HMIN122M
Entrepôts de données et big-data

TP Hadoop / Map-Reduce

Auteur :

Gracia-Moulis Kévin (21604392)
Canta Thomas (21607288)

Master 1 - AIGLE/DECOL
Faculté des sciences de Montpellier
Année universitaire 2020/2021

Table des matières

Partie 1

Exercice 1

✎ Modifier la fonction reduce du programme WordCount.java afin que seulement les mots dont le nombre d'occurrences est supérieur ou égal à deux soient affichés :

```
0 public void reduce(Text key, Iterable<IntWritable> values, Context context)
1 throws IOException, InterruptedException {
2   int sum = 0;

4   for (IntWritable val : values)
5     sum += val.get();

7   if(sum >= 2)
8     context.write(key, new IntWritable(sum));
9 }
```

Listing 1 – Modification de la fonction reduce

Exercice 3

✎ Compléter le code dans la classe GroupBy.java qui est fournie afin d'implémenter un opérateur de regroupement sur l'attribut Customer-ID du fichier de données fourni dans le répertoire input-groupBy :

```
0 private final static String emptyWords[] = { "" };

2 @Override
3 public void map(LongWritable key, Text value, Context context) throws
      IOException, InterruptedException {
4   String line = value.toString();
5   String[] words = line.split(",");

7   if(Arrays.equals(words, emptyWords)) return;

9   try {
10    String word = words[5]; // Customer ID
11    double number = Double.parseDouble(words[20]); // Profit
12    DoubleWritable write = new DoubleWritable(number);
13    context.write(new Text(word), write);
14  }catch (Exception e){}
15 }
```

Listing 2 – Fonction map de GroupBy

```java
public void reduce(Text key, Iterable<DoubleWritable> values, Context context)
    throws IOException, InterruptedException {
  double sum = 0;

  // Somme total
  for(DoubleWritable val : values)
    sum += val.get();

  context.write(key, new DoubleWritable(sum));
}
```

Listing 3 – Fonction reduce de GroupBy

Exercice 4

Pour cette exercice il n'est pas nécessaire de modifier la fonction *reduce* précédente.

Montant des ventes par Date et State

```java
public void map(LongWritable key, Text value, Context context) throws
    IOException, InterruptedException {
  String line = value.toString();
  String[] words = line.split(",");

  if(Arrays.equals(words, emptyWords)) return;

  try {
    String word1 = words[2]; // Date
    String word2 = words[10]; // State
    String word = word1 + "||" + word2; // concaténation de Date et State
    double number = Double.parseDouble(words[17]); // Sales
    DoubleWritable write = new DoubleWritable(number);
    context.write(new Text(word), write);
  }catch (Exception e){}
}
```

Listing 4 – Fonction map de GroupBy

Nombre de produit différent vendu

Il suffit de modifier notre variable *write* pour qu'elle renvoie 1 pour chaque produit différent. La somme effectué dans le reduce, sera donc le total de produits différents.

```java
public void map(LongWritable key, Text value, Context context) throws
    IOException, InterruptedException
...  ...
  DoubleWritable write = new DoubleWritable(1);
...  ...

}
```

Listing 5 – Fonction map de GroupBy

Nombre total d'exemplaire d'un produit vendu

```
0  public void map(LongWritable key, Text value, Context context) throws
       IOException, InterruptedException {
...    ...
7        String word1 = words[2]; // Date
8        String word2 = words[10]; // State
9        String word3 = words[13]; // Product ID
10       String word = word1 + "||" + word2 + "||" + word3;
11       double number = Double.parseDouble(words[18]); // Sales
12       DoubleWritable write = new DoubleWritable(1);
13       context.write(new Text(word), write);
14     }catch (Exception e){}
15 }
```

Listing 6 – Fonction map de GroupBy

Montant des ventes par Date / Category

Il suffit de modifier notre variable *word2* comme étant la catégorie.

```
...    ...
8      String word2 = words[14]; // Category
...    ...
```

Listing 7 – Fonction map de GroupBy

Exercice 5

✎ Sur la base des programmes *WordCount.java* et *GroupBy.java*, définir une classe *Join.java* permettant de joindre les lignes concernant les informations des clients et des commandes contenus dans le répertoire input-join :

Nous nous sommes servis du nombre de colonnes de chaque tables afin de les différencier (Customer possède 8 colonnes et Orders en possède 9). Les différencier nous permettra de choisir ce que l'on va renvoyer à notre fonction *reduce*.

De plus, nous avons ajouté un "séparateur" lors de l'écriture de notre couple (clé, valeur) dans la table *Customer* permettant aussi à notre fonction *reduce* d'elle aussi faire la différence lors de la réception et de stocker correctement ces valeurs.

```
... public static class Map extends Mapper<LongWritable, Text, Text, Text> {
...   private final static String emptyWords[] = { "" };

...   @Override
...   public void map(LongWritable key, Text value, Context context) throws
       IOException, InterruptedException {
...     String line = value.toString();
...     String[] words = line.split("\\|");

...     if(Arrays.equals(words, emptyWords))
...       return;

...     if (words.length == 8) // on Customer (key: custkey, value: name)
...       context.write(new Text(words[0]), new Text("|"+words[1]));

...     else // on Order (key: custkey, value: comment)
...       context.write(new Text(words[1]), new Text(words[8]));
...   }
... }
```

Listing 8 – Fonction map de Join

```
... public static class Reduce extends Reducer<Text, Text, Text, Text> {

...   @Override
...   public void reduce(Text key, Iterable<Text> values, Context context) throws
       IOException, InterruptedException {

...     ArrayList<String> cust = new ArrayList<>();
...     ArrayList<String> comment = new ArrayList<>();

...     // recopie de nos valeurs
...     for (Text val : values) {
...       String line = val.toString();
...       String[] words = line.split("\\|");

...       if (words.length == 2) cust.add(words[1]);
...       else comment.add(words[0]);
...     }

...     // on écrit nos couples (Customer, Comment)
...     for(String _cust : cust)
...       for (String _comment : comment)
...         context.write(new Text(_cust), new Text(_comment));
...   }
... }
```

Listing 9 – Fonction reduce de Join

Exercice 6

✎ Modifiez le programme de l'exercice 5 afin de calculer le montant total des achats faits par chaque client :

Pour notre fonction *map*, une seule ligne sera modifié afin de remplacer le commentaire par le prix total

```
...   ...
14      else // on Order (key: custkey, value: totalprice)
15        context.write(new Text(words[1]), new Text(words[3]));
...   ...
```

Listing 10 – Fonction map de Join

Pour la fonction *reduce* plusieurs changement sont effectués. Dans un premier temps nous modifions la valeur de retour afin qu'elle renvoi un couple (Text, DoubleWritable) *(ligne 0)*. Dans un second temps nous remplaçons notre précédente *ArrayList* contenant les commentaires par une variable *DoubleWritable total* qui nous permettra de calculer la somme total *(ligne 7)*.

```
0  public static class Reduce extends Reducer<Text, Text, Text, DoubleWritable> {
...    ...
6      ArrayList<String> cust = new ArrayList<>();
7      DoubleWritable total = new DoubleWritable(0);
...    ...
13       if (words.length == 2) cust.add(words[1]);
14       else total.set(total.get() + Double.parseDouble(words[0]));
15     }

17     for(String _cust : cust)
18       if(total.get() != 0) // on écrit que ceux qui ont fait des achats (optionnel)
19         context.write(new Text(_cust), total);
20   }
21 }
```

Listing 11 – Fonction reduce de Join

Exercice 7

✎ Donner la liste des clients (sans doublons) présents dans le dataset du répertoire input-groupBy :

La fonction *map* renverra tout les couples (Customer ID, Customer Name). C'est dans la fonction *reduce* que nous allons vérifié si l'ID d'un Customer n'as pas déjà été vue. Si elle ne l'est pas nous l'écrivons sinon on passe.

```
0  public static class Map extends Mapper<LongWritable, Text, Text, Text> {
1    private final static String emptyWords[] = { "" };
2
3    @Override
4    public void map(LongWritable key, Text value, Context context) throws
       IOException, InterruptedException {
5      String line = value.toString();
6      String[] words = line.split(",");
7
8      if(Arrays.equals(words, emptyWords))
9        return;
10
11     try {
12       String word = words[5]; // Customer ID
13       String word1 = words[6]; // Customer Name
14       if(word != "Customer ID")
15         context.write(new Text(word), new Text(word1));
16     }catch (Exception e){}
17   }
18 }
```

Listing 12 – Fonction map de GroupBy

```
0  public static class Reduce extends Reducer<Text, Text, Text, Text> {
1    private final static ArrayList<String> list = new ArrayList<>();
2
3    @Override
4    public void reduce(Text key, Iterable<Text> values, Context context) throws
       IOException, InterruptedException {
5
6      // Si l'ID n'as pas été vue
7      if(!list.contains(key.toString())) {
8        String line = "";
9        for (Text val : values) line = val.toString();
10
11       context.write(key, new Text(line));
12       list.add(key.toString());
13     }
14   }
15 }
```

Listing 13 – Fonction reduce de GroupBy

Exercice 8

✎ Donner le code SQL équivalent aux traitements Map/Reduce implémentés pour les questions 4, 5, 6 et 7 :

Question 4)

```sql
-- Par Date
SELECT OrderDate, State, SUM(Sales)
  FROM Superstore
    GROUP BY OrderDate, State;

-- Par Catégorie
SELECT OrderDate, Category, SUM(Sales)
  FROM Superstore
    GROUP BY OrderDate, Category;

-- Nombre de produits différents
SELECT OrderID, COUNT(ProductID)
  FROM Superstore
    GROUP BY OrderID;

-- Nombre total d'exemplaires.
SELECT OrderID, SUM(Quantity)
  FROM Superstore
    GROUP BY OrderID;
```

Question 5)

```sql
-- Couple (CUSTOMERS.name,ORDERS.comment)
SELECT c.name, o.comment
  FROM Customers c
    JOIN Orders o On c.custkey = o.custkey;
```

Question 6)

```sql
-- Montant total des achats faits par chaque client.
SELECT c.name, SUM(o.totalprice)
  FROM Customers c
    JOIN Orders o On c.custkey = o.custkey
      GROUP BY c.custkey, c.name;
```

Question 7)

```sql
-- Liste des clients (sans doublons)
SELECT CustomerID, CustomerName
  FROM Superstore
    GROUP BY CustomerID, CustomerName;
```

Exercice 9

✎ Donner un aperçu des trams et bus de la station OCCITANIE. Plus précisément, donner le nombre de (bus ou trams) pour chaque heure et ligne. Exemple : <Ligne 1, 17h, 30> (lire : à 17h, passent 30 tram de la ligne 1) :

Afin de simplifier l'affichage de la forme <Ligne X, Y h, nbTrams>, nous formons une clef de format '<Ligne X, Y h, ' tout en vérifiant que route_short_name soit égal à "OCCITANIE", nous mettons 1 en valeur afin de compter le nombre de bus/tram sur la ligne plus tard avec le reduce.

```
0  public static class Map extends Mapper<LongWritable, Text, Text, Text> {
1    private final static String emptyWords[] = { "" };
2
3    @Override
4    public void map(LongWritable key, Text value, Context context) throws
       IOException, InterruptedException {
5
6      String line = value.toString();
7      String[] words = line.split(";");
8
9      if(Arrays.equals(words, emptyWords))
10       return;
11
12     try{
13       String word = words[4]; // numLigne
14       String[] times = words[7].split(":"); // time
15       String time = times[0];
16       String txt = "<Ligne "+word+", "+time+"h,";
17       if(words[3].equals("OCCITANIE"))
18         context.write(new Text(txt), new Text("1"));
19     }catch (Exception e){}
20   }
21 }
```

Listing 14 – Fonction map de GroupBy

Nous pouvons calculer la somme sur les valeurs et finaliser notre clef de format précédente en ajoutant celle-ci, suivie d'un chevron fermant.

```
0  public static class Reduce extends Reducer<Text, Text, Text, Text> {
1    private static ArrayList<String> list = new ArrayList<String>();
2
3    @Override
4    public void reduce(Text key, Iterable<Text> values, Context context) throws
       IOException, InterruptedException {
5
6      double sum = 0;
7      for(Text val : values)
8        sum += Double.parseDouble(val.toString());
9
10     context.write(key, new Text(""+sum+">"));
11   }
12 }
```

Listing 15 – Fonction reduce de GroupBy

✎ Pour chaque station, donner le nombre de trams et bus par jour :

Nous stockons une clef différente dans le cas ou c'est un bus ou un tram afin de différencier les deux dans le reduce.

```
0  public static class Map extends Mapper<LongWritable, Text, Text, Text> {
1    private final static String emptyWords[] = { "" };
2
3    @Override
4    public void map(LongWritable key, Text value, Context context) throws
       IOException, InterruptedException {
5      String line = value.toString();
6      String[] words = line.split(";");
7
8      if(Arrays.equals(words, emptyWords)) return;
9
10     try {
11       String word = words[3]; // nomStation
12       String txt = "";
13
14       if(Integer.parseInt(words[4]) < 5)
15         txt = "<Station "+word+", X_tram =";
16       else
17         txt = "<Station "+word+", X_bus =";
18
19       context.write(new Text(txt), new Text("1"));
20     }catch (Exception e){}
21   }
22 }
```

Listing 16 – Fonction map de GroupBy

La fonction *reduce* reste inchangée.

✎ Pour chaque station et chaque heure, afficher une information X_tram correspondant au trafic des trams, avec X_tram="faible" si au plus 8 trams sont prévus (noter qu'une ligne de circulation a deux sens, donc au plus 4 trams par heure et sens), X_tram="moyen" si entre 9 et 18 trams sont prévus, et X="fort" pour toute autre valeur. Afficher la même information pour les bus. Pour les stations où il a seulement des trams (ou des bus) il faut afficher une seule information :

Même chose qu'au cas précédent mais en ajoutant l'heure dans la clef.

```java
public static class Map extends Mapper<LongWritable, Text, Text, Text> {
  private final static String emptyWords[] = { "" };

  @Override
  public void map(LongWritable key, Text value, Context context) throws
   IOException, InterruptedException {
    String line = value.toString();
    String[] words = line.split(";");

    if(Arrays.equals(words, emptyWords)) return;

    try {
      String word = words[3]; // nomStation
      String[] times = words[7].split(":"); // time
      String time = times[0];
      String txt = "";

      if(Integer.parseInt(words[4]) < 5)
        txt = "<Station "+word+", "+time+"h, X_tram =";
      else
        txt = "<Station "+word+", "+time+"h, X_bus =";

      context.write(new Text(txt), new Text("1"));
    }catch (Exception e){}
  }
}
```

Listing 17 – Fonction map de GroupBy

Il ne reste plus qu'à vérifier la valeur de sum afin de savoir si le trafic est faible, moyen ou fort.

```java
public static class Reduce extends Reducer<Text, Text, Text, Text> {
  private static ArrayList<String> list = new ArrayList<String>();

  @Override
  public void reduce(Text key, Iterable<Text> values, Context context)
    throws IOException, InterruptedException {

    int sum=0; for(Text val : values)
      sum += Integer.parseInt(val.toString());

    if(sum <= 8)
      context.write(key, new Text("faible>"));
    else if(sum <= 18)
      context.write(key, new Text("moyen>"));
    else
      context.write(key, new Text("fort>"));
  }
}
```

Listing 18 – Fonction reduce de GroupBy

✎ Optionnel : comment peut-on prendre en compte la direction des trams pour donner des informations plus précises ? :

Pour donner des informations plus précises sur la direction des tramways nous pouvons utiliser les colonnes *stop_name* et *trip_headsign*, nous pourrions ainsi déterminer le départ et les différentes stations traversées.

Exercice 10

✎ Trier les commandes clients du fichier superstore.csv (dans input-groupBy) par date d'expédition en ordre croissant, puis décroissant (créer deux classes différentes) :

Nous avons ajouté la fonction *compare* dans le fichier *GroupBy.java*, elle permet de comparer deux dates et renverra 0 si les dates sont équivalents, $x < 0$ si $a < b$ et $x > 0$ si $a > b$. Pour faciliter la gestion de la comparaison des dates nous avons importés les packages suivant :

```
import java.util.Date;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
```

Listing 19 – En-tête du fichier GroupBy

```
0  @Override
1  public int compare(WritableComparable a, WritableComparable b) {

3    // Spécification du format de date
4    DateFormat dateForm = new SimpleDateFormat("mm/dd/yyyy");

6    // Initialisation de deux dates
7    Date dateA = null;
8    Date dateB = null;

10   try {
11     // On parse les dates
12     dateA = dateForm.parse(a.toString());
13     dateB = dateForm.parse(b.toString());
14   } catch (ParseException e) {
15     e.printStackTrace();
16   }

18   // on compare les dates et on renvoie dans l'ordre croissant
19   return dateA.compareTo(dateB);
20 }
```

Listing 20 – Fonction compare de GroupBy

Pour afficher dans un ordre décroissant il suffit de modifier le return pour renvoyer la date la plus grande de la manière suivante :

```
0  @Override
1  public int compare(WritableComparable a, WritableComparable b) {
...    ...
17   return -dateA.compareTo(dateB);
18 }
```

Listing 21 – Fonction compare de GroupBy

✎ Trier les clients (identifiant+nom) par profit généré :

Au vue du nombre de modification pour la seconde partie de cette exercice, il nous as semblé plus judicieux de l'ajouter en annexe. Vous le retrouverez sous le nom suivant : *TriAvecComparaison.java*. Le résultat du premier job, visible dans le dossier input-tmp est la jointure de chaque utilisateurs et de son profit total. Le dernier job quant à lui, visible dans le dossier *output* est l'affichage dans l'ordre décroissant de la jointure précédente.

Exercice 11

✎ La classe TopkWordCount.java permet de compter les k mots les plus fréquents d'un fichier de texte. Pour évaluer des requêtes top-k la fonction reduce doit évoluer par rapport aux exemples vus précédemment.

1. La méthode reduce doit connaître la valeur du paramètre k donné en entrée. Le passage du paramètre se fait par une notion de contexte liée au job Map/Reduce. La méthode setup() permet de récupérer la valeur.
2. Les différents appels à la méthode reduce doivent être au courant des fréquences calculées par les autres méthodes. Pour cela, on utilise des variables globales (sortedwords et nsortedwords) dans la méthode reduce.
3. On doit attendre la fin de toutes les appels à la méthodes reduce pour écrire la sortie. Cela est fait par la méthode cleanup().

✎ Modifier la classe TopkWordCount.java pour répondre aux requêtes suivantes.

Comme précédemment, au vue du nombre de modifications, le fichier lié à cet exercice sera en annexe de ce rapport sous le nom : *TopkWordCount.java*

Exercice 12

✎ Quels sont les 10 stations les plus desservies par les tram sur la journée ?

Le code de cette question est disponible en annexe, *exo12.java*, voici le résultat obtenus :

- COMEDIE 726.0
- CORUM T1 649.0
- GARE ST-ROCH T1 726.0
- LOUIS BLANC 649.0
- MOULARES 713.0
- NOUVEAU ST-ROCH 526.0
- PL. DE L'EUROPE 649.0
- PORT MARIANNE 713.0
- RIVES DU LEZ T1 649.0
- RONDELET 526.0

✎ Quels sont les 10 stations les plus desservies par les bus ?

Pour répondre à cette question il suffit de modifier le code afin de récupérer les bus et non les tramways, comme suit :

```
...   ...
101   if(Integer.parseInt(words[4]) > 5) // BUS
...   ...
```

Listing 22 – Fonction Map de exo12

Voici le résultat obtenus :

- ANATOLE FRANCE 425.0
- BERTHELOT 337.0
- CHATEAU D'O 337.0
- GOUARA 348.0
- HOT. DEPARTEMENT 341.0
- LYCEE CLEMENCEAU 337.0
- PEYROU 337.0
- SAINT-DENIS 425.0
- SAINT-ELOI 322.0
- SAINT-GUILHEM 337.0

✎ Quels sont les 10 stations les plus desservies (tram et bus) ?

Pour répondre à cette question il suffit de supprimer la condition sur si c'est un bus ou un tram, et nous obtenons le résultat suivant :

- CHATEAU D'O 754.0
- COMEDIE 726.0
- GARE ST-ROCH T1 726.0
- LOUIS BLANC 649.0
- MOULARES 713.0
- OBSERVATOIRE 776.0
- PL. DE L'EUROPE 818.0
- PORT MARIANNE 713.0
- RONDELET 694.0
- SAINT-ELOI 749.0