Report Cloud And BigData

Rémi DE BAUDRY D'ASSON
Arthur BRUGIERE

December 2018

1 Introduction

As presented in the subject, the goal of this project is to use a Spark infrastructure inside Docker containers to simulate a cluster of virtual machines.

In our case, we will execute a wordcount application on a big text file, according to several scenarios, then we will measure and compare the performances of different architectures to figure where the improvement or non improvement comes from.

2 Scenarios

In this section, we will describe the different scenarios we will implement to test our application, from the most simple to the most complicated.

For each scenario, we will test it's performance on a big text file (4.28Go) with a 4 cores computer, then compare the execution time. With the project we packaged a small script called **incrementFileSize.sh** at the root of the project. This file create a 4.28Go file from the small 2Ko starter file, so you can choose to test with either a small or a big file.

2.1 Scenario 1 : One Spark container running locally

The first version of the scenario consist in using just one Docker container with Spark, which is the equivalent of running Spark locally on our computer.

To launch this scenario, simply go into **V2** folder and execute the script **launch.sh**. The script will do the following:

1. Run Docker-compose:

   ```
   sudo docker-compose up -d
   ```

2. Launch work in master container:

```
sudo docker exec -ti v1_spark_1 sh -c
"/usr/spark-2.3.1/bin/spark-submit --class
WordCount --master local /tmp/data/wc.jar
/tmp/data/sample1.txt"
```

3. And finally display the result:

```
cat ./data/result/*
```

In this configuration, with our 4.28Go file we get an execution time of about **4 minutes and 29 seconds** .

2.2 Scenario 2 : Several Spark containers running with one master and several workers

The second version of the scenario uses this time several Spark containers:

- One will be the master
- The others will be the workers

To launch this scenario, simply go into **V2** folder and execute the script **launch.sh**. You will be asked how many workers you want to start and how many core you want to affect to each worker, then the script does the following:

1. Run Docker-compose and specifying the number of workers:

```
sudo docker-compose up -d --scale worker=$nbrWorkers
```

2. Launch work in master container:

```
sudo docker exec -ti v2_master_1 sh -c
"/usr/spark-2.3.1/bin/spark-submit --class WordCount --master
spark://master:7077 /tmp/data/wc.jar /tmp/data/sample1.txt"
```

3. And finally display the result:

```
cat ./data/result/*
```

We get the following results with different number of workers:

FILE SIZE	NUM WORKER	NUM CORE/WORKER	TIME
4.28Go	1	1	**4m 26s**
4.28Go	1	2	**2m 20s**
4.28Go	1	3	**2m 20s**
4.28Go	2	1	**2m 30s**
4.28Go	2	2	**2m 21s**
4.28Go	3	1	**2m 28s**

2.3 Scenario 3 : Same as previous + several HDFS containers

In this third and last version of the scenario, we will use the same architecture as before but this time we will also add several HDFS docker containers to manage the HDFS system. We have here an HDFS architecture with 3 namenodes where everything is distributed on several docker containers.

To launch this scenario, simply go into **V3** folder and execute the script **launch.sh**. You will be asked how many workers you want to start, then the script does the following:

1. Run Docker-compose and specifying the number of workers:

   ```
   sudo docker-compose up -d --scale worker=$nbrWorkers
   ```

2. (Optional) Format the HDFS:

   ```
   sudo docker exec -ti namenode sh -c
   "/opt/hadoop-2.7.1/bin/hdfs namenode -format"
   ```

3. (Optional) Add file into HDFS system:

   ```
   sudo docker exec -ti namenode sh -c
   "/opt/hadoop-2.7.1/bin/hdfs dfs -mkdir /input"

   sudo docker exec -ti namenode sh -c
   "/opt/hadoop-2.7.1/bin/hdfs dfs -put /tmp/data/sample1.txt /input"
   ```

4. Launch work in master container:

   ```
   sudo docker exec -ti v3_master_1 sh -c
   "/usr/spark-2.3.1/bin/spark-submit --class
   WordCount --master spark://master:7077
   /tmp/data/wc.jar hdfs://namenode:8020/input/sample1.txt"
   ```

5. And finally display the result:

   ```
   cat ./data/result/*
   ```

We get the following results with different number of workers:

FILE SIZE	NUM WORKER	NUM CORE/WORKER	TIME
4.28Go	1	1	**4m 16s**
4.28Go	2	1	**2m 36s**

Unfortunately, we could only do those two test since the computer used couldn't handle more.

3 Conclusion

What we can conclude from all those test is that, without much surprise, increasing the number of workers and the number of cores reduce the execution time.

However, by looking at the results, we understand better how much each of this parameters influence the result. For example, in the V2, using 2 workers with 1 core is equivalent of using 1 worker with 2 cores but uses less cores.

Furthermore, adding more workers most likely reduces even more the computation time, even though the computer used here saturated and couldn't show great result with 3 workers. Also, for the same reason, the use of HDFS didn't seem to make much improvements even though it was supposed to.