POLYTECHNIQUE MONTRÉAL

UNIVERSITÉ D'INGÉNIERIE

Lab 2 : MapReduce with Hadoop on AWS

Presented to Lab Teacher Vahid Majdinasab

By :
Jérémy Thai - 2016336
Jules Lefebvre - 1847158
Ismail Bakkouri - 1954157
Milen Bozhilov - 1899742

LOG8415E - Advanced Concepts of Cloud Computing

October 31st, 2022

1. Introduction

The object of this assignment was to leverage the MapReduce programming model through both Hadoop and Spark executed on a single AWS EC2 instance. The objectives of the assignment were four-fold:

- Automate the setup and deployment of both Hadoop and Spark on an AWS EC2 instance.
- Compare the processing time of a simple WordCount task with Hadoop versus an approach using Linux-native commands and pipes—both running on an AWS EC2 instance.
- Compare the performance of Hadoop and Spark when processing WordCount problems on a single AWS EC2 instance.
- Implement a MapReduce algorithm using Hadoop to solve a Social Network problem.

In this report, we present the results and our analysis of the performance comparisons, we describe the algorithm we developed to solve the Social Network problem and provide instructions to run our code.

2. Hadoop and Spark Deployment Procedure

Our code leverages the python script EC2_instances_creator.py developed for Assignment 1 to create an AWS EC2 m4.large instance. We fed this new instance with a bash script (launch_script.py) which automates the installation and configuration of Java, SSH, Git, Hadoop and Spark on the instance.

During the deployment phase, a public Git repository (https://github.com/miboz/files) is cloned on the AWS EC2 instance, it contains the following:

- The datasets for the WordCount and Social Network problems;
- The provided Hadoop implementation of the WordCount problem;
- The JAR file for our Social Network algorithm.

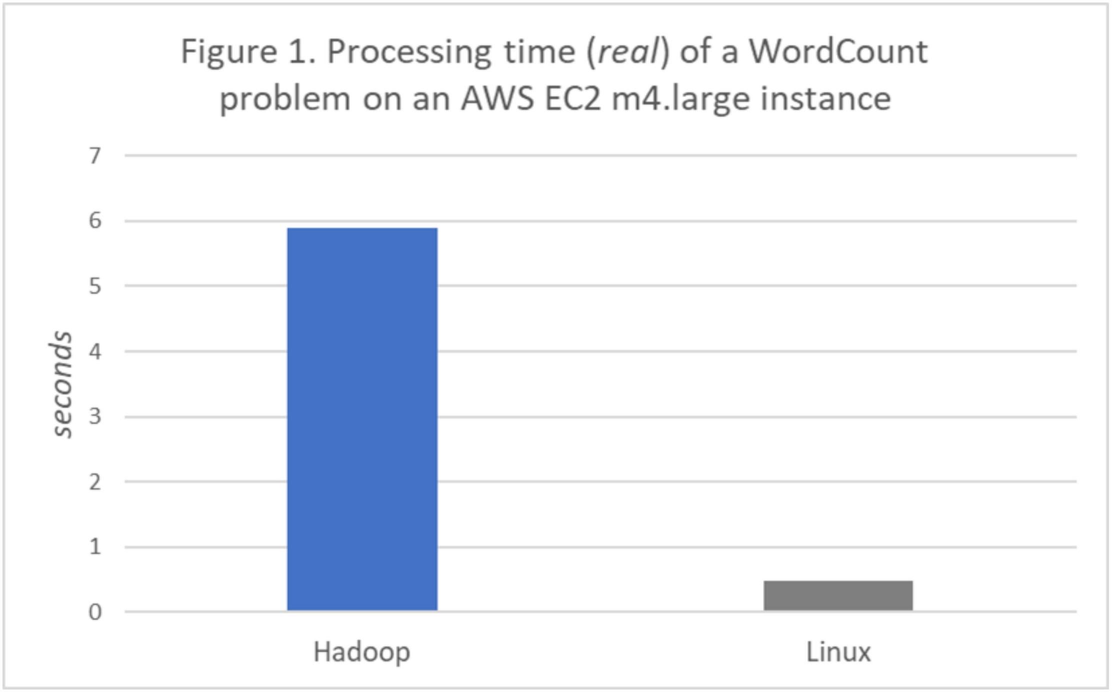
3. WordCount results

3.1 Hadoop vs Linux

The first experiment consisted in computing the word frequency of the contents of James Joyce's Ulysses (n = 265,222) on an AWS EC2 m4.large instance using WordCount.java on Hadoop versus simply using the following Linux commands :

```
cat 4300.txt | tr '␣' '␣[ret]␣' | sort | uniq -c
```

The processing times (given by the Linux **time** command) are presented in **figure 1** below and show that the Linux command is about 12 times more efficient than the single-node Hadoop algorithm for this particular task.



The reason is because Hadoop was created with the intent to process a large volume of large files. Therefore, it will not perform well when storing and processing a small volume of small files.

3.2 Hadoop vs Spark

The second experiment consisted in running WordCount with Hadoop and Spark on an AWS EC2 m4.large instance for nine (9) datasets. The processing jobs were run three (3) times each.

The results presented in **figures 2**, **3** and **4** show that Hadoop's average processing time (real) was 38.9% of Spark's processing time however, the CPU time spent in user-mode for Spark was only 2.38% of the same statistic for Hadoop.

This disparity can be explained by the fact that the setup for Spark requires more time than Hadoop but the actual processing rate is up to 100 times faster with Spark than Hadoop.

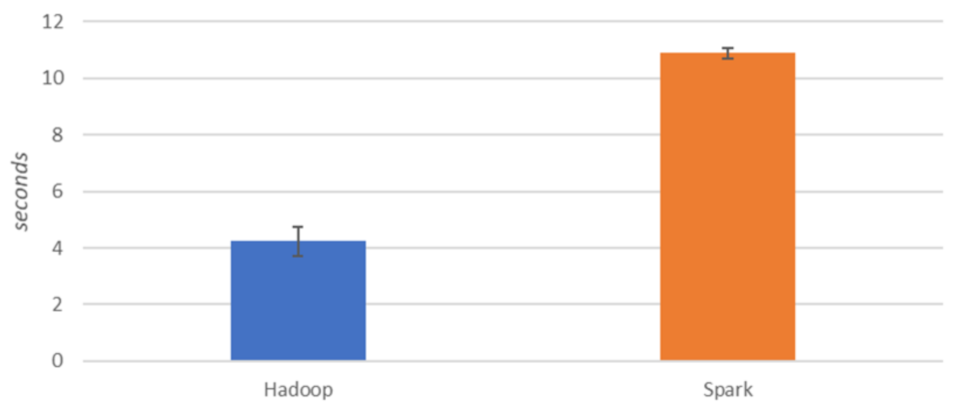


Figure 2. Average processing times (*real*) of WordCount problems on an AWS EC2 m4.large instance

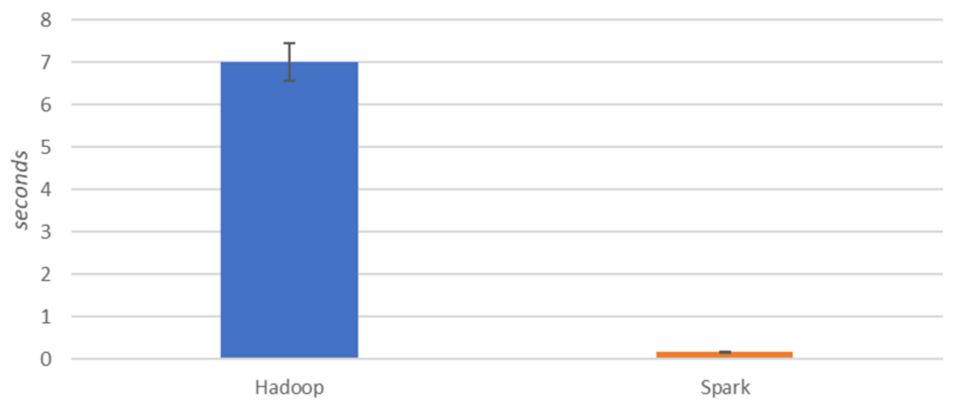


Figure 3. Average processing times (*user*) of WordCount problems on an AWS EC2 m4.large instance

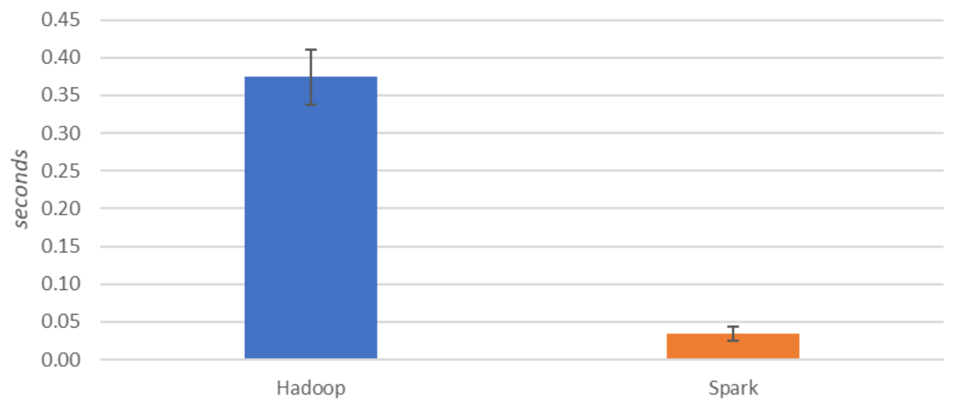


Figure 4. Average processing times (*sys*) of WordCount problems on an AWS EC2 m4.large instance

4. Social Network Problem

The algorithm we developed to solve the Social Network problem leverages the MapReduce paradigm with Hadoop. It consists of the following four Java classes :

`PeopleYouMightKnow.java`
is the main class which links :
- the mapper class,
- the reducer class,
- the helper class,
- the input and output paths.

`RecommendedFriend.java`
Is a helper class which defines an object with attributes :
- `Integer recommended`
- `Integer mutual`

Where both attributes are set to -1 when two users are friends.

`Map.java`
The mapper parses a row of text containing the list of friends of a user (`<user><tab><friendA, friendB, friendC, …>`) and does the following :
- For all user-friend combinations : create a pair (user, RecommendedFriend) with a mutual friend value of -1 and write it to the context.
- For all friend-friend combinations : create a pair of mutual friends and write it to the context.

We provide a simple example below to illustrate the behavior of the mapper class.

Table 1. Example input/output of the Map class of our Social Network algorithm

	Input	**Output**
Format	User friendA, friendB	Key, (recommended, mutual)
Example	1 2,3	1, (2, -1) 1, (3, -1) 2, (3, 1) 3, (2, 1)

`Reduce.java`
The reducer executes the following steps :

- The `HashMap mutualFriendMap` is populated by iterating through the recommended friends of a given key (user). The resulting map has a friend recommendation (ID) as key and the number of mutual friends as value.
- The `SortedMap sortedMutualFriends` is populated from `mutualFriendMap` where the comparator function sorts by value (number of mutual friends) and by key (ID) if the values are equal.
- The output is limited to 10 recommended friends per user and written to the context.

We present a subset of the resulting recommendations of the execution of our MapReduce algorithm on the provided Social Network dataset in **table 2** below.

Table 2. Friend recommendations for a subset of users of dataset soc-LiveJournal1Adj.txt

Users	Recommandations
924	439, 2409, 6995, 11860, 15416, 43748, 45881
8941	8943, 8944, 8940
8942	8939, 8940, 8943, 8944
9019	9022, 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

5. Instructions to run the code

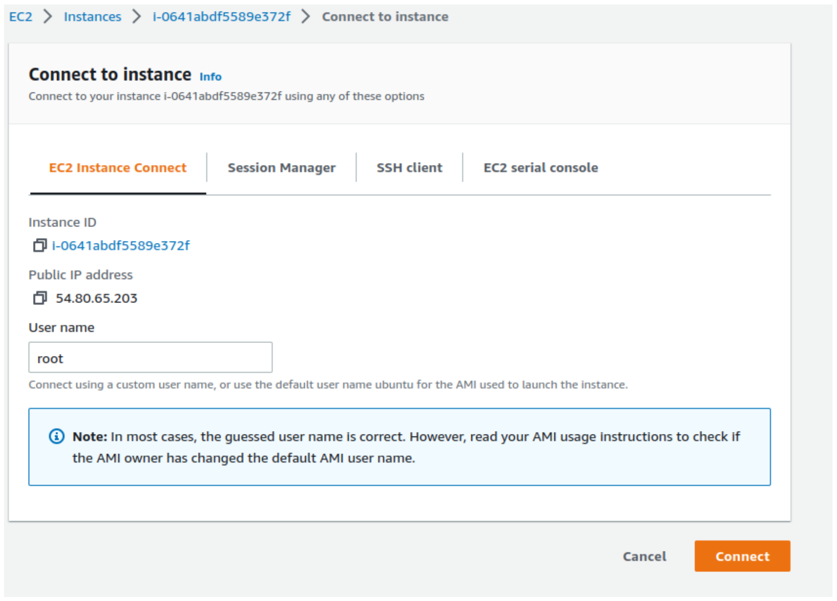
Before executing the main Python script, the following prerequisites must be met:

- The code must be located on the local machine
- Python 3 must be installed locally on the machine
- The Boto3 library (Python SDK for AWS) must be installed locally on the machine
- User should have an AWS account

The next step is to set up the authentication credentials for the AWS account. If it does not already exist, a file named "credentials" should be created under "~/.aws/". Inside this file, both the aws access key id and secret access key should be specified. This will allow the boto3 client to have access to AWS. Another important file to have is the "config" file, also located under "~/.aws/". This file should specify a default region to use such as "region=us-east-1".

Afterwards, the main Python script called main.py can be run. This script will automatically spin up an EC2 instance. On its launch, the launch_script.sh will be called to set up the configurations for both Hadoop and Spark and perform the "WordCount" execution time comparison between Hadoop, Linux and Spark (results found in time_results.txt).

As for the friend recommendation algorithm, its command can be run manually on the ec2 instance through the console (see the command at the end of launch_script.sh). Results will be stored inside the folder "sn_output". Here is an example on how to connect to the instance using Instance Connect:



Connect to your instance i-0641abdf5589e372f using any of these options

Instance ID
i-0641abdf5589e372f

Public IP address
54.80.65.203

User name
root

Connect using a custom user name, or use the default user name ubuntu for the AMI used to launch the instance.

Note: In most cases, the guessed user name is correct. However, read your AMI usage instructions to check if the AMI owner has changed the default AMI user name.

The scripts are publicly available in the following GitHub repository: https://github.com/J-Lefebvre/LOG8415E.