Lab Hadoop/Scala/Spark

In this lab, we will discover the basis of Hadoop, Spark, and Scala. To do so, we will emulate a Hadoop cluster using Docker.

1 Setup

To run this lab, you will need to have Docker, Docker-compose, Git and SBT installed. You can find the instructions for each of them on the official websites:

- Docker: https://docs.docker.com/get-docker/
- Docker-compose: https://docs.docker.com/compose/install/
- SBT: https://www.scala-sbt.org/1.x/docs/Setup.html

Then, we will clone a GitHub repository that contains everything we need to simulate our cluster:

```
git clone https://github.com/Marcel-Jan/docker-hadoop-spark
```

Go in the cloned directory and have a look at the *docker-compose.yml*.

1.1. What are the ports accessible from outside for the containers related to Hadoop and Spark? What are they used for (you might need to read the documentation matching the version of Hadoop/Spark we are using. Some default parameters are accessible here: https://hadoop.apache.org/docs/r3.2.1/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml and https://hadoop.apache.org/docs/r3.2.1/hadoop-project-dist/hadoop-common/core-default.xml. The environment variables in the docker-compose can also help)?

1.2. How many datanodes and Spark workers do you see? Modify the *docker-compose.yml* to add another Spark worker. Set its port to *"8082:8081"*.

1.3. Add another datanode. To do so, rename the first one *datanode-1*. Modify its container name, the volume (do not forget to modify it at the bottom also), and its location. Then duplicate *datanode-1* and also change its container-name, volume (create a new one at the bottom), and file location. You also need to modify the visible port (or you will have a collision). Set it to *"9865:9864"*. Look for **ALL** the locations where *datanode* is mentioned in the file. You must modify it to *datanode-1* and add an equivalent command for *datanode-2*.

1.4. We want to be able to see the jobs running on the Hadoop cluster. To do so, add:

```
ports:
  - 8188:8188
```

to the definition of the container *historyserver*.

Similarly, we want to be able to access the Web UI of the resource manager. Add in the container *resourcemanager*:

```
ports:
  - 8088:8088
```

Same for Spark, add "- 4040:4040" to the ports list to make the runtime WebUI accessible.

1.5. Now run docker-compose to start all your containers (you might need to be *sudo* on your machine):

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

Check that all the containers are running correctly by executing:

```
docker ps -a
```

1.6. Using what you learned before, access the Web UI of Hadoop and Spark. Check that you have the correct number of datanodes and spark workers. Then, through the Web interface of Hadoop, access the files in the HDFS. What do you see?

2 Hadoop

We can connect to a container using the command:

```
docker exec -it name_of_the_container /bin/bash
```

For example, if we want to connect to "namenode" (the name is the same than in the *docker-composer.yml*):

```
docker exec -it namenode /bin/bash
```

This command is a bit equivalent to doing *ssh* on a real server.

2.1. Connect to the namenode. You will arrive at the root of a Linux system where you can use all traditional commands. Check the number of datanodes by using the command *hdfs dfsadmin -report*.

Commands specific to Hadoop generally start with the keyword *hadoop* or *hdfs*. When we want to interact with the HDFS, the commands start by *hadoop fs* - followed by commands that look like UNIX ones. For example:

```
hadoop fs -cat file_on_the_hdfs
```

Note that these commands only deal with the files on the HDFS, not your local file system (here, not the file system you can explore when you connect to the container).

List the files at the root of the HDFS. Do you see the same ones as you observed before?

2.2. Now, we want to transfer a file on the HDFS. First, create a new directory */data* on the HDFS. Next, download the raw version of Peter Pan: https://www.gutenberg.org/files/16/16-0.txt and call it *peter_pan.txt*. We need to send this file to the docker container. That can be done with the command:

```
docker cp peter_pan.txt namenode:/
```

(This is kind of equivalent to an *scp*). We can now put this file on the HDFS. To do so, use the command *hadoop fs -put*. This is one of the few commands that allows the interaction between the local file system and the HDFS. Put *peter_pan.txt* in */data* of the HDFS. Check that you see the file. The inverse operation of *put* is *get*. Download *peter_pan.txt* from the HDFS and call it *peter_pan2.txt*.

2.3. You can get information about a file by using *hdfs fsck*. How many replicas were required for *peter_pan.txt*? How many are missing?

3 Map Reduce (optional)

In this section, we will create a Map-Reduce Java program that runs on Hadoop. It is important to use **Java 8** during this stage. Install it if necessary. Start by creating a new Maven project with your IDE (if you have a problem with your IDE, please read the note at the bottom of the lab). Then add the dependencies in the *pom.xml* file:

```
<dependencies>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-core</artifactId>
            <version>1.2.1</version>
        </dependency>
</dependencies>
```

Here, your IDE should ask to download the dependencies. This step varies depending on your IDE.

Now go to https://hadoop.apache.org/docs/r3.2.1/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html and copy the code of the WordCount v1.0 into your IDE (create a new file).

3.1. What are the arguments of this program?

3.2. Construct a JAR file for WordCount (this step depends on your IDE). You want to have a *wordcount.jar* file at the end of this stage.

Send *wordcount.jar* to the namenode (like *peter_pan.txt*). You can now run the program with:

```
hadoop jar wordcount.jar /data/peter_pan.txt /data/word_count_peter
```

(Note that depending on how you built the jar, you may need to specify the class name, like in *hadoop jar wordcount.jar WordCount /data/peter_pan.txt /data/word_count_peter*)

You will see that the job is submitted to the cluster. To check that your job actually ran/is running on the cluster, you can go to http://localhost:8188 or http://localhost:8188.

Once the task is done, check the results. What do you notice about */data/word_count_peter*? What does it contain?

3.3. Let's now have a look at the code. It contains two classes *TokenizerMapper* and *IntSumReducer* that extend the generic classes *Mapper* and *Reducer*. By using the documentation of Hadoop, find what the parameters of these classes are.

3.4. What are the inputs and outputs of the map and reduce functions? How is the emission of the results done? What is the type of data emitted?

3.5. Using the code for the word count as an example, write a map-reduce task that computes the mean length of words. Before starting, think carefully about what the inputs and outputs of the map and reduce methods are. Besides, think about what the keys are. We want the result as a double, so replace the *IntWritable* by *DoubleWritable*. Also, modify the job name in the main method and be careful with all the parameters there.

What is the mean length of words in the file *peter_pan.txt*?

4 Scala

Spark is a complex language that is often used with Spark. You had an introduction during the previous lecture, and you will find many resources online to deepen your knowledge. Here, we are going to practice the basis that we will use in the labs. To code with Scala, you can use the same IDEs as Java (like Eclipse or IntelliJ). In general, they have special plugins for Scala.

4.1. Create a new SBT project in Scala. Then, in *src/main/scala*, create a new singleton object that contains a main method. Make it print "Hello World!" to test that you can run a program.

4.2. Scala is a functional programming language. Therefore, it is essential to understand how functions work. Start by creating a new object with the main method. Then, using the **def** keyword, write a function that takes as input an Int and prints its squared value. Test that it works correctly.

4.3. In the main method, define a list that contains five elements: 1, 2, 3, 5, 10. Then, using a *for* loop, apply the scare method to all the elements.

4.4. When possible, we try to avoid using loops in Scala, and we prefer a functional approach. For this purpose, Scala data structures come with methods such as *def map[B](f: A => B): Seq[B]* that take as input a function. Notice here how the type of functions is constructed. To apply a function to all elements of a collection, we have two functions: *map* and *foreach*. The difference is that *map* returns a new collection whereas *foreach* return nothing (*Unit*). Modify the code of the previous question to replace the loop by a call to the *foreach* method.

4.5. With this kind of construction, using the keyword *def* can be pretty heavy, especially for small functions. So instead, replace the call to the square method with an *anonymous function*.

4.6. Using methods like *map* is very practical when we need to chain operations on a single line (e.g. *l.map(...).reduce(...).filter(...)*). It is often used with methods like *filter*, *reduce* or *flatMap* to produce more complex results. Using the same list as before and anonymous functions, chain the methods we have just mentioned to print square values that are bigger than 20.

4.7. Using the *reduce* function, compute the sum of the elements in a list.

5 Spark

As you may have noticed, Scala comes with functions naturally fitting the map-reduce formalism, thus making it a perfect language for this application. Besides, as we observed before, writing a Map-Reduce program in Java can be a tedious task: We need to properly give all types, inherit more general classes, configure many details of a job, ... As we will see, writing similar programs in Scala will be much easier.

5.1 Spark Shell

Spark comes with a shell that allows fast experiments. Connect to the *spark-master* container and go to */spark/bin*. There, execute *./spark-shell --master spark://spark-master:7077*. Note that if you do not specify the master, the code will be executed on the local machine rather than the cluster. You will get access to a shell in which you can program in Scala. In the messages appearing when starting the spark-shell, you can see that a Spark context Web UI is available on port 4040. Connect to it from your browser to observe the jobs in your current application (here, a spark-shell).

5.1. The shell comes with a default *SparkContext* called *sc* to interact with the cluster. You can read a text file using *sc.textFile*. By default, it will get the file in the local file system (not the HDFS). Associate the file */spark/README.md* to a variable. What is the class of this object? In how many partitions is it divided? Call the *first* function on this variable to obtain the beginning of the file (the function *collect* allows to get everything). Now call the *count* function on the text file. What does it count? Can you find why?

5.2. Now, associate the file */data/peter_pan.txt* to a variable. Print the first line. What do you observe, and why did it not happen earlier?

5.3. We need to say the file in on the HDFS. To do so, you will have to prefix the file name with *hdfs://namenode:9000*. What is the first line of *peter_pan.txt* and how many lines does it contain?

5.4. To turn a text file into a list of words, we can do: *text_file.flatMap(line => line.split(" "))*. What does the *flatMap* function do? What is the class of the returned object of *flatMap* when applied to an RDD?

5.5. We have seen that Scala comes with *map* and *reduce* functions. However, when we want to do a Map-Reduce, we need to execute the *reduce* function for each key and not for the complete collection. To do so, we have access to the *reduceByKey* function that does the same thing as *reduce* for each key (a key-value pair is represented by a tuple *(a, b)*). What is the difference between the reduce function for a real Map-Reduce (like we saw in Java/in class) and the function *reduceByKey*? Can you find how we can emulate a "real" Map-Reduce (you need to find a function that emulates the shuffle stage)? In general, we prefer to use the *reduceByKey* method because it is faster and avoids a lot of data transfers. In the rest of the lab, we will use this function.

5.6. Using the functions we have just seen, count the number of occurrences of each word. What do you notice? Now add *.take(10)*. What do you notice now? Can you guess why?

5.7. Compute the average word length. Note that you can use *.collect()* to get all results. Try to use *reduceByKey* and to not precompute the total number of words.

5.8. Connect to the Spark context WebUI (port 4040). Do you see the jobs that you executed in the spark-shell? Do you see them on the timeline? On the *Stage* tab at the top, you can access the stages of your jobs. By clicking on one of them, you will get various metrics that can be useful to check the health of your application. Now go to the *Executors* tab. How many active nodes do you see? How many tasks were executed for each worker?

5.9. Connect to the spark-master WebUI (port 8080). Do you see the spark-shell? How many workers do you see? How much are they occupied? Quit the spark-shell. What is the worker occupation now? Can you still access the Spark context WebUI?

5.2 Spark Program (optional)

Having a shell to test your code can be convenient but should be avoided when going to production or for long computations. Here, we will write a Spark program, compile it and submit it to the cluster.

Add the following lines to *build.sbt*:

```
scalaVersion := "2.12.10"
val sparkVersion = "3.1.2"
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % sparkVersion
)
```

5.10. Using the following skeleton, write a word count program.

```
import org.apache.spark.{SparkConf, SparkContext}

object WordCount {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("Word count")
    val sc = new SparkContext(conf)
    // Your code here
  }
}
```

You can use the function *saveAsTextFile* instead of *collect* or *take* to save the results of the computation in a file (do not forget the prefix hdfs://namenode:9000). Generate a jar file for this program, send it to *spark-master* and submit the job with (in */spark/bin*):

```
./spark-submit --master spark://spark-master:7077 --class WordCount /spark.jar
```

Check the WebUI to see if your job was executed. You can use a bigger text file to see a running job in the WebUI. Check the resulting file. How many parts does it contain?

6 Notes

- If the namenode enters safe mode: *hadoop dfsadmin -safemode leave*. Often, if you enter in safe mode, it means that you have closed your containers in an unexpected way. As a result, you will usually have to remove the volumes and redo the setup (create /data in the HDFS and copy peter_pan.txt there).
- Delete all volumes:

```
docker volume rm docker-hadoop-spark_hadoop_datanode1
    docker-hadoop-spark_hadoop_datanode2
    docker-hadoop-spark_hadoop_historyserver
    docker-hadoop-spark_hadoop_namenode
```

- If you have problems using your IDE to compile the Map-Reduce in Java, you can also compile it directly in a container. For that, you need to send your Java file to namenode and execute there:

```
export HADOOP_CLASSPATH=$(hadoop classpath)
mkdir WordCount
javac -classpath ${HADOOP_CLASSPATH} -d WordCount/ WordCount.java
jar -cvf WordCount.jar -C WordCount/ .
hadoop jar WordCount.jar WordCount /data/peter_pan.txt /data/word_count
```

- If you start the same program on Hadoop several times, it will fail if the output path already exists. So do not forget to remove it if necessary.