Lab1: programmation avec l'API HDFS/MapReduce

L'objectif de ce TP est de :

- s'initier à la programmation avec l'API HDFS
  - ***** Lire/ecrire un fichier sur HDFS
- s'initier à la programmation avec L'API mapreduce
  - ***** Implémenter l'exemple **WordCount** en **Java**.
  - ***** Exécuter MapReduce en **Python** avec **Hadoop Streaming**.

I. Démarrer le Cluster Hadoop

1. Démarrage des containers

- Démarrer le cluster hadoop créé précédemment

```
docker start hadoop-master hasdoop-slave1 hadoop-slave2
```

2. Accéder au master

Entrer dans le conteneur master pour commencer à l'utiliser

```
docker exec -it hadoop-master bash
```

3. Démarrer hadoop et yarn

lancer hadoop et yarn en utilisant un script fourni appelé start-hadoop.sh.

```
./start-hadoop.sh
```

- A la fin du démarrage, vérifier si hadoop et yarn ont démarré correctement. Pour ce faire :
  - Ressource manager web UI `http://localhost:8088`
  - HDFS web UI: `http://localhost:9870`
- utiliser la commande shell `jps` pour vérifier si les processus en relation sont en cours d'exécution

II. Programmation avec l'api HDFS

Dans cette partie, nous allons développer quelques jar pour la manipulation des fichiers avec l'api HDFS.

1. Installation de l'environnement de développement

- Outils et environnement dont on aura besoin :
  - Visual Studio Code (ou tout autre IDE de votre choix)
  - Java Version 1.8
  - Unix-like ou Unix-based Systems
- Créer un projet Maven (no archtype) dans VSCode (ajouter les extensions nécessiares ***Maven for Java*** *et* ***Extension Pack for Java)***
  - choisir **no archetype / groupId :** *edu.ensias.hadoop* / **artifactId** *hadoop_lab*
  - Créer un répertoire ***BigdataLabs*** où vous allez mettre votre projet **hadoop_lab**
  - ajouter les dépendances au fichier **pom.xml**

```
<properties>
	<maven.compiler.source>1.8</maven.compiler.source>
	<maven.compiler.target>1.8</maven.compiler.target>
</properties>
<dependencies>
	<dependency>
		<groupId>org.apache.hadoop</groupId>
		<artifactId>hadoop-hdfs</artifactId>
		<version>3.2.0</version>
	</dependency>
	<dependency>
		<groupId>org.apache.hadoop</groupId>
		<artifactId>hadoop-common</artifactId>
		<version>3.2.0</version>
	</dependency>
	<dependency>
		<groupId>org.apache.hadoop</groupId>
		<artifactId>hadoop-mapreduce-client-core</artifactId>
		<version>3.2.0</version>
		</dependency>
	</dependencies>
<build>
	<plugins>
		<plugin>
			<groupId>org.apache.maven.plugins</groupId>
			<artifactId>maven-compiler-plugin</artifactId>
			<version>3.8.1</version>
		</plugin>
		<plugin>
			<groupId>org.apache.maven.plugins</groupId>
			<artifactId>maven-jar-plugin</artifactId>
			<version>3.2.2</version>
			<configuration>
				<archive>
					<manifest>
						<mainClass>edu.ensias.hadoop.Main</mainClass>
					</manifest>
				</archive>
				<finalName>hadoop-app</finalName>
			</configuration>
		</plugin>
	</plugins>
</build>
</project>
```

- Créer un package hdfslab sous le répertoire src/main/java/edu/ensias/hadoop/

2. Premier exemple

Créer une première classe `HadoopFileStatus` qui permet de retourner le nom et la taille d'un fichier sur hdfs et de changer son nom

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;
public class HadoopFileStatus {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Configuration conf = new Configuration();
		FileSystem fs;
		try {
			fs = FileSystem.get(conf);
		Path filepath = new Path("/user/root/input", "purchases.txt") ;
		FileStatus infos = fs.getFileStatus(filepath);
		if(!fs.exists(filepath)){
			System.out.println("File does not exists");
			System.exist(1);
		}
		System.out.println(Long.toString(infos.getLen())+" bytes");
		System.out.println("File Name: "+filepath.getName());
		System.out.println("File Size: "+status.getLen())
		System.out.println("File owner: "+status.getOwner())
		System.out.println("File permission: "+status.getPermission())
		System.out.println("File Replication: "+status.getReplication());
		System.out.println("File Block Size: "+status.getBlockSize());
		BlockLocation[] blockLocations = fs.getFileBlockLocations(status, 0,
		status.getLen());
		for(BlockLocation blockLocation : blockLocations) {
		    String[] hosts = blockLocation.getHosts();
		    System.out.println("Block offset: " + blockLocation.getOffset());
		    System.out.println("Block length: " + blockLocation.getLength());
		    System.out.print("Block hosts: ");
		    for (String host : hosts) {
		        System.out.print(host + " ");
		    }
		    System.out.println();
		}

		fs.rename(filepath, new Path("/user/root/input", "achats.txt") ;

		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
}
```

- à travers le fichier pom.xml
  - nommer le jar à créer par HadoopFileStatus.jar
  - N'oublier pas de spécifier la classe du main (HadoopFileStatus)
- Copier le jar créé vers le dossier de partage /hadoop_project
- sur l'invité de commande shell de votre container lancer la commande

hadoop jar /shared_volume/HadoopFileStatus.jar

- modifier la classe HadoopFileStatus (le jar évidemment) pour qu'elle puisse lire les paramètres *chemin_fichier nom_fichier nouveau_nom_fichier* lors de l'exécution

Exemple

```
hadoop jar /shared_volume/HadoopFileStatus.jar /user/root/input purchases.txt achats.txt
```

3. Lire un fichier sur HDFS

la deuxième classe à créer une classe java (ReadHDFS) qui permet de lire un fichier sur HDFS

Créer une classe qui permet de lire les informations se trouvant dans un fichier sur HDFS

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;

public class ReadHDFS {
                public static void main(String[] args) throws IOException{

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

                Path nomcomplet = new Path("/user/root/purchases.txt");
                FSDataInputStream inStream = fs.open(nomcomplet);
                InputStreamReader isr = new InputStreamReader(inStream);
                BufferedReader br = new BufferedReader(isr);
                String line= null;
                while((line = br.readLine())!=null) {
                      System.out.println(line);
                }
                System.out.println(line);
                inStream.close();
                fs.close();
          }

}
```

- Créer un fichier jar que vous allez nommer ***ReadHDFS.jar***
- Copier le jar créé vers le dossier de partage /hadoop_project
- sur l'invité de commande shell de votre container lancer la commande

```
hadoop jar /shared_volume/ReadHDFS.jar
```

- Modifier la classe pour qu'elle puisse lire tout le fichier
- Modifier la classe pour qu'on puisse passer le nom du fichier à lire en paramètre

Exemple

```
hadoop jar /shared_volume/ReadHDFS.jar ./purchases.txt
```

4. Ecrire un fichier sur HDFS

la troisième classe à créer est une classe java (WriteHDFS) qui permet de créer un fichier sur HDFS

```
import java.io.IOException;
```

```
import org.apache.hadoop.conf.*;
import org.apache.hadoop.fs.*;

public class HDFSWrite {
     public static void main(String[] args) throws IOException {

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

          Path nomcomplet = new Path(args[0]);
          if (! fs.exists(nomcomplet)) {
     FSDataOutputStream outStream = fs.create(nomcomplet);
               outStream.writeUTF("Bonjour tout le monde !");
               outStream.writeUTF(args[1]);
               outStream.close();
          }
          fs.close();
     }
}
```

- Créer un fichier jar que vous allez nommer ***WriteHDFS.jar***
- Copier le jar créé vers le dossier de partage /hadoop_project
- sur l'invité de commande shell de votre container lancer la commande

```
hadoop jar /shared_volume/WriteHDFS.jar ./input/bonjour.txt
```

III. Programmation avec l'api MapReduce

L'objectif de ce TP est de simuler l'exemple wordcount vu dans le cours. Pour rappel, le job à créer permet de de **compter le nombre d'occurrences de chaque mot** présent dans un fichier texte. Ce traitement est réalisé en deux phases principales :

- **Phase de Mapping** : le texte est découpé en mots. Pour chaque mot identifié, le programme génère une paire clé/valeur sous la forme *(mot, 1)*, indiquant que ce mot est apparu une fois.
- **Phase de Reducing** : les paires issues du mapping sont regroupées par mot (clé). Le réducteur applique une fonction d'agrégation (addition) sur toutes les valeurs associées à chaque mot, ce qui permet d'obtenir le nombre total d'occurrences du mot dans le document.
- Créer un package mapreducelab sous le répertoire src/main/java/edu/ensias/hadoop/

1)classe Mapper

Créer une première classe `Mapper`

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;

public class TokenizerMapper   extends Mapper<Object, Text, Text, IntWritable>{
     private final static IntWritable one = newIntWritable(1);
     private Text word = new Text();
```

```
	public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
		System.out.println(key.toString());
		StringTokenizer itr = new StringTokenizer(value.toString());
		while (itr.hasMoreTokens()) {
			word.set(itr.nextToken());
			context.write(word, one);
		}
	}
}
```

2)classe reducer

Créer une la classe `reducer`

```
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Reducer;

public class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
	private IntWritable result = new IntWritable();

	public void reduce(Text key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException {
		int sum = 0;
		for (IntWritable val : values) {
			sum += val.get();
		}
		result.set(sum);
		context.write(key, result);
	}
}
```

3)classe Principale

Créer une la classe `qui permettra de lancer le job`

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {
	public static void main(String[] args) throws Exception {
		// TODO Auto-generated method stub
		Configuration conf = new Configuration();
	       Job job = Job.getInstance(conf, "word count");
```

```
        // classe principale
        job.setJarByClass(WordCount.class);

        // classe qui fait le map
        job.setMapperClass(TokenizerMapper.class);

        // classe qui fait le shuffling et le reduce
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);

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

        // spécifier le fichier d'entrée
        FileInputFormat.addInputPath(job, new Path(args[0]));

     // spécifier le fichier contenant le résultat
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

- Créer un fichier jar que vous allez nommer WordCount.jar
- Copier le jar créé vers le dossier de partage /hadoop_project
- sur l'invité de commande shell de votre container lancer la commande

4)MapReduce avec python

L'objectif est d'implémenter l'exemple wordcount à base de mapreduce en python et de l'utilitaire hadoop streaming. pour ce faire :

- *Écrire le mapper qui implémente la logique map. Il lira les données de STDIN et divisera les lignes en mots, et générera une sortie de chaque mot avec une occurrence égale à 1*

```
#!/usr/bin/env python
import sys
# input comes from standard input STDIN
for line in sys.stdin:
    line = line.strip() #remove leading and trailing whitespaces
    words = line.split() #split the line into words and returns as a list
    for word in words:
        #write the results to standard output STDOUT

        print('%s\t%s' % (word,1)) #print the results
```

- *Vous pouvez tester le mapper.py sur votre machine*

```
cat alice.txt | python mapper.py
```

- Écrire le fichier **reducer.py** qui implémente la logique reduce. Il lira la sortie de mapper.py à partir de l'entrée standard et agrégera l'occurrence de chaque mot et écrira la sortie finale sur STDOUT

```
#!/usr/bin/env python
```

```
from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

for line in sys.stdin:
	line = line.strip() # remove leading and trailing whitespace
	# splitting the data on the basis of tab provided in mapper.py
	word, count = line.split('\t', 1)
	# convert count (currently a string) to int
	try:
		count = int(count)
	except ValueError:# ignore/discard this line if count is not a number
		continue

# Hadoop sorts map output by key (word) before it is passed to the reducer
	if current_word == word:
		current_count += count
	else:
		if current_word:
			# write result to STDOUT
			print('%s \t %s' % (current_word, current_count))
		current_count = count
		current_word = word

# output the last word
if current_word == word:
	print('%s\t%s' % (current_word, current_count))
```

- Vérifier si le reducer fonctionne correctement

```
cat alice.txt | python mapper.py | sort -k1,1 | python reducer.py
```

- pour exécuter le mapper.py et reducer.py,
- ouvrir le terminal du container master
- localiser le fichier JAR de l'utilitaire hadoop streaming.

```
find / -name 'hadoop-streaming*.jar'
```

Le chemin devrait ressembler à PATH/hadoop-streaming-3.2.1.jar

/opt/hadoop-3.2.1/share/hadoop/tools/lib/hadoop-streaming-3.2.1.jar

- finalement exécuter le programme map/reduce avec la commande suivante

```
hadoop jar /opt/hadoop-3.2.1/share/hadoop/tools/lib/hadoop-streaming-3.2.1.jar \
-files chemin/mapper.py,chemin/reducer.py  -mapper "python3 mapper.py" \
 -reducer "python3 reducer.py" \
-input chemin/inputfile -output chemin/outputfolder
```

- vérifier les résultats de l'exécution sur HDFS

IV. Initialiser Git et Github

- Sur la ligne de commande, accéder à votre répertoire ***BigdataLabs.***
- **Vérifier l'installation de Git**

```
git version
```

- **Configurer votre compte github**

```
git config --global user.name "votre_user_name"
git config --global user.email "votre_user_email"
```

- vérifier la configuration

```
git config --list
```

- **Initialiser le dépot git de votre projet**

```
git init
```

- **ajouter le dépôt distant github**

```
git remote add origin https://github.com/username/BIGDATA_ENGINEERING_LABS.git
```

- Premier commit et push

```
git add .
git commit -m "Initialisation du dépôt Big Data Labs"
git branch -M main
git push -u origin main
```

Vérifier sur github

```
A chaque nouveau Lab :
git add lab2
git commit -m "Ajout du lab2 : …"
git push
```

- Sortir de bash de hadoop-master `exit`
- Arrêter les trois conteneurs

```
docker stop hadoop-master hadoop-slave1 hadoop-slave2
```

Exercice ouvert

- Choisir un jeu de donner de votre choix (calls.txt ou bien purchases.txt
- Définir une problématique d'analyse pertinente (ex : identifier les produits les plus vendus)
- Charger le fichier sur HDFS
- développer le job mapreduce
- Analyser les résultats