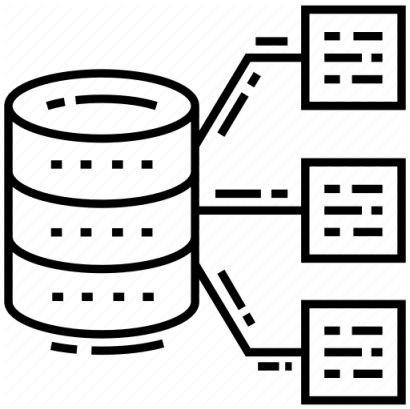
DEPARTEMENT INFORMATIQUE
DE LA FACULTE DES SCIENCES

Quentin Yeche (21520370), Yanis Allouch (21708237)

Rapport du TP N°1 Partie 1 : Hadoop & Map-Reduce



HMIN122M — Entrepôts de données et Big-Data

Référent: Federico Ulliana et Anne-Muriel Chifolleau

2020

Table des matières

Introduction

Nous étudions et proposons des solutions aux problèmes en JAVA8+ de Hadoop Map-Reduce rencontré sur différents jeux de données.
L'environnement de travail est composé de Eclipse qui est utilisé au développement et aux traitement des solutions sous Windows 10 et Ubuntu 20.04 LTS.

Le patch "Hadoop for Windows" proposé sur moodle à du être appliqué pour le développement Windows.

Préambule

Dans tous les exemples suivants nous avons en entrée de MAP le couple (n° de ligne, ligne). Puisque les fichiers sont tous en format CSV, on peut simplement effectuer un *ligne.split* avec le bon séparateur (généralement la virgule et occasionnellement | , soit une tabulation plus une barre verticale plus une tabulation) afin de récupérer dans un tableau les valeurs correspondant à chaque colonne. Le souci est qu'il peut arriver que le séparateur soit présent en tant que caractère dans la valeur d'une colonne. Nous verrons comment nous avons traité ce cas qui se présente dans l'exercice 4.

Pour gérer les cas où une lettre pourrait se glisser dans une colonne qui devrait contenir des nombres, on peut simplement faire un try/catch autour de l'appel à Double.parseDouble. On choisit alors d'ignorer la ligne (return dans le catch).

1 WordCount + Filter

Il suffit de reprendre l'exemple d'origine et de seulement écrire le nombre d'occurrences calculé s'il est supérieur à 2.

```
public void reduce(Text key, Iterable<IntWritable> values,
    Context context) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values){
                sum += val.get();
        }
        if(sum>=2){
            context.write(key, new IntWritable(sum));
        }
}
```

2 Group-By

En MR il est assez simple d'effectuer un *group by*. Le REDUCE effectue des opérations sur un ensemble de valeurs qui partagent une même clé. Si on choisit comme clé la colonne sur laquelle on souhaite faire un *group by* on a alors directement le résultat attendu. On aura bien traité les données séparément pour chaque valeur de la colonne concernée.

Nous remarquons que les données contiennent des champs de texte. Il est donc possible que ces champs contiennent malheureusement le séparateur du fichier. C'est effectivement le cas ici. On peut donc avoir deux approches possibles :

1. **Ignorer toutes les lignes** pour lesquelles le *ligne.split* renvoie un nombre de colonnes incohérent.
2. **Considérer que seules les colonnes de texte** peuvent potentiellement contenir le séparateur. Dans cas-là pour *superstore.csv* on peut tout de

même accéder aux quatre dernières colonnes puisqu'elles sont toutes des nombres, en indexant par rapport à la fin du tableau produit par *ligne.split*. Cet exemple est visible dans le fichier GroupByDateCategory.

2.1 Un opérateur de regroupement sur l'attribut <Customer-ID>

Signatures

```
num de ligne, ligne ->  MAP     ->  Customer ID, profit
Customer ID, profit ->  REDUCE  ->  Customer ID, somme
```

Comme on l'a dit en introduction de cette partie, on envoie comme clé la valeur de la colonne sur laquelle on *group by*, c'est-à-dire Customer ID. La valeur est le profit, qu'on somme dans le REDUCE.

2.2 Le montant des ventes par Date et State

Un *group by* sur plusieurs colonnes Date et State peut être vu comme un *group by* sur **une unique clé composite** <Date, State>.

Si l'on arrive à encoder plusieurs colonnes en une seule clé le tour est joué.

Il y a deux façons de combiner plusieurs valeurs en une clé composite :

1. **Les combiner en une seule variable**, probablement de **type Text**. On choisit un séparateur qu'il faudra gérer dans le REDUCE si on ne veut pas le retrouver en sortie. C'est le choix que nous faisons ici pour <Date,State>
2. **On crée une nouvelle classe** pour représenter la clé composite. Ici **Date et State seraient des attributs** de cette classe. Nous utiliserons cette approche pour la question suivante.

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

        if(!columns[20].equals("Profit")) {
                String s = columns[2] + "       |       " +
                    columns[10];
                context.write(new Text(s), new
                    DoubleWritable(Double.parseDouble(columns[20])));
        }
}
```

Nous modifions également la configuration d'Hadoop afin que le séparateur soit le même que celui que nous avons utilisé pour notre clé composite :

```
conf.set("mapred.textoutputformat.separator",
                "       |       ");
```

2.3 Le montant des ventes par Date et Category

Nous nous intéressons ici à la deuxième option pour les clés composites. Les types utilisés par Hadoop doivent implémenter l'interface *WritableComparable* (combinaison des interfaces *Writable* et *Comparable*). Voici les méthodes que nous devons fournir pour les interfaces que nous implémentons :

1. ***Writable***. Cette interface assure et définit comment un objet de la classe peut être transmis par des fichiers. C'est effectivement nécessaire pour permettre la parallélisation inhérente à MR. Voici les méthodes :
   a) *write*. Cette méthode définit comment un objet de la classe peut être sauvegardé (écrit) dans un fichier. Ici on fait simplement un appel à *writeUTF* sur les attributs de notre classe.
   b) *readFields* pour la lecture depuis un fichier. On utilise la méthode *readUTF* pour récupérer et initialiser les valeurs de nos attributs.
2. ***Comparable***. Bien que ce ne soit pas strictement nécessaire dans ce cas, la comparaison des clés permet dans ce cas de les trier. Cela permet alors une distribution des taches REDUCE plus facile. On définit donc une méthode *compareTo* pour comparer deux clés. On a choisi ici l'ordre lexicographique de la concaténation de Date et State. Mais si nous voulions réellement ordonner nos résultats, par exemple par Date, il serait simple de le faire. Il faudra simplement extraire de la String Date une représentation d'une date qui permette la comparaison, comme par exemple la classe java.util.Date.

Enfin, bien que ça ne soit pas requis par Hadoop, il est nécessaire d'implémenter la méthode *toString* pour renvoyer les deux attributs, séparés par le séparateur de notre choix. Autrement l'appel à *write* dans REDUCE conduira simplement à l'écriture de la représentation canonique de notre clé composite, c'est-à-dire la référence de l'objet.

2.4 Le nombre de produits différents (distincts) achetés, ainsi que le nombre total d'exemplaires

Signatures

```
num de ligne, ligne ->  MAP     ->  Order ID, Quantity\\
Order ID, Quantity  ->  REDUCE  ->  Order ID, <compteur,somme>
```

Nous faisons ici l'hypothèse raisonnable que les couples (Order ID, Product ID) sont uniques. Cela nous permet de nous affranchir d'envoyer l'information Product ID qui serait nécessaire pour éviter de compter un produit plusieurs fois s'il apparaît plusieurs fois dans une même commande.

Dans le MAP en sortie la clé est Order ID (colonne sur laquelle le *group by* porte). La valeur est Quantity. Dans le REDUCE on compte le nombre de valeurs pour avoir le nombre de produits différents, et la somme des valeurs pour

le nombre total de produits. On peut ensuite écrire ces deux valeurs comme une seule valeur en les concaténant.

3 Join

L'utilisation d'un JOIN en MR est facilitée par le fait que le programme prend en entrée tous les fichiers présents dans le dossier spécifié. Il suffit donc de mettre les fichiers représentant les deux tables dans le même dossier.

Le point important va alors être de différencier entre les donneés de ORDERS et celles de CUSTOMERS. Dans le MAP on peut différencier grâce au nombre de colonnes. Nous marquons avec un token ",&" les String des valeurs de CUSTOMERS (choix arbitraire) envoyées par MAP pour que REDUCE puisse facilement les différencier.

Dans le REDUCE on doit d'abord vider l'itérateur et remplir un tableau pour permettre la jointure par une double boucle for. Pour améliorer l'efficacité de l'algorithme nous choisissons de différencier (les mettre dans deux listes différentes) les valeurs au moment où nous vidons l'itérateur plutôt que lors de la jointure. Cela nous permet de tester chaque valeur une seule fois. On passe donc d'une taille $(n+m)^2$ à $(n*m)$ pour le nombre de de tours de boucle, où et n et m sont le nombre de valeurs pour chaque table.