LOG8415
Advanced Concepts of Cloud Computing

TP2:
MapReduce with Hadoop on AWS

November 12, 2022

Petr Šmejkal
Felipe Manoel
Koen van Oijen
Lucia Čahojová

Contents

1 Abstract

As an elaboration of our previous work, the aim of this project is to experience running MapReduce using Amazon Web Services. Firstly, we run the proposed big data operations using two big data tools, Hadoop and Spark. We then compare and report the performance results. Secondly, we solve the social network problem using MapReduce. Our results show that for our assignment, running WordCount program on Linux is the fastest. Hadoop is the second best and Spark the slowest in generating the output.

2 Introduction

As an extension of our previous work with Amazon Web Services, in this team project we focus on running MapReduce on AWS.

Firstly, we execute experiments with WordCount program, to see what tool is more time efficient in calculating the output. We try three different approaches, Linux, Hadoop and Spark on a given dataset. After executing all three versions of the calculations, we provide and discuss the results, comparing the tools we used.

Secondly, we apply our new skills to implement "People You Might Know" algorithm using Spark and the MapReduce paradigm. In our algorithm, for each user we recommend ten other users who they are not friends with yet, but have the most number of mutual friends in common.

We provide an automated solution of both of these problems and also detailed instructions on how to run the code.

3 Experiments with WordCount Program

In this section, we discuss three different approaches to running the WordCount Program and compare their time performance to see which one is the fastest. The input files were provided to us and we downloaded them to the virutal machine with **curl** command.

The WordCount problem has been solved by three different approaches (Linux, Hadoop, Spark) and the time was in each case measured as an average of three separate runs.

3.1 Linux

When counting the number of word occurrences in a file using Linux commands and pipes, the command sort firstly reads the entire file into memory and reorders it. Then the uniq command only compares adjacent lines of code. Below is the command used in our script.

```
cat input.txt | tr ' ' '\n' | sort | uniq -c [ret]
```

3.2 Hadoop

To count the number of word occurrences in a file, Hadoop firstly has to read and write files to Hadoop Distributed File System (HDFS), which stores files and parallelizes them across a cluster. Then it parallelly processes the data using the MapReduce algorithm [1]. The solution to this problems comes as an example with Hadoop, so we only needed to run the provided .jar after loading all the data to HDFS.

3.3 Spark

When solving the WordCount problem, Spark unlike Hadoop processes data in RAM using Resilient Distributed Dataset(RDD). RDD is an immutable collection of elements that can be operated on in parallel, therefore the computations are carried out parallelly in memory and stored there. Spark also creates a Directed Acyclic Graph (DAG), which enables optimizations between MapReduce steps [1].

3.4 Performance Comparison of Linux vs. Hadoop

As we have already mentioned, the major use of Hadoop is to handle a process big data, which by definition is data that contains greater variety, arriving in increasing volumes and with more velocity [2]. However, as it turns out, out data set does not actually meet these requirements, so using just the Linux commands generates the results much faster, average time shown in Figure 1.

```
The average time for all files to be processed by Linux is 0.140000 seconds.
```

Figure 1: Content of wordcount_linux.log

Most extra time has been spent when actually setting up Hadoop in order to run (moving files to HDFS and other overhead), so Hadoop is in our case a bit of an overkill, the result time is shown in Figure 2. However, it could definitely be faster on different (bigger) data sets.

```
The average time for all files to be processed by Hadoop is 49.451000 seconds.
```

Figure 2: Content of wordcount_hadoop.log

3.5 Performance Comparison of Hadoop vs. Spark on AWS

The provided data set is quite small (in terms of big data), so the disk operations are not that demanding in our case. It means that most of the "heavy lifting" is done in the memory.

Hadoop is used mainly for disk-heavy operations with the MapReduce paradigm, and Spark is a more flexible, but more costly in-memory processing architecture.

Spark has particularly been found to be faster on machine learning applications, but for our WordCount case, we actually found that Spark is slower than Hadoop. This makes sense since the data set is small, hence the most time is spent by loading data to the memory rather than actual operations done to process the data.

```
The average time for all files to be processed by Spark is 51.374000 seconds.
```

Figure 3: Content of wordcount_spark.log

3.6 Performance Comparison Conclusion

Finally, to sum the results up, Linux was the fastest way to run the WordCount with provided datasets. Second fastest was Hadoop and Spark was the slowest. This is to be expected if you

consider the principle on which each of those tools is built and the size of the data set. The results would differ if the data set was bigger.

4 Social Network Problem Resolution Using MapReduce

To solve the Social Network Problem, we adopted a mapper to connect friends from a user by the key-value pair showing that they have a friend in common, it will also identify the direct friends. Then, the reducer receives these values related to a specific user and finds 10 of them with the highest number of friends in common as specified, not considering the direct friends. The algorithms used on both of them are specified below.

4.1 Social Network Problem Resolution Algorithm

We used the Hadoop Streaming API with Python programs to develop our MapReduce solution.

In our solution for the Social Network problem, the mapper outputs a key-value pair for each pair of friends in the friend list of an user. The key is this friend and the value is the other friend.

The mapper also generates a key-value pair for each friend in friend list of an user, where the user is the key and the friend is the value, this was done in the code by a nested loop with both of them iterating the array of friends. To all these key-value pairs, an identifier is added in the value to discern which are direct friends and which are not.

Then, the reducer counts the number of friends in common for each candidates presents in the key-value pairs, eliminating the ones which are direct friends.

Finally, it sorts the candidates and gets the 10 more suited to recommend. To count the number of friends, we created a dictionary with the id of the friend being the key and the number of friends in common being the value, then if the value that was in the dictionary was 0 or the new value is 0 it will set the value in the dictionary to 0 which means that they are direct friends and should not be considered in the recommendations, otherwise they will be added.

To sort the values, we turn the dictionary into an array and then we first sort by id in ascending order and then we sort by number of friends in common in descending order, so if two possible friends have the same number of friends in common, we will get them in ascending order. After that we select the first 10 elements of the result array and remove any invalid recommendations, which are the ones with number of friends in common equal to zero.

4.2 Recommendations of Connections

Using our algorithm, we got a output file from which it was possible to get recommendations shown in Table 1.

User	Recommendations
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	13134,13478,13877,34299,34485,34642,379
9992	9987,9989,35667,9991
9993	9991,13134,13478,13877,34299,34485,34642,37941

Table 1: Recommendations for some users

5 Instructions to Run the Code

In this section, we provide the steps with commands to run our automated procedure:

1. Configure AWS Credentials on your computer:
   (a) Open the file with AWS credentials:
   `code ~/.aws/credentials`
   (b) Navigate to your AWS Academy account, then to AWS Details and copy your AWS CLI information.
   (c) Paste the AWS CLI information to the opened credentials file and save and close the file.
2. Clone the GitHub repository to your desired location:
   `git clone https://github.com/smejkalpetr/cc-tp2.git`
3. Proceed to the project's directory:
   `cd cc-tp2`
4. Run the automated application:
   `. ./setup.sh`
5. See the WordCount results in the `./logs/wordcount_*.log` files.
6. See the Social Network Problem results in the `./logs/social_network_problem.log` file.

6 Conclusion

We implemented two different programs using MapReduce on AWS. In the first one, we solved the WordCount problem on Linux, Hadoop and Spark. Our results show that Linux was the fastest, followed by Hadoop and then Spark. Even though the literature suggests that Spark should be the fastest, our results prove that it heavily depends on the size of the dataset. With Hadoop and Spark, additional resources are needed to be downloaded, which makes solving the problem on a small set of data too long, however, it is beneficial on bigger dataset.

Then we successfully implemented the Social Network problem using MapReduce paradigm and computed friend recommendations for ten users, shown in Table 1.

References

[1] https://logz.io/blog/hadoop-vs-spark/

[2] https://www.oracle.com/ca-en/big-data/what-is-big-data/