# Introduction à Hadoop + Map/Reduce

# Certificat Big Data

TME Hadoop

Ce TME a pour objectif de se familiariser avec le framework distribué Apacha Hadoop. Dans un premier temps, nous considèrerons les différentes commandes permettant le chargement et l'exportation de données sur le système de fichiers virtuel qu'il utilise : le système HDFS. Ensuite nous développerons un premier programme JAVA Hadoop réalisant un simple comptage de mots dans des fichiers d'entrée de manière parallèlisée grâce au fonctionnalités MapReduce proposées par Hadoop. Enfin, un classique classifieur NaiveBayes parallèlisé pourra être envisagé.

## 1 Prise en Main de HDFS

Quelques commandes après s'être loggué à la machine en tant que hduser (par la commande *ssh hduser@XXX.XXX.XXX.XXX* ):

- *jps* : permet de lister les services java lancés.
- *start-dfs.sh* : permet de lancer les services NameNode, DataNode et SecondaryNameNode. Ces services sont indispensables pour la gestion du système de fichier hdfs.
- *start-yarn.sh* : permet de lancer les services RessourceManager et NodeManager. Ces services sont indispensables pour les fonctionnalités Map Reduce.

**!! Dans notre TP, hadoop est déjà démarré, pas besoin de le démarrer !!**

### 1.1 Chargement et exportation de données à partir du shell

La commande *hadoop fs* permet de lister les commandes disponibles sur le système de fichiers:

```
[cloudera@localhost Desktop]$ hadoop fs
Usage: hadoop fs [generic options]
        [-cat [-ignoreCrc] <src> ...]
        [-chgrp [-R] GROUP PATH...]
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-copyFromLocal <localsrc> ... <dst>]
        [-copyToLocal [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-count [-q] <path> ...]
        [-cp <src> ... <dst>]
        [-df [-h] [<path> ...]]
        [-du [-s] [-h] <path> ...]
        [-expunge]
        [-get [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-getmerge [-nl] <src> <localdst>]
        [-help [cmd ...]]
        [-ls [-d] [-h] [-R] [<path> ...]]
        [-mkdir [-p] <path> ...]
        [-moveFromLocal <localsrc> ... <dst>]
        [-moveToLocal <src> <localdst>]
        [-mv <src> ... <dst>]
        [-put <localsrc> ... <dst>]
        [-rm [-f] [-r|-R] [-skipTrash] <src> ...]
        [-rmdir [--ignore-fail-on-non-empty] <dir> ...]
        [-setrep [-R] [-w] <rep> <path/file> ...]
        [-stat [format] <path> ...]
        [-tail [-f] <file>]
        [-test -[ezd] <path>]
        [-text [-ignoreCrc] <src> ...]
        [-touchz <path> ...]
        [-usage [cmd ...]]
```

Les noms des commandes et leurs fonctionnalités ressemblent énormément à celles du shell Unix. L'option *help* fournit des informations complémentaires sur une commande précise. La documentation officielle, consultable à l'adresse `http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/FileSystemShell.html`, recense les commandes du shell et en donne de courtes descriptions.

Dans cette section, nous allons utiliser les commandes du shell Hadoop de façon à importer des données dans HDFS et à exporter des données à partir de HDFS. Ces commandes sont souvent employées pour charger des données appropriées, à télécharger des données traitées, à maintenir le système de fichiers et à consulter le contenu des dossiers. Il est donc indispensable de les connaître pour exploiter efficacement HDFS.

Réalisez les opérations minimales suivantes pour prendre en main hdfs :

1. Créer un dossier dans HDFS
2. Placer un fichier dans le repertoire créé
3. Lister le contenu du repertoire hdfs
4. Récupérer le fichier placé sur hdfs et l'enregistrer dans un fichier local

Note: Si toutes les commandes aboutissent à l'affichage du message "No such file or directory", il est possible que la répertoire racine n'existe pas pour hadoopuser. Dans ce cas, créez le par la commande : *hadoop fs -mkdir -p /user/hadoopuser*.

### 1.2 Chargement et exportation de données à partir de Java

On souhaite réaliser les mêmes opérations que précédemment mais à partir d'un programme Java dans une classe *HdfsManagement*. Pour cela on utilise l'objet FileSystem récupéré par :

```
Configuration conf=new Configuration();
FileSystem fs=FileSystem.get(conf);
```

Cet objet propose les méthodes suivantes:

- *listStatus* : retourne un tableau des fichiers contenus au chemin hdfs dénoté par l'objet Path passé en paramètre
- *create* : crée le chemin hdfs correspondant à l'objet Path passé en paramètre
- *copyFromLocalFile* : copie l'arborescence pointée par le chemin local source à l'adresse hdfs spécifiée par le chemin destination passée en paramètre
- *moveToLocalFile* : copie l'arborescence pointée par le chemin hdfs source à l'adresse locale spécifiée par le chemin destination passée en paramètre

Pour information, voici une liste d'imports hadoop dont vous pouvez avoir besoin:

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.LocatedFileStatus;
import org.apache.hadoop.fs.RemoteIterator;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FileUtil;
import org.apache.hadoop.fs.Path;
```

Une fois la classe *HdfsManagement* écrite, il s'agit de la compiler en jar executable. En supposant que les sources sont dans un repertoire src, on peut définir un script de compilation makeJar.sh tel que (vous pouvez retrouver ce script dans le repertoire */Vrac/lamprier/FDMS*):

```
mkdir bin
find src -name "*.java" > "sources.txt"
l1="/usr/local/hadoop/share/hadoop/common/hadoop-common-2.5.0.jar"
l2="/usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.5.0.jar"
l3="/usr/local/hadoop/share/hadoop/common/lib/commons-cli-1.2.jar"
l4="/usr/local/hadoop/share/hadoop/common/lib/hadoop-annotations-2.5.0.jar"
javac -classpath $l1":"$l2":"$l3":"$l4":bin" -d bin @sources.txt
echo Main-Class: $1 > MANIFEST.MF
echo Class-Path: ./bin/ >> MANIFEST.MF
jar cvmf MANIFEST.MF Main.jar -C bin .
```

La compilation se fait alors par :

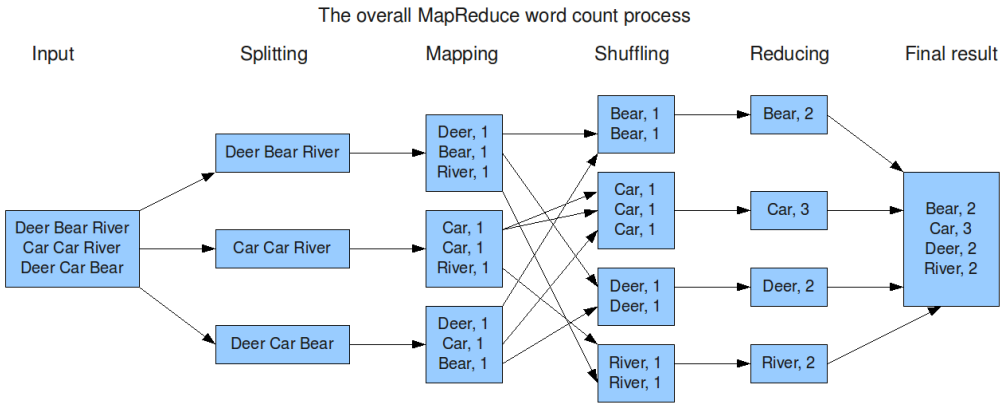
```
sh makeJar.sh HdfsManagement
```

Enfin, on peut lancer le jar Main.jar sur hadoop par la commande :

```
hadoop jar Main.jar
```

## 2 MapReduce : WordCount

Nous proposons de commencer l'initiation à la programmation MapReduce par le très classique programme de comptage de mots *WordCount*. Il s'agit de compter les mots de fichiers texte plaçés dans un repertoire HDFS. Nous définirons pour cela une classe *WordCount* qui sera la classe principale du programme (comportant le main) et deux classes *WordCountMapper* et *WordCountReducer* qui correspondent respectivement à l'étape Map et à l'étape Reduce du programme.




La classe *WordCountMapper* hérite de *Mapper<Object, Text, Text, IntWritable>* et possède une méthode *map(Object offset, Text value, Context context)* qui découpe en mots la ligne de texte contenue dans la variable value passée en paramètre et émet pour chacun d'entre eux un couple <mot,new IntWritable(1)> en appelant la méthode *write* de l'objet context.

L'objectif du Reducer est alors d'additionner les valeurs des couples de même clé. On écrit alors une classe *WordCountReducer* qui hérite de *Reducer<Text, IntWritable, Text, Text>* et définit une méthode *reduce(Text key, Iterable<IntWritable> values,Context context)* qui somme les valeurs contenues dans la liste values passée en paramètre et émet le résultat sous forme d'un couple <mot,nombre d'occurences> en utilisant la méthode *write* de l'objet context.

Il reste alors à définir le job, déclarer le Mapper et le Reducer ainsi que les fichiers d'entrée en incluant les lignes suivantes dans le main de la classe *WordCount*:

```
Configuration conf=new Configuration();
FileSystem fs=FileSystem.get(conf);
Job job=Job.getInstance(conf,"Compteur de mots v1.0");
job.setMapperClass(WordCountMapper.class);
job.setReducerClass(WordCountReducer.class);
job.setJarByClass(WordCount.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job,new Path(args[0]));
FileOutputFormat.setOutputPath(job,new Path(args[1]));
job.waitForCompletion(true);
```

Pour information, les imports hadoop suivants sont nécessaires ici:

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
```

## 3 MapReduce : classifieur NaiveBayes

Un modèle probabiliste de classification classique considère la probabilité conditionnelle suivante:

$$p(c|x_1, \ldots, x_n)$$

où $c \in C$ est une classe parmi les classes possible pour un objet donné $x$ et $x_1, \ldots, x_n$ un vecteur de ses $n$ caractéristiques. On considère ici des vecteurs binaires $\{0,1\}^n$
La classe prédite par le modèle est alors la classe qui maximise cette probabilité conditionnelle:

$$C(x) = \arg\max_{c \in C} p(c|x_1, \ldots, x_n)$$

À l'aide du théorème de Bayes, nous écrivons :

$$p(c|x_1, \ldots, x_n) = \frac{p(c)\ p(F_1 = x_1, \ldots, F_n = x_n|c)}{p(x_1, \ldots, x_n)}.$$

avec $F_i$ la valeur d'une caractéristique binaire donnée i.
On peut alors considérer le classifieur équivalent suivant :

$$C(x) = \arg\max_{c \in C} \frac{p(c)\ p(F_1 = x_1, \ldots, F_n = x_n|c)}{p(x_1, \ldots, x_n)}$$

dans lequel on peut ignorer $p(x_1, \ldots, x_n)$ indépendant de la classe et donc ne perturbant pas le choix de la classe de probabilité maximale:

$$C(x) = \arg\max_{c \in C} \quad p(c)\ p(F_1 = x_1, \ldots, F_n = x_n|c)$$

En considérant une hypothèse d'indépendance de réalisation des différentes caractéristiques $F_1, \ldots, F_n$ et en passant au logarithme de la probabilité conditionnelle $p(C|x_1, \ldots, x_n)$, on peut alors considérer la formulation suivante:

$$C(x) = \arg\max_{c \in C} \quad \log p(c) + \sum_{i=1}^{n} \log p(F_i = x_i|c)$$

### 3.1 Inférence

En considérant que l'on dispose pour chaque classe $C$ de sa probabilité a priori $p(C)$ et des probabilités conditionnelles marginales $p(F_i|C)$ des différentes caractéristiques $F_i$ pour $i \in \{1 \dots n\}$ selon cette classe $C$, développer un modèle MapReduce de classification NaiveBayes sur Hadoop qui prédit la classe la plus probable d'instances contenues dans un ou plusieurs fichiers texte (une instance par ligne).

Que doit faire le Mapper pour ces prédicitions de classe? Et le Reducer ?

Tester le modèle de prédiction développé sur les données contenues dans les fichiers du repertoire */Vrac/lamprier/FDMS/WebKB/content* (4 fichiers texte). Chaque ligne de ces fichiers représente un site Web accompagné d'une liste de caractéristiques binaires et d'un label à prédire.

### 3.2 Apprentissage

En posant la log-vraissemblance suivante sur l'ensemble des données d'entraînement $X$ :

$$L(\theta) = \sum_{x \in X} \log p(C_x) + \sum_{x \in X} \sum_{i=1}^{n} \log p(F_i = x_i | C_x)$$

avec $C_x$ correspond à la classe de l'exemple $x$, la maximisation de cette quantité sous les contraintes $\sum_{c \in C} p(c) = 1$ et $\forall i \in \{1 \dots n\} \forall c \in C, p(F_i = 0|c) + p(F_i = 0|c) = 1$ permet l'estimation des paramètres selon :

$$\forall c \in C : p(c) = \frac{|\{x \in X, C_x = c\}|}{|X|}$$

$$\forall i \in \{1 \dots n\} \forall c \in C : p(F_i|c) = \frac{|\{x \in X, x_i = F_i \wedge C_x = c\}|}{|\{x \in X, C_x = c\}|}$$

Que doit faire le Mapper pour cette tâche d'estimation des paramètres? Et le Reducer?

Développer ce processus d'apprentissage pour le modèle de classification et expérimenter l'estimation de paramètres sur des données des fichiers du repertoire */Vrac/lamprier/FDMS/WebKB/content*.