CentraleSupélec

Big Data : Informatique pour les données et calculs massifs

6 – Algorithmique Map-Reduce

Stéphane Vialle

CentraleSupélec université PARIS-SACLAY — ÉCOLE DOCTORALE Sciences et technologies de l'information et de la communication (STIC) — RISEGrid — Grand Est ALSACE CHAMPAGNE-ARDENNE LORRAINE

Stephane.Vialle@centralesupelec.fr
http://www.metz.supelec.fr/~vialle

CentraleSupélec

Démarche algorithmique

- **Définir des tâches *Mappers* et *Reducers***
  → Bâtir la base de la solution *Map-Reduce*
- **Identifier des routines *Combiner***
  → Eviter des E/S temporaires sur disque *(Hadoop)*
  → Réduire le trafic sur le réseau entre *Mappers* et *Reducers*
  → Eviter de saturer la mémoire des *Reducers*
- **Identifier le nombre de *Reducers* optimal**
  → Répartir la charge au maximum entre plusieurs machines
- **Concevoir des *Partitioners* pour optimiser la solution**
  → Améliorer la répartition de charge des *Reducers*
  (si on connait la variété des clés et la distribution des paires)
- **Ecrire des fonctions *keyComparator()* et *groupComparator()***
  → Pour profiter des fonctions de tri intégrées au *Shuffle & Sort*

⇒ « Patterns » ou « Patrons » de conception types

Familles de patrons Map-Reduce

Patrons de récapitulation (*Summarization)*

Patrons de filtrage

Patrons de restructuration des données

Patrons de tris

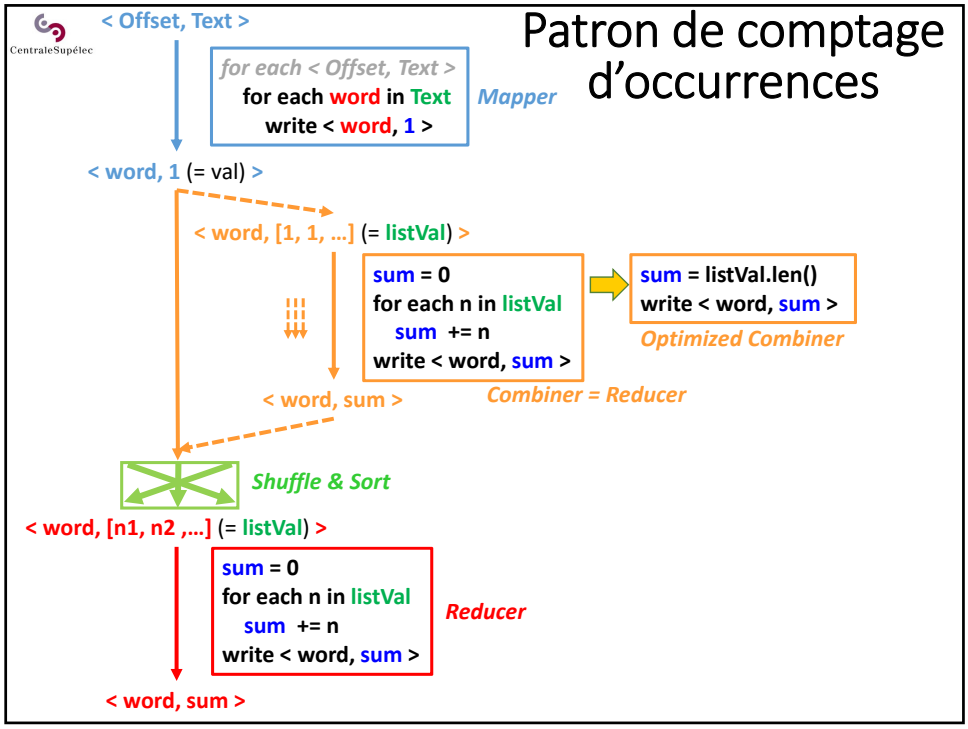
Patrons de jointure

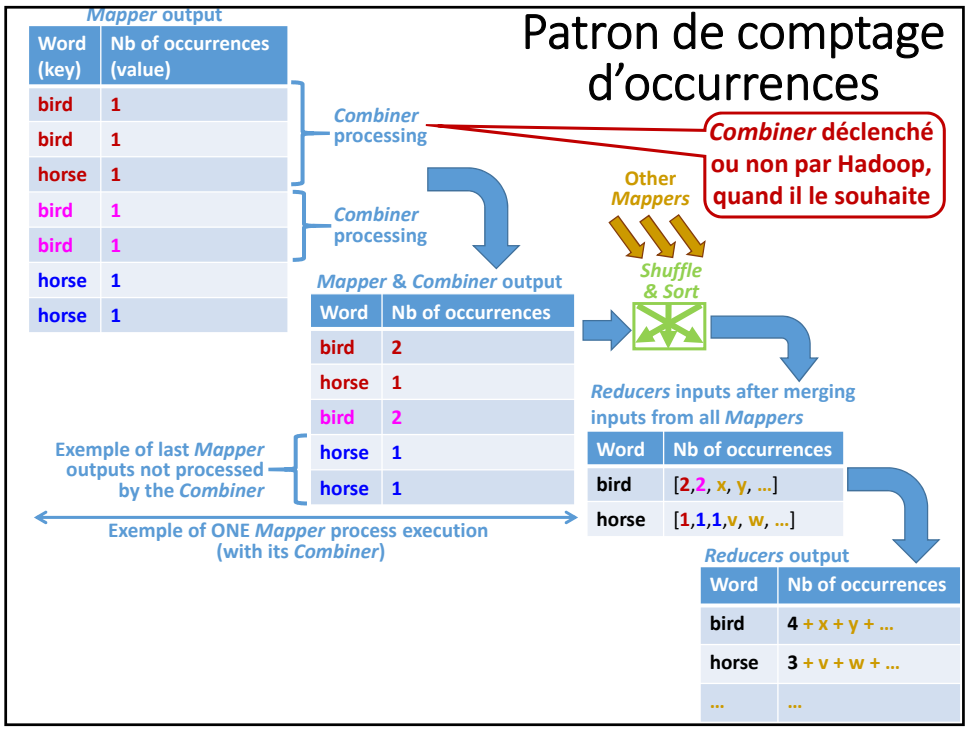
Patrons de récapitulation

1 – Patron de comptage d'occurrences de termes

2 – Patron de calcul d'une moyenne

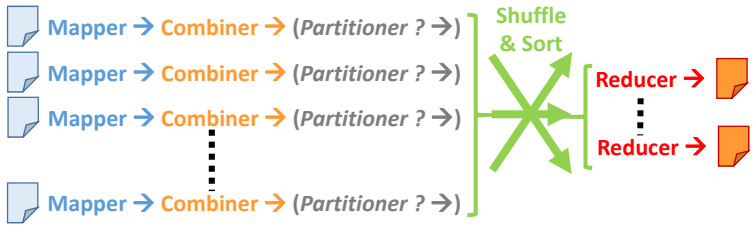
3 – Patron de réalisation d'un index inversé


CentraleSupélec
Patron de comptage d'occurrences
< Offset, Text >
for each < Offset, Text >
for each word in Text
write < word, 1 >
Mapper
< word, 1 (= val) >
< word, [1, 1, ...] (= listVal) >
sum = 0
for each n in listVal
sum += n
write < word, sum >
sum = listVal.len()
write < word, sum >
Optimized Combiner
Combiner = Reducer
< word, sum >
Shuffle & Sort
< word, [n1, n2 ,...] (= listVal) >
sum = 0
for each n in listVal
sum += n
write < word, sum >
Reducer
< word, sum >


Patron de comptage d'occurrences
Mapper output
Word (key)
Nb of occurrences (value)
bird 1
bird 1
horse 1
bird 1
bird 1
horse 1
horse 1
Combiner processing
Combiner processing
Combiner déclenché ou non par Hadoop, quand il le souhaite
Other Mappers
Shuffle & Sort
Mapper & Combiner output
Word
Nb of occurrences
bird 2
horse 1
bird 2
horse 1
horse 1
Exemple of last Mapper outputs not processed by the Combiner
Exemple of ONE Mapper process execution (with its Combiner)
Reducers inputs after merging inputs from all Mappers
Word
Nb of occurrences
bird [2,2, x, y, ...]
horse [1,1,1,v, w, ...]
Reducers output
Word
Nb of occurrences
bird 4 + x + y + ...
horse 3 + v + w + ...
... ...

Patron de comptage d'occurrences

Déploiement :



- Un *Mapper* par bloc de fichier d'entrée (beaucoup de *Mappers*)
- *Un Combiner* associé à chaque *Mapper* pour réduire le trafic dans le *Shuffle & Sort*
- Quelques *Reducers*, fonctions du nombre de termes attendus

- *Un Partitioner* pour améliorer l'équilibrage de charge entre *Reducers*
  … ssi connaissances pertinentes sur les résultats attendus!

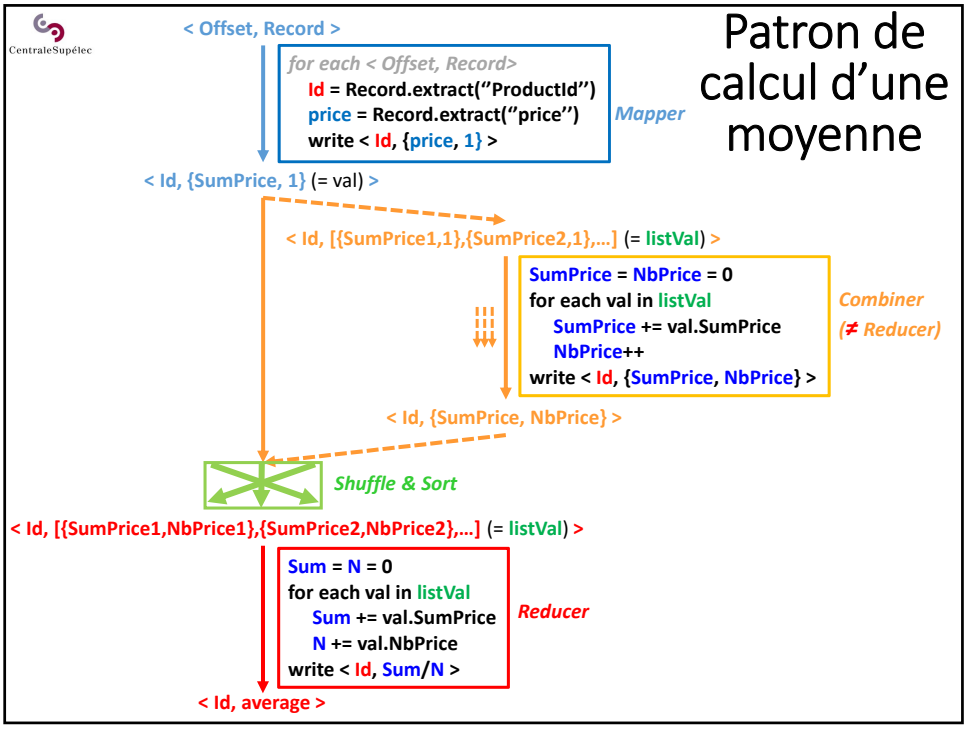
Patrons de récapitulation

1 – Patron de comptage d'occurrences de termes

2 – Patron de calcul d'une moyenne

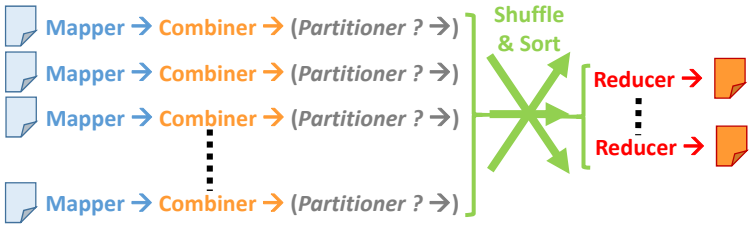
3 – Patron de réalisation d'un index inversé

Patron de calcul d'une moyenne



Patron de comptage d'occurrences

Déploiement :



Même déploiement que pour le compteur d'occurences

- Un *Mapper* par bloc de fichier d'entrée (beaucoup de *Mappers*)
- *Un Combiner* associé à chaque *Mapper* pour réduire le trafic dans le *Shuffle & Sort*
- Quelques *Reducers*, fonctions du nombre de produits attendus
- *Un Partitioner* pour améliorer l'équilibrage de charge entre *Reducers* … ssi connaissances pertinentes sur les résultats attendus!

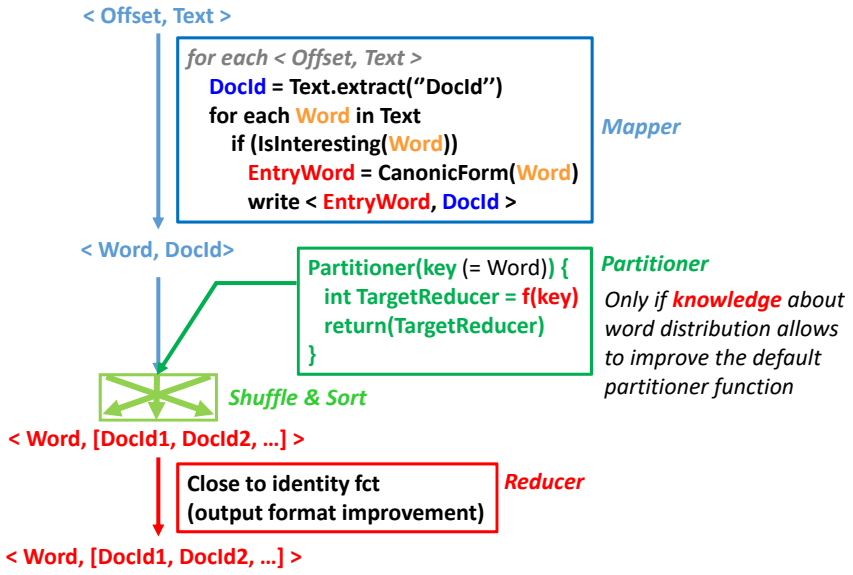
Patrons de récapitulation

1 – Patron de comptage d'occurrences de termes

2 – Patron de calcul d'une moyenne

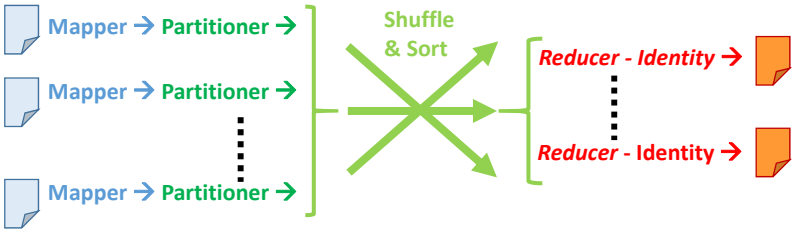
3 – Patron de réalisation d'un index inversé

Patron d'index inversé



Patron d'index inversé

Déploiement :



- *Combiner* inutile : non implanté
- ***Partitioner*** **pour améliorer l'équilibrage de charge entre** ***Reducers***

  - informations sur la distribution des mots ?
  - en fonction de la langue des documents ?
  → calcul d'une fonction de hashage optimale

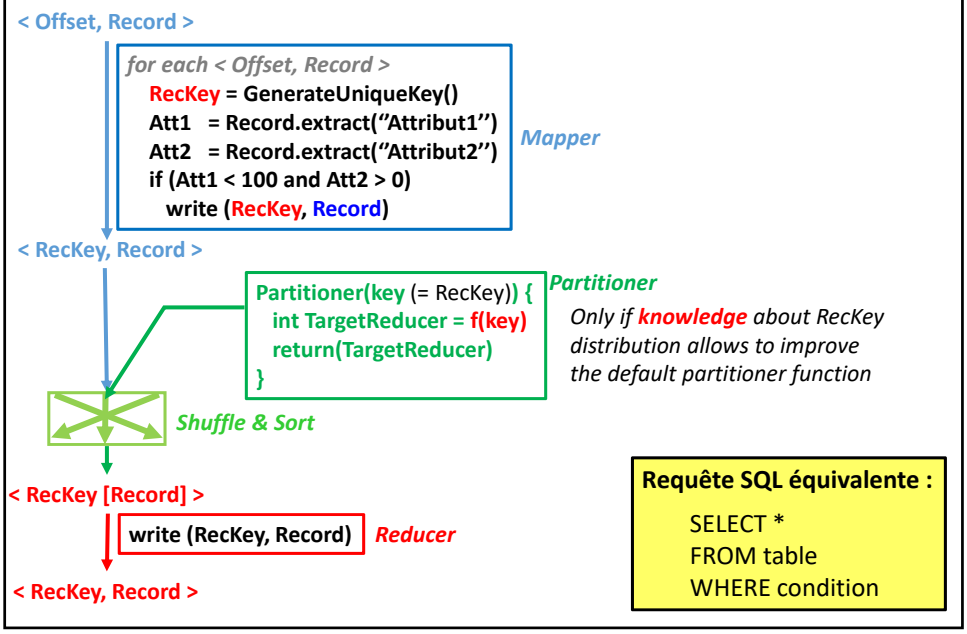
Patrons de Filtrage

1 – Patron de filtrage accepter/rejeter

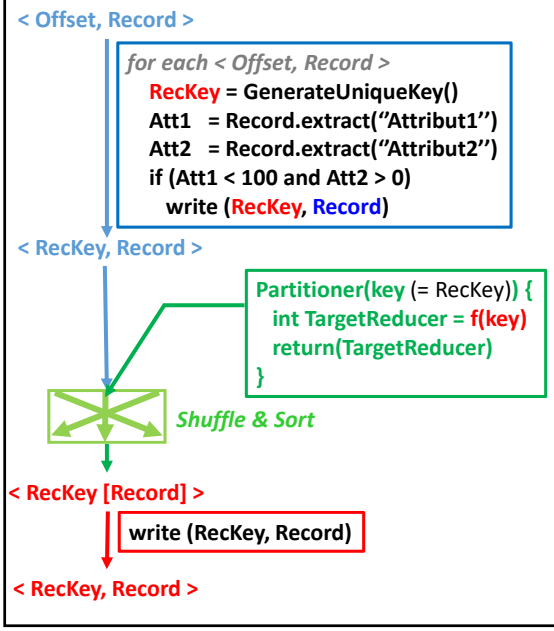
2 – Patron de filtrage *Top Ten*

3 – Patron de filtrage dé-duplicateur

Patron de filtrage accepter/rejeter



Patron de filtrage accepter/rejeter



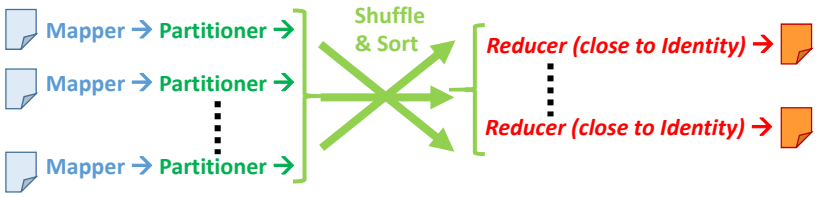
Besoin de générer une clé unique pour chaque enregistrement

Fournir le *Partitioner* qui répartit équitablement les clés sur beaucoup de *Reducers*

→ Générateur de clé et *Partitioner* conçus en cohérence

Patron de filtrage accepter/rejeter

Déploiement :



- *Combiner* inutile (pas de réduction de trafic) : non implanté
- ***Partitioner*** **pour améliorer l'équilibrage de charge entre** ***Reducers***
  → clés générées et ***Partitioner*** **conçus ensembles**

Patrons de Filtrage

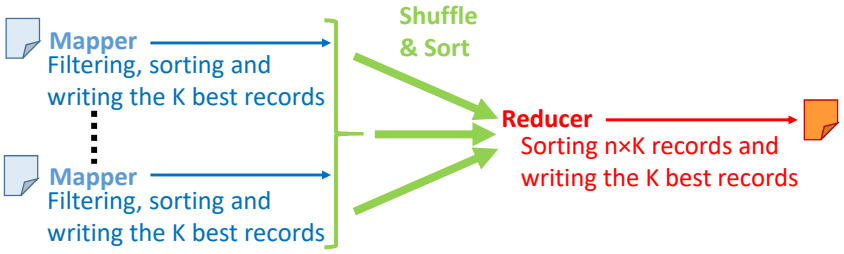
1 – Patron de filtrage accepter/rejeter

2 – Patron de filtrage *Top Ten*

3 – Patron de filtrage dé-duplicateur

Patron de filtrage *Top Ten / Top K*

Principe et déploiement :



- Chaque *Mapper* filtre des enregistrements, tri ceux acceptés, et écrit les paires clé-valeur des K meilleurs locaux
- UN *Reducer* reçoit les n × K meilleurs, les interclasse, et écrit les paires des K meilleurs globaux
- Pas d'optimisations :
  pas de *Combiner* possible (sorties de chaque *Mapper* déjà réduites),
  pas de *Partitioner* possible (un seul *Reducer* cible)

Patron de filtrage *Top Ten / Top K*

< Offset, Record >

```
List<Record> ListRec
for each < Offset, Record >
   Att1   = Record.extract("Attribut1")
   if (Att1 < 100)
     ListRec.add(Record)
Sort(ListRec)
for (i = 0; i < K  and  i < ListRec.size; i++)
   write (null-key, ListRec[i])
```

Mapper
(not the most efficient code)

< null-key, Record >

Shuffle & Sort

< null-key, [Rec0, Rec1, Rec2, …] (= ListVal) >

```
Sort(ListVal)
for (i = 0; i < K  and  i < ListVal.size; i++)
   write (i, ListVal[i])
```

Reducer

< rank, Record >

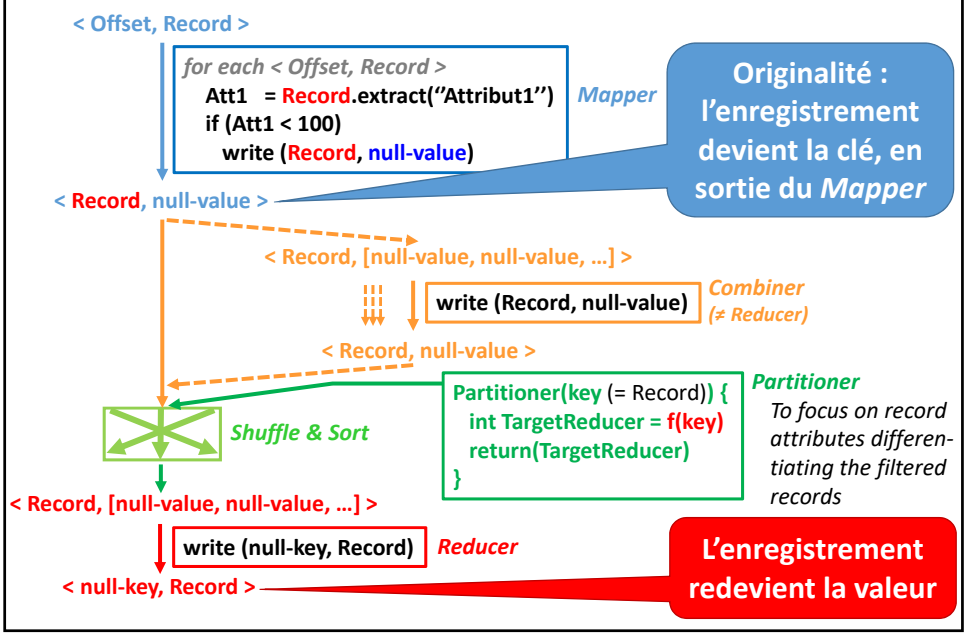
Patrons de Filtrage

1 – Patron de filtrage accepter/rejeter

2 – Patron de filtrage *Top Ten*

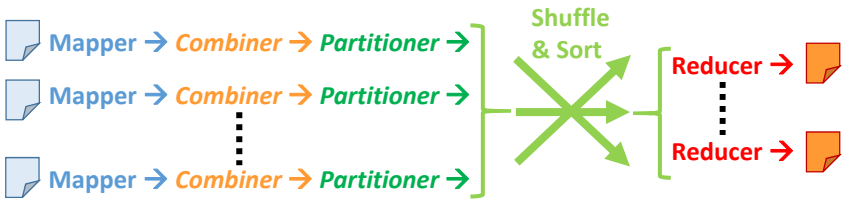
3 – Patron de filtrage dé-duplicateur

Patron de filtrage dé-duplicateur



Patron de filtrage dé-duplicateur

Déploiement :



Optimisations :

- **Nombre de *Reducers*** important pour répartir sur différentes machines la charge d'écriture des enregistrements
- **Un *Combiner*** réduira le trafic si et seulement si il y a beaucoup de doublons en sortie de chaque *Mapper*
- **Un *Partitioner*** peut améliorer l'équilibrage de charge en se focalisant sur les attributs différentiant des enregistrements retenus

Patrons de restructuration des données

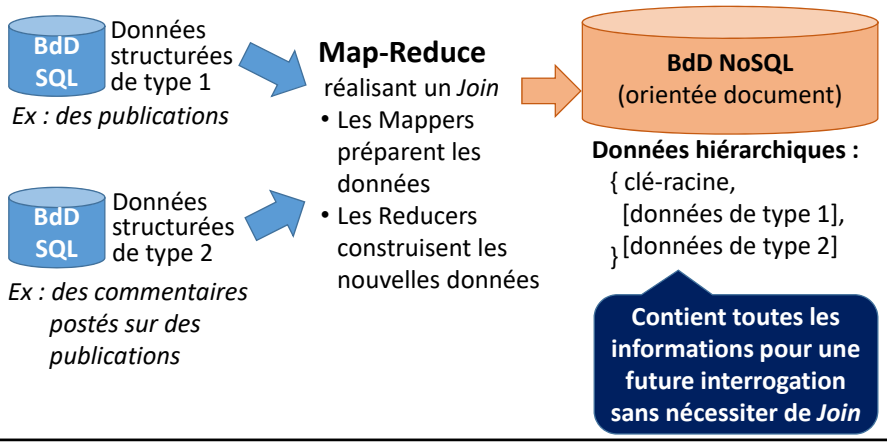
1 – Patron de transformation *Structured-to-Hierarchical*

2 – Patron de partitionnement et *binning*

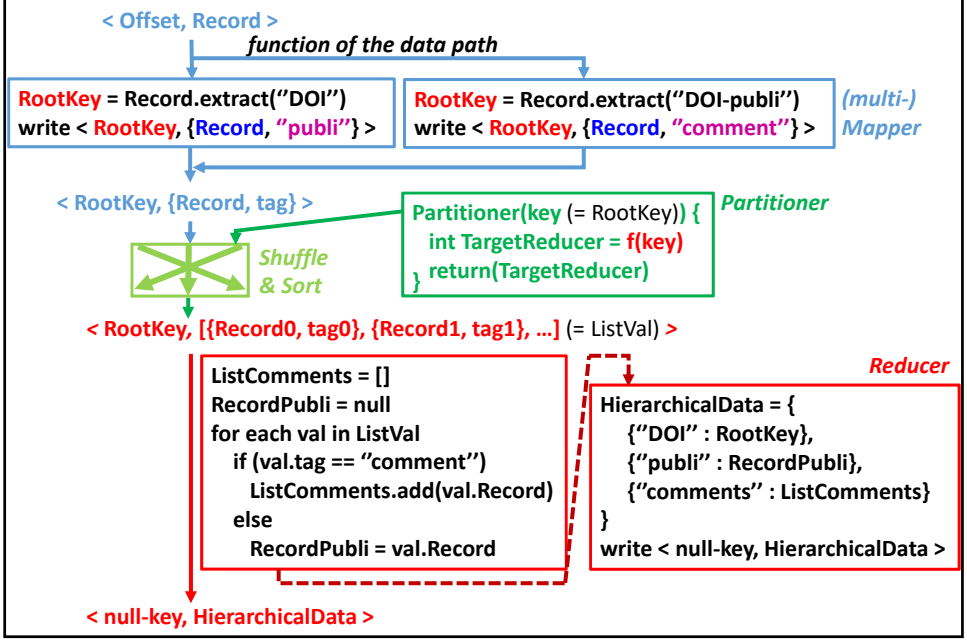
Patron *Structured-to-Hierarchical*

Objectif et principes

Use case : assembler des données de tables de BdD relationnelles dans une BdD NoSQL (faire un *Join* lors du remplissage la BdD NoSQL)

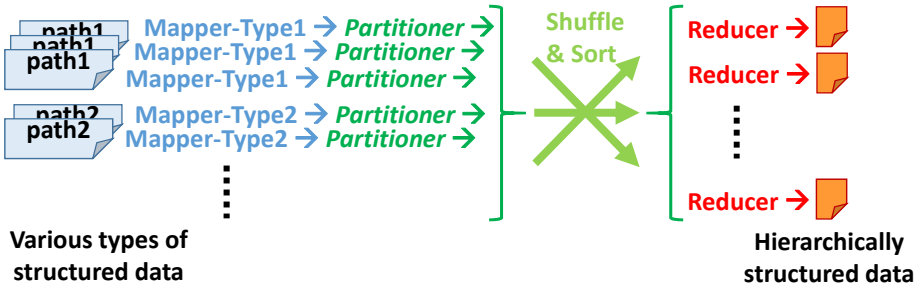


Patron *Structured-to-Hierarchical*



Patron *Structured-to-Hierarchical*

Déploiement :



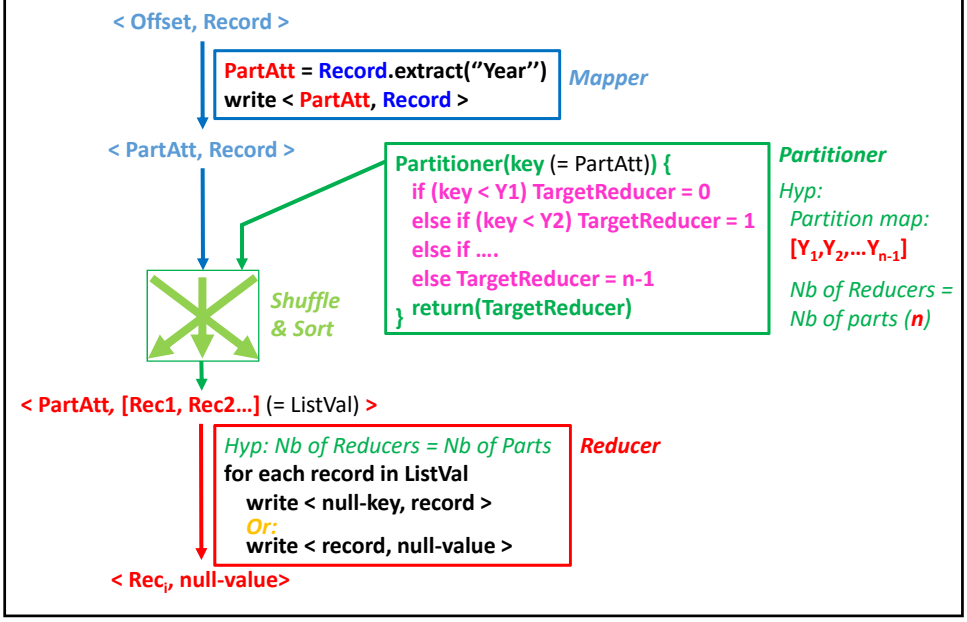
- Bcp de données à mélanger et pas de pertes d'info
  → bcp de trafic, et bcp de travail dans les *Reducers*
  → **déployer bcp de *Reducers***
- **Pas de *Combiner*** utile (rien à regrouper pour réduire le trafic)
- **Un *Partitioner* peut optimiser** la répartition des clés sur les *Reducers,* et l'équilibrage de charge des *Reducers.*

Patrons de restructuration des données

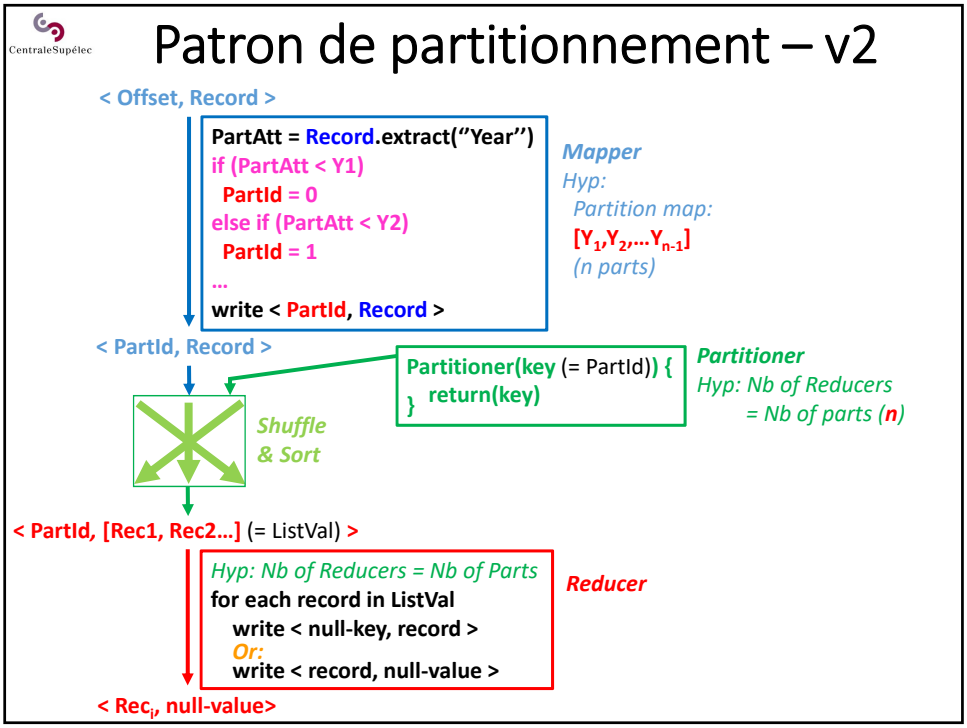
1 – Patron de transformation *Structured-to-Hierarchical*

2 – Patron de partitionnement et *binning*

Patron de partitionnement – v1

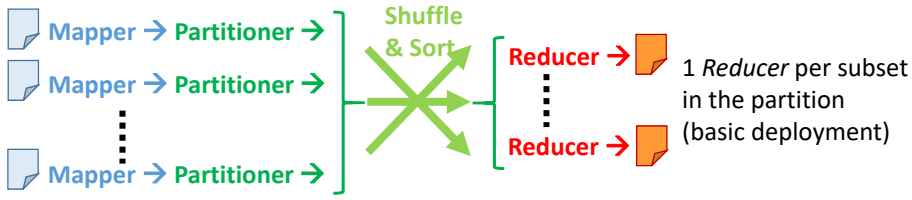


Patron de partitionnement – v2



Patron de partitionnement

Déploiement :



- Pas de *Combiner* utile (rien à regrouper pour réduire le trafic)
- Toute donnée lu doit être ré-écrite → bcp d'écritures sur disque
- Démarche la plus simple :
  - Un *Reducer* par partition, pour sauver chaque partition dans un fichier différent
  - Un *Partitioner* qui envoie la partition *i* sur le *Reducer i*

Patron de partitionnement

Variante du *Patron de Binning* (ou de *mise en récipients*) :

Une donnée peut aller dans plusieurs partitions

Ex : les enregistrements à la limite de deux sous-ensembles sont rangés dans les deux !

Adaptation du *Mapper* de la 2ème version du patron de partitionnement :

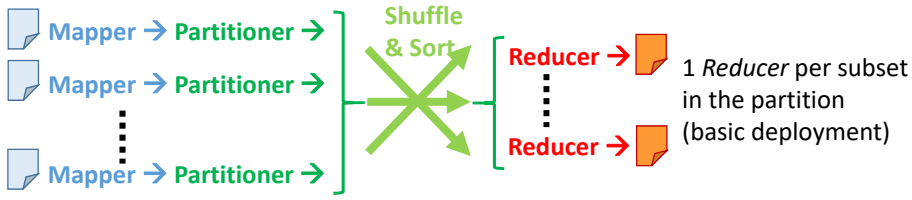
```
PartAtt = Record.extract("Attribute1")
if (PartAtt < V0)
  write < 0, Record >
else if (PartAtt == V0)
  write < 0, Record >
  write < 1, Record >
else if (PartAtt < V1)
  write < 1, Record >
...
```

Mapper

Les Mappers peuvent générer plusieurs paires de sorties pour un même enregistrement d'entrée

Patron de partitionnement

Déploiement :



- Pas de *Combiner* utile (rien à regrouper pour réduire le trafic)
- Toute donnée lu doit être ré-écrite → bcp d'écritures sur disque
- Autre démarche :
  - plus de *Reducers* que de partitions, et …
  - …le *Partitioner* répartit les grosses partitions sur plusieurs *Reducer*

Patrons de tri de données

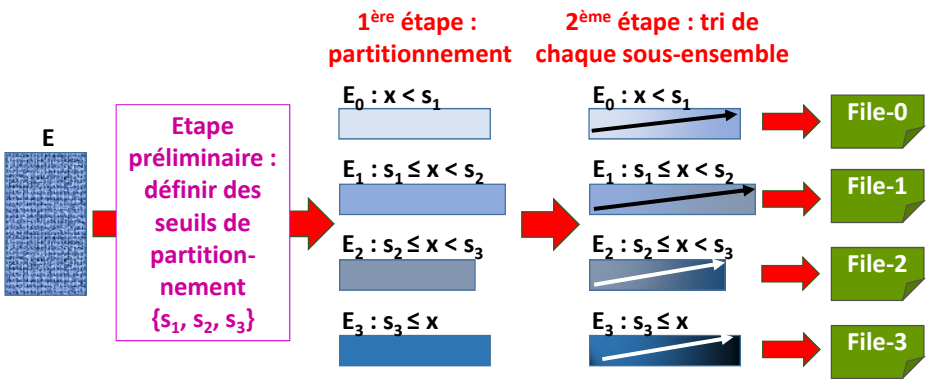
1 - Tri total avec tri final dans les *Reducers*

2 - Reconfiguration du *Shuffle & Sort* pour optimisation

3 - Tri optimisé par le *Shuffle & Sort* sur clés simples et continues

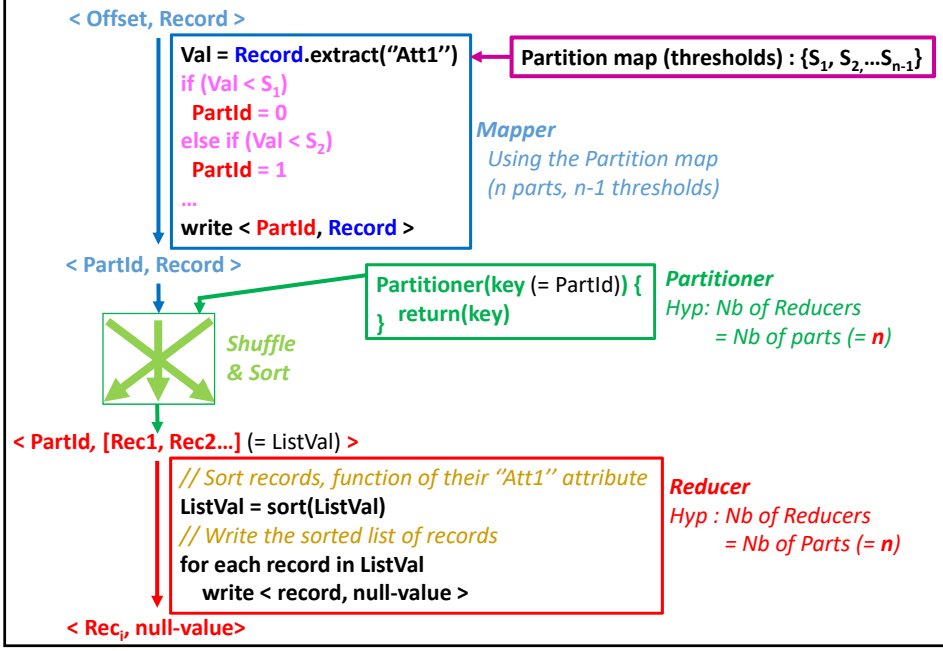
4 - Tri optimisé par le *Shuffle & Sort* sur clés composites

Tri total avec tri final dans les *Reducers*



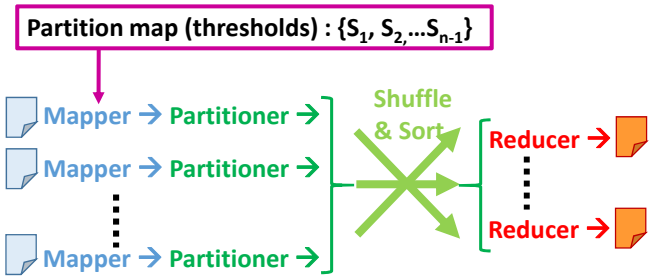
- Extension du patron de partitionnement : ajoute un tri local dans chaque *Reducer*
- Besoin de connaissance a priori sur les données pour définir la carte de partitionnement équilibrée (les seuils)

Tri total avec tri final dans les *Reducers*



Tri total avec tri final dans les *Reducers*

Déploiement et bilan



- Suit le principe algorithmique de base du tri total (1+2 étapes)
- Nbr de *Reducers* = Nbr de sous-ensembles de la partition
- **Ne tire pas partie des tris systématiques et implicites sur les clés à l'entrée des *Reducers*…**

Patrons de tri de données

1 - Tri total avec tri final dans les *Reducers*

2 - Reconfiguration du *Shuffle & Sort* pour optimisation

3 - Tri optimisé par le *Shuffle & Sort* sur clés simples et continues

4 - Tri optimisé par le *Shuffle & Sort* sur clés composites

Reconfiguration du *Shuffle & Sort*

Concept de clé composite

Clé simple : < k, v >
k : une seule valeur (ex : 10, ''toto''…)

Clé composite : ex : < {k1, k2, k3}, v >
clé composite de 3 attributs

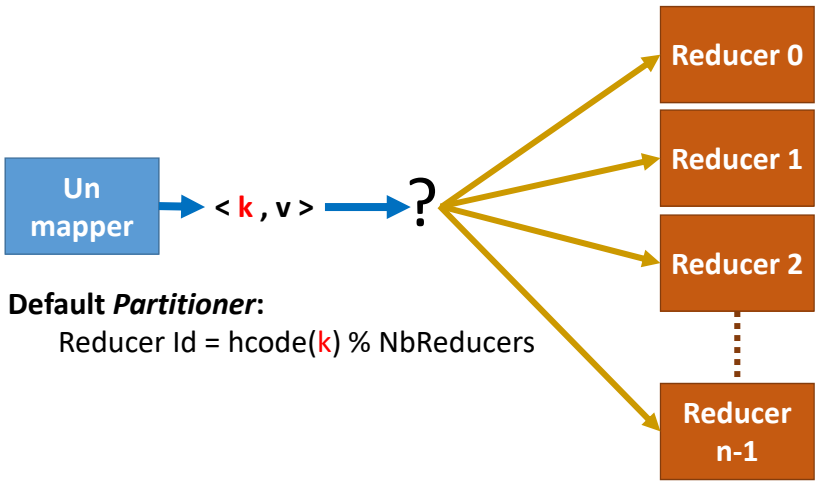
k1 : une valeur (ex : 10, ''toto''…)
k2 : une autre valeur (ex : 'A')
k3 : une autre valeur (ex : ''hf5676klkhjujgh798'')

En général, les attributs d'une clé composite correspondent à des critère de tris emboités des enregistrements

Ex : trier selon k1 en ordre croissant, puis les exæquos selon k2 en ordre décroissant, puis …

Reconfiguration du *Shuffle & Sort*

1er niveau de contrôle/reconfiguration : redéfinition du *Partitioner*



Default *Partitioner*:
Reducer Id = hcode(k) % NbReducers

Reconfiguration du *Shuffle & Sort*

1er niveau de reconfiguration : redéfinition du *Partitioner*

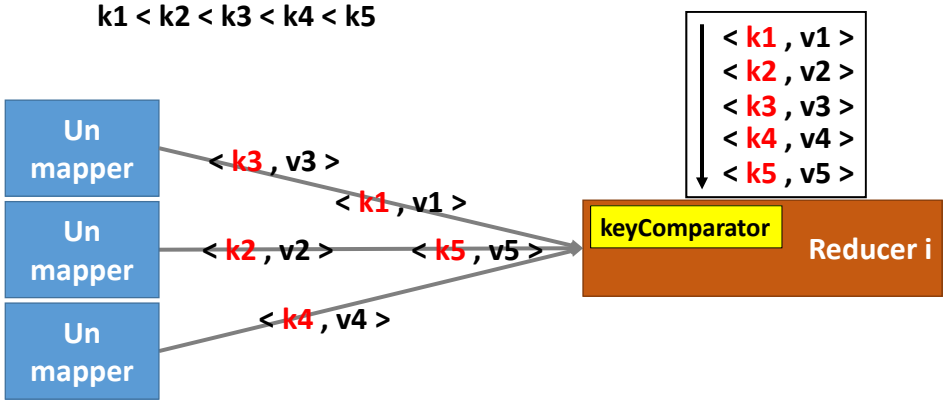
```
public static class MyPartitioner
  extends Partitioner<keyClass,valueClass> {
  ...
  public int getPartition(keyClass key,
                          valueClass val,
                          int numPartitions) {
    ...
    // Must return a Reducer index [0 ... NbReducers-1]
    int index = ... // Specific code
    return(index)
  }
}
```

```
//Set the new partitioner to the Map-Reduce job
job.setPartitionerClass{MyPartitioner.class}
// Run the Map-Reduce job
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

Reconfiguration du *Shuffle & Sort*

2ème niveau de reconfiguration : redéfinition du *keyComparator*
(ou sortComparator)

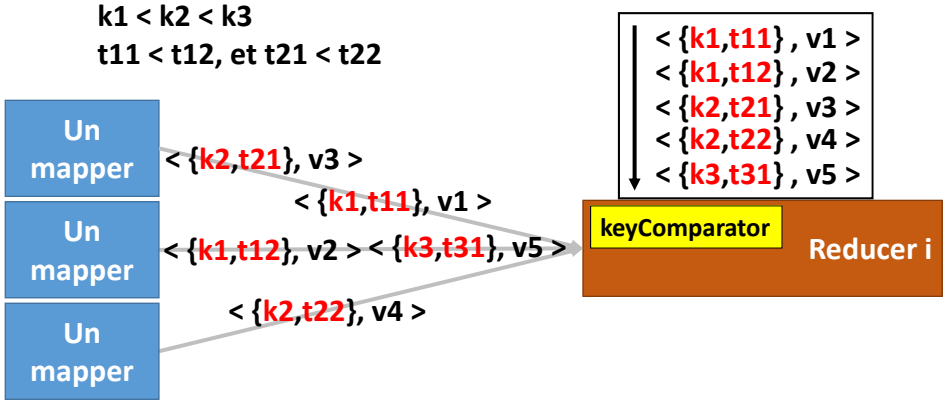
Hyp : clés simples
k1 < k2 < k3 < k4 < k5



Reconfiguration du *Shuffle & Sort*

2ème niveau de reconfiguration : redéfinition du *keyComparator*
(ou sortComparator)

Hyp : clés composites
k1 < k2 < k3
t11 < t12, et t21 < t22



keyComparator sur les 2 attributs de la clé composite

Reconfiguration du *Shuffle & Sort*

2ème niveau de reconfiguration : redéfinition du *keyComparator*
(ou sortComparator)

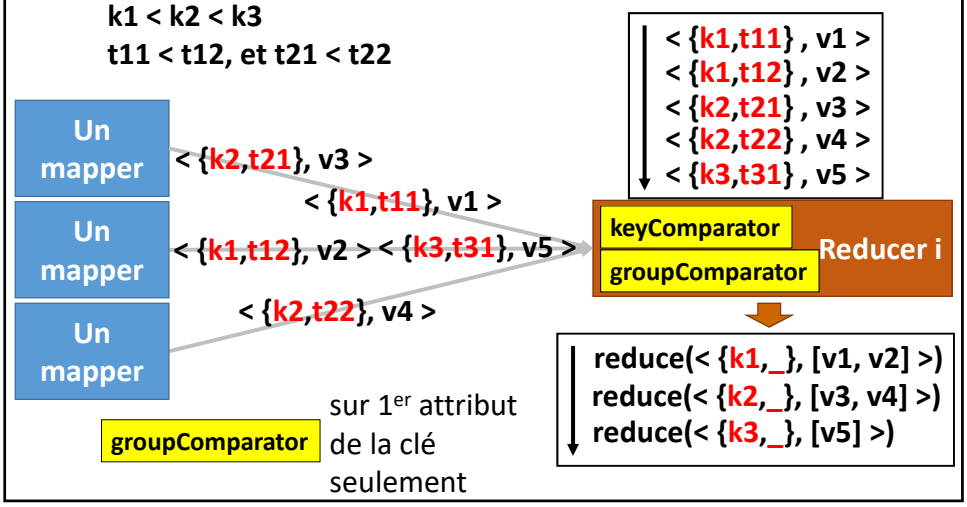
```
public static class MyKeyComparator
  extends WritableComparator {
  ...
  public int compare(WritableComparable w1,
                     WritableComparable w2) {
    ...
    // Must return -1/0/+1,
    // considering w1 < w2, or w1 = w2, or w1 > w2
    int ComparisonResult = ... // Specific code
    return(ComparisonResult)
  }
}
```

```
//Set the new keyComparator to the Map-Reduce job
job.setSortComparatorClass{MyKeyComparator.class}

// Run the Map-Reduce job
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

Reconfiguration du *Shuffle & Sort*

2ème niveau de reconfiguration : redéfinition du *groupComparator*

Hyp : clés composites
k1 < k2 < k3
t11 < t12, et t21 < t22



Reconfiguration du *Shuffle & Sort*

2ème niveau de reconfiguration : redéfinition du *groupComparator*

```
public static class MyGroupComparator
  extends WritableComparator {
  ...
  public int compare(WritableComparable w1,
                     WritableComparable w2) {
    ...
    // Must return -1/0/+1,
    // considering w1 < w2, or w1 = w2, or w1 > w2
    int ComparisonResult = ... // Specific code
    return(ComparisonResult)
  }
}
```

```
//Set the new groupComparator to the Map-Reduce job
job.setGroupingComparatorClass{MyGroupComparator.class}

// Run the Map-Reduce job
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

Patrons de tri de données

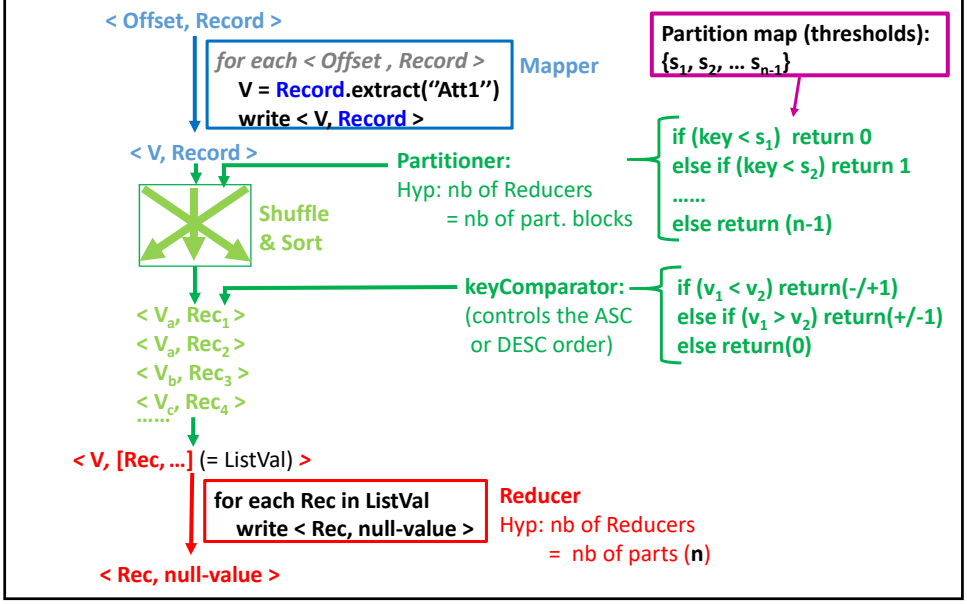
1 - Tri total avec tri final dans les *Reducers*

2 - Reconfiguration du *Shuffle & Sort* pour optimisation

3 - Tri optimisé par le *Shuffle & Sort* sur clés simples et continues

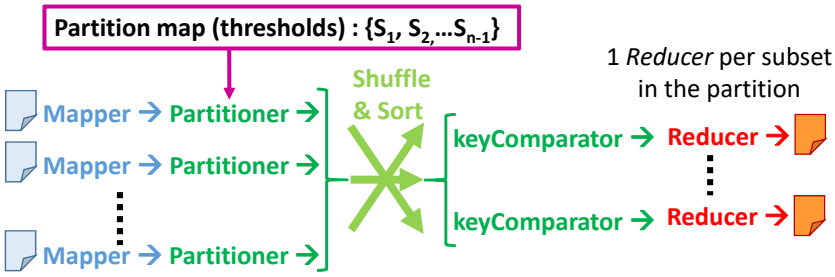
4 - Tri optimisé par le *Shuffle & Sort* sur clés composites

Tri optimisé par le *Shuffle & Sort* sur clés simples et continues



Tri optimisé par le *Shuffle & Sort* sur clés simples et continues

Déploiement et bilan



- Profite du tri des clés du *Shuffle & Sort*, et évite un tri final dans les *Reducers*
- Nbr de *Reducers* = Nbr de sous-ensembles de la partition
- Ne trie les enregistrements que selon UN SEUL critère

Patrons de tri de données

1 - Tri total avec tri final dans les *Reducers*

2 - Reconfiguration du *Shuffle & Sort* pour optimisation

3 - Tri optimisé par le *Shuffle & Sort* sur clés simples et continues

4 - Tri optimisé par le *Shuffle & Sort* sur clés composites

Tri optimisé par le *Shuffle & Sort* sur clés composites

Problématique / Objectifs

- **Trier selon deux critères (ou plus) emboités :**

  SELECT * FROM etudiant ORDER BY spécialité, moyenne DESC

- **Faire un traitement sur les sous-listes du dernier critère**

  Ex : calcul d'une valeur moyenne médiane ou maximale par spécialité

- **Profiter du tri implicite et systématique d'Hadoop sur les clés**

Démarche

- Construire une **clé composite** avec les deux critères :

  Ex : {spécialité, moyenne}

- Redéfinir les **3 niveaux de contrôles** du *Shuffle & Sort*

Tri optimisé par le *Shuffle & Sort* sur clés composites

Problème illustratif

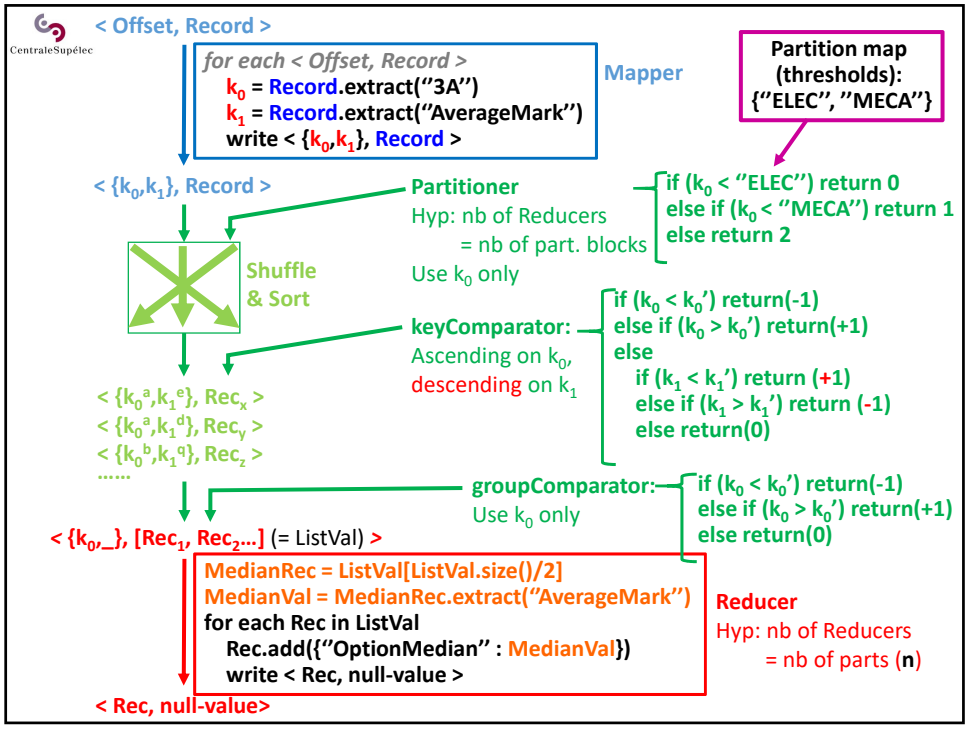
Trier des enregistrements d'étudiants selon 2 critères emboités :

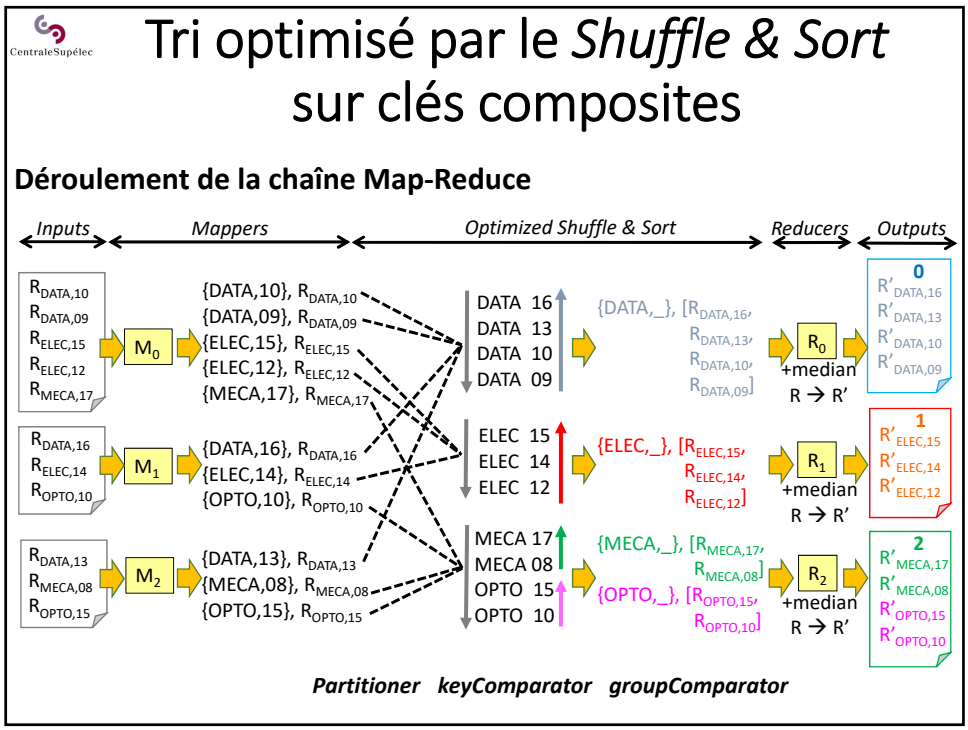
1 – par option de 3A en ordre lexicographique croissant

2 – par moyenne générale en ordre décroissant

Pour chaque option de 3A :

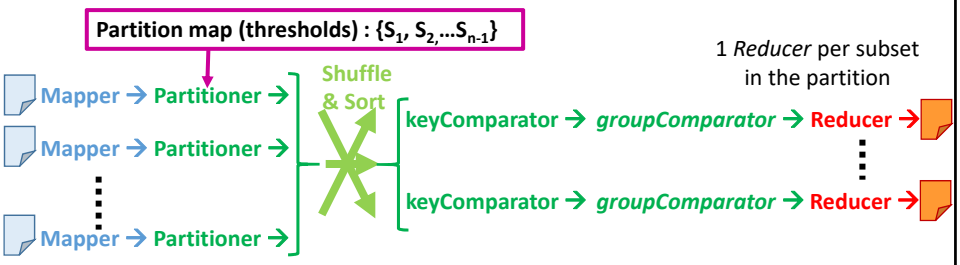
- calculer la médiane des moyennes générales
- enrichir chaque enregistrement avec cette valeur médiane
- écrire sur disque dans l'ordre les enregistrements enrichis


CentraleSupélec
< Offset, Record >
for each < Offset, Record >
k0 = Record.extract("3A")
k1 = Record.extract("AverageMark")
write < {k0,k1}, Record >
Mapper
Partition map (thresholds): {"ELEC", "MECA"}
< {k0,k1}, Record >
Partitioner
Hyp: nb of Reducers = nb of part. blocks
Use k0 only
if (k0 < "ELEC") return 0
else if (k0 < "MECA") return 1
else return 2
Shuffle & Sort
keyComparator:
Ascending on k0, descending on k1
if (k0 < k0') return(-1)
else if (k0 > k0') return(+1)
else
if (k1 < k1') return (+1)
else if (k1 > k1') return (-1)
else return(0)
< {k0^a,k1^e}, Recx >
< {k0^a,k1^d}, Recy >
< {k0^b,k1^q}, Recz >
.....
groupComparator:
Use k0 only
if (k0 < k0') return(-1)
else if (k0 > k0') return(+1)
else return(0)
< {k0,_}, [Rec1, Rec2...] (= ListVal) >
MedianRec = ListVal[ListVal.size()/2]
MedianVal = MedianRec.extract("AverageMark")
for each Rec in ListVal
Rec.add({"OptionMedian" : MedianVal})
write < Rec, null-value >
Reducer
Hyp: nb of Reducers = nb of parts (n)
< Rec, null-value>


CentraleSupélec
Tri optimisé par le Shuffle & Sort sur clés composites
Déroulement de la chaîne Map-Reduce
Inputs
Mappers
Optimized Shuffle & Sort
Reducers
Outputs
RDATA,10 RDATA,09 RELEC,15 RELEC,12 RMECA,17
M0
{DATA,10}, RDATA,10
{DATA,09}, RDATA,09
{ELEC,15}, RELEC,15
{ELEC,12}, RELEC,12
{MECA,17}, RMECA,17
RDATA,16 RELEC,14 ROPTO,10
M1
{DATA,16}, RDATA,16
{ELEC,14}, RELEC,14
{OPTO,10}, ROPTO,10
RDATA,13 RMECA,08 ROPTO,15
M2
{DATA,13}, RDATA,13
{MECA,08}, RMECA,08
{OPTO,15}, ROPTO,15
DATA 16
DATA 13
DATA 10
DATA 09
ELEC 15
ELEC 14
ELEC 12
MECA 17
MECA 08
OPTO 15
OPTO 10
{DATA,_}, [RDATA,16, RDATA,13, RDATA,10, RDATA,09]
{ELEC,_}, [RELEC,15, RELEC,14, RELEC,12]
{MECA,_}, [RMECA,17, RMECA,08]
{OPTO,_}, [ROPTO,15, ROPTO,10]
R0
R1
R2
+median
R → R'
0
R'DATA,16 R'DATA,13 R'DATA,10 R'DATA,09
1
R'ELEC,15 R'ELEC,14 R'ELEC,12
2
R'MECA,17 R'MECA,08 R'OPTO,15 R'OPTO,10
Partitioner keyComparator groupComparator

Tri optimisé par le *Shuffle & Sort* sur clés composites

Déploiement et bilan

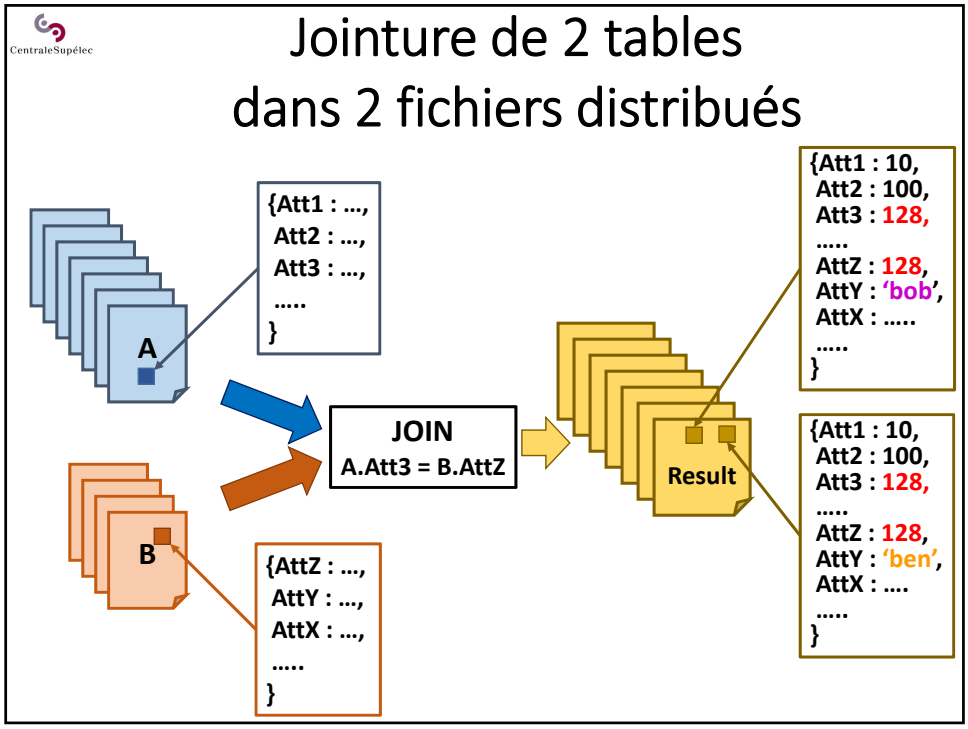


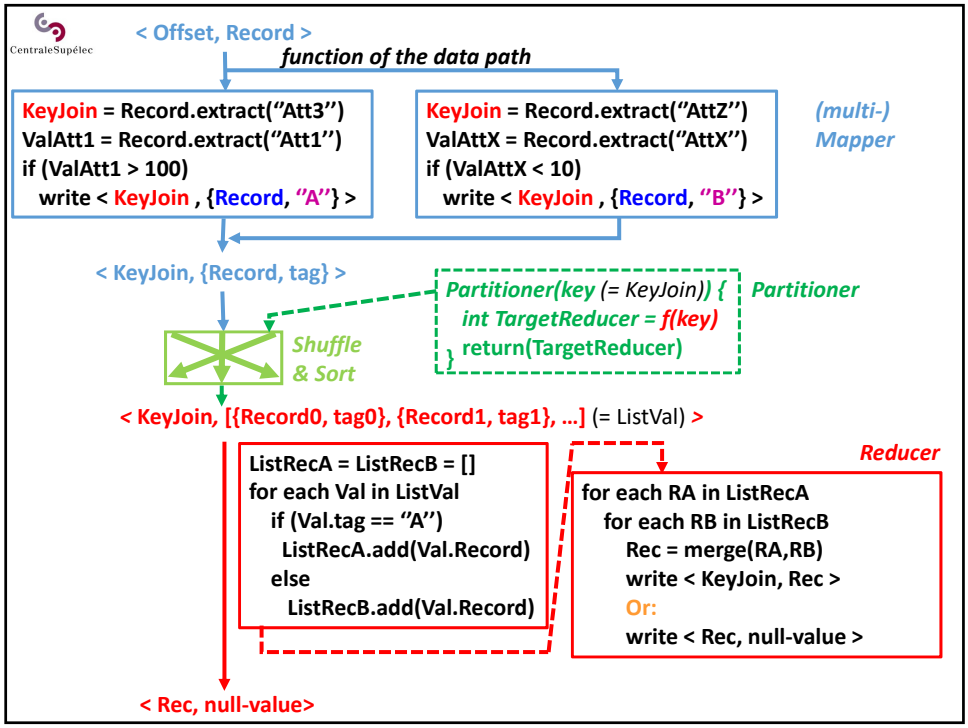
- Tri multi-critères (emboîtés)
- Exploite les tris de clés du *Shuffle & Sort*, **aucun tri final dans les *Reducers***
- Regroupement dans les *Reducers* et traitements par sous-listes
- Suppose l'existence d'un plan de partitionnement prédéfini…

Patrons de jointure

1 – Jointure de 2 tables dans 2 fichiers distribués

2 – Jointure de grande taille avec prétraitement


CentraleSupélec
Jointure de 2 tables
dans 2 fichiers distribués
A
{Att1 : ...,
Att2 : ...,
Att3 : ...,
.....
}
B
{AttZ : ...,
AttY : ...,
AttX : ...,
.....
}
JOIN
A.Att3 = B.AttZ
Result
{Att1 : 10,
Att2 : 100,
Att3 : 128,
.....
AttZ : 128,
AttY : 'bob',
AttX : .....
.....
}
{Att1 : 10,
Att2 : 100,
Att3 : 128,
.....
AttZ : 128,
AttY : 'ben',
AttX : ....
.....
}


CentraleSupélec
< Offset, Record >
function of the data path
KeyJoin = Record.extract("Att3")
ValAtt1 = Record.extract("Att1")
if (ValAtt1 > 100)
write < KeyJoin , {Record, "A"} >
KeyJoin = Record.extract("AttZ")
ValAttX = Record.extract("AttX")
if (ValAttX < 10)
write < KeyJoin , {Record, "B"} >
(multi-)
Mapper
< KeyJoin, {Record, tag} >
Partitioner(key (= KeyJoin)) {
int TargetReducer = f(key)
return(TargetReducer)
}
Partitioner
Shuffle
& Sort
< KeyJoin, [{Record0, tag0}, {Record1, tag1}, ...] (= ListVal) >
ListRecA = ListRecB = []
for each Val in ListVal
if (Val.tag == "A")
ListRecA.add(Val.Record)
else
ListRecB.add(Val.Record)
Reducer
for each RA in ListRecA
for each RB in ListRecB
Rec = merge(RA,RB)
write < KeyJoin, Rec >
Or:
write < Rec, null-value >
< Rec, null-value>

Jointure de 2 tables dans 2 fichiers distribués

1 - Paire {*clé, liste de valeurs*} reçue par un *Reducer* :

< KeyJoin = ''Dupont'', [{R_1,''A''}, {R_2,''A''}, {R'_1,''B''}, {R'_2,''B''}, {R_3,''A''}] >

2 - Actions du *Reducer* :

Re-construction de deux listes d'enregistrements :

→ ListRecA = (R_1, R_2, R_3)
ListRecB = (R'_1, R'_2)

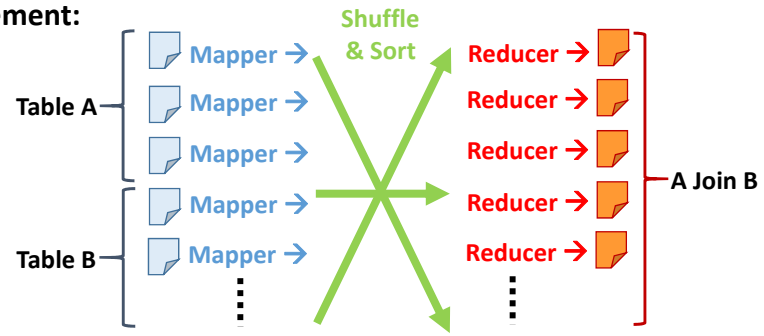
Génération de paires *clé – valeur de jointure*

→ < ''Dupont'', R_1 U R'_1 >
< ''Dupont'', R_1 U R'_2 >
< ''Dupont'', R_2 U R'_1 >
< ''Dupont'', R_2 U R'_2 >
< ''Dupont'', R_3 U R'_1 >
< ''Dupont'', R_3 U R'_2 >

ou bien : → < R_1 U R'_1, null-value >
< R_1 U R'_2, null-value >
< R_2 U R'_1, null-value >
< R_2 U R'_2, null-value >
< R_3 U R'_1, null-value >
< R_3 U R'_2, null-value >

Jointure de 2 tables dans 2 fichiers distribués

Déploiement:



Prévoir beaucoup de *Reducers*, car le résultat du *Join* peut être gros, et entraîner :

- beaucoup de traitements dans les Reducers
- beaucoup d'E/S sur diques!

CentraleSupélec

Patrons de jointure

1 – Jointure de 2 tables dans 2 fichiers distribués

2 – Jointure de grande taille avec prétraitement

CentraleSupélec

Jointure de grande taille avec prétraitement

Principe

1. Identifier un format de donnée structurée à partir duquel il serait simple et rapide de faire un *A JOIN B*

   Supposer que les attributs de l'équi-jointure sont connus et fixés

2. Implanter le pré-traitement (en *Map-Reduce*) pour obtenir de nouvelles versions des documents structurés/tables *A* et *B*
   → réaliser un *« co-partitionnement »…*

3. Implanter l'opération *JOIN* (en *Map-Reduce*) sur les nouvelles versions des tables *A* et *B*

Exercice

Outils d'aide au partitionnement

1 – *TotalOrderPartitioner*

2 – Configuration d'un partitionneur par échantillonnage

3 – Echantillonneurs prédéfinis

Classe *TotalOrderPartitioner*

La classe Java d'Hadoop *TotalOrderPartitioner* implémente un *Partitioner* qui exploite une carte de partitionnement prédéfinie

```
// Create the 'Map-Reduce job'
Job MRJob = new Job(getConf());
// Configure the 'Map-Reduce job'
// - Set the Mapper and Reducer to use
MRJob.setMapperClass(MyMapper.class)
MRJob.setReducerClass(Myreducer.class)
// - Set the number of Reducers
MRJob.setNumReduceTasks(REDUCE_TASKS);
// - Set the Partitioner to use: 'TotalOrderPartitioner'
MRJob.setPartitionerClass(TotalOrderPartitioner.class);
// - Set the partition map used by the 'TotalOrderPartitioner' (set partition file path)
TotalOrderPartitioner.setPartitionFile(MRJob.getConfiguration(), partitionPath);
// Run the Map-Reduce job, using the TotalOrderPartitioner
System.exit(MRJob.waitForCompletion(true) ? 0 : 1);
```

→ Il faut disposer d'une carte de partitionnement prédéfinie et menant à un bon équilibrage de charge !!

Outils d'aide au partitionnement

1 – *TotalOrderPartitioner*

2 – Configuration d'un partitionneur par échantillonnage

3 – Echantillonneurs prédéfinis

Configuration d'un partitionneur par échantillonnage

Principe

Un **partitionnement équilibré** évite qu'UN REDUCER reçoive et traite beaucoup plus de valeurs que les autres :

- serait couteux en temps de traitement
- risquerait de saturer la mémoire du *Reducer*

Mais sans connaissance a priori sur les données ni expérience sur des données similaires…on ne sait pas établir une carte de partitionnement équilibrée !

→ **Il faut analyser le jeu de données**
Long ! Et demande beaucoup de moyens !

→ **Analyser un échantillon du jeu de données**

Configuration d'un partitionneur par échantillonnage

Principe

→ **Analyser un échantillon du jeu de données**

- Définir une politique d'échantillonnage
- Implanter un échantillonneur qui accède aux données sur HDFS

Hadoop fournit des outils d'échantillonnage de données et de construction d'une carte de partitionnement équilibrée

Exemples :

- Echantillonnage aléatoire
- Echantillonnage à intervalles réguliers
- Collecte des *n* premières valeurs

Configuration d'un partitionneur par échantillonnage

Principe

→ **Analyser un échantillon du jeu de données**

- Définir une politique d'échantillonnage
- Implanter un échantillonneur qui accède aux données sur HDFS

Hadoop fournit des outils d'échantillonnage de données et de construction d'une carte de partitionnement équilibrée

+ simples à utiliser

- Tournent dans l'appli cliente sur une seule machine (la solution sort du paradigme Map-Reduce)

CentraleSupélec

Outils d'aide au partitionnement

1 – *TotalOrderPartitioner*

2 – Configuration d'un partitionneur par échantillonnage

3 – Echantillonneurs prédéfinis

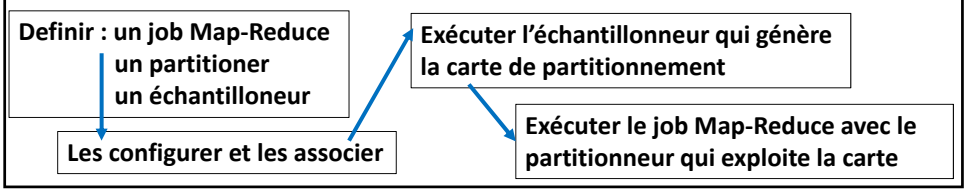
CentraleSupélec

Echantillonneurs prédéfinis (1)

```
// Define and start to configure a new Map-Reduce job
Job job = ...
job.setXXXX...
job.setYYYY...

// - Set the number of Reducers (required before to run the sampler)
job.setNumReduceTasks(REDUCE_TASKS);
// - Set the partition output path to use for the Map-Reduce job
//   (do it across a Partitioner configuration)
TotalOrderPartitioner.setPartitionFile(job.getConfiguration(), partitionOutputPath);

// Choice and configuration of a RandomSampler
//      10% of chance to choose a data, 10000: max number of choosen data
//      10: max number of split analyzed
InputSampler.Sampler mySmpl =  new InputSampler.RandomSampler(0.1,10000,10);

// Run the Sampler that builds and writes the partition map:
// - job is the Map-Reduce job requiring the partition map
// - Its reference allows to get the number of planned Reducers, the input data path,
//   the partition output path...
InputSampler.WritePartitionFile(job, mySmpl);
```

Echantillonneurs prédéfinis (2)

```
TotalOrderPartitioner.setPartitionFile(job.getConfiguration(), partitionOutputPath);
InputSampler.Sampler mySmpl = new InputSampler.RandomSampler(0.1,10000,10);
InputSampler.WritePartitionFile(job, mySmpl);
// Complete the Map-Reduce job configuration
job.setPartitionerClass(TotalOrderPartitioner.class);
job.setMapperClass(MyMapper.class);
job.setReducerClass(MyReducer.class);
...
// Run the Map-Reduce job, using the Partitioner that uses the partition map
// computed by the sampler.
System.exit(job.waitForCompletion(true) ? 0 : 1);
```



Bilan

Bilan du Map-Reduce d'Hadoop

Finalement :

- Les *Mappers* peuvent sortir du cadre du SPMD (multi-mappers) quand on utilise plusieurs fichiers (test sur le *path* des fichiers)
  → Gagne en généricité
- Le *Shuffle & Sort* semble être un schéma de comm contraint. Mais c'est une sorte de all-to-all des *Mappers* vers les *Reducers*, et reconfigurable en 3 points !
  → C'est un schéma de comm assez générique
- Un pipelining recouvre les calculs des *Mappers*, les communications et les regroupements de données
  → Le *Map-Reduce* possède une implantation optimisée
- Le paradigme *Map-Reduce* masque les aspects d'informatique distribuée aux développeurs applicatifs
  → Permet l'adhésion d'une large ''communauté d'utilisateurs Big Data'' (à l'opposé de la stratégie du HPC)

Algorithmique Map-Reduce