High Performance Computational Infrastructures

Federico Alessandro Tullio Riva

1944264

Introduction

When talking about data, White (2015) argues that if over the years datasets and storage capacity have both significantly increased, access speeds have not kept up with the change. The result is a massive amount of data stored but difficult to extract and analyse all at the same time. Traditional Relational database models failed at keeping up with this change and their traditional query languages were not sophisticated enough to read and write the data at the needed velocity and with good resources' allocation.

In order to face these problematics, the Hadoop system was implemented in 2004. Hadoop (2019) is an open-source framework which permits distributed computing for large datasets. Its implementation allows to easily scale up from a single to thousands of machines. The basic idea behind the framework is that, if we share data to multiple servers accessing and processing it in parallel, the time to perform the task would be considerably reduced. Therefore, the aim of the software is to perform data analysis on big sets of data. The core of the framework lies in MapReduce, a software which performs batch query processing, as White (2015) states. Accordingly, it is specifically indicated when the data needs ad-hoc analysis and needs to be accessed multiple times.

Thanks to its characteristics, Hadoop could be particularly indicated to be used in a smart-city environment. In such a context, massive amounts of data are created by different sources (streaming data). The various datasets need to be ingested and analysed through specific queries to create outcomes that can improve the life of each citizen. In addition, such tasks could be performed by clusters of computers, acting as servers, which need to access the data simultaneously.

Our analysis and implementation are based precisely on this close relationship between smart cities environments and Hadoop.

Problem description & associated dataset

Hadoop, as we stated before, is a powerful tool to run ad-hoc analysis on big distributed data. Smart cities could benefit from the use of Hadoop to analyse the massive amounts of data they create. For

our purpose, we chose to investigate the possibilities of Hadoop in the transport field, more precisely taking into consideration the San Francisco bike sharing system. Our analysis aimed at improving the citizens' travel experience. This would happen if not only the transport field's companies could have access to their data, but if also the city, neighbourhood shops and organizations could. This is where the clustering comes in: every organization shares server space in exchange of data and analysis tools. In this environment, Hadoop would be the best framework to make multiple analysis of the dataset in a fast and efficient way.

As we said before, our analysis was run on a database concerning anonymised trip data of the Bay Wheels bike sharing system from Lyft. The dataset, which is approximately 100 MB in size, includes trips from January to May 2019 and it can be considered as a portion of the whole database. One file per month is generated and is presented as a .csv file. As stated in the Lyft (2019) website, each row has the following information: Trip Duration (seconds), Start Time and Date, End Time and Date, Start Station ID, Start Station Name, Start Station Latitude, Start Station Longitude, End Station ID, End Station Name, End Station Latitude, End Station Longitude, Bike ID, User Type (Subscriber or Customer – "Subscriber" = Member or "Customer" = Casual), Member Year of Birth, Member Gender. Unfortunately, the files provided by Lyft have been updated recently and they do not have the last two variables anymore. Kaggle aided, though, since Iñaki (2019) had previously shared the old data, only, without the latitude and longitude of each station. Since for the following analysis we would have needed both the coordinates and the users' information (birth year and gender), a merging of the two databases was performed.

The analysis has two search fields. The first one is purely statistical and is based on the users. This field can be directed only to the bike sharing company in order to gain useful insights about their users. The second field is more complex, since it is based on the trips. It can be used either by Lyft or by a business in the city.

The trips data, as we will see further on, give a clear view of the users' itinerary. For each station reached, it considers indeed how many people have travelled which route.

Design & Implementation

We chose to implement our application using mainly two programming languages: Java and R. With the first one we designed all the Hadoop MapReduce ad-hoc analysis, while the second was useful to realise k-means clustering based on the stations' coordinates and to get insights about the trips.

The Hadoop investigation, as we discussed before, was mainly divided into two sections: one was

focused on the users and the other on the trips. It is important to say that for each key or value passed, when multiple variables were included, the implementation choice was to pass them as a unique Text variable, where the single variables were separated by a comma.

The first class we present is the one that encloses the main() function. Through its structure, we will explain both the arguments it requires and passes to the configuration and, for each possible analysis, the classes invoked by the MapReduce functions.
To run the job the user needs to initialise, alongside the calling of the .jar file, the type of investigation he needs (by typing 1, 2 or 3) and then the input and the output paths. The three possible investigations are, in order: an inverted-index algorithm to match end stations with each start station they relate to, a further analysis on one end station and a statistical report about users. When the first job is requested, the function calls MapReduce on the *EndStationsInvertedIndexMapper* and the *EndStationsInvertedIndexReducer* (this is called also from the combiner function). Both the key and the value outputs are set as Text. The second task serves as a further analysis on the first. When invoked, it prompts the user a station number, an age target and a gender target. Both the age target and the gender target are taken directly from Google Ads services targeting (2020). The sex categories are 'Male', 'Female', 'Other' or 'Unknown' while the age ones are 'under 18', '18-24', '25-34', '35-44', '45-54', '55-64', 'over 65' or 'Unknown'. These new variables are then passed to the configuration and can be accessed from the MapReduce job. Here MapReduce works with three classes*: StationAdvertisingMapper, StationAdvertisingCombiner* and *StationAdvertisingReducer.* The Combiner and the Reducer are differentiated here since the value's output of the mapper and the reducer do not match (first IntWritable and then NullWritable). We will explain this mismatch later in the text. The third type of analysis is a straightforward statistical analysis on the users. It does not request any further arguments, calls MapReduce on the *AverageTripDurationNDistanceMapper* and *AverageTripDurationNDistanceReducer* classes (with the last one running as a combiner as well) and has a Text output for the key and the value.
In the cell below the *BikeSharing* class is shown. In our paper, we will now proceed to explain the classes we have mentioned above. Firstly, we will analyse the ones related to the 3rd job, then the 1st and the 2nd.

```
package org.myorg;

import java.util.Scanner;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
```

```
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class BikeSharing {

	public static void main(String[] args)
			throws Exception
	{
		Configuration conf = new Configuration();

		if(args.length != 4) {
			System.err.println("Usage: BikeSharing <1\\2\\3> <input path> <output
path>");
			System.exit(-1);
		}

		int UT = Integer.parseInt(args[1]);

		if (UT == 2) {

			Scanner scanner = new Scanner(System.in);

			System.out.println("<Station Number>");
			String stationNumber = scanner.next();
			scanner.nextLine();
			conf.set("StationNumber", stationNumber);

			System.out.println("<AgeTarget: -1 (none) \\ 0 (under 18) \\ 1 (18-24) \\ 2
(25-34) \\ 3 (35-44) \\ 4 (45-54) \\ 5 (55-64) \\ 6 (over 65)>");
			String ageTarget = scanner.next();
			scanner.nextLine();
			conf.set("AgeTarget", ageTarget);

			System.out.println("<Male\\Female\\Other\\None>");
			String genderTarget = scanner.next();
			scanner.nextLine();
			conf.set("GenderTarget", genderTarget);

			scanner.close();

		}

		Job job;
		job=Job.getInstance(conf, "Bike Sharing");
		job.setJarByClass(BikeSharing.class);

		FileInputFormat.addInputPath(job, new Path(args[2]));
		FileOutputFormat.setOutputPath(job, new Path(args[3]));

		if (UT == 1) {

			job.setMapperClass(EndStationsInvertedIndexMapper.class);
			job.setReducerClass(EndStationsInvertedIndexReducer.class);
			job.setCombinerClass(EndStationsInvertedIndexReducer.class);

			job.setOutputKeyClass(Text.class);
			job.setOutputValueClass(Text.class);

		} else if (UT == 2){

			job.setMapperClass(StationAdvertisingMapper.class);
			job.setReducerClass(StationAdvertisingReducer.class);
			job.setCombinerClass(StationAdvertisingCombiner.class);

			job.setOutputKeyClass(Text.class);
			job.setOutputValueClass(NullWritable.class);
			job.setMapOutputValueClass(IntWritable.class);

		} else if (UT == 3) {

			job.setMapperClass(AverageTripDurationNDistanceMapper.class);
			job.setReducerClass(AverageTripDurationNDistanceReducer.class);
			job.setCombinerClass(AverageTripDurationNDistanceReducer.class);

			job.setOutputKeyClass(Text.class);
			job.setOutputValueClass(Text.class);
```

```
		} else {
			System.err.println("Usage: BikeSharing <1/2> <input path> <output path>");
			System.exit(-1);
		}

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

}
```

As we already mentioned, in order to run a statistical analysis on the users we divided them into target categories based on their sex and age. The 'Unknown' values were associated to a gender and age pair if no value was found for them. Since in the data either the sex and the age variable were filled or none of them was, our implementation does not check them individually. The idea of splitting the data into targets was used to have a cleaner and more readable outcome. In a preliminary design, in fact, the system was producing a different output for each gender and age pair.

Gender and age targets were used as a MapReduce key. In the Mapper, from each trip, the duration, the distance covered and the user type were extracted and passed as the value. To extract the variable distance, the solution given by George (2013) on Stack Overflow was implemented (passing the arguments related to the height as 0.0). The variable subscriber, instead, was passed as a 0 if the user had been registered as a "Customer", or a 1 if he was labelled "Subscriber". In the Combiner and in the Reducer, instead, the average was calculated for each one of the three values associated to the keys. The code for the *AverageTripDurationNDistanceMapper*, *AverageTripDurationNDistanceReducer* and the *EarthDist* classes are provided in the cells below.

```
package org.myorg;

import java.io.IOException;
import java.util.Calendar;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class AverageTripDurationNDistanceMapper extends Mapper<LongWritable, Text, Text, Text> {

	private Text durationNDistance = new Text();
	private Text user = new Text();

	@Override
	public void map(LongWritable key, Text value, Context context)
			throws IOException, InterruptedException
	{
		String[] line = value.toString().split(",");

		String ageTarget = new String();
		String genderTarget = new String();

		try {

			int userAge = Calendar.getInstance().get(Calendar.YEAR) -
Integer.parseInt(line[13]);
```

```
                        //https://support.google.com/google-ads/answer/2580383?hl=it
                        //Google age targets

                        if (userAge < 18) {
                                ageTarget = "under 18";
                        } else if (userAge >= 18 && userAge < 25) {
                                ageTarget = "18-24";
                        } else if (userAge >= 25 && userAge < 35) {
                                ageTarget = "25-34";
                        } else if (userAge >= 35 && userAge < 45) {
                                ageTarget = "35-44";
                        } else if (userAge >= 45 && userAge < 55) {
                                ageTarget = "45-54";
                        } else if (userAge >= 55 && userAge < 65) {
                                ageTarget = "55-64";
                        } else if (userAge >= 65) {
                                ageTarget = "over 65";
                        }

                        genderTarget = line[14];

                } catch(ArrayIndexOutOfBoundsException | NumberFormatException e) {

                    ageTarget = "Unknown";
                    genderTarget = "Unknown";

                }

                    user.set(genderTarget + "," + ageTarget);

                    double distance =
EarthDist.distance(Double.parseDouble(line[5]),Double.parseDouble(line[9]),Double.parseDouble(line
[6]),Double.parseDouble(line[10]),0.0,0.0);
                    int duration = Integer.parseInt(line[0].trim());
                    if (line[12].trim().equals("Subscriber")) {
                            durationNDistance.set(duration + "," + distance + ",1");
                    } else {
                            durationNDistance.set(duration + "," + distance + ",0");
                    }

                    context.write(user, durationNDistance);

            }
}
```

```
package org.myorg;

import java.io.IOException;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class AverageTripDurationNDistanceReducer extends Reducer<Text, Text, Text, Text> {

        @Override
        public void reduce (Text key, Iterable<Text> values, Context context)
                        throws IOException, InterruptedException
        {
                double sumDuration = 0.0, avgDuration;
                double sumDistance = 0.0, avgDistance;
                double sumSubs = 0.0, subsRatio;
                int count = 0;

                for(Text value : values) {

                        String[] line = value.toString().split(",");

                        sumDuration += Double.parseDouble(line[0]);
                        sumDistance += Double.parseDouble(line[1]);
                        sumSubs += Double.parseDouble(line[2]);
                        count++;
                }

                avgDuration = sumDuration / count;
                avgDistance = sumDistance / count;
                subsRatio = sumSubs / count;
```

```
                context.write(key, new Text(avgDuration + "," + avgDistance + "," + subsRatio));
        }
}
```

```
package org.myorg;

/**
 * Calculate distance between two points in latitude and longitude taking
 * into account height difference. If you are not interested in height
 * difference pass 0.0. Uses Haversine method as its base.
 *
 * lat1, lon1 Start point lat2, lon2 End point el1 Start altitude in meters
 * el2 End altitude in meters
 * @returns Distance in Meters
 *
*/

public class EarthDist {

        public static double distance(double lat1, double lat2, double lon1,
                double lon2, double el1, double el2) {

            final int R = 6371; // Radius of the earth

            double latDistance = Math.toRadians(lat2 - lat1);
            double lonDistance = Math.toRadians(lon2 - lon1);
            double a = Math.sin(latDistance / 2) * Math.sin(latDistance / 2)
                    + Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2))
                    * Math.sin(lonDistance / 2) * Math.sin(lonDistance / 2);
            double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
            double distance = R * c * 1000; // convert to meters

            double height = el1 - el2;

            distance = Math.pow(distance, 2) + Math.pow(height, 2);

            return Math.sqrt(distance);
        }
}
```

As for the trips, instead, the architecture has been further divided into two parts.
The first one was designed to give a broad view of the trips done. Its implementation starts from an Inverted index algorithm baseline with the purpose of giving, for each end station, a list of all the start stations associated with it and the number of times that the trip between the two was performed. There are two peculiarities in this design. The first one is represented by the value passed from the mapper to the reducer and from the reducer to the output. Since our intention was to get a list of the stations, we wanted to find the best possible way to showcase such list. We therefore decided to implement a system in which each value is passed already with peculiar structure which is shown in the textbox alongside. This allowed us to have one or multiple lines indifferently forwarded in a clear structure, which would fill up between the two brackets. The second distinctiveness is found in the Reducer (which by the way is also the Combiner). To have a readable outcome, we needed the list to be shown in a descending order considering the number of trips. This collided with MapReduce sorting system, so we had to force a custom strategy at the end of each values iteration.

```
{
stationID, stationName, numTrips
}
```

The implementations for the *EndStationsInvertedIndexMapper* and the *EndStationsInvertedIndexReducer* are showcased in the cells below.

```
package org.myorg;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class EndStationsInvertedIndexMapper extends Mapper<LongWritable, Text, Text, Text>
{
	private Text startStationID = new Text();
	private Text endStationID = new Text();

    @Override
	protected void map(LongWritable key, Text value, Context context)
			throws java.io.IOException, InterruptedException
	{
	String[] line = value.toString().split(",");
		startStationID.set(line[3] + "," + line[4]);
		endStationID.set(line[7] + "," + line[8]);

		context.write(endStationID, new Text ("{\n" + startStationID.toString() +
",1\n}"));

	}
}
```

```
package org.myorg;

import java.util.ArrayList;
import java.util.Comparator;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class EndStationsInvertedIndexReducer extends Reducer<Text, Text, Text, Text>
{
	protected void reduce(Text key, Iterable<Text> values, Context context)
		throws java.io.IOException, InterruptedException
	{
		ArrayList<String> bufferStations = new ArrayList<String>();

		for (Text value : values) {

			String[] splitValue = value.toString().split("\n");

			if (splitValue[0].equals("{")) {

				int i = 1;

				while (!(splitValue[i].equals("}"))) {

					String[] stationInfo = splitValue[i].split(",");

					//0 sID, 1 sName, 2 sNum

					int index = -1;
					for (int j = 0; j < bufferStations.size(); j++) {
						if (bufferStations.get(j).startsWith(stationInfo[0] +
"," + stationInfo[1])) {
							index = j;
							j = bufferStations.size();
						}
					}

					if (index == -1) {
						bufferStations.add(splitValue[i].toString());
					} else {
						stationInfo =
bufferStations.remove(index).split(",");
						int updateValue = Integer.parseInt(stationInfo[2]) +
1;
```

```
                                        String newValue = stationInfo[0] + "," +
stationInfo[1] + "," + updateValue;
                                        bufferStations.add(index, newValue);
                                    }

                                    i++;
                                }
                            }

                }

                bufferStations.sort(new Comparator<String>() {
                        @Override
                        public int compare(String str1, String str2) {
                                String[] splitStr1 = str1.split(",");
                                String[] splitStr2 = str2.split(",");
                                return Integer.parseInt(splitStr2[2]) -
Integer.parseInt(splitStr1[2]);
                        }
                });

                StringBuffer buffer = new StringBuffer();
                buffer.append("{\n");
                for (String station : bufferStations) {
                        buffer.append(station.toString());
                        buffer.append("\n");
                }
                buffer.append("}");

                Text documentList = new Text();
                documentList.set(buffer.toString());
                context.write(key, documentList);
        }
}
```

The second section of the trips' investigation aimed at gaining insights about one single end station and a precise target of people that were reaching it. As previously stated, the Mapper retrieves information about the asked end station, the age and the gender targets. It then passes as a key a Text value containing comma separated information about the start station related to the requested end station (ID, Name, Latitude, Longitude) and then the age and gender target. As a value it just writes 1. The combiner and the reducer, then, work almost like a Word Count algorithm. The singularity is given by the fact that, since we wanted to reuse the MapReduce output as you would do for a .csv file, we set the reducer to change the value output to NullWritable and to incorporate in the keys the count of the trips as a last argument. We show the three classes in the cell below.

```
package org.myorg;

import java.io.IOException;
import java.util.Calendar;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class StationAdvertisingMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
```

```
        private Text startStationInfo = new Text();
        private final static IntWritable one = new IntWritable(1);

        @Override
        public void map(LongWritable key, Text value, Context context)
                        throws IOException, InterruptedException
        {
                Configuration conf = context.getConfiguration();
                Pattern patternStationNumber = Pattern.compile(conf.get("StationNumber"));
                String confAgeTarget = conf.get("AgeTarget");
                String confGenderTarget = conf.get("GenderTarget");

                String[] line = value.toString().split(",");
                String ID = line[7];

                Matcher matcherStationNumber = patternStationNumber.matcher(ID);

                if (matcherStationNumber.find()) {

                        String ageTarget = "-1", genderTarget;

                        if (!confGenderTarget.equals("None")) {
                                try {

                                        if (line[14].equals("Male")) {
                                                genderTarget = "Male";
                                        } else if (line[14].equals("Female")) {
                                                genderTarget = "Female";
                                        } else {
                                                genderTarget = "Other";
                                        }

                                } catch(ArrayIndexOutOfBoundsException | NumberFormatException e) {
                                        genderTarget = "None";
                                }
                        } else {
                                genderTarget = "None";
                        }

                        if (!confAgeTarget.equals("-1")) {
                                try {

                                        int userAge = Calendar.getInstance().get(Calendar.YEAR) -
Integer.parseInt(line[13]);

                                        if (userAge < 18) {
                                                ageTarget = "0";
                                        } else if (userAge >= 18 && userAge < 25) {
                                                ageTarget = "1";
                                        } else if (userAge >= 25 && userAge < 35) {
                                                ageTarget = "2";
                                        } else if (userAge >= 35 && userAge < 45) {
                                                ageTarget = "3";
                                        } else if (userAge >= 45 && userAge < 55) {
                                                ageTarget = "4";
                                        } else if (userAge >= 55 && userAge < 65) {
                                                ageTarget = "5";
                                        } else if (userAge >= 65) {
                                                ageTarget = "6";
                                        }

                                } catch(ArrayIndexOutOfBoundsException | NumberFormatException e) {
                                        ageTarget = "-1";
                                }
                        } else {
                                ageTarget = "-1";
                        }

                        if (confGenderTarget.equals(genderTarget) &&
confAgeTarget.equals(ageTarget)) {

                                String startStationString = line[3] + "," + line[4] + "," + line[5]
+ "," + line[6] + "," + ageTarget + "," + genderTarget;
                                startStationInfo.set(startStationString);

                                context.write(startStationInfo,one);
                        }
                }
```

```
        }
}
```

```
package org.myorg;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class StationAdvertisingCombiner extends Reducer<Text, IntWritable, Text, IntWritable> {

        @Override
        public void reduce (Text key, Iterable<IntWritable> values, Context context)
                        throws IOException, InterruptedException
        {
                int sum = 0;
                for (IntWritable value : values) {
                        sum += value.get();
                }
                context.write(key, new IntWritable(sum));
        }
}
```

```
package org.myorg;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class StationAdvertisingReducer extends Reducer<Text, IntWritable, Text, NullWritable> {

        @Override
        public void reduce (Text key, Iterable<IntWritable> values, Context context)
                        throws IOException, InterruptedException
        {
                int sum = 0;
                for (IntWritable value : values) {
                        sum += value.get();
                }

                String stationCompleteData = key.toString() + "," + sum;

                context.write(new Text(stationCompleteData), NullWritable.get());
        }
}
```

As we already mentioned, we wanted to make a further analysis with the output from the last MapReduce job. By saving the .csv-like file that we created from the Hadoop output, we used R to process it. Just after saving the data, we manually added the header on top of it, in order to make the R code more readable. Our intention was to find, in the city of San Francisco, which borough (found by the clustering algorithm) had the greatest number of trips, performed by the requested target, to the selected end station. For the implementation of the algorithm we relied on two sources: from Anand (2017) we understood the elbow method, while from Jaiswal (2018) we learnt how to perform the k-means clustering algorithm as we needed it. To show the results, we plotted the coordinates of the stations on a San Francisco map using the RgoogleMaps library. A snippet of the code we implemented can be seen in the cell below.

```
baywheels <- read.table("baywheels-station", header = TRUE, sep = ",")


set.seed(20)
k.max <- 15
wss <- sapply(1:k.max,function(k){kmeans(baywheels[,3:4], k,)$tot.withinss})
wss
plot(1:k.max, wss,type="b", pch = 19, frame = FALSE, xlab="Number of clusters K",ylab="Total
within-clusters sum of squares")

clusters <- kmeans(baywheels[,3:4],2)
str(clusters)
baywheels$Boroughs <- as.factor(clusters$cluster)
col <- as.numeric(baywheels$Boroughs)
table(baywheels$Boroughs)

library(RgoogleMaps)
par(pty="s")
SFMap <- plotmap("San Francisco", zoom = 11)
plotmap(lat = St_lat, lon = St_lon, col = col, data = baywheels)
```

Results

The results of the three jobs are very different from each other, but they give a broad idea of the possible applications and of the expected outputs that Hadoop processes can give. Before delving into the specific outcomes, we should mention that, even if very different from one another, they embody the principles of Hadoop. As we mentioned before, they are totally scalable and they can apply to many similar problematics. Even if such dataset could be analysed with a traditional methodology, these tasks would be difficult to reproduce with satisfactory results. The three implemented tasks, in fact, ingest and aggregate large chunks of data in a very specific way. When examining the results, we will follow the same pattern chosen for the design and implementation

```
Female,18-24    947.0882808466702,1562.6022210308945,0.7893649974186887
Female,25-34    842.8233526608606,2036.9261653325786,0.855012245983865
Female,35-44    777.0506016765819,2203.5490880300335,0.880881557598702
Female,45-54    882.1276413765345,2371.9837523092997,0.8683336687462266
Female,55-64    956.6460401053271,1755.4401104794,0.9099655661332793
Female,over 65    820.8176337603617,1398.777490565746,0.9035418236623964
Male,18-24    734.3789041320956,1627.3083649361452,0.8583750924176456
Male,25-34    700.5290615069557,1933.1556154447997,0.8919002895275757
Male,35-44    701.8121308507755,2038.9164796602167,0.8983436545964991
Male,45-54    746.9841244090109,2012.0791951666954,0.9075159914712153
Male,55-64    744.6352131494693,1584.9580656105518,0.9240426551037386
Male,over 65    723.3162022945446,1494.8364879595729,0.9340903769608991
Other,18-24    991.5575279421433,1532.5977840406108,0.8704799474030244
Other,25-34    958.0281213383674,1897.1316014635306,0.8229397213904439
Other,35-44    956.0039015606243,1811.4680280430694,0.852641056422569
Other,45-54    933.767578125,1602.6281731116992,0.90185546875
Other,55-64    1061.103716508211,1829.586707626876,0.9403630077787382
Other,over 65    1011.8923076923077,1622.8900558591283,0.8492307692307692
Unknown,Unknown    1416.8756075826313,1719.547276847304,0.45886665586519765
```

section: we begin with the third task, then we look at the first and the last will be the second with the related R analysis.

The results given by the statistical analysis on the users are very simple yet very peculiar. By having used the age and sex targets, we have created a fixed set of 22 possible results. This indicates that no matter how big the database is, the outcome will be simple and easy to read and analyse: from an input of nearly 203 MB, the output weighted less than 1.3 KB. MapReduce took just over 1 second to perform the task if the combiner was turned on and a little less than 2 seconds if it was not. The output, shown in the textbox above, were ready to give a simple but clear view of the bike sharing's usage. The younger the user, the higher the chance he is only a customer; from 25 to 54 years old people, the distance covered is higher both in men and women but the trip times are lower; females and others spend significantly more time on the bikes than men… All this data, even if it leads only to a surface analysis, could be used to implement ways to gain more subscribers among the youngsters, make bikes more comfortable for long usage or other improvements that the Lyft bike sharing could perform.

A very different set of data is the one produced by the first task. The output comes in the form of a long list of ordered objects, as if the dataset created a whole new data structure. Here, though, if the input is always the same, the output weights 1.2 MB and it could obviously increase based on the data ingested by the Mapper and Reducer. The working times, instead, have not increased, being around 1 second by using the combiner and less than 2 seconds without it. An interesting usage of such results could be to adopt them to display a Bayesian network with San Francisco bikers preferred routes. It is therefore clear that this data could have an impact on the whole city: the bike sharing company could use it to understand where to place a new station, but the city council could look at the result and decide where to place new bike lanes. In either way the citizens would benefit from it. Endless results like this one could be used, only if the relative datasets are shared and made of public domain. Furthermore, the fact that the dataset has a precise schema and that it is already ordered in a useful way, could easily lead to the implementation of the above display or other simpler investigations. Such designs would remain

```
99,Folsom St at 15th St    {
89,Division St at Potrero Ave,28
112,Harrison St at 17th St,23
58,Market St at 10th St,21
223,16th St Mission BART Station 2,19
16,Steuart St at Market St,17
125,20th St at Bryant St,17
110,17th & Folsom Street Park (17th St at Folsom St),16
123,Folsom St at 19th St,16
15,San Francisco Ferry Building (Harry Bridges Plaza),14
139,Garfield Square (25th St at Harrison St),14
…
}
```

suitable for the Hadoop environment. Since the whole data is too long to display here, we will copy in the textbox alongside just a portion of it, containing one station and the first ten stations from the ordered list of start stations related to it.

```
10,Washington St at Kearny St,37.7953929373,-122.4047702551,2,Male,21
100,Bryant St at 15th St,37.7671004,-122.410662,2,Male,6
101,15th St at Potrero Ave,37.7670785046,-122.4073585868,2,Male,28
104,4th St at 16th St,37.7670445797,-122.390833497,2,Male,4
105,16th St at Prosper St,37.764285,-122.4318042,2,Male,12
106,Sanchez St at 17th St,37.7632417,-122.4306746,2,Male,1
108,16th St Mission BART,37.7647100858,-122.4199569225,2,Male,4
109,17th St at Valencia St,37.7633158,-122.4219039,2,Male,11
11,Davis St at Jackson St,37.79728,-122.398436,2,Male,14
110,17th & Folsom Street Park (17th St at Folsom St),37.7637085,-122.4152042,2,Male,36
```

The results for the last task require great attention in their analysis. They were made to create an ad-hoc subset from the data, onto which further investigations could be made. The output is a dataset in the .csv style, which is very suitable for further investigations using all kinds of technologies. We could not create an actual .csv file, but, if needed, it would be quite easy to do it with R or other tools, since the format is already set. This task is particularly suited for Hadoop, since, here as above, the problem could be related to a much larger dataset of trips, but the outcome would be limited by the number of available stations. This would be the first substantial reduction of the data that Hadoop would handle. The second reduction is determined by the fact that the user can choose the exact needed target. In the textbox above, we reported the first ten values obtained by running the second tasks with station number set as 99, age target 2 (25 to 34) and gender target male. The process took approximately 1 second to run (both with and without the combiner) and wrote 11.4 KB of data. This data was given a header for comfort, as already mentioned, and saved into R. By running the R code shown above, the first clustering results shown that the optimal k-value was 2. This was due to the fact that there were bikers who came to San Francisco (station 99 is in the city centre), all the way from the stations in San Jose. This was surprising, but what we wanted to explore was the usage's possibility of this tool by neighbourhoods' shops to make targeted ads in other areas of the city (from where their customers were coming). This could prove Hadoop's and related tools to be important not only to big data providers, but also

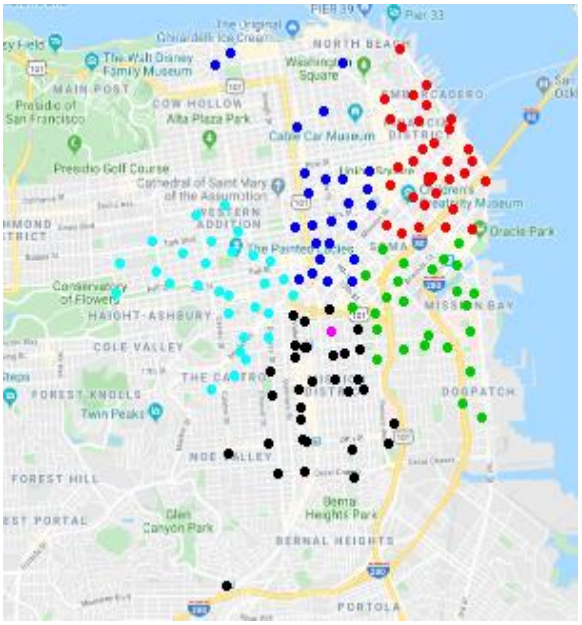

Figure 1: San Francisco's station 99 cluster map

for small companies. We therefore isolated the San Francisco stations, by creating a subset of the baywheels table, and we run the program again. The new values were more satisfactory if compared to the others: neighbourhoods were set out nicely (the optimal k-value was a number between 4 and 6 and we tried them all) and even almost matched real San Francisco's ones, as it can be seen from the map in Figure 1. The map shows in purple the chosen station, while the blue dots represent the station's cluster in which there are the most people matching the target.

Conclusions

The aim of this paper was to investigate a possible real-life use of the Hadoop framework. A smart city environment was taken into consideration, thanks to its characteristic of creating large amounts of streaming data. In addition to this, running cluster computing in such environment could bring many interesting opportunities. As we have seen, with Hadoop, ad-hoc analysis on the datasets could be shared to bring wealth and improvements for citizens. Hadoop outputs, if well prepared, could also be used with other analysis tools. Therefore, we should not think about it as a closed environment suitable just for one task (digesting large amounts of data), but as a dynamic tool which should be taken as a starting point to create many custom data analysis opportunities.
In our case, we have seen how powerful Hadoop really is, even if the dataset at our disposal was not very large. The scale up possibilities, though, by maintaining the same design and implementation, are many. We demonstrated how each result could be successfully used to gain and aggregate useful information, which could even be furtherly investigated through a simple yet effective R implementation.

References

Lyft, I. (2019). *Bike share in the San Francisco Bay Area | Bay Wheels | Lyft*. [online] Lyft. Available at: https://www.lyft.com/bikes/bay-wheels [Accessed 20 Dec. 2019].

Iñaki (2019). *Bay Area Bike Sharing Trips*. [online] Kaggle.com. Available at: https://www.kaggle.com/jolasa/bay-area-bike-sharing-trips [Accessed 20 Dec. 2019].

Hadoop.apache.org. (2019). *Apache Hadoop*. [online] Available at: https://hadoop.apache.org/ [Accessed 28 Dec. 2019].

White, T. (2015), *Hadoop: The Definitive Guide*, O'Reilly Media, Available from: ProQuest Ebook Central. [28 Dec. 2019].

Support.google.com. (2020). *About demographic targeting - Google Ads Help*. [online] Available at: https://support.google.com/google-ads/answer/2580383?hl=en-GB [Accessed 2 Jan. 2020].

George, D. (2013). *Calculating distance between two points, using latitude longitude?*. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/3694380/calculating-distance-between-two-points-using-latitude-longitude [Accessed 23 Dec. 2019].

Anand, S. (2017). *Finding Optimal Number of Clusters*. [online] R-bloggers. Available at: https://www.r-bloggers.com/finding-optimal-number-of-clusters/ [Accessed 2 Jan. 2020].

Jaiswal, S. (2018). *K-Means Clustering in R Tutorial*. [online] DataCamp Community. Available at: https://www.datacamp.com/community/tutorials/k-means-clustering-r [Accessed 2 Jan. 2020].