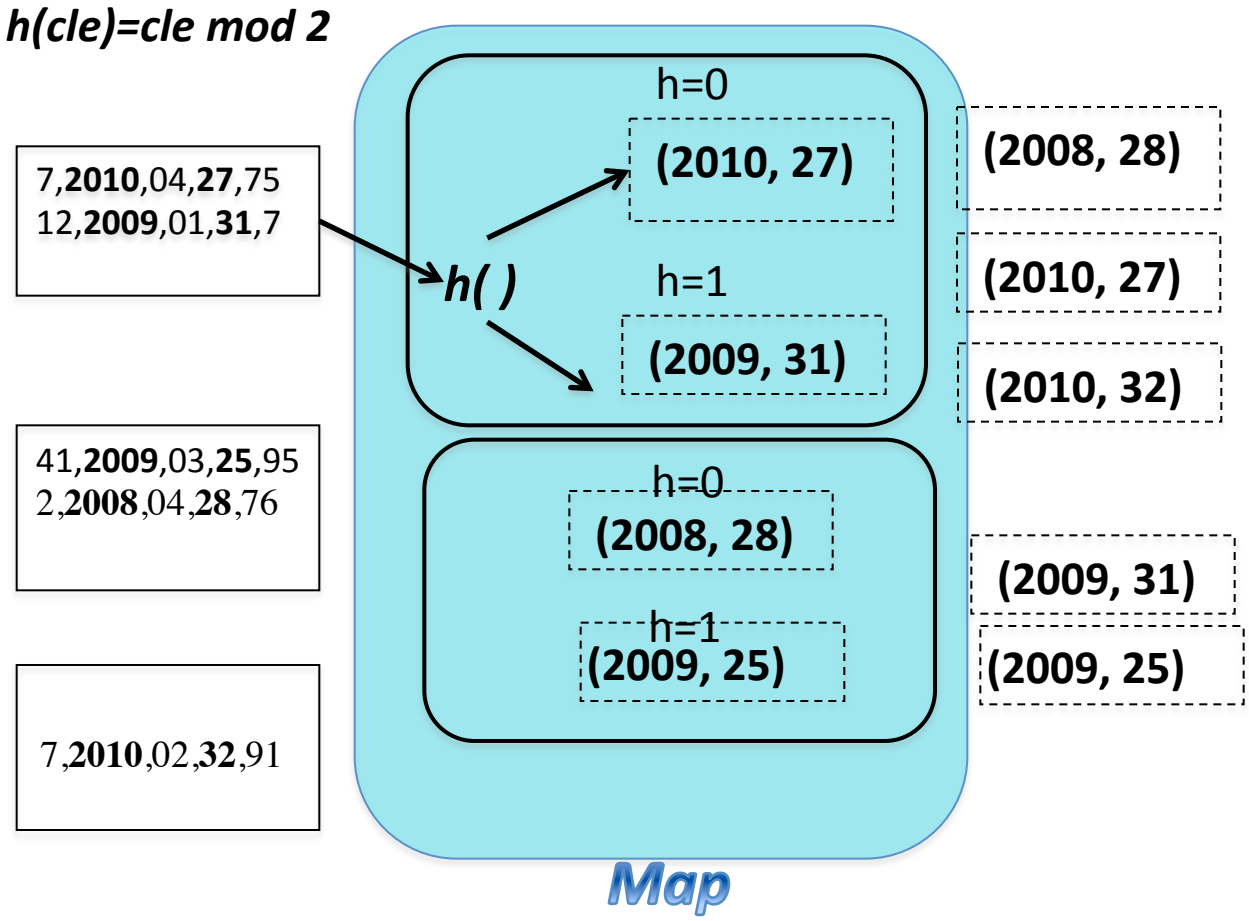
Possibilité d'utiliser interface graphique

Exécution Hadoop Map Reduce

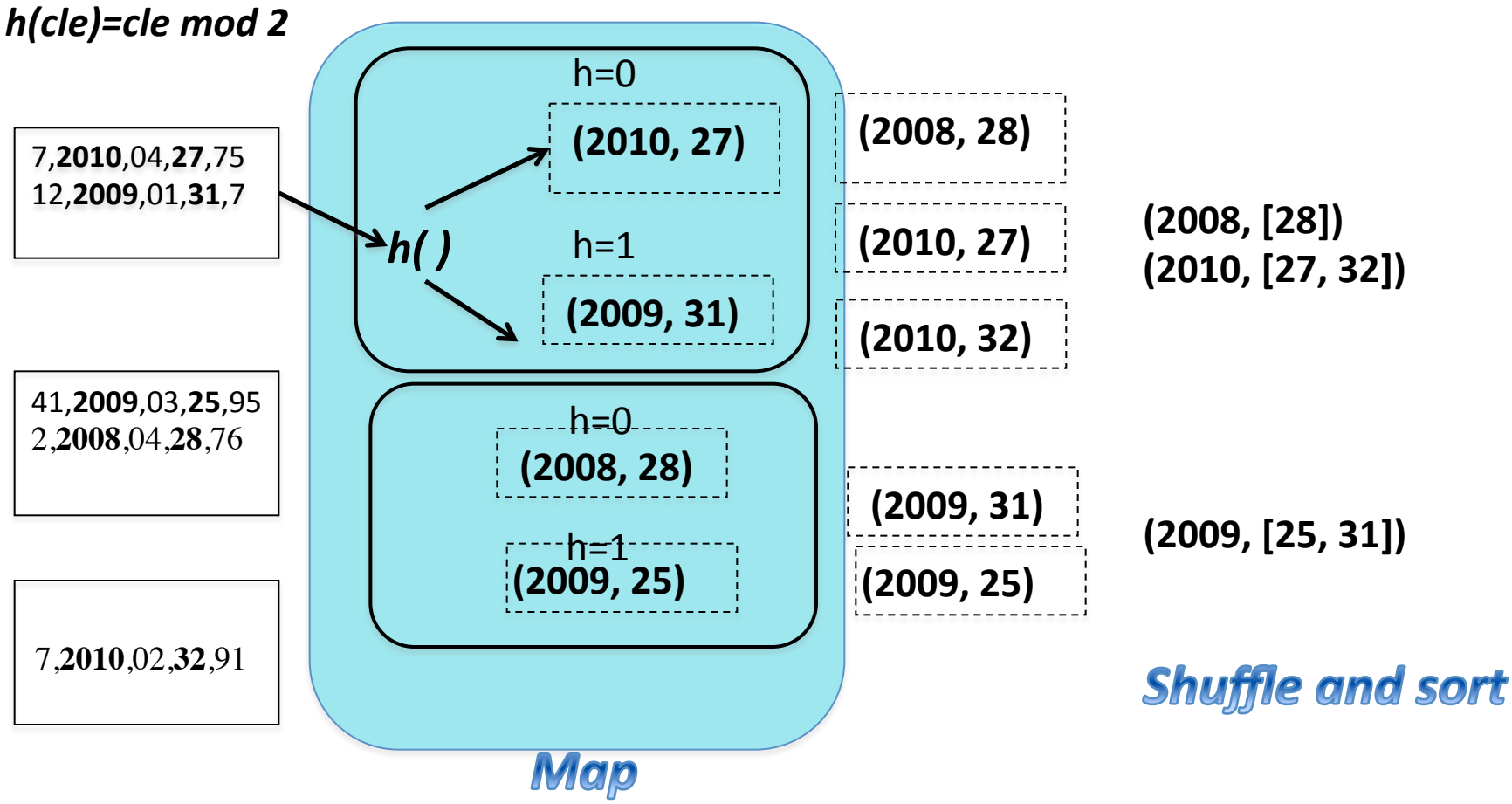


Entrée : n-uplets (station, **annee**, mois, **temp**, dept)
Résultat : select annee, Max(temp) group by annee

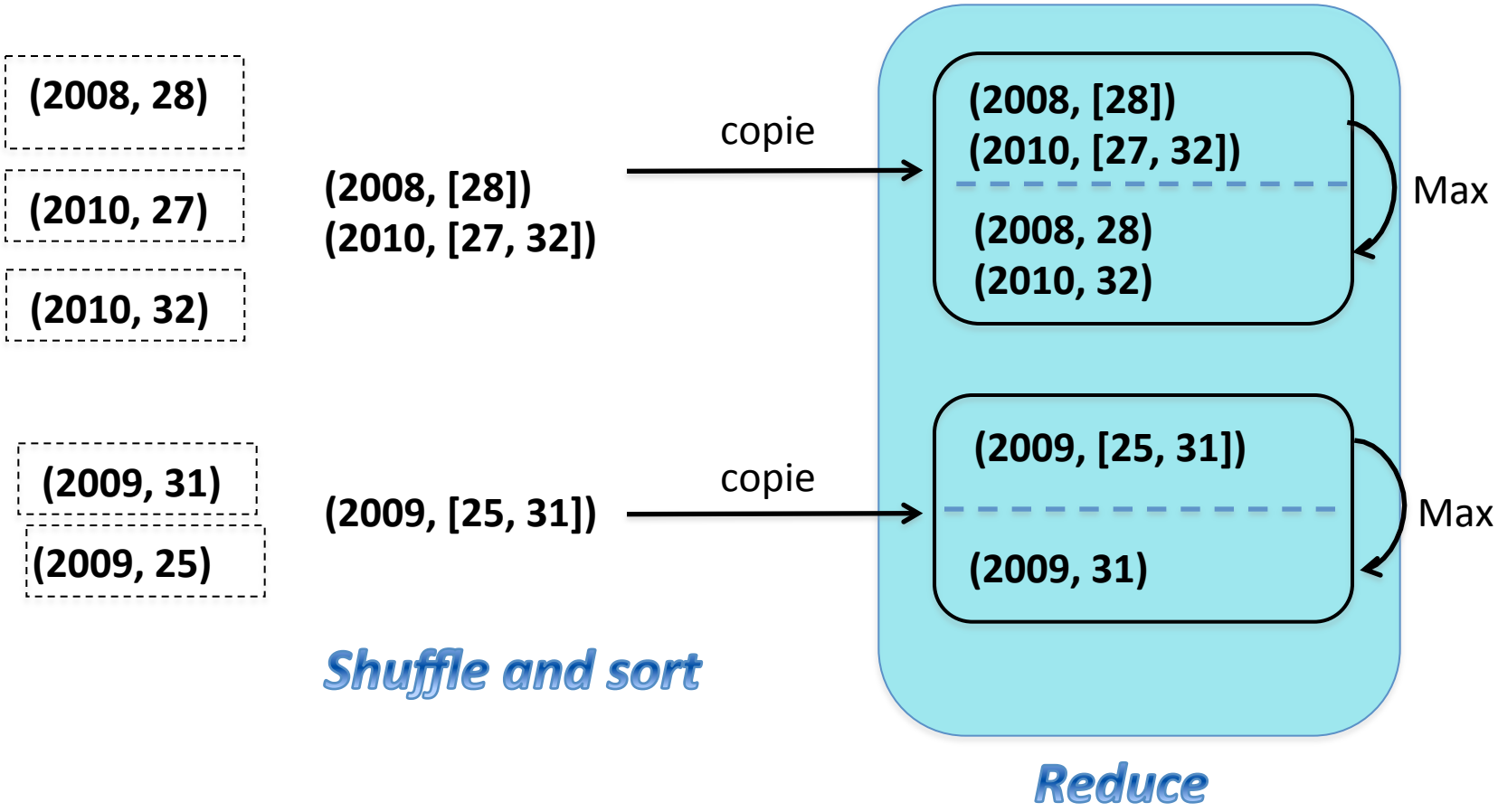
Exécution Map Reduce



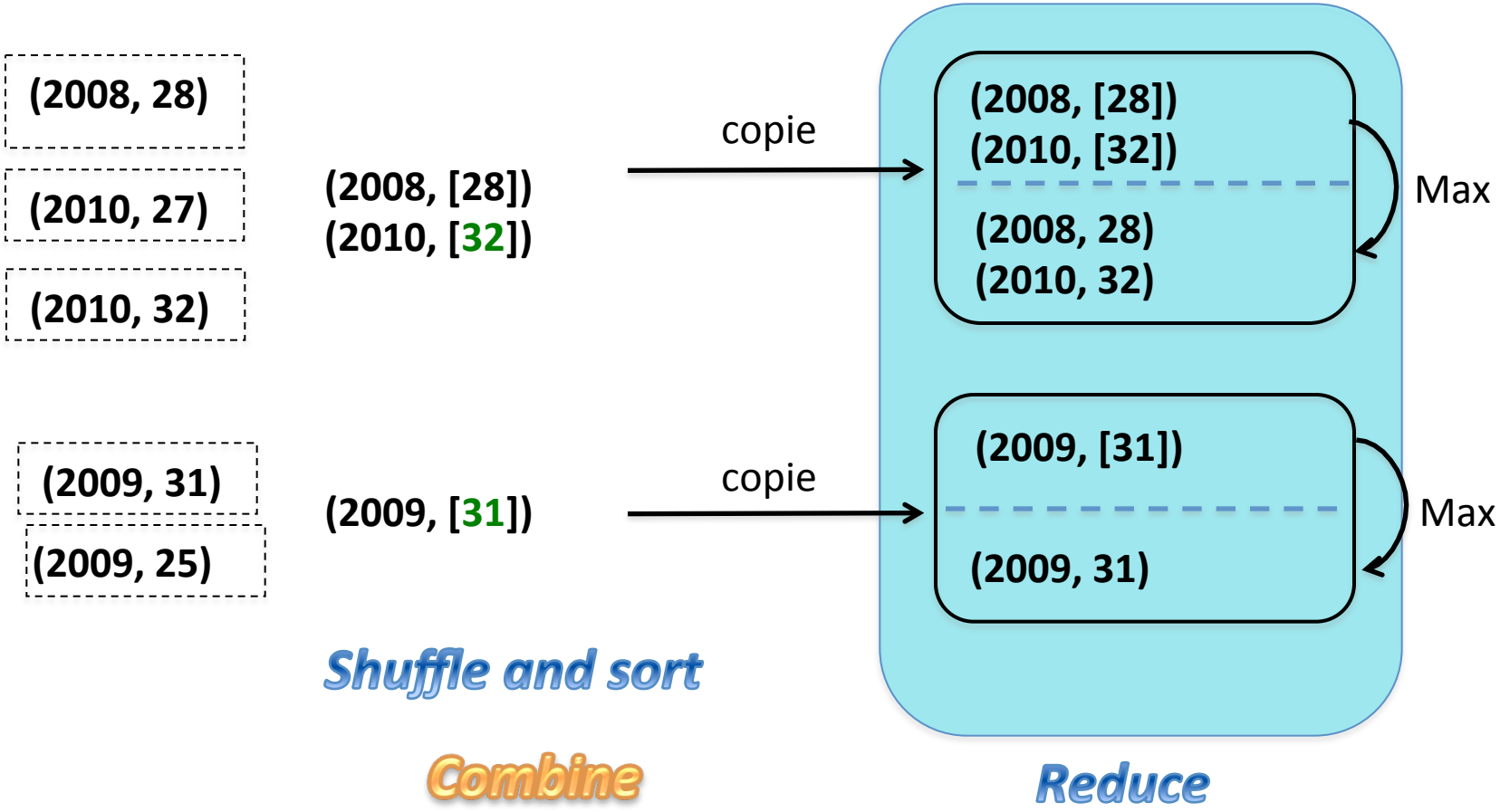
Exécution Map Reduce



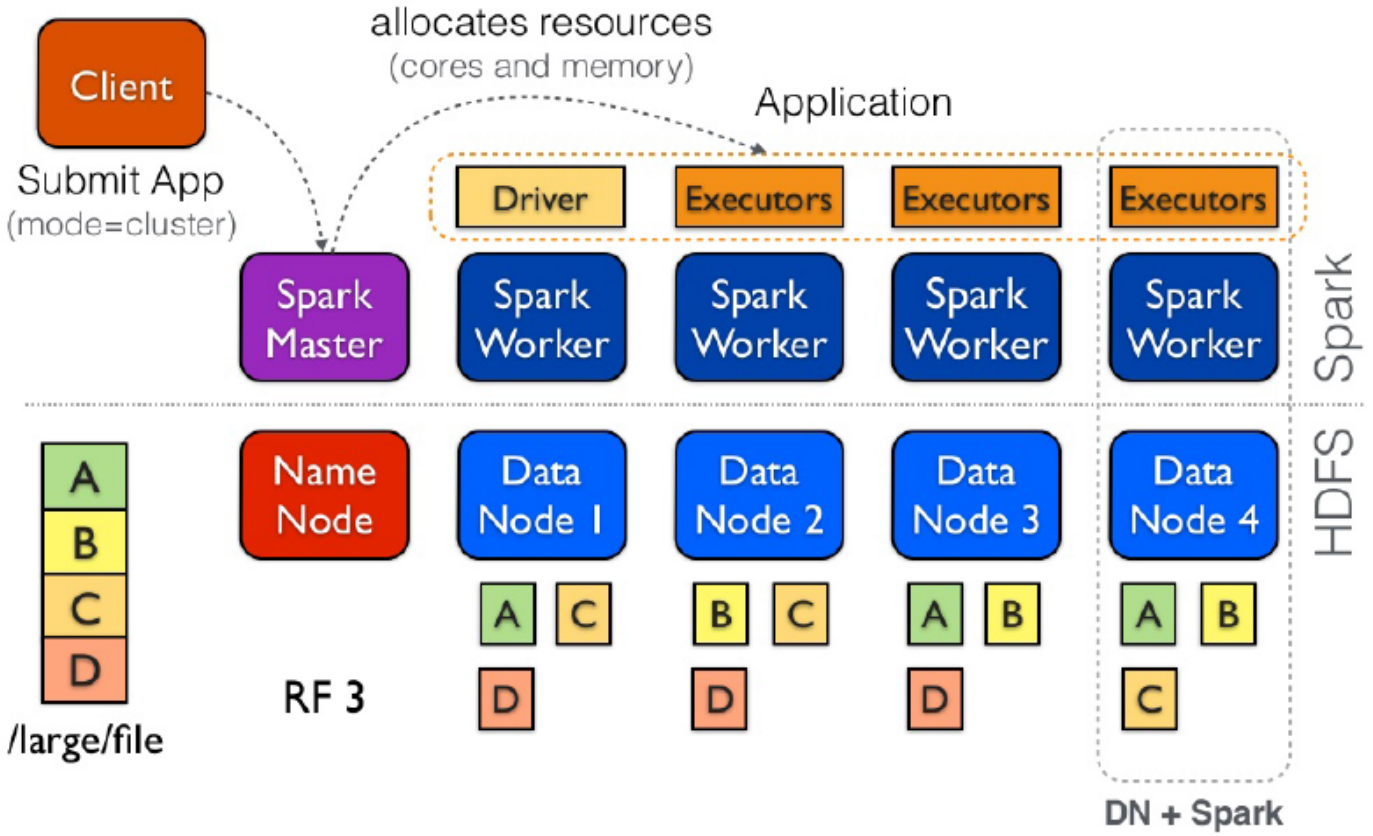
Exécution Hadoop Map Reduce



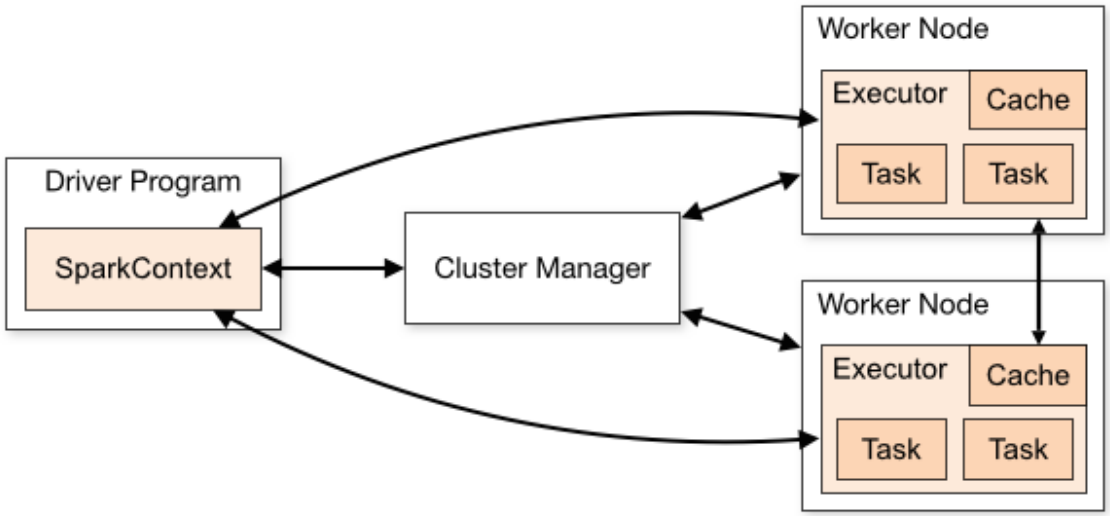
Exécution Hadoop Map Reduce



Spark avec HDFS



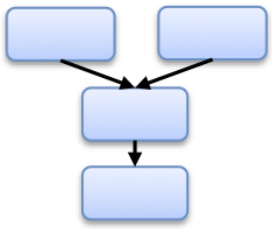
Composants Spark



- Driver : prog. utilsant API Spark pour spécifier les calculs d'une application
- Executor : processus lancé par une application, un par worker (par défaut)
- Task : unité d'exécution réalisée par un executor (plusieurs)

Cycle de vie d'un programme

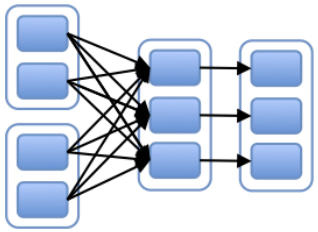
RDD Objects



```
rdd1.join(rdd2)
.groupBy(...)
.filter(...)
```

Build the operator DAG

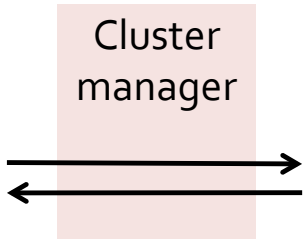
DAG Scheduler



Split the DAG into *stages* of *tasks*

Submit each stage and its tasks as ready

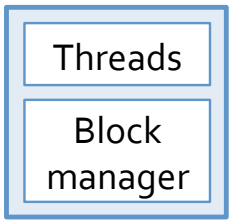
Task Scheduler



Launch tasks via Master

Retry failed and strag-gler tasks

Worker



Execute tasks

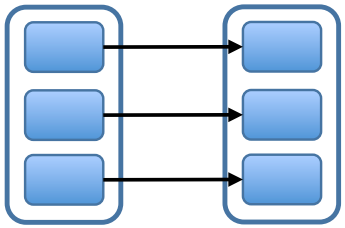
Store and serve blocks

Terminologie

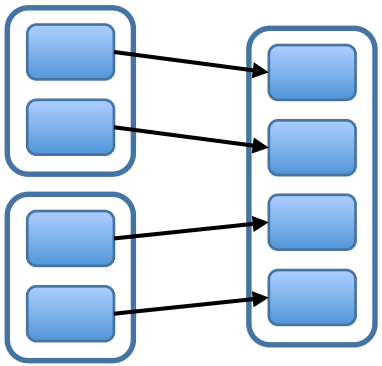
- Opération
  - Transformation : crée une nouvelle RDD à partir d'autre(s) RDD
    - retourne RDD du type de la transformation (MappedRDD, …)
    - locale (map, filter) ou distribuée (join, reduceByKey)
  - Action : évalue la RDD en exécutant la chaine de transformation
    - retourne type de base ou *User Defined Type*
- Stage : Séquence de transformations locales terminée par une transformation distribuée ou par une action
  - exécution *pipelined* des transformations locales
- Plan : Séquence de *stages* terminée par une action

Transformations locales

- Application sur les partitions locales
  - pas de shuffle (pas de matérialisation de données intermédiaires)
  - map, filter, union, flatMap, mapValues



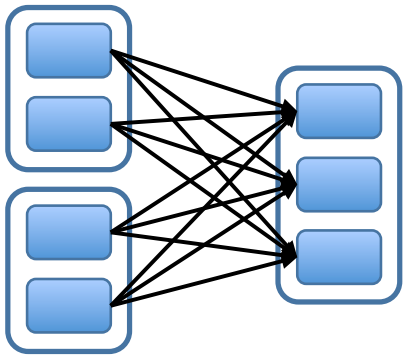
map, filter



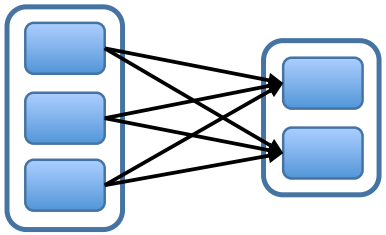
union

Transformations distribuées

- Accès requis à toutes les partitions
  - Matérialisation du résultat intermédiaire pour le passer au stage suivant
  - join, reduceByKey, groupByKey, distinct, intersect



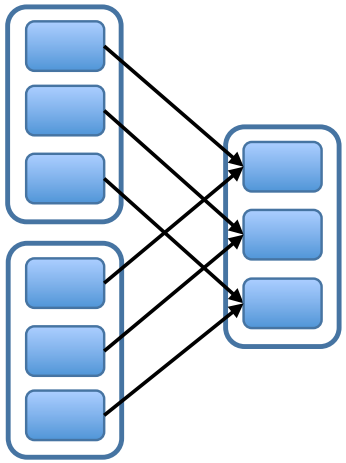
join with inputs not co-partitioned



groupByKey

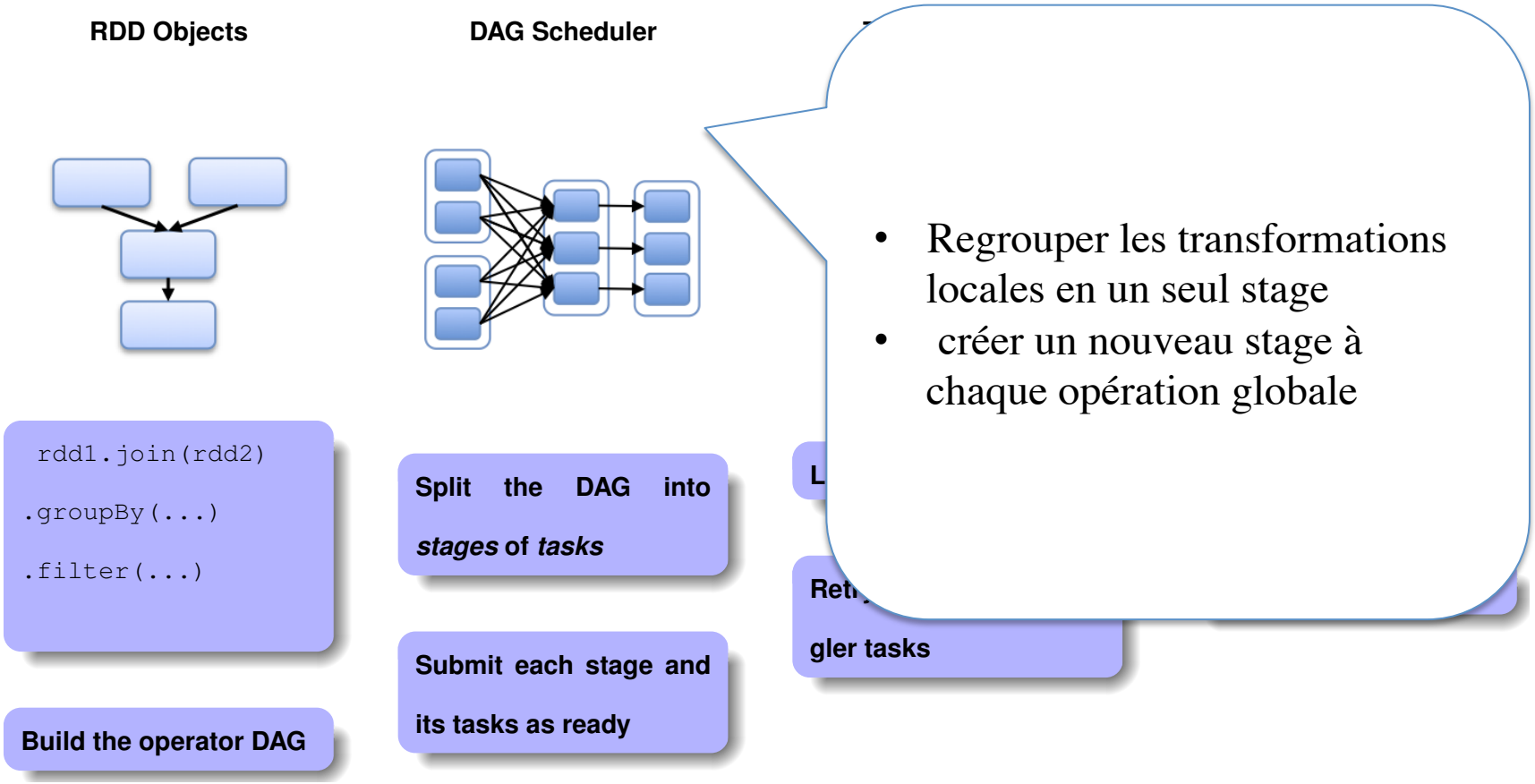
Transformations distribuées s'exécutant localement

- Tirer profit du partitionnement effectué par transformation antérieure
  - jointure sur une clé pour laquelle les deux relations sont déjà partitionnées



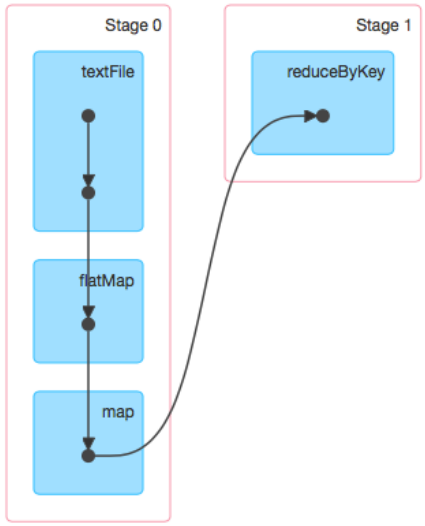
join with inputs co-partitioned

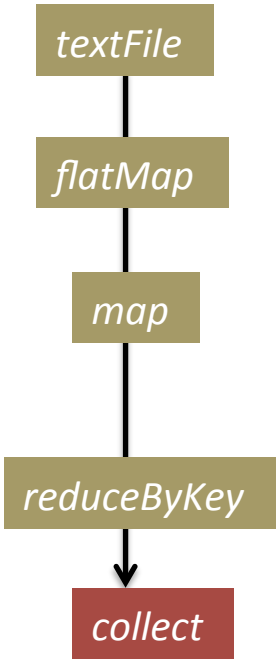
Génération du plan d'exécution



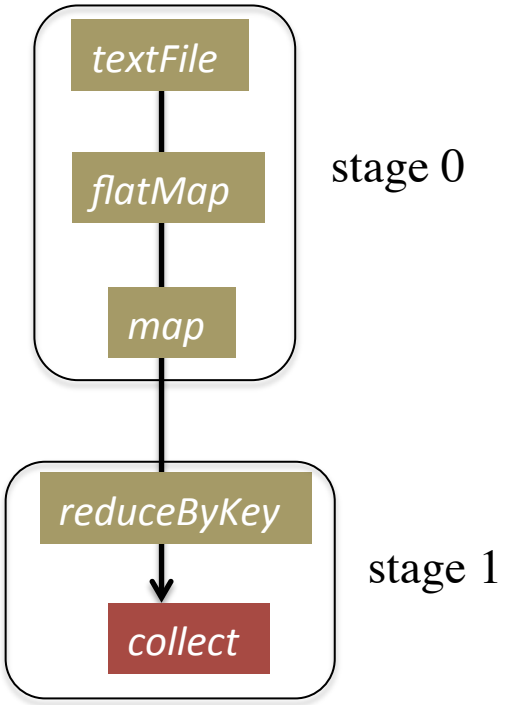
Wordcount

```
val lines = sc.textFile(filename)
val words = lines.flatMap(x=>x.split(" "))
val pairs = words.map(x=>(x,1))
val counts = pairs.reduceByKey(_+_)
val results = counts.collectAsMap
```



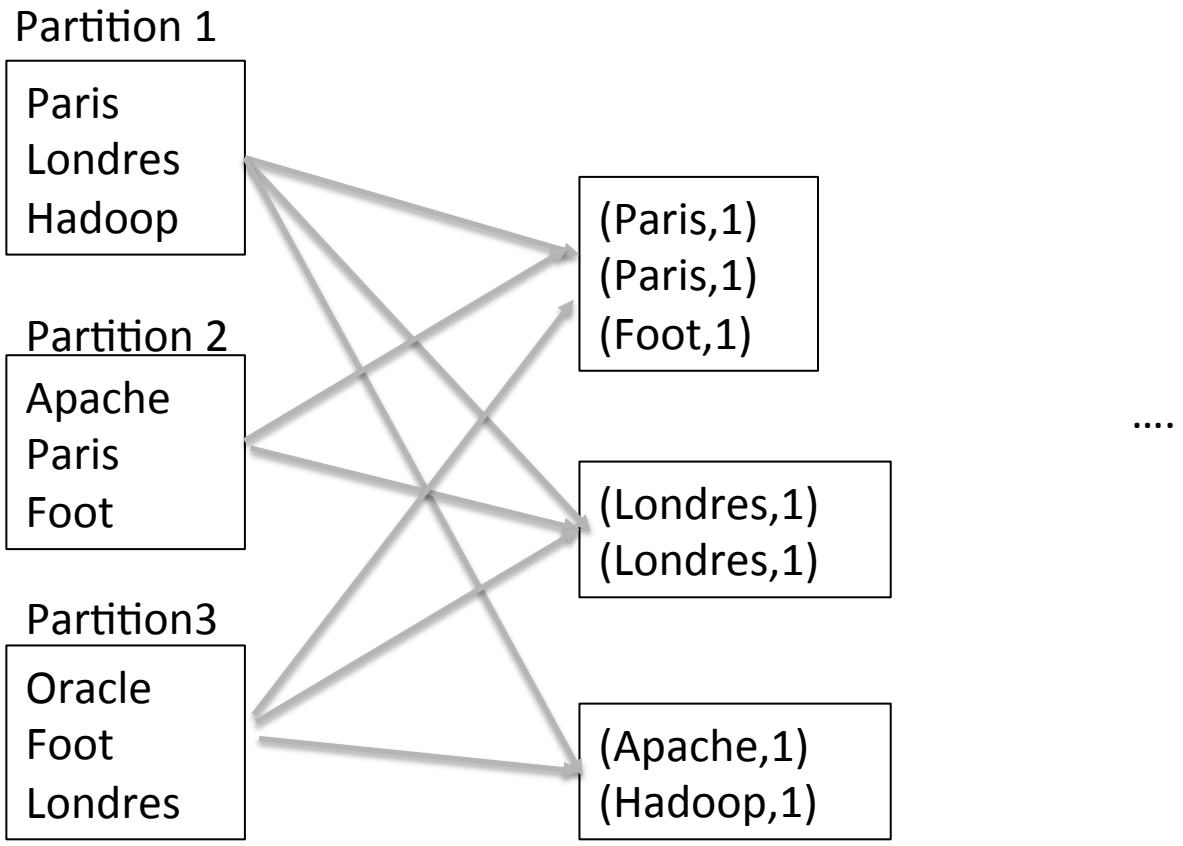


DAG d'opérateurs



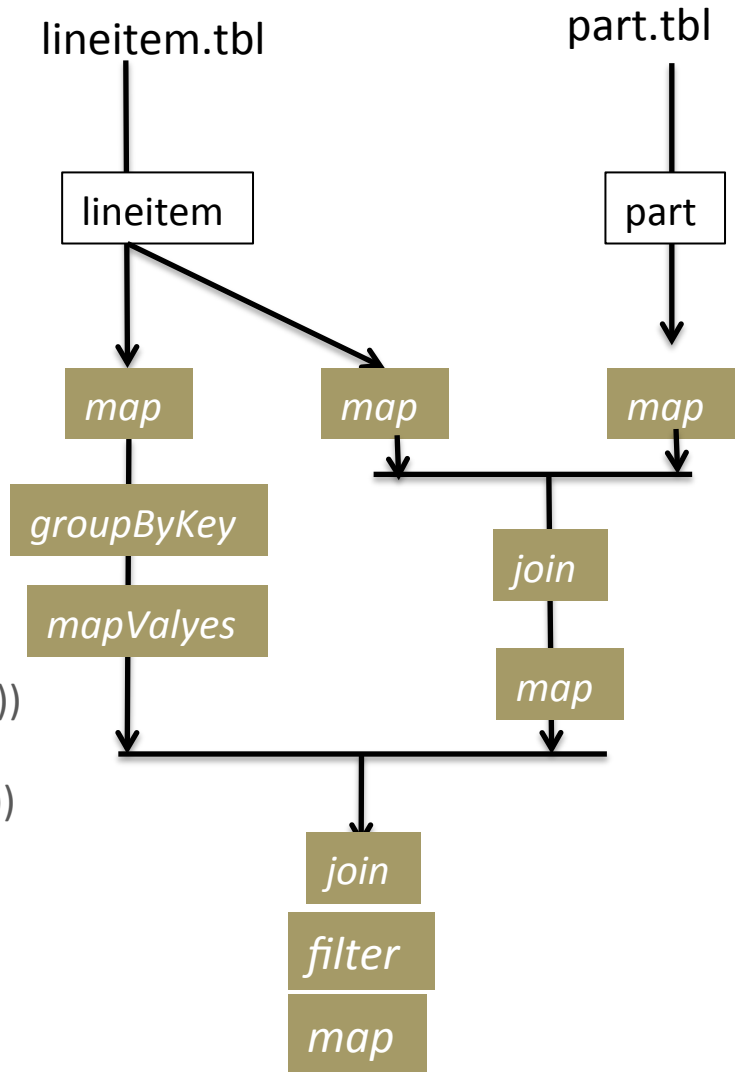
DAG de Stages

Wordcount illustré



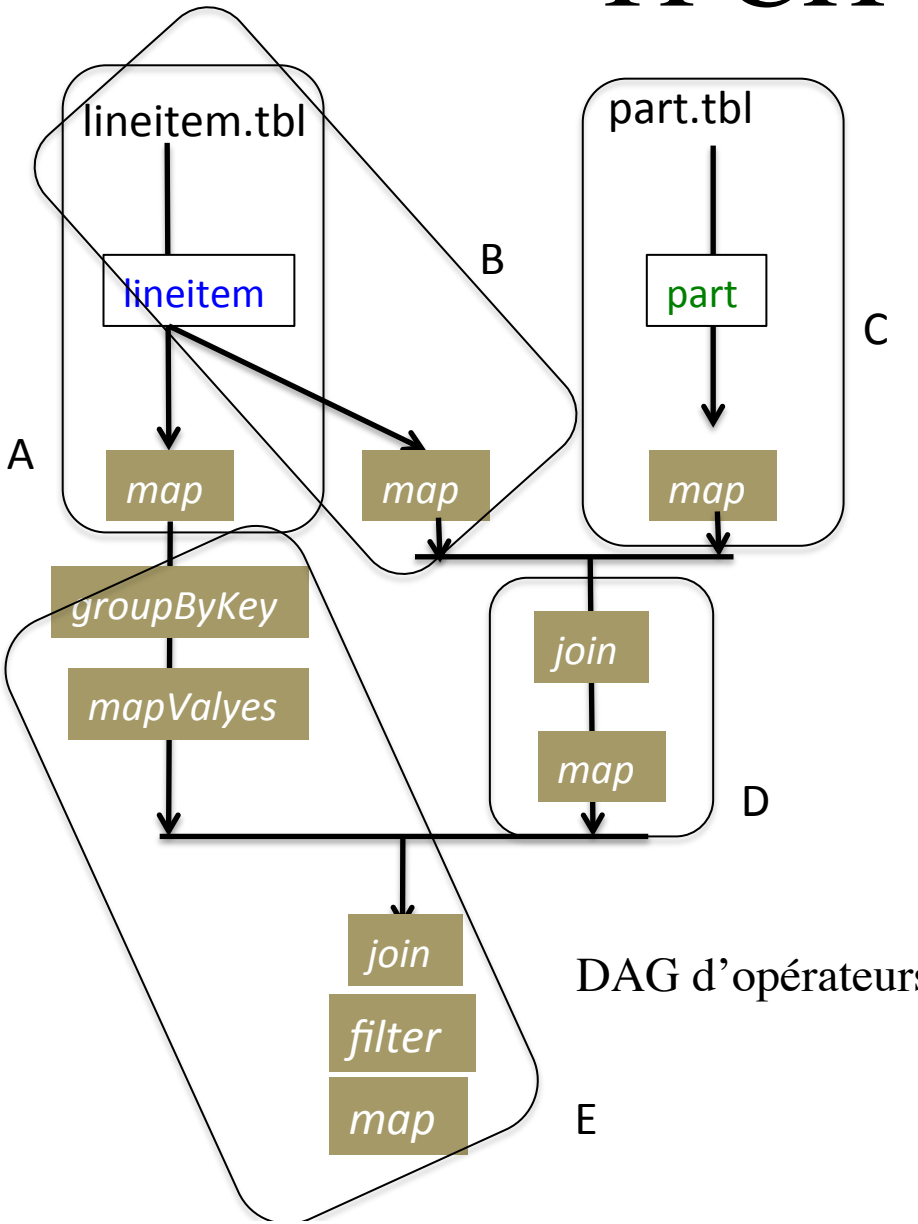
TPCH Q17

```
val inner = lineitem.
            map{case(partkey,quantity,_)=>
                (partkey,quantity)}
            .groupByKey
.mapValues(x=>.2*myAvg(x))

val outer = lineitem.map{case(partkey,quantity,ext)=>
                         (partkey,(quantity,ext))}
            .join(part.map(x=>(x,null)))
            .map{case(partkey,((quantity,ext),_))=>
                         (partkey,(quantity,ext))}

val query = inner.join(outer)
            .filter{case(partkey,(t1,(quantity,extended)))
                =>quantity<t1}
            .map{case(partkey,(t1,(quantity,extended)))
                =>extended}

val res = query.sum/7
```

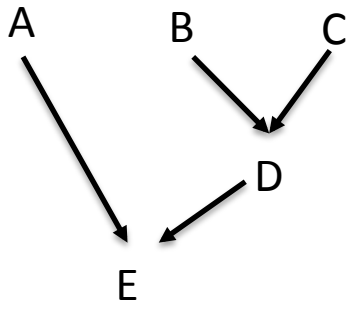


DAG d'opérateurs

TPCH Q17

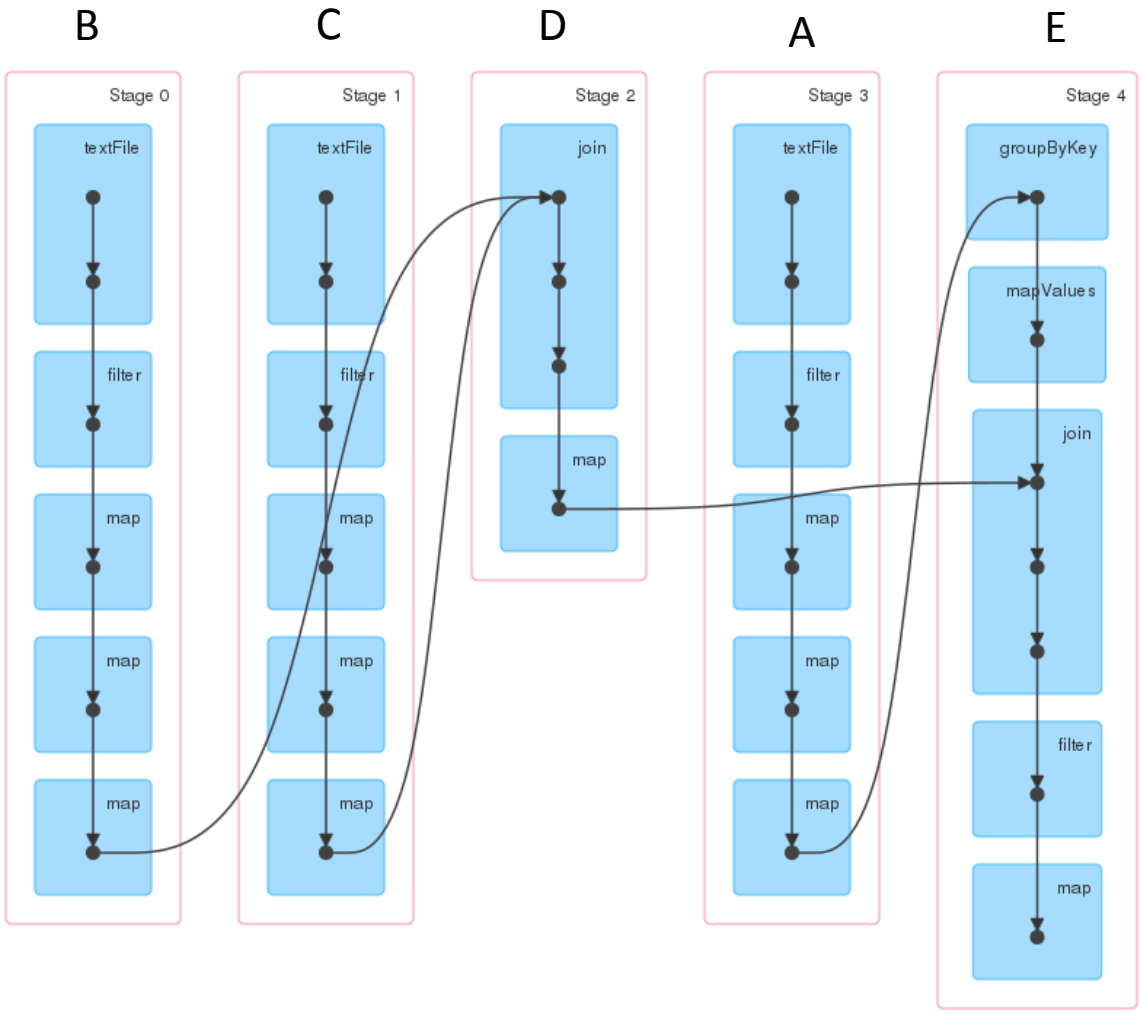


DAG d'opérateurs

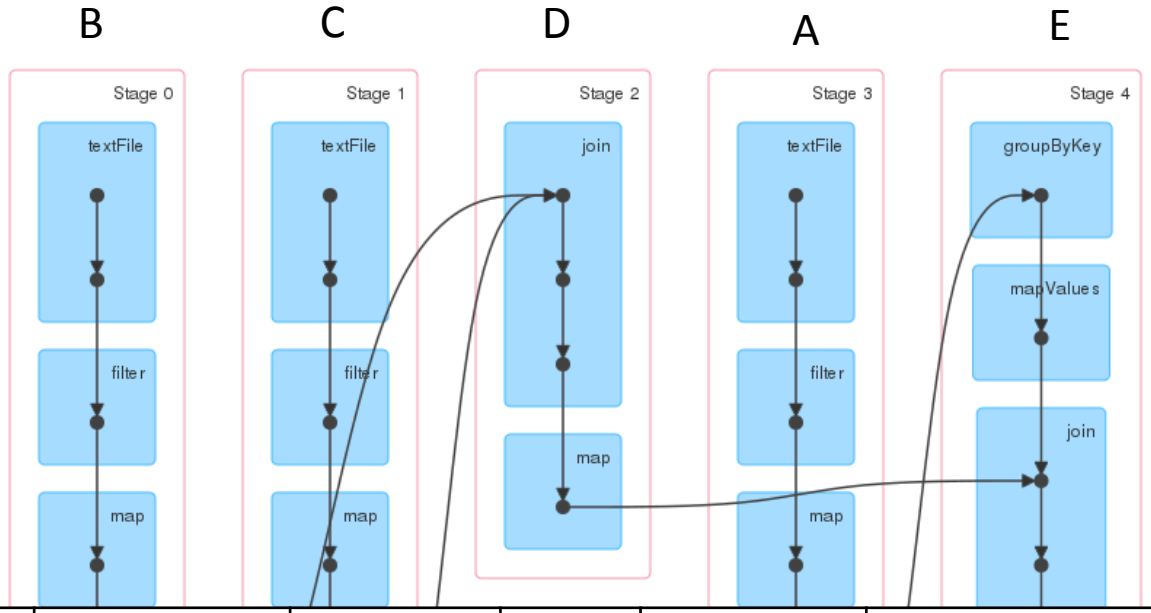


DAG de Stages

Enchainement des stages

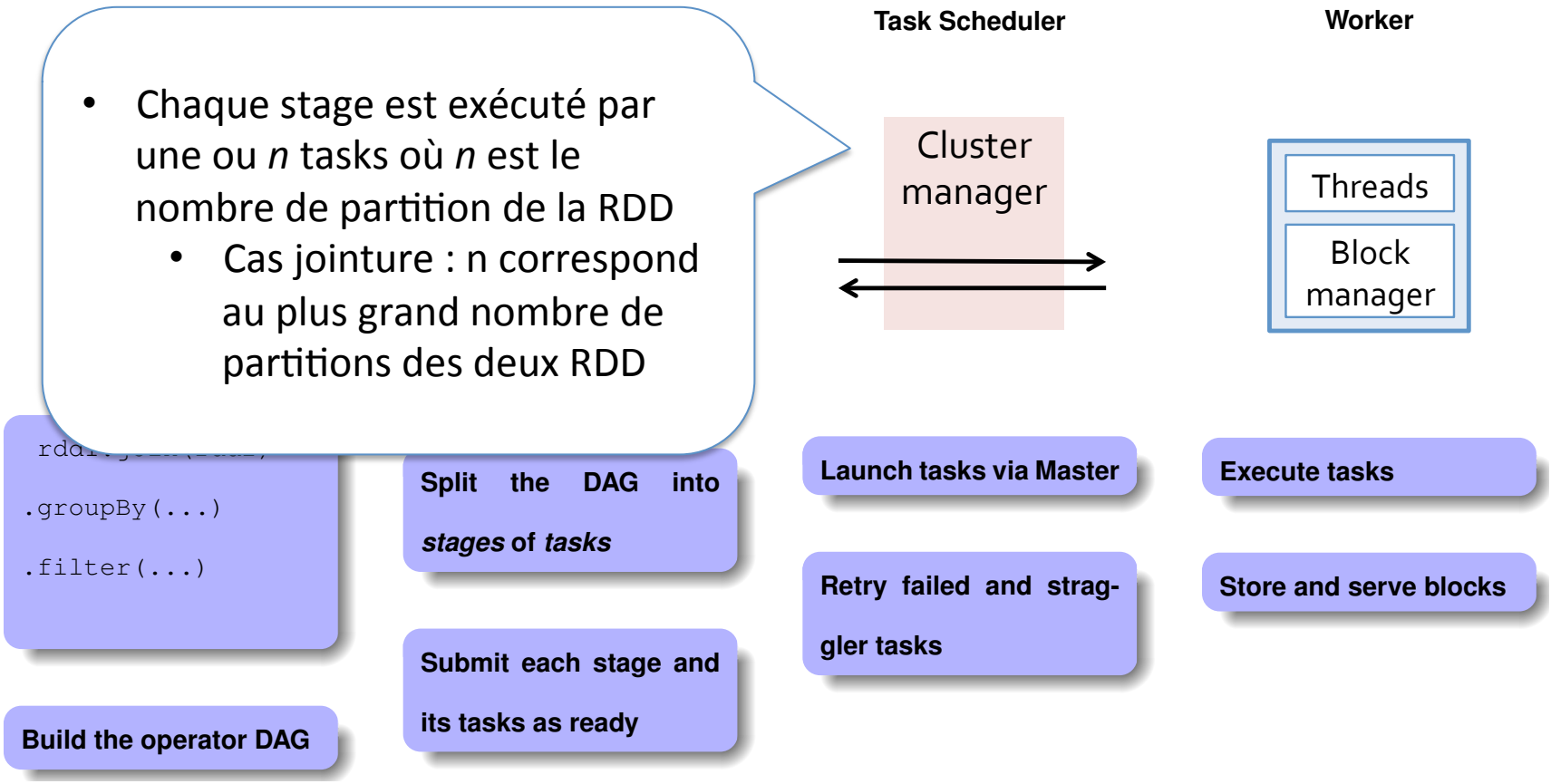


Enchainement des stages



Stage Id	Desc.	Submitted	duration	tasks	Input	Shuffle Read	Shuffle Write
4	sum	12:12:49	2 s	6		74.5 MB	
3	map	12:12:41	5 s	6	719.2 MB		72.0 MB
2	map	12:12:47	2 s	6		72.0 MB	783.4 KB
1	map	12:12:41	2 s	2	22.9 MB		783.4 KB
0	map	12:12:41	5 s	6	719.2 MB		71.2 MB

Exécution du plan

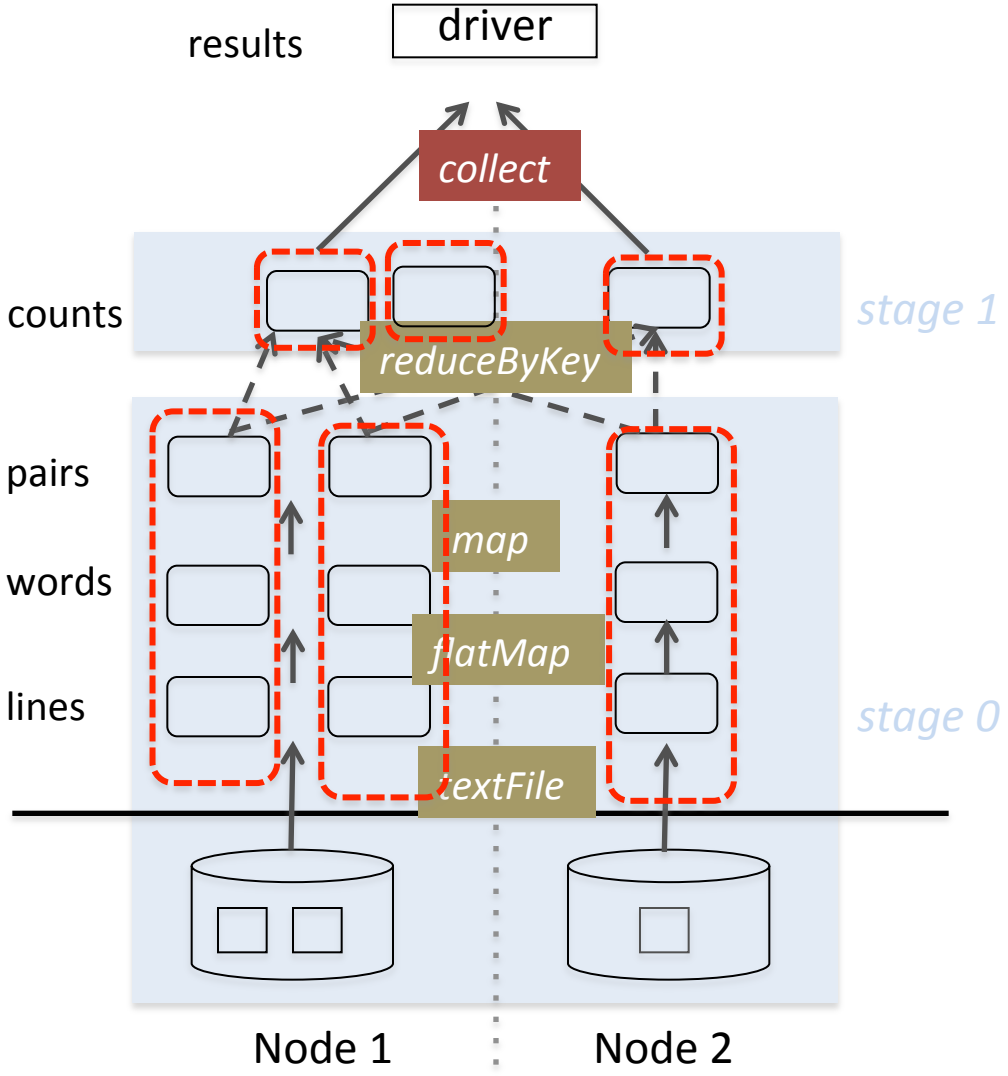


Exécution du plan

- Affectation de *tasks* aux stages en privilégiant la localité des données
  - exécuter une tâche dans le nœud de la partition
    - si partition dans HDFS, alors tâche du même dataNode
- Matérialisation des résultats produits pour chaque stage
  - en cas de panne, par exemple perte de certaines partitions, ne recalculer que celles-ci

Exécution du plan Wordcount

```
val lines = sc.textFile(filename)
val words = lines.flatMap(x=>x.split(" "))
val pairs = words.map(x=>(x,1))
val counts = pairs.reduceByKey(_+_)
val results = counts.collectAsMap
```



partition d'une RDD

task

action

transformation

Scénario réel

config. matériel : 5 nœuds de calcul+stockage, 20 cœurs/nœud
Données tiennent sur 22 blocs HDFS

Noeud	**Nb. tasks**	**Input (Mo)**	**Shuffle Write (Mo)**	**Temps**
1	5	640.3	41.3	1.2 min
2	4	512.3	32.8	49 s
3	4	494.2	32.8	47 s
4	4	512.3	34.3	46 s
5	5	640.3	41.9	1.2 min
Total	*22*	*~2.799*	*181.1*	

Stage 0
ShuffleMapStage

Noeud	**Nb. tasks**	**Shuffle Read (Mo)**	**Temps**
1	11	91.6	1.8 min
4	11	91.5	1 min
Total	*22*	*181.1*	

Stage 1
ResultStage

Taille Shuffle write réduite
car utilisation systématique du *combiner*

Optimisation manuelle

- Stage 1 utilise 22 tasks sur petites partitions
- fusionner les partitions pour réduire le nombre de tasks

coalesce(numPartitions: Int, shuffle: Boolean = false, partitionner …)

quand *shuffle=true* possibilité de déplacer les données

```
val lines = sc.textFile(filename)
val words = lines.flatMap(x=>x.split(" "))
val pairs = words.map(x=>(x,1))
val counts = pairs.reduceByKey(_
+_).coalesce(2, false)
val results = counts.collectAsMap
```

Noeud	Nb. tasks	Shuffle Read (Mo)	Temps
1	1	91.6	23 s
4	1	91.5	25 s
Total	*2*	*181.1*	

Stage 1

Exécution TPCH Q17

config. matériel : 5 nœuds de calcul+stockage, 20 coeurs/nœud
Lineitem : 719.2 MB, 6 blocs HDFS, 6*10^6 tuples
Part : 22.9 MB, 2 blocs HDFS, 2*10^5 tuples

Noeud	**Nb. tasks**	**Input (Mo)**	**Shuffle Write (Mo)**	**Temps (s)**
1	2	207.0	20.5	9
2	1	128.1	12.7	5
3	2	256.1	25.4	10
4	1	128.1	12.7	5
Total	*6*			

Stage 0

Noeud	**Nb. tasks**	**Input (Mo)**	**Shuffle Write (Mo)**	
2	1	11.4	<1	5
3	1	11.5	<1	10
Total	*2*			

Stage 1

Exécution TPCH Q17

Noeud	**Nb. tasks**	**Shuffle Read (Mo)**	**Shuffle Write (Mo)**	**Temps (s)**
1	1	12.0	7.3	2
2	1	idem	idem	idem
3	1	idem	idem	idem
4	1	idem	idem	idem
5	2	24.0	14.6	5
Total	*6*			

Stage 2

Noeud	**Nb. tasks**	**Input (Mo)**	**Shuffle Write (Mo)**	**Temps (s)**
1	2	207.0	8.8	9
2	1	128.1	5.4	4
3	1	idem	idem	idem
4	1	idem	idem	idem
5	1	idem	idem	idem
Total	*6*			

Stage 3

Noeud	**Nb. tasks**	**Shuffle Read(Mo)**	**Temps**
1	6	75	11

Stage 4

Exercice : SQL en Spark RDD

- Select : map
- Where : filter
- Jointure : map puis join
- Aggregation (fct) :
  - reduceByKey (fct), si fct associative
  - groupByKey et fct' équivalente à fct, sinon
- Imbrication dans le from ou le select :
  - résultat intermediaries stocké dans une variable

Exercice

```
SELECT sum(l_extendedprice) / 7.0 AS avg_yearly
FROM (SELECT l_partkey, 0.2* avg(l_quantity) AS t1
        FROM lineitem GROUP BY l_partkey) AS inner,
       (SELECT          l_partkey,l_quantity,l_extendedprice
       FROM lineitem, part
       WHERE p_partkey = l_partkey) AS  outer
WHERE outer.l_partkey = inner.l_partkey; AND outer.l_quantity < inner.t1;
```

Q17 TPCH modifiée

LINEITEM(ORDERKEY,PARTKEY,SUPPKEY,LINENUMBER,QUANTITY,EXTENDEDPRICE, …)
PART(PARTKEY,NAME,MFGR,BRAND,TYPE,SIZE,CONTAINER,RETAILPRICE,COMMENT)

```
val lineitem = sc.textFile(lineitem_t).map(x=>x.split(","))
            .map(x=>(x(1).toInt,x(4).toInt,x(5).toDouble))

val part = sc.textFile(part_t).map(x=>x.split(","))
            .map(x=>(x(0).toInt))
```

Exercice

SELECT l_partkey, 0.2* **avg**(l_quantity) AS t1
FROM lineitem GROUP BY l_partkey

inner

```
def myAvg(tab:Iterable[Int])=tab.reduce(_+_)/tab.size
val inner = lineitem.map{case(partkey,quantity,_)=>(partkey,quantity)}
            .groupByKey.mapValues(x=>.2*myAvg(x))
```

(**SELECT**
l_partkey,l_quantity,l_extendedprice
FROM lineitem, part
WHERE p_partkey = l_partkey) **AS** outer

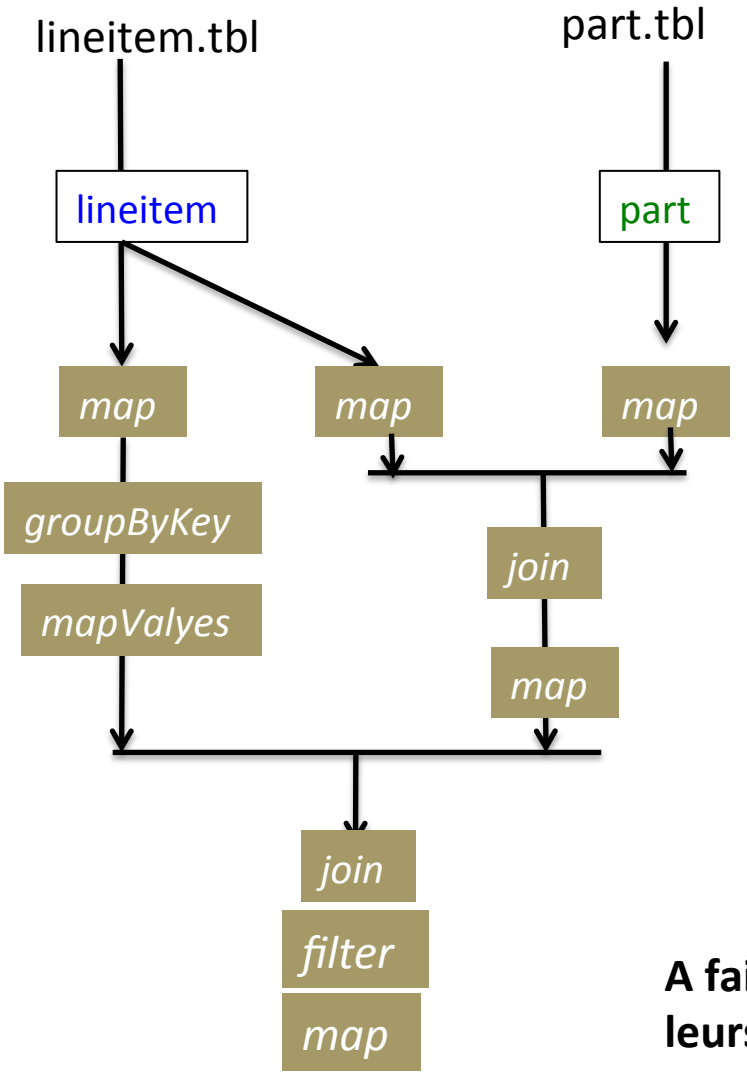
```
val outer = lineitem.map{case(partkey,quantity,ext)=>(partkey,(quantity,ext))}
            .join(part.map(x=>(x,null)))
            .map{case(partkey,((quantity,ext),_))=>(partkey,(quantity,ext))}
```

Exercice

```
SELECT sum(l_extendedprice) / 7.0 AS avg_yearly
FROM inner, outer
WHERE outer.l_partkey = inner.l_partkey; AND outer.l_quantity < inner.t1;
```

```
val query = inner.join(outer)
        .filter{case(partkey,(t1,(quantity,extended)))=>quantity<t1}
        .map{case(partkey,(t1,(quantity,extended)))=>extended}

val res = query.sum/7
```

TPCH Q17




```
val inner = lineitem.
            map{case(partkey,quantity,_)=>
                (partkey,quantity)}
            .groupByKey
.mapValues(x=>.2*myAvg(x))

val outer = lineitem.map{case(partkey,quantity,ext)=>
                         (partkey,(quantity,ext))}
            .join(part.map(x=>(x,null)))
            .map{case(partkey,((quantity,ext),_))=>
                         (partkey,(quantity,ext))}

val query = (inner.join(outer)
            .filter{case(partkey,(t1,(quantity,extended)))
                =>quantity<t1}
            .map{case(partkey,(t1,(quantity,extended)))
                =>extended}.sum)/7
```

A faire : traduire d'autres requêtes TPCH et observer leurs plans d'exécution

Bilan Algèbre RDD

- Algèbre riche
- Optimisation limitée à la notion de stages
  - Pas de notion de plan logique
  - Code utilisateur difficile à optimiser contrairement aux langages déclaratifs comme SQL
  - Absence de modèle de coût
  - Faible performances des agrégations