Advanced Concepts of Cloud Computing
TP-2 Report

Rad Ali
Polytechnique Montréal
rad.ali@polymtl.com
1900574

Mathurin Chritin
Polytechnique Montréal
mathurin.chritin@polymtl.com
1883619

Adam Naoui-Busson
Polytechnique Montréal
adam.naoui@polymtl.com
1887197

Philipp Peron
Polytechnique Montréal
philipp.peron@polymtl.com
2230874

***Index Terms*—AWS EC2, Python, Hadoop, Spark, MapReduce**

I. Introduction

The repository can be found under: https://github.com/PhilippPeron/advanced-cloud-log8415

II. Choice of Cloud Platform

We selected AWS as the platform with which we would deploy our solution. We made this choice because we wanted an automated solution and we already had a big portion of the automation code written. This would save us time in testing and debugging because all the instances could be created without having to manually input all the specifications.

III. Experiments with WordCount program

The first test we ran was aimed at understanding Hadoop and the WordCount program. We ran this program on the Ulysses text file to count all instances of different words inside it. We timed the operation and obtained a real time of 1.494 seconds.

IV. Performance comparison

A. Hadoop vs. Linux

We ran two tests to compute the word frequencies of the contents of Ulysses to allow us to understand the performance differences that Hadoop provides. The first test was on a standard Ubuntu operating system and we obtained a real time of 0.811s. The second test, done with Hadoop, gave us a real time of 1.494s. From these results, we can observe that the Hadoop performance is slower compared to the native Linux performance. The reason for this is the relatively long initialization time of Hadoop.

B. Performance comparison of Hadoop vs. Spark on AWS

We ran multiple tests to compare the performance of Hadoop and Spark on 9 datasets. These datasets contained lengthy texts on which we used the WordCount program from Hadoop and Spark. These tests were run 3 times and we used the average of these results to create the following figure 1. The results show that Spark was about 0.03 seconds faster than Hadoop in our tests. This is contrary to what the results are supposed to be. Actually, Spark should be slower than

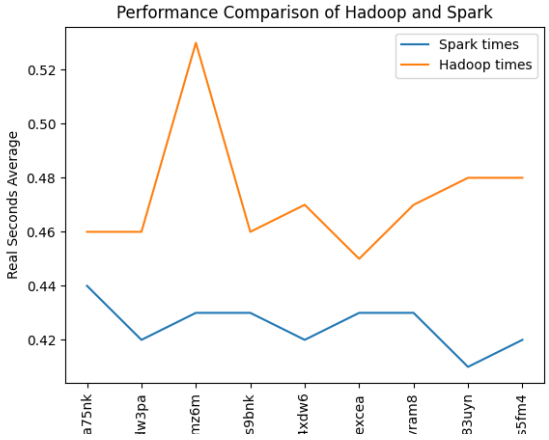

Fig. 1. Runtime comparison of Hadoop and Spark

Hadoop for this problem because it takes longer to initialize and start the task. For a short task like this word-counting example Hadoop should therefore outperform Spark due to its quicker initialization time.

V. Describe how you have used MapReduce jobs to solve the social network problem

We developed a fully automated solution that executes our MapReduce jobs on an AWS instance. Our approach consists of creating a key pair, and starting an m4.large instance configured to be accessible via the previously created key. That way, we can log in through ssh, which makes the automation easier. This part was done using the `boto3` Python API : we create a security group (or take the one previously created if needed), create a key pair and save the .pem private key file locally, and setup an instance up and running with all of this.

Even though we use `wait_until_running` in order to wait for the instance to start before trying to ssh into it, we noticed it was not enough for the instance's `openssh` deamon to start properly. Trying to ssh into the instance right after `wait_until_running` will fail most of the time. That's why we use `netcat` in our script : we ping the port 22 every 3 seconds until we can see it is ready

to accept connections, and proceed to the next step only after that. This makes the Python setup operations and the Hadoop and Map Reduce setup operations easily pipeline-able.

The next part consists of logging in the instance and installing Hadoop on it. Using ssh, we start by cloning our repository there. Then, we execute our little Hadoop installation script (`install-hadoop.sh`). It will take care of installing the required packages, downloading Hadoop, and set it up to work in our environment.

Once Hadoop is installed, we can use it to compile our MapReduce implementation (`MapReduce.java`). We create a single .jar file that will be then executed on the dataset.

Our .jar takes an input and an output directory as arguments. We therefore use hdfs to create these directory directly on the Hadoop file system. We use the hdfs CLI to simulate a real distributed Hadoop execution, but we could have directly used a classic `mkdir -p`, since we are only executing on one instance and locally. Then, we unzip the dataset (pushed in our repo) and copy it in the input directory using `hdfs dfs -copyFromLocal`.

We are now all set to execute our Map Reduce algorithm on the dataset : `hadoop jar mapreduce.jar MapReduce input output`. We wrapped this command to be able to redirect the output to a file `stats.time` and print it as well in the terminal. We also `time` this command.

At this point, the resulting recommendations are located in a `output/part-r-xxx` text file. We use `hdfs dfs -copyToLocal` to copy the output to the instance filesystem.

This ends our ssh command. Once done, we can then proceed and copy back locally the results and the statistics. We use `scp` with the same key used to ssh into the instance. That's it ! The results are the following file :

- `output.txt` : The friend recommendations following the specified format
- `stats.time` : The outputted logs from the Map Reduce executed with Hadoop. At the end of this file, we can also find the execution time of the algorithm.

The complete Map Reduce execution time typically will take between 1min and 1min30sec.

VI. Describe your algorithm to tackle the social network problem

To solve this problem we had at our disposal a file of entries which respected a particular format. Each user was represented by an integer number (its identifier). In this way, each line of the input file indicated the id of the user in question followed by the list of ids of his friends (each id being separated by a comma). So we could easily read this input file by reading it line by line, then for each line by separating the text between each comma. Then comes our MapReduce implementation.

Since we had to determine the potential friends we could have through mutual friends, we wanted our reduce function to have as a key the id of a user and in values a list of potential friends. This being the case, the problem was also to recommend no more than 10 potential friends per individual. Moreover, this recommendation had to prioritize the potential friends having the most friends in common with the said individual. Therefore, to meet all these requirements, we first implemented the map function.

To do so, as it was said earlier, we used the id of a user as the key and as the value we used an object containing the id of a potential friend with the id of the mutual friend. In this way, for each line of the input file, we can say that: for each friend of the list of friends of the user, this friend is a potential friend with all other friends of this list with each time the individual as mutual friend. However, the algorithm must not propose a user who is already a friend.Therefore, for each existing friend pair we have placed a flag (-1 in the mutual friend id field) in the emitted object. In that way, in the reduce function, we will be able to not consider users that are already friends. Let's now see the implementation of the reduce function.

Let's recap. The reduce function will have for key the id of a user and a list of objects containing a potential friend with its mutual friend as values. Therefore, we have declared a hashmap to record for each potential friend the list of it's mutual friends. That's when the flag we talked about earlier comes into play. Indeed if we detect this flag it means that the potential friend is already a friend and we will put a value of *null* for the list of mutual friends.

Once we have processed all the potential friends and filtered out those with whom the user is already friend with, we had then sorted these potentials friends according to the size of their mutual friends list.

Finally we have only kept the first N (N=10) potential friends having the most friends in common with the user.

VII. Recommendations of connections

Below are some examples of the results we obtained for the friend recommendations:

- User 924 : 439,2409,6995,11860,15416,43748,45881
- User 8941 : 8943,8944,8940
- User 8942 : 8939,8940,8943,8944
- User 9019 : 9022,317,9023
- User 9020 : 9021,9016,9017,9022,317,9023
- User 9021 : 9020,9016,9017,9022,317,9023
- User 9022 : 9019,9020,9021,317,9016,9017,9023
- User 9990 : 13134,13478,13877,34299,34485,34642,37941
- User 9992 : 9987,9989,35667,9991

- User 9993 : 9991,13134,13478,13877,34299,34485,34642, 37941

VIII. Summary of results and instructions to run our code

A. Hadoop-Spark Benchmark and Map Reduce problem solving

To run the Hadoop-Spark benchmark and the Map Reduce job on the dataset, just run the following :

```
$ chmod +x script.sh && ./script.sh
```

This will, in order, do the following :

- Fire up an m4.large instance and setup it to use a created key pair
- ssh to this instance and :
  - install Hadoop
  - clone our repository
  - compile `MapReduce.java` and create a .jar file
  - unzip the input dataset
  - create the hdfs input directory and copy the input dataset inside it
  - run the Map Reduce algorithm on the dataset
  - save the results in the output directory
  - save the Map Reduce logs and the execution time in a file `stats.time`
- Copy locally the results of our Map Reduce friend recommendation to a local `output.txt` file
- Copy locally the execution logs and time to a local `stats.time` file
- Print the execution time
- Terminate the instance
- Fire up another m4.large instance to run the benchmarks on
- ssh to this new instance and:
  - install Hadoop and Spark
  - clone out repository
  - install matplotlib
  - run the benchmark script
  - run the results analysis script
- Copy locally the benchmark log file
- Copy locally the results graph
- Terminate the instance