AVIGNON UNIVERSITÉ

M2 ILSEN & IA – 2022/23

UE Business intelligence & Systèmes décisionnels

ECUE Data Analytics & Big Data

Vincent Labatut

TP 1 | Introduction à MapReduce et Hadoop

MapReduce est un modèle de programmation parallèle défini par Google et inspiré de concepts issus de la programmation fonctionnelle. Il est conçu pour manipuler de grandes quantités de données via une architecture distribuée. L'implémentation la plus connue de ce modèle est probablement celle d'Apache, appelée Hadoop. Il s'agit d'un logiciel libre écrit en Java et destiné à supporter l'exécution d'applications parallèles.

Le but de ce TP est de comprendre le principe de parallélisation à l'œuvre derrière MapReduce, et de l'appliquer sur des exemples simples en utilisant Hadoop.

Votre travail n'est pas évalué pour cette séance : aucun rendu n'est donc demandé. Cependant, il le sera pour la seconde séance. Il est donc nécessaire que vous finissiez ce TP 1 pour avoir les bases nécessaires à la réalisation du TP 2, qui portera sur une application plus élaborée. N'hésitez pas à poster des questions sur le forum du cours entre les deux séances, si vous êtes bloqué·e.

1 Principe de MapReduce

Le principe est le suivant : on découpe d'abord les données en plusieurs parties traitables indépendamment par des processeurs distincts, puis on combine les résultats obtenus pour chaque partie de manière à construire le résultat global.

1.1 Décomposition du traitement

Le traitement est distribué sur un ensemble de nœuds de calcul. Il se compose des étapes suivantes, illustrées par la Figure 1 :

1. **Découpage des données :** les données d'entrée sont lues par le nœud principal, et découpées en n parties. Chaque partie P_i ($1 \leq i \leq n$) est associée à une clé unique K_i permettant de l'identifier.
2. **Fonction `Map` :** une fonction `Map` est invoquée séparément pour chaque partie, généralement en parallèle sur plusieurs nœuds. Elle est donc invoquée n fois au total. Cette fonction prend en entrée une paire (K_i,P_i) issue du découpage des données, et renvoie une liste de paires $L_i = \langle(K'_j,V_{ij}),...\rangle$. Chaque paire contient une clé K'_j et une valeur V_{ij} ($1 \leq j \leq m$) résultant du traitement réalisé par la fonction. Une même clé peut apparaître plusieurs fois dans la liste. L'ensemble des m clés possibles est le même pour toutes les exécutions de `Map`.
3. **Répartition des paires :** chaque clé K'_j est associée à un nœud de traitement (plusieurs clés peuvent être associées à un même nœud). Ce nœud reçoit toutes les paires (K'_j,V_{ij}) produites par l'ensemble des nœuds ayant exécuté la fonction `Map`. Une paire (K'_j, L'_j) est construite, dont le deuxième terme est une liste contenant toutes les valeurs associées à la clé considérée : $L'_j = \langle V_{ij},...\rangle$.
4. **Fonction `Reduce` :** chaque nœud sélectionné lors de la répartition exécute une fonction `Reduce` pour chacune des clés K'_j qui lui sont associées. Autrement dit, la fonction est exécutée un total de m fois (en considérant tous les nœuds). La fonction prend en entrée la paire (K'_j, L'_j) issue de la répartition. À l'issue de son traitement, elle renvoie une valeur V'_j (ou parfois plusieurs).
5. **Résultat final :** le résultat final est une liste de paires $L'' = \langle(K'_j,V'_j),...,(K'_j,V'_j)\rangle$.

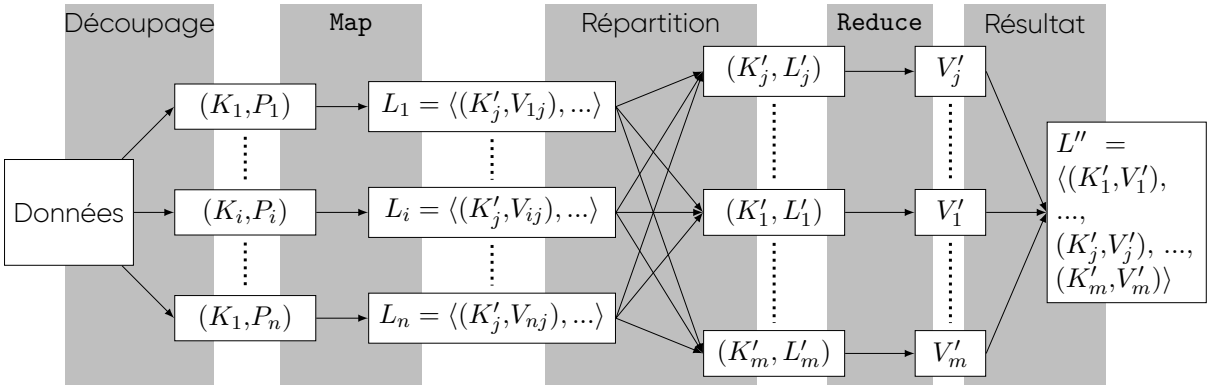

Figure 1. Décomposition d'un traitement de type *MapReduce*.

1.2 Exemple

L'exemple typique utilisé pour illustrer simplement cette approche est celle du décompte de mots dans un texte. Supposons qu'on a le texte suivant :

```
Voici la première ligne,
et voici une autre ligne : la seconde ligne.
La troisième ligne.
Et enfin la dernière ligne.
```

Découpage. Dans cet exemple, on pourrait envisager que le découpage des données se fasse par phrases. À l'issue de cette étape, on aurait donc les paires (l_1,"Voici la première ligne,"), (l_2,"et voici une autre ligne : la seconde ligne."), (l_3,"La troisième ligne."), et (l_4,"Et enfin la dernière ligne.") (où l_i est la ligne numéro i, avec $1 \leq i \leq 4$).

`Map`. Supposons qu'on ait 4 nœuds de calcul disponibles, exécutant chacun la fonction `Map` pour une clé différente (donc pour une ligne). Notez qu'il peut y avoir moins de 4 nœuds, auquel cas certains d'entre eux devront traiter plusieurs clés (indépendamment). Dans cet exemple, le rôle de `Map` est de décomposer la phrase reçue en entrée, et de renvoyer les mots obtenus, en leur associant la valeur 1 (chaque mot apparaissant une fois dans la phrase). Ainsi, pour la ligne l_2, on a : $L_2 = \langle$("et",1), ("voici",1), ("une",1), ("autre",1), ("ligne",1), ("la",1), ("seconde",1), ("ligne",1)$\rangle$.

Les clés K'_j obtenues à l'issue de cette étape correspondent aux mots uniques contenus dans le texte traité, à savoir : "autre", "dernière", "enfin", "et", "la", "ligne", "première", "seconde", "troisième", "une", "voici". On a donc $m = 11$. De plus, dans cet exemple, toutes les valeurs V_{ij} sont égales à 1.

Répartition. Pour chaque clé K'_j détectée lors du `Map`, on construit la paire (K'_j, L'_j) associée. Par exemple, pour la clé K'_6 ="ligne", on a $L'_6 = \langle 1, 1, 1, 1, 1\rangle$ (i.e. un 1 pour chaque occurrence du mot).

`Reduce`. Supposons qu'au moment de la répartition, on ait seulement 3 nœuds disponibles : chacun va traiter indépendamment 3 ou 4 des 11 clés K'_j. Dans cet exemple, `Reduce` consiste simplement à additionner les valeurs trouvées dans la paire de l'étape précédente. Ainsi, pour K'_5 ="la", la fonction renverra $V'_5 = 4$, alors que pour K'_6 ="ligne" on aura $V'_6 = 5$.

Résultat. Le résultat final associe à chaque clé K'_j le nombre d'occurrences du mot correspondant. On peut ici le représenter sous la forme d'une liste de paires : $L'' = \langle$ ("autre", 1), ("dernière", 1), ("enfin", 1), ("et", 2), ("la", 4), ("ligne", 5), ("première", 1), ("seconde", 1), ("troisième", 1), ("une", 1), ("voici", 2) $\rangle$.

2 Présentation sommaire de Hadoop

Outre MapReduce, la plateforme Hadoop comprend notamment un système de fichiers distribué spécifique appelé HDFS (pour *Hadoop Distributed File System*) et exécuté par-dessus le système de fichier local du système d'exploitation ; et un gestionnaire de ressources nommé YARN (pour *Yet Another Resource Negotiator*).

Vérifiez d'abord que vous pouvez vous connecter à `localhost` en SSH sans mot de passe :

```
ssh localhost
```

Si c'est le cas, déconnectez-vous de SSH (CTRL-D) et passez à l'étape suivante. Sinon, créez une clé SSH sans mot de passe :

```
ssh-keygen -t rsa
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
chmod 0600 ~/.ssh/authorized_keys
```

Appuyez sur entrée quand un nom ou un mot de passe vous est demandé. Vous n'avez à faire cette opération que la première fois que vous utilisez Hadoop.

2.1 Système de fichiers

Les principales caractéristiques de HDFS sont qu'il est distribué et extensible, ce qui lui permet de manipuler de grandes quantités de données. Il donne à Hadoop une vue unifiée d'une arborescence physiquement répartie sur des machines distinctes. Ce système est constitué de différents types de composants, dont les principaux sont les nœuds de nommage (NameNodes) et nœuds de données (DataNodes).

Le nœud de nommage est unique au cluster (il existe toutefois un nœud de nommage secondaire en cas de problème), et est confondu avec le maître MapReduce. Il gère l'espace des noms de fichiers et dossiers, les métadonnées des fichiers, et permet de les localiser dans le cluster. Un nœud de données correspond à un esclave MapReduce, et a pour but de stocker les blocs de données constituant les fichiers. Une demande d'accès effectuée auprès du nœud de nommage permet de localiser les blocs concernés sur les différents nœud de données constituant le cluster.

L'accès au système de fichiers se fait au moyen d'un service dédié. Il faut d'abord formater le système de fichiers :

```
$HADOOP_HOME/bin/hdfs namenode -format
```

Et on peut ensuite lancer le service avec la commande :

```
$HADOOP_HOME/sbin/start-dfs.sh
```

L'arrêt du service est réalisé au moyen de la commande `stop-dfs.sh`. Il existe un accès Web à HDFS, via l'URL `http://localhost:9870/`.

Le service permet d'exécuter un certain nombre d'instructions effectuant des opérations classiques sur le système de fichiers. Pour créer un dossier, on utilise ainsi `mkdir` :

```
$HADOOP_HOME/bin/hdfs dfs -mkdir <chemin>
```

où `<chemin>` est le chemin du dossier à créer.

La suppression d'un dossier passe par la commande `rm -r` :

```
$HADOOP_HOME/bin/hdfs dfs -rm -r <chemin>
```

On peut copier des fichiers du système local vers Hadoop grâce à `put` :

```
$HADOOP_HOME/bin/hdfs dfs -put <source> <cible>
```

où `<source>` est le chemin local du ou des fichier(s) à copier, et `<cible>` le dossier HDFS dans lequel on veut les copier.

La commande **get** permet de faire le contraire, i.e. copier des fichiers à partir de HDFS vers le système local :

```
$HADOOP_HOME/bin/hdfs dfs -get <source> <cible>
```

où **<source>** est le chemin HDFS du ou des fichier(s) à copier, et **<cible>** le chemin du dossier de destination sur le système local.

On liste le contenu d'un dossier avec **ls** :

```
$HADOOP_HOME/bin/hdfs dfs -ls <chemin>
```

où **<chemin>** est le chemin du dossier dont on veut afficher le contenu.

Enfin, il est possible d'afficher le contenu d'un fichier dans la console avec **cat** :

```
$HADOOP_HOME/bin/hadoop fs -cat <chemin>
```

où **<chemin>** est le chemin du fichier texte dont on veut afficher le contenu.

La liste complète des opérations supportées par le service est disponible en ligne [1].

Exercice 1

Avant d'exécuter des jobs MapReduce, il est nécessaire de créer une arborescence spécifique destinée à contenir les fichiers d'entrée et de sortie des jobs. Elle prend la forme **/user/<username>** où **<username>** est le nom de l'utilisateur *sur la machine*. Au moyen des instructions décrites ci-dessus, créez ces deux dossiers.

2.2 Gestion des jobs et ressources

Le moteur MapReduce repose sur deux types de gestionnaires : le *JobTracker* est exécuté par le maître et gère les jobs MapReduce lancés par l'utilisateur ; tandis que le *TaskTracker* est exécuté sur chaque esclave et reçoit du JobTracker les tâches que l'esclave doit traîter.

À partir de la version 2 d'Hadoop, YARN permet de gérer les ressources plus efficacement. Il comprend le gestionnaire de ressources proprement dit (*Resource manager*), qui tient à jour la liste de jobs courants et les ressources qui leur sont allouées, et l'*Application master* qui surveille leur état d'avancement.

Comme HDFS, YARN prend concrètement la forme d'un service, qui se lance *après* HDFS au moyen de la commande :

```
$HADOOP_HOME/sbin/start-yarn.sh
```

Là aussi, l'accès au service est possible par le navigateur Web, à l'URL http://localhost:8088/.

Exercice 2

Exécutez l'un des programmes de démonstration pour vérifier si Hadoop fonctionne correctement. Pour cela, vous devez d'abord avoir lancé HDFS, créé les dossiers de l'Exercice 1, et lancé YARN (cf. ci-dessus). Créez ensuite un autre dossier **input** dans votre dossier personnel sur Hadoop :

```
$HADOOP_HOME/bin/hdfs dfs -mkdir input
```

Copiez-y quelques fichiers texte pour le test :

```
$HADOOP_HOME/bin/hdfs dfs -put $HADOOP_HOME/etc/hadoop/*.xml input
```

Vérifiez qu'ils y sont bien avec la commande **ls** (cf. Section 2.1). Exécutez l'un des JAR d'exemple disponibles (attention, il s'agit d'une seule commande) :

```
$HADOOP_HOME/bin/hadoop jar
                $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.3.4.jar
                                                        grep input output 'dfs[a-z.]+'
```

1. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html

et allez vérifier que le programme a bien été exécuté en visualisant le contenu des fichiers de résultat qu'il a créés dans le dossier `output` (sur HDFS).

> **Attention :** si le dossier `output` existe déjà (par exemple suite à un test précédent), l'exécution du JAR est susceptible de lever une exception. Il faut alors supprimer ce dossier avant de relancer son exécution.

Il est possible de lister les jobs MapReduce en cours au moyen du script `mapred` et de sa commande `job` :

```
$HADOOP_HOME/bin/mapred job -list
```

Et on peut forcer la terminaison d'un job en cours avec l'option `-kill` :

```
$HADOOP_HOME/bin/mapred job -kill <jobid>
```

où `<jobid>` est l'identifiant du job tel qu'affiché par avec `-list`.

3 Décompte de mots

À titre d'exemple, nous allons reprendre le problème de décompte de mots dans des fichiers textuels décrit en Section 1.2.

Exercice 3

Les données à utiliser sont contenues dans l'archive multi-volume [2] fournie sur e-uapv pour ce TP. La décompression se fait en ligne de commande :

```
cat donnees.tar.* | tar xvzf -
```

Vous devriez obtenir quatre dossiers : `socio`, `wp_20`, `wp_200` et `wp_2000`. Le dossier `socio` sera utilisé en Section 4. Copiez les dossiers en `wp_*` dans le système HDFS, comme expliqué en Section 2.1.

Le corpus complet (`wp_2000`) contient 2 871 textes tirés de la version française de Wikipedia, obtenus en crawlant automatiquement le site à partir de la page *Recherche d'information* (avec une limite de 2 hops et en restant dans Wikipedia). Les articles sont nettoyés pour ne contenir que lettres, chiffres, et ponctuation. Les dossiers `wp_20`, `wp_200` sont des versions réduites du corpus complet, contenant respectivement 20 et 200 articles. On considère chaque article comme une partie des données (notation P_i dans la Section 1.1), identifiée de façon unique par le nom du fichier qui le contient (K_i).

Exercice 4

Si vous voulez écrire vos classes Hadoop à partir d'Eclipse (ou quelque autre IDE), vous devez ajouter les JAR suivants au *Build Path* [3] : `$HADOOP_HOME/share/hadoop/client`, `common`, `common/lib`, `hdfs`, `mapreduce`, et `yarn`.

Notez que vous pouvez tout aussi bien écrire vos classes dans un éditeur de texte. De toute façon, la compilation finale devra être faite en ligne de commande, comme expliqué dans l'Exercice 8.

> **Attention :** les classes contenues dans le package `org.apache.hadoop.mapred` correspondent à l'ancienne API d'Hadoop (avant la v2). Vous ne devez utiliser que les classes du package `org.apache.hadoop.mapreduce` dans vos programmes.

Exercice 5

Une application Hadoop de base se compose de trois parties : le mapper, le reducer, et la fonction principale `main`. Cette dernière permet de configurer le job et de charger les données, tandis que les deux premières sont responsables du traitement.

2. Il y a une limite de 100 Mo sur la taille des fichiers que l'on peut déposer sur e-uapv.
3. Propriétés du projet > *Java Build Path* > *Libraries*, puis sélectionnez *Classpaths*, et cliquez sur le bouton *Add external JARs...*

La phase *Map* de MapReduce est implémentée via une classe dédiée, nommée `org.apache.hadoop.mapreduce.Mapper`. Elle prend en paramètres les types `KEYIN` et `VALUEIN` correspondant à la clé et à la valeur prises en entrée (notées K_i et P_i dans la Section 1.1), ainsi que `KEYOUT` et `VALUEOUT` qui correspondent à la clé et à la valeur produites en sortie (notées K'_j et V_{ij} dans la Section 1.1). Le traitement est réalisé dans la méthode `map(KEYIN, VALUEIN, Context)`, où `org.apache.hadoop.mapreduce.Mapper.Context` est une classe Hadoop permettant notamment d'écrire le résultat du traitement.

Créez une classe `WordCount`, contenant une classe `WordCountMapper` qui hérite de `Mapper`. On utilisera les types Hadoop suivants en paramètres de `Mapper` : `LongWritable`, `Text`, `Text`, `IntWritable` (tous contenus dans le package `org.apache.hadoop.io`). Autrement dit, la phase *Map* va prendre en entrée des clés correspondant à des entiers longs et des valeurs correspondant à du texte, et va produire en sortie des clés correspondant à du texte et des valeurs correspondant à des entiers. Implémentez la méthode `map` de cette classe, de manière à décomposer le texte reçu en mots. Comme expliqué en Section 1.2, pour l'application considérée ici : les K'_j sont les mots, et les V_{ij} sont tous égaux à 1. La méthode `map` doit donc produire des paires de la forme (mot, 1).

Pour la tokénisation (i.e. la décomposition du texte en mots), on considèrera que tout ce qui n'est pas une lettre est un séparateur de mots (y compris les chiffres). Le plus simple est d'utiliser `String.split` avec une classe regex appropriée. Notez qu'en regex Java, `\pL` permet de faire référence à n'importe quelle lettre (y compris accentuée) dans un document Unicode. De plus, on ne veut pas tenir compte de la casse[4], ce qui implique un traitement supplémentaire du texte. Enfin, pensez à gérer le cas particulier des chaînes vides.

Pour instancier un `IntWritable` de valeur 1, il faut simplement faire :

```
IntWritable value = IntWritable (1);
```

Enfin, pour écrire une paire dans le contexte `context`, on fait :

```
context.write(key, value);
```

où `key` et `value` sont respectivement la clé et la valeur à écrire.

> **Types des entrées/sorties :** À noter que les classes de la forme `XxxxWriter` de Hadoop correspondent à des réimplémentations de classes existant déjà dans le JDK standard, mais avec un mécanisme de sérialisation plus simple et plus rapide. Outre les entiers longs (`LongWritable`) et courts (`IntWritable`), on peut aussi manipuler des booléens (`BooleanWritable`), des réels (`FloatWritable`), des tableaux (`ArrayWritable`), etc. Les chaînes de caractères sont représentées par `Text`.

Exercice 6

De façon similaire à *Map*, la phase *Reduce* est décrite via une classe `Reducer`. Comme `Mapper`, elle est paramétrée par des types `KEYIN`, `VALUEIN`, `KEYOUT` et `VALUEOUT`. Cependant, ces types ne sont pas nécessairement les mêmes que pour le mapper. Cette fois, les types `KEYIN` et `VALUEIN` correspondent aux termes K'_j et V_{ij} de la Section 1.1 (autrement dit : les sorties de *Map*), tandis que `KEYOUT` et `VALUEOUT` correspondent aux termes K'_j et V'_j. Le traitement est spécifié par la méthode `reduce(KEYIN, Iterable<VALUEIN>, Context)`, où `org.apache.hadoop.mapreduce.Reducer.Context` sert là aussi à écrire le résultat. Comme indiqué par l'utilisation d'un `Iterable`, la valeur attendue en entrée est de nature séquentielle (une liste par exemple).

Dans `WordCount`, créez une classe `WordCountReducer`, héritant de la classe `Reducer`, et sa méthode `reduce`. La classe prend les types suivants en paramètres : `Text`, `IntWritable` (comme les sorties de `WordCountMapper`), et `Text`, `IntWritable`. Pour une clé (donc un mot) donnée, la méthode `reduce` doit sommer les valeurs composant la liste associée (ce qui revient à compter les mots, puisque chaque valeur est 1), et écrire la paire (clé,somme)

4. https://fr.wikipedia.org/wiki/Sensibilité_à_la_casse

obtenue dans le contexte.

Exercice 7

Une fois ces deux classes et méthodes définies, il faut les relier en décrivant le job MapReduce. Cela se fait au travers d'une instance des classes `org.apache.hadoop.conf.Configuration` et `org.apache.hadoop.mapreduce.Job`. Effectuez cette configuration dans une méthode `main` à rajouter à votre classe `WordCount`.

On crée d'abord l'instance de `Configuration` et on l'utilise pour instancier un `Job` :

```
Configuration config = new Configuration();
Job job = new Job(conf,"Custom Word Count Program");
```

La ligne suivante permet d'identifier le fichier JAR concerné sur la base d'une classe :

```
job.setJarByClass(WordCount.class);
```

On spécifie ensuite les types des `Mapper` et `Reducer` à utiliser :

```
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);
```

La ligne suivante permet d'indiquer quelle classe utiliser pour lire le(s) fichier(s) d'entrée et les découper en blocs destinés à être traités par les esclaves :

```
job.setInputFormatClass(TextInputFormat.class);
```

La valeur `TextInputFormat.class` signifie que Hadoop va lire un fichier texte ligne par ligne. Le numéro de ligne sera automatiquement utilisé comme clé, et la ligne de texte elle-même comme valeur. Le tout est transmis au mapper. Il s'agit du format par défaut : en l'absence d'instruction `setInputFormatClass`, c'est le format qui sera attendu en entrée. D'autres formats d'entrée prédéfinis sont disponibles : `FixedLengthInputFormat`, `KeyValueTextInputFormat`, `NLineInputFormat`, etc. (cf. la documentation de la classe `FileInputFormat`) Il est aussi possible de définir son propre format.

Même chose pour la classe destinée à enregistrer les résultats finaux du traitement dans un fichier de sortie :

```
job.setOutputFormatClass(TextOutputFormat.class);
```

Là aussi, il s'agit du format par défaut, et il est possible d'en définir d'autres si besoin (cf. la documentation de la classe `FileOutputFormat`).

On doit également indiquer les types des clés et valeurs renvoyées par le job (donc par le reducer) :

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
```

Enfin, on configure les chemins des dossiers d'entrée et de sortie, qui sont supposés passés en paramètres de l'application :

```
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

Il ne reste plus qu'à lancer le job, en renvoyant un code d'erreur le cas échéant :

```
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

Sortie terminal : notez que dans un programme Hadoop, `System.out.print` et ses variantes n'affichent pas le texte dans le terminal, mais l'écrivent dans un fichier log placé dans `$HADOOP_HOME/userlogs`. Une sortie terminal est possible seulement depuis la méthode principale, en utilisant le système de log d'Apache. Déclarez d'abord un logger dans la classe concernée :

```
public final static Log LOGGER = LogFactory.getLog(MaClasse.class);
```

où `MaClasse` est la classe concernée. Puis, loggez vos messages de la façon suivante :

```
LOGGER.info("mon message");
```

Exercice 8

Une fois la classe écrite, on peut la compiler et l'exécuter avec Hadoop. Pour la compiler et en faire un fichier JAR :

```
$HADOOP_HOME/bin/hadoop com.sun.tools.javac.Main WordCount.java
jar cf wc.jar WordCount*.class
```

En supposant que les services d'Hadoop (HDFS et YARN, cf. Section 2) soient en cours d'exécution, le programme se lance avec :

```
$HADOOP_HOME/bin/hadoop jar wc.jar WordCount input output
```

où `input` et `output` sont respectivement les dossiers d'entrée et de sortie attendus par le programme.

À titre de vérification, le début et la fin du fichier résultat obtenus pour `wp_20` *doivent* être exactement (si vous avez suivi les consignes à la lettre) :

```
a 159
aarhus 1
abhandlung 1
abondamment 1
abord 3
abordent 1
aboutir 1
aboutira 1
```

```
abraham 1
abritant 1
abrite 2
abrègement 1
abrégé 4
abrégée 1
absence 3
...
```

```
...
évoquent 1
évoquer 1
évoqués 1
évènements 1
événements 3
évêque 1
êtes 1
```

```
être 76
êtres 1
ë 1
über 4
C 4
N 1
Q 1
R 8
```

Incidemment, on peut remarquer que la façon dont Hadoop classe les clés n'est pas conforme à l'ordre lexicographique employé en français.

Exercice 9

Examinez le log affiché par Hadoop dans la console à l'issue de l'exécution de votre programme. Identifiez les informations suivantes :

- Nombre de fichiers à traiter ;
- Nombre de blocs de données (splits) créés au moment du découpage ;
- Nombre d'instances de `Mapper` et `Reducer` exécutées ;
- Temps total utilisé par ces instances de `Mapper` et `Reducer` ;

On voudrait aussi connaître le temps *effectif* mis pour traiter les données, i.e. temps écoulé entre le moment où vous lancez la commande et celui où vous récupérez la main dans la console. Se trouve-t-il parmi les informations affichées dans la console ? Vérifiez aussi l'historique des jobs disponible sur la page (locale) de YARN accessible avec votre navigateur Web (cf. Section 2.2). Si vous ne trouvez pas cette information, utilisez la commande shell `time` ou mesurez le temps écoulé directement dans le `main` de votre programme.

Exercice 10

Exécutez votre programme sur les dossiers de différentes tailles, relevez les temps de calcul (map, reduce, effectif), présentez-les dans une table, comparez et discutez les valeurs obtenues.

4 Indice de masse corporelle

Dans le cadre de l'informatique décisionnelle et de la fouille de données, MapReduce peut être utile pour effectuer du pré-traitement sur les données brutes, c'est-à-dire pour les préparer à être manipulées par des algorithmes tels que des classificateurs, par exemple. On peut en particulier filtrer ces données, i.e. écarter les instances qui ne respectent pas certains critères, ou bien transformer ces données (par exemple : normaliser un champ numérique).

Exercice 11

Le dossier `socio` fourni avec le sujet contient 4 versions d'une table qui décrit une population d'individus grâce aux 5 champs suivants :

- `id` : identifiant unique (entier),

- `age` : âge en année (entier),
- `height` : taille en cm (entier),
- `weight` : masse en kg (entier),
- `sex` : sexe (catégorie : `female` vs. `male`).

La table complète contient 1 000 000 d'individus, et chaque autre table est constituée seulement d'une partie de ces individus. Notez qu'à la différence de l'exemple du décompte de mots, les données sont ici toutes contenues dans le même fichier (et non pas dans une multitude de fichiers). Copiez toutes ces tables dans le système HDFS.

Exercice 12

Écrivez un programme MapReduce permettant de calculer l'indice de masse corporelle (IMC) moyen, son écart-type, son minimum et son maximum, seulement pour les individus adultes, en les distinguant en fonction de leur sexe. Autrement dit, on veut obtenir en fin de traitement un quadruplet de valeurs (moyenne, écart-type, minimum, maximum) pour les femmes, et un autre quadruplet pour les hommes.

Le mapper doit traiter une ligne à la fois : décomposition, récupération des données pertinentes, filtrage des instances selon les critères ciblés. Le reducer doit calculer les statistiques demandées, pour une catégorie d'individus à la fois.

Que remarquez-vous d'inhabituel dans le fonctionnement de l'`Iterable` transmis par Hadoop au reducer ? Comment y remédier ?

Pour vérification, les statistiques obtenues pour le plus petit fichier sont :

- Femmes : $\mu = 24{,}14$, $\sigma = 4{,}74$, $\min = 6{,}53$, $\max = 39{,}03$;
- Hommes : $\mu = 25{,}16$, $\sigma = 4{,}53$, $\min = 14{,}02$, $\max = 39{,}30$.

> **Note :** Il est possible que vous ayez cette fois à spécifier explicitement au job les types utilisés par le mapper. Cela se fait au moyen de `Job.setMapOutputKey` et `setMapOutputKey`.

Exercice 13

Relevez les temps de calcul et nombres de splits obtenus pour les différentes tailles de données, et présentez-les dans une table. Comparez les performances obtenues.

Que remarquez-vous en ce qui concerne le nombre de splits ? Comment l'expliquez-vous ? Identifiez le paramètre de Hadoop permettant de contrôler (indirectement) le nombre de splits, et comparez les temps de calculs obtenus en faisant décroître leur taille. Là-aussi, utilisez une table pour présenter vos résultats, puis discutez-les.