Filière
Ingénierie des Systèmes Ubiquitaires et Distribués – Cloud et IoT (SUD)

TP 2 : Installation et utilisation d'Apache Spark

Matière
Sécurité dans le cloud et l'IoT

Élèves :
Mahjouba LABAALLI
Fatima OUBELKAS

Enseignant
Mme Dounia ZAIDOUNI

11 novembre 2023

Table des matières

Table des figures

Inroduction

Spark représente un système de traitement rapide et parallèle, offrant des interfaces de haut niveau dans des langages tels que Java, Scala, Python, et R. Il intègre un moteur optimisé qui prend en charge l'exécution de graphes, ainsi qu'un ensemble d'outils avancés incluant Spark SQL pour le traitement de données structurées, MLlib pour l'apprentissage de données, GraphX pour le traitement de graphes, et Spark Streaming pour le traitement des données en continu.

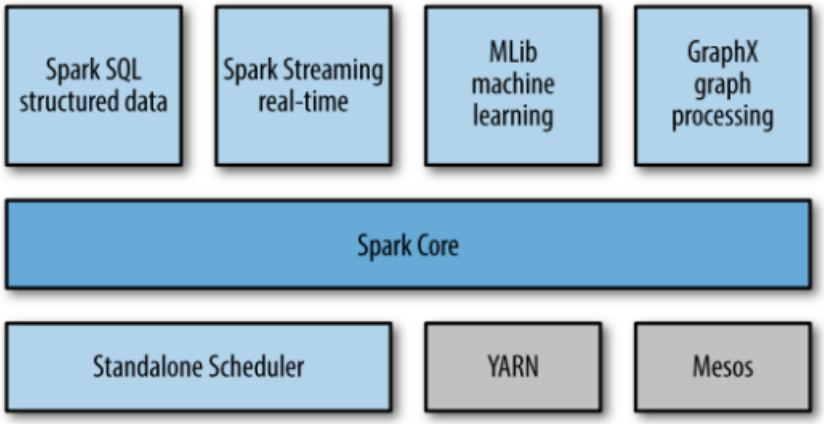

FIGURE 1 – L'écosystème Spark

Les objectifs de ce TP sont les suivants :

— Installation et configuration de Spark dans un nœud unique en local.
— Manipulation des RDDs en utilisant le terminal « pyspark ».
— Connexion de Spark à une distribution de Hadoop.
— Exécution du traitement « Word Count » en utilisant le terminal scala et python.
— Exécution du traitement « Word Count » en utilisant un script python.
— Exécution d'une application Spark Batch en Java.

1 Installation et configuration de Hadoop dans un nœud unique en local

Tout d'abord, créons une machine virtuelle 'ubuntu' en utilisant 'Virtualbox' avec les caractéristiques suivantes : 4096 Mio de mémoire et 20 Go de disque. Suivons ensuite les étapes d'installation et de configuration de Hadoop3 dans un nœud unique, comme détaillé dans la section 1 du TP1. Pour ce faire, nous effectuerons les actions suivantes :

1. Création d'un utilisateur hduser
2. Mise en place de la clé ssh
3. Installation de JAVA 8
4. Installation et configuration d'Apache Hadoop

Finalement, testons notre configuration en tapant : **$ hdfs dfsadmin -report**

```
hduser@labaallioubelkas-VirtualBox:/usr/local/hadoop/etc/hadoop$ hdfs dfsadmin -
report
2023-10-24 22:41:42,215 WARN util.NativeCodeLoader: Unable to load native-hadoop
 library for your platform... using builtin-java classes where applicable
Configured Capacity: 21312888832 (19.85 GB)
Present Capacity: 1900490752 (1.77 GB)
DFS Remaining: 1900466176 (1.77 GB)
DFS Used: 24576 (24 KB)
DFS Used%: 0.00%
Replicated Blocks:
        Under replicated blocks: 0
        Blocks with corrupt replicas: 0
        Missing blocks: 0
        Missing blocks (with replication factor 1): 0
        Low redundancy blocks with highest priority to recover: 0
        Pending deletion blocks: 0
Erasure Coded Block Groups:
        Low redundancy block groups: 0
        Block groups with corrupt internal blocks: 0
        Missing block groups: 0
        Low redundancy blocks with highest priority to recover: 0
        Pending deletion blocks: 0

-------------------------------------------------
Live datanodes (1):

Name: 127.0.0.1:9866 (localhost)
Hostname: labaallioubelkas-VirtualBox
Decommission Status : Normal
Configured Capacity: 21312888832 (19.85 GB)
DFS Used: 24576 (24 KB)
```

FIGURE 2 – Test de la configuration du hadoop

2 Installation et configuration de spark en local

Dans ce TP, nous allons utiliser Spark en local sur la VM ubuntu créée précédemment. Nous allons exécuter Spark sur Hadoop YARN. YARN s'occupera ainsi de la gestion des ressources pour le déclenchement et l'exécution des Jobs Spark.

2.1 Récupération des fichiers sources

Nous pouvons télécharger la version 3.3.0 du spark à partir du lien suivant : https://archive.apache.org/dist/spark/spark-3.3.0/spark-3.3.0-bin-hadoop3.tgz

Ensuite nous récupérons le fichier : « poeme.txt » à partir de la plateforme moodle de l'INPT.

Et voici la totalité des documents nécessaires pour ce TP.

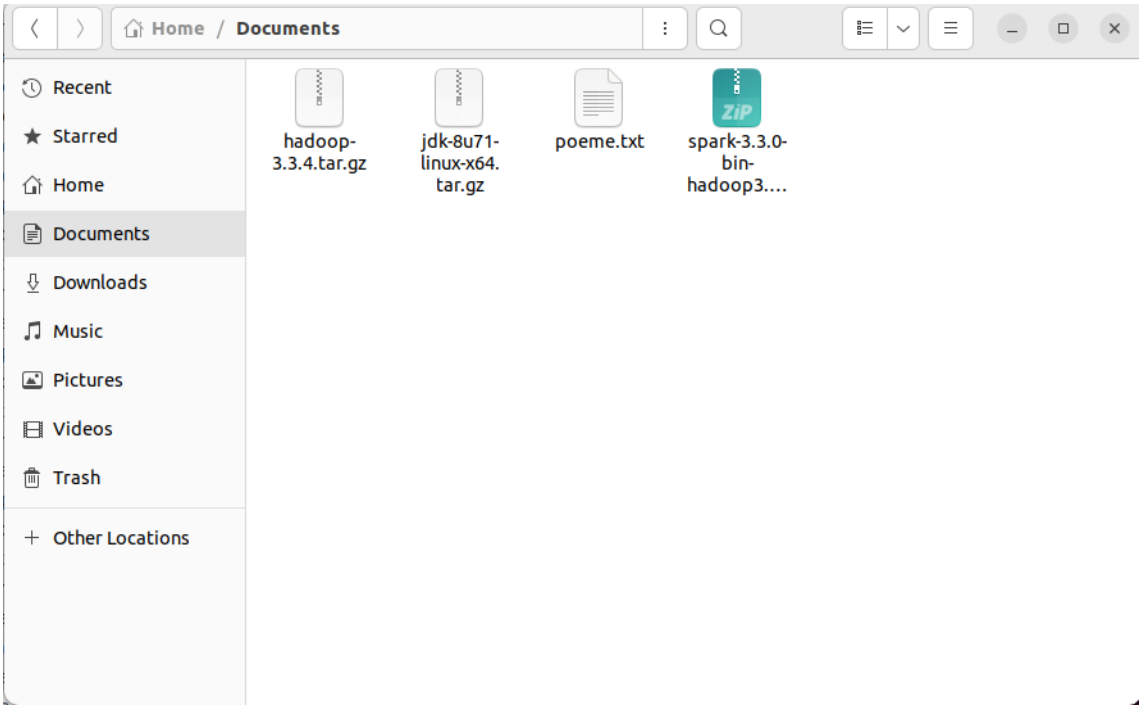

FIGURE 3 – Les fichiers nécessaires pour le TP

Tous ces fichiers sont existes au niveau du lien drive suivant : lien drive

2.2 Décompression du fichier spark récupéré

Pour décompresser le fichier récupéré, accédons au répertoire où le fichier existe et tapons la commande suivante : **$ tar xvzf spark-3.3.0-bin-hadoop3.tgz**

Ensuite, nous pouvons voir qu'un répertoire spark-3.3.0-bin-hadoop3 a été créé ; Nous le renommons en utilisant la commande : **$ mv spark-3.3.0-bin-hadoop3 spark**, et plaçons le dans le répertoire **/usr/local** avec la commande : **$ sudo mv spark /usr/local/**

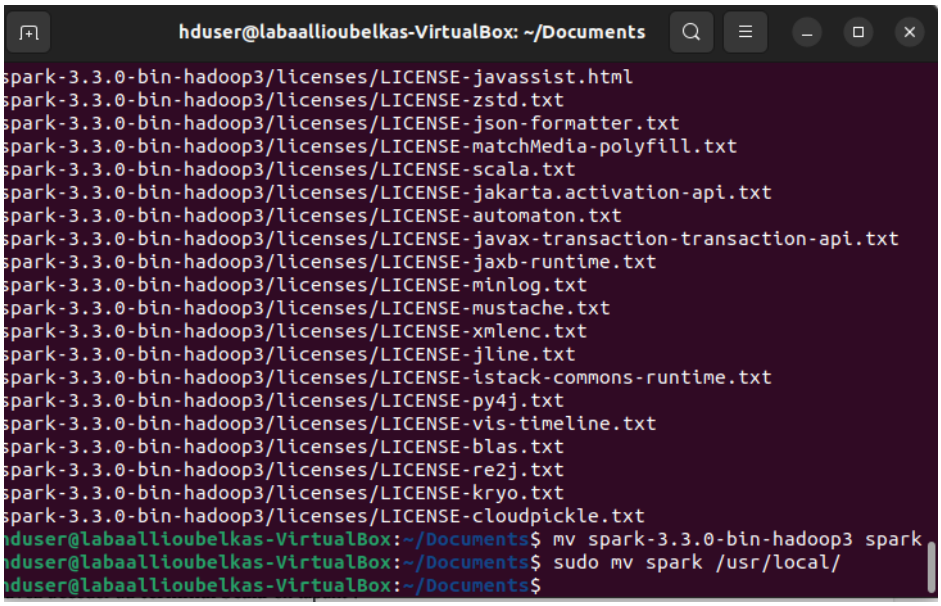

FIGURE 4 – Décompression du fichier spark récupéré

2.3 Configuration du PATH dans le fichier .bashrc

Nous ouvrons le fichier **.bashrc** en utilisant **$vim .bashrc** et modifions le **PATH** en ajoutant les deux lignes suivantes enfin d'ajouter le chemin où se trouve le bin de spark :
export SPARK_HOME=/usr/local/spark
export PATH=$PATH :$SPARK_HOME/bin

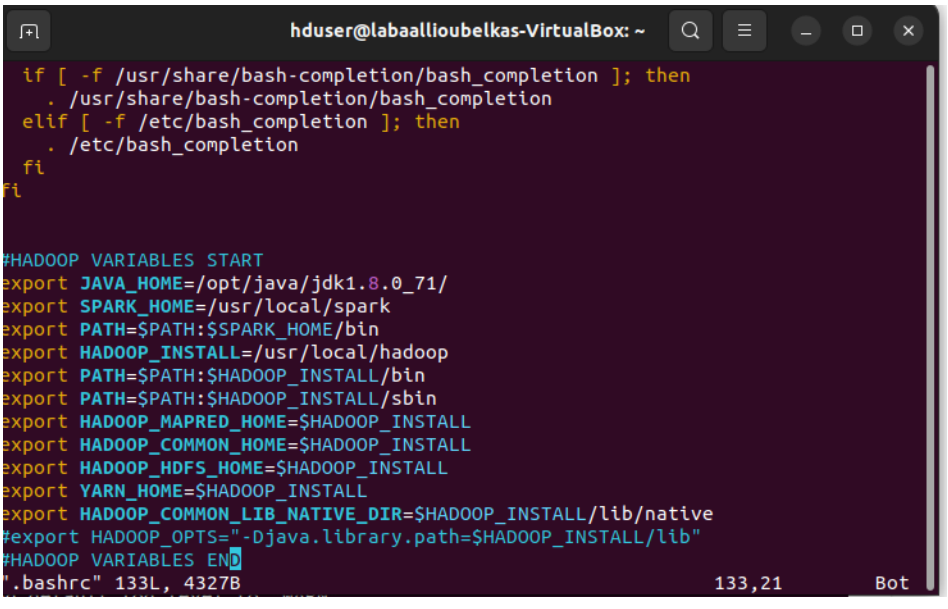

FIGURE 5 – Configuration du PATH dans le fichier .bashrc

En fin, pour sauvegarder la configuration effectuée nous tapons la commande suivante :
$ source .bashrc

2.4 Installation de Python

Pour installer Python, nous tapons dans notre terminal : **$ sudo apt-get install python2**

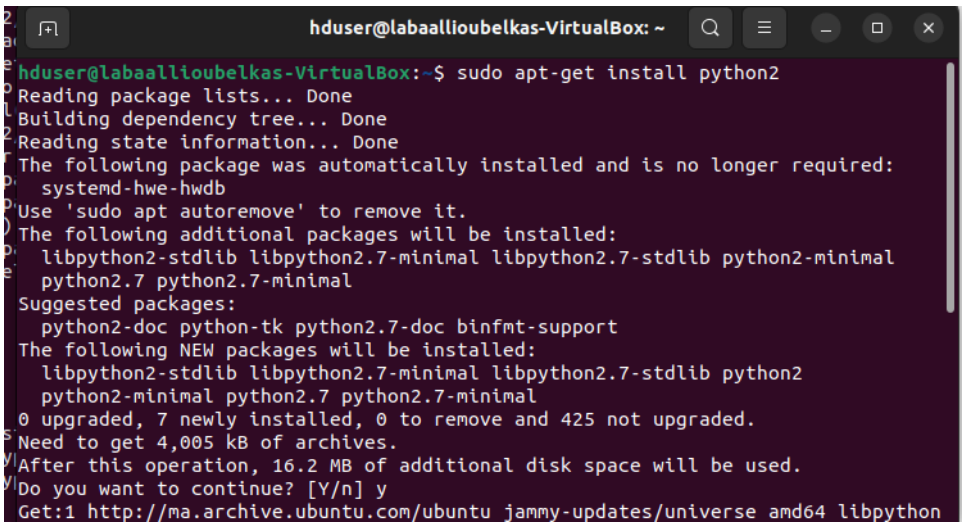

FIGURE 6 – Installation de Python2

À ce stade , nous avons Spark et il est utilisable.

Nous pouvons accéder au terminal Scala en accédant tou d'abord au répértoire spark (**$ cd /usr/local/spark/** puis nous executons le fichier **spark-shell** en tapant : **$ ./bin/spark-shell**

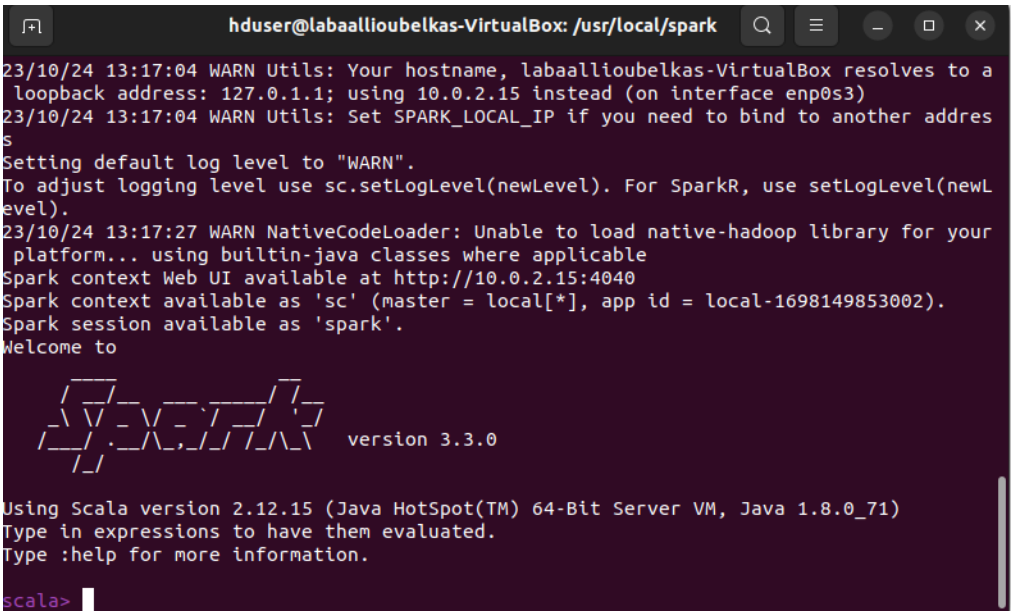

FIGURE 7 – Le terminal Scala

Nous pouvons également accéder au terminal Python en executant le fichier **pyspark** en tapant : **$ ./bin/pyspark**

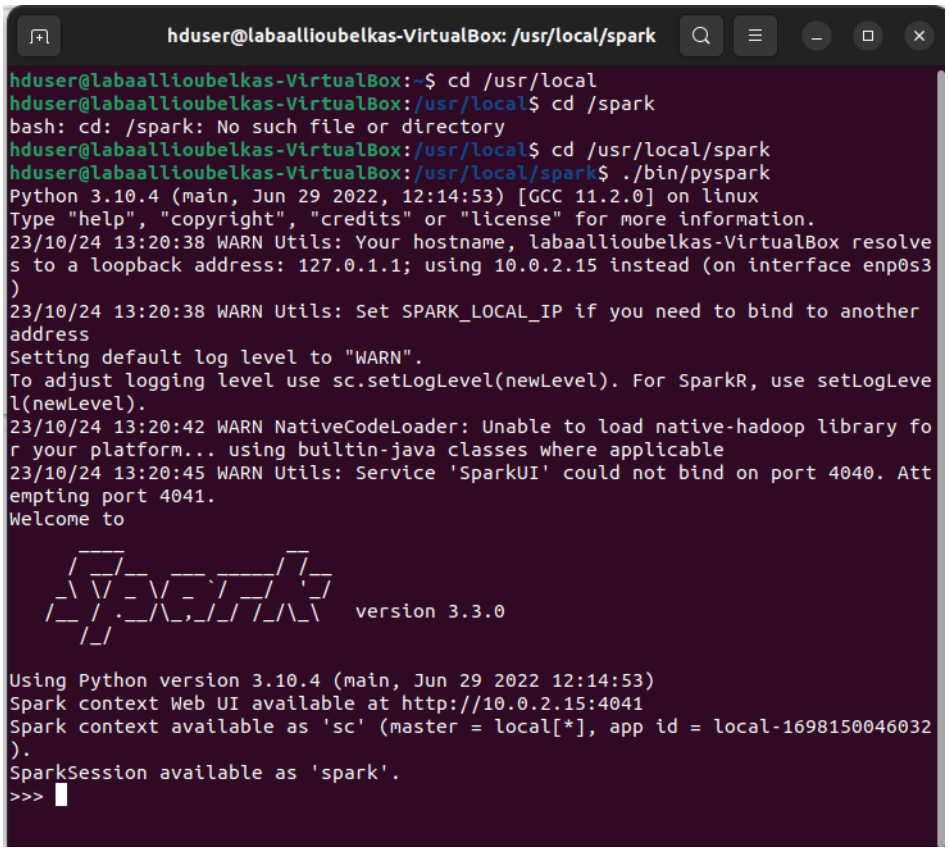

FIGURE 8 – Le terminal pyspark

3 Manipulation des RDDs en utilisant le terminal pyspark

Dans cette section, nous allons appliquer les différents exemples vus dans le cours.

3.1 Enregistrement et chargement des Textfile

Créons un fichier **ValeursINPT.txt** dans le répertoire **usr/local/spark** comme suit :

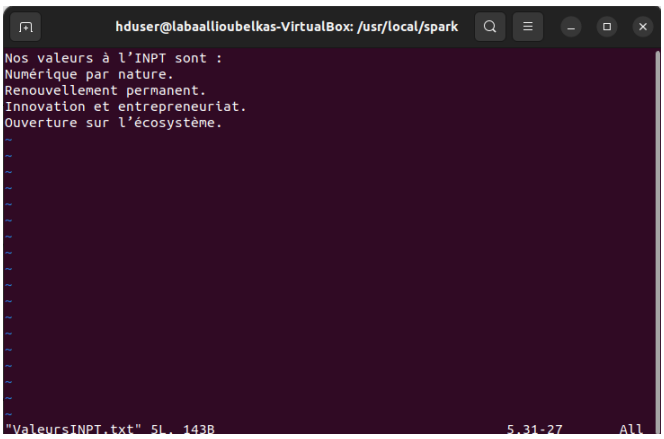

FIGURE 9 – Le fichier **ValeursINPT.txt**

Accédons de nouveau au terminal pyspark (**$ ./bin/pyspark**) et tapons les commandes suivantes :

En premier temps, on récupère le contenu du fichier **ValeursINPT.txt** comme suit :

» **mydata = sc.textFile("file :/usr/local/spark/ValeursINPT.txt")**

Puis, on affiche le contenu en utilisant :

» **for line in mydata.collect() :**
... **print(line)**

```
>>>
>>>
>>> for line in mydata.collect():
...     print(line)
...
Nos valeurs à l'INPT sont :
Numérique par nature.
Renouvellement permanent.
Innovation et entrepreneuriat.
Ouverture sur l'écosystème.
>>>
```

FIGURE 10 – Affichage du contenu de **ValeursINPT.txt** en utilisant le termina pyspark

Nous filtrons maintenant les lignes qui commencent par la lettre **"N"** avec les commandes suivantes :

» **mydata_filt = mydata.filter(lambda s : s.startswith('N'))**
» **mydata_filt.saveAsTextFile("file :/usr/local/spark/values_starts_withN")**

Finalement, nous présentons initialement le nombre total de lignes en utilisant :

» **mydata.count()**

Puis procédons au calcul du nombre de lignes débutant par la lettre **"N"** en tapant :

» **mydata_filt.count()**

```
Nos valeurs à l'INPT sont :
Numérique par nature.
Renouvellement permanent.
Innovation et entrepreneuriat.
Ouverture sur l'écosystème.
>>> mydata_filt = mydata.filter(lambda s: s.startswith('N'))
>>> mydata_filt.saveAsTextFile("file:/usr/local/spark/values_starts_withN")
>>> mydata.count()
5
>>> mydata_filt.count()
2
>>>
```

FIGURE 11 – Filtrage du contenu de **ValeursINPT.txt** en utilisant le termina pyspark

Vérifions alors que dans le répertoire /usr/local/spark qu'un répertoire nommé values_starts_withN est bien crée, ce répertoire doit contenir deux fichiers : part-00000 et _SUCCESS.

```
hduser@labaallioubelkas-VirtualBox:/usr/local/spark$ ls
bin   examples    LICENSE   python     RELEASE          values_starts_withN
conf  jars        licenses  R          sbin             yarn
data  kubernetes  NOTICE    README.md  ValeursINPT.txt
hduser@labaallioubelkas-VirtualBox:/usr/local/spark$ cat values_starts_withN/
cat: values_starts_withN/: Is a directory
hduser@labaallioubelkas-VirtualBox:/usr/local/spark$ cd values_starts_withN/
hduser@labaallioubelkas-VirtualBox:/usr/local/spark/values_starts_withN$ ls
part-00000  part-00001  _SUCCESS
hduser@labaallioubelkas-VirtualBox:/usr/local/spark/values_starts_withN$
```

FIGURE 12 – Le contenu du répertoire **values_starts_withN**

3.2 Enregistrement et chargement des SequenceFiles

Maintenant, nous allons créer, sauvegarder et récupérer un fichier de séquence en suivant les étapes suivantes :

1. Création d'une RDD (Resilient Distributed Dataset) :
   » **rdd = sc.parallelize(range(1, 4)).map(lambda x : (x, "a" * x))**
2. Sauvegarde de la RDD en tant que fichier de séquence :
   » **rdd.saveAsSequenceFile("file :/usr/local/spark/fileseq1")**
3. Récupération et tri du fichier de séquence :
   » **sorted(sc.sequenceFile("file :/usr/local/spark/fileseq1").collect())**

```
>>> rdd = sc.parallelize(range(1, 4)).map(lambda x: (x, "a" * x))
>>> rdd.saveAsSequenceFile("file:/usr/local/spark/fileseq1")
>>> sorted(sc.sequenceFile("file:/usr/local/spark/fileseq1").collect())
[(1, 'a'), (2, 'aa'), (3, 'aaa')]
>>>
```

FIGURE 13 – Enregistrement et chargement des SequenceFiles

Vérifions alors que dans le répertoire /usr/local/spark qu'un répertoire nommé fileseq1 est bien crée, ce répertoire doit contenir deux fichiers : part-00000 et _SUCCESS.

```
hduser@labaallioubelkas-VirtualBox: /usr/local/spark/fileseq1
hduser@labaallioubelkas-VirtualBox:~$ cd /usr/local/spark
hduser@labaallioubelkas-VirtualBox:/usr/local/spark$ ls
bin   examples  kubernetes  NOTICE   README.md  ValeursINPT.txt
conf  fileseq1  LICENSE     python   RELEASE    values_starts_withN
data  jars      licenses    R        sbin       yarn
hduser@labaallioubelkas-VirtualBox:/usr/local/spark$ cat  fileseq1/
cat: fileseq1/: Is a directory
hduser@labaallioubelkas-VirtualBox:/usr/local/spark$ cd fileseq1/
hduser@labaallioubelkas-VirtualBox:/usr/local/spark/fileseq1$ ls
part-00000  part-00001  _SUCCESS
hduser@labaallioubelkas-VirtualBox:/usr/local/spark/fileseq1$
```

FIGURE 14 – Le contenu du répertoire /**fileseq1**

3.3 Utilisation d'une fonction nommée

Nous considérons l'exemple suivant :

1. Définition de la fonction toUpper :
   » **def toUpper(s) :**
   .. return s.upper()
   ..
2. Lecture du fichier texte dans une RDD :
   » **mydata = sc.textFile("file :/usr/local/spark/ValeursINPT.txt")**
3. Application de la transformation map :
   » **mydataupper = mydata.map(toUpper)**
4. Affichage des résultats avec une boucle for :
   » **for line in mydataupper.collect() :**
   .. print(line)

```
>>> def toUpper(s):
...     return s.upper()
...
>>> mydata = sc.textFile("file:/usr/local/spark/ValeursINPT.txt")
>>> mydataupper = mydata.map(toUpper)
>>> for line in mydataupper.collect():
...     print(line)
...
NOS VALEURS À L'INPT SONT :
NUMÉRIQUE PAR NATURE.
RENOUVELLEMENT PERMANENT.
INNOVATION ET ENTREPRENEURIAT.
OUVERTURE SUR L'ÉCOSYSTÈME.
>>>
```

FIGURE 15 – Utilisation d'une fonction nommée

3.4 Utilisation d'une fonction anonyme

Déposons d'abord le fichier : « poeme.txt » dans /usr/local/spark.

1. Lecture du fichier texte dans une RDD :
   » **mydata = sc.textFile("file :/usr/local/spark/poeme.txt")**
2. Application de la transformation map avec une fonction lambda :
   » **mydata_upper = mydata.map(lambda line : line.upper()).take(5)**
3. Affichage des résultats avec une boucle for :
   » **for line in mydata_upper :**
   .. **print(line)**

```
>>> mydata = sc.textFile("file:/usr/local/spark/poeme.txt")
>>> mydata_upper = mydata.map(lambda line: line.upper()).take(5)
>>> for line in mydata_upper:
...     print(line)
...
CELUI QUI CROYAIT AU CIEL
CELUI QUI NY CROYAIT PAS
TOUS DEUX ADORAIENT LA BELLE
PRISONNIERE DES SOLDATS
LEQUEL MONTAIT A LECHELLE
>>>
```

FIGURE 16 – Utilisation d'une fonction anonyme

3.5 Utilisation de « parallelize »

Dans cette partie, nous allons utiliser la fonction parallelize :

1. Création d'une RDD à partir d'une liste :
   » **data = [10, 20, 30, 40, 50, 100, 250]**
   » **distData = sc.parallelize(data)**
2. Réduction de la RDD avec une fonction lambda :
   » **total = distData.reduce(lambda a,b : a + b)**
3. Affichage du résultat :
   » **print (total)**

```
>>> data = [10, 20, 30, 40, 50, 100, 250]
>>> distData = sc.parallelize(data)
>>> total = distData.reduce(lambda a,b: a + b)
>>> print(total)
500
```

FIGURE 17 – Utilisation de « parallelize »

3.6 Utilisation de « wholeTextFiles »

Tout d'abord dans le répertoire /usr/local/spark, créons un répertoire nommé : **json_files** et dans ce répertoire, créons les deux fichiers json : **file1.json** et **file2.json** avec le contenu suivant :

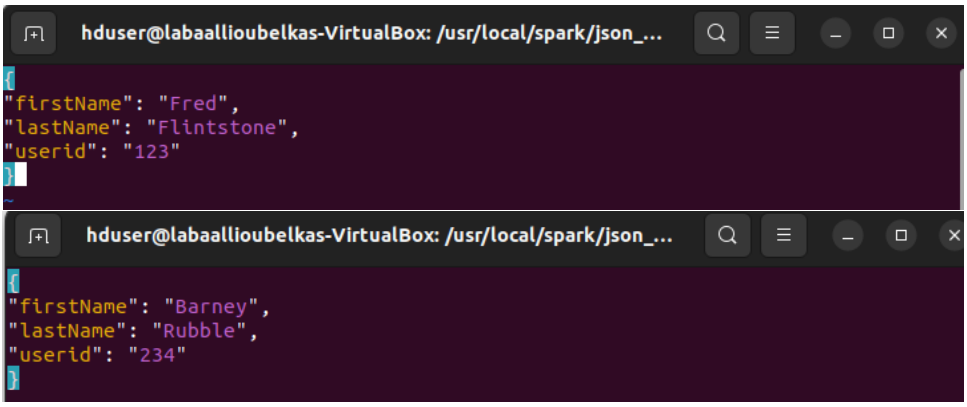

FIGURE 18 – Les fichiers **file1.json** et **file2.json**

Ensuite, nous tapons le code suivante :

1. Importation de la bibliothèque JSON :
   » **import json**
2. Création d'une RDD à partir de fichiers texte complets :
   » **myrdd1 = sc.wholeTextFiles("file :/usr/local/spark/json_files")**
3. Conversion des données JSON dans la RDD :
   » **myrdd2 = myrdd1.map(lambda a : json.loads(a[1]))**
4. Affichage des valeurs associées à la clé "firstName" pour les deux premiers enregistrements :
   » **for record in myrdd2.take(2) :**
   .. **print(record.get("firstName",None))**

```
>>> data = [10, 20, 30, 40, 50, 100, 250]
>>> distData = sc.parallelize(data)
>>> total = distData.reduce(lambda a,b: a + b)
>>> print(total)
500
>>> import json
>>> myrdd1 = sc.wholeTextFiles("file:/usr/local/spark/json_files")
>>> myrdd2 = myrdd1.map(lambda a: json.loads(a[1]))
>>> for record in myrdd2.take(2):
...     print(record.get("firstName",None))
...
Fred
Barney
>>>
```

FIGURE 19 – Utilisation de « wholeTextFiles »

3.7 Utilisation de « flatMap » et « distinct »

Pour utiliser flatMap et distinct, on suit les étapes suivantes :

1. Lecture du fichier texte dans une RDD :
   » **mydata = sc.textFile("file :/usr/local/spark/poeme.txt")**
2. Application de la transformation flatMap avec une fonction lambda :
   » **mynewdata = mydata.flatMap(lambda line : line.split(' ')).distinct()**
3. Affichage des résultats avec une boucle for :
   » **for line in mynewdata.collect() :**
   .. **print(line)**

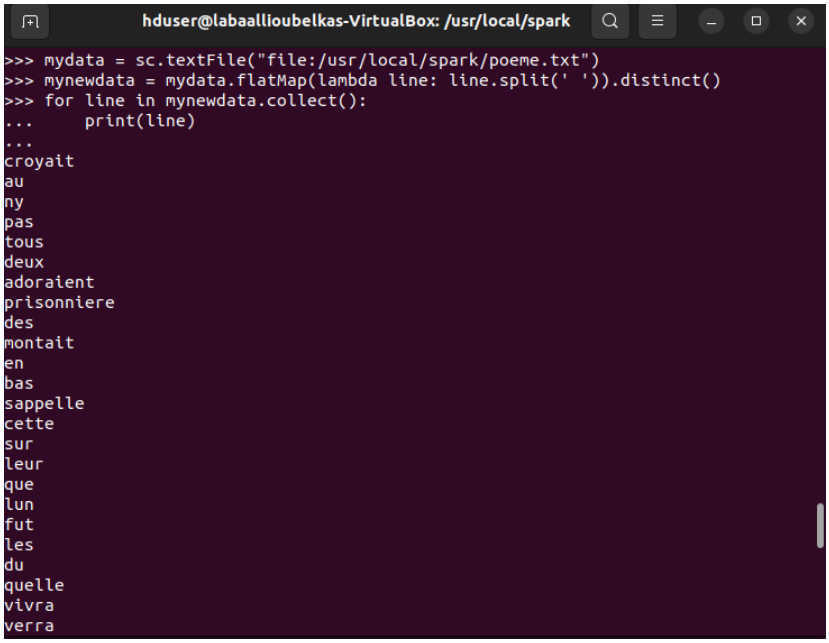

FIGURE 20 – Utilisation de « flatMap » et « distinct »

3.8 Utilisation de « subtract » et « zip »

En premier temps, nous allons créer deux RDD (rdd1 et rdd2) à partir de listes et appliquer ensuite la transformation subtract pour obtenir les éléments uniques de la première RDD (rdd1).

1. Création des RDDs à partir de listes :
   » **mydata = ["Chicago", "Boston", "Paris", "San Francisco", "Tokyo"]**
   » **rdd1 = sc.parallelize(mydata)**
   » **data = ["San Francisco", "Boston", "Amsterdam", "Mumbai", "McMurdo Station"]**
   » **rdd2 = sc.parallelize(data)**
2. Application de la transformation subtract :
   » **newrdd = rdd1.subtract(rdd2)**
3. Affichage des résultats avec une boucle for :
   » **for line in newrdd.collect() :**
   .. **print(line**

```
>>> mydata = ["Chicago", "Boston", "Paris", "San Francisco", "Tokyo"]
>>> rdd1 = sc.parallelize(mydata)
>>> data = ["San Francisco", "Boston", "Amsterdam", "Mumbai", "McMurdo Station"]
>>> rdd2 = sc.parallelize(data)
>>> newrdd = rdd1.subtract(rdd2)
>>> for line in newrdd.collect():
...     print(line)
...
Tokyo
Chicago
Paris
>>>
```

FIGURE 21 – Utilisation de « subtract »

Et en deuxième temps, nous allons utiliser la transformation « zip » :

1. Application de la transformation zip :
   » **ziprdd = rdd1.zip(rdd2)**
2. Affichage des résultats avec une boucle for :
   » **for line in ziprdd.collect() :**
   .. **print(line)**

```
>>> ziprdd = rdd1.zip(rdd2)
>>> for line in ziprdd.collect():
...     print(line)
...
('Chicago', 'San Francisco')
('Boston', 'Boston')
('Paris', 'Amsterdam')
('San Francisco', 'Mumbai')
('Tokyo', 'McMurdo Station')
>>>
```

FIGURE 22 – Utilisation de « zip »

3.9 Utilisation de intersection et union

Dans cette partie, nous allons effectuer des opérations d'intersection et d'union entre deux RDD (rdd1 et rdd2).

1. Opération d'intersection (intersection) :
   » **interdd = rdd1.intersection(rdd2)**
   » **for line in interdd.collect() :**
   .. **print(line)**

```
>>> interdd = rdd1.intersection(rdd2)
>>> for line in interdd.collect():
...     print(line)
...
San Francisco
Boston
>>>
```

FIGURE 23 – Utilisation de intersection

2. Opération d'union (union) :
   » **unionrdd = rdd1.union(rdd2)**
   » **for line in unionrdd.collect() :**
   .. **print(line)**

```
>>> unionrdd = rdd1.union(rdd2)
>>> for line in unionrdd.collect():
...     print(line)
...
Chicago
Boston
Paris
San Francisco
Tokyo
San Francisco
Boston
Amsterdam
Mumbai
McMurdo Station
>>>
```

FIGURE 24 – Utilisation de union

4 Connexion de Spark à une distribution de Hadoop

Spark peut utiliser les bibliothèques clientes Hadoop pour HDFS et YARN.

À partir de la version Spark 1.4, les packages de projet «Hadoop free» sont conçus pour nous permettre de connecter plus facilement un seul fichier binaire Spark à n'importe quelle version de Hadoop.

Pour utiliser ces packages , nous devons modifier **SPARK_DIST_CLASSPATH** afin d'inclure les fichiers jar relatifs à ces packages.

Pour ce faire, il est préférable d'ajouter une entrée dans **conf/spark-env.sh** :

```
hduser@labaallioubelkas-VirtualBox:/usr/local/spark/conf$ cp spark-env.sh.templa
te spark-env.sh
hduser@labaallioubelkas-VirtualBox:/usr/local/spark/conf$ vim spark-env.sh
```

FIGURE 25 – Connexion de Spark à une distribution de Hadoop

```
### in conf/spark-env.sh ###
# If 'hadoop' binary is on your PATH
export SPARK_DIST_CLASSPATH=/usr/local/hadoop
# With explicit path to 'hadoop' binary
export SPARK_DIST_CLASSPATH=/usr/local/hadoop/bin
# Passing a Hadoop configuration directory
export SPARK_DIST_CLASSPATH=/usr/local/hadoop/etc/hadoop
-- INSERT --                                    86,57         Bot
```

FIGURE 26 – Le contenu du fichier conf/spark-env.sh

5 Exécution du « Word Count » en utilisant le terminal scala et python

Une fois le nœud hadoop est bien configuré et spark bien connecté à hadoop. Nous pouvons déposer des données dans HDFS et les utiliser par spark.

5.1 Dépôt du poeme.txt dans le HDFS

Dans cette section, nous allons déposer le poeme.txt dans le HDFS :

```
hduser@labaallioubelkas-VirtualBox:/usr/local/spark/conf$ cd /usr/local/hadoop/
hduser@labaallioubelkas-VirtualBox:/usr/local/hadoop$ bin/hdfs dfs -put /home/hd
user/Documents/poeme.txt /
2023-10-24 19:23:09,077 WARN util.NativeCodeLoader: Unable to load native-hadoop
 library for your platform... using builtin-java classes where applicable
put: `/home/hduser/Documents/poeme.txt': No such file or directory
hduser@labaallioubelkas-VirtualBox:/usr/local/hadoop$ bin/hdfs dfs -put /home/hd
user/Documents/poeme.txt /
2023-10-24 19:30:53,514 WARN util.NativeCodeLoader: Unable to load native-hadoop
 library for your platform... using builtin-java classes where applicable
hduser@labaallioubelkas-VirtualBox:/usr/local/hadoop$ bin/hdfs dfs -ls /
2023-10-24 19:31:53,147 WARN util.NativeCodeLoader: Unable to load native-hadoop
 library for your platform... using builtin-java classes where applicable
Found 1 items
-rw-r--r--   1 hduser supergroup       1669 2023-10-24 19:31 /poeme.txt
hduser@labaallioubelkas-VirtualBox:/usr/local/hadoop$
```

FIGURE 27 – Dépôt du poeme.txt dans le HDFS

5.2 Exécution du code du « Word Count »

5.2.1 Exécution du code du « Word Count » en scala

Pour exécuter le traitement du « Word Count » en utilisant le terminal spark-shell, on tape le code scala suivant :

```
scala> val lines = sc.textFile("/poeme.txt")
lines: org.apache.spark.rdd.RDD[String] = /poeme.txt MapPartitionsRDD[1] at
File at <console>:23

scala> val words = lines.flatMap(_.split("\\s+"))
words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at flatMap at
sole>:23

scala> val wc = words.map(w => (w, 1)).reduceByKey(_ +_)
wc: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey
console>:23

scala> wc.saveAsTextFile("file1.count")

scala>
```

FIGURE 28 – Exécution du code du « Word Count » en scala

Ensuite, nous suivant les étapes suivantes enfin d'afficher le résultat :

1. Assurons-vous d'être dans le bon répertoire où Hadoop est installé :
   $ cd /usr/local/hadoop
2. Copions le fichier 'file1.count' depuis HDFS vers le système de fichiers local :
   $ hadoop fs -get file1.count
3. Listons les fichiers dans le répertoire courant de HDFS :
   $ bin/hdfs dfs -ls .
4. Affichons le contenu du fichier 'part-00000' de 'file1.count' dans HDFS :
   $ bin/hdfs dfs -cat file1.count/part-00000

Et voilà, le résultat du programme Word Count :

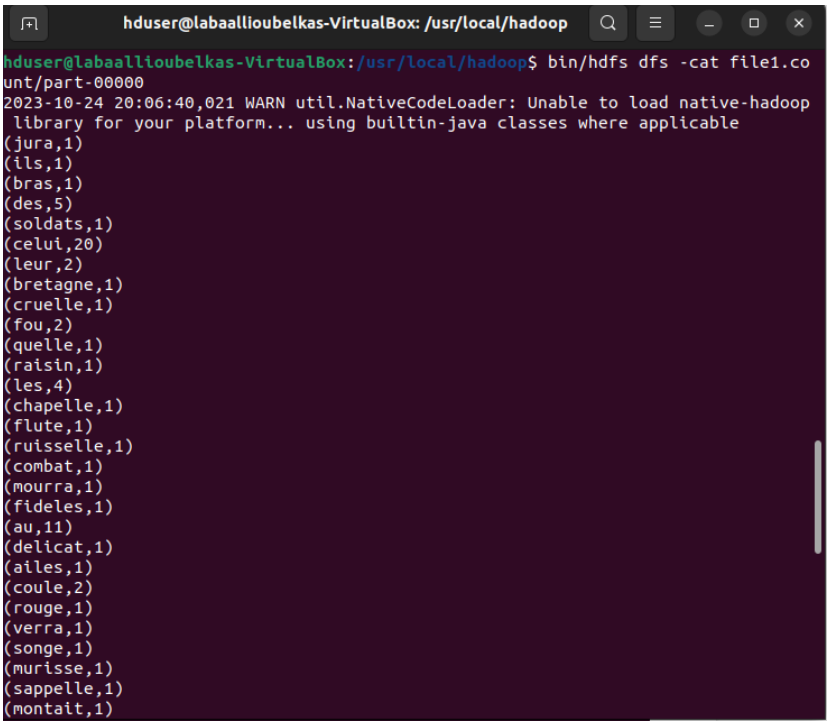

FIGURE 29 – Le résultat du programme Word Count

NB : Pour pouvoir écrire dans le hdfs, il faut donner tous les droits au répertoire : « /usr/local/hadoop_store/hdfs/datanode » pour cela on tape les commandes suivantes :
 # cd /usr/local/hadoop
 # chmod 777 /usr/local/hadoop_store/hdfs/datanode

5.2.2 Exécution du code du « Word Count » en pyspark

Pour exécuter le traitement du « Word Count » en utilisant le terminal Pyspark, on tape le code suivant pour lancer le terminal pyspark :
$ cd /usr/local/spark
$ ./bin/pyspark
Puis, on exécute le code suivant :

FIGURE 30 – Exécution du code du « Word Count » en pyspark

En fin, nous pouvons afficher le resultat en utilisant hdfs :

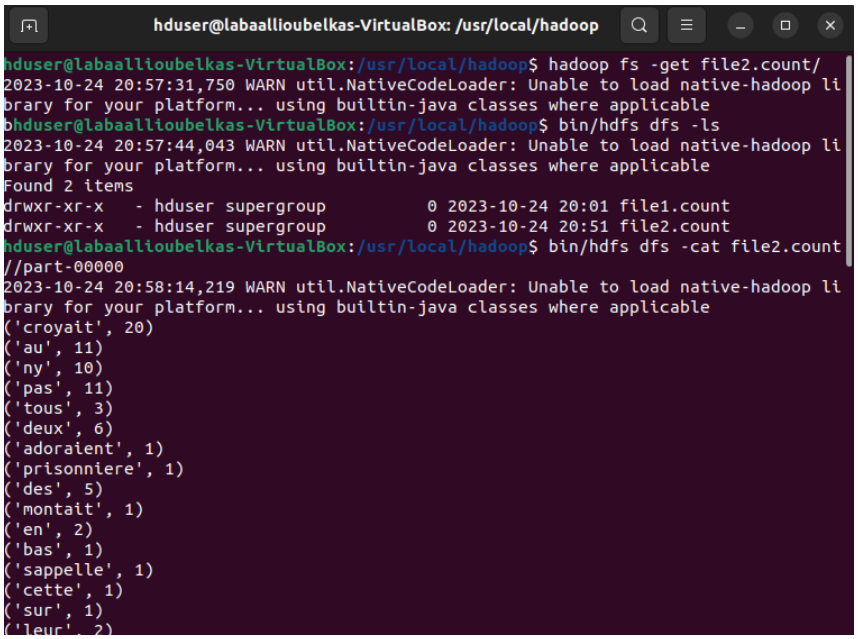

FIGURE 31 – Le résultat du programme Word Count en pyspark

6 Exécution du « Word Count » en utilisant un script python

Créons d'abord un fichier « word_count.py » et insérons le code python suivant :

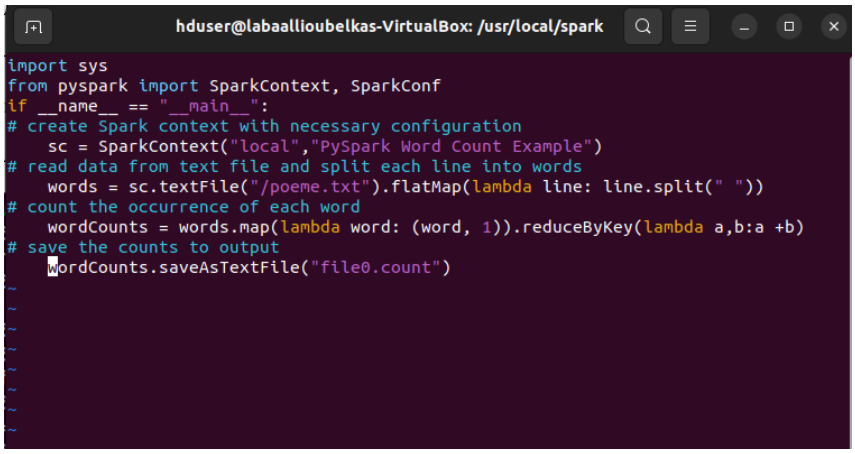

FIGURE 32 – Le fichier « word_count.py »

Tapons ensuite la commande suivante pour exécuter le script python créé :
$ ./bin/spark-submit word_count.py

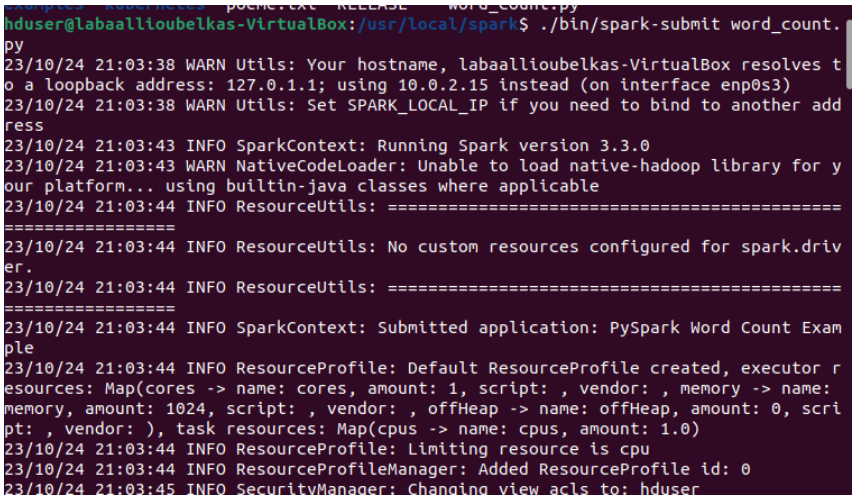

FIGURE 33 – L'exécution du fichier « word_count.py »

Visualisons donc le résultat :

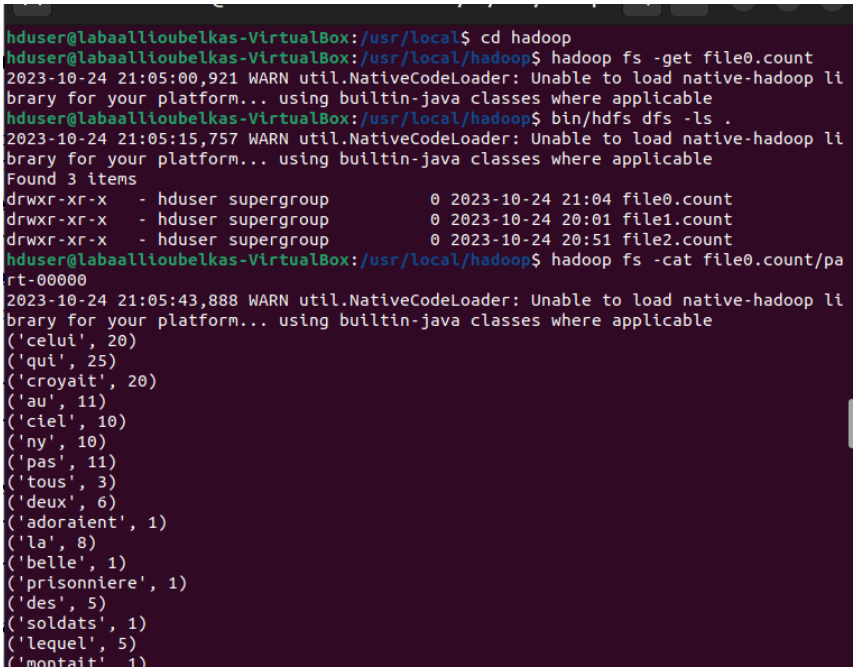

FIGURE 34 – Visualisation des résultat du fichier « word_count.py »

7 Exécution d'une application Spark Batch en Java

Dans cette section, nous allons créer une application Spark Batch en Java (un simple WordCount), le charger sur le nœud en local et le lancer.

7.1 Installation d'Apache Maven

Téléchargons d'abord « apache-maven-3.5.0-bin.tar.gz » à partir du lien suivant : Lien
Déposons ce fichier dans notre répertoire /home/hduser/Documents/

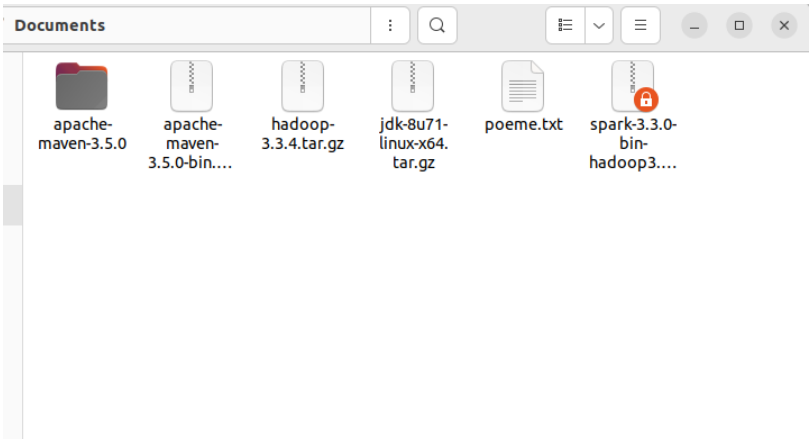

FIGURE 35 – Le fichier « apache-maven-3.5.0-bin.tar.gz »

Ensuite nous décompressons ce fichier en utilisant les commandes suivantes :

$ cd /home/hduser/Documents/

$ tar -zxvf apache-maven-3.5.0- bin.tar.gz

Et puis nous le plaçons dans /opt/ :

$ sudo -i

 # mv /home/hduser/Documents/apache-maven- 3.5.0 /opt/

Pour mettre en place de manière permanente la variable d'environnement PATH pour tous les utilisateurs :

Ouvrons le fichier /etc/profile et modifions le PATH en ajoutant le chemin où se trouve le bin de maven dans export PATH :

```
export JAVA_HOME=/opt/java/jdk1.8.0_71/
export JRE_HOME=/opt/java/jdk1.8.0_71/jre
export PATH=$PATH:/opt/java/jdk1.8.0_71/bin:/opt/java/jdk1.8.0_71/jre/bin
export PATH=$PATH:/opt/java/jdk1.8.0_71/bin:/opt/java/jdk1.8.0_71/jre/bin:/opt/apache-maven-3.5.0/bin
~
~
```

FIGURE 36 – Le fichier /etc/profile

Après avoir enregistré le fichier profile, exécutons la commande source pour recharger le fichier dans la session hduser :

$ source /etc/profile

Testons la configuration de maven en tapant :**$ mvn -v**

```
root@labaallioubelkas-VirtualBox:~# sudo vim /etc/profile
root@labaallioubelkas-VirtualBox:~# source /etc/pro
profile    profile.d/ protocols
root@labaallioubelkas-VirtualBox:~# source /etc/profile
root@labaallioubelkas-VirtualBox:~# mvn -v
Apache Maven 3.5.0 (ff8f5e7444045639af65f6095c62210b5713f426; 2017-04-03T20:39:06+01:00)
Maven home: /opt/apache-maven-3.5.0
Java version: 1.8.0_71, vendor: Oracle Corporation
Java home: /opt/java/jdk1.8.0_71/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "6.2.0-35-generic", arch: "amd64", family: "unix"
root@labaallioubelkas-VirtualBox:~#
```

FIGURE 37 – Teste de la configuration de maven

Créons après un projet Maven avec la commande suivante :

$ mvn archetype :generate - DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.1

FIGURE 38 – Les paramètres pour la création du projet maven

À la fin de la génération du projet maven, un message de « Build success » s'affiche :

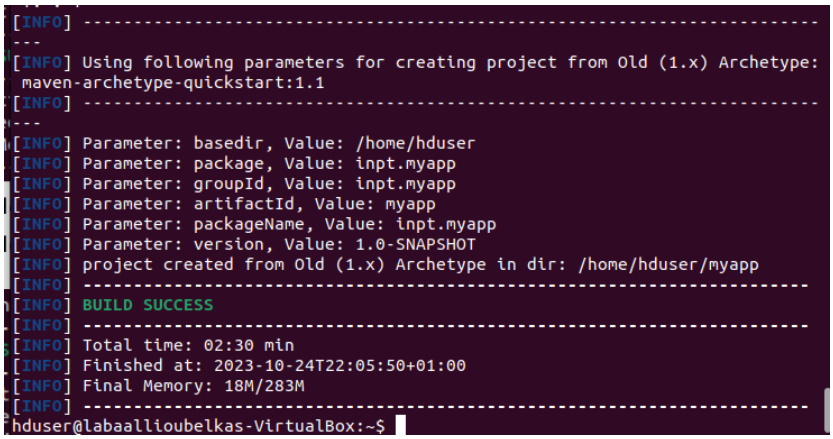

FIGURE 39 – Message de « Build success »

Puis nous exécutons les commande suivantes :

$ cd myapp/
$ mvn package

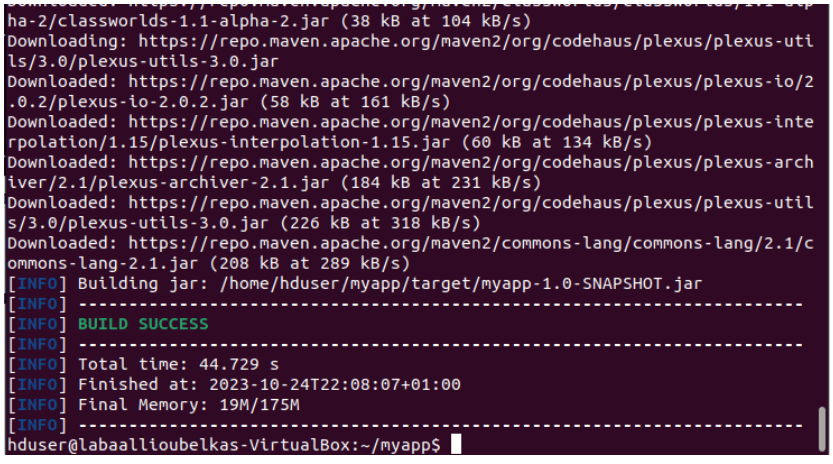

FIGURE 40 – mvn package

Pour voir toute l'arborescence du projet, nous pouvons installer la commande « tree » et ainsi visualiser toute l'arborescence du projet :

$ cd
$ sudo apt install tree
$ tree myapp/

FIGURE 41 – L'arborescence du projet

Le projet maven créé correspond à un projet java « Hello World » sur lequel nous allons nous baser pour créer notre application java « WordCount ».

7.2 Reconfiguration du projet Maven

Rajoutons dans le fichier « pom.xml » les dépendances nécessaires, pour cela :
Nous allons ajouter entre <properties>et </properties>, le contenu suivant :

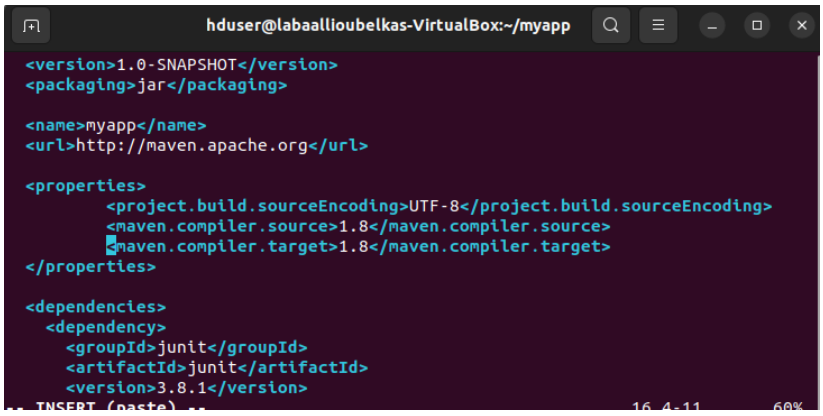

FIGURE 42 – Le contenu entre <properties>et </properties>

Et nous allons ajouter entre <dependencies> et </dependencies>, le contenu suivant :

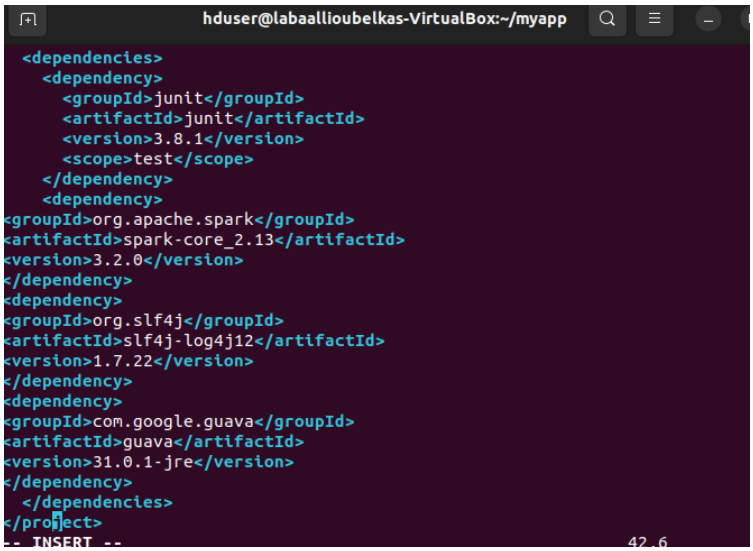

FIGURE 43 – Le contenu entre <dependencies> et </dependencies>

Dans le répertoire /home/hduser/myapp/src/main/java/DataEngineer/myapp nous allons :

- Renommer App.java en WordCountTask.java par :

$ cd /home/hduser/myapp/src/main/java/inpt/myapp

$ mv App.java WordCountTask.java

- Modifier le fichier WordCountTask.java en supprimant l'ancien contenu et copier le contenu du traitement « WordCountTask » ci-dessous dans « WordCountTask.java » :

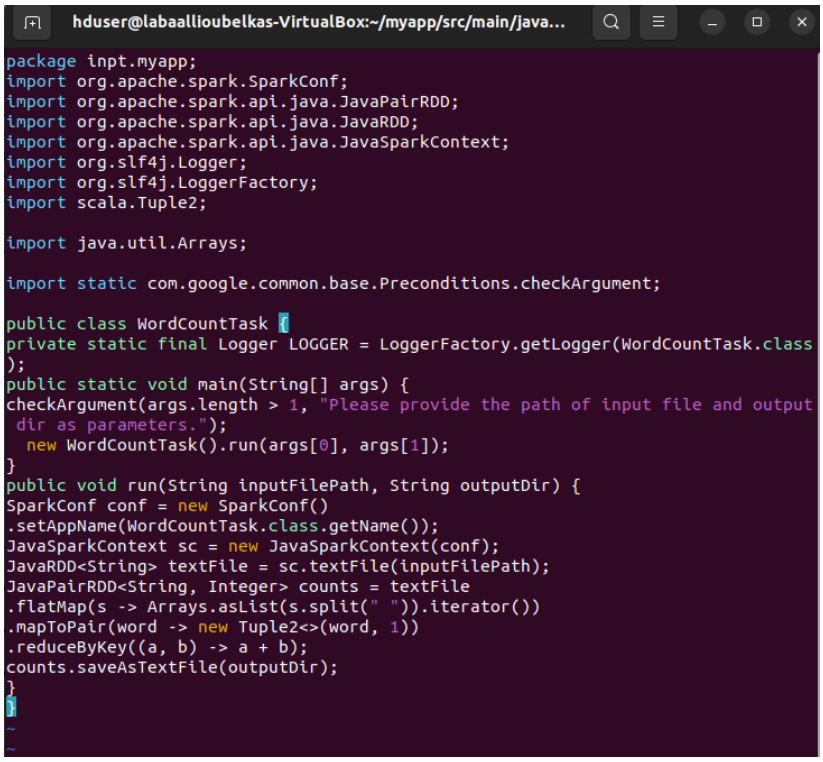

FIGURE 44 – Le contenu du fichier WordCountTask.java

- Enregistrer le fichier WordCountTask.java, et lancer la commande suivante pour

compiler le code :
$ cd myapp/
$ mvn package

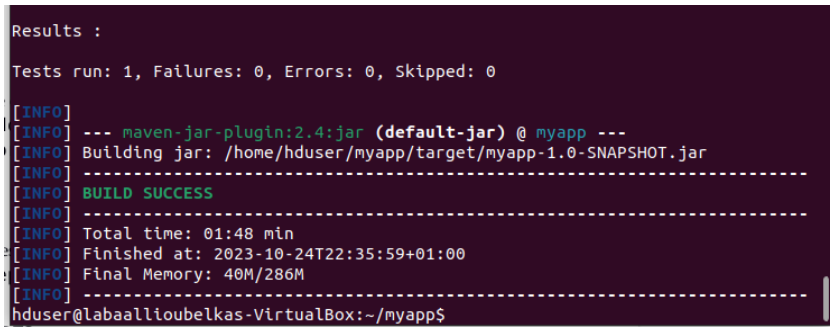

FIGURE 45 – Compilation du code

7.3 Nettoyage et Formatage du nœud hadoop

Nous lançons les commandes shell sont utilisées pour effectuer certaines opérations de maintenance sur le stockage associé à Hadoop, principalement sur les répertoires de stockage des nœuds du système de fichiers distribué HDFS (Hadoop Distributed File System).

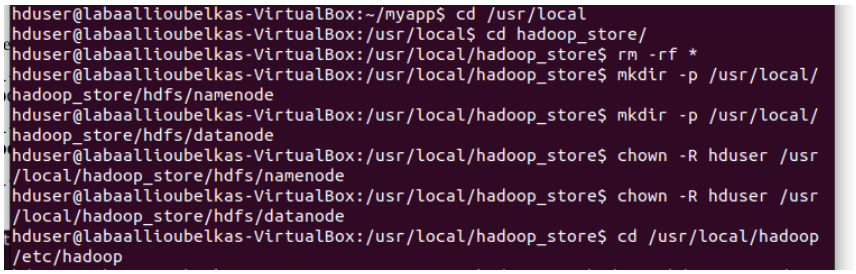

FIGURE 46 – Nettoyage du nœud hadoop

Ensuite, nous relançons le formatage du nœud :

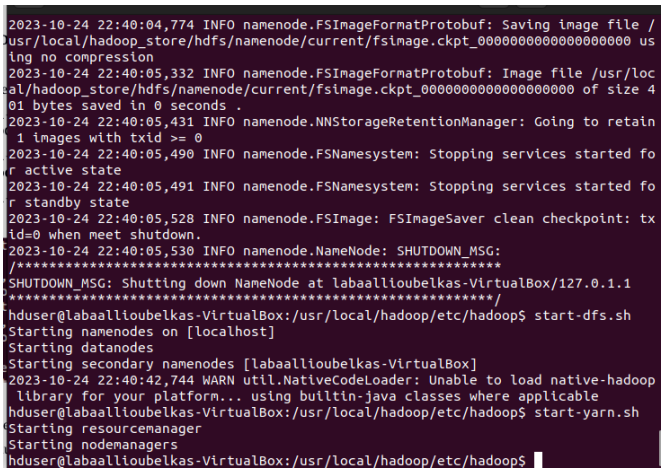

FIGURE 47 – Formatage du nœud hadoop

Finalement, nous testons si le nœud est disponible (live) :

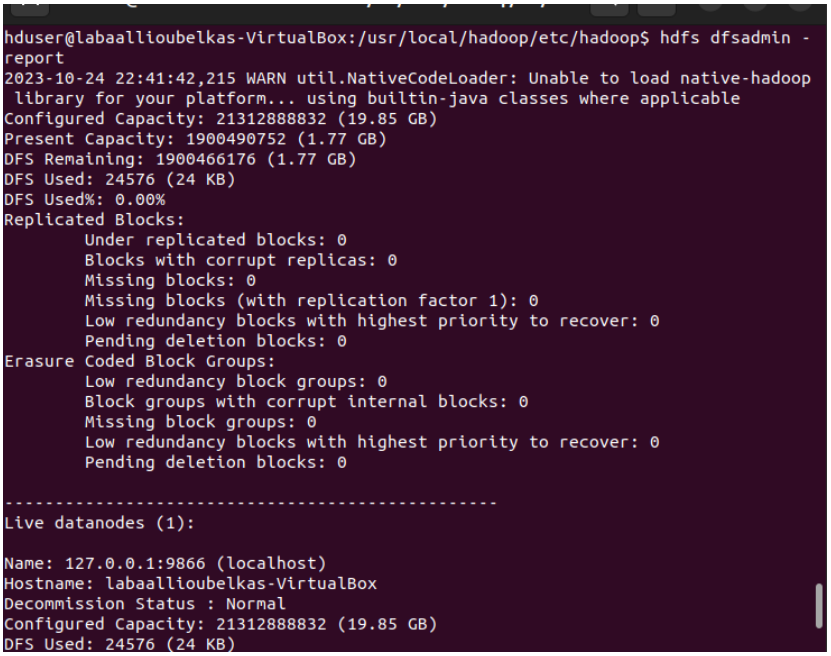

FIGURE 48 – Test si le nœud est disponible

7.4 Dépôt du poeme.txt dans HDFS

Nous allons déposer le poeme.txt dans le HDFS comme précedemment :

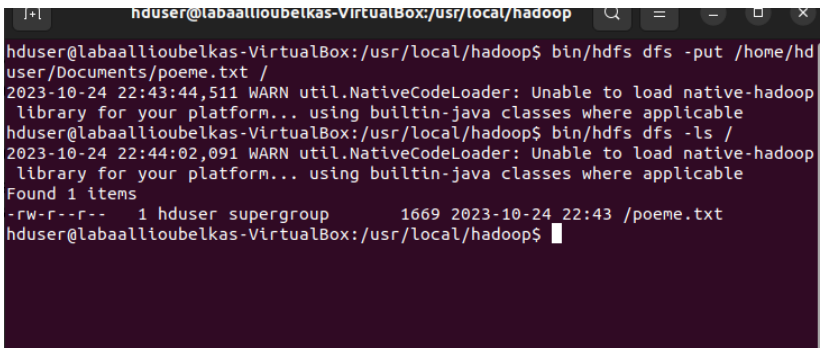

FIGURE 49 – Dépôt du poeme.txt dans HDFS

Maintenant nous pouvons lancer la commande spark-submit pour exécuter le programme Word Count :

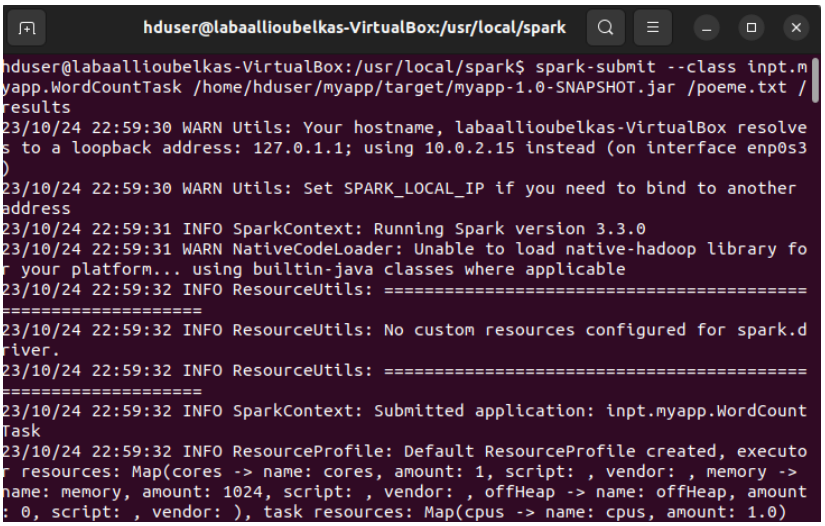

FIGURE 50 – spark-submit

Pour afficher le résultat, nous tapons les commandes :

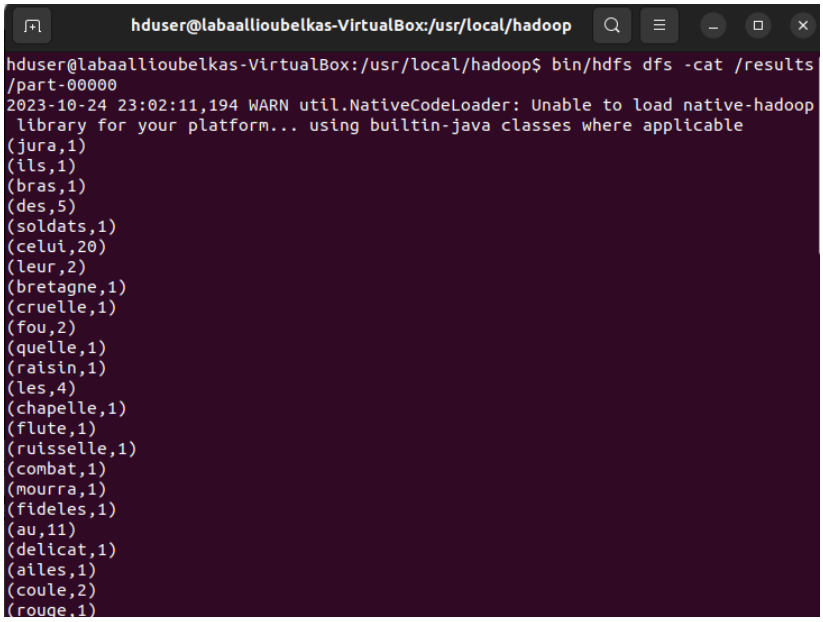

FIGURE 51 – Affichage des résultats

Conclusion

Au cours de ce TP, notre exploration d'Apache Spark en association avec Hadoop a été enrichissante. Nous avons acquis des compétences fondamentales en installant et en utilisant ces frameworks de traitement distribué. La mise en œuvre d'un programme de Word Count a été un excellent point de départ, nous permettant de comprendre les principes de base de la programmation distribuée avec Spark. Ce processus nous a permis de manipuler efficacement de grands ensembles de données tout en exploitant la puissance de traitement parallèle offerte par ces technologies. Ces connaissances constituent une base solide pour explorer davantage les fonctionnalités avancées d'Apache Spark et Hadoop dans des projets plus complexes et stimulants.