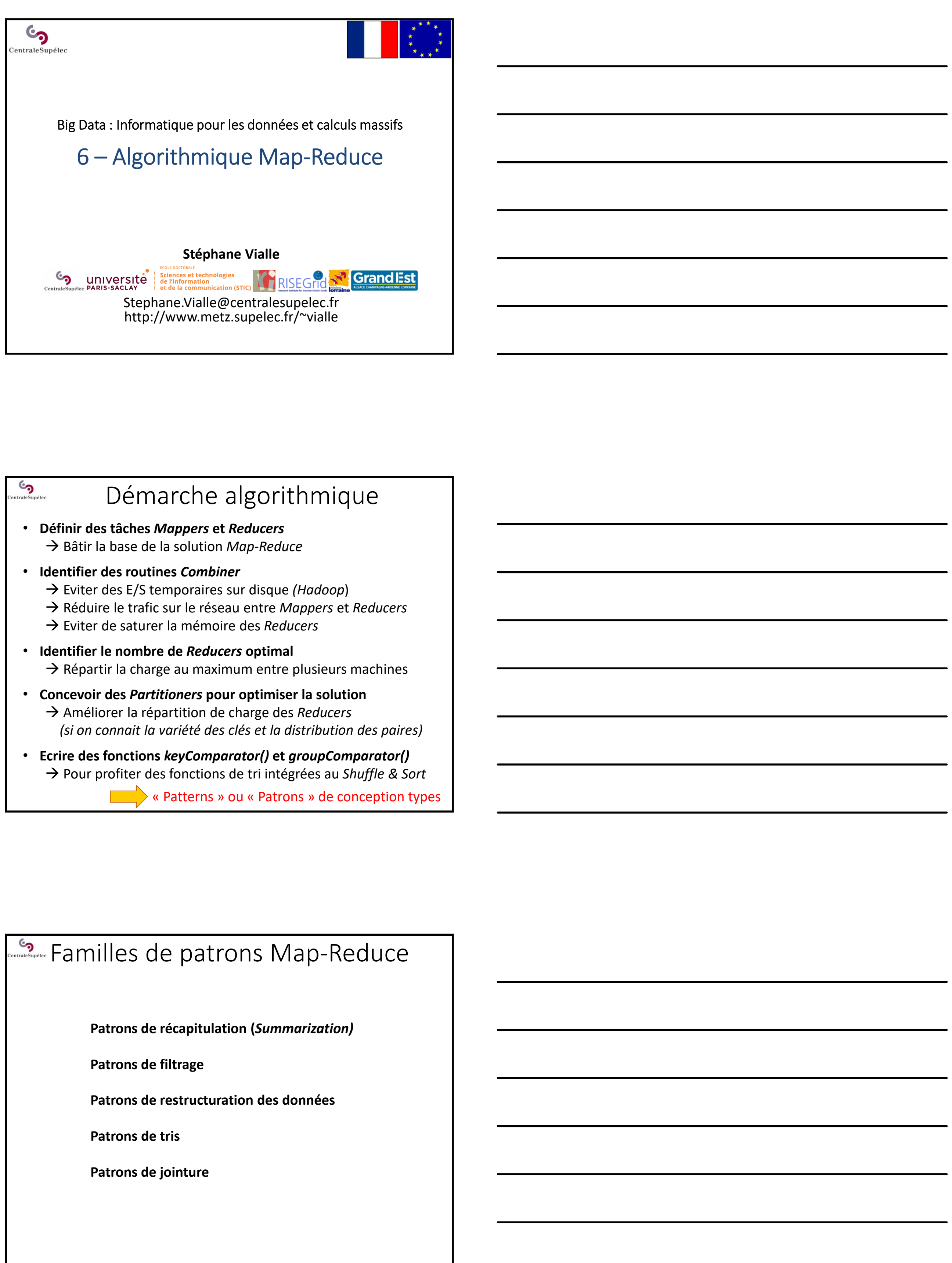
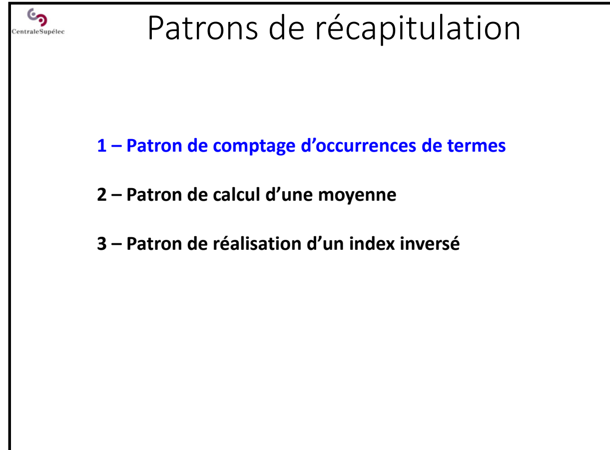
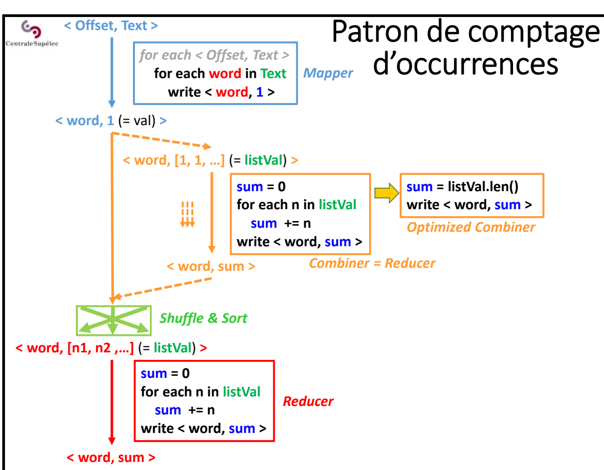
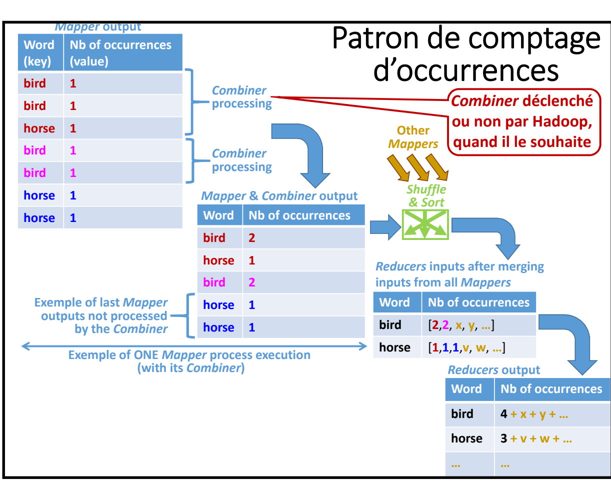

CentraleSupélec
Big Data : Informatique pour les données et calculs massifs
6 – Algorithmique Map-Reduce
Stéphane Vialle
université PARIS-SACLAY
Sciences et technologies de l'information et de la communication (STIC)
RISEGrid
GrandEst
Stephane.Vialle@centralesupelec.fr
http://www.metz.supelec.fr/~vialle
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



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
< word, sum >
Combiner = Reducer
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 :

Mapper → Combiner → (*Partitioner ?* →)
Mapper → Combiner → (*Partitioner ?* →)
Mapper → Combiner → (*Partitioner ?* →)
⋮
Mapper → Combiner → (*Partitioner ?* →)

Shuffle & Sort

Reducer →
⋮
Reducer →

- 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!

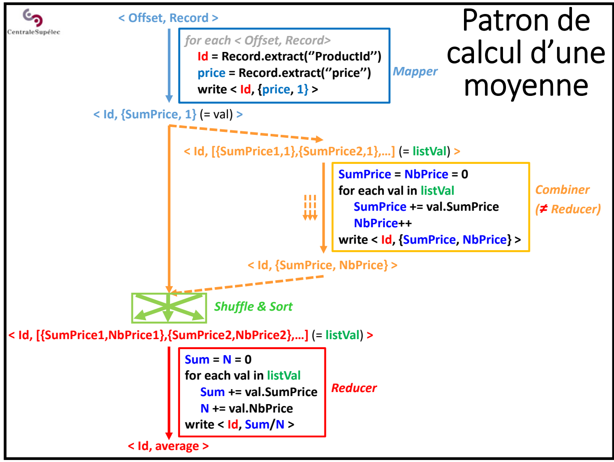
Patrons de récapitulation

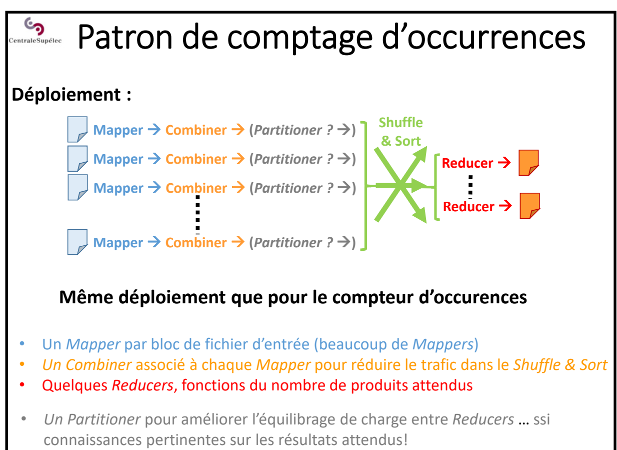
1 – Patron de comptage d'occurrences de termes

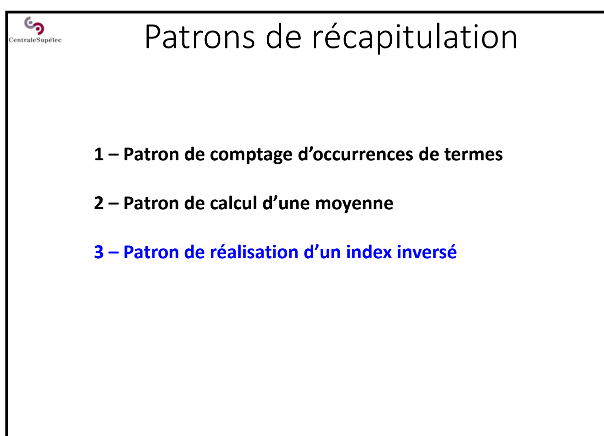
2 – Patron de calcul d'une moyenne

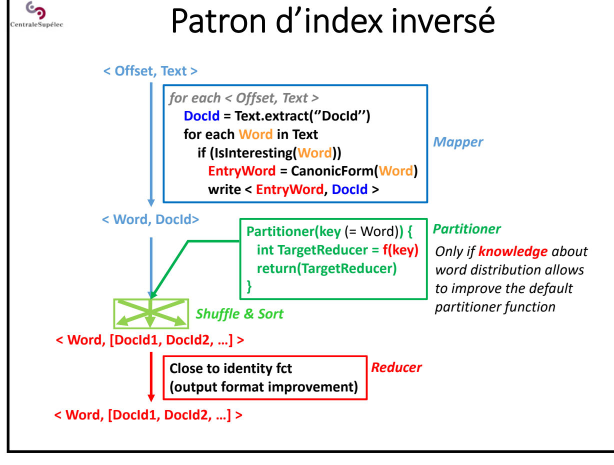
3 – Patron de réalisation d'un index inversé

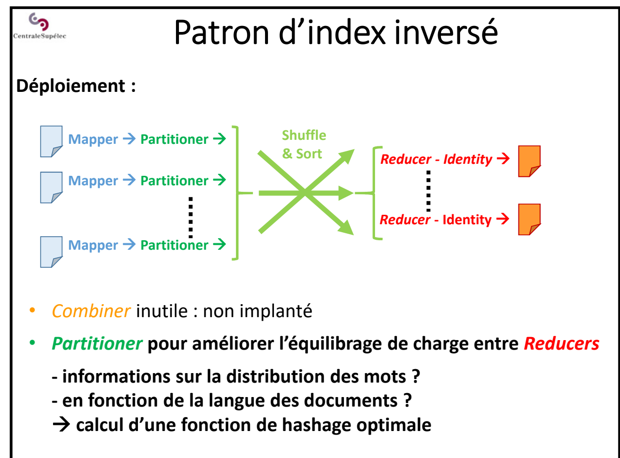
Patron de calcul d'une moyenne

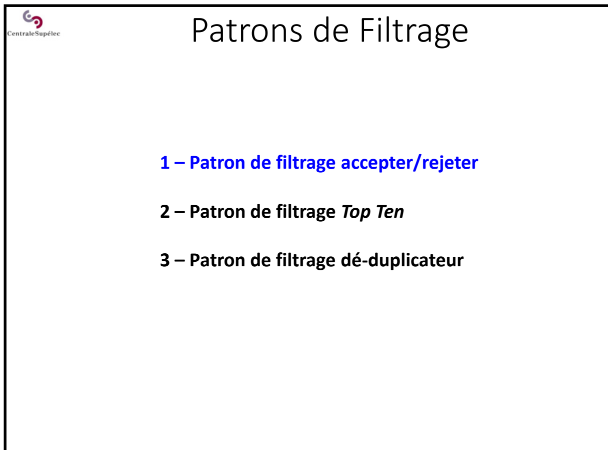


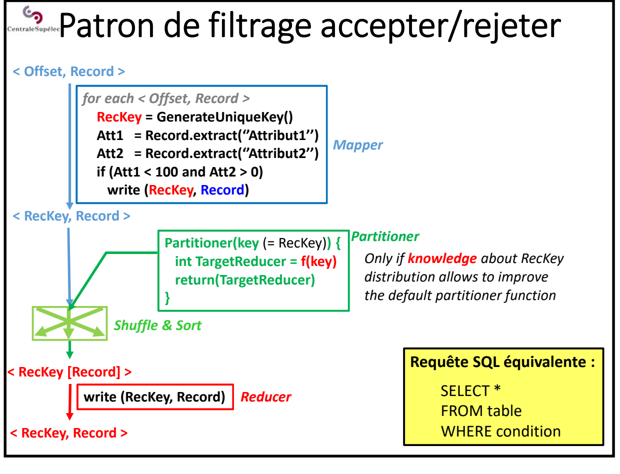

Patron de comptage d'occurrences
Déploiement :
Mapper → Combiner → (Partitioner ? →)
Shuffle & Sort
Reducer →
Même déploiement que pour le compteur d'occurences
Un Mapper par bloc de fichier d'entrée (beaucoup de Mappers)
Un Combiner associé à chaque Mapper pour réduire le trafic dans le Shuffle & Sort
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!

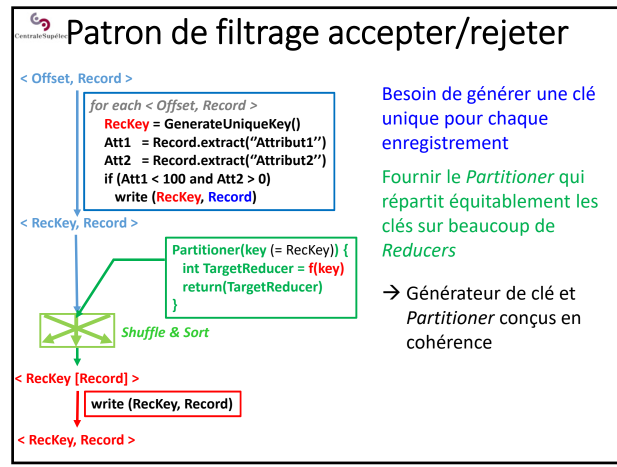

Patrons de récapitulation
1 – Patron de comptage d'occurrences de termes
2 – Patron de calcul d'une moyenne
3 – Patron de réalisation d'un index inversé

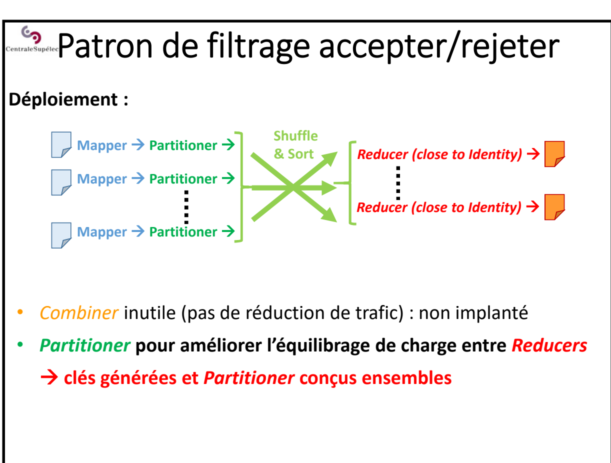

Patron d'index inversé
< Offset, Text >
for each < Offset, Text >
DocId = Text.extract("DocId")
for each Word in Text
if (IsInteresting(Word))
EntryWord = CanonicForm(Word)
write < EntryWord, DocId >
Mapper
< Word, DocId>
Partitioner(key (= Word)) {
int TargetReducer = f(key)
return(TargetReducer)
}
Partitioner
Only if knowledge about word distribution allows to improve the default partitioner function
Shuffle & Sort
< Word, [DocId1, DocId2, …] >
Close to identity fct
(output format improvement)
Reducer
< Word, [DocId1, DocId2, …] >


Patron d'index inversé
Déploiement :
Mapper → Partitioner →
Mapper → Partitioner →
Mapper → Partitioner →
Shuffle
& Sort
Reducer - Identity →
Reducer - Identity →
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
< Offset, Record >
for each < Offset, Record >
RecKey = GenerateUniqueKey()
Att1 = Record.extract("Attribut1")
Att2 = Record.extract("Attribut2")
if (Att1 < 100 and Att2 > 0)
write (RecKey, Record)
Mapper
< RecKey, Record >
Partitioner(key (= RecKey)) {
int TargetReducer = f(key)
return(TargetReducer)
}
Partitioner
Only if knowledge about RecKey distribution allows to improve the default partitioner function
Shuffle & Sort
< RecKey [Record] >
write (RecKey, Record)
Reducer
< RecKey, Record >
Requête SQL équivalente :
SELECT *
FROM table
WHERE condition



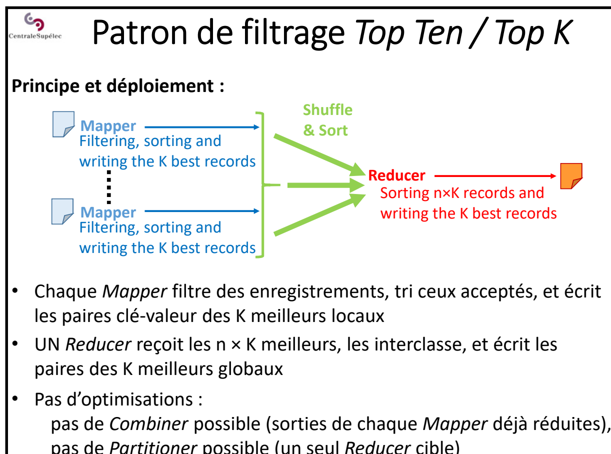


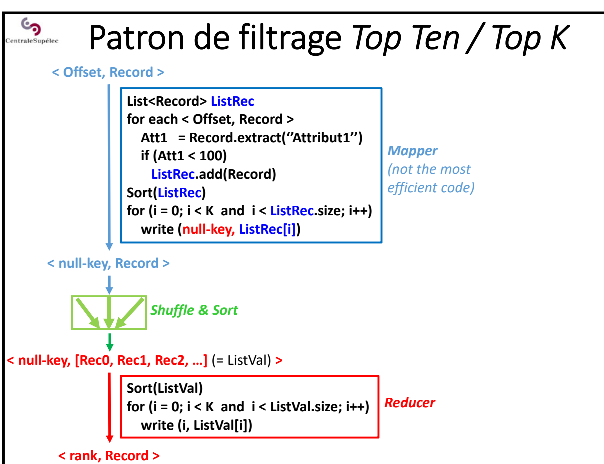
Patrons de Filtrage

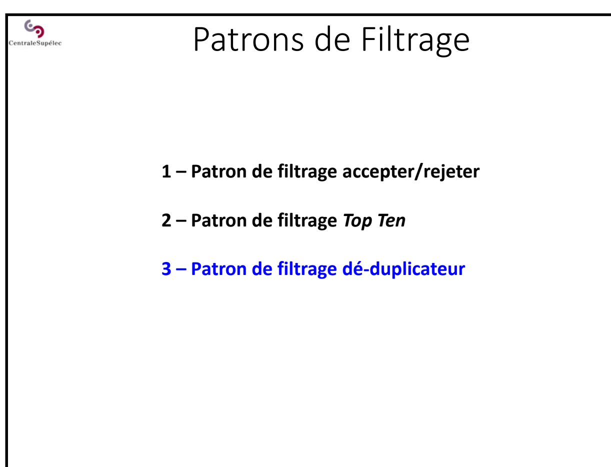
1 – Patron de filtrage accepter/rejeter

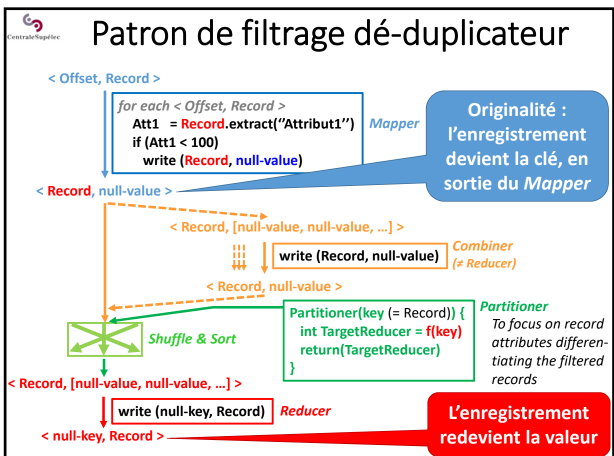
2 – Patron de filtrage *Top Ten*

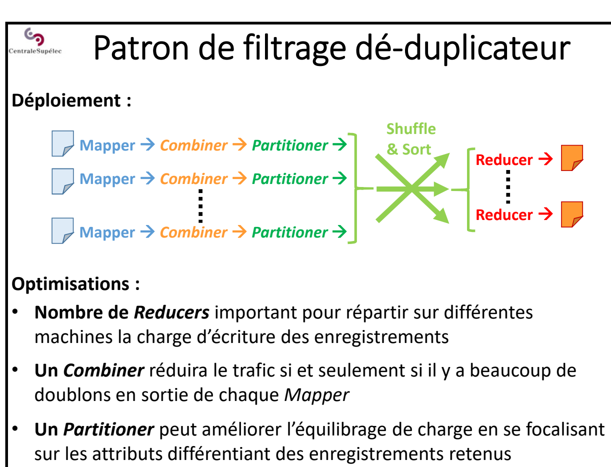
3 – Patron de filtrage dé-duplicateur

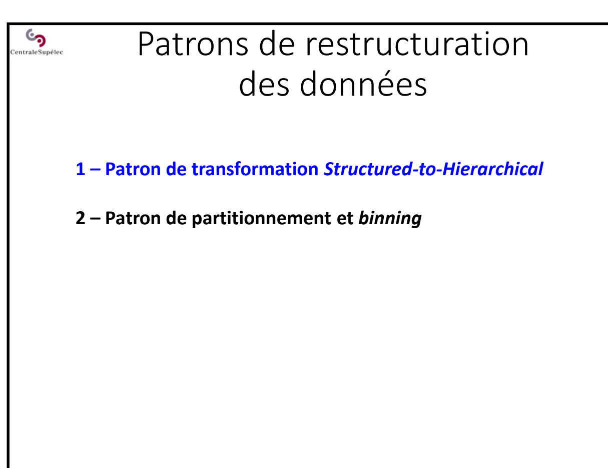

Patron de filtrage *Top Ten / Top K*
Principe et déploiement :
Mapper
Filtering, sorting and writing the K best records
Mapper
Filtering, sorting and writing the K best records
Shuffle & Sort
Reducer
Sorting n×K records and writing the K best records
• Chaque *Mapper* filtre des enregistrements, tri 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)

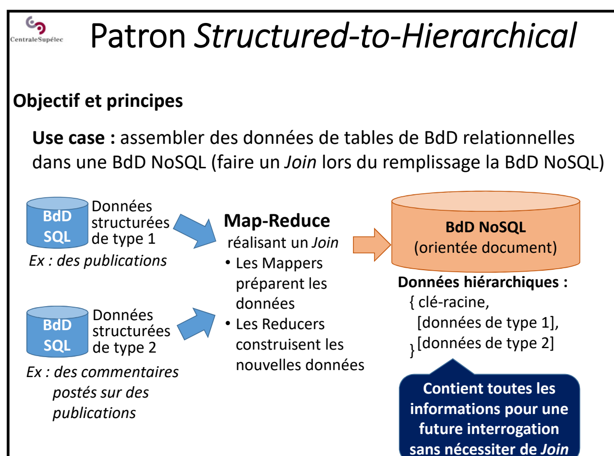

Patron de filtrage *Top Ten / Top K*
< Offset, Record >
List<Record> ListRec
for each < Offset, Record >
Att1 = Record.extract("Attribut1")
if (Att1 < 100)
ListRec.add(Record)
Sort(ListRec)
for (i = 0; i < K and i < ListRec.size; i++)
write (null-key, ListRec[i])
Mapper
(not the most efficient code)
< 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 >

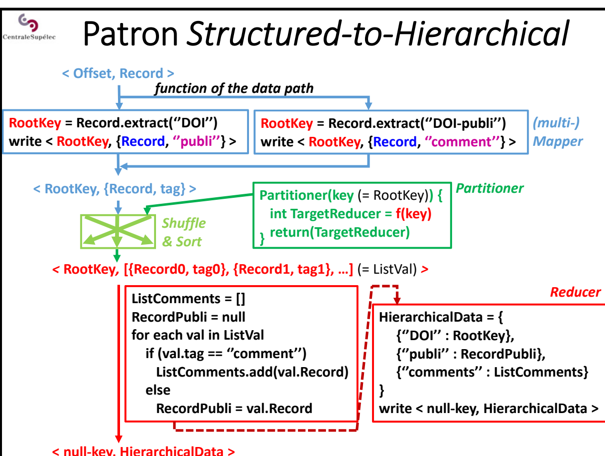

Patrons de Filtrage
1 – Patron de filtrage accepter/rejeter
2 – Patron de filtrage *Top Ten*
3 – Patron de filtrage dé-duplicateur

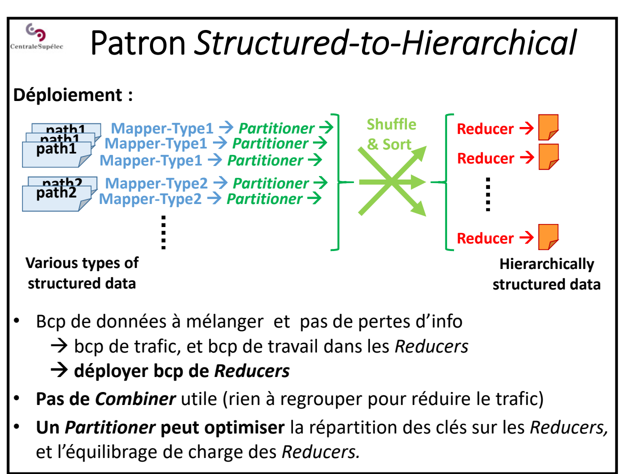

Patron de filtrage dé-duplicateur
< Offset, Record >
for each < Offset, Record >
Att1 = Record.extract("Attribut1")
if (Att1 < 100)
write (Record, null-value)
Mapper
Originalité : l'enregistrement devient la clé, en sortie du Mapper
< Record, null-value >
< Record, [null-value, null-value, ...] >
write (Record, null-value)
Combiner
(≠ Reducer)
< Record, null-value >
Partitioner(key (= Record)) {
int TargetReducer = f(key)
return(TargetReducer)
}
Partitioner
To focus on record attributes differentiating the filtered records
Shuffle & Sort
< Record, [null-value, null-value, ...] >
write (null-key, Record)
Reducer
< null-key, Record >
L'enregistrement redevient la valeur


Patron de filtrage dé-duplicateur
Déploiement :
Mapper → Combiner → Partitioner →
Mapper → Combiner → Partitioner →
Mapper → Combiner → Partitioner →
Shuffle & Sort
Reducer →
Reducer →
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)
BdD SQL
Données structurées de type 1
Ex : des publications
BdD SQL
Données structurées de type 2
Ex : des commentaires postés sur des publications
Map-Reduce réalisant un Join
• Les Mappers préparent les données
• Les Reducers construisent les nouvelles données
BdD NoSQL (orientée document)
Données hiérarchiques :
{ clé-racine,
[données de type 1],
[données de type 2]
}
Contient toutes les informations pour une future interrogation sans nécessiter de Join

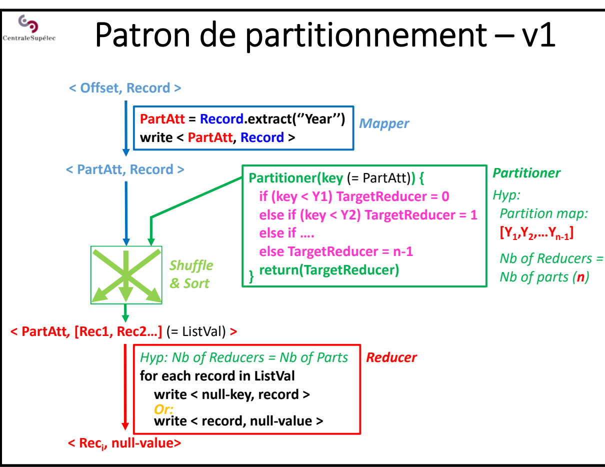

Patron Structured-to-Hierarchical
< Offset, Record >
function of the data path
RootKey = Record.extract("DOI")
write < RootKey, {Record, "publi"} >
RootKey = Record.extract("DOI-publi")
write < RootKey, {Record, "comment"} >
(multi-) Mapper
< RootKey, {Record, tag} >
Partitioner(key (= RootKey)) {
int TargetReducer = f(key)
return(TargetReducer)
}
Partitioner
Shuffle & Sort
< RootKey, [{Record0, tag0}, {Record1, tag1}, …] (= ListVal) >
Reducer
ListComments = []
RecordPubli = null
for each val in ListVal
if (val.tag == "comment")
ListComments.add(val.Record)
else
RecordPubli = val.Record
HierarchicalData = {
{"DOI" : RootKey},
{"publi" : RecordPubli},
{"comments" : ListComments}
}
write < null-key, HierarchicalData >
< null-key, HierarchicalData >

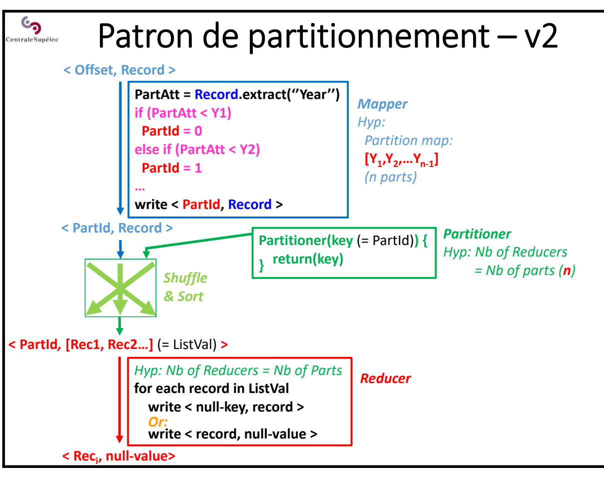

Patron Structured-to-Hierarchical
Déploiement :
path1
Mapper-Type1 → Partitioner →
Mapper-Type1 → Partitioner →
Mapper-Type1 → Partitioner →
path2
Mapper-Type2 → Partitioner →
Mapper-Type2 → Partitioner →
Shuffle & Sort
Reducer →
Reducer →
Reducer →
Various types of structured data
Hierarchically structured data
• 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.

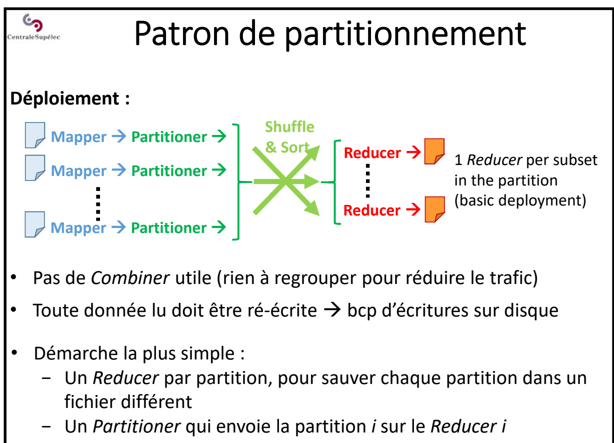
Patrons de restructuration des données

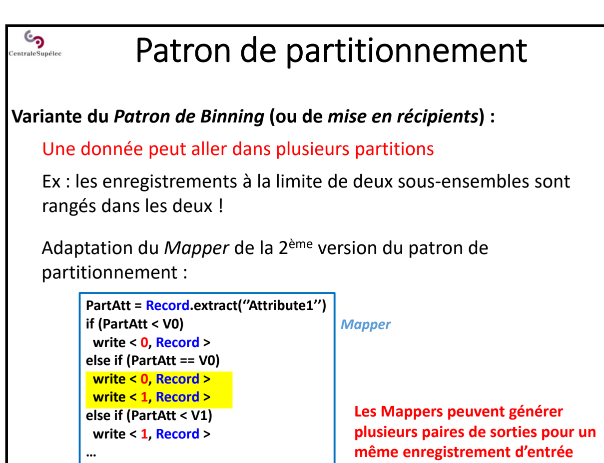
1 – Patron de transformation ***Structured-to-Hierarchical***

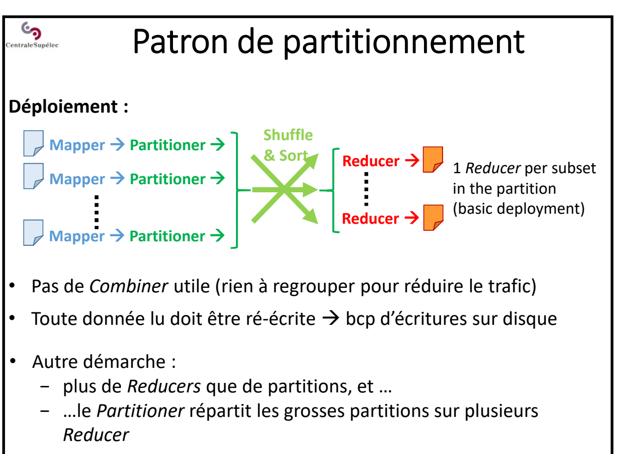
2 – Patron de partitionnement et *binning*






Patron de partitionnement
Déploiement :
Mapper → Partitioner →
Mapper → Partitioner →
Mapper → Partitioner →
Shuffle & Sort
Reducer →
Reducer →
1 Reducer per subset in the partition (basic deployment)
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 :
Mapper → Partitioner →
Mapper → Partitioner →
Mapper → Partitioner →
Shuffle & Sort
Reducer →
Reducer →
1 Reducer per subset in the partition (basic deployment)
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*

E

Etape préliminaire : définir des seuils de partitionnement $\{s_1, s_2, s_3\}$

1ère étape : partitionnement

$E_0 : x < s_1$

$E_1 : s_1 \leq x < s_2$

$E_2 : s_2 \leq x < s_3$

$E_3 : s_3 \leq x$

2ème étape : tri de chaque sous-ensemble

$E_0 : x < s_1$ → File-0

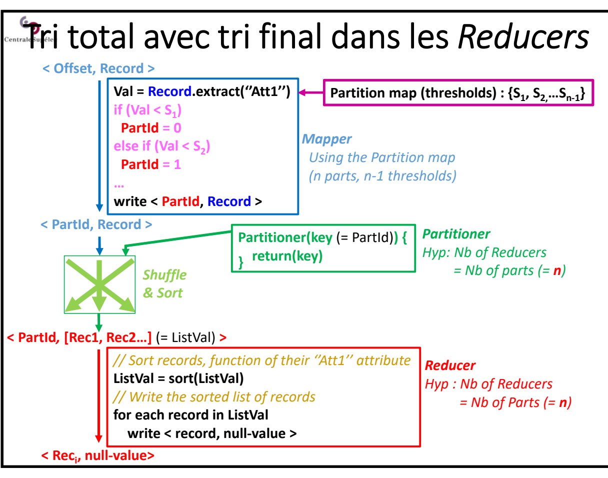
$E_1 : s_1 \leq x < s_2$ → File-1

$E_2 : s_2 \leq x < s_3$ → File-2

$E_3 : s_3 \leq x$ → File-3

- 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

Partition map (thresholds) : $\{S_1, S_2, \ldots S_{n-1}\}$

Mapper → Partitioner →
Mapper → Partitioner →
Mapper → Partitioner →
Shuffle & Sort
Reducer →
Reducer →
1 *Reducer* per subset in the partition (basic deployment)

- 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

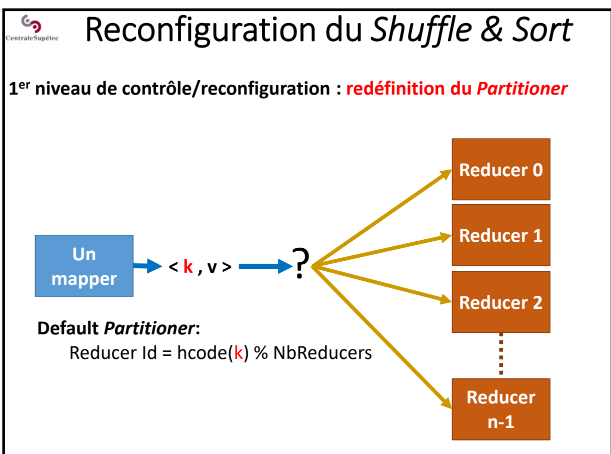
Clé simple : < k, v >
k : une seule valeur (ex : 10, ''toto''…)

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 …



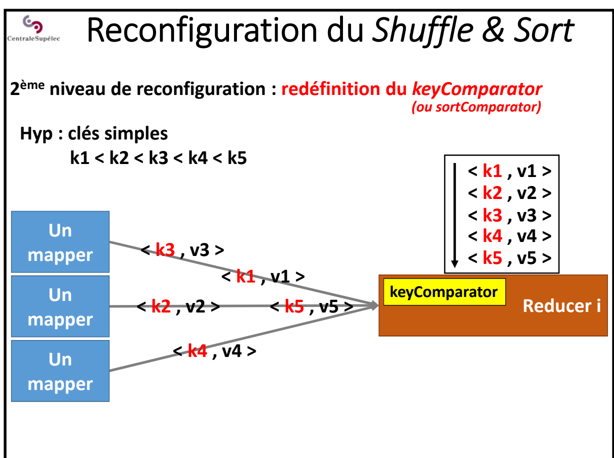
Reconfiguration du *Shuffle & Sort*

1er niveau de reconfiguration : redéfinition du *Partitioner*

```java
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)
  }
}
```

```java
//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 composites
k1 < k2 < k3
t11 < t12, et t21 < t22

Un mapper → < {k2,t21}, v3 >
Un mapper → < {k1,t11}, v1 > ; < {k1,t12}, v2 > ; < {k3,t31}, v5 >
Un mapper → < {k2,t22}, v4 >

keyComparator — Reducer i :
< {k1,t11} , v1 >
< {k1,t12} , v2 >
< {k2,t21} , v3 >
< {k2,t22} , v4 >
< {k3,t31} , v5 >

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

Un mapper → < {k2,t21}, v3 >
Un mapper → < {k1,t11}, v1 > ; < {k1,t12}, v2 > ; < {k3,t31}, v5 >
Un mapper → < {k2,t22}, v4 >

keyComparator, groupComparator — Reducer i :
< {k1,t11} , v1 >
< {k1,t12} , v2 >
< {k2,t21} , v3 >
< {k2,t22} , v4 >
< {k3,t31} , v5 >

reduce(< {k1,_}, [v1, v2] >)
reduce(< {k2,_}, [v3, v4] >)
reduce(< {k3,_}, [v5] >)

groupComparator sur 1er attribut de la clé seulement

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

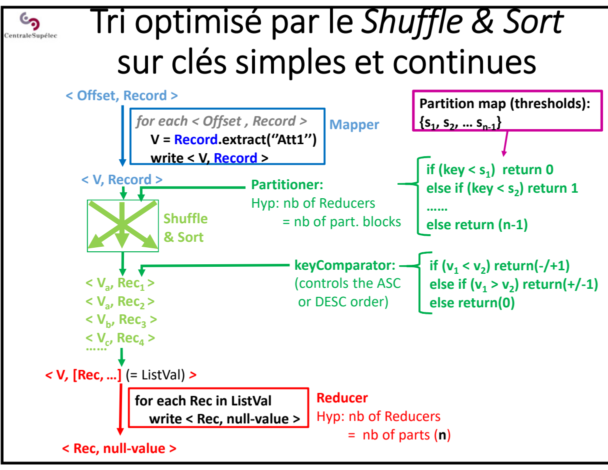
1 - Tri total avec tri final dans les *Reducers*

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

Partition map (thresholds) : $\{S_1, S_2, ... S_{n-1}\}$

Mapper → Partitioner →
Mapper → Partitioner →
Mapper → Partitioner →

Shuffle & Sort

1 *Reducer* per subset in the partition

keyComparator → Reducer →
keyComparator → Reducer →

- 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

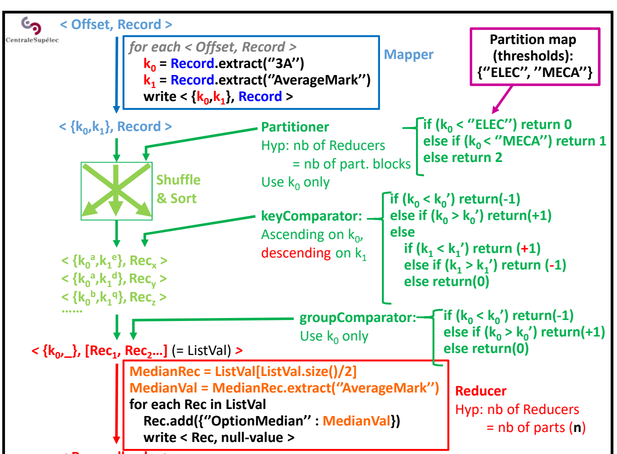
Trier des enregistrements d'étudiants selon 2 critères emboités :

1 – par option de 3A en ordre lexicographique croissant

2 – par moyenne générale en ordre décroissant

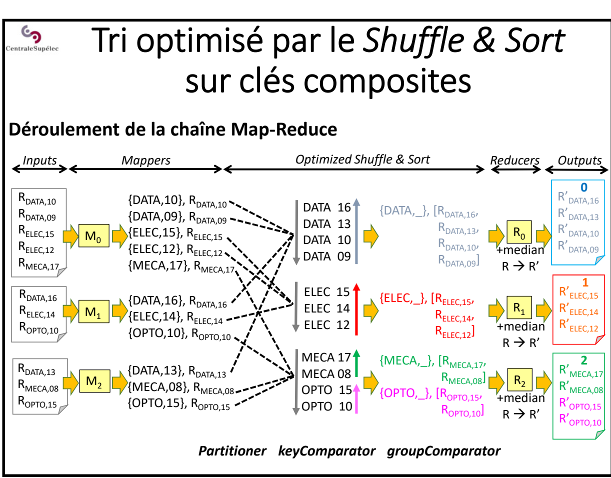
Pour chaque option de 3A :

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



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

Déroulement de la chaîne Map-Reduce



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

Déploiement et bilan

Partition map (thresholds) : $\{S_1, S_2, \ldots S_{n-1}\}$

1 *Reducer* per subset in the partition

Mapper → Partitioner →
Mapper → Partitioner →
Mapper → Partitioner →
Shuffle & Sort
keyComparator → *groupComparator* → Reducer →
keyComparator → *groupComparator* → Reducer →

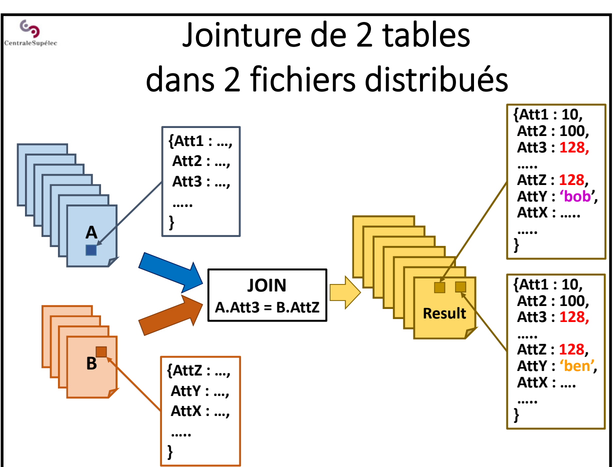
- Tri multi-critères (emboîtés)
- Exploite les tris de clés du *Shuffle & Sort*, **aucun tri final dans les *Reducers***
- Regroupement dans les *Reducers* et traitements par sous-listes
- Suppose l'existence d'un plan de partitionnement prédéfini…

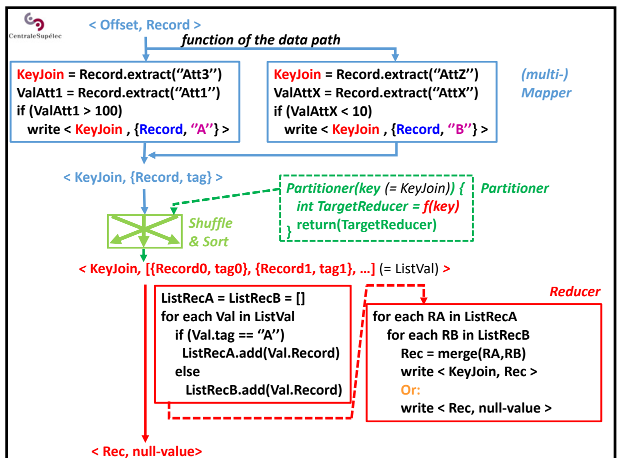
Patrons de jointure

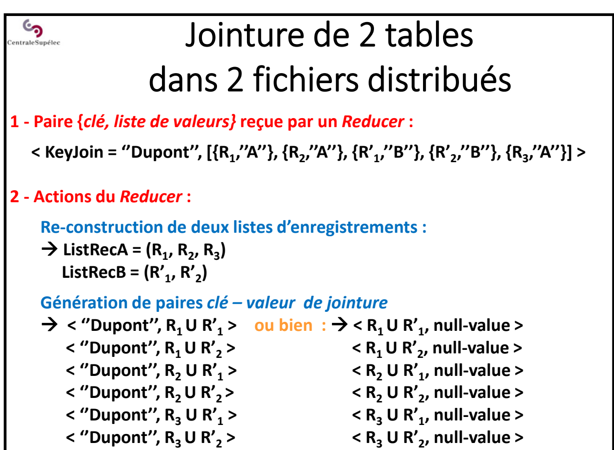
1 – Jointure de 2 tables dans 2 fichiers distribués

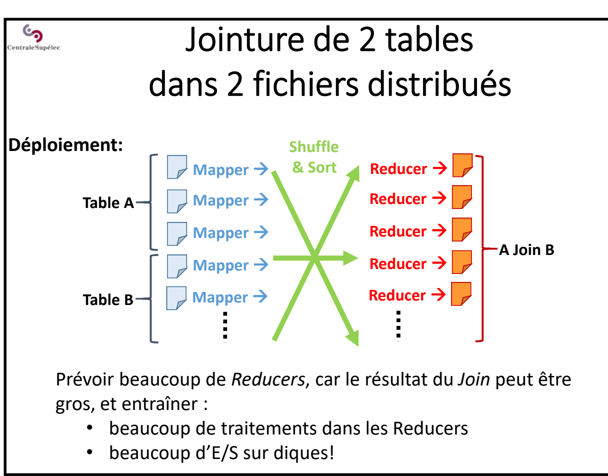
2 – Jointure de grande taille avec prétraitement

Jointure de 2 tables dans 2 fichiers distribués




< 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) >
Reducer
ListRecA = ListRecB = []
for each Val in ListVal
if (Val.tag == "A")
ListRecA.add(Val.Record)
else
ListRecB.add(Val.Record)
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", [{R1,"A"}, {R2,"A"}, {R'1,"B"}, {R'2,"B"}, {R3,"A"}] >
2 - Actions du Reducer :
Re-construction de deux listes d'enregistrements :
→ ListRecA = (R1, R2, R3)
ListRecB = (R'1, R'2)
Génération de paires clé – valeur de jointure
→ < "Dupont", R1 U R'1 >
< "Dupont", R1 U R'2 >
< "Dupont", R2 U R'1 >
< "Dupont", R2 U R'2 >
< "Dupont", R3 U R'1 >
< "Dupont", R3 U R'2 >
ou bien :
→ < R1 U R'1, null-value >
< R1 U R'2, null-value >
< R2 U R'1, null-value >
< R2 U R'2, null-value >
< R3 U R'1, null-value >
< R3 U R'2, null-value >


Jointure de 2 tables
dans 2 fichiers distribués
Déploiement:
Table A
Table B
Mapper →
Mapper →
Mapper →
Mapper →
Mapper →
Shuffle
& Sort
Reducer →
Reducer →
Reducer →
Reducer →
Reducer →
A Join B
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

CentraleSupélec

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)

Outils d'aide au partitionnement

1 – *TotalOrderPartitioner*

2 – Configuration d'un partitionneur par échantillonnage

3 – Echantillonneurs prédéfinis

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);
```

Definir : un job Map-Reduce
un partitioner
un échantillonneur

Les configurer et les associer

Exécuter l'échantillonneur qui génère la carte de partitionnement

Exécuter le job Map-Reduce avec le partitionneur qui exploite la carte

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