Université de Bordeaux I

Programmation large échelle

Compte rendu du TP 7
Map Reduce Patterns 2

Réalisé par :

GAMELIN Antoine
LASSOUANI Sofiane

Université Bordeaux 1 (2016–2017) – Master Génie Logiciel (UE PLE)

TABLE DES MATIÈRES

Chapitre 1

Introduction

1.1 Introduction

Ce TP est divisé en deux partie principale :

Partie 1 : Reduce-side join

Le but est d'implémenter un pattern courrant en Map Reduce : le "side join". vue en cours.

Le principe de ce pattern consiste à faire un produit cartésien à partir d'une clef prédéfini entre deux sorties de mappers différents et les fournir au reducer et sortir un seul resultat final.

Partie 2 : Scalable join
L'objectif de cette partie est de faire en sorte que les paire <clé,valeur> arrive triée au reducer afin de palier a un problème de mémoire qu'on verra en premiere partie.

Chapitre 2

Projet

2.1 Partie 1 : Reduce-side join

2.1.1 Introduction :

Le but de cette la partie 1 est d'implémenté un inner join dans le reducer. Le inner join à besoin d'une key commune entre les deux ensembles d'entrés. Pour ce TP on effectuera le join entre les deux fichiers worldcitiespop.txt et region_codes.csv (qui contient le nom de chaque région en fonction de son identifiant pays et de l'identifiant de la région).

2.1.2 Exercice 1 : TaggedValue

L'exercice 1 consiste a écrire une nouvelle classe **TaggedValue** qui va nous servir à unifier les villes et les régions.
Elle a comme attribut name, qui contient la chaine de caractère correspondant soit à la ville ou à la région, et un attribue type qui est a 0 si c'est une ville et 1 si c'est une region.

2.1.3 Exercie 2 : Mappers

Afin de réussir un Inner join on doit implementer un mapper pour chaque entrée de donnée. Dans notre cas, nous aurons deux mappers, un qui va parcourir le fichier worldcitiespop.txt et l'autre parcourira region_codes.csv.

Chaque mappers nous retournera un <Text, **TaggedValue**> , le Text contient la clef naturel qui permet de faire la jointure entre les deux fichiers, et le TaggedValue, contiendra toutes les villes régions (structure définie ci-dessus).

Afin d'exécuter les deux mappers on a utiliser la classe **.apache.hadoop.mapreduce.lib.input.MultipleInputs** et sa methode **addInput-Path** lors de la configuration du job.

2.1.4 Exercie 3 : Reducer

Le réduceur effectuera le join entre les deux fichiers permettant de génerer un fichier sortie qui comportera le nom de chaque villes avec le nom de sa région.

Pour cela on initialise une liste nommé **waitCity** qui contiendra le nom des villes, jusqu'à ce qu'on reçoit le TaggedValue de la région. Le réducer lit tous les messages envoyés par le mapper, grouper par la clef (concaténation de la région code et num code).

Tant que nous n'avons pas reçu le TaggedValue, contenant la région, nous ajoutons chaque ville, dans la ArrayList **waitCity**.

Une fois qu'on tombe sur le message contenant la région. Nous mettons cette région dans une variable et nous parcourons l'ArrayList waitCity, et on fait un context.write, de la ville avec la région.

Pour le parcours de ville restante nous faisons un context.write de la ville concaténée à la région.

2.2 Partie 2 : Scalable join

2.2.1 Introduction

Comme nous avons pu voir lors de la partie précédente nous sommes obligés de stocker les villes dans une ArrayList, dans notre cas waitCity, le temps que nous tombons sur la région pour ensuite pouvoir réaliser les context.write.

Le problème de cette solution, si l'arrayList devient très importante, nous devrons stocker toutes les informations, qui peut lever une exception de type OutOfMemory.

C'est pourquoi l'objectif de cet exercice et de trouver un moyen de pallier a ce probléme et de faire en sorte que le reduceur reçoit un message trié.

2.2.2 Exercice 1 : TaggedKey

Dans un premier temps nous allons écrire une classe TaggedKey, au lieu de stocker juste la clef naturel permettant de faire la jointure, cette classe contiendera la clé naturelle et d'un champ keyType, permettant de dire si la value de cette clef est une région ou une ville. Cette classe implémente WritableComparable, il est donc nécessaire de réimplémenter la méthode compare.

2.2.3 Exercice 2 : Partitionner

Désormais les mappers nous renvoi une clef composite, contenant la clef naturel de jointure et le type permettant de dire que la valeur de cette clef sera une région ou une ville.

L'avantage d'hadoop c'est de pouvoir réaliser des calculs en parallèles, mais dans notre cas il est important que les messages avec la région et les message avec la ville pour la même clef naturel, soit traité sur la même machine, sinon il est impossible de pouvoir faire la concaténation.

C'est là qu'intervient le partitionner, son objectif c'est de s'assurer que le message avec la même clef naturel soit envoyé sur le même réduceur.

Pour cela, nous prenons le hashcode de cette clef et nous l'envoyons au reduceur en utilisant le modulo %numpartitions

2.2.4 Exercice 3 : Grouping & sort

2.2.4.1 Grouping

Désormais que le partitionner nous envoi les messages avec la même Clef naturel sur le bon réducer, il est préférable de ne faire qu'un seul appel à la méthode réduce pour la même clef naturel.

En effet actuellement pour une clef naturel, nous allons faire deux appel à réduce, un premier appel avec la région, un second avec les messages des villes.

L'objectif de ce grouping c'est de ne faire qu'un seul appel reduce. Pour cela nous avons créer une classe RSJGrouping qui implémente WritableComparator, il est important de

définir le constructeur sinon nous avons une erreur avec JavaNullPointerException, en effet le constructeur par défaut (celui de WritableComparator) fait un super(null).

Il faut également implémenté la méthode compare, qui va faire un compareTo, sur les clef naturel, renverra 0 si ceux sont les mêmes ce qui permettra d'ignorer le type de la clef composite et de les grouper en une seul clef.

2.2.4.2 Sort

Pour la fonction de sort, c'est le même principe que le grouping, sauf que cette fois-ci on ne regarde pas uniquement la clef, on compare également le type, on utilise la méthode compareTo sur deux Integer et comme on veut dans l'ordre décroissant on souhaite que la région soit tout en haut on multiplie le résultat par -1.

2.2.5 Exercice 4 : Reducer

Cette fois-ci, dans le réducer nous avons d'abord la région avant les villes, nous avons donc plus besoin de l'ancienne ArrayList waitCity, le premier message on le stock dans une variable région, puis après on lis chaque ville et on peut faire un context.write de la ville concaténer à la région.

2.3 Difficultés rencontrées

Lors de la réalisation de ce TP, deux problème ont été rencontré, dans un premier temps l'erreur avec le NullPointerException, car nous avons pas implémenté le constructeur.

Le second problème rencontré c'est le grouping, en effet pour une raison obscure le Grouping n'arrive pas faire de sorte de faire un appel à un seul réduce, pourtant nous avons dans un premier temps utiliser la fonction compareTo sur la chaine de caractère de la clef naturel, mais celle-ci ne grouper pas nos résultats.

En second temps nous avons implémenter nous même la fonctions, si c'était equals, on retourner 0 sinon on retourner 1 ou -1. Mais le grouping ne s'executer toujours pas.

Et enfin, on a voulu faire un compareTo sur le hashcode de cette clef naturel, mais cela ne fonctionnait pas non plus.

Après plusieurs essai, nous avons essayé de trouver un moyen d'ignorer ce grouping, étant donné que le sort fonctionnait, nous avons donc créer une variable globale au reduce, qui contient le nom de la région. Comme les messages sont trié le premier message sera une région, donc dès qu'on a une région on met à jour la variable, et si ce n'est pas une région on fait la concaténation de la ville avec la variable gloabale.

2.3.1 Aperçu du code pour essayer de debugger le programme.

```
public static class RSJGrouping extends WritableComparator {
  public RSJGrouping() {
    super(TaggedKeyWritable.class, true);
  }
  public int compare(TaggedKeyWritable w1, TaggedKeyWritable w2) {
    //return w1.t.keyName.hashCode() < w2.t.keyName.hashCode() ? -1 :
        (w1.t.keyName==w2.t.keyName ? 0 : 1);
    //return w1.t.getName().equals(w2.t.getName()) ? 0 : w1.t.keyName.hashCode() <
        w2.t.keyName.hashCode() ? -1 : 1 ;
    return (w1.t.getName()).compareTo(w2.t.getName());
  }

}

public static class RSJReducer extends Reducer<TaggedKeyWritable,
    TaggedValueWritable,NullWritable,Text> {
  int reduce = 0;
  public void reduce(TaggedKeyWritable key, Iterable<TaggedValueWritable> values,
      Context context) throws IOException, InterruptedException {
    String region = null;

    for(TaggedValueWritable taggedvalue : values){
    if(taggedvalue.t.typeValue == 1){
      region = taggedvalue.t.name;
      context.write(NullWritable.get(), new
          Text(reduce+"*"+key.t.getName().hashCode()+","+region));
    }
    else {
      context.write(NullWritable.get(), new
          Text(reduce+"*"+key.t.getName().hashCode()+","+taggedvalue.t.name+","+region));
    }
  }

  this.reduce++;
  }
}
```

Chapitre 3

Conclusion

Ce tp nous a permis de mettre en pratique le pattern Reduce-side join étudié en cours .

Ce pattern est très simple d'implémentation et permet de faire une jointure entre deux ensemble d'entrée.

Dans notre implementation on a utilisé une liste afin de sauvgarder les valeur d'une clé etant donné qu'il n'arrivé pas trié,ce qui peut lever une exception OutOfMemory si le nombre de ces valeur est important, c'est pourquoi dans la deuxieme partie de ce TP, on fait en sorte de trier les valeurs pour avoir en premier temps la région..

Ce TP nous a permis de bien comprendre les fondamenteaux du join.