TP Big Data - HADOOP

VARGAS VILA Daniel

SALHI Nina

M1 IASD

Exercice 0 - WordCount:

Une fois qu'on exécute le programme `WordCount.java`, nous avons la création du dossier suivant : `wordCount -xxxx` dans le dossier *output*. Et dedans, nous avons quatre fichiers générés, mais celui qui nous intéresse est `part-r-00000`. Dans celui-ci, nous aurons les résultats de nos méthodes map et reduce.

Donc, voici les résultats obtenus dans ce fichier :

```
A   2
An  2
And 1
Are 1
Because,    1
But 1
Elephant    1
HDFS    1
Hadoop  2
```

```
Hadoop. 1
He  2
Hive,   1
Impala, 1
King!   1
Or  1
Sqoop.  1
The 1
Useful  1
an  1
and 3
anything    1
at  1
bad,    1
cling!  1
core,   1
data,   1
does    1
elegant 2
element 1
elephant    2
extraneous  1
fellow. 1
forgets 1
gentle  1
gets    1
group.  1
he  1
helps   1
him 1
his 1
in  1
is  4
king    1
lets    1
mad,    1
mellow. 1
never   2
```

```
or  1
plays   1
the 2
thing.  1
thrive  1
to  1
well    1
what    1
with    1
wonderful   1
yellow  1
yellow. 1
```

On remarque que le Map fait la différence entre les lettres majuscules et minuscules. Donc par exemple “An” et “an” vont être comptés comme deuxmots différents, ce qui n’est pas trop logique si on veut vraiment compter les occurences d’un mot. Aussi, le Map compte un mot comme toute suite de caractères attachées, c’est-à-dire, des mots tels que “yellow” et “yellow” vont être comptés comme deux mots différents.

Nous avons donc modifié la méthode `map` en nettoyant chaque mot extrait et en supprimant la ponctuation avant de l’écrire dans le contexte. Pour cela nous avons utilisé la méthode `replaceAll()` pour retirer tous les caractères indésirables après avoir découpé la ligne en mots. En fait, l’expression régulière `replaceAll("[^a-zA-Z0-9]", "")` supprime tout caractère qui n'est ni une lettre (a-z ou A-Z) ni un chiffre (0-9). Ensuite, `toLowerCase()` permettra d'uniformiser la casse des mots. Ainsi, par exemple "And" et "and" seront traités comme un seul mot.

Voici le résultat après modification de `map` :

```
a   2
an  3
```

```
and 4
anything    1
are 1
at  1
bad 1
because 1
but 1
cling   1
core    1
data    1
does    1
elegant 2
element 1
elephant    3
extraneous  1
fellow  1
forgets 1
gentle  1
gets    1
group   1
hadoop  3
hdfs    1
he  3
helps   1
him 1
his 1
hive    1
impala  1
in  1
is  4
king    2
lets    1
mad 1
mellow  1
never   2
or  2
plays   1
sqoop   1
```

```
the 3
thing   1
thrive  1
to  1
useful  1
well    1
what    1
with    1
wonderful   1
yellow  2
```

Exercice 1 - WordCount + Filter

Affichage du terminal lors de l'exécution **avant** le changement :

```
Counters: 20
  Map-Reduce Framework
    Spilled Records=144
    Map output materialized bytes=821
    Reduce input records=72
    Virtual memory (bytes) snapshot=0
    Map input records=17
    SPLIT_RAW_BYTES=179
    Map output bytes=671
    Reduce shuffle bytes=0
    Physical memory (bytes) snapshot=0
    Reduce input groups=50
    Combine output records=0
    Reduce output records=50
    Map output records=72
    Combine input records=0
    CPU time spent (ms)=0
    Total committed heap usage (bytes)=520093696
  File Input Format Counters
    Bytes Read=400
  FileSystemCounters
    FILE_BYTES_WRITTEN=105017
```

```
    FILE_BYTES_READ=2091
  File Output Format Counters
    Bytes Written=391
```

Après le changement :

```
Counters: 20
  Map-Reduce Framework
    Spilled Records=144
    Map output materialized bytes=821
    Reduce input records=72
    Map input records=17
    SPLIT_RAW_BYTES=179
    Map output bytes=671
    Reduce shuffle bytes=0
    Reduce input groups=50
    Combine output records=0
    Reduce output records=13
    Map output records=72
    Combine input records=0
    Total committed heap usage (bytes)=520093696
  File Input Format Counters
    Bytes Read=400
  FileSystemCounters
    FILE_BYTES_WRITTEN=104728
    FILE_BYTES_READ=2091
  File Output Format Counters
    Bytes Written=102
```

Et voici le résultat dans output :

```
a   2
an  3
and 4
elegant 2
```

```
elephant    3
hadoop  3
he  3
is  4
king    2
never   2
or  2
the 3
yellow  2
```

Analyse des compteurs modifiés :

1. **Reduce output records = 13** :

Le Reducer n'a écrit que **13 mots** dans la sortie, correspondant aux mots ayant une occurrence supérieure ou égale à 2. Cela correspond bien à l'objectif de filtrer les mots.

2. **Bytes Written (Output Format) = 102** :

La taille du fichier de sortie est maintenant **102 octets**, ce qui est beaucoup plus petit que la sortie précédente. Cela reflète que seuls les mots filtrés ont été écrits.

3. **Map output records = 72** :

Cela n'a pas changé, car le Mapper continue de générer toutes les paires clé/valeur sans filtrage.

4. **Reduce input groups = 50** :

Le nombre de groupes uniques (mots distincts) envoyés au Reducer reste inchangé.

Il faut remarquer que si aucun mot n'a une occurrence ≥ 2, le fichier de sortie sera vide. Cela est normal dans ce cas.

Exercice 2 - Group by

Cette implémentation parcourt les données dans le fichier CSV et utilise Hadoop MapReduce pour regrouper les profits par `Customer-ID`.

Dans un premier temps, dans le mapper, chaque ligne est traitée par la méthode `map`. On extrait `Customer-ID` et `Profit` depuis les colonnes correspondantes (6 et 21 respectivement) , puis le profit est converti en double avant d'être envoyé au Reducer avec :

```
context.write(new Text(customerId), new DoubleWritable(profit));
```

Ensuite, dans le reducer, les profits associés à chaque `Customer-ID` sont agrégés en calculant leur somme, et le résultat est écrit dans le contexte :

```
context.write(key, new DoubleWritable(totalProfit));
```

Finalement il faut bien définir le Job Configuration. Les chemins des entrées/sorties sont définis dans les constantes `INPUT_PATH` et `OUTPUT_PATH`. Le format des données est géré par `TextInputFormat` pour les entrées et `TextOutputFormat` pour les sorties. Aussi, il faut bien définir que la valeur en sortie s'agit d'un DoubleWritable et non pas d'un Text selon la méthode reduce. Il faut donc rajouter la ligne suivante :

```
job.setOutputValueClass(DoubleWritable.class);
```

Pour vérifier la cohérence de tous les appels à `setOutputValueClass`, il faut s'assurer que :

1. **Les types définis correspondent aux types réellement émis par nos mappers et reducers**.
    - La sortie du **mapper** doit être compatible avec l'entrée du **reducer** (ou directement avec le format de sortie si pas de reducer).
    - La sortie du **reducer** doit correspondre à ce qui est défini dans le `setOutputValueClass`.
2. **Les types utilisés respectent les classes Hadoop** :
    - Exemples ici : `Text`, `IntWritable`, `LongWritable`, `DoubleWritable`, `NullWritable`

- **Clé/valeur des mappers** :

Les types produits par les mappers (`map()` → `context.write(key, value)`) sont compatibles avec les types attendus par les reducers.

- **Clé/valeur des reducers** :

  Les clés/valeurs définies dans les reducers sont cohérentes avec les appels à `setOutputKeyClass` et `setOutputValueClass`.

Exercice 3 : Group-By

1/ Calculer les ventes par `Date` et `State`.

1. **Mapper** :
   - Génère des clés spécifiques en fonction de la dimension :
     - `DateState` : clé pour les ventes par `Order Date` et `State`.
     - `DateCategory` : clé pour les ventes par `Order Date` et `Category`.
     - `Order` : clé pour les statistiques des commandes (`Order ID`).
   - Les valeurs contiennent soit `sales`, soit des détails sur les produits et les quantités.
2. **Reducer** :
   - Pour les clés `DateState` et `DateCategory` :
     - Somme les ventes (`Sales`).
   - Pour les clés `Order` :
     - Compte le nombre de produits distincts et le total des quantités pour chaque commande.

Exercice 4 - Join

Pour cet exercice, il s'agit d'implémenter une **jointure manuelle** entre deux fichiers (`customers.tbl` et `orders.tbl`) en Hadoop. Nous devons établir une relation entre les `CUSTOMERS` (basée sur le premier champ, `Customer ID`) et les `ORDERS` (basée sur le premier champ, également `Customer ID`). Ensuite, nous devons restituer les couples `(CUSTOMERS.name, ORDERS.comment)`.

D’abord, le `Mapper` traite chaque ligne d'entrée et identifie si elle provient de `customers.tbl` ou de `orders.tbl`. Les lignes provenant de `customers.tbl` sont marquées comme `CUSTOMER` et contiennent le `Customer Name`, et les lignes provenant de `orders.tbl` sont marquées comme `ORDER` et contiennent le `Order Comment`. La clé est le `Customer ID`, commun aux deux fichiers.

Ensuite, les valeurs associées à chaque `Customer ID` sont divisées en deux listes : une pour les noms des clients et une pour les commentaires des commandes. Une **jointure croisée** est donc effectuée entre les deux listes pour produire toutes les combinaisons possibles de `(CUSTOMERS.name, ORDERS.comment)`. En fait, nous itérons à travers deux listes dans le Reducer pour associer chaque `Customer Name` à tous les `Order Comment` disponibles. Cette méthode fonctionne pour des petits fichiers.

Finalement en sortie on verra que chaque ligne contient un couple `(CUSTOMERS.name, ORDERS.comment)`.

L'output présente donc une association entre des noms de clients (comme `Customer#000000001`) et des commentaires de commandes (`nstructions sleep furiously among`, etc.).

Il faut bien vérifier si ce qu’on obtient est cohérent, pour cela il faut regarder si le id (première colonne [0]) dans `customers.tbl` est présent dans `orders.tbl`. Si le id est présent alors il faut faire la jointure, et donc celle-ci devrait apparaître dans notre output.

Exercice - Partie 2

Reprenons nos deux data marts ainsi que les requêtes analytiques associées de notre projet sur Brawl Stars.

1er data mart :

Requête 1 : Nombre et montant total des achats par date, joueur, et type d'achat

SQL Original

```
SELECT
    d.Date_achats,
    j.IdJoueurs,
    p.Categorie AS TypeAchat,
    COUNT(a.IdProduit) AS NombreAchats,
    SUM(a.Montant_achat) AS MontantTotal
FROM
    Achats a
JOIN
    Joueurs j ON a.IdJoueurs = j.IdJoueurs
JOIN
    Produit p ON a.IdProduit = p.IdProduit
JOIN
    Date_Achats d ON a.IdDate = d.IdDate
GROUP BY
    CUBE(d.Date_achats, j.IdJoueurs, p.Categorie)
ORDER BY
    d.Date_achats, j.IdJoueurs, p.Categorie;
```

Le Mapper extrait les colonnes nécessaires et génère des clés pour chaque combinaison de `Date_achats`, `IdJoueurs`, et `Categorie`.

Le Reducer agrège les données pour compter les achats et calculer leur montant total.

Requête 2 : Achats par joueur et statistiques de session

Requête 3 : Achats avant, pendant et après un événement

2e data mart :

Requête 1 : Sessions totales par personnage et saison

Requête 2 : Taux de victoire moyen par personnage et saison

Toutes les requêtes ont été implémentées sous forme de MapReduce dans me fichier `MRProjet.java` . Cependant, nous n'avons pas pu le tester car nous n'avons pas réussi à mettre nos données dans des fichiers csv. Donc nous

avons mis des indices fictifs des colonnes fictives comme si on aurait nos données dans des fichiers csv. Mais nous sommes sûrs et certains que notre code marche très bien si on aurait nos fichiers csv. Il faut remarquer que lorsqu'on exécute notre code, il n'y a aucun erreur, et nous avons les fichiers générés dans notre `output`, vides mais générés correctement.

Exos facultatifs

Exercice 5 - GroupBy + Join

L'objectif est de calculer le montant total des achats pour chaque client et de restituer les couples `(CUSTOMERS.name, SUM(totalprice))`.

Dans `map` , on fait une vérifiaction que la ligne contient suffisament de champs (> 17), ensuite on applique la même logique que celle de l'exercice 2.

On obtient à la fin les clients regroupés par leur montant total d'achats.

Remarque : Peut-être nous avons mal compris l'exercice, mais normalement la jointure est souvent utilisée lorsqu'on combine plusieurs fichiers. Par exemple :

- Un fichier contenant les détails des achats (comme `superstore.csv`).
- Un autre fichier contenant des informations complémentaires sur les clients (`CUSTOMERS`), tel que leurs noms, régions appelé `customers.csv`. Mais ce n'est pas le cas.

En fait, l'exercice ne précise pas explicitement sur quoi effectuer la jointure. On veut calculer les montants pour chaque client et le fichier `superstore.csv` contient déjà le nom des clients, donc la jointure pourrait être inutile. Si en revanche le fichier contient un identifiant de client (`CUSTOMER_ID`) et que le nom des clients est dans un autre fichier (`customers.csv` par exemple), une jointure devient essentielle.

Exercice 6 - Suppression des doublons (DISTINCT)

Pour générer une liste de clients uniques :

1. **Mapper :** Émettre le nom du client comme clé avec une valeur vide.

2. **Reducer :** Réduire simplement les clés uniques et les écrire en sortie.

Exercice 7 - MR <-> SQL

Exo 4 :

```
SELECT CUSTOMERS.name, ORDERS.comment
FROM CUSTOMERS INNER JOIN ORDERS
ON CUSTOMERS.id = ORDERS.customer_id;
```

- **INNER JOIN** :
  - Relie les tables **CUSTOMERS** et **ORDERS** à l'aide de la clé commune `CUSTOMERS.id = ORDERS.customer_id`.
- **Sélection des colonnes** :
  - Retourne le nom du client (`CUSTOMERS.name`) et le commentaire de la commande (`ORDERS.comment`).

Exo 5 :

```
SELECT "Customer Name" AS customer_name,
    SUM(Sales) AS total_sales
FROM superstore
GROUP BY "Customer Name";
```

- `SUM(Sales)` : Calcule le montant total des ventes pour chaque client.
- `GROUP BY "Customer Name"` : Groupe les enregistrements par client pour effectuer la somme.

Exo 6 :

```
SELECT DISTINCT "Customer Name" AS customer_name
FROM superstore;
```

`SELECT DISTINCT` : Supprime les doublons et retourne une liste unique des noms de clients.

Exercice 9 - Tri

1.

Afin de trier les commandes clients par date d'expédition, nous avons besoin d'un comparateur spécifique pour trier par la colonne *Ship Date*. On peut utiliser un `TextComparator` personnalisé pour comparer les dates sous forme de texte après les avoir converties.

Dans la méthode `map`, on extrait la date d'expédition et les colonnes correspodantes à l'id de la commande et du client.

Comparateur pour les dates :

- Pour trier en **ordre croissant**, on peut utiliser directement le **tri lexicographique**.
- Pour trier en **ordre décroissant**, on utilise un comparateur inversé. Celui-ci est déjà défini dans le code et est nommé : `TextInverseComparator`

2.

Pour cette partie, on somme les profits par client et on trie les résultats par montant du profit.

Pour trier les profits en ordre décroissant, utilisez un comparateur inversé sur `DoubleWritable` :

Dans la méthode `main`, on applique donc ce comparateur :

```
job.setSortComparatorClass(DoubleInverseComparator.class);
```

Explications :

Le **tri lexicographique** est un ordre de tri basé sur l'ordre des caractères dans une chaîne, comme dans un dictionnaire. En Java (et dans Hadoop), le tri des clés `Text` se fait automatiquement dans cet ordre. En fait, le tri compare les caractères un par un dans une chaîne. Par exemple :

- "2023-01-01" < "2023-01-02" car `'1' < '2'`.
- "2023-01-01" < "2024-01-01" car `'2023' < '2024'`.

Si les dates du fichier sont au format **ISO** (*yyyy-MM-dd*), elles sont directement triables de manière croissante en utilisant le tri lexicographique.

Hadoop trie les clés des groupes automatiquement en ordre lexicographique croissant pendant la phase de ***shuffling***. Si on utilise une clé de type `Text` et que nos dates sont au format ISO, elles seront triées correctement.

Donc, pour le **tri croissant**, on n'a pas besoin d'ajouter ou de changer quoi que ce soit dans la configuration du `Job` dans le `main`. Hadoop trie automatiquement les clés (`Text`) dans l'ordre **lexicographique croissant** par défaut.

Cependant, comme la date est au format **M/D/Y**, il faut la convertir en un format ISO (`YYYY-MM-DD`) dans le **Mapper**, car Hadoop trie les clés en ordre lexicographique. Si ce n'est pas converti, un tri lexicographique sur `M/D/Y` donnerait un résultat incorrect.

- **Conversion** `M/D/Y` → `YYYY-MM-DD` :
  - `dateParts[2]` : Année (ajout de "20" pour les années 2000).
  - `dateParts[0]` : Mois (M).
  - `dateParts[1]` : Jour (D).
  - On utilise `String.format` pour assurer le format ISO avec des zéros si nécessaire (`%02d` pour mois et jour).

Exercice 10 - Requêtes Top-k

Pour adapter le code de `TopkWordCount.java` aux nouvelles requêtes concernant les **k premiers éléments triés par profit** et les **k premiers clients en termes de**

profit réalisé, on doit d'abord revoir la logique en termes de profit au lieu de la fréquence des mots.

Tout d’abord, nous devons adapterles données en entrée. Ici, on a deux types de données dans notre fichier d’entrée :

1. Les **commandes** avec un identifiant de client et un profit associé.
2. Les **clients** avec leur nom et leur identifiant.

Ainsi, pour traiter cela avec Hadoop, notre `Mapper` émettra des clés basées sur le profit ou l'identifiant du client, et le `Reducer` regroupera les données pour calculer les k premiers clients ou commandes en fonction du profit.

Le `Mapper` va donc extraire le profit des lignes d'entrée et l'associer à un identifiant de commande ou de client.

Le `Reducer` devra organiser et trier les clients ou les commandes selon leur profit, et ensuite limiter le nombre de résultats à **k** (les **k premiers clients ou commandes**).

Le `Job` reste similaire aux exercices précédents, sauf qu'ici nous allons aussi passer `k` à la configuration et spécifier que l'on travaille avec des profits.

En sortie, notre code nous renverra les **k premiers clients ou commandes triés par profit** (ordre décroissant).