University of Salerno

.DIEM

Department of Information and Electrical Engineering and Applied Mathematics

Master Degree in Computer Science

High Performance Computer: Big Data

A.A. 2023/2024

Student:
Pepe Lorenzo, mat.: 0622702121

Professor
Giuseppe D’Aniello

Index

Introduction

This report summaries the work done for the examination of “Big Data”, part of the “High Performance Computing” ,held by prof. D’Aniello Giuseppe, for the master’s degree course in Computer Science of the Department of Information and Electrical Engineering and Applied Mathematics, in the academic year 2023/2024.

The project will be developed in Java, using IntelliJ IDEA IDE and Docker Desktop.

1. Project introduction

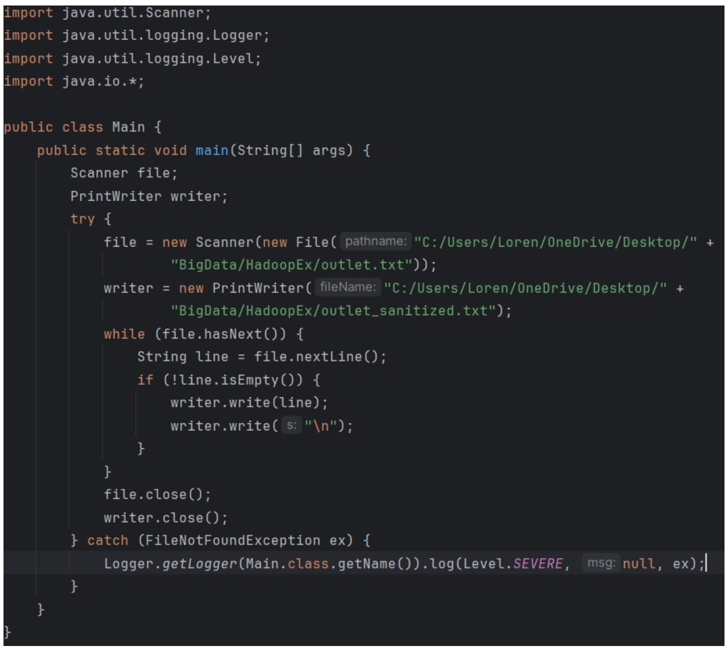
The project is divided in two parts:

- Hadoop: analysis of a dataset with metrics chosen by the student;
- Spark: analysis of the same dataset with metrics chosen by the professor.

The dataset contains data on customers and marketing initiatives and is composed by the following fields:

- **Item Identifier**: unique product ID;
- **Item Weight**: weight of product;
- **Item Fat Content**: whether the product is low fat or not;
- **Item Visibility**: the percentage of the total display area of all products in a store allocated to the product;
- **Item Type**; the category to which the product belongs;
- **Item MRP**: Maximum Retail Price (list price) of the product;
- **Outlet Identifier**: unique store ID;
- **Outlet Establishment Year**: the year in which the store was established;
- **Outlet Size**: the size of the store in terms of ground area covered;
- **Outlet Location Type**: the type of city in which the store is located;
- ***Outlet Type**: whether the outlet is just a grocery store or some sort of supermarket;
- **Item Outlet Sales**: sales of the product in the store.

The Hadoop project will analyze the dataset and extract the Top-K outlet with the most Low-Fat product sold. The Spark project, assigned by the professor, will find the 3 most expensive shops for every dimension, sorting the result in an ascending manner.
The dataset has been first sanitized, eliminating the field header line manually and then using a simple Java code, visible in Figure 1, to delete the empty lines.



```java
import java.util.Scanner;
import java.util.logging.Logger;
import java.util.logging.Level;
import java.io.*;

public class Main {
    public static void main(String[] args) {
        Scanner file;
        PrintWriter writer;
        try {
            file = new Scanner(new File( pathname: "C:/Users/Loren/OneDrive/Desktop/" +
                    "BigData/HadoopEx/outlet.txt"));
            writer = new PrintWriter( fileName: "C:/Users/Loren/OneDrive/Desktop/" +
                    "BigData/HadoopEx/outlet_sanitized.txt");
            while (file.hasNext()) {
                String line = file.nextLine();
                if (!line.isEmpty()) {
                    writer.write(line);
                    writer.write( s: "\n");
                }
            }
            file.close();
            writer.close();
        } catch (FileNotFoundException ex) {
            Logger.getLogger(Main.class.getName()).log(Level.SEVERE,  msg: null, ex);
        }
    }
}
```

Figure 1 - Java program used to remove the empty lines

2. Setup phase

2.1 Docker and Hadoop setup

Before starting with the project, a setup phase for docker is needed. We need to create a Hadoop container to use when our java code is ready to use. The setup phase is generally easy and newer version of Docker Desktop implements a Windows PowerShell Console directly inside the interface.
Before creating anything, we need the Hadoop cluster file folder containing the Hadoop code; in this project the version 3.3.6 has been used. Once docker is loaded, using the "cd" command, we must move inside the Hadoop Cluster folder. Once inside the folder, we can create first the network through Hadoop and docker, and then the master and slave process. We use the following commands:

- **Bridge/network creation**: "*docker network create --driver bridge hadoop_network*";
- **Master process creation**: "*docker run -t -i -p 9870:9870 -d --network=hadoop_network --name=master hadoop-new*";
- **Slaves process creation**: "*docker run -t -i -p 9862:9864 -d --network=hadoop_network --name=slaveX hadoop-new*";

We need a total of three slaves, and we can use any name for them. For simplicity we called them slave1, slave2 and slave3.
Now we can create the Docker container and the image for Hadoop, using the following commands:

- "*docker compose up -d*";
- "*docker container exec -ti master bash*".

It is now the time to initialize the Hadoop environment, with the commands:

- "*hdfs namenode -format*";
- "*$HADOOP_HOME/sbin/start-dfs.sh*";
- "*$HADOOP_HOME/sbin/start-yarn.sh*".

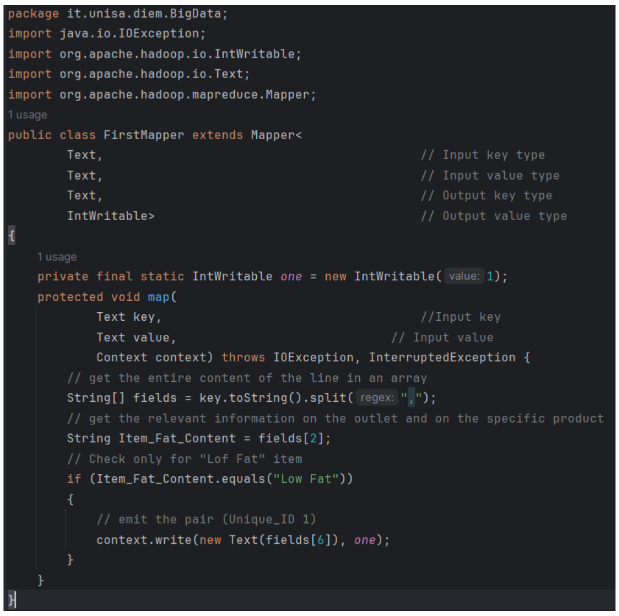
Using the basic bash commands "*ls*" we can check if the "*data*" folder is present: this is the folder inside of which we will paste the dataset file and the JAR file, used to run the Java program.
The folder should be present and the setup phase for Hadoop and Docker is completed.
All the above commands can be launched in a classic Windows PowerShell; the docker PowerShell is not mandatory.

3. Hadoop project

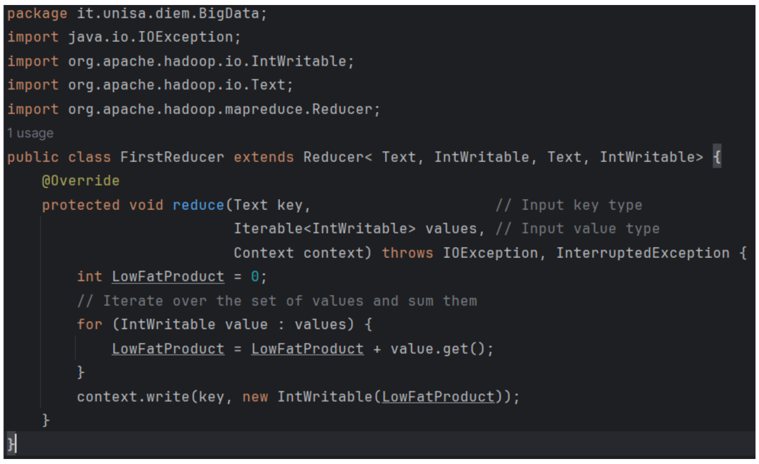
The Hadoop project analyze the dataset and extract the Top-K outlet with the most Low-Fat product sold, in a descending manner.

3.1 Java program

The Java program, consist of a job chain with two mappers and two reducers, a driver, a comparator class and a class containing various functions for the getting/setter and the function "toString" for the final string to write in the final output .txt file.
The first mapper simply "selects" all the outlet that sell low fat product, while the first reducer aggregates this result so there will be only the pair key-value with the key being the outlet ID (unique for every store) and the number of Low-Fat products that store sells.
The code for the mapper and the reducer can be seen in Figure 2.



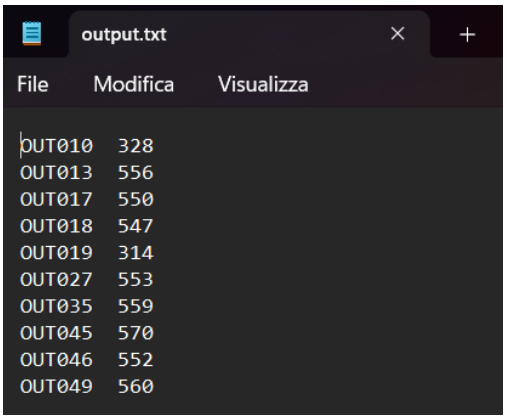
```java
package it.unisa.diem.BigData;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
1 usage
public class FirstMapper extends Mapper<
        Text,                                          // Input key type
        Text,                                          // Input value type
        Text,                                          // Output key type
        IntWritable>                                   // Output value type
{
    1 usage
    private final static IntWritable one = new IntWritable( value: 1);
    protected void map(
            Text key,                                  //Input key
            Text value,                            // Input value
            Context context) throws IOException, InterruptedException {
        // get the entire content of the line in an array
        String[] fields = key.toString().split( regex: ",");
        // get the relevant information on the outlet and on the specific product
        String Item_Fat_Content = fields[2];
        // Check only for "Lof Fat" item
        if (Item_Fat_Content.equals("Low Fat"))
        {
            // emit the pair (Unique_ID 1)
            context.write(new Text(fields[6]), one);
        }
    }
}
```



```java
package it.unisa.diem.BigData;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
1 usage
public class FirstReducer extends Reducer< Text, IntWritable, Text, IntWritable> {
    @Override
    protected void reduce(Text key,                  // Input key type
                          Iterable<IntWritable> values, // Input value type
                          Context context) throws IOException, InterruptedException {
        int LowFatProduct = 0;
        // Iterate over the set of values and sum them
        for (IntWritable value : values) {
            LowFatProduct = LowFatProduct + value.get();
        }
        context.write(key, new IntWritable(LowFatProduct));
    }
}
```

Figure 2 - Left: FirstMapper class, Right: FirstReducer Class

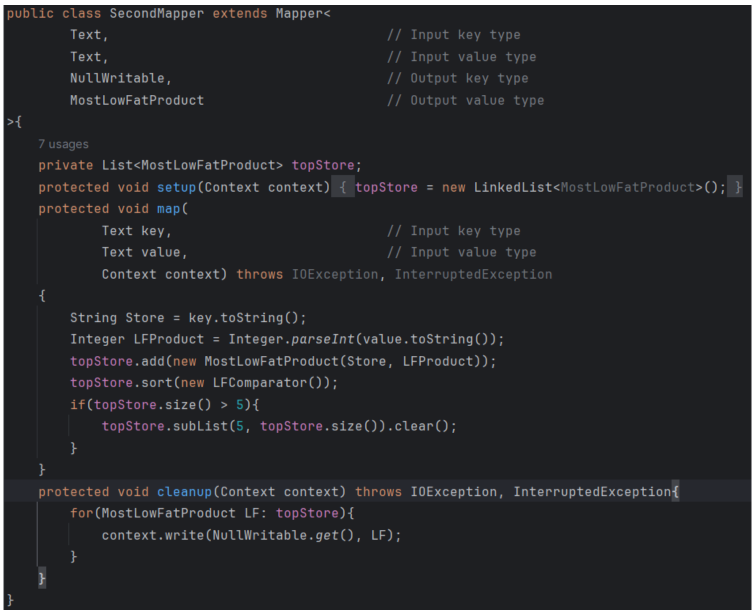
The output is saved in the first output file called "output.txt" and it's visible in Figure 3. As can be seen the file it's composed of just 10 lines, with the pairs key-values already discussed, in a mixed order.



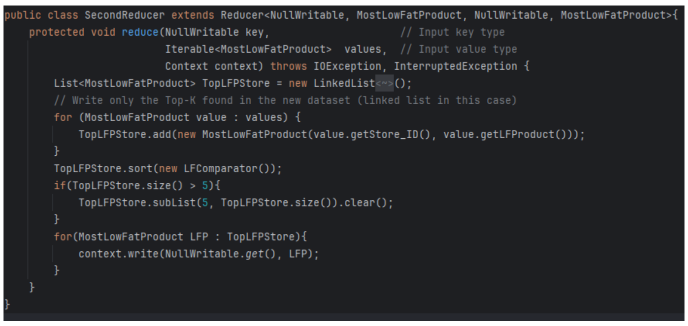
```
OUT010   328
OUT013   556
OUT017   550
OUT018   547
OUT019   314
OUT027   553
OUT035   559
OUT045   570
OUT046   552
OUT049   560
```

Figure 3 - Intermediate output, generated from the first Job

The second mapper take as input the output file just created and create a linked list that will contain the Top-K store that most sell Low-Fat product (for simplicity, in this case K = 5). The mapper simply reads store names and their respective low-fat product counts while keeping track of the top 5 stores, sorting them in a descending order. In the last function, it writes in the linked list the top 5 store found. This mapper makes use of the already mentioned classes "LFComparator" and "MostLowFatProduct". The reducer takes all the stores and their respective low-fat product counts across all the mappers, sorts them, and outputs only the top 5 stores with the most low-fat products. The reducer ensures that only the top 5 stores are written as the result. Both of these classes can be seen in Figure 4.



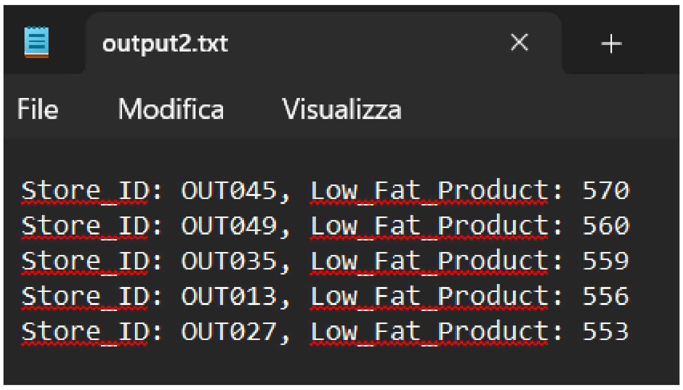
```java
public class SecondMapper extends Mapper<
        Text,                                   // Input key type
        Text,                                   // Input value type
        NullWritable,                           // Output key type
        MostLowFatProduct                       // Output value type
>{
    7 usages
    private List<MostLowFatProduct> topStore;
    protected void setup(Context context) { topStore = new LinkedList<MostLowFatProduct>(); }
    protected void map(
            Text key,                           // Input key type
            Text value,                         // Input value type
            Context context) throws IOException, InterruptedException
    {
        String Store = key.toString();
        Integer LFProduct = Integer.parseInt(value.toString());
        topStore.add(new MostLowFatProduct(Store, LFProduct));
        topStore.sort(new LFComparator());
        if(topStore.size() > 5){
            topStore.subList(5, topStore.size()).clear();
        }
    }
    protected void cleanup(Context context) throws IOException, InterruptedException{
        for(MostLowFatProduct LF: topStore){
            context.write(NullWritable.get(), LF);
        }
    }
}
```



```java
public class SecondReducer extends Reducer<NullWritable, MostLowFatProduct, NullWritable, MostLowFatProduct>{
    protected void reduce(NullWritable key,                     // Input key type
                          Iterable<MostLowFatProduct>  values,  // Input value type
                          Context context) throws IOException, InterruptedException {
        List<MostLowFatProduct> TopLFPStore = new LinkedList<~>();
        // Write only the Top-K found in the new dataset (linked list in this case)
        for (MostLowFatProduct value : values) {
            TopLFPStore.add(new MostLowFatProduct(value.getStore_ID(), value.getLFProduct()));
        }
        TopLFPStore.sort(new LFComparator());
        if(TopLFPStore.size() > 5){
            TopLFPStore.subList(5, TopLFPStore.size()).clear();
        }
        for(MostLowFatProduct LFP : TopLFPStore){
            context.write(NullWritable.get(), LFP);
        }
    }
}
```

Figure 4 - Left: SecondMapper Class, Right: SecondReducer Class

The output is visible in the next Figure 5, saved in the file "output2.txt".

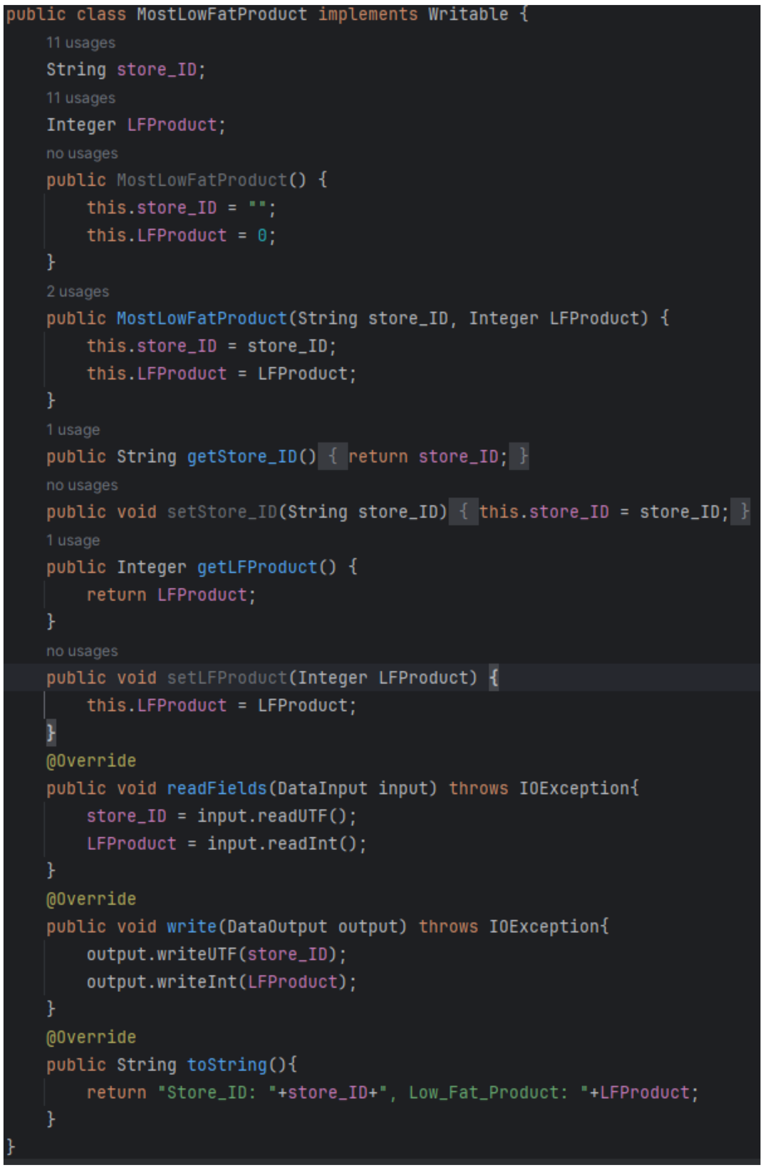


```
Store_ID: OUT045, Low_Fat_Product: 570
Store_ID: OUT049, Low_Fat_Product: 560
Store_ID: OUT035, Low_Fat_Product: 559
Store_ID: OUT013, Low_Fat_Product: 556
Store_ID: OUT027, Low_Fat_Product: 553
```

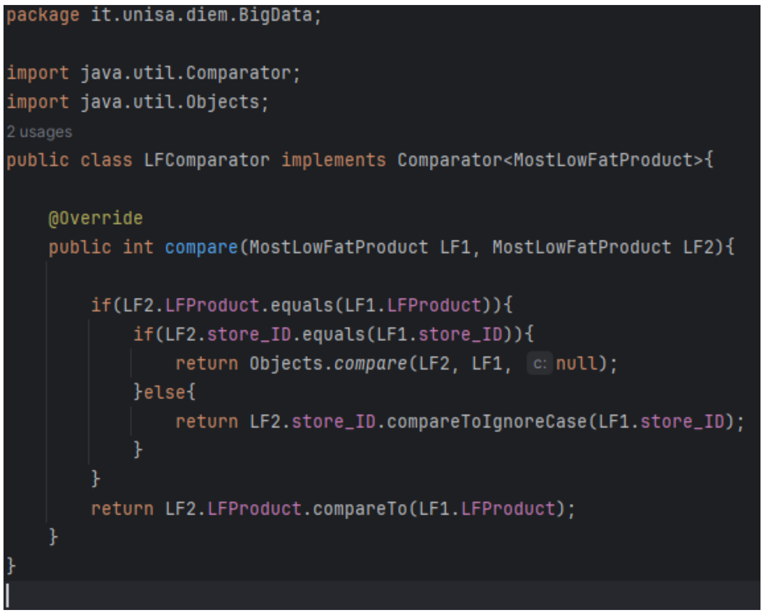
Figure 5 - Final output, generated from the second Job

The output has indeed the key-pair value desired, with labels for each row.

The next figure represents the code for the two Java class aforementioned.



```java
public class MostLowFatProduct implements Writable {
    11 usages
    String store_ID;
    11 usages
    Integer LFProduct;
    no usages
    public MostLowFatProduct() {
        this.store_ID = "";
        this.LFProduct = 0;
    }
    2 usages
    public MostLowFatProduct(String store_ID, Integer LFProduct) {
        this.store_ID = store_ID;
        this.LFProduct = LFProduct;
    }
    1 usage
    public String getStore_ID() { return store_ID; }
    no usages
    public void setStore_ID(String store_ID) { this.store_ID = store_ID; }
    1 usage
    public Integer getLFProduct() {
        return LFProduct;
    }
    no usages
    public void setLFProduct(Integer LFProduct) {
        this.LFProduct = LFProduct;
    }
    @Override
    public void readFields(DataInput input) throws IOException{
        store_ID = input.readUTF();
        LFProduct = input.readInt();
    }
    @Override
    public void write(DataOutput output) throws IOException{
        output.writeUTF(store_ID);
        output.writeInt(LFProduct);
    }
    @Override
    public String toString(){
        return "Store_ID: "+store_ID+", Low_Fat_Product: "+LFProduct;
    }
}
```



```java
package it.unisa.diem.BigData;

import java.util.Comparator;
import java.util.Objects;
2 usages
public class LFComparator implements Comparator<MostLowFatProduct>{

    @Override
    public int compare(MostLowFatProduct LF1, MostLowFatProduct LF2){

        if(LF2.LFProduct.equals(LF1.LFProduct)){
            if(LF2.store_ID.equals(LF1.store_ID)){
                return Objects.compare(LF2, LF1, c: null);
            }else{
                return LF2.store_ID.compareToIgnoreCase(LF1.store_ID);
            }
        }
        return LF2.LFProduct.compareTo(LF1.LFProduct);
    }
}
```

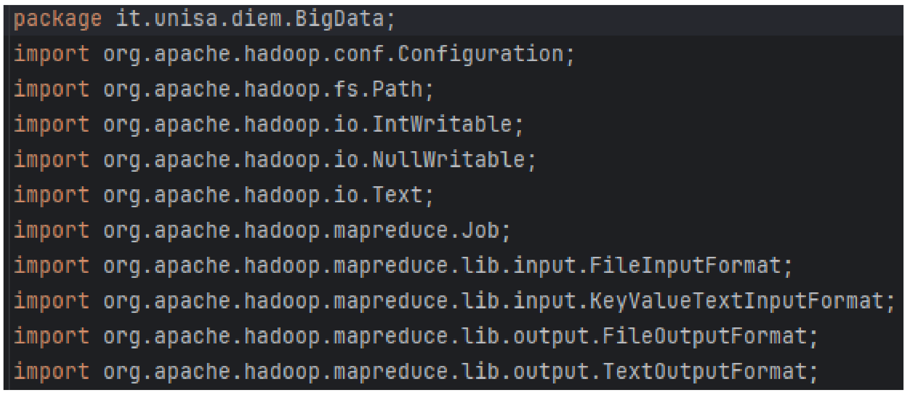
Figure 6 - Other classes used in the Java program

They are simple classes, self-explanatory, that implements simple functions like getter and setter, constructor and an override to the toString method. A compare method is present in the "LFComparator" class, that just compares the number of Low-Fat products for each line.

The last class, that implements the main, it's the Driver class visible in the figures in the next page. It's a simple class that has few uses:

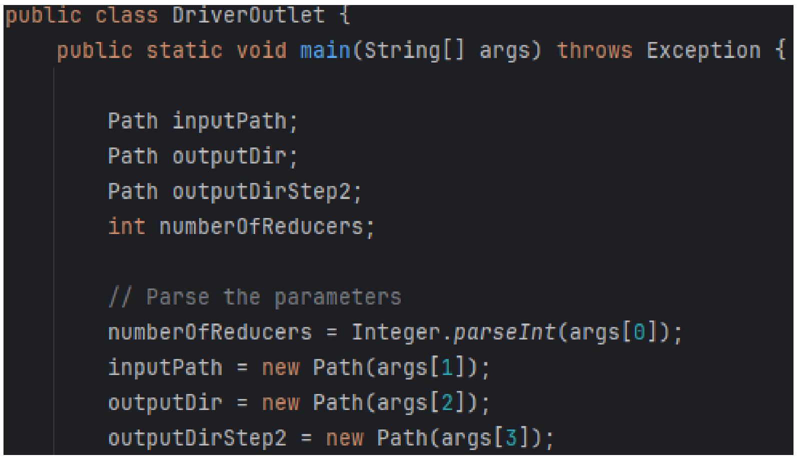
- get the arguments from the user and parse them;
- set the path for input and outputs files;
- creates the jobs and assign to each a job name;
- format the input and the outputs of each class;
- call the classes in the order specified;

Using this class, it's possible to chain the two jobs in the right order. It's a long and self-explanatory class and will be divided into multiple images.



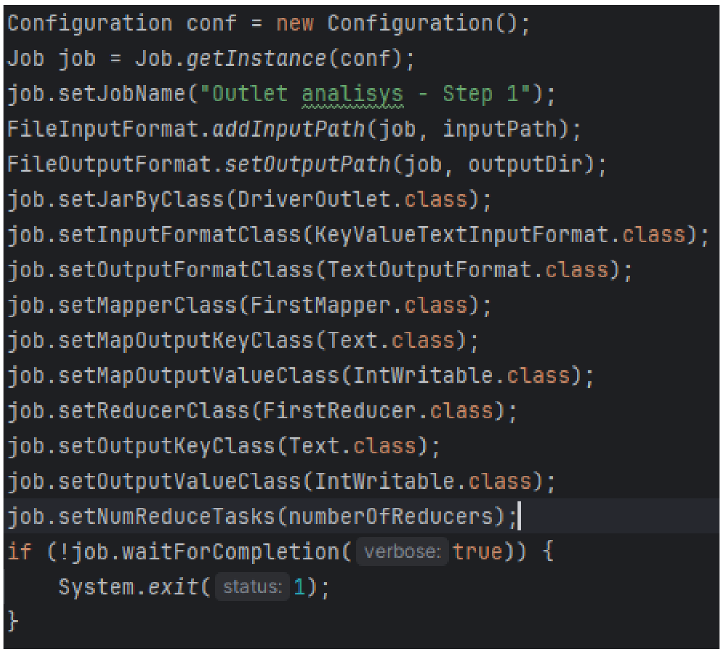
```java
package it.unisa.diem.BigData;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.KeyValueTextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
```

Figure 7 - Package and import used in the Driver Class



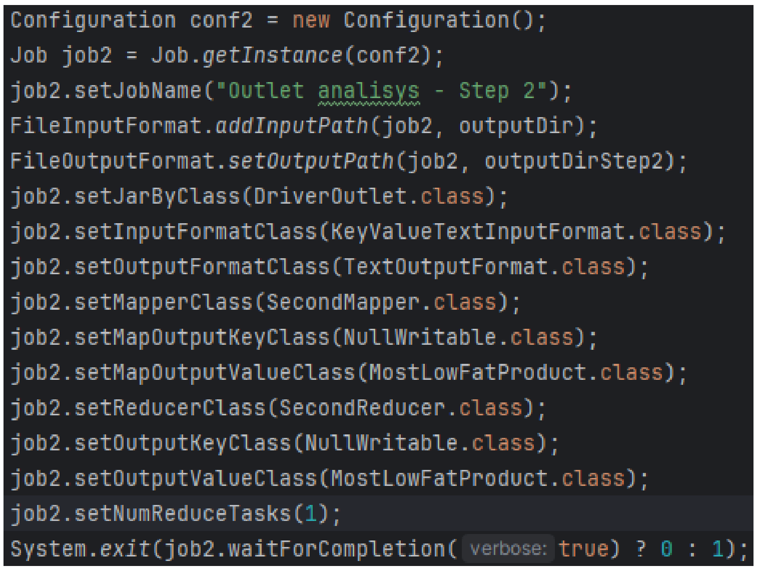
```java
public class DriverOutlet {
    public static void main(String[] args) throws Exception {

        Path inputPath;
        Path outputDir;
        Path outputDirStep2;
        int numberOfReducers;

        // Parse the parameters
        numberOfReducers = Integer.parseInt(args[0]);
        inputPath = new Path(args[1]);
        outputDir = new Path(args[2]);
        outputDirStep2 = new Path(args[3]);
```

Figure 8 - Setup phase of all the needed parameters



```java
Configuration conf = new Configuration();
Job job = Job.getInstance(conf);
job.setJobName("Outlet analisys - Step 1");
FileInputFormat.addInputPath(job, inputPath);
FileOutputFormat.setOutputPath(job, outputDir);
job.setJarByClass(DriverOutlet.class);
job.setInputFormatClass(KeyValueTextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
job.setMapperClass(FirstMapper.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);
job.setReducerClass(FirstReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
job.setNumReduceTasks(numberOfReducers);
if (!job.waitForCompletion( verbose: true)) {
    System.exit( status: 1);
}
```

Figure 9 - First Configuration and First Job



```java
Configuration conf2 = new Configuration();
Job job2 = Job.getInstance(conf2);
job2.setJobName("Outlet analisys - Step 2");
FileInputFormat.addInputPath(job2, outputDir);
FileOutputFormat.setOutputPath(job2, outputDirStep2);
job2.setJarByClass(DriverOutlet.class);
job2.setInputFormatClass(KeyValueTextInputFormat.class);
job2.setOutputFormatClass(TextOutputFormat.class);
job2.setMapperClass(SecondMapper.class);
job2.setMapOutputKeyClass(NullWritable.class);
job2.setMapOutputValueClass(MostLowFatProduct.class);
job2.setReducerClass(SecondReducer.class);
job2.setOutputKeyClass(NullWritable.class);
job2.setOutputValueClass(MostLowFatProduct.class);
job2.setNumReduceTasks(1);
System.exit(job2.waitForCompletion( verbose: true) ? 0 : 1);
```

Figure 10 - Second Configuration and Second Job

3.2 Docker commands

Once the Java program is complete it is possible to generate the JAR files, copy and paste this file and the dataset in the Hadoop cluster (under the hddata folder) and utilize the docker commands to run the program. Following from the steps done in chapter 2.1, once launched the last commands, we must change directory and go inside the "data" folder (using the basic command "*cd data*"). Here we must set the input file with the command:

- *hdfs dfs -put FileName.txt hdfs:///input* .

We can paste the file in the terminal using the command:

- *hdfs dfs -cat /input* .

It is time now to launch the program with the command:

- *hadoop jar NomeFileJar.jar #Reducer /input /intermediateOutput /finalOutput* ,

specifying the right order and number of arguments and the right name of the JAR file. The arguments are:

- # recuders: self-explanatory, specify the number of reducers to be used;
- /input: set the input file;
- /intermediateOutput: set the name of the intermediate output file (used for the first job);
- /finalOutput: set the name of the final output file (used for the second job).

The program should run without errors, and we can check the results once it is finished.

We can use the following commands:

- *hdfs dfs -cat /finalOutput /part-r-00000*: to put in the terminal the content of the file;
- *hdfs dfs -get / finalOutput /part-r-00000*: to save the file locally.

Once finished we must clean the file system with the following commands:

- *hdfs dfs -rm -r hdfs:///input*: clean the input set before;
- *hdfs dfs -rm -r hdfs:///output*: clean the output set before.

Then using the word “exit” we can exit the container. Now we can kill all the process with the command:

- *docker kill master slave1 slave2 slave3*;

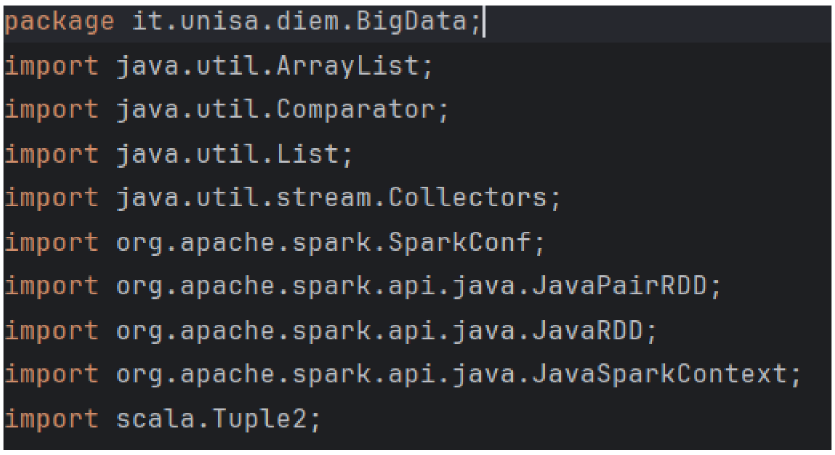
it is not necessary to remove them or the Hadoop network, if you’re going to use them again.

4. Spark Project

The Spark project analyze the dataset and extract the Top-K most expensive outlet (based on the mean of the Maximum Retail Price for each outlet) for every outlet dimension present in the dataset. Since the dimension are three (Small, Medium and High) and, at first visual impact there is at least one outlet for each size, the program should return at least 3 rows. Ideally, the program should return a maximum of 9 rows.

4.1 Java Program

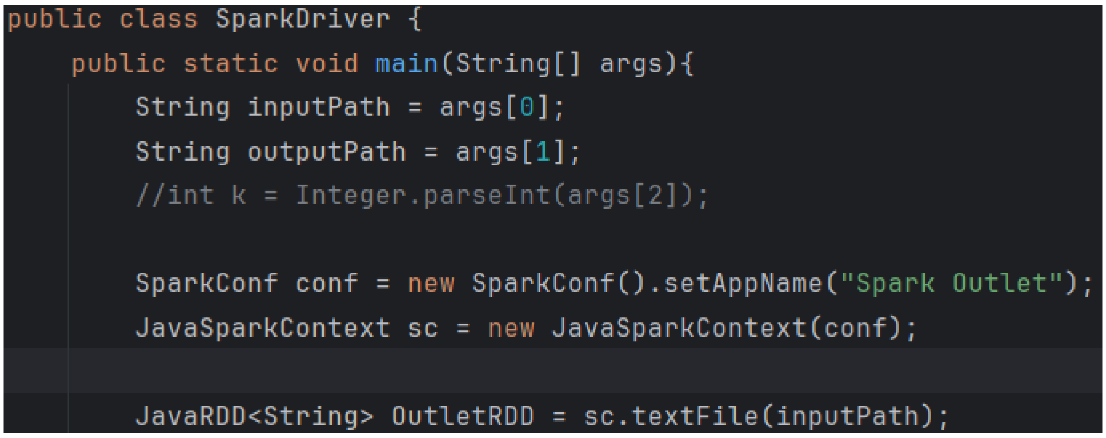
The Java program is much simpler than it's Hadoop counterpart, being made of only one class. In fact, in Spark, everything is contained in the main class. In Figure 11we can see the package and all the imported class used by the program.



```java
package it.unisa.diem.BigData;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import scala.Tuple2;
```

Figure 11 – Spark program imported classes and package

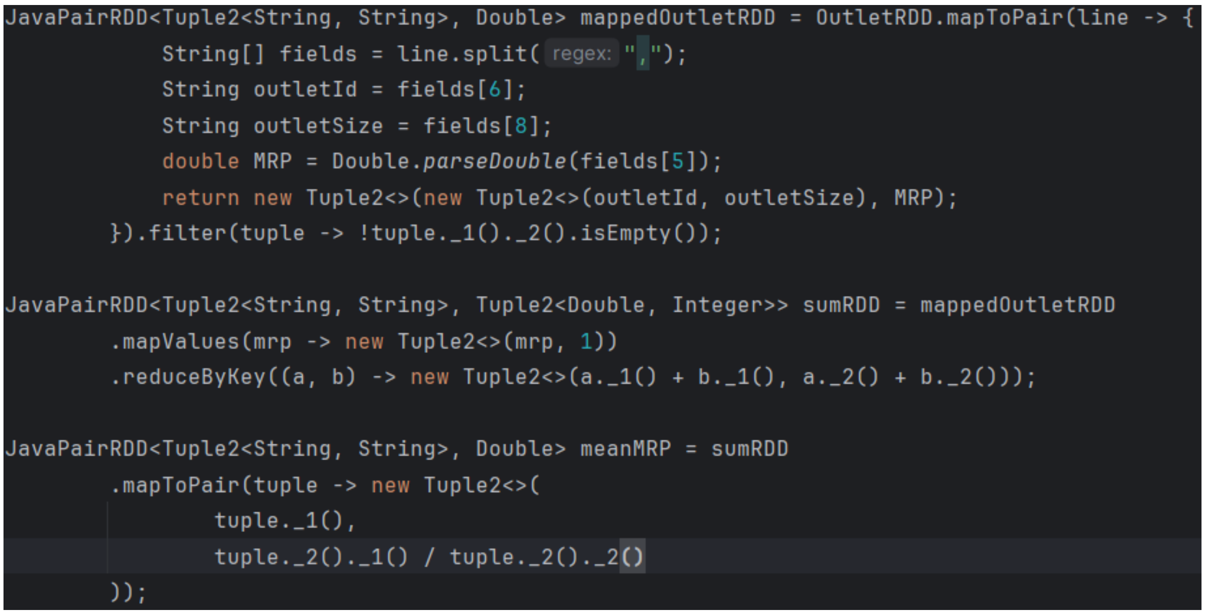
As can be seen in Figure 12, the first lines on the program set the input and output path into two variables. Additionally, if we want to specify the K parameters, we can add a third variables to store it (commented in the program since the K was already assigned and equal to three).



```java
public class SparkDriver {
    public static void main(String[] args){
        String inputPath = args[0];
        String outputPath = args[1];
        //int k = Integer.parseInt(args[2]);

        SparkConf conf = new SparkConf().setAppName("Spark Outlet");
        JavaSparkContext sc = new JavaSparkContext(conf);

        JavaRDD<String> OutletRDD = sc.textFile(inputPath);
```

Figure 12 - Setup phase for the Spark program

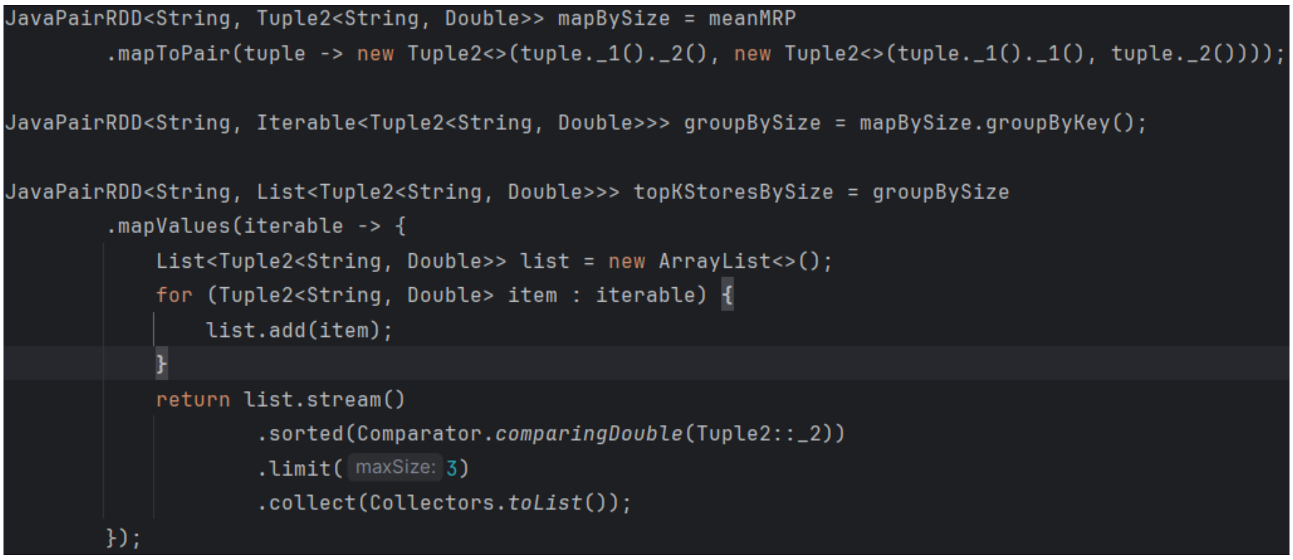
Then we can create the Configuration Object and the Spark Context Object. Lastly, we read the input file.
The next step is to parse the input and to get only the parameters we need, in this case the Store ID (unique for every store), the size of the store and the MRP (referring to the MRP present in that specific line). This step can be seen in Figure 13, on the next page, with the passage to compute the sum (for each pair "Store_ID, outlet_Size") and the successive mean of the MRP, for each store.



```java
JavaPairRDD<Tuple2<String, String>, Double> mappedOutletRDD = OutletRDD.mapToPair(line -> {
            String[] fields = line.split( regex: ",");
            String outletId = fields[6];
            String outletSize = fields[8];
            double MRP = Double.parseDouble(fields[5]);
            return new Tuple2<>(new Tuple2<>(outletId, outletSize), MRP);
        }).filter(tuple -> !tuple._1()._2().isEmpty());

JavaPairRDD<Tuple2<String, String>, Tuple2<Double, Integer>> sumRDD = mappedOutletRDD
        .mapValues(mrp -> new Tuple2<>(mrp, 1))
        .reduceByKey((a, b) -> new Tuple2<>(a._1() + b._1(), a._2() + b._2()));

JavaPairRDD<Tuple2<String, String>, Double> meanMRP = sumRDD
        .mapToPair(tuple -> new Tuple2<>(
                tuple._1(),
                tuple._2()._1() / tuple._2()._2()
        ));
```

Figure 13 - Parsing of the input and computing of sum and mean of the MRP

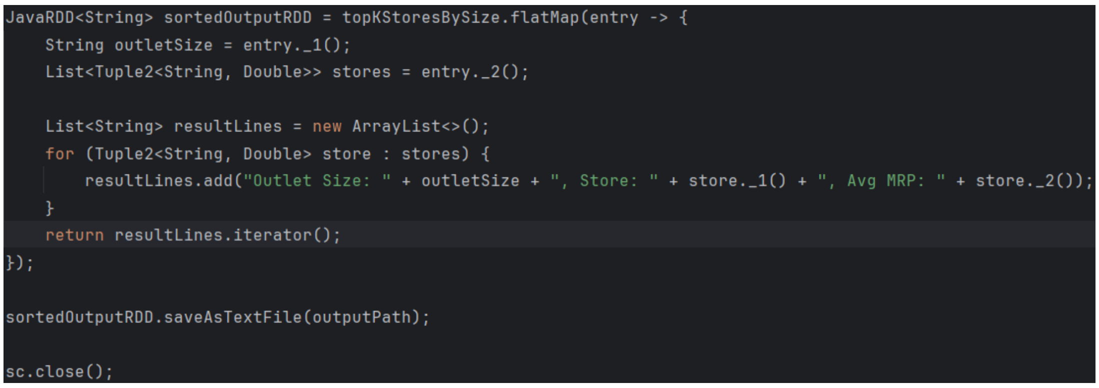
It's important to note that, on the "mappedOutletRDD", it's applied a filter to delete any lines with a missing size.



```java
JavaPairRDD<String, Tuple2<String, Double>> mapBySize = meanMRP
        .mapToPair(tuple -> new Tuple2<>(tuple._1()._2(), new Tuple2<>(tuple._1()._1(), tuple._2())));

JavaPairRDD<String, Iterable<Tuple2<String, Double>>> groupBySize = mapBySize.groupByKey();

JavaPairRDD<String, List<Tuple2<String, Double>>> topKStoresBySize = groupBySize
        .mapValues(iterable -> {
            List<Tuple2<String, Double>> list = new ArrayList<>();
            for (Tuple2<String, Double> item : iterable) {
                list.add(item);
            }
            return list.stream()
                    .sorted(Comparator.comparingDouble(Tuple2::_2))
                    .limit( maxSize: 3)
                    .collect(Collectors.toList());
        });
```

Figure 14 - Ascending ordering and Top-K

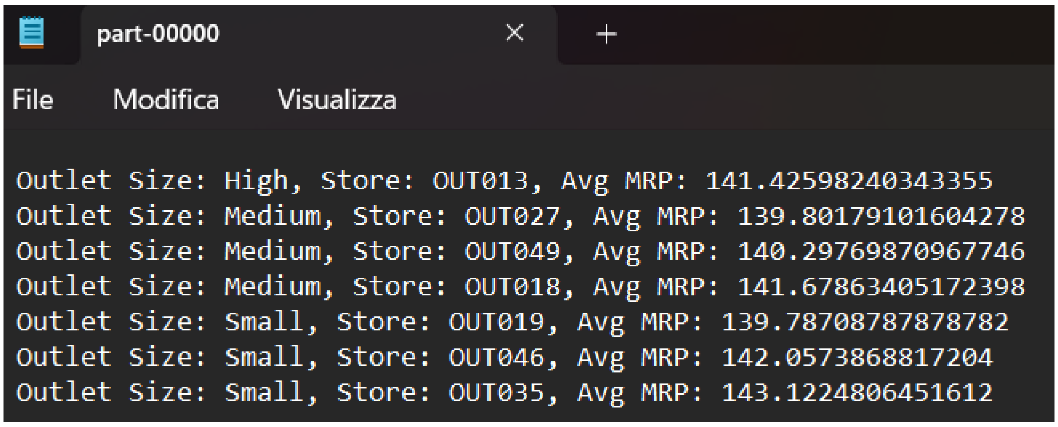
The next piece of code implements the sorting in ascending manner and the selection of only the top three store, for each size available and it's visible in Figure 14. The last part of the code simply saves the results in the output file, specified in the variables at the start of the program. It's visible in Figure 15.



```java
JavaRDD<String> sortedOutputRDD = topKStoresBySize.flatMap(entry -> {
    String outletSize = entry._1();
    List<Tuple2<String, Double>> stores = entry._2();

    List<String> resultLines = new ArrayList<>();
    for (Tuple2<String, Double> store : stores) {
        resultLines.add("Outlet Size: " + outletSize + ", Store: " + store._1() + ", Avg MRP: " + store._2());
    }
    return resultLines.iterator();
});

sortedOutputRDD.saveAsTextFile(outputPath);

sc.close();
```

Figure 15 - Final part of the program: write on the output file and close of the Spark Context

The output file is visible in Figure 16: as expected, there are at least three lines. In the specific, we can count seven lines. This because only the Small and Medium outlet have (at least) three occurrences in the dataset, while the High outlet seems to be only one.



```
Outlet Size: High, Store: OUT013, Avg MRP: 141.42598240343355
Outlet Size: Medium, Store: OUT027, Avg MRP: 139.80179101604278
Outlet Size: Medium, Store: OUT049, Avg MRP: 140.29769870967746
Outlet Size: Medium, Store: OUT018, Avg MRP: 141.67863405172398
Outlet Size: Small, Store: OUT019, Avg MRP: 139.78708787878782
Outlet Size: Small, Store: OUT046, Avg MRP: 142.0573868817204
Outlet Size: Small, Store: OUT035, Avg MRP: 143.1224806451612
```

Figure 16 - Output generated

In the end, the program writes seven lines in the file, with an ascending MRP order for each size of outlet.

4.2 Docker commands

The docker commands for using Spark are even simpler than those to use Hadoop.
The first step is to go into the Spark cluster directory. Here, put the dataset (the input text file) already sanitized (if necessary) into the "*Input*" directory. Generate the JAR files from your IDE and paste it into the Spark Cluster directory.
Then open the file "./launch_single.sh" with an editor and make the following changes:

- modify the "*class*" line with the package you've create in the program;
- set the right name of the JAR file;
- if necessary, set the K value using the notation "*$K*", where K should be an integer value;
- if present, delete the "*Output*" folder.

Save the file, start Docker and open a PowerShell Console. Then move to the Spark Cluster directory with the command:

- "*cd Spark_Cluster_Path*".

Once inside the directory use the following two commands:

- "*docker compose up -d*", to start the container;
- "*docker container exec -ti sp_master bash*", to start the network.

Here, change the directory and enter first in "*bin*" then in "*testfiles*".
Once here, it's possible to launch the program, writing in the console "*./launch_single.sh*". If no error is generated, the output should be available in the just created "*output*" folder.
To exit from the network, simply follow the same steps done for Hadoop, described in 3.2.

Table of figures