POLYTECHNIQUE MONTRÉAL

UNIVERSITÉ D'INGÉNIERIE

POLYTECHNIQUE MONTRÉAL

LOG8415E

ADVANCED CONCEPTS OF CLOUD COMPUTING

Lab 2

MapReduce with Hadoop on AWS

Authors:
2009913 - Jordan Mimeault
2018968 - Antoine Lombardo
2020511 - Jacob Dupuis
2024785 - Alexandre Dufort

Lab Instructor:
Vahid Majdinasab

Instructor:
Amin Nikanjam

https://github.com/JordMim/LOG8415E

November 7 , 2022

Contents

1 Experiments with WordCount program.

To wrap all WrapCount implementations, we decided to create a Docker image that expose a REST API allowing us to send WordCount tasks to any desired implementation. This Docker image is available on Docker Hub (‘lombardoa/tp2‘) and can be run locally or on a AWS Instance.

Running the image locally is straight forward. Deploying the image on a AWS instance can be automatically done using our script and will be shown in the last section of this document.

We started by creating a Docker image based on Ubuntu 20.04. We then installed Java and Hadoop and did the configuration for Hadoop to run in standalone mode. We then copied the input file ‘pg4300.txt‘. After testing it by passing through the instance shell, we then created a Flask wrapper that run Hadoop MapReduce task using the linux ‘time‘ command, wait for the completion and read and process the output of both Hadoop and time. By doing so, we can use the REST API to start MapReduce task and receive its execution time and its output as a response. We used this interface to do the performance comparisons of the next section.

For the Linux WordCount program, we simply created a bash script that splits the content of the input file in words by using the ‘tr‘ command, then sort these words by using the ‘sort‘ command, then count the occurence of each words using the ‘uniq -c‘ command. The script output its result in the stdout stream. We then added a Flask route for running a WordCount using this script, again using the ‘time‘ command.

Finally, implementing Spark WordCount was straigth forward, since there are multiple guides online explaining how to it. We then created a Python script that execute a WordCount program using Spark, then once again wrapped it inside our Flask app.

This gives us three different routes that we used in the next section to create a benchmark script:

- http://{address}/hadoop/wordcount/{input_file}
- http://{address}/spark/wordcount/{input_file}
- http://{address}/linux/wordcount/{input_file}

2 Performance comparison of Hadoop vs. Linux

To perform this comparison, we run the WordCount script 10 time for both Hadoop and Linux to get an approximate average run-time for each. These tests were made on the AWS Instance. However, the same tests have been done locally and results were confirmed the ones showed bellow. Here is a graph comparing those two metrics.

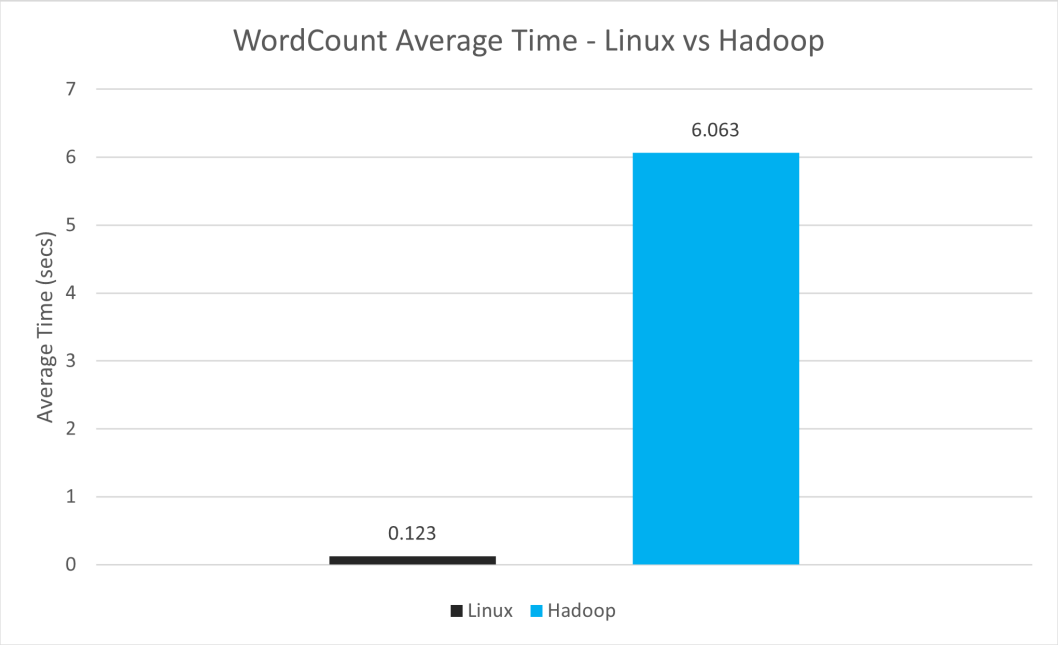

Figure 2.1 - Linux vs Hadoop average time

From the above graph, we can conclude that Linux is about 50 time faster than Hadoop. This can be explained by the fact that Hadoop works on the the disk, unlike Linux script which runs in memory. Here, the dataset is not big enough for Hadoop to outperform Linux, and the initialization phase of Hadoop alone is enough to make it way slower than the Linux script.

3 Performance comparison of Hadoop vs. Spark on AWS.

To perform this comparison, we run the WordCount script 3 times on each file for both Hadoop and Spark to get an approximate average run-time for each file. Again, these tests were made on the AWS Instance, but have also been done locally to confirm their validity. Here is a graph comparing Hadoop and Spark performances.

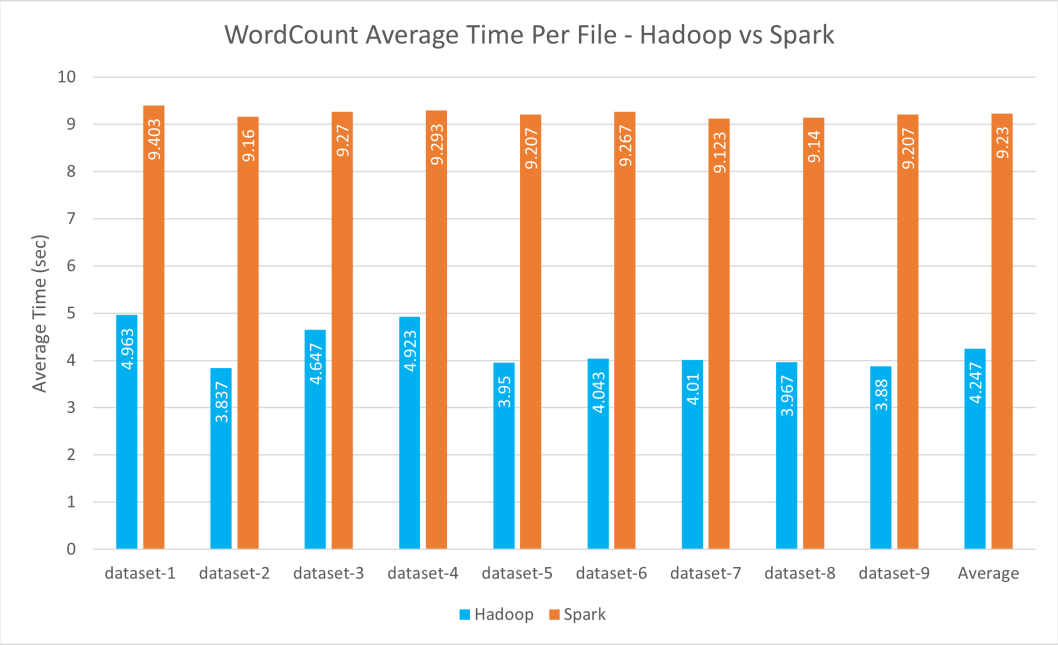

Figure 3.1 - Linux vs Hadoop average time

From the above graph, we can conclude that Hadoop is about 2 times faster than Spark. This is in contradiction with what we tought would happen, as Spark is known to be able to run up to 100 times faster than Hadoop. This once again can be explained by the fact that Spark has to load the dataset in memory and make connections between everything it has loaded in memory, which are two very expensive operations. With a very large dataset, this step would have a minimal impact over the whole task, but since our dataset is very small, these two steps have a large impact on the results.

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

We used MapRoduce to solve the Social Network problem. MapReduce is a technique used to divide a problem then solve in parallel using multiple machine. It divides the problem in smaller groups so it can be solved faster. The sum of all the result obtain are then all put together to obtain your the final solution.

Figure 4.1 - MapReduce on Big Data

In our implementation, the input is splitted in users/friends entries. These entries can then be divided in smaller groups. We process these entries in our Map function, which outputs first and second degree friends associated to each user. A reduce step is then used to process the output of the map step in order to build a set of recommendation for each user. This way, the problem can be solved in parallel on different threads/machines.

5 Describe your algorithm to tackle the social network problem.

To solve the social network problem we had to use the map reduce algorithm to treat the data. Initially, we had every user id and their list of friends. The goal is to recommend to the user a list of mutual friend between himself and his friends.

To approach this problem, we created a Recommendation class that has two attributes: a int representing the friend to add as a recommendation and another int representing the mutual friend. The mutual friend field has a value of -1 if he is a direct friend of the current user.

For the Map function, it process a text line to retrieve the user and his list of friend. From the moment that a user has friends, for each friend we add him to the list of friends of the user. Then, we write with the user as the key and a recommendation of the friend with a mutual friend value of -1 as the value, the format is as followed : Recommendation(friend, -1). This indicates that this direct friend can be omitted in the reduce step. After all friends are added to the friends list, we wrote recommendations in both ways for each pair of friend, because they all have the current user as mutual.

For the Reduce function, we start by counting all the recommended users that have the same key and that are not direct friend (!= -1) by merging the same keys in an HashMap. Then, we used a comparator to sort our map by reverse key and then by value. After filtering the entries by removing the recommended users already friends with the user, we kept the 10 most recommended friends and added them to a final recommendation list of friend. The final step was to write this final recommendation as the values and the user as the key in the format of the initial text file.

6 Recommendations of connection for the users

This is our recommendation for the user with following user IDs: 924, 8941, 8942, 9019, 9020, 9021, 9022, 9990, 9992, 9993.

- **924:** 439, 2409, 6995, 11860, 15416, 43748, 45881
- **8941:** 8943, 8944, 8940
- **8942:** 8939, 8940, 8943, 8944
- **9019:** 022, 317, 9023
- **9020:** 9021, 9016, 9017, 9022, 317, 9023
- **9021:** 9020, 9016, 9017, 9022, 317, 9023
- **9022:** 9019, 9020, 9021, 317, 9016, 9017, 9023
- **9990:** 13134, 13478, 13877, 34299, 34485, 34642, 37941
- **9992:** 9987, 9989, 35667, 9991
- **9993:** 9991, 13134, 13478, 13877, 34299, 34485, 34642, 37941

The list of all filtered Ids (*output-report.json*) and the list of all Ids (*output-all.json*) can be found in the *tp2/social-network-client* directory

7 Summary of results and instructions to run your code.

To conclude, the above results shows us that both Hadoop and Spark gives no advantages when working with small datasets. However, if our datasets were a lot larger and the problem more complex, using Hadoop or Spark could have a significant performance impact over a plain Linux solution.

The script needs to be run and should run correctly on any linux environment. However, it has only been tested on a Debian environment. To run our code, Python 3, Pip 3, Git and AWS CLI have to be installed on your machine. To make sure the script is executable, the command `sudo chmod +x script.sh` must be executed. Our bash script can then be run as root with the command `sudo ./script.sh`. Out git repository will then be automatically cloned and the `run.sh` script will be executed, which will launch an interactive dialog.

Alternatively, the script can be run manually. To do so, the repository must be cloned using this command: `git clone https://github.com/JordMim/LOG8415E.git`. The script will then be located in the directory `LOG8415E/tp2`. The script must be set as executable using the command `sudo chmod +x run.sh`, and can then be run using the command `sudo ./run.sh`.

The deployment of our application on an AWS instance can be done using the first option of our run.sh script. This script guides the user while configuring AWS, then run a Python script that uses Boto3 for creating and configuring the Security Group and the Instance.

To simplify the deployment on our AWS Instance, we decided to create a Docker image on which every dependencies are installed. By user a user_data script, we automatically install Docker on the Instance and run a container based on this Docker image. This Docker image is based on Ubuntu 20.04 and contains Java 11, Hadoop 3.3.4, Python 3, PySpark 3.3.1. It also contain all the datasets. This all-in-one image gives the user the access over all the functionality through a REST API that respond to the port 80.

More infos on the usage of the script is provided in the ***README.md*** file of our repository.

For each step of the script, the normal output should look like these:

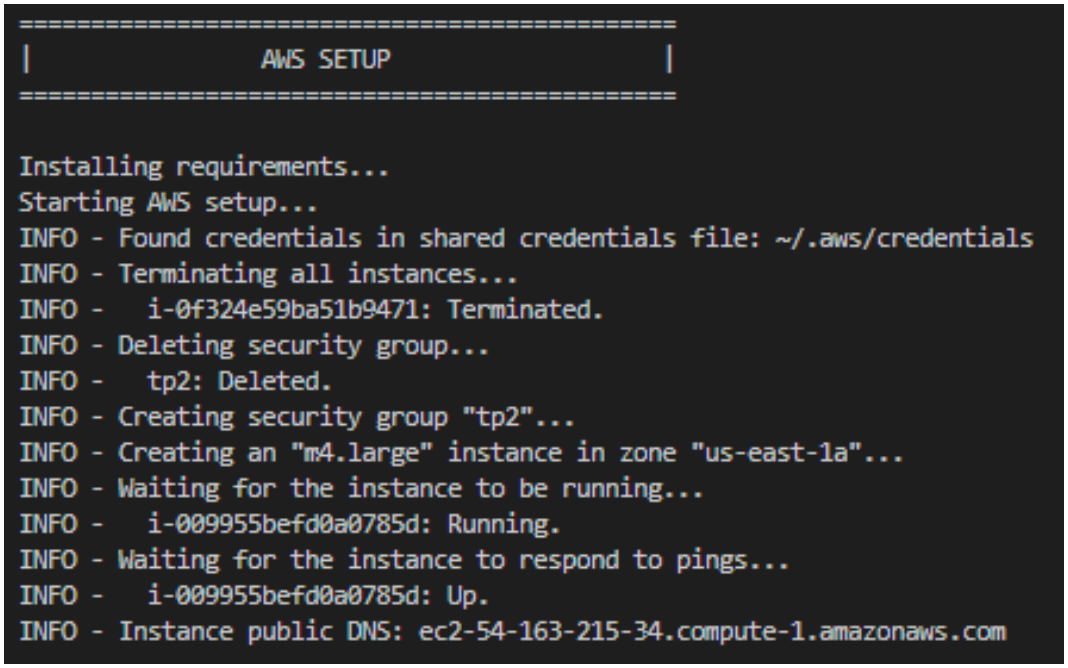

Figure 7.1 - Output of the AWS Setup step

Figure 7.2 - Output of the Benchmark step

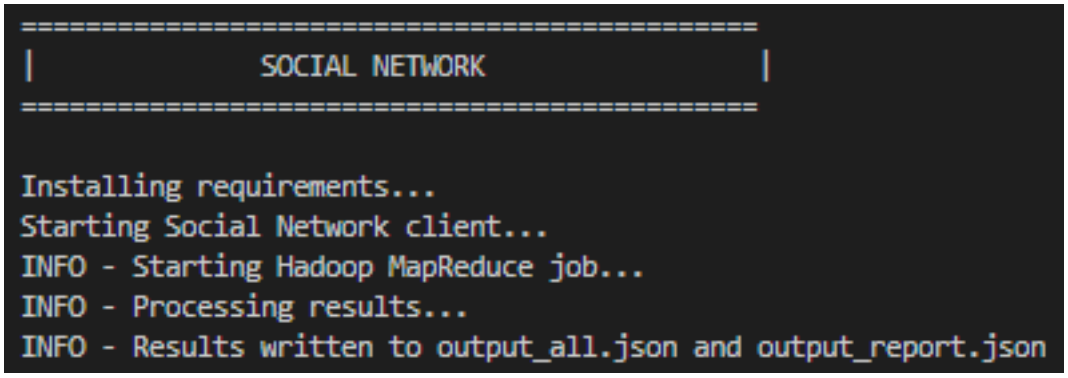

Figure 7.3 - Output of the Social Network step