AWS LAB 2: MapReduce with Hadoop on AWS

November 11, 2022

Aura Kiiskinen - aura-aksiina.kiiskinen@polymtl.ca
Cedric Helewaut - cedric.helewaut@polymtl.ca
Nick Reiter - nick.reiter@polymtl.ca
Victor Mesterton - victor.mesterton@polymtl.ca

Abstract

This report presents the written elaboration of the second assignment for the Advanced Cloud Computing lecture of École Polytechnique de Montréal (LOG8415E). The first section of the report documents the two experiments with the WordCount MapReduce program. A first experiment consists of comparing the performance for WordCount between a Linux system and Apache Hadoop. A second experiment compares the performances of Apache Hadoop versus Apache Spark on Amazon Web Services EC2 instance. The second section reports on a proposed MapReduce program for solving the "People You Might Know" program. This problem requires determining a list of suggested friends to all users in the input file, based on mutual friends.

1 Introduction

The target of the assignment was to create an automatic script that is comparing the processing time of a word count problem on Linux, Spark and Hadoop. This was done by using results of the previous assignment and adding additional code snippets. Furthermore, this project report contains the detailed description of a MapReduce program solving the "People You Might Know" problem.
The following sections of the documentation will explain each step more in detail in order to solve the previously described tasks. The first section puts a focus on the word count problem. Foremost, background knowledge of technologies and other important terms are given. This is followed by the instance set-up, providing the hardware environment on AWS on which the benchmarking on Linux, Hadoop and Spark is running. Thirdly, the outcomes of the comparison and the influencing theoretical functioning are depicted. The second section mainly covers the development of a suitable solution for the social network problem. The first subsection explains, how MapReduce jobs can solve the social network problem. This is done by considering mapper and reducer separately. The algorithm is written in Java and the explanation of its structure is revealed in the second subsection. Lastly, a recommendation sample set is given to provide insights into the outcomes of the algorithms.

2 Experiments with WordCount program

2.1 Background

- Hadoop: Apache Hadoop is a framework, storing and processing large data sets in a distributed manner.
- MapReduce: Mapreduce is a programming paradigm, enabling large scalability and parallelism across a multitude of servers. MapReduce is the main code programming paradigm behind Hadoop.
- general MapReduce program: The MapReduce service exists of two main operations, namely the Mapper and Reducer. A general MapReduce process will start with one Mapper or a sequence of Mappers mapping input key/value pairs onto intermediate key/value pairs. At this stage the intermediate values with the same key values might be combined to reduce the total amount of intermediate values, if desired. Next the Reducer or sequence of Reducers will combine the intermediate values and return the desired output key/value pairs.

- Spark: Apache Spark is a multi-language engine for executing data engineering, data science and machine learning on single-node machines or clusters.
- WordCount: WordCount is a simple application counting the number of occurances of each word in a given input set. This algorithm is often used as an introductory program into working with Hadoop and MapReduce in general.

2.2 Instance set-up

The Github repository providing all necessary code and information is available at:
https://github.com/elcaballero69/cc-assignment1/tree/main/Assignment_2 Use the 'lab2_script.py' script to...

We automated the creation of two m4.large instances. The instances are using us-east-1a availability zone and security group that allows SSH and HTTP traffic. We included java and hadoop as well as java and spark installation commands to the userdata, the first for one instance and the following for another. Therefore while the instances are being created, all the needed libraries are installed to them automatically. We also included the commands to download the input files to the instances while creating them. After the instance creation we wait for the instances to be running as well as 5 minutes to make sure the installations have finished. This was necessary since the hadoop installation file is relatively large and takes some time to download.

2.3 Performance comparison of Hadoop vs. Linux

After the instance creation, including installation of Hadoop, we connect to the instance with paramiko and execute shell commands. For the connection, one needs to download the .pem file from the AWS lab and add it to the root of the project. Script is automatically accessing this .pem file from the root as well as using the public IP address of the instance in order to connect. After the connection we run two shell commands, one for Linux and another one for Hadoop.

Linux command splits the text in the input file at all spaces and line breaks, sorts the words and counts the unique words. While this command is executed, we create a file: time_linux.txt and store the execution time there. This is later used to access the execution time in order to compare it with the Hadoop execution time.

For the Hadoop WordCount, we execute the example MapReduce WordCount algorithm that is already preinstalled with the Hadoop library. The time is also stored in the time_hadoop.txt in a similar way as was done for the Linux execution. At the end, we have two text files stored at the instance showing the execution times.

After the WordCount is performed with both Linux and Hadoop, we read the values in the text files. In our test run, the execution times were 0m0.5666s for Linux and 0m5.986 for Hadoop. The execution time of the Linux is 5.4 seconds faster than for the Hadoop. We consider this to be caused by Hadoop having overhead. Hadoop needs to load, read and transform the files into HDFS format and the execution is using storage instead of memory.

Compared to Linux, that is performing the WordCount directly in memory without making any file transformations. To conclude, memory is more efficient than storage in computation. However, the Linux memory has it limits that makes it inefficient for large data processing jobs. Hadoop is constructed to handle large data sets compared to a few text files.

2.4 Performance comparison of Hadoop vs. Spark on AWS

To compare the performance of Hadoop and Spark for running the WordCount MapReduce program, we run a sequence of nine input data sets on both systems, deployed on AWS EC2 instances. We repeat this experiment three times for both programs, to verify the consistency and reproducibility of the results.

The connection to the instance, on which Spark is running, and the storing of runtime information is realized equally to the previous chapter.

In figure 1, we can see the outcome of the three iterations. The first iteration of Spark needs around seven seconds to process the text files. Subsequently, a sharp decline can be detected in the second iteration to under four seconds. This is kept stable in the third iteration. The behaviour of Hadoop is different. The first iteration needs six seconds, followed by an increase to around seven seconds, which is kept stable throughout the benchmarking.

The difference in computing time can be traced back to the nature of logical functioning. Like already stated, Hadoop makes use of storage during loading, reading and transformation of the text files, whilst Spark make use of memory. Thus, Spark should be faster than Hadoop. It seems to be wrong that the processing time of the first iteration reveals that a Hadoop environment processes the WordCount problem faster. During initialization, Spark is facing overhead time due to the creation of a SparkContext object, which tells Spark how to access a cluster. This is the reason why Spark is slower in the first iteration compared to Hadoop. After the second run, Spark can exploit its fast processing advantages and is, hence, faster than Hadoop.

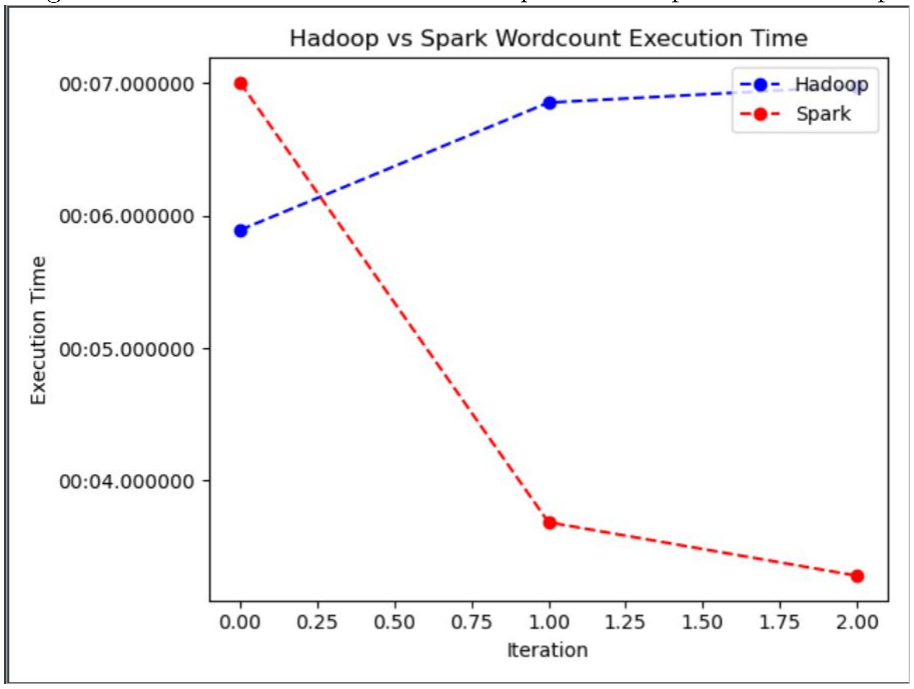
3 A proposed "People You Might Know" MapReduce program

3.1 How MapReduce jobs solve the social network problem

The second part of the assignment consists of designing and implementing our own MapReduce program, to solve the "People You Might Know" problem. This problem requires determining a list of suggested friends to all users in the input file, based on mutual friends.
For our solution we took inspiration from this blog post for the intermediate value pairs:
https://www.dbtsai.com/blog/2013/hadoop-mr-to-implement-people-you-might-know-friendship-recommendation/

1. **Mapper:** The provided input consists of a list of users and their friends in the ¡USER¿¡TAB¿¡FRIENDS¿ format. We decided to map these values onto intermediate values containing three person IDs. Two potential friends and either a mutual friend or an invalid value indicating

Figure 1: Three Iteration runtime Comparison of Spark and Hadoop



they are already friends. The second type of intermediate value are values indicating all possible variations of two people that have the user as a mutual friend. For each variation the key wil be the first persons ID and the value will be a combination of the userID and the second persons ID.

2. **Reducer:** In the reducer, we construct a hashmap, which maps all potential friends of the personID passed in the input key to a list of all mutual friends. Each time an entry passes with two valid IDs, the first ID (mutual friend) is added to the list of the second ID. When an entry passes with only the second ID valid, it means this valid ID is already friends with the input key person. Therefore their entry in the hashmap is permanently set to null.
Next, after the whole value is processed, we iteratively select and remove the key from the hashmap which has most friends (and lowest ID when an ex aequo occurs).

3.2 Social Network Algorithm

Our solution code can be found at:
https://github.com/elcaballero69/cc-assignment1/blob/main/Assignment_2/socialNetwork-/PeopleYouMightKnow.java
The solution was coded in Java, since this allowed us to reuse the WordCount.Java code as a template. We replaced the mapper and reducer classes by our own versions and adjusted the configuration accordingly.
This solution uses Writable Integers, for all its writable values. However, since bitwise shifts are used to pass two integers in one Writable Integer, the maximum allowed user ID is 65533. This is no problem for the specified input for this assignment. If larger values are required,

it is suggested to replace all Writable Integers by Writable longs.
The solution consists of three classes:

1. **PeopleYouMightKnow.java class:**
   - Parent of the PYMKMapper.java class, which is our Mapper class.
   - Parent of the PYMKReducer.java class, which is our Reducer class
   - main(): Function defining the job configuration, which indicates which classes act as Mapper, which classes act as Reducer and what the input and output types are for all classes involved.
2. **PYMKMapper.java class:**
   - The key input is not used.
   - The value input is of the format '¡USER¿¡TAB¿¡FRIENDS¿' and is used to create the intermediate values of the MapReduce program.
   - The key output is an IntWritable, representing one person.
   - The value output is an IntWritable, representing a combination of either an invalid ID and a valid ID or two valid IDs. When a invalid ID is present, it indicates the valid ID and the person represented by the key output are already friends. We are able to combine two IDs in one IntWritable using bitwise shifts.
3. **PYMKReducer.java class:**
   - The key input is an IntWritable, representing one person. This is the person we are making a recommendation list for.
   - The value input is an IntWritable, representing combinations of either an invalid ID and a valid ID or two valid IDs. Based on this input, we construct a hashmap. The keys in the hashmap are all users that the person represented by the key input is not friends with yet, but has mutual friends with. The values will represent the amount of mutual friends.
   - A second part of the reducer consists of iteratively selecting the next potential friend with most mutual friends (and lowest userID in case of ex-acquo's) and removing this suggested friend from the hashmap. When no potential friends remain or ten suggestions are selected, this process stops.
   - The key output is an IntWritable, representing the person we made a recommendation list for.
   - The value output is a Text version of the recommendation list.

3.3 Recommendations for sample set of user IDs

ID	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	13134,13478,13877,34299,34485,34642,37941
9992	9987,9989,35667,9991
9993	9991,13134,13478,13877,34299,34485,34642,37941

4 Conclusion

4.1 Result summary

To put it in a nutshell, the processing time of a WordCount problem is exemplarily shown for Linux, Hadoop and Spark. From the outcomes, it is possible to derive the overall performance of the three different environments. In addition, the theoretical functioning is contributing a conclusion statement that reveals, in descending order, the generalized processing speed: Spark, Hadoop, Linux. Our proposed MapReduce-based program solves the "People You Might Know" problem. Accordingly, a set of suggested friends is provided, which is based on a list of users and a list of their current friends.

4.2 Code instructions

This section provides a step-by-step instruction to run the given implementation with ease.

1. Download the AWS Command Line Interface and install it
2. Learner Lap ≫ AWS Details ≫ AWS CLI Show
3. Execute `aws configure` in the command line
4. Copy and paste the suiting credentials from step 2 and *region=us-east-1; output=json*
5. Unzip the compressed project file to your location of wish
6. Learner Lap ≫ AWS Details ≫ Download PEM
7. Copy the `labuser.pem` file into the project directory of Assignment 2
8. Navigate with your command line to the location where you stored the project folder
9. The script requires you to pip install `boto3, paramiko, matplotlib` via the command line. Ignore if already done.
10. Run the Python script with `python3 lab2_script.py` in your command line

4.2.1 Running the proposed 'People You Might Know' program on Ubuntu

1. Install and Configure Apache Hadoop on Ubuntu OS:
   https://www.vultr.com/docs/install-and-configure-apache-hadoop-on-ubuntu-20-04/#Introduction

2. Download the input file 'input.txt' from:
   https://github.com/elcaballero69/cc-assignment1/blob/main/Assignment_2/socialNetwork-

   /input_data

3. Download the jar file PeopleYouMightKnow.jar from:
   https://github.com/elcaballero69/cc-assignment1/blob/main/Assignment_2/socialNetwork-

   /PeopleYouMightKnow.jar

4. Starts the Hadoop DFS daemons, the namenode and datanodes using:
   'start-dfs.sh'

5. Allocate system resources by launching YARN:
   'start-yarn.sh'

6. Create a PeopleYouMightKnow directory:
   'hadoop fs -mkdir /PeopleYouMightKnow'

7. Create a PeopleYouMightKnow/Input directory:
   'hadoop fs -mkdir /PeopleYouMightKnow/Input'

8. Upload the input.txt file to your Hadoop filesystem:
   'hadoop fs -put '/home/hadoop/SocialNetwork/Input/input.txt' /PeopleYouMightKnow/Input'

9. Run the 'People You Might Know' program using:
   'hadoop jar '/home/hadoop/SocialNetwork/PeopleYouMightKnow.jar' PeopleYouMightKnow /PeopleYouMightKnow/Input /PeopleYouMightKnow/Output'