TP Map-Reduce (version étendue)

M2GI 2018-2019

4 décembre 2018

Remarque : ce TP est une version étendue du premier TP. Vous n'avez qu'un compte-rendu à rendre (pour les 2 TPs).

Votre compte-rendu (une archive `zip` ou `tar.gz`) devra contenir :
- Un fichier pdf avec (1) les réponses aux questions qui suivent, et (2) une description et une explication des codes sources que vous aurez produits.
- Vos sources. Ecrivez un fichier par problème qui vous est soumis. Exemple : `Question3_1.java` pour la question 3.1.

Envoyez l'archive à l'adresse vivien.quema@imag.fr, avec [TPHadoop] au début du sujet. Mettez votre binôme en copie de cet e-mail.

Remarque : ce travail est noté et doit donc être effectué en autonomie.

Le compte-rendu est à rendre le 18 décembre 2018 au plus tard.

0 Kit de survie sous Eclipse

Voici une liste de raccourcis clavier utiles :
- *Ctrl + Espace* pour l'autocomplétion, qui permet aussi de générer des méthodes (essayez dans le corps d'une classe)
- *Ctrl + Shift + 1* "Quick fix", propose des actions pertinentes selon ce qu'il y a sous le curseur.
- *Ctrl + Shift + O* pour ajouter les imports manquants et retirer les inutiles.
- *Ctrl + clic* pour atteindre la définition du nom sous le curseur de la souris (utilisez *Alt* + ← pour revenir là où vous étiez).
- *Alt + Shift + R* pour renommer ce qui est sous le curseur (partout où c'est pertinent).

Voir aussi `http://eclipse-tools.sourceforge.net/Keyboard_shortcuts_(3.0).pdf`

1 Prise en main

Nous avons préparé un projet Eclipse qui contient les librairies nécessaires. Vous pourrez ainsi tester vos programmes localement, sur de petits fichiers, avant de les lancer sur le cluster. Ce qui est très pratique si vous avez besoin de debugger avec des points d'arrêt.

1. Téléchargez sur votre poste de travail `TPIntroHadoop.zip`
2. Dans les menus d'Eclipse, cliquez sur `File > Import ...`
3. Dans la catégorie `General`, sélectionnez `Existing Projects into Workspace` (et non `Archive`)
4. Choisissez `select archive file` et retrouvez l'archive téléchargée
5. Cliquez sur `Finish`

Le répertoire des sources contient `Question0_0.java`, qui peut vous servir d'exemple pour écrire chaque programme (mais pour certaines questions, attention à modifier les types de clés/valeurs qu'il déclare). Vous remarquerez que le `Mapper` et `Reducer` sont déclarés comme des classes internes : ce n'est pas imposé par le framework, c'est même déconseillé. On se le permet dans le cadre du TP pour faciliter les corrections.

1.1 Exécution locale

Avant de lancer un programme MapReduce sur un cluster, on le teste *toujours* localement, sur un échantillon des données à traiter. Dans ce cas, le framework ne lit/écrit pas sur HDFS, mais sur votre système de fichiers local.

1. Créez un programme qui compte le nombre d'occurrences de chaque mot dans un fichier texte (cf. "WordCount" vu en cours).
2. Créez à la racine du projet un fichier texte, avec le (petit) contenu de votre choix.
3. La fonction `main` prend en argument le chemin du fichier d'entrée et de sortie. Avant de lancer le programme avec Eclipse, il faut donc lui indiquer ces chemins (vous pouvez utiliser des chemins relatifs à la racine du projet) .

Lancez le programme. Les traces d'éxécution du framework apparaîssent dans l'onglet "Console". En cas de bug, vous y verrez l'exception à l'origine du plantage. Ouvrez le fichier généré et vérifiez vos résultats.

Quand le programme marche, à la fin des traces d'éxécution se trouvent les compteurs tenus à jour par Hadoop. Ajoutez/retirez du contenu à votre fichier de texte, relancez votre programme et observez l'évolution des compteurs pour répondre aux questions suivantes.

1. Que signifie `Map input records` ? Et `Map output records` ?
2. Quel est le lien entre `Map output records` et `Reduce input records` ?
3. Que signifie `Reduce input groups` ?

1.2 Premier contact avec HDFS

Un cluster a été déployé pour les besoins du TP. Un nom d'utilisateur et un mot de passe vous ont été remis, ils vous permettent de vous connecter via `ssh` au noeud maître "NameNode" (qui est aussi un système Linux classique).

1. Ouvrez un shell sur le noeud maître avec `ssh votrelogin@152.77.78.100`[1]
2. Executez `hdfs dfs` pour lister les commandes disponibles.
3. Executez `hdfs dfs -ls /` pour afficher le contenu de la racine d'HDFS.
4. Executez `hdfs dfs -ls` pour afficher le contenu de votre répertoire HDFS personel.

Prenez le temps d'explorer ce que contient notre cluster.

Question : quel est le chemin, dans HDFS, vers votre répertoire personel ?

1.3 Exécution sur le cluster

Le programme écrit en partie 1.1, s'il fonctionnait localement, devrait fonctionner sur le cluster sans modifications. Mais il faut d'abord en faire un paquet compilé.

1. Dans le navigateur d'Eclipse, faites un clic droit sur le répertoire `src`. Cliquez sur "Export..."
2. Ouvrez la catégorie "Java", puis double-cliquez sur "JAR file".

[1] Vous pouvez changer votre mot de passe, mais nous vous recommandons plutôt de copier votre clé publique dans `/.ssh/authorized_keys`,`/.ssh/authorized_keys` avec `sh-copy-id`,`sh-copy-id`

3. Parmi les ressources à exporter, **vérifiez que le répertoire `lib` n'est pas sélectionné.**
4. Entrez le nom du paquet, par exemple `tp.jar`.

Le paquet généré sera à la racine de votre espace de travail Eclipse : transferez-le avec `scp` vers votre répertoire personnel sur le noeud maître.

Nous allons maintenant compter le nombre d'occurrences de chaque mot dans les 5 tomes des *Misérables* de Victor Hugo. Si votre classe principale s'appelle `Question1_1`, lancez votre programme sur le noeud maître en tapant :

```
hadoop jar tp.jar Question1_1 /data/miserables wordcount
```

Question : dans les traces d'éxécution (au début), retrouvez le nombre de "splits" lus sur HDFS. A quel compteur ce nombre correspond-il ?

En préparation de la question suivante, conservez les traces d'éxécution de ce programme dans un fichier temporaire, et veillez à conserver le répertoire qui contient les résultats.

1.4 Combiner

— Ajoutez `job.setCombinerClass(???)` dans le programme principal (à vous de compléter avec la bonne valeur en argument).
— Vérifiez que le programme fonctionne localement, puis éxécutez-le sur les 5 tomes des *Misérables* (sans écraser le résultat de la partie 1.3).

Questions :

1. Quels compteurs permettent de vérifier que le combiner a fonctionné ?
2. Quels compteurs permettent d'estimer le gain effectivement apporté par le combiner ? Comparez aux valeurs obtenues sans combiner pour justifier votre réponse.
3. Quel est le mot le plus utilisé dans *Les Misérables* ?

1.5 Nombre de reducers

Par défaut, Hadoop n'instancie qu'un seul `Reducer`. Remédiez à sa solitude en ajoutant
`job.setNumReduceTasks(3)` dans la méthode principale.

Question : Quelle est la différence entre le répertoire de résultats obtenu ici, et celui de la partie 1.3 ? Pourquoi ?

2 Top-tags Flickr par pays, avec tri en mémoire

Nous allons maintenant analyser des meta-données de photos, en utilisant un extrait du jeu de données "Yahoo ! Flickr Creative Commons 100M". Le but est de trouver les tags les plus utilisés par pays. Pour identifier un pays à partir des coordonnées d'une photo, vous utiliserez la classe fournie `Country`.

L'archive que vous avez téléchargée au début du TP contient deux fichiers qui vont vous aider à vous familiariser avec ces donneés :

— `flickrSpecs.txt` décrit le format du fichier. Pour décoder les textes, par exemple les tags, utilisez `java.net.URLDecoder.decode(String s)`,

— `flickrSample.txt` est un extrait à utiliser pour les tests locaux.

A partir d'ici, la référence de l'API Hadoop pourra vous être utile : `https://hadoop.apache.org/docs/r2.2.0/api/index.html`

Attention : dans cette référence, de nombreuses classes existent en double. Quand c'est le cas, choisissez toujours celle qui appartient au package `org.apache.hadoop.mapreduce`, qui correpond à l'API moderne, dite "YARN". Les doublons appartiennent au package `org.apache.hadoop.mapred`, qui est celui de l'ancienne API, dite "MR1".

2.1 Map et Reduce

Puisque l'on veut trouver les K tags les plus utilisés *par pays*, a priori l'identifiant d'un pays (les deux lettres retournées par `country.toString()`) est une bonne clé intermédiaire. Les valeurs associées seront les tags (les chaines de caractères) qui ont été associés à une photo prise dans ce pays. En sortie, il ne restera qu'au plus K couples $(pays, tag)$ par pays.

A vous d'implémenter le programme complet. Pour la fonction `reduce` :

1. Comptez le nombre d'occurrences de chaque tag avec `java.util.HashMap<String,Integer>`.
2. Puis triez les tags par nombre d'occurrences décroissants :
   — Créez (dans un fichier séparé) une classe `StringAndInt` qui implémente l'interface `Comparable<StringAndInt>` et contient deux champs, qui dans ce cas représenteront le tag et son nombre d'occurrences. La méthode `compareTo` ne prendra en compte que le nombre d'occurrences.
   — Utilisez la classe `MinMaxPriorityQueue<StringAndInt>` [2] afin de ne conserver que les K tags les plus populaires.

[2] cf. `http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/MinMaxPriorityQueue.html` Créez l'objet avec `inMaxPriorityQueue.maximumSize(k).create(),inMaxPriorityQueue.maximumSize(k).create()`.

K sera un nouveau paramètre à passer au programme principal via la ligne de commande. Utilisez l'objet `Configuration` pour transmettre sa valeur du programme principal aux *reducers* (via l'objet `Context`).

Implémentez puis testez *localement* ce programme.

2.2 Combiner

Question : pour pouvoir utiliser un *combiner*, quel devrait être le type des données intermédiaires ? Donnez le type sémantique (que représentent ces clés-valeurs ?) et le type Java.

Avant de tester ce programme sur le cluster :

1. Modifiez la classe que l'on a ajouté en 2.1, de sorte qu'elle implémente aussi l'interface `Writable`. La référence de cette interface vous fournira quelques indices, toutefois :
   — encapsulez un `Text` pour la chaine de caractères
   — dans leur exemple, la méthode statique `read` est inutile
   — dans leur exemple, il manque un constructeur sans arguments
2. Modifiez une copie du programme écrit dans la partie 2.1, en y ajoutant un *combiner*. Cette fois, on ne peut pas réutiliser le *reducer*, il faut implémenter une troisième classe (qui doit étendre `Reducer<K,V,K,V>`).

Executez cette variante (avec $K = 5$) sur `/data/flickr.txt`. Quels sont les tags les plus utilisés en France ?

Question : Dans le *reducer*, nous avons une structure en mémoire dont la taille dépend du nombre de tags distincts : on ne le connaît pas a priori, et il y en a potentiellement beaucoup. Est-ce un problème ?

3 Top-tags Flickr par pays, avec mémoire limitée

Une fonction (parfois) importante du framework MapReduce est le tri des clés intermédiaires. Le framework permet au développeur, en plus de spécifier des types ad-hoc, de fournir ses propres fonctions de tri. On peut ainsi contrôler :

— la formation des groupes avant la phase *reduce*, c'est à dire quelles valeurs seront associées à quelle clé dans le *reducer*, et
— l'ordre des couples $(cle, valeur)$ qui seront itérés par le *reducer*.

Vous trouverez les explications détaillées dans la référence de la classe Reducer.

L'objectif de cette partie est de faire faire à Hadoop le tri des tags par nombre d'occurrences décroissant afin de limiter autant que possible l'utilisation mémoire (RAM) du programme.

Question préliminaire : spécifiez les 2 jobs nécessaires (le second va utiliser le résultat du premier), en précisant le type des clés/valeurs en entrée, sortie et intérmédiaires. Pour le second, spécifiez aussi les comparateurs à utiliser pour le tri et pour le découpage en groupes.

3.1 Passes MapReduce en chaîne

Commencez par implémenter et tester localement le premier *job*.

Cette première passe matérialise un fichier intermédiaire, qui ne sera utilisé que par une autre passe MapReduce. Dans ces cas-là, plutôt que d'écrire le fichier au format texte, on utilise le format binaire d'Hadoop. Cela permet d'utiliser des types sémantiques en entrée du `Mapper` qui va suivre, au lieu du couple `LongWritable, Text` imposé par le format texte.

Pour cela, dans la fonction qui prépare le `job` que vous venez d'écrire, changez l'*input format* afin d'utiliser `SequenceFileOutputFormat`.

Exécutez cette version-là sur le cluster (n'oubliez pas le *combiner*).

Question : un avantage de cette méthode est que le tri est réalisé sur le disque dur des *data nodes*, plutôt qu'en mémoire : on peut donc trier des quantités de données plus importantes[3]. Dans notre application, cette méthode présente un autre avantage pour la fonction `reduce` finale ; lequel ?

Implémentation du deuxième job :

- notre classe `StringAndInt` doit maintenant implémenter une seule interface :
  `WritableComparable<StringAndInt>`
- afin de lire le fichier binaire généré par la première passe, utilisez `SequenceFileInputFormat`.
- pour faciliter la lecture du résultat, utilisez `TextOutputFormat` en sortie.
- les fonctions de tri sont à implémenter dans une classe dédiée, qui

[3] C'est d'ailleurs le test classique des performances d'un cluster, cf. `http://sortbenchmark.org/`.

  - doit étendre `WritableComparator`, et
  - doit avoir un constructeur sans arguments, qui appelle `super(StringAndInt.class, true)`.
- **attention** le tri des données intermédiaires est fait *avant* le découpage en groupes. La fonction de tri principale doit donc pre-placer les groupes.

Question : s'il existe des tags classés *ex aequo* dans le top-K d'un pays, a-t-on la garantie d'obtenir toujours le même résultat d'une éxécution à l'autre ? Pourquoi ?