


BIG DATA  
2016-2017

# TP N°7: PATTERN MAP REDUCE

JOIN

YAHYA BACHIR  
NAJI ZAOU



Dans cet exercice on va implémenter le Pattern Join dans un programme Map-Reduce dans le but d'associer les villes à la région dans lesquelles elles appartiennent, comme d'habitude on va travailler avec le fichier « *worldcitiespop.txt* » qui contient toutes les villes du monde en plus d'un autre fichier « *region\_codes.csv* » qui contient les codes des pays, le nom de leurs régions et des codes 'lettres/chiffres' qui identifient les régions dans un pays.

---

## PARTIE 1 :

Dans cette partie on va utiliser deux Mappeurs : un pour les villes « **CityMapper** » et l'autre pour les régions « **RegionMapper** »

Pour atteindre notre objectif notre Mapper va envoyer un Key de type Text et une valeur de type « **TaggedValue** »

- La Classe **TaggedValue** : est la valeur qu'on envoie du Mapper au Reducer, il faut donc qu'elle implémente l'interface « Writable ». Elle est constituée d'une partie « Data » qui est une chaîne de caractères contenant les données qu'on veut envoyer et d'une partie « tag » qui est un booléen qui sert à identifier si les données viennent du CityMapper, ou bien du RegionMapper.
- Classe **CityMapper** : cette classe va lire notre fichier « *worldcitiespop.txt* » et, pour chaque ligne du fichier, il va nous renvoyer un Text comme clé et un TaggedValue comme la valeur:
  - Text : contient le code du pays et le code de la région pour une ville donnée, mais cela va être en minuscules car les codes sont écrits en minuscules dans le fichier *region\_codes.csv*
  - TaggedValue : contient le nom de la ville en plus d'un booléen qui est mis à 'True' pour dire que c'est une ville.
- Classe **RegionMapper** : cette classe va lire notre fichier « *region\_codes.csv* » et, pour chaque ligne du fichier, il va nous renvoyer un Text comme clé et un TaggedValue comme la valeur:
  - Text : contient le code du pays et le code de la région, qui seront écrits en minuscules juste pour être sûr qu'ils vont être les mêmes que ceux du CityMapper.
  - TaggedValue : contient le nom de la région et un booléen qui est mis en 'False' pour dire que ce n'est pas une ville, mais plutôt une région.

- **Classe *TP7Reducer*** : Pour que le Reducer puisse lire les sorties des deux Mapper, ces sorties doivent correspondre à [Text, TaggedValue] qui est la forme d'entrée de notre reducer.

Pour chaque ensemble d'information le reducer reçoit une liste de valeurs regroupées selon leurs clés, parmi elles, on trouve une valeur venant du RegionMapper et le reste du CityMapper. Pour identifier la valeur venant du RegionMapper des autres on test sur le « tag » de ces valeurs (vu que c'est des TaggedValue). Si le « tag » est True alors on la met dans une liste dédiée pour les villes, et quand c'est False on stock la valeur dans une variable réservée à la région. A la fin on écrit (code du pays, code de la région) comme clé et (code de la région      nom de la ville) comme valeur, donc on va avoir dans notre fichier de sortie quelque chose de la forme :

```
fr,33  Aquitaine  Bordeaux
fr,33  Aquitaine  Talence
fr,33  Aquitaine  Pessac
.....
```

- Pour le [\*main\(\)\*](#) il faut spécifier qu'on va travailler avec deux Mappeurs, pour cela on l'indique en ajoutant le ***MultipleInputs.addInputPath()*** au lieu du ***job.setInputFormat()***.

---

## PARTIE 2 :

Pour cette deuxième partie on veut envoyer au Reducer la région en premier puis les villes qui appartiennent à cette région, pour cela on a besoin d'un ***partitioner***, ***grouper*** et ***sorter***.

Ici est à la place d'envoyer une clés 'simple' au Reducer contenant le code et le pays de la ville ou la région , on va envoyer un « TaggedKey » qui a la même structure que « TaggedValue » utilisé dans la première partie.

Le ***partitioner*** va recevoir ces TaggedKey puis il va calculer leurs 'Hash', et finalement il va envoyer les clés qui ont le même 'Hash' au même Reducer (on va utiliser deux Reducers pour cette partie pour que notre *partitioner* puissent partitionner les données en deux. Cela marche aussi avec un seul reducer, cependant le *partitioner* ne va rien partitionner). Le ***grouper*** va recevoir ce qu'envoie le *partitioner*, et il va regrouper les clés qui sont exactement pareils, avant que les informations arrivent au Reducer, le ***sorter*** assure que la région et la première à être envoyée au Reducer dans chaque ensemble d'information.

- **Classe *TaggedKey*** : est la Clé qu'on envoie du Mapper au Reducer, il faut qu'il implémente l'interface « Writable », et l'interface « Comparable » qui va nous donner la méthode de comparaison 'compareTo(AnotherTaggedKey)' qui est appelée dans le « Sorter » cette méthode retourne 3 valeurs :

- 0 : les deux TaggedKeys ont le même tag.
- -10 : La 1ère TaggedKey concerne une ville avec et la 2ème une région.
- 10 : La 1ère TaggedKey concerne une région avec et la 2ème une ville.

La TaggedKey est constituée d'une chaîne de caractère qui contient l'information concrète : la clé naturelle, et un entier comme « tag » pour savoir si les informations viennent du « CityMapper » (tag mit à 10) ou bien « RegionMapper » (tag mit à 0)

- La méthode `compareTo(AnotherTaggedKey)` : compare les deux TaggedKey en comparant en premier leurs clés naturelles puis leurs tags.
  - Si les tags sont pareils le résultat sera "0", en d'autres termes si les deux clés viennent du même Mappeur le résultat sera "0".
  - Si la première clé arrive du CityMapper et la deuxième du RegionMapper le résultat va être "-10" cela va mettre les informations de la première TaggedKey à la fin d'un groupe de données qui sera lu par le Reducer.
  - Si la première clé arrive du RegionMapper et la deuxième du CityMapper le résultat va être "10" est cela va positionner les informations de la première TaggedKey dans la tête d'un groupe de données qui sera lu par le Reducer.

➤ **Classe *CityMapper*** : cette classe va lire notre fichier « *worldcitiespop.txt* » et, pour chaque ligne du fichier, il va nous renvoyer un TaggedKey comme clé et un TaggedValue comme la valeur:

- TaggedKey : Contient le code de la région et le code du pays pour une ville donnée et "10 " comme tag pour dire que c'est une ville.
- TaggedValue : pareil que la première partie de ce TP, il contient le nom de la ville et un tag 'True'.

➤ **Classe : *RegionMapper*** : cette classe va lire notre fichier « *region\_codes.csv* » et, pour chaque ligne du fichier, il va nous renvoyer un TaggedKey comme clé et un TaggedValue comme la valeur:

- TaggedKey : Contient le code de la région et du pays pour une région donnée et "0" comme tag pour dire que ce n'est pas une ville, mais plutôt une région.
- TaggedValue : pareil que la première partie de ce TP, il contient le nom de la région, et un tag 'False'

➤ **Class *Partitionner*** : Le partitionner se positionne entre le Mapper et le Reducer, son rôle est de Contrôler le partitionnement de flux de sortie du Mapper ou Mappers en se basant sur les clés : des TaggedKey.

Autrement dit le Partitionner prend le flux de sortie du Mapper et le divise en plusieurs (nombre de reduceurs dans le job) partitions d'informations suivant leur clé et comment on a implémenté la méthode `getPartitioner()`, la plupart du temps elle est implémentée comme étant une méthode de `Hash()`.

Ici dans notre exercice on va prendre la chaîne de caractères de notre clé (`TaggedKey.getData()`) et on va appliquer sur cette chaîne de caractères le `String.hashCode()`, qui retourne le Hash de notre chaîne de caractères puis on va faire un modulo avec le nombre des partitions qui n'est rien d'autre que le nombre de Reducers dans notre job.

Dans notre implémentation, et comme le nombre des tâches de réduction est deux (indiqué dans le main de la classe `TP7.java`), le résultat de `getPartitioner()` va être soit "0" soit "1", la partition 0 va contenir toutes les données dont les Hash de leurs `TaggedKey.getData()` est pair, de même la partition 1 va contenir les données dont le Hash de leurs `TaggedKey.getData()` est impair.

- **Classe *Grouper*** : le Grouper rassemble les Values en prenant en considération la clé naturelle qui représente pour nous (code du pays, code de la région) c.à.d. le `TaggedKey` sans le "tag".

Sans Grouper on risque de trouver des informations avec la même Clé qui se traitent avec les Reducers différents et notre résultat final va être faux.

Vu que notre Partitionner divise la sortie des Mappers en deux (partition 0 : paire et partition 1 : impair) notre Grouper a besoin de regrouper dans la partition 0 les données qui ont la même Clé, et de même pour la partition 1.

Pour faire cela il faut quand Override la méthode `compare()` de notre superclasse `WritableComparator` qui compare deux `WritableComparable`, et comme notre `TaggedKey` est une instance de cette classe on peut faire un cast pour accéder à la Clé naturelle puis on la compare avec la Clé naturelle suivante en utilisant le `String.compareTo()` qui retourne 0 si les deux chaînes de caractères sont égales, et une valeur différente de 0 d'autre sinon. En faisant cela on est sûrs que toutes les données avec la même clé seront regroupées dans le même ensemble de données traité par un seul appel de `Reducer.reduce()`.

- **Classe *Sorter*** : le Sorter prend place après le Grouper. Le rôle du Sorter va être de positionner l'information venant du `RegionMapper` à la tête de l'ensemble de données qui a été regroupé par le Grouper (les informations avec la même Clé naturelle). Pour faire cela il faut quand Override la méthode `compare()` de notre superclasse `WritableComparator` qui compare deux `WritableComparable`, et comme notre `TaggedKey` est une instance de cette classe, donc on peut faire un cast pour pouvoir

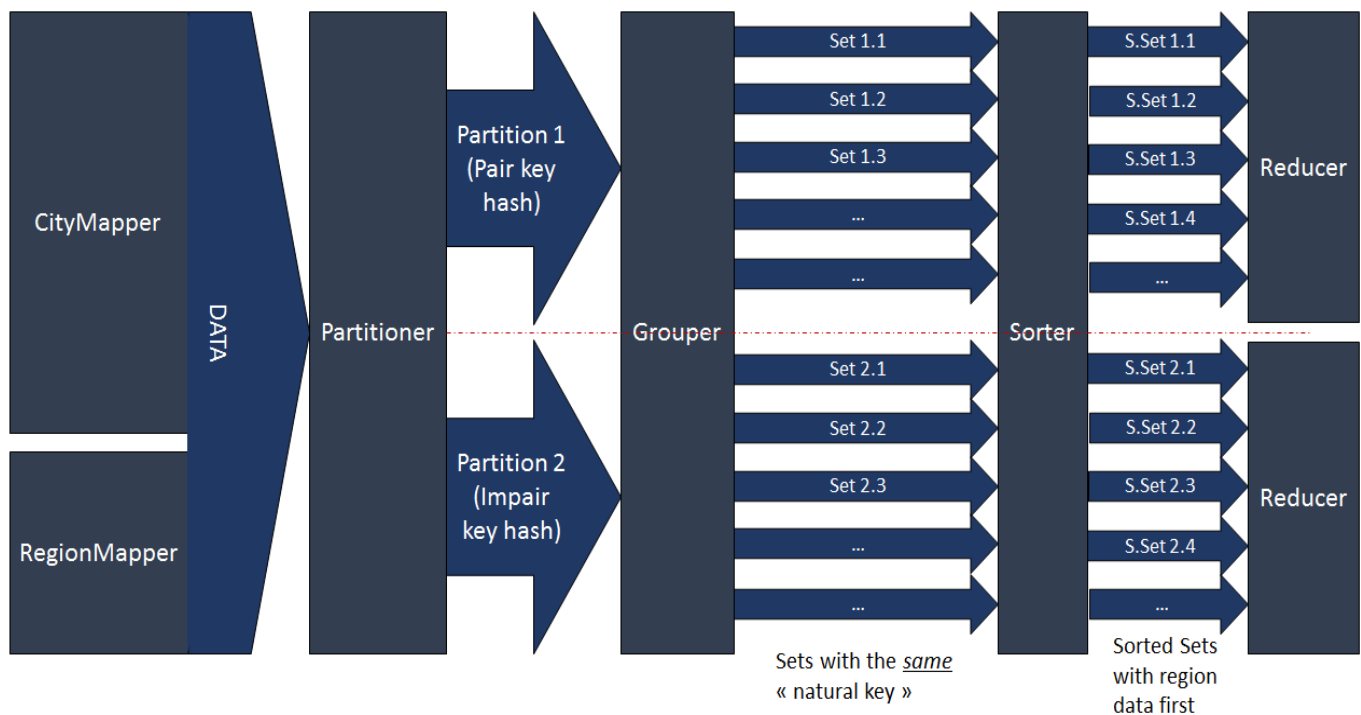
appeler la méthode `TaggedKey.compareTo()`, cette méthode va mettre la ligne de données qui a le 'tag' de la région en premier suivi par celles qui ont le 'tag' de la ville, et cela dans chaque ensemble de données.

- **Classe `TP7Reducer`** : Pour que le Reducer puisse lire les sorties des deux Mapper, ces sorties doivent correspondre à `[TaggedKey, TaggedValue]` qui est la forme d'entrée de notre Reducer.

On a choisi de garder le même reducer que la première partie. Cependant, on aura pu le changer de façon à lui préciser que la première donnée, dans chaque ensemble lu, vient du `RegionMapper` et que celles qui viennent après viennent tous du `CityMapper`, ainsi on évite de faire des tests, et notre programme gagne considérablement en performance et en temps d'exécution, vu qu'il suffira de stocker la première valeur et de l'écrire avec toutes celles qui viennent après.

- Pour le [`main\(\)`](#) : il faut spécifier qu'on va travailler avec deux Mappeurs, pour cela on l'indique en ajoutant le **`MultipleInputs.addInputPath()`** au lieu du **`job.setInputFormat()`**. Comme on a travaillé avec un Partitionner, Grouper et Sorter il faut qu'on les indique à notre job pour qu'il les prenne en considération :

- **`job.setPartitionerClass()`**
- **`job.setGroupingComparatorClass()`**
- **`job.setSortComparatorClass()`**



N.B. : Pensez à regarder la JavaDoc.