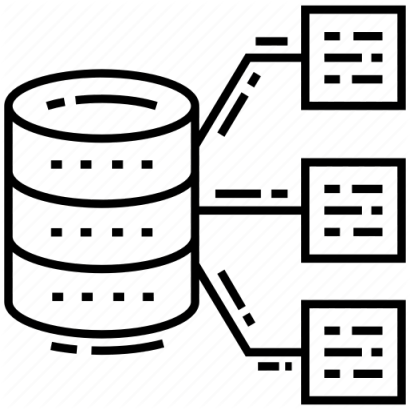
DEPARTEMENT INFORMATIQUE
DE LA FACULTE DES SCIENCES

Quentin Yeche (21520370), Yanis Allouch (21708237)

Rapport du TP N°1 Partie 3 : Hadoop & Map-Reduce



HMIN122M — Entrepôts de données et Big-Data

Référent: Federico Ulliana et Anne-Muriel Chifolleau

2020

Table des matières

1 Tri **3**
1.1 Par date d'expédition 3

2 Requêtes Top-k **3**
2.1 Triées par profit descendant 4
2.2 Clients en terme de profit réalisé 4

3 TAM (suite) **4**

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 pour le développement et 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 certains cas.

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 des l'appel à *Double.parseDouble* ou *Integer.parseInt* . On choisit alors d'ignorer la ligne (rien ou un return dans le catch, selon, selon la situation).

1 Tri

En nous basant sur le code fourni et étant donné que la classe Date de Java n'est pas adapté pour notre travail, nous avons défini une classe *DateWritable* implémentant *WritableComparable* et ayant comme attribut ***annee***, ***mois*** et le ***jour***.

1.1 Par date d'expédition

Pour définir un ordre sur les objets d'une classe il faut implémenter l'interface *Comparable*, c'est-à-dire fournir une méthode *compareTo*. Un appel à *a.compareTo(b)* devra renvoyer un entier strictement négatif si $b > a$, strictement positif si $a > b$ et 0 sinon (ce qui siginifie que $a = b$ seulement si l'ensemble est totalement ordonné, ce qui est le cas ici). On implémente donc l'ordre usuel des dates, en testant l'année, puis le mois et enfin le jour.

En ordre croissant

L'ordre naturel est croissant, notre travail s'arrête donc là.

En ordre décroissant

Pour pouvoir réaliser ce travail en ordre décroissant, il faut ajouter la classe Java *InverseComparator* et définir l'opération de comparaison inverse sur le contrat WritableComparable, notre classe respectant alors le contrat possède un opérateur de comparaison inverse. Une fois spécifié dans la configuration du job, il n'y a plus rien a faire.

2 Requêtes Top-k

Le code fourni, effectue déjà le traitement nécessaire pour répondre partiellement aux problèmes suivant. Vous trouverez le détail dans les sections correspondantes.

2.1 Triées par profit descendant

Le vrai travail se fait au niveau du *REDUCE*. Pour pouvoir ne garder que les k lignes de plus grand profit, on utilise donc le profit comme clé pour la *TreeMap* utilisée.

2.2 Clients en terme de profit réalisé

La réponse à cette requête consiste simplement à combiner l'approche *TopK* avec un *group by*. Ce dernier se fait en utilisant *Customer ID* comme clé en sortie du *MAP*. Dans le *REDUCE* on calculera le profit total en itérant sur les valeurs, avant d'insérer les paires (*Customer ID, profit*) dans la *TreeMap*. A noter ici que nous avons décidé d'implémenter une classe *BigDecimalWritable*, qui est un wrapper pour la class BigDecimal. En effet utiliser des nombres décimaux nous paraît plus adapté que des *flottants* (ou *double*) et les problèmes liés à leur représentation.

3 TAM (suite)

Les trois questions sont très similaires. Le fichier d'entrée porte sur la journée, il n'y a donc rien à faire sur ce front. Ce qui différencie les trois questions est un simple test sur la valeur de la colonne *route_short_name* :

1. les 10 stations les plus desservies par les trams : on émet un message seulement si la valeur est inférieure à 4 ;
2. les 10 stations les plus desservies par les bus : on émet un message seulement si la valeur est supérieure à 5 ;
3. les 10 stations les plus desservies (tram et bus) : on émet un message quelque soit la valeur.

Le reste de l'implémentation ressemble à la question précédente. Il faut évidemment faire un *group by* sur le nom de l'arrêt. Le contenu de la valeur émise par le *MAP* n'est pas utile (comme dans *WordCount*) puisque c'est le nombre de valeurs qui importe, et non les valeurs en elle-même. La *TreeMap* a donc comme clé le nombre de valeurs (calculé en préambule du *REDUCE*), et comme valeur le nom de l'arrêt.