Distributed architectures for big data processing and analytics

Month Day, Year – Exam Example #4

Student ID ____

First Name ____

Last Name ____

The exam lasts **2 hours**

Part I

Answer to the following questions. There is only one right answer for each question.

1. (2 points) Consider the HDFS folder "inputFolder" containing the following two files:

Filename	**Size**	**Content of the file**
HumidityA.txt	16 bytes	51.45 9.55 8.15
HumidityB.txt	18 bytes	40.53 12.98 52.99

Suppose that you are using a Hadoop cluster that can potentially run up to 4 mappers in parallel and suppose that the HDFS block size is 2048MB.

Suppose that the following MapReduce program is executed by providing the folder "inputFolder" as input folder and the folder "results" as output folder.

```
/* Driver */
import … ;
public class DriverBigData extends Configured implements Tool {
        @Override
        public int run(String[] args) throws Exception {
                Configuration conf = this.getConf();
                Job job = Job.getInstance(conf);
                job.setJobName("2018/09/03 - Theory");

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

                job.setJarByClass(DriverBigData.class);

                job.setInputFormatClass(TextInputFormat.class);
                job.setOutputFormatClass(TextOutputFormat.class);

                job.setMapperClass(MapperBigData.class);
                job.setMapOutputKeyClass(DoubleWritable.class);
                job.setMapOutputValueClass(NullWritable.class);
                job.setNumReduceTasks(0);

                if (job.waitForCompletion(true) == true)
                        return 0;
                else
                        return 1;
        }

        public static void main(String args[]) throws Exception {
                int res = ToolRunner.run(new Configuration(), new DriverBigData(), args);
                System.exit(res);
        }
}
```

```
/* Mapper */
import …;

class MapperBigData extends Mapper<LongWritable, Text, DoubleWritable, NullWritable> {
        Double top1;

        protected void setup(Context context) {
                top1 = null;
        }

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

                Double val = new Double(value.toString());

                if (top1 == null || val.doubleValue() > top1) {
                        top1 = val;
                }
        }

        protected void cleanup(Context context) throws IOException, InterruptedException {
                // emit the content of top1
                context.write(new DoubleWritable(top1), NullWritable.get());
        }
}
```

What is the output generated by the execution of the application reported above?

a) One file containing 52.99

b) Two files

- One containing the value 52.99
- One empty file

c) Two files

- One containing the value 51.45

- One containing the value 40.53

[✓ d)] Two files

- One containing the value 51.45
- One containing the value 52.99

Each Mapworks specifically on one single File

2. (2 points) Consider the HDFS files prices.txt and prices2.txt. The size of prices.txt is 2000MB and the size of prices2.txt is 1072MB. Suppose that you are using a Hadoop cluster that can potentially run up to 20 mappers in parallel and suppose to execute a map-only MapReduce-based program that receives as input the folder containing prices.txt and prices2.txt and selects the rows of the two files containing prices greater than 10.5. How many mappers are instantiated by Hadoop if the HDFS block size is 1024MB?

   a) 2 mappers

   b) 3 mappers

   [✓ c)] 4 mappers

   d) 20 mappers

prices.txt $\rightarrow$ $\frac{2000MB}{1024}$ = 2 Mappers

prices 2.txt $\rightarrow$ $\frac{1072MB}{1024MB}$ = 2 Mappers

= 4 Mappers are Instantiated By Hadoop

Part II

PoliStocks is an international company that collects and analyzes stock data. To identify interesting stocks, PoliStocks computes a set of statistics based on the following dataset file.

- Stocks_Prices.txt
  - Stocks_Prices.txt is a text file containing the historical information about the last 10 years of prices of thousands of stocks on several international financial markets.
  - The sampling rate is 1 minute (i.e., every minute the system collects the prices of the stocks under analyses and a new line for each stock is appended at the end of Stocks_Prices.txt)
  - Each line of the input file has the following format
    - stockId,date,hour:minute,price

      where *stockId* is a stock identifier (e.g., GOOG) and *price* is a floating point number whose value indicates the price of the stock *stockId* at time *date,hour:minute.*
    - For example, the line

      GOOG,2015/05/21,15:05,45.32

      means that the price of stock ***GOOG*** on ***May 21, 2015*** at ***15:05*** was ***45.32€***

Exercise 1 – MapReduce and Hadoop (8 points)

The managers of PoliStocks are interested in selecting the highest price reached by the GOOG stock (stockId="GOOG") during year 2017 and the first time stamp of year 2017 in which that highest price has been reached.

Design a single application, based on MapReduce and Hadoop, and write the corresponding Java code, to address the following point:

A. *Highest GOOG price in year 2017 and associated time stamp*. Considering only the prices of the GOOG stock (stockId="GOOG") in year 2017, the application must select the highest price reached by the GOOG stock during year 2017 and the first time stamp (date+hour+minute) of year 2017 in which that price was reached. Store the result of the analysis in a HDFS folder. The output contains one single line with the selected highest price and the associated first time stamp in the form

   price\t*date,hour:minute*

The arguments of the Hadoop application are (i) the path of the input file Stocks_Prices.txt and (ii) the name of the output folder. Note that Stocks_Prices.txt contains the prices associated with the last 10 years and all stocks but the analysis we are interested in is limited to year 2017 and the GOOG stock only.

Fill out the provided template for the Driver of this exercise. Use your sheets of paper for the other parts (Mapper and Reducer).

Exercise 2 – Spark and RDDs (19 points)

The managers of PoliStocks are interested in (A) performing some analyses about the daily variations of each stock during year 2017 and (B) identifying "stable trends in year 2017" for each stock.

The managers of PoliStocks asked you to develop **one single application** to address all the analyses they are interested in. The application has the following arguments: the path of the input file Stocks_Prices.txt and two output folders (associated with the outputs of the following points A and B, respectively).

Specifically, design a single application, based on Spark RDDs or Spark DataFrames, and write the corresponding code, to address the following points:

A. *Analyses of daily variations in year 2017*. Considering only the prices in year 2017, the application must compute for each stock the number of dates associated with a daily price variation greater than 10 euro. Given a stock and a date, the daily price variation of that stock in that date is given by the difference between the highest price and the lowest price reached by that stock in that specific date (highest price–lowest price). Store in the first HDFS output folder the set of "*stockId*,*number_of_dates_dailyVariation>10*", one line for each *stockId*, reporting **only** the stocks associated with **at least one date** in year 2017 with a daily price variation greater than 10 euro.
Sorting the results is not required.

B. *Stable trends in year 2017 for each stock*. Considering only the prices in year 2017, for each stock, the application must select all the sequences of two consecutive dates characterized by a "*stable trend*". Given a stock and two consecutive dates, the trend of that stock for those two dates is classified as a "*stable trend*" if and only if the absolute difference between the daily price variations of the two dates is at most 0.1 euro (i.e., abs(daily price variation of the first date - daily price variation of the second date) <= 0.1). The application stores in the second HDFS output folder the full set of all stable trends, one per line. Specifically, each line contains the following information about one stable trend: stockId, first date of the stable trend. Each output line has the following format:

stockId,first_date

For instance, suppose that stock GOOG has a daily variation equal to 2.03 in April 2, 2017 and a daily variation equal to 2.05 in April 3, 2017. It means that "GOOG,2017/04/02,2017/04/03" is classified as a "stable trend" because abs(2.03-2.05)<=0.1 and the following line is inserted in the output:

GOOG,2017/04/02

Note that you can have many pairs of consecutive dates of stable trends for each stock, hence having multiple lines in the output file for each stock.
Sorting the results is not required.

Suppose that someone has already implemented the following function

- *previousDate(String date)*
    - The parameter of this function is a string representing a date (in the format yyyy/mm/dd). The returned value is a string representing the previous date
    - For example, the invocation

      yesterday=previousDate("2016/06/20")

      returns and stores "2016/06/19" in the variable *yesterday.*

Stock_Prices :
stockID, date, hour.minute, price

A)
- Filter only the measurement computed in 2017

  filter(lambda : line.split(',')[1].split('/')[0] == '2017')
- Map each input lines to a tuple ((stockID, date), (price, price)) in order to compute in the next step for each stockID, date the max and min price

  map(definedFunction)
- Compute for each StockID, date the max and min price

  Cached → reduceByKey(lambda v1,v2 : (max(v1,v2), min(v1,v2)))
- Filter lines associated with max price - min price > 10

  filter(lambda pair : pair[1][0] - pair[1][1] > 10)
- Map to each line the tuple (stockID, 1) to count the n° of days

  map(lambda pair : (pair[0][0], 1))
- Count the n° of days

  reduceByKey(lambda v1,v2 : v1+v2)

→ Obtain for each StockID the n° of days associated with difference between max and min price > 10

B) • Cache the RDD related to:

(stockID, date), (max(price), min(price))

• Compute the daily variation for each line:

mapValues(lambda pair: pair[0] - pair[1])

((GOOG, 2017/06/20), 50)

((GOOG, 2017/06/21), 50)

((GOOG, 2017/06/22), ...)

Consecutives Dates

• Produce for each (StockID, date) two pairs:

First → (stockID, date), dailyVariation

Second → (stockID, date-1), dailyVariation

→ Since each Pair (stockID, date) is both the first element of the sequence starting in that date and also the second element of the sequence started the date before.

flatMap(definedFunction)

- GroupBy the stockID, date , obtaining in the Value Part the DailyVariations associated with 2 Consequent date.

  groupByKey()

- Filter the stockID, date associated with $|Val1 - Val2| \leq 0.1$

  filter(defined Function)

- Select only the stockIn and date

  Keys()

Distributed architectures for big data processing and analytics

Month Day, Year – Exam Example #4

Student ID ____

First Name ____

Last Name ____

Use the following template for the Driver of Exercise 1

Fill in the missing parts. You can strikethrough the second job if you do not need it.

```
import ….
/* Driver class. */
public class DriverBigData extends Configured implements Tool {
        public int run(String[] args) throws Exception {
        Path inputPath = new Path(args[0]); Path outputDir = new Path(args[1]);
        Configuration conf = this.getConf();

        // First job
        Job job1 = Job.getInstance(conf);
        job1.setJobName("Exercise 1 - Job 1");
        // Job 1 - Input path
        FileInputFormat.addInputPath(job,________________);

        // Job 1 - Output path
        FileOutputFormat.setOutputPath(job,________________);

        // Job 1 - Driver class
        job1.setJarByClass(DriverBigData.class);

        // Job1 - Input format
        job1.setInputFormatClass(__________________________________);

        // Job1 - Output format
        job1.setOutputFormatClass(__________________________________);

        // Job 1 -  Mapper class
        job1.setMapperClass(Mapper1BigData.class);
        // Job 1 – Mapper: Output key and output value: data types/classes
        job1.setMapOutputKeyClass(__________________________________);

        job1.setMapOutputValueClass(__________________________________);

        // Job 1 -  Reducer class
        job.setReducerClass(Reducer1BigData.class);

        // Job 1 – Reducer: Output key and output value: data types/classes
        job1.setOutputKeyClass(__________________________________);

        job1.setOutputValueClass(__________________________________);

        // Job 1 - Number of instances of the reducer of the first Job
        job1.setNumReduceTasks( 0[ _ ] or exactly 1[ _ ] or any number >=1[ _ ] ); /* Select only one of
                                                                                     these three options */
}

        // Execute the first job and wait for completion
        if (job1.waitForCompletion(true)==true)
        {
                // Second job
                Job job2 = Job.getInstance(conf);
                job2.setJobName("Exercise 1 - Job 2");
                // Set path of the input folder of the second job
                FileInputFormat.addInputPath(job2,___________________________________);

                // Set path of the output folder for the second job
                FileOutputFormat.setOutputPath(job2,___________________________________);

                // Class of the Driver for this job
                job2.setJarByClass(DriverBigData.class);

                // Set input format
                job2.setInputFormatClass(___________________________________);

                // Set output format
                job2.setOutputFormatClass(___________________________________);

                // Set map class
                job2.setMapperClass(Mapper2BigData.class);

                // Set map output key and value classes
                job2.setMapOutputKeyClass(___________________________________);

                job2.setMapOutputValueClass(___________________________________);

                // Set reduce class
                job2.setReducerClass(Reducer2BigData.class);

                // Set reduce output key and value classes
                job2.setOutputKeyClass(___________________________________);

                job2.setOutputValueClass(___________________________________);

                // Job 2 - Number of instances of the reducer of the second Job
                Job2.setNumReduceTasks( 0[ _ ] or exactly 1[ _ ] or any number >=1[ _ ] ); /* Select only
                                                                                one of these three options */

                // Execute the job and wait for completion
                if (job2.waitForCompletion(true)==true)
                        exitCode=0;
                else
                        exitCode=1;
          }
          else
                exitCode=1;

          return exitCode;
        }
        /* Main of the driver  */
        public static void main(String args[]) throws Exception {
        int res = ToolRunner.run(new Configuration(), new DriverBigData(), args);
        System.exit(res);
        }
}
```