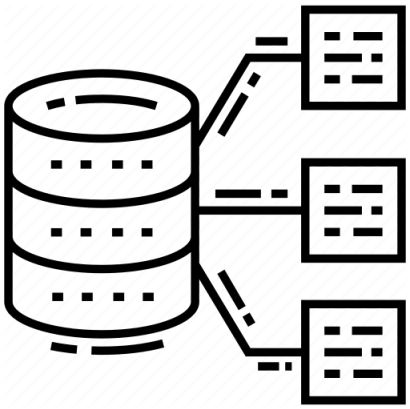
DEPARTEMENT INFORMATIQUE
DE LA FACULTE DES SCIENCES

Quentin Yeche (21520370), Yanis Allouch (21708237)

Rapport du TP N°1 Partie 2 : 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 GroupBy + Join

Il suffit de reprendre notre code et de seulement changé nos index paramétré par les variables **keyIndex** et **valueIndex**, prenant respectivement les valeurs 1 et 3 pour le fichier *Orders*, 0 et 1 pour le fichier *Customers*.

Le dernier paramètre de la classe *REDUCE* à été adapté en conséquence vers le type ***DoubleWritable***.

2 Suppression des doublons (DISTINCT)

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.

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

        Text value = values.iterator().next();
        context.write(key, new Text(value));
}
```

2.1 MR <-> SQL

Dans cette partie il nous aient demandés de traduire nos opérations Map/-Reduce implémentés pour la question 4, 5 du TP1 partie 1 ainsi que les deux premières implémentations de ce rapport.

Voici les questions ayant servi aux implémentations :

4. a) Calculer le montant des ventes par Date et State.
   b) Calculer le montant des ventes par Date et Category.
   c) Pour chaque commande calculer 1) le nombre de produits différents (distincts) achètes, ainsi que 2) le nombre total d'exemplaires.
5. Joindre les lignes concernant les informations des clients et des commandes contenus dans le répertoire input-join. La jointure doit être réalisée sur l'attribut custkey.
6. calculer le montant total des achats faits par chaque client.
7. Donner la liste des clients (sans doublons) présents dans le dataset du répertoire input-groupBy

Et leur traductions en SQL :

4. Le **group by**
   a) composé

```sql
SELECT  SUM(s.Profit)
FROM    SUPERSTORE s
GROUP   BY (s.OrderDate, s.State);
```

   b) composé n°2

```sql
SELECT  SUM(s.Profit)
FROM    SUPERSTORE s
GROUP   BY (s.OrderDate, s.Category);
```

   c) composé n°3

```sql
SELECT  COUNT(DISCTINCT s.ProductID),
    COUNT(s.Quantity)
FROM    SUPERSTORE s
GROUP   BY (s.OrderID, s.Category);
```

5. La **jointure**

```sql
SELECT  *
FROM    ORDERS o
JOIN    CUSTOMERS c ON c.custkey = o.custkey;
```

6. La **jointure** + **group by**

```sql
SELECT  SUM(o.totalprice)
FROM    ORDERS o
JOIN    CUSTOMERS c ON c.custkey = o.custkey
GROUP   BY (c.custkey);
```

7. Le **distinct**

```sql
SELECT  DISTINCT s.CustomerID
FROM    SUPERSTORE s;
```

3 La TAM et les offres du jour

Le jeu de donnée nommé TAM_MMM_OffreJour.zip se trouve à l'adresse http://data.montpellier3m.fr/dataset/offre-de-transport-tam-en-temps-reel.

Nous ré-utilisons toutes les techniques abordées précédemment pour pouvoir donner un aperçu des trams et bus

Plus précisément :

- de la station OCCITANIE pour 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).
- on souhaite aussi, pour chaque station, donner le nombre de trams et bus par jour.
- 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. Optionnel : comment peut-on prendre en compte la direction des trams pour donner des informations plus précises ?

Nous avons une utilisé une première clé composé pour représenter : une ligne et l'heure au format *String*. Et utilisons la méthode *toString()* pour commencer à formater l'affichage convenu.

Par rapport à la fonction *MAP*, on ne change que peu de chose, il n'est nécessaire que de vérifier que notre station soit bien "Occitanie", ainsi que crée une clé composé nos colonnes 4 et 7, respectivement le numéro de la ligne et l'heure associé. Enfin le *REDUCE* ne fait que retourner la cardinalité de notre ensemble de valeurs.

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",  " ");
```

Pour traiter le second cas, nous n'avons besoin de rien d'autre que de différencier les bus et trams dans le *REDUCE*. Qui se traduit par un *switch case* sur notre ensemble de valeurs :

```
int cptBus = 0;
int cptTram = 0;
for(IntWritable ittVal : values){
        int val = ittVal.get();
        switch (val){
        case 1:
        case 2:
```

```
        case 3:
        case 4:
                cptTram++;
                break;
        default:
                cptBus++;
                break;
        }
}
context.write(key, new Text(cptBus + "\t" + cptTram));\textbf{}
```

Nous voici au dernier traitement, qui consiste a faire un affichage des fréquences en fonction du nombre d'aller et retour des trams ou bus, et de conditionner l'affichage si une des valeurs venait a manquer.

Il a été nécessaire d'utiliser une Clé composite, et une Valeur composite, décrite de la façon suivante :

```
class CompositeKey3 implements
    WritableComparable<CompositeKey3>{
        public String station;
        public String heure;
        ...
        public String toString() {
                return station + "\t" + heure;
        }
}
```

```
class CompositeValue implements
    WritableComparable<CompositeValue>{
        public int type;
        public int direction;
        ...
}
```

Il ne nous reste plus qu'a reprendre le code précédent, instancier notre nouvelle clé de la façon suivante :

```
CompositeKey3 ckey = new
    CompositeKey3(columns[3],columns[7].substring(0,2));
```

Nos valeur instancier de la façon suivante :

```
context.write(ckey, new CompositeValue(type,
    Integer.parseInt(columns[6])));
```

Et notre fonction *MAP* transporte alors toute l'information nécéssaire au traitement du *REDUCE*.

Le *REDUCE* compte le nombre de bus, tram, et la fréquence tel que spécifier quelques lignes avant.