Polytechnique Montréal

LOG8415 : Lab 2

Advanced Concepts in Cloud Computing

MapReduce with Hadoop and Spark on AWS

Authors
Yanis Toubal (1960266)
Guy-Hermann Adiko (2005119)
Estefan Vega-Calcada (1934346)

November 12, 2021

Contents

1 Abstract

When dealing with Big Data, we can choose from a variety of softwares to manage the data sets. A modern, scalable and cost-effective solution is to use software such as Apache Hadoop or Apache Spark to efficiently split the workload across a network of computers (*"nodes"*) and therefore lower the execution time of large tasks. In this paper, we will explore both softwares and compare their differences and evaluate their performances by conducting a few experiments.

Keywords: Amazon Elastic Compute Cloud, Benchmark, Instance Performance, Cloud Application, MapReduce, Hadoop, Spark, Big Data, AWS

2 Introduction

In this laboratory, we had the opportunity to explore the MapReduce paradigm. We successfully compared the performance of the algorithm on Linux, Hadoop and Spark with different experiments. At first, we compared their performances in a simple WordCount program and observed the differences. The WordCount program simply counts the occurrence of every single word in a document. We ran it all on AWS, Amazon's cloud computing platform, by creating an EC2 instance. Then, we used Hadoop and MapReduce to solve a social networking problem and process bigger data sets. The goal is to suggest friendships based on their connection (A.K.A People You May Know Algorithm).

This paper presents some of our experiments with a WordCount program (section 3), the results of our performance comparison between Hadoop and Linux (section 4), the results of our performance comparison between Hadoop and Spark (section 5) and our solution to the MapReduce program that implements the *People You Might Know* social network friendship recommendation algorithm (section 6 and section 7). All the code presented in this report can be found on our GitHub repository. The link is provided in the Annex (section 11).

3 Experiments with WordCount program

We implemented our own version of a WordCount program in Java. In the two following sections (section 4 and 5), we present the performances of our program using Linux, Hadoop and Spark. The code we used for our Apache Spark program is inspired from Cloudera's solution [4] and the one we used for our Apache Hadoop program is inspired by the examples from Hadoop's official documentation [5].

4 Performance comparison of Hadoop vs Linux

Figure 1: Comparison between Hadoop and Linux for the 9 texts

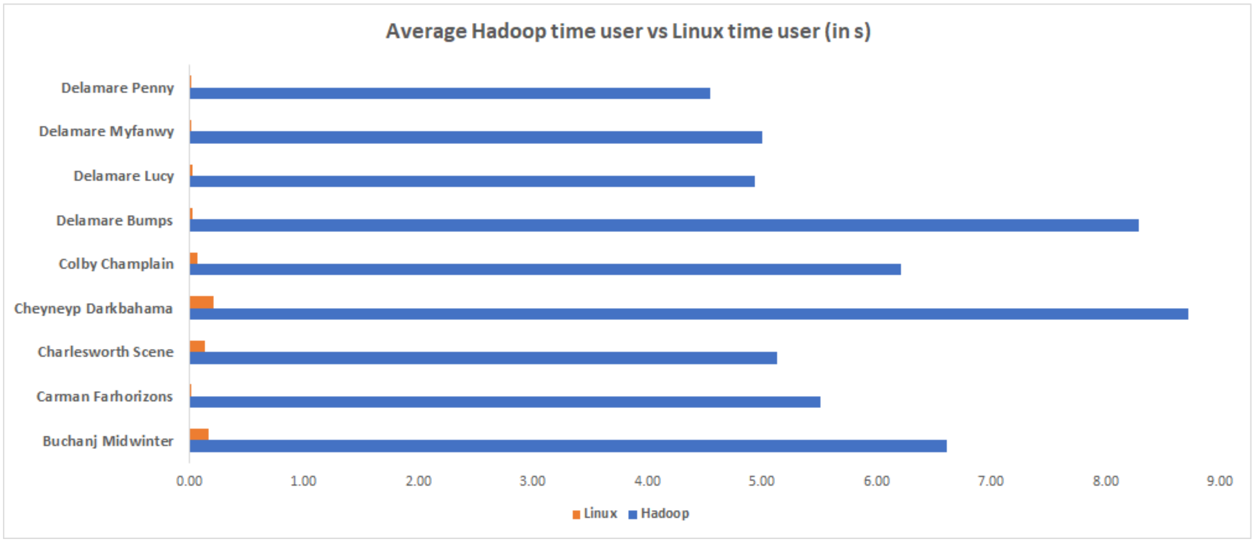


Table 1: Comparison between Hadoop and Linux for the 9 texts

Document	Hadoop time (in s)	Linux time (in s)
buchanj-midwinter-00-t.txt	6.608	0.172
carman-farhorizons-00-t.txt	5.512	0.016
colby-champlain-00-t.txt	6.216	0.076
cheyneyp-darkbahama-00-t.txt	8.720	0.212
delamare-bumps-00-t.txt	8.288	0.028
charlesworth-scene-00-t.txt	5.136	0.132
delamare-lucy-00-t.txt	4.936	0.028
delamare-myfanwy-00-t.txt	5.004	0.020
delamare-penny-00-t.txt	4.544	0.012

4.1 Results

The results show that our local Linux machine outperformed Hadoop by alot. This can be explained by the fact Hadoop is meant to process very large data sets. In this scenario, we are dealing with pretty small data sets, that is why our local Linux easily outperformed Hadoop. We would expect Hadoop to perform much better than our local Linux machine if we were dealing with larger data sets, such as the *People You May Know* algorithm 6.

5 Performance comparison of Hadoop vs Spark on AWS

For this experimentation, we leveraged AWS EC2 instances. As for the instance type, we used t3.xlarge which enables 4 vCPUs and 16 GiB of Memory. Below is the report for performance of Hadoop vs Spark. Basically, Spark outperforms Hadoop as we can see in the graph. To properly evaluate both Hadoop and Spark, we ran the WordCount three times on each machine and took the average time for both. In the following table (Table 2: Comparison between Hadoop and Spark for the 9 texts), we only show the average and not the individual results of each execution to avoid overloading this report. We also presented the data in a plot for a more visual comparison.

We used to time command in to measure the time it took to execute the wordcount program on each system:

– Hadoop: time hadoop ...

– Apache Spark : time spark-submit ...

Figure 2: Comparison between Hadoop and Spark for the 9 texts

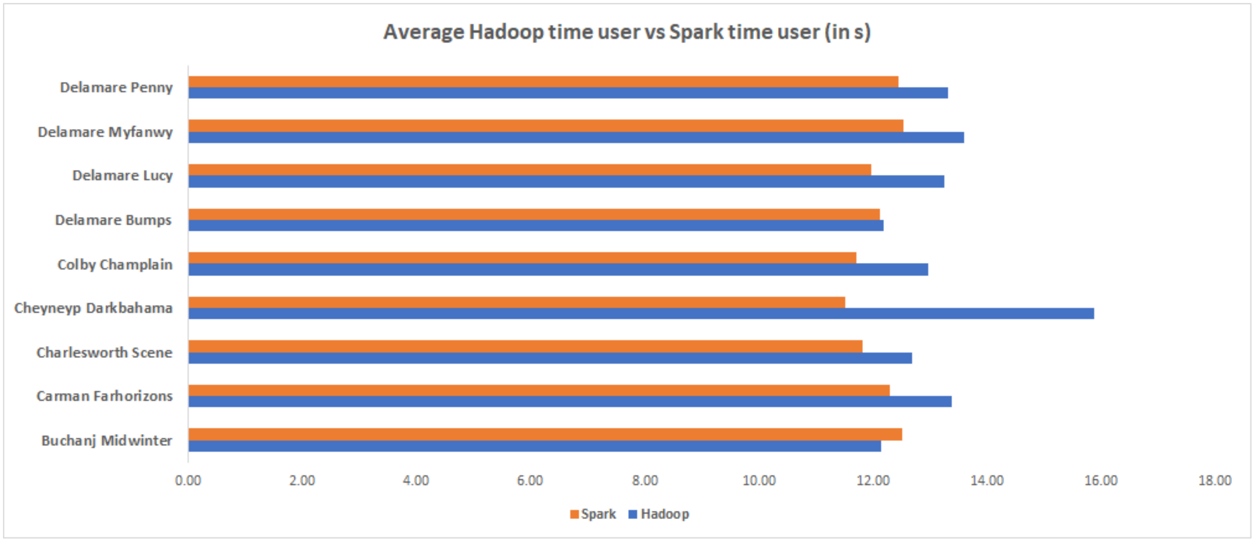


Table 2: Comparison between Hadoop and Spark for the 9 texts

Document	Hadoop time (in s)	Spark time (in s)
buchanj-midwinter-00-t.txt	12.15	12.51
carman-farhorizons-00-t.txt	13.38	12.29
colby-champlain-00-t.txt	12.97	11.71
cheyneyp-darkbahama-00-t.txt	15.88	11.50
delamare-bumps-00-t.txt	12.18	12.12
charlesworth-scene-00-t.txt	12.7	11.81
delamare-lucy-00-t.txt	13.26	11.96
delamare-myfanwy-00-t.txt	13.59	12.53
delamare-penny-00-t.txt	13.31	12.44

5.1 Results

We began to experiment with the assumption that Spark would outperform Hadoop due to the fact Apache Spark processes the data in-memory instead of disk after performing a MapReduce. The previous table did not show us what we expected. In fact, we believed Spark would be at least two times faster than Hadoop. When dealing with Big Data, Spark's performance could spike up to three times faster according to Goran Jevtic [1] and according to IBM, Apache Spark can be up to 100 times faster when dealing with smaller workloads [2]. These results could therefore be explained by the fact we aren't dealing with Big Data. Also, the times we obtained might include Spark's warmup phase and therefore is not representative of the actual processing time differences between Hadoop and Spark.

6 Description of MapReduce jobs to solve the social network problem

For the social network problem, we need an algorithm that recommends friends for each user. More particularly, It needs to find the top 10 of new friends recommendation based on the number of mutual friends. Based on the information given, we can conclude that a MapReduce approach to this problem is very much possible since there is a high amount of data involved (big data) and since this problem is very much parallelizable with the right approach.

Our approach was that for each user we wanted the key-value pair to contain the user as the key and a recommended friend as the value. To achieve this, here is how we separated the map and the reduce part.

> **Note**: The following code runs Hadoop in standalone mode, therefore we only use a single node to execute the program. Performances may vary depending on the user's computer. The average execution time was around one minute.

6.1 Map

For the map part, the idea was to represent two types of relationships between the users which are a relationship of friendship (already friends) and a relationship of potential friend recommendation. Here we use potential because at this stage we can't know if the two users are already friends or not. This verification will be done during the reduce part. For this problem, the friends of the current user are represented by a friendship relationship since they are already friends. As for the potential friends, they are simply represented by the pairs of permutations between the friends of the current user. As an example, if user 0 has friends 1,2,3, the potential friends are (1,2) (2,1) (1,3) (3,1) (2,3) (3,2). Since each map process uses as an input one user and it's friend, which are represented by a line of input text file, it's easy to scale up by using, for example, one worker per line of text.

6.2 Reduce

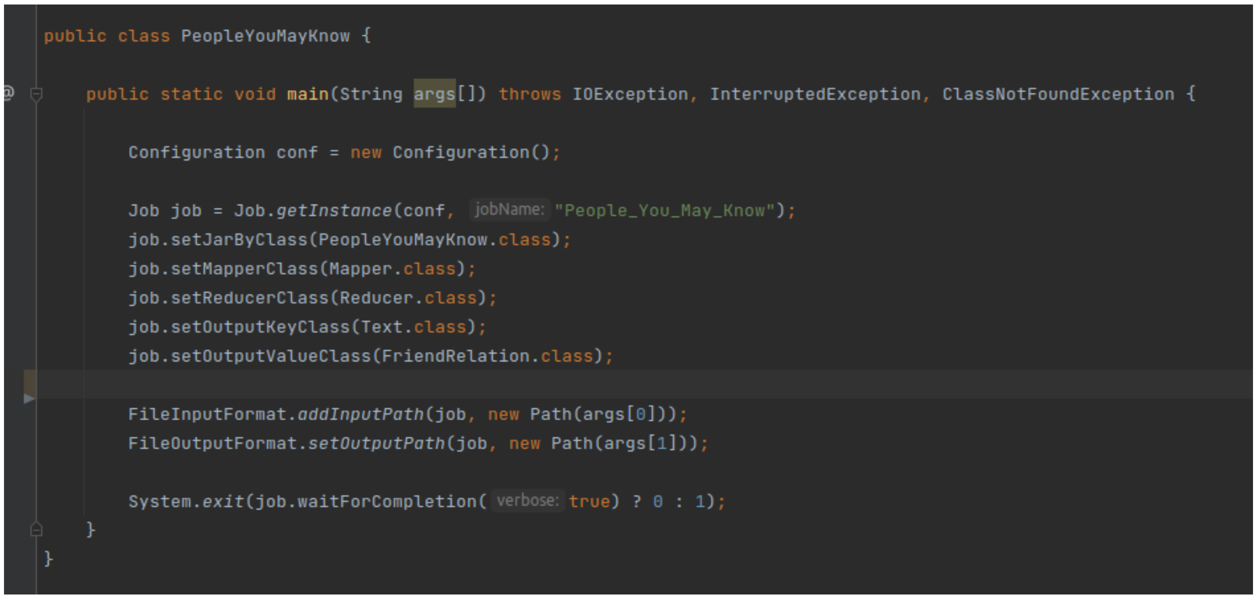
The reduce part is where most of the work happens in this situation. The idea here is to separate, for the current user, the users that are already friends with him and the users that are recommended as new friends. For the users that are already friends with the user, they are going to be ignored. As for the users that are recommended as new friends, they will be stocked along with the number of mutual friends between them and the current user. The number of mutual friends is given by incrementing everytime there is an occurrence of the recommended user (1 per mutual friend). Finally, the 10 top users with the most mutual friends with the current user will be displayed.

7 Algorithm description

7.1 Main

The main class is very straightforward. It defines the configuration of the Hadoop Job that will be run. This includes the Mapper class, the Reducer class, the input, the output and more.

Figure 3: Main class PeopleYouMayKnow.java



```java
public class PeopleYouMayKnow {

    public static void main(String args[]) throws IOException, InterruptedException, ClassNotFoundException {

        Configuration conf = new Configuration();

        Job job = Job.getInstance(conf, jobName: "People_You_May_Know");
        job.setJarByClass(PeopleYouMayKnow.class);
        job.setMapperClass(Mapper.class);
        job.setReducerClass(Reducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FriendRelation.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        System.exit(job.waitForCompletion( verbose: true) ? 0 : 1);
    }
}
```

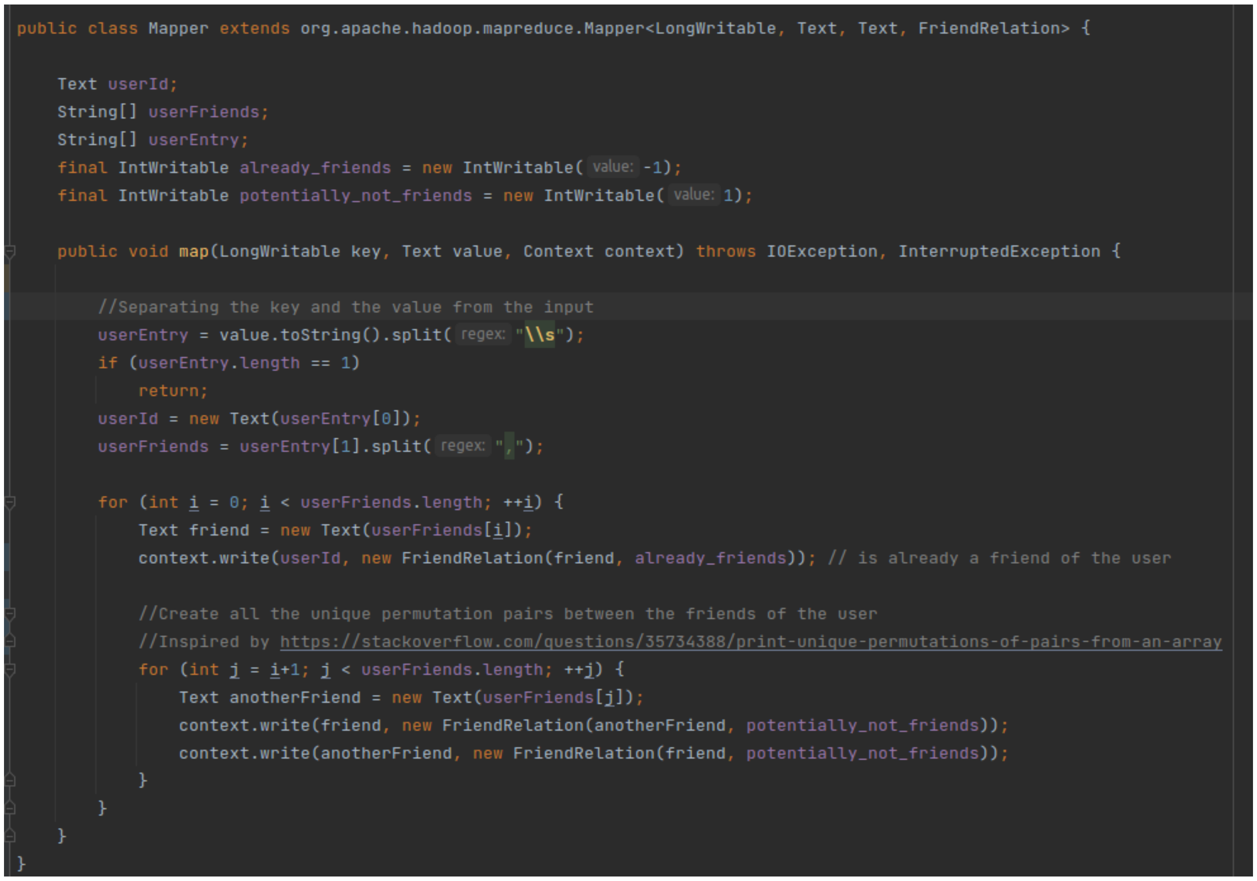
7.2 Map

1. The mapper takes the line of input from the input file and separates the current user (key) and it's friends (value).

2. An iteration is done through it's friend list and the program output the current user and his friend with an *already friend* relationship. The output is a key-value pair with the current user as key and a FriendRelation (see below) as the value.

3. After that, the program finds all the **unique pairs of permutations** between the friends of the current user and output a potential recommended friend for each permutation. The output is then a friend of the current user as a key and a FriendRelation as the value.

Figure 4: Mapper class Mapper.java

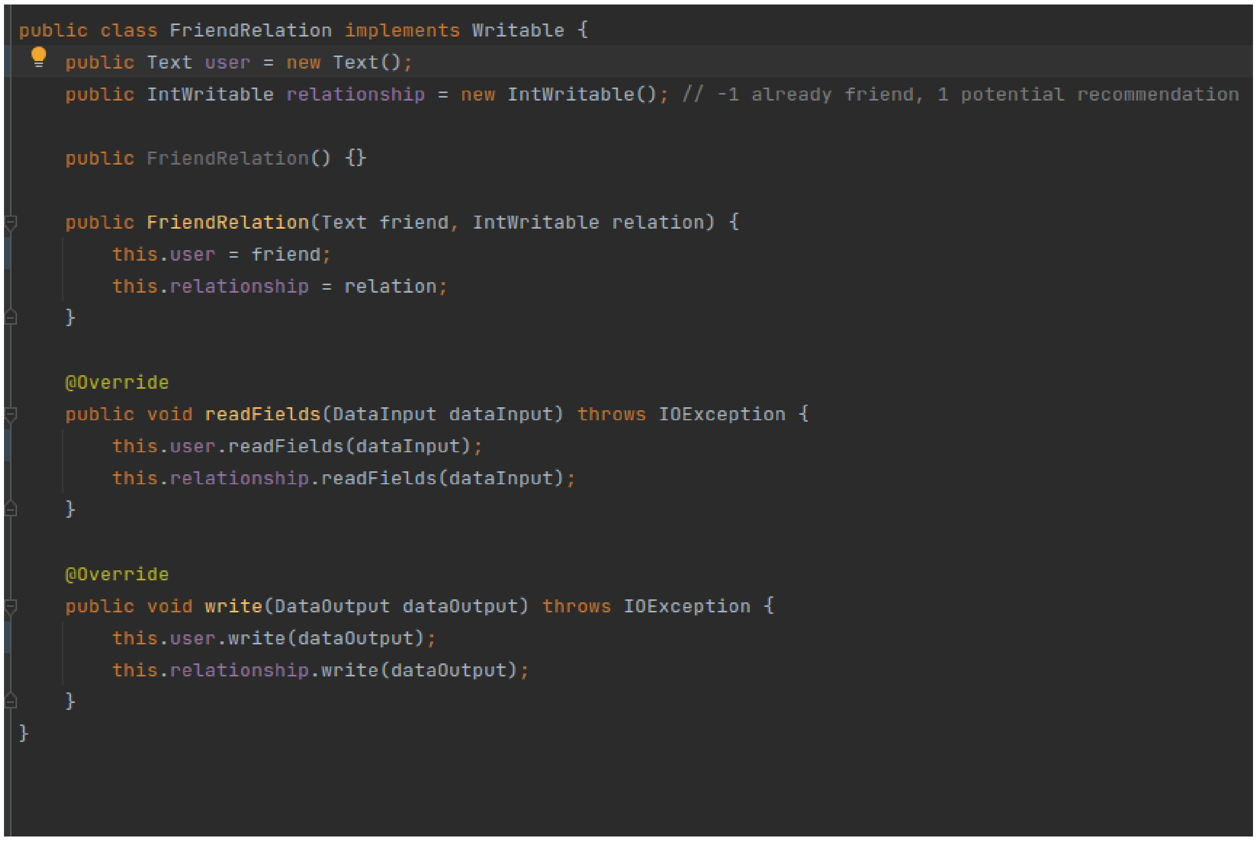


```
public class Mapper extends org.apache.hadoop.mapreduce.Mapper<LongWritable, Text, Text, FriendRelation> {

    Text userId;
    String[] userFriends;
    String[] userEntry;
    final IntWritable already_friends = new IntWritable( value: -1);
    final IntWritable potentially_not_friends = new IntWritable( value: 1);

    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

        //Separating the key and the value from the input
        userEntry = value.toString().split( regex: "\\s");
        if (userEntry.length == 1)
            return;
        userId = new Text(userEntry[0]);
        userFriends = userEntry[1].split( regex: ",");

        for (int i = 0; i < userFriends.length; ++i) {
            Text friend = new Text(userFriends[i]);
            context.write(userId, new FriendRelation(friend, already_friends)); // is already a friend of the user

            //Create all the unique permutation pairs between the friends of the user
            //Inspired by https://stackoverflow.com/questions/35734388/print-unique-permutations-of-pairs-from-an-array
            for (int j = i+1; j < userFriends.length; ++j) {
                Text anotherFriend = new Text(userFriends[j]);
                context.write(friend, new FriendRelation(anotherFriend, potentially_not_friends));
                context.write(anotherFriend, new FriendRelation(friend, potentially_not_friends));
            }
        }
    }
}
```

7.3 FriendRelation

This class is used as the output (value of the key-value pair) of the Mapper. It defines a user with it's relationship with the current user (defined in the key of the key-value pair). For example the key-value pair 1, (2, -1) could be interpreted as *user 1 is already friend with user 2*.

Figure 5: FriendRelation class FriendRelation.java



```java
public class FriendRelation implements Writable {
    public Text user = new Text();
    public IntWritable relationship = new IntWritable(); // -1 already friend, 1 potential recommendation

    public FriendRelation() {}

    public FriendRelation(Text friend, IntWritable relation) {
        this.user = friend;
        this.relationship = relation;
    }

    @Override
    public void readFields(DataInput dataInput) throws IOException {
        this.user.readFields(dataInput);
        this.relationship.readFields(dataInput);
    }

    @Override
    public void write(DataOutput dataOutput) throws IOException {
        this.user.write(dataOutput);
        this.relationship.write(dataOutput);
    }
}
```

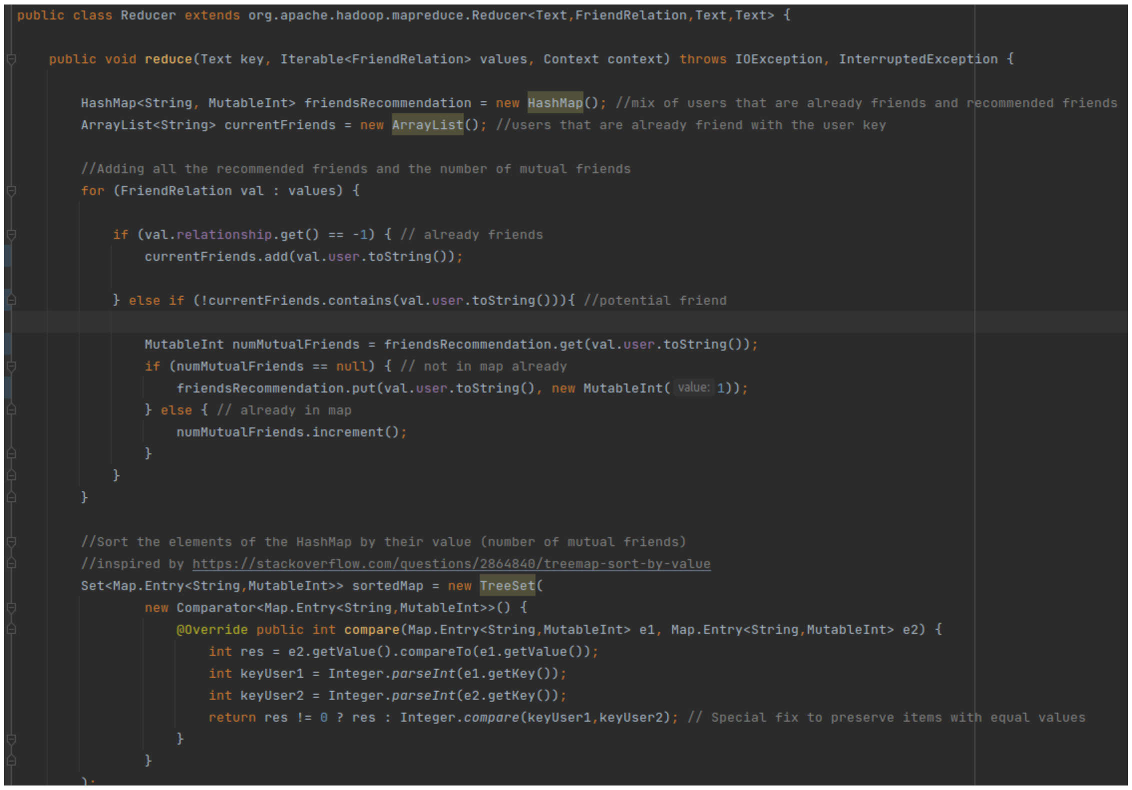
7.4 Reduce

1. The reducer takes all the FriendRelation received from the current user (defined in the key) and iterates over all of them. As mentionned before, if the relationship value in the FriendRelation is -1 then the users are already friends. In that case we put the user in a separate array (currentFriends ArrayList). If it's not -1 then they are potential friends if the other user isn't in the friends array. We then either add the new potential friend in the map (friendsRecommendation HashMap) with a value of 1 or we increment the value by 1. As previously mentionned, this value represents the number of mutual friends between the current user and the other user.

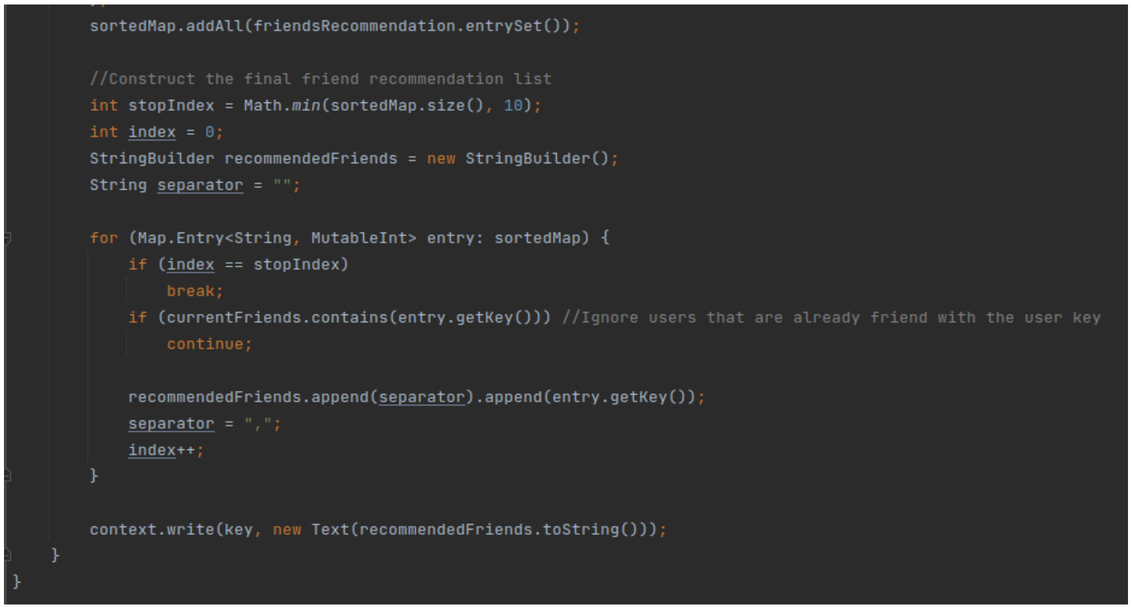
2. The next step is to take the map containing potential friends and to sort it in descending order of the number of mutual friends (value of the map). A TreeSet with a custom comparator was used for that purpose. The custom comparator first compares the value of 2 elements and then, if the values are equal, it compares the user id and prioritizes the smaller one.

3. The last step is to take the first 10 ordered recommended friends by ignoring those who are already friends with the current user. Each recommendation is added to the result that will be outputed.

Figure 6: Reducer class Reducer.java



```java
public class Reducer extends org.apache.hadoop.mapreduce.Reducer<Text,FriendRelation,Text,Text> {

    public void reduce(Text key, Iterable<FriendRelation> values, Context context) throws IOException, InterruptedException {

        HashMap<String, MutableInt> friendsRecommendation = new HashMap(); //mix of users that are already friends and recommended friends
        ArrayList<String> currentFriends = new ArrayList(); //users that are already friend with the user key

        //Adding all the recommended friends and the number of mutual friends
        for (FriendRelation val : values) {

            if (val.relationship.get() == -1) { // already friends
                currentFriends.add(val.user.toString());

            } else if (!currentFriends.contains(val.user.toString())){ //potential friend

                MutableInt numMutualFriends = friendsRecommendation.get(val.user.toString());
                if (numMutualFriends == null) { // not in map already
                    friendsRecommendation.put(val.user.toString(), new MutableInt( value: 1));
                } else { // already in map
                    numMutualFriends.increment();
                }
            }
        }

        //Sort the elements of the HashMap by their value (number of mutual friends)
        //inspired by https://stackoverflow.com/questions/2864840/treemap-sort-by-value
        Set<Map.Entry<String,MutableInt>> sortedMap = new TreeSet(
                new Comparator<Map.Entry<String,MutableInt>>() {
                    @Override public int compare(Map.Entry<String,MutableInt> e1, Map.Entry<String,MutableInt> e2) {
                        int res = e2.getValue().compareTo(e1.getValue());
                        int keyUser1 = Integer.parseInt(e1.getKey());
                        int keyUser2 = Integer.parseInt(e2.getKey());
                        return res != 0 ? res : Integer.compare(keyUser1,keyUser2); // Special fix to preserve items with equal values
                    }
                }
        );
```



```java
        sortedMap.addAll(friendsRecommendation.entrySet());

        //Construct the final friend recommendation list
        int stopIndex = Math.min(sortedMap.size(), 10);
        int index = 0;
        StringBuilder recommendedFriends = new StringBuilder();
        String separator = "";

        for (Map.Entry<String, MutableInt> entry: sortedMap) {
            if (index == stopIndex)
                break;
            if (currentFriends.contains(entry.getKey())) //Ignore users that are already friend with the user key
                continue;

            recommendedFriends.append(separator).append(entry.getKey());
            separator = ",";
            index++;
        }

        context.write(key, new Text(recommendedFriends.toString()));
    }
}
```

8 Recommendations of connection for some users

By looking at the output file provided by the Hadoop MapReduce job, here are the results we found for the requested users:

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

9 Instructions to run the code

9.1 WordCount on Spark

1. Install Spark 3.2.0 and Maven
2. Create a new directory (e.g. sparkwordcount) which will include the SparkWordCount.scala program
3. Copy-paste the pom.xml file (available in the git repository) in the *sparkwordcount* directory and generate the application jar by running the following command in that same directory:

   ```
   $ mvn package
   ```

   Note: this will generate a file named *sparkwordcount-0.0.1-SNAPSHOT.jar* in a new directory named *target* (e.g. ./sparkwordcount/target)
4. Launch the SparkWordCount.scala script from its directory (/tp2/wordcount) by typing the following command:

   ```
   $ time spark-submit --class SparkWordCount \
   YOUR_LINK/target/sparkwordcount-0.0.1-SNAPSHOT.jar \
   YOUR_LINK/YOUR_FILE.txt
   ```

9.2 PeopleYouMayKnow

The link for the git repo that contains the code is: `https://github.com/ytoubal/SocialMediaProblem`

Running the code with the jar file is the easiest way to run the code without even needing Hadoop configured in the environment. The only requirement is Java to run the jar file and a Linux Distribution.

The jar file is located in the **out/artifacts/PeopleYouMayKnow_jar/** directory and the command to run it is simply:

java -jar PeopleYouMayKnow.jar path/to/input/file path/to/output/directory

10 Conclusion

In conclusion, this study helped us achieve a deeper understanding of Big Data and how to use software such as Hadoop and Spark to solve intricate data problems. We experimented with simple programs such as a WordCount program and used that knowledge to solve a more complex social networking problem (*People You Might Know Algorithm*). We measured the performances of Hadoop and Spark's software and were able to conclude that in fact Spark operates at a much faster pace than Hadoop. The understanding of these softwares is primordial in todays world to achieve optimal performances when dealing with large data sets and complex data problems, At last, we are content with the outcome of this assignment. The next step would be to create an application using Apache Spark and see how it performs.

References

[1] Goran Jevtic. (2020) Hadoop vs Spark – Detailed Comparison. [Online]
Available: `https://phoenixnap.com/kb/hadoop-vs-spark#ftoc-heading-4`

[2] IBM Cloud Education. (2021) Hadoop vs. Spark: What's the Difference? [Online]
Available: `https://www.ibm.com/cloud/blog/hadoop-vs-spark`

[3] Edureka! . (2021) MapReduce Tutorial – Fundamentals of MapReduce [Online]
Available: `https://www.edureka.co/blog/mapreduce-tutorial/`

[4] Cloudera. (2021) Developing and running an Apache Spark WordCount application [Online]
Available: `https://docs.cloudera.com/runtime/7.2.10/developing-spark-applications/topics/spark-develop-run-wordcount-app.html`

[5] The Apache Software Foundation. (2021) Apache Hadoop – MapReduce Tutorial [Online]
Available:`https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`

11 Annex

Link to the git repository:
`https://github.com/ytoubal/MapReduce-with-Hadoop-and-Spark-on-AWS`