Rendu Question TP Hadoop

ZHAO Tuo
SAZERAT Jérémy

Q1.1.1

**Map Input Records:** Le nombre de ligne entrées dans le mapper. C'est le nombre total de données traitées (dans notre cas 9 lignes de Lorem Ipsum)
**Map Output Records:** Le nombre de ligne en sortie du mapper. C'est le nombre de mots total qui ont été trouvé dans le texte d'entrée (puisqu'on génère des nuplets : <mot, 1> pour chaque mot du texte)

Q1.1.2

**Map Output Records** et **Reduce Input Records** sont équivalents puisque le reducer vient réduire le nombre de ligne générées par le mapper, il travaille sur l'intégralité des données qu'à généré le mapper. Donc la quantité de donnée est équivalente.

Q1.1.3

**Reduce Input Groups** c'est le nombre de clé unique existante dans les donnée générées par le mapper (dans notre cas la clé étant mot, c'est le nombre de mots distincts présents dans le texte). On peut donc observer qu'il y a 156 mots différents dans les 9 lignes de lorem ipsum données en entrée. (Attention: Les mots ont été nettoyés de la ponctuation donc : "lorem," => "lorem")

Q1.2

Dans HDFS le chemin vers notre répertoire personnel est **/home/lQbrod/**

Q1.3

Il y a 5 splits lus sur HDFS. Ce nombre correspond au nombre de tomes des misérables.
La trace d'exécution est enregistrée dans **/home/lQbrod/Q1-3.txt**

Q1.4

La trace d'exécution est disponible dans **/home/lQbrod/Q1-4.txt**

1) **Combine Input Records** et **Combine Output Records** permettent de vérifier que le combiner a bien travaillé. Si on utilise le combiner on remarque que l'entrée du reducer est équivalent à la sortie du combiner. (Puisqu'on a Mapper => Combiner => Reducer)
2) Les indicateurs : **GC time elapsed** et **CPU time spent** sont les deux indicateurs de temps d'exécution qui nous permettent d'évaluer le gain provoqué par le combiner. Cependant sur notre jeu de donnée le gain de temps par le combiner n'est pas visible puisque qu'avec combiner le temps d'exécution est de 16s alors qu'il est de 15s sans.
3) Pour récupérer le retour on utilise **hdfs dfs -copyToLocal wordcount/part-r-00000**
   Qui permet de copier localement (sur le cluster) le fichier
   Puis on utilise:
   **scp part-r-00000 sazeratj@mandelbrot.e.ujf-grenoble.fr:/home/s/sazeratj**

Pour le récupérer sur la machine

Enfin nous avons utilisé un petit script python rapide pour parser les données et récupérer le mot qui a le plus d'occurrences :
https://github.com/IQbrod/Hadoop-Parser-Wordcount
On obtient après exécution du script que le mot le plus fréquent dans les cinq tomes des misérables est : **DE** avec **16922** occurrences. (On aurait pu passer par une map pour obtenir les N éléments les plus fréquents).

Q1.5
En utilisant 3 reducer on obtient **un résultat en 3 parties** différentes alors qu'avec un seul on obtient un résultat dans un seul fichier. Ce qui est logique puisque le reducer est à la fin de la chaîne mapreduce :

IN => Mapper => Combiner => Reducer => OUT

Q 2.2
Le combiner doit prendre **en entrée** un ensemble d'objets liés (clé, valeur). La valeur donnée par le mapper sera 1 (puisque dans notre cas un tag ne peut être mentionné qu'une fois par post). Ces données n'ont pas d'ordre et les clés peuvent être répétées (si deux posts ont le même tag on aura : <tag,1>, <tag,1> …). C'est donc une **liste non ordonnée** (**List** en Java).
**En sortie**, le combiner renverra aux reducers des sous ensembles de données liés (clé,valeur) où la clé est unique et la valeur correspondra au nombre d'occurrences du tag (clé) dans le sous ensemble. Les données en sorties sont des **ensembles non ordonnées** (**Set**, ou **Map** pour des données liées).

Le combiner doit pouvoir modifier les données liées qu'on lui fourni donc les données (clé, valeur) doivent pouvoir être modifiée (**Writable** en Java)
On a donc en entrée une List<? : implements Writable> et en sortie une Map<? : implements Writable>.

Q3.Préliminaire
Le premier Job est un Mapper/Reducer basique qui parse les données.
Le Mapper recoit le fichier de données Flickr et sort tuples : <Pays, (Tag, 1)>
Où Pays est la clé de type String (Text) et la valeur (Tag, 1) de type StringAndInt.
Le Reducer réduit la map (l'ensemble de tuples) de façon à obtenir par Pays pour un même tag une seule quantité commune. Ex: <"FR", ("photo", 4)>.
Le second job est également un Mapper/Reducer, le Mapper ne fait que transmettre la donnée du premier reducer en le triant par ordre décroissant (Ainsi les tags le plus populaires seront placés en premier pour chaque pays). Le second Reducer viendra récupérer les K premiers de chaque pays. Les types échangés sont pour le second job des tuples <Text, StringAndInt>.
Les comparateurs pour le tri s'effectuent sur l'entier du string and int (afin de placer les plus fréquemment utilisés en premier).

Q3.1
La fonction de reduce finale a un avantage en terme de performances puisque les données que l'on lui fourni en entrée sont déjà triées. Donc il n'a plus qu'à récupérer les K premier éléments de la map pour chaque pays (qui sont déjà triés, pas besoin de parcourir le DataSet intégralement, aussi énorme soit-il).

Q3.2
S'il existe des tags classés ex-aequo pour un même pays (avec le même nombre d'occurence) le résultat de deux exécution peut différer selon lequel est trié et placé en premier puisque le comparateur ne se soucie que de l'entier. Un moyen de pallier à ce problème est de trier d'abord sur la quantité (l'entier) puis sur le tag (la string) pour obtenir toujours le même résultat.