



Ministry of Higher Education and Research
Higher School of Computer Science 08 May 1945 - Sidi Bel Abbas
Second Year Second Cycle - Artificial Intelligence and Data Science

LAB 02 : SPARK RDD

Presented By : FELLAH Abdelnour.

Date: March 1, 2024

1 INTRODUCTION

In this lab, we explored the configuration of a Spark cluster and the execution of RDD (Resilient Distributed Dataset) Python programs using both the local file system and the HDFS (Hadoop Distributed File System). Spark is a powerful distributed computing framework that provides high-level APIs in Python, Java, and Scala for parallel data processing. Understanding how to configure a Spark cluster and interact with different file systems is crucial for efficiently processing large-scale datasets.

2 PART 01 : SETUP

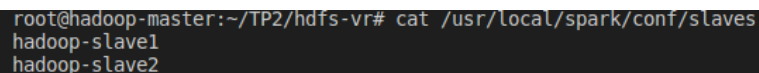
1- starting the three containers

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

2- create the file slaves in the directory /usr/local/spark/conf

```
root@hadoop-master:~#touch /usr/local/spark/conf/slaves
```

3- add the name of the workers to the file slaves



```
root@hadoop-master:~/TP2/hdfs-vr# cat /usr/local/spark/conf/slaves
hadoop-slave1
hadoop-slave2
```

Figure 1: The content of the file slaves

4 - Fix the python path :

```
root@hadoop-master:~#which python3
>> /usr/bin/python3
root@hadoop-master:~#cd /usr/local/spark/sbin
root@hadoop-master:~#cp spark-env.sh.template spark-env.sh
```

and add the following line to the file spark-env.sh : PYSPARK_PYTHON=/usr/bin/python3

5- start the spark services on all nodes

```
root@hadoop-master:~#cd /usr/local/spark/sbin
root@hadoop-master:~#start-all.sh
```

3 PART 02 : SPARK RDD

First we need to move the arbres.csv file to the containers' local file system :

```
# docker cp arbres.csv hadoop-master:/root/arbres.csv
# docker cp arbres.csv hadoop-slave1:/root/arbres.csv
# docker cp arbres.csv hadoop-slave2:/root/arbres.csv
```

1- a program to count the number of rows in the file

```
from pyspark import SparkConf, SparkContext

nomappli = "pl"
config = SparkConf().setAppName(nomappli)
sc = SparkContext(conf=config)

data = sc.textFile("file:///root/arbres.csv")

print("\n\n --- size = ",data.count()," --- \n\n")
```

Figure 2: The program of the first question

2- a program to calculate the mean of the height of trees

```
from pyspark import SparkConf, SparkContext

nomappli = "p2"
config = SparkConf().setAppName(nomappli)
sc = SparkContext(conf=config)

# load the file
file = sc.textFile("file:///root/arbres.csv")

# get the header
header = file.first()

# remove the header
file_without_header = file.filter(lambda row: row != header)

# get the columns
columns = header.split(";")

# get the index of the desired column
idx = columns.index("HAUTEUR")

# 1- split the rows
# 2- project the desired column
# 3- remove elements that can not be converted to float
# 4- convert the remaining elements to float
hauteur = file_without_header \
    .map(lambda row : row.split(";")) \
    .map(lambda row : row[idx]) \
    .filter(lambda row : row.replace(".", "").isnumeric()) \
    .map(lambda row : float(row))

# compute the mean
mean = hauteur.mean()

print("\n\n----- mean,size = ", (mean,hauteur.count()), " ----- \n\n")
```

Figure 3: The program of the second question

3- a program to get the highest tree

```
from pyspark import SparkConf, SparkContext

nomappli = "p3"
config = SparkConf().setAppName(nomappli)
sc = SparkContext(conf=config)

### load the file
file = sc.textFile("file:///root/arbres.csv")

### get the header
header = file.first()

### remove the header
file_without_header = file.filter(lambda row: row != header)

### get the columns
columns = header.split(";")

### get the index of the desired columns
genre = columns.index("GENRE")
hauteur = columns.index("HAUTEUR")

### construct key-value pairs
# 1- split the row
# 2- construct key-value pairs
# 3- remove the heights that are not convertible to float
# 4- convert the keys to float
pairs = file_without_header \
    .map(lambda row : row.split(";")) \
    .map(lambda row: (row[hauteur], row[genre])) \
    .filter(lambda row : row[0].replace(".", "").isnumeric()) \
    .map(lambda row : (float(row[0]), row[1]))

### sort the pairs by key
sorted_pairs = pairs.sortByKey(ascending=False)

### get the first pair
first_pair = sorted_pairs.first()

### print out the result
print("\n\n --- pair = ", first_pair, " --- \n\n")
```

Figure 4: The program of the third question

4- a program to get the mean of number of trees by genre

```
from pyspark import SparkConf, SparkContext
from operator import add

nomappli = "p4"
config = SparkConf().setAppName(nomappli)
sc = SparkContext(conf=config)

### load the file
file = sc.textFile("file:///root/arbres.csv")

### get the header
header = file.first()

### remove the header
file_without_header = file.filter(lambda row: row != header)

### get the columns
columns = header.split(";")

### get the index of the desired columns
genre = columns.index("GENRE")

### construct key value pairs (map)
# 1- split the rows
# 2- project the desired column
pairs = file_without_header \
    .map(lambda row : row.split(";")) \
    .map(lambda row : (row[genre], 1))

### reduce
reduced_pairs = pairs.reduceByKey(add)

### print the results
print("\n\n"+str(reduced_pairs.collect()+"\n\n"))
```

Figure 5: The program of the fourth question

move the python files to the file system of the master node :

```
root@hadoop-master:~#mkdir local-vr
# docker cp p1.py hadoop-master:/root/local-vr/p1.py
# docker cp p2.py hadoop-master:/root/local-vr/p2.py
# docker cp p2.py hadoop-master:/root/local-vr/p3.py
# docker cp p2.py hadoop-master:/root/local-vr/p4.py
```

run the python programs :

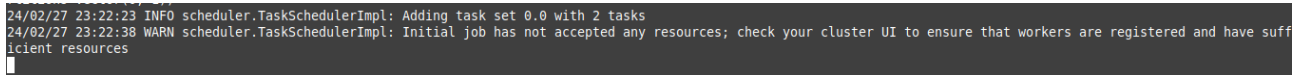
```
root@hadoop-master:~#cd local-vr
```

run the programs with spark-submit :

```
root@hadoop-master:~#spark-submit --master spark://hadoop-master:7077 p1.py
root@hadoop-master:~#spark-submit --master spark://hadoop-master:7077 p2.py
root@hadoop-master:~#spark-submit --master spark://hadoop-master:7077 p3.py
root@hadoop-master:~#spark-submit --master spark://hadoop-master:7077 p4.py
```

NOTE :

if you got the following warning when trying to run the programs :



```
24/02/27 23:22:23 INFO scheduler.TaskSchedulerImpl: Adding task set 0.0 with 2 tasks
24/02/27 23:22:38 WARN scheduler.TaskSchedulerImpl: Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient resources
```

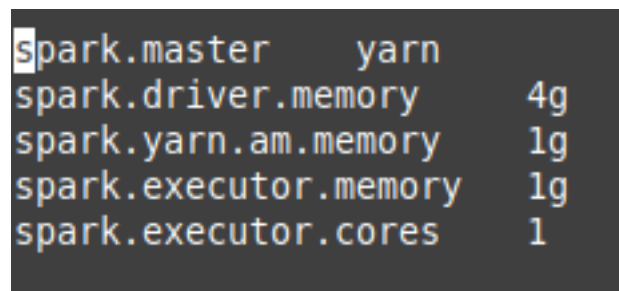
Figure 6: Resources problem

then it's probably a problem of resources allocation.

move the config directory, and try modifying the spark-defaults.conf file

```
root@hadoop-master:~#cd /usr/local/spark/conf
```

in my case reducing the spark.executor.memory to 1g did the trick (its value is 2g by default).



```
spark.master      yarn
spark.driver.memory 4g
spark.yarn.am.memory 1g
spark.executor.memory 1g
spark.executor.cores 1
```

Figure 7: Resources allocation problem

the result of the programs :

```
24/02/24 17:20:39 INFO scheduler.TaskSchedulerImpl: Removed
24/02/24 17:20:39 INFO scheduler.DAGScheduler: ResultStage
24/02/24 17:20:39 INFO scheduler.DAGScheduler: Job 0 finish

--- size = 98 ---

24/02/24 17:20:39 INFO spark.SparkContext: Invoking stop()
24/02/24 17:20:39 INFO server.AbstractConnector: Stopped Sp
24/02/24 17:20:39 INFO ui.SparkUI: Stopped Spark web UI at
```

Figure 8: The output of the first program

```
24/02/24 17:21:13 INFO scheduler.TaskSchedulerImpl
24/02/24 17:21:13 INFO scheduler.DAGScheduler: Re
24/02/24 17:21:13 INFO scheduler.DAGScheduler: Jo

----- mean,size = (22.3125, 96) -----

24/02/24 17:21:13 INFO spark.SparkContext: Invoki
24/02/24 17:21:13 INFO server.AbstractConnector:
24/02/24 17:21:13 INFO ui.SparkUI: Stopped Spark
```

Figure 9: The output of the second program

```
24/02/24 17:21:44 INFO scheduler.DAGScheduler:
24/02/24 17:21:44 INFO scheduler.DAGScheduler:

--- pair = (45.0, 'Platanus') ---

24/02/24 17:21:44 INFO spark.SparkContext: Invo
24/02/24 17:21:44 INFO server.AbstractConnector
24/02/24 17:21:44 INFO ui.SparkUI: Stopped Spar
```

Figure 10: The output of the third program


```
[('Sequoiadendron', 5), ('Robinia', 1), ('Juglans', 1), ('Ulmus', 1), ('Quercus', 4), ('Gymnocladus', 1), ('Ginkgo', 5), ('Zelkova', 4), ('Taxus', 2), ('Davidia', 1), ('Fraxinus', 1), ('Pinus', 5), ('Ailanthus', 1), ('Taxodium', 3), ('Sequoia', 1), ('Diospyros', 4), ('Aesculus', 3), ('Styphnolobium', 1), ('Liriodendron', 2), ('Araucaria', 1), ('Tilia', 1), ('Platanus', 19), ('Catalpa', 1), ('Eucommia', 1), ('Paulownia', 1), ('Acer', 3), ('Alnus', 1), ('Cedrus', 4), ('Calocedrus', 1), ('Magnolia', 1), ('Maclura', 1), ('Pterocarya', 3), ('Broussonetia', 1), ('Corylus', 3), ('Celtis', 1), ('Fagus', 8)]
```

Figure 11: The result of the fourth program

4 PART 03 : WORKING WITH HDFS

if you have files stored in hdfs file system you'll need to change the path in the previously showed programs.

recall that to put the file arbres.csv in the hdfs file system you need to use the following command :

```
root@hadoop-master:~#hadoop fs -put arbres.csv
```

and unlike in the previous method where we used the local file system,the file needs only to exist in the master's file system.

then you can update the python programs to :

```
### load the file
file = sc.textFile("hdfs:///user/root/arbres.csv")
```

5 CONCLUSION

In conclusion, this lab provided valuable insights into configuring a Spark cluster and running RDD Python programs using different file systems. We successfully configured a Spark cluster with a master and multiple worker nodes, demonstrating how to set up a distributed computing environment. We also learned how to interact with the local file system and the HDFS, understanding the advantages of using HDFS for distributed data storage and processing.