université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

lundi 28 novembre 2022

Traitement distribué de données
M1 ISD

TP nº 3

Exercice 0

Cet exercice consiste à configurer son compte pour pouvoir utiliser Hadoop.

1. Éditez le fichier `~/.ssh/config` de manière à ce qu'il contienne les lignes :

   ```
   AddressFamily inet
   Host 0.0.0.0 127.0.0.1 localhost ip6-localhost
        StrictHostKeyChecking no
        UserKnownHostsFile=/dev/null
   ```

2. Exécutez la commande suivante : `ssh localhost`. Si vous ne pouvez pas vous logguer sans mot de passe, effectuez l'opération suivante :

   ```
   ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
   cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
   ```

   puis vérifier que la commande `ssh localhost` fonctionne sans problème. Cette commande vous connecte, via le réseau à la machine locale (un nouveau shell est démarré dans votre terminal). Quittez ce dernier en faisant ctrl-D ou `exit`.

3. Ajouter à la fin de votre fichier `~/.bashrc` la ligne :

   ```
   source /public/kn/setup-kn.sh
   ```

   et relancer le terminal

4. Exécutez la commande suivante : `mkdir -p ~/.hadoop_logs`

5. Vérifier si des fichiers temporaires `hadoop` existent :

   ```
   ls -d /tmp/hadoop*
   ```

   Si cette commande affiche des répertoires dont le nom contient votre login Unix, les effacer :

   ```
   rm -rf /tmp/hadoop*
   ```

6. Formatter le HDFS

   ```
   hdfs namenode -format
   ```

7. Démarrer HDFS

   ```
   start-dfs.sh
   ```

   Vérifier que HDFS est démarré en vous rendant à l'URL `http://localhost:9870` et contrôler que la ligne `Configured Capacity` n'indique pas 0 octets.

8. Créer une arborescence de répertoire sur HDFS et y déposer les fichiers textes du TP (on suppose que le répertoire courant contient les fichiers texte). Il faut remplacer VOTRELOGIN par votre login Unix.

   ```
   hdfs dfs -mkdir  -p /user/VOTRELOGIN/input/
   hdfs dfs -put *.txt /user/VOTRELOGIN/input/
   ```

9. Vérifier que les fichiers sont bien présents :

   ```
   hdfs dfs -ls /user/VOTRELOGIN/input/*
   ```

Exercice 1

Dans cet exercice, on va implémenter un comptage de mot, c'est à dire utiliser Map/Reduce pour compter de manière distribuée [1] le nombre d'occurrences de chaque mots dans un texte.

1. Importez le projet Eclipse sur la page du cours
2. Lisez attentivement la classe `DriverWordCount`. La transformation utilise un *map* et un `reduce`. Le `map` prend en argument des paires $(nomdefichier, texte)$ (on se fiche du nom du fichier). Le *reduce* prend en argument des chaines de caractères et des tableaux de *null*
3. Compléter le code des méthodes `map` et `reduce` comme vu en cours pour compter les occurrences de chaque mot. On pourra convertir le `Text` d'entrée du *map* en une chaîne Java et utiliser la méthode `split` des chaînes pour la découper en tableau de mots.
4. Une fois le code complété, vous devez générer un fichier jar à partir du projet (`Export -> Jar file`) à l'endroit que vous souhaitez puis tester avec :

   ```
   hdfs dfs -rm -r /user/VOTRELOGIN/output/

   hadoop jar ~/Untitled.jar tdd.tp03.DriverWordCount \
                             /user/VOTRELOGIN/output /user/VOTRELOGIN/input/jungle2.txt
   ```

   vérifier le résultat du calcul (s'il n'y a pas eu d'erreur :)

   ```
   hdfs dfs -cat /user/VOTRELOGIN/output/* | less
   ```

   (q pour quitter).

Réponse: Voir le code fourni.

Exercice 2

Copier (sous eclipse) votre classe `DriverWordCount` en une classe `DriverInter`. Modifier le code pour qu'il effectue la transformation suivante :

entrée une suite de fichier contenant sur chaque ligne 2 nombres séparés par un espace

sortie l'intersection des 2 colonnes, c'est à dire l'ensemble des nombres qui apparaissent en première position et en deuxième position au moins une fois dans les fichiers.

Réponse: L'algorithme est le suivant :

map le `map` prend en entrée des lignes de fichier texte. On coupe la ligne en deux, on récupère les deux entiers (x, y) et renvoie au *reducer* $(x, \texttt{G})$ et $(y, \texttt{D})$, pour signifier que l'on a vu le nombre x dans la colonne de gauche et y dans la colonne de droite.

reduce le `reduce` prend en entrée un entier z et un tableau de `G` et `D`. Si le tableau contient au moins un `G` et un `D` on sait que le nombre z était présent à gauche et à droite et on l'envoie en sortie sinon on peut l'ignorer.

L'algorithme ci-dessus est implémenté dans la classe `DriverInter`. Dans le corrigé on utilise deux constantes `IntWritable(0)` et `IntWritable(1)` pour signifier gauche et droite. En TP on a utilisé `BooleanWritable(true)` et `BooleanWritable(false)` mais à par cela le code est le même.

1. en pratique, vous n'aurez qu'un seul *worker*, la machine sur laquelle vous vous trouvez, mais il suffit d'ajouter des machines autres que localhost dans le fichier de configuration pour avoir plusieurs nœuds, le reste est identique

Exercice 3

Copier (sous eclipse) votre classe `DriverWordCount` en une classe `DriverUnion`. Modifier le code pour qu'il effectue la transformation suivante :

entrée une suite de fichier contenant sur chaque ligne 2 nombres séparés par un espace

sortie l'union des 2 colonnes, c'est à dire l'ensemble des nombres qui apparaissent en première position ou en deuxième position au moins une fois dans les fichiers.

Réponse: La solution (cf la classe `DriverUnion`) est très similaire au cas du *WordCount*.

map le `map` prend en entrée une ligne. Elle est scindée en deux entiers (x, y). On envoie $(x, 1)$ et $(y, 1)$ au *reduceur*

reduce le `reduce` reçoit en argument un entier z et un tableau de `1`. Pour chaque entrée dans le tableau, on écrit l'entier dans la sortie z.

Cela implémente la vraie union de multi-ensemble où le cardinal de sortie est exactement la somme des cardinaux d'entrée. Si on ne fait pas une sortie pour chaque élément, alors cela équivaut a retirer les doublons de la sortie (ou faire un `DISTINCT`) en SQL.

Exercice 4

On considère les deux fichiers `exo2_1.txt` et `exo2_2.txt`. Proposer un algorithme (en pseudo-code) pour calculer leur jointure sur la première colonne.

Implémenter cette solution.

Réponse: On présente d'abord l'algorithme à haut niveau et on le généralise un peu. Étant donnés des fichiers F_1, ..., F_n on veut implémenter la jointure naturelle, c'est à dire quelque chose comme :

`SELECT * FROM` F_1`, ...,` F_n `WHERE` F_1`.x = ` F_2`.x AND ... AND` F_{n-1} `=` F_n

L'algorithme est le suivant :

map Chaque ligne de texte est séparée en deux entiers (x, y) où x est la valeur de jointure. De plus on peut récupérer la nom du F_i du fichier dans lequel se trouve la ligne. On renvoie donc comme sortie du `map` la clé x et la valeur (y, F_i)

reduce Le `reduce` prend en clé d'entrée x et un tableau de paires $[(y_1, F_1), (y_2, F_3), \ldots, (y_k, F_m)]$. On regroupe les y_i par fichier d'origine, obtenant ainsi un tableau associatif :

$$\begin{array}{rcl} F_1 & \mapsto & [y_1, \ldots, y_k] \\ F_2 & \mapsto & [y_3, \ldots, y_j] \\ & \vdots & \\ F_n & \mapsto & [y_{42}, \ldots, y_l] \end{array}$$

Si on appelle T_i chacun des tableau associé au F_i alors :

— si l'un des T_i est vide c'est que l'entier x (la clé) n'apparaît pas dans la colonne x du fichier F_i correspondant. Le `reduce` ne fait rien (le résultat n'est pas dans la jointure)

— sinon on génère tous les produits n-uplets possibles :

```
foreach e_1 in T_1 do
    foreach e_2 in T_2 do
                ⋮
        foreach e_n in T_n do
                output (x, e_1, ..., e_n)
```

La class `DriverJoin` implémente cette solution complète. Le `Mapper` ne pose pas de difficulté (si ce n'est l'utilisation du contexte pour récupérer le nom du fichier d'où vient la ligne d'entrée). Le `Reducer` est un peu plus compliqué. Premièrement on récupère tous les noms de fichiers dans la méthode `main` de la classe `DriverJoin` (au moment où on lit la ligne de commande) et on stocke ces noms de fichiers dans l'objet `Configuration`. On peut ensuite accéder à ces noms de fichiers depuis le `Reducer`, en prenant l'objet `Configuration` se trouvant dans le `Context` du `Job` courant.

Une fois que l'on a ces noms de fichier, on peut initialiser une `HashMap` qui associe aux nom de fichiers des vecteurs d'entiers (comme ci-dessus). Une fois la `HashMap` remplie on teste qu'elle ne contient pas de `Vector` vide (sinon on ne fait rien) et on appelle une petite méthode récursive `crossProduct` qui implémente les « for each » imbriqués du pseudo-code (on ne peut pas faire de boucle for directement car on ne connait pas a priori le nombre de fichiers).

La méthode `crossProduct` dispose les T_i sur une pile. Récursivement si la pile est vide, la chaîne de caractère accumulée (i.e. la ligne de sortie). Sinon, on prend le sommet de pile T_i et pour chacun de ses éléments, on l'ajoute à l'accumulateur (tour à tour) et on se rappelle recursivement sur la suite de la pile. Au retour des appels récursifs, on repose T_i sur la pile pour les appels suivants.