Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática

Project 3

Big data with *Hadoop* and *Spark*

Group 3:

André Filipe Bernardes Oliveira, 45648 - Mestrado em Informática (MI)

Report of:

Computação em Nuvem / Cloud Computing

To the teachers:

Mário Calha and Alysson Bessani

2016/2017

1 Index

2 Introduction

With the constant increase of data size, nowadays it is important to have tool to process and analyze such amounts of information (Big Data). There are some frameworks available like Hadoop MapReduce (based on Google MapReduce) and Apache Spark.

In this project, such tools were used to process log files which contained network traffic events. This will allow draw conclusion about the data itself but also about the frameworks used. The main objective of this work is to test the indicated tools and answer a set of questions that might yield insight into this events information meaning.

In the sections, ahead it can be seen the work that was performed to process the given data, the answer to the questions provided as well as an overall evaluation.

It should be noted that later it was decided to ignore the MapReduce implementations due to time constraints to finish this project. Despite that, an example of a MapReduce application is still provided in this work.

3 Data description

The dataset is described in the project description which also includes the location of the dataset. In this project, it was used the *.csv* file provided. It is important to recall the structure of the mentioned file:

```
sIP,dIP,sPort,dPort,protocol,packets,bytes,flags,sTime,duration,eTime,sensor,label
192.168.0.94,192.168.0.200,34618,80,6,55,2989,FSPA,1416846921.828,0.027,1416846921.855,S0,NORMAL
```

As stated in the project description, this traffic logs file includes communication information with the following fields: source IP and port, destination IP and port, protocol, number and size of packets, flags, start time, duration, end time, sensor that acquired the data and a label.

There are some questions that were proposed as well to retrieve relevant information from this events data. These questions were:

1. Who sends more information: total and average per day?
2. With whom each source communicates and how much information is exchanged between pairs (source, destination)?
3. How frequently a pair of machines communicate (flows per day)?
4. Who communicates with more destinations?
5. Who sends and who receives more information?

6. Is anyone acting as a gateway?

Given the question that were proposed some fields seem to be irrelevant (or to provide no additional information), this set includes: flags, duration, sensor and label. On a deeper analysis, it can be seen that the attributes mostly used were: source IP, destination IP, number of packets and size of packets.

4 System set-up

Due to issues with the provided machines (*Quinta Navigators* cluster which included machines t5 and t8) all the development was performed on Eclipse IDE with a local installation of Apache Hadoop and Apache Spark on an Ubuntu virtual machine, while the results were obtained from the *Quinta* machines. This virtual machine was set up with the configurations used in the classes and in the cluster machines and following a tutorial do deploy *Hadoop* and *Apache Spark* in a local mode.

MapReduce programs were written using Java programming language (with default Eclipse IDE, and local Apache Hadoop configuration) while Apache Spark applications were written in Scala, since is the original language the framework was built-on, and because the teachers seemed to prefer that language. The development of the Spark application was also performed on Eclipse IDE using the Scala plugin and a local Apache Spark installation (no cluster) which made the coding process much easier.

All tests that run on my local machine used only a small portion of the dataset instead of the all file. this made the development much easier because it allowed to debug faster and easier, also making possible to visual inspection the test cases. This tests cases had an increasing test data (number of lines): 10, 100, 10000, 20000, 50000, 500000 lines; while the original dataset had a total of 2201439 lines. As previously stated this was done because it was not possible to run the applications for the full dataset on my virtual machine, it also allowed faster debug and application testing and while smaller datasets might be beneficial because of the previous reasons, if the dataset was too small it didn’t provide any answer for some questions (like question 6). Thus, a hybrid approach was needed with an increasing number of lines to allow fast testing of all the questions results.

Despite this, the results presented here only correspond to the results obtained from running the application at the Quinta machines on the full dataset, so that this report can be kept short and avoid redundant information.

The smaller datasets were created with the following Unix Bash command:

head -n number_of_lines > output_file.csv

The *.jar* (***Question1.jar***) for the Hadoop MapReduce application was created with the following commands:

```
$ mkdir InformationSent

$ mv InformationSent.java InformationSent

$ javac -classpath /usr/local/hadoop/share/hadoop/common/hadoop-common-2.8.0.jar:/usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.8.0.jar:/usr/local/hadoop/share/hadoop/hdfs/lib/commons-io-2.4.jar -d InformationSent InformationSent/InformationSent.java

$ jar -cvf Question1.jar -C InformationSent/ .
```

While the *.jar* for the Spark application (*TrafficLogsProcessing.jar*) was created with the appropriate Eclipse tool, but it could be also done in a similar way. These files are made available together with this report.

It was also necessary to specify the commands needed to run the application on *HDFS* filesystem. Below, it can be seen the commands used to run the *Hadoop MapReduce* application (***Question1.jar***), as well as creating a directory, insert the input file and retrieve the results for booth screen (*cat*) and to the local filesystem (*get*).

```
$ sudo hdfs dfs -mkdir /hdfs/input

$ sudo hdfs dfs -put /home/andre/Desktop/Data/project3-dataset.csv /hdfs/input

$ sudo hadoop jar Question1.jar InformationSent /hdfs/input /hdfs/output

$ hdfs dfs -cat /hdfs/output/part-r-00000

$ hdfs dfs -get /hdfs/output/ ./results
```

It should be noted that these commands were run on the virtual machine and not on *Quinta Navigators* machines due to a problem of running them there. So, the results for the *MapReduce* application are not presented in this report due to lack of memory problems (of running the full application in the virtual machine) and since the output should be exactly the same. Similar commands can be used to process the files that output from the *Spark* application.

To run the *Spark* application (***TrafficLogsProcessing.jar***) the following command was used:

```
$ spark-submit --driver-memory 6g --executor-memory 6g --master
local --class
pt.ulisboa.ciencias.di.cloud_computing.project3.TrafficLogsProcessing
./TrafficLogsProcessing.jar /home/cc3/project3-dataset.csv
/home/cc3/results > results.txt
```

This allowed to execute the *Spark Scala* code inside the *.jar* file while increasing the memory allocated to the *JVM*. The input log file is provided in the first parameter (*/home/cc3/project3-dataset.csv*), while the output files are stored in the directory specified by the second parameter (*/home/cc3/results*). The second argument is optional, and if not provided the application will automatically store the files in new folder inside the input file parent directory (this folder is automatically emptied when the application runs). This command also saves the results that are outputted to the default I/O (screen) to a file (*results.txt*).

With all the set up done, we can proceed to analyze the developed code and the results obtained from the *Quinta Navigators* machines.

5 Results

In this section, I examine the code development to shed some understanding into what some chucks of code do. All the source files are made available with this report, so it will only be presented the most relevant ones, thus this will ignore configurations. This will focus on the *Spark* implementation and give an example of a *MapReduce* usage.

After the configurations required, we need to read the input log file. In Spark, we can read a *.csv* file and ignore the header like this:

```
val csv = sc.textFile(inputFile)

val headerAndRows = csv.map(_.split(",").map(_.trim))
val header = headerAndRows.first

val data = headerAndRows.mapPartitionsWithIndex((idx, iter) =>
if (idx == 0) iter.drop(1) else iter)

data.cache()
```

This result was cached to prevent its recomputation by lazy evaluation, since it will be used multiple times (one for each question).

In *MapReduce*, the same can be accomplished by defining the input file:

```
FileInputFormat.addInputPath(job, new Path(args[0]))
```

and by stripping the header and splitting the fields in the map job:

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

            if (!(key.get() == 0 && value.toString()
.equals("sIP,dIP,sPort,dPort,protocol,packets,bytes,flags,sTime,duration,eTime
,sensor"))) {
                    StringTokenizer st = new StringTokenizer(value.toString(),
",");

                    […]
            }
    }
```

In the *Spark* application, some variables were defined to make the code to the questions proposed more human readable.

```
val headerList = header.toList
val sourceIndex = headerList.indexOf("sIP")
val destinationIndex = headerList.indexOf("dIP")
val protocolIndex = headerList.indexOf("protocol")
val packetsIndex = headerList.indexOf("packets")
val bytesIndex = headerList.indexOf("bytes")
val startTimeIndex = headerList.indexOf("sTime")
val endTimeIndex = headerList.indexOf("eTime")
```

Also, since some questions used dates and the results were grouped by day, it was important to convert some fields (start time and end time) in *Unix Timestamp* to a date format. This was accomplished with the following *Scala* functions, which yield a day in the yyyy-MM-dd format:

```
def convertDate(x: Double): DateTime = {
  return new DateTime(x.toLong * 1000).toDateTime
}

def getDay(x: Double): String = {
  return convertDate(x).toString("yyyy-MM-dd")
}
```

These functions were not defined on MapReduce because the example provided didn't used them, but that should be done if such example was provided.

It is also important to finish *Spark* when all the jobs were performed to avoid consumption of resources when it is no longer required:

```
sc.stop()
```

Ahead I will present how each question was tackle with *Spark* and the results that were obtained. Some of these questions were split on two different problems because they required two different *Spark* implementations.

5.1 Question 1

The first question was “Who sends more information: total and average per day?” which was divided in the problems “Who sends more information in total?” and “Who has a higher average information sent per day?”. The first problem was tackled with booth frameworks (*MapReduce* and *Spark*) while the second one was exclusively done in *Spark.* The relevant information to extract is who sends (source IP), amount of information sent (number of bytes) and the day that the event happened (I chose to use start time). Then the processing will group the results and sum information for each group, obtaining the totals or averages required.

5.1.1 Total information sent

The sub-section will solve the problem related to “Who sends more information in total?”. Here only are required the following attributes: source IP and number of bytes.

5.1.1.1 MapReduce

Extending the MapReduce implementation already shown, we need to define a Map class and a Reducer class. The first will split the lines by fields and retrieve the relevant ones (for the question, like source IP and bytes) and the latter will perform some operation on results that are similar (for example sum the number of bytes for a given source IP or calculate the maximum).

Map class to process each line of the file and retrieve the source IP and the number of bytes sent:

```
	public static class Map extends Mapper<LongWritable, Text, Text,
LongWritable> {
		private Text sourceIP = new Text();
		private LongWritable bytesSent = new LongWritable();

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

			if (!(key.get() == 0 && value.toString()
	.equals("sIP,dIP,sPort,dPort,protocol,packets,bytes,flags,sTime,durati
on,eTime,sensor"))) {

				StringTokenizer st = new
StringTokenizer(value.toString(), ",");

				sourceIP.set(st.nextToken());

				for (int i = 0; i < 5; i++) {
					st.nextToken();
				}

				bytesSent.set(Long.parseLong(st.nextToken()));
				context.write(sourceIP, bytesSent);
			}
		}
	}
```

Reduce class to sum all the bytes sent for each IP and then only save (*cleanup*) the result of the largest value:

```
	public static class Reduce extends Reducer<Text, LongWritable, Text,
LongWritable> {
		Long max = Long.MIN_VALUE;
		Text maxWord = new Text();

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

			Long sum = Long.valueOf(0);
			for (LongWritable val : values) {
				sum += val.get();
			}

			if (sum > max) {
				max = sum;
				maxWord.set(key);
			}
		}

		@Override
		protected void cleanup(Context context) throws IOException,
InterruptedException {
			context.write(maxWord, new LongWritable(max));
		}
	}
```

This are the basics of a *MapReduce* application: *Map* jobs to process each line and transform it and *Reduce* jobs to take each this pairs of key-value (in this case being the key the source IP and the value the number of bytes sent) and apply some actions like sum all.

Spark applications use the same core concepts but in a slightly different way as it can be seen in the next sections. The results for this question are shown next to the *Spark* code.

5.1.1.2 Spark

In *Spark*, the same can be done with fewer lines of code. First, we create another RDD

```
      val pairsInfoSent = data.map(line => (line(sourceIndex),
line(bytesIndex).toLong))

      val sumInfoByPair = pairsInfoSent.reduceByKey((a, b) => a + b)

      val maxInfo = sumInfoByPair.max()(new Ordering[Tuple2[String,
Long]]() {
        override def compare(x: (String, Long), y: (String, Long)): Int =
            Ordering[Long].compare(x._2, y._2)
      })

      sc.parallelize(Array(maxInfo)).saveAsTextFile(outputDirectory +
"/Question_1_total/")

      println(s"QUESTION 1 (total) - Higher total information sent from
sIP: ${maxInfo._1} (with ${maxInfo._2} bytes)")
```

with just the source IP as key and the number of bytes as value. Then all the values (number of bytes) of the same key (source IP) are added, creating another RDD. In the end, the tuple with the most information sent is calculated on the executors, to spare driver from doing this work. After all is done the results are output to the screen in the form of a string and save to a file following the *HDFS*/*MapReduce* output structure.

5.1.1.3 Results outputted

The previous Spark code outputted the following to the screen:

```
      QUESTION 1 (total) - Higher total information sent from sIP:
128.3.70.97 (with 10586254399 bytes)
```

Which corresponds to the following tuple that is saved on disk:

```
(128.3.70.97,10586254399)
```

These results are simple and fast to analyze, we can clearly see which is that source IP address that sent the most information.

5.1.2 Average information sent per day

Here the remaining problem will be solved, related to the question: “Who has a higher average information sent per day?”. Therefore, a new attribute will be required: start time.

5.1.2.1 Spark code

The remaining problem is quite similar. However, to aggregate results by day, the key of the tuple needs to be a tuple that contains booth the source IP and the day of the event (in a date format, using getDay function described) which then needs to be transform with keys with only the address. In this case first is calculate the sum of bytes sent per day per source IP, which is then used to calculate the average information sent per day (counting the number of days and adding the total number of bytes for each day). Then everything else proceeds as before: outputting to screen and file.

The main difference of this problem is in the first tuple that is created also contains the date (to sum the values by address and date) and then this value dropped saving only the number of days and the total information by each day.

```
        val pairsInfoSentDay = data.map(line => ((line(sourceIndex),
getDay(line(startTimeIndex).toDouble)), line(bytesIndex).toLong))

        val sumInfoByPairDay = pairsInfoSentDay.reduceByKey((a, b) => a +
b).map(x => (x._1._1, (x._2, 1)))

        val avgInfoByPairDay = sumInfoByPairDay.reduceByKey((x, y) => (x._1 +
y._1, x._2 + y._2)).mapValues(y => 1.0 * y._1 / y._2)

        val maxInfoDay = avgInfoByPairDay.max()(new Ordering[Tuple2[String,
Double]]() {
          override def compare(x: (String, Double), y: (String, Double)): Int =
              Ordering[Double].compare(x._2, y._2)
        })

        sc.parallelize(Array(maxInfoDay)).saveAsTextFile(outputDirectory +
"/Question_1_average_per_day/")

        println(s"QUESTION 1 (average per day) - Higher average information sent
from sIP: ${maxInfoDay._1} (with a average of ${maxInfoDay._2} bytes)")
```

5.1.2.2 Results outputted

In this case, the application outputs:

```
        QUESTION 1 (average per day) - Higher average information sent from
   sIP: 128.3.211.58 (with a average of 2.242225633E9 bytes)
```

Which corresponds to the following tuple that is saved on disk:

```
(128.3.211.58,2.242225633E9)
```

We can see that in the first case, the address that sent the most information was 128.3.70.97 while the address 128.3.211.58 had a constantly higher send rate than the remaining machines.

5.2 Question 2

The next question was “With whom each source communicates and how much information is exchanged between pairs (source, destination)?”. We can see that the most relevant information to extract is the addresses that communicate (source and destination) and the amount of information that is exchanged in each communication (number of bytes). The processing of this query seems trivial as it is only required to define the pairs as the group and then sum for each group.

5.2.1 Spark code

The Spark code is very similar to the first question, but in this case the key is defined as the pair (tuple of the addresses: source and destination) and the value remains the number of bytes. Then the results are added by group and sent to the output channels. As can the seen in the following snippet:

```
        val pairsCommunicateInformation = data.map(line => (
          (line(sourceIndex), line(destinationIndex)),
(line(bytesIndex).toLong)))

        val sumInformationExchanged = pairsCommunicateInformation.reduceByKey((a,
b) => a + b).map(x => ((x._1._1), (x._1._2, x._2)))

        sumInformationExchanged.saveAsTextFile(outputDirectory + "/Question_2/")

        println("QUESTION 2:\n" +
          "(sIP,[(dIP, bytes)])\n" +
          sumInformationExchanged.groupByKey().mapValues(_.map(y => (y._1,
y._2)).mkString("[", ",", "]")).collect().mkString(",\n"))
```

5.2.2 Results outputted

Here we want to know every pair that communicates so it is not a single result as before. Therefore, only a small excerpt of the results is provided. Below the raw tuples can be seen with the source address as key and the value is a list of receivers (destination IP) and the information that was receive. The RDD that was written to file contains a destination IP per line because `groupByKey()` was not applied on the RDD that was save, that could also be

performed but since the purpose of the saving the RDD is to resume the computation where it let, it would be harder to undo the transformations that were made to obtain a human readable string. Example of the results obtained:

```
       QUESTION 2:
       (sIP,[(dIP, bytes)])
       (130.236.95.119,[(131.243.95.168,94117)]),
       (118.147.29.9,[(131.243.85.244,1989)]),
       (128.3.189.253,[(128.3.48.249,14308)]),
       (128.55.7.161,[(128.3.23.244,7728),(128.3.23.51,5592),(128.3.23.185,4416)
,(131.243.91.15,2208)]),
       (204.132.110.95,[(131.243.203.162,115058),(131.243.124.167,29218),(131.24
3.106.146,15662),(131.243.107.192,48030),(131.243.61.27,28046),(131.243.91.106,
60893),(131.243.141.63,47595)]),
       (118.177.31.177,[(131.243.93.235,44034),(131.243.87.214,33358),(131.243.2
19.214,27860)]),
       (146.93.92.64,[(128.3.164.38,66248)]),
       (56.126.37.99,[(128.3.164.249,244),(128.3.164.248,244)]),
       (221.19.25.149,[(128.3.164.248,208)]),
       (117.108.136.35,[(131.243.125.52,40),(131.243.219.96,44),(131.243.234.134
,88)]),
       (221.115.195.230,[(131.243.124.105,7498)]),
       (209.86.82.25,[(131.243.10.189,274)]),
       (210.246.35.104,[(128.3.15.252,96)]),
       (117.40.185.252,[(131.243.85.146,230)]),
       (59.219.148.221,[(128.3.164.249,10770)]),
       (148.180.121.143,[(128.3.26.249,579)]),
       (128.3.19.26,[(128.3.164.194,2341),(131.243.142.214,40),(131.243.140.105,
80),(131.243.219.216,78)]),
       (200.188.221.194,[(131.243.10.189,624)]),
       (207.235.114.179,[(131.243.235.67,3343)]),
       (169.185.8.134,[(128.3.164.249,2481)]),
       (208.202.246.224,[(128.3.164.249,36818)]),
       […]
```

From a total of 31183 lines outputted, we can check with whom each source address communicates and how much information was sent to that destination. Some send information for several destinations while others sent information for a single destination (depending of the number of elements in the list with the following structure: `[(dIP, bytes)]`).

5.3 Question 3

The next question is similar with the last one but instead of amount of information we want to check the number of communications that happened between a pair in one day: “How frequently a pair of machines communicate (flows per day)?”. To answer this question, we need to extract the addresses (source and destination) and the date (that is always be used the start date). All of this will be used to identify each group (key) then is only a matter of counting the number of occurrences of each group.

5.3.1 Spark code

As before a key is defined (in this case, all the relevant information: addresses and date) and is attributed the value 1 for each group. This value will be used to count the number of occurrences of each group (adding all the values for each group). Then the RDD is transformed to calculate the average per day (by removing the day from the key and calculate the average of the values of the reaming groups). In the end, it is outputted to the typical channels:

```
        val pairsCommunicate = data.map(line => ((line(sourceIndex),
line(destinationIndex), getDay(line(startTimeIndex).toDouble)), (1)))

        val pairsCommunicateDay = pairsCommunicate.reduceByKey((a, b) => a +
b).map(x => ((x._1._1, x._1._2), (x._2, 1)))

        val avgPairsCommunicateDay = pairsCommunicateDay.reduceByKey((x, y) =>
(x._1 + y._1, x._2 + y._2)).mapValues(y => 1.0 * y._1 / y._2).map(x =>
((x._1._1, x._1._2), x._2))

        avgPairsCommunicateDay.saveAsTextFile(outputDirectory + "/Question_3/")

        println("QUESTION 3:\n" +
          "((sIP, dIP), flowsPerDay)\n" +
          avgPairsCommunicateDay.collect().mkString(",\n"))
```

5.3.2 Results outputted

As in question 2, only as snippet of the results is display in the report, the full results can be checked in the files that are sent along with this report. Here it also outputs raw tuples to booth channels but in this case, there is no difference between the results (no transformations are applied to make the results more readable as can be checked in the *Spark* code).

```
QUESTION 3:
((sIP, dIP), flowsPerDay)
((218.221.198.85,131.243.142.216),4.0),
((131.243.88.176,204.116.247.253),1.0),
((192.41.221.11,128.3.164.91),2.75),
((128.3.164.249,56.216.37.112),1.0),
((56.170.171.105,131.243.219.11),2.0),
((131.243.106.253,128.3.164.194),2.0),
((131.243.62.217,128.3.164.15),2.6),
((199.62.227.7,128.3.210.134),1.0),
((131.243.91.77,128.3.164.248),1.3333333333333333),
((128.3.7.36,131.243.105.102),3.0),
((131.243.89.144,128.3.97.204),2.0),
((131.243.88.13,204.116.242.22),1.0),
((204.1.74.202,131.243.140.174),1.0),
((131.243.93.56,208.0.11.232),1.0),
((131.243.95.181,131.243.105.180),2.0),
((128.3.164.248,131.243.170.212),2.0),
((128.3.236.72,128.3.164.194),2.0),
((131.243.219.65,131.243.234.132),2.0),
((128.3.191.216,131.243.12.29),2.0),
((128.3.15.84,128.3.209.152),1.0),
[…]
```

In these results, we can see that some pairs only communicate once a day while other do more than that. The key here is booth addresses (source and destination) and the value shows a decimal number with an average number of flows per day. This results contains a total of 223534 (a tenth of the total number of lines).

5.4 Question 4

The list continues with a question with a different approach: “Who communicates with more destinations?”. In this context, the only relevant fields seem to be the addresses (source IP and destination IP). Then is just a matter of counting distinct destination IP for each source IP.

5.4.1 Spark code

This *Spark* implementation is very similar with the first question. The main difference is that here, we count the number of distinct values for each group (done in a similar manner as calculating the average: adding the number of 1’s like in the previous question). After is just calculating the maximum and before and write to the output channels. The code is the following, using similar expressions was the previous implementations:

```
        val pairsDistinctDestinations = data.map(line => (line(sourceIndex),
line(destinationIndex))).distinct().map(x => (x._1, 1))

        val pairsNumberDestinations = pairsDistinctDestinations.reduceByKey((a,
b) => a + b)

        val maxDestinations = pairsNumberDestinations.max()(new
Ordering[Tuple2[String, Int]]() {
          override def compare(x: (String, Int), y: (String, Int)): Int =
              Ordering[Int].compare(x._2, y._2)
        })

        sc.parallelize(Array(maxDestinations)).saveAsTextFile(outputDirectory +
"/Question_4/")

        println(s"QUESTION 4 - The sIP: ${maxDestinations._1}, communicates with
more destinations than the remaining (communicates with ${maxDestinations._2}
destinations)")
```

5.4.2 Results outputted

Unlike the previous two questions, for this one the application output a single result:

```
QUESTION 4 - The sIP: 128.3.164.249, communicates with more
destinations than the remaining (communicates with 3890 destinations)
```

Which corresponds to the following tuple that is saved on disk:

```
(128.3.164.249,3890)
```

We can clearly see how communicates with more destinations (the source IP 128.3.164.249) and how many destinations that source communicates with (3890), the number which granted the status of communicating with more destinations.

5.5 Question 5

The fifth question is a repetition of the first one as it states: “Who sends and who receives more information?”; if we consider total information sent. The question should be split in two problems: “Who sends more information?” and “Who receives more information?”. If we check we can see that the first problem is exactly the first problem of the first question. So, despite there is an implementation for this problem in the source code, and the application outputs a result for this problem, it is really not required. As it should be exactly the same thing as what the first problem outputs. With that in consideration no results are shown here to this problem and the first problem (5.1.1) should be checked. This is due to a problem in interpret the two questions or a mistake from the teachers which didn’t realize that there as a question duplicated.

Despite that the attributes that would be required are: the addresses (source and destination) and the amount of information (number of bytes) as we want the context of sending or receiving information. The remaining problem mentioned before is tackled in the next sub section.

5.5.1 Received More information

Here it is display the solution and results for the problem that remained: “Who receives more information?”. It only requires the destination field and the amount of information received.

5.5.1.1 Spark code

The difference between the two subproblems of this question was what attribute that was considered: in the sent case the attribute was the source IP while here it is the destination IP. Consequently, the solution should be almost identical to the solution of the first problem from the first question (5.1.1) with a slight change on the attribute that was select (while doing the map to create the first RDD).

```
        val pairsInformationReceived = data.map(line => (line(destinationIndex), 
line(bytesIndex).toLong))

        val sumInformationReceived = pairsInformationReceived.reduceByKey((a, b) 
=> a + b)

        val maxInformationReceived = sumInformationReceived.max()(new 
Ordering[Tuple2[String, Long]]() {
          override def compare(x: (String, Long), y: (String, Long)): Int =
              Ordering[Long].compare(x._2, y._2)
        })

        sc.parallelize(Array(maxInformationReceived)).saveAsTextFile(outputDirect
ory + "/Question_5_receives/")

        println(s"QUESTION 5 (receives) - The dIP that received more information 
was: ${maxInformationReceived._1} (with ${maxInformationReceived._2} bytes 
received)")
```

5.5.1.2 Results outputted

The code outputs similar results to the first question:

```
        QUESTION 5 (receives) - The dIP that received more information was: 
   131.243.86.253 (with 9342195135 bytes received)
```

Which corresponds to the following tuple that is saved on disk:

```
(131.243.86.253,9342195135)
```

As stated, with address that receives a greater amount of information is 131.243.86.253. Which is not the same address that sent the most information (as seen in the first question). Despite that, we can see a clear difference in order of magnitude in maximum information sent and received: 10586254399 vs 9342195135. With this it can be conclude that the source IP that sent the most information doesn't send everything to a single address but to several addresses.

5.6 Question 6

The last question is a bit more complex as it does not involve trivial processing of the input files. It goes like "Is anyone acting as a gateway?". If we assume gateway as a device that interconnect 2 different networks and retransmit some packets from one network to the other, we can simply look for the result of this action in the logs. This means, we have to check for an address that receives (destination address) some packets and then resends those packets to somebody else, that is a destination address is also a source address for the same communication.

We also must check if the communication is the same. I assume two communications are equal if they have the same number of bytes, the same number of packets and used the same protocol. The results are further restricted with a time limit for the retransmission to happen; I only considered a retransmitted (and thus it might be a possible gateway) if the new communication event starts less than five minutes after the corresponding communication event has finished.

This set of criteria is used to filter the results in order to find the gateways and to select the attributes that will be necessary. These include: addresses (source and destination), protocol used, number of packets and bytes sent and the time of the events (start and end times).

Given the complexity of this question two alternatives are demonstrated in this report. The first makes use of RDD like was lectured in the classes and uses a join operation. While the second one is a SQL query which is more intuitive to understand (to filtering results with such specific criteria) and was the first implementation that was attempted for this problem so it was left as an alternative solution.

5.6.1 RDD

This section describes the implementation that is based on RDD's and joining of RDD's. First, we need to define a function two check if two communications are a

retransmission of one another. This function takes as argument a row of the joined RDD, where each communication is encoded as a tuple inside the value of the row (value of row contains two tuples, one for each communication), as defined below:

```
      def checkRetransmission(row: (String, ((Int, Int, Long, Double), (Int,
Int, Long, Double)))): Boolean = {
        val (_, value) = row
        val (firstValue, secondValue) = value

        return if (firstValue._1 == secondValue._1 && firstValue._2 ==
secondValue._2 && firstValue._3 == secondValue._3
            && (secondValue._4 - firstValue._4 < 5)) true else false
      }
```

Then two RDD can be created, one with the destination address as key and the remaining relevant information (protocol, number of packer, number of bytes and end of communication) and another RDD with the source address as key and the remaining as value (protocol, number of packer, number of bytes and start of communication). Joining these RDD will create a new RDD in which the key is the same in booth original RDD's and the value will be a tuple with the values of the two original RDD. Until here all it does check if the destination address is the same as the source address in another line. Then it will check if the result lines correspond to retransmissions (filtering results that comply with checkRetransmission function). In the end, the results are outputted to the usual channels. The snippet follows:

```
      val destinationRDD = data.map(line => (line(destinationIndex),
(line(protocolIndex).toInt, line(packetsIndex).toInt, line(bytesIndex).toLong,
line(endTimeIndex).toDouble)))

      val sourceRDD = data.map(line => (line(sourceIndex),
(line(protocolIndex).toInt, line(packetsIndex).toInt, line(bytesIndex).toLong,
line(startTimeIndex).toDouble)))

      val joinRDD = destinationRDD.join(sourceRDD)
      joinRDD.persist(StorageLevel.MEMORY_AND_DISK)

      val resultRDD = joinRDD.filter(x =>
checkRetransmission(x)).map(_._1).distinct()


      resultRDD.saveAsTextFile(outputDirectory + "/Question_6/")

      println("QUESTION 6:\n" +
        resultRDD.collect().mkString(",\n"))
```

It should be that that the result of the join is persisted. Although it was not required, it might help the execution of the code, as I was getting out of memory errors.

5.6.2 Spark SQL

This provides an alternative to achieve the same results. New version of Spark allows to work with data frames which allow the use of SQL queries over the data. This snippet makes use of such tools and demonstrates how to create the data frame, to query over it (with the criteria than we want to meet) and then convert the result to a RDD and output it to the same channels as before. The query seems intuitive (making sure it follows the criteria defined) and leaves plenty of room for optimization.

```
        val dfLogs = sparkSession.read
          .format("csv")
          .option("header", "true")
          .csv(inputFile)

        dfLogs.createOrReplaceTempView("logs")

        val resultDF = sparkSession.sql(
          "SELECT DISTINCT l2.sIP"
              + " FROM logs AS l1, logs AS l2"
              + " WHERE l1.dIP = l2.sIP AND l1.bytes= l2.bytes AND l1.protocol =
l2.protocol AND l1.packets = l2.packets"
              + " AND l2.sTime - l1.eTime < 5")

        val rowsResult: RDD[Row] = resultDF.rdd

        rowsResult.saveAsTextFile(outputDirectory + "/Question_6_alternative/")

        println("QUESTION 6 (alternative):\n" +
          rowsResult.collect().mkString(",\n"))
```

It should be noted that this version seems a bit slower than the alternative, thus it was only used in the smaller datasets and therefore its results are not shown but are expected to be the same as the implementation before.

5.6.3 Results outputted

As in question 2 and 3, this implementation outputs too many lines, hence only a short snippet is shown as demonstration:

```
QUESTION 6:
131.243.15.11,
131.243.103.105,
131.243.88.19,
128.3.189.253,
131.243.92.252,
131.243.60.80,
131.243.86.219,
128.3.204.172,
155.23.171.144,
128.55.231.49,
213.217.245.249,
131.243.126.41,
146.69.138.57,
128.3.190.114,
128.3.209.90,
35.223.112.236,
128.3.204.220,
131.243.10.142,
131.243.85.49,
131.243.219.145,
[…]
```

We can conclude that there are many gateways (total of 6531), this could make sense. Even if there is only one physical gateway, this gateway might have different addresses (like one for each day). This is just a hypothesis since the number of gateways seem too much high.

This concludes the results section where we got a glimpse of what the network looked like in terms of communications habits and devices present.

It is worth to mention that some data types were adjusted by trial and error if they didn't have the precision required to store the values for some cases (for example use Long for number of bytes).

6 Conclusion

With this work, it was possible to try two important frameworks to process *Big Data*. Even though the *Hadoop MapReduce* was not heavily used it was still possible to compare this tool with *Apache Spark*. The latter seems much simple to use and more powerful, allowing

operations than would not be possible on *MapReduce* or would need several sequential runs of algorithm. *Spark* allows a wider range of operations and using simple functions to achieve that. For instances, results for the question 6, to find if there are gateways present in the network would not be possible with *MapReduce* alone.

Despite *Hadoop MapReduce* came to have a minor impact in this project than what was originally intended, I still think is relevant to say that this tool has some limitations when compared to *Spark* (and that is the reason that it was discarded). *Spark* seems to me like a greater framework and more powerful, and even when *MapReduce* can be applied it should not, as it results in more verbose solutions. Therefore, for this kind of processing I would advise *Spark* in every case even because it is more versatile and can be used with more languages.

Also, it worth to mention that this project was much more complex than the remaining. Despite that is was a very good experience and provided an opportunity to learn about some tools that will be important in my future.

Using *Spark* provided an occasion to learn *Scala* which was a good experience as well. This language seems to me like a mix of *Java* and *Python*, using the power of *Java* but everything is as simple as in *Python* or even simpler. Therefore, it was really a great experience and I think made the development of the *Spark* application much easier (with the help of *Eclipse IDE*). However, learning a language always take some time and most of the time spent in the beginning of this project was to learn *Scala*, and then how MapReduce, *HDFS* and *Spark* works.

Another thing that took some time was to spare the work of the driver and develop the application in a way that it parallelizes the most possible. Since the idea of this framework is to parallelize performing some calculations (like the maximum) in the driver would be counterproductive.

This work allowed to reach some conclusions about the network that this log represented, like common communication patterns. However, some questions don’t yield relevant information because their output is still too big to process (questions 2, 3 and 6). Most of the questions should yield an output like question 1, 4 and 5 which is clear to make sense of.

I think that these tools are a great help in Business Intelligence and allowed to discover some relevant information about communications traffic from 3 months of some corporation. But to retrieve more relevant conclusions the questions that are made should be more targeted and specific.

Overall, I think this project was a greater experience because it allowed me to learn so much and gave me tools that might be helpful in the future. Nevertheless, it was a bit complex and hard to do it alone and in such short time and in bad timing, it should have been done with more time and not on the time students are overloaded with work.