Big data processing and analytics

September 12, 2024

Student ID _____

First Name _____

Last Name _____

The exam is **open book**

Part I

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

1. (2 points) Consider the following Spark Streaming application.

```
Consider the following Spark application
# Create an RDD associated with a first input file
Temp1RDD = sc.textFile("Temperature1.txt")

# Print on the standard output the number of elements of Temp1RDD
print("elements Temp1RDD "+str(Temp1RDD.count()))

# Create an RDD associated with a second input file
Temp2RDD = sc.textFile("Temperature2.txt")

# Print on the standard output the number of elements of Temp2RDD
print("elements Temp2RDD "+str(Temp2RDD.count()))

# Create an RDD that contains the union of Temp1RDD and Temp2RDD
UnionRDD = Temp1RDD.union(Temp2RDD)

# Computes the number of lines of UnionRDD
numLinesUnion = UnionRDD.count()

# Print on the standard output the value of numLinesUnion
print("numLinesUnion "+str(numLinesUnion))

# Store the content of UnionRDD in the output folder
UnionRDD.saveAsTextFile("outputFolder/")
```

Suppose the input files Temperature1.txt and Temperature2.txt are read from HDFS. Suppose this Spark application is executed only 1 time. Which one of the following statements is true?

a) This application reads the content of Temperature1.txt 1 time and the content of Temperature2.txt 1 time.

b) This application reads the content of Temperature1.txt 2 times and the content of Temperature2.txt 2 times.
c) This application reads the content of Temperature1.txt 3 times and the content of Temperature2.txt 3 times.
d) This application reads the content of Temperature1.txt 4 times and the content of Temperature2.txt 4 times.

2. (2 points) Consider the following MapReduce application for Hadoop.

DriverBigData.java

```
/* Driver class */
package it.polito.bigdata.hadoop;
import ….;


/* Driver class */
public class DriverBigData extends Configured implements Tool {
  @Override
public int run(String[] args) throws Exception {
   int exitCode;

   Configuration conf = this.getConf();

   // Define a new job
   Job job = Job.getInstance(conf);

   // Assign a name to the job
   job.setJobName("Exercise 12/09/2024 - Question");

   // Set path of the input file/folder for this job
   FileInputFormat.addInputPath(job, new Path("inputFolder/"));

   // Set path of the output folder for this job
   FileOutputFormat.setOutputPath(job, new Path("outputFolder/"));

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

   // Set job input format
   job.setInputFormatClass(TextInputFormat.class);

   // Set job output format
   job.setOutputFormatClass(TextOutputFormat.class);

   // Set map class
   job.setMapperClass(MapperBigData.class);
```

```
   // Set map output key and value classes
   job.setMapOutputKeyClass(Text.class);
   job.setMapOutputValueClass(NullWritable.class);



   // Set reduce class
   job.setReducerClass(ReducerBigData.class);

   // Set reduce output key and value classes
   job.setOutputKeyClass(IntWritable.class);
   job.setOutputValueClass(NullWritable.class);

   // Set the number of reducers to 3
   job.setNumReduceTasks(3);



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

MapperBigData.java

```
/* Mapper class */
package it.polito.bigdata.hadoop;
import …;

class MapperBigData extends
                  Mapper<LongWritable, // Input key type
                              Text, // Input value type
                              Text, // Output key type
                              NullWritable> { // Output value type

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

                  // Emit the pair (value, NullWritable) if line starts with "C"
```

```
                if (value.toString().startsWith("C") == true) {
                        context.write(new Text(value), NullWritable.get());
                }
        }
}
```

ReducerBigData.java

```
/* Reducer class */
package it.polito.bigdata.hadoop;
import …;

class ReducerBigData extends
                Reducer<Text,    // Input key type
                        NullWritable,   // Input value type
                        IntWritable,    // Output key type
                        NullWritable> { // Output value type

        // Define numLinesC
        int numLinesC;

        protected void setup(Context context) {
                // Initialize numLinesC
                numLinesC = 0;
        }

        protected void reduce(Text key, // Input key type
                        Iterable<NullWritable> values, // Input value type
                        Context context) throws IOException, InterruptedException {
                // Increment numLinesC
                numLinesC++;
        }

        protected void cleanup(Context context) throws IOException, InterruptedException {
        // Emit the pair (numLinesC, NullWritable))
                context.write(new IntWritable(numLinesC), NullWritable.get());
        }
}
```

Suppose that inputFolder contains the files Cities1.txt and Cities2.txt. Suppose the HDFS block size is 1024 MB.

Content of Cities1.txt and Cities2.txt:

Filename (size and number of lines)	Content

Cities1.txt (65 bytes – 8 lines)	Beijing **Cairo** Delhi Dhaka Dortmund Mexico City Mumbai São Paulo
Cities2.txt (39 bytes – 5 lines)	**Cairo** **Chongqing** Delhi Istanbul Kolkata

Suppose we run the above MapReduce application (note that the input folder is set to inputFolder/).

What is a ***possible*** output generated by running the above application?

a) The content of the output folder is as follows.

-rw-r--r-- 1 paolo paolo 2 set 3 14:00 part-r-00000

-rw-r--r-- 1 paolo paolo 2 set 3 14:00 part-r-00001

-rw-r--r-- 1 paolo paolo 2 set 3 14:00 part-r-00002

-rw-r--r-- 1 paolo paolo 0 set 3 14:00 _SUCCESS

The content of the three part files is as follows.

Filename (number of lines)	Content
part-r-00000 (1 line)	2
part-r-00001 (1 line)	0
part-r-00002 (1 line)	0

b) The content of the output folder is as follows.

-rw-r--r-- 1 paolo paolo 2 set 3 14:00 part-r-00000

-rw-r--r-- 1 paolo paolo 2 set 3 14:00 part-r-00001

-rw-r--r-- 1 paolo paolo 2 set 3 14:00 part-r-00002

-rw-r--r-- 1 paolo paolo 0 set 3 14:00 _SUCCESS

The content of the three part files is as follows.

Filename (number of lines)	Content
part-r-00000 (1 line)	1
part-r-00001 (1 line)	2
part-r-00002 (1 line)	0

c) The content of the output folder is as follows.

-rw-r--r-- 1 paolo paolo 2 set 3 14:00 part-r-00000

-rw-r--r-- 1 paolo paolo 2 set 3 14:00 part-r-00001

-rw-r--r-- 1 paolo paolo 2 set 3 14:00 part-r-00002

-rw-r--r-- 1 paolo paolo 0 set 3 14:00 _SUCCESS

The content of the three part files is as follows.

Filename (number of lines)	Content
part-r-00000 (1 line)	1
part-r-00001 (1 line)	1
part-r-00002 (1 line)	1

d) The content of the output folder is as follows.

-rw-r--r-- 1 paolo paolo 2 set 3 14:00 part-r-00000

-rw-r--r-- 1 paolo paolo 0 set 3 14:00 part-r-00001

-rw-r--r-- 1 paolo paolo 0 set 3 14:00 part-r-00002

-rw-r--r-- 1 paolo paolo 0 set 3 14:00 _SUCCESS

The content of the three part files is as follows.

Filename (number of lines)	Content
part-r-00000 (1 line)	3
part-r-00001 (1 line)	0
part-r-00002 (1 line)	0

Part II

PoliSales is an international e-commerce company that asked you to develop two applications:

- a MapReduce program (Exercise 1)
- a Spark-based program (Exercise 2)

to address the analyses they are interested in. The applications can read data from the input files described in the following.

- **Products.txt**
  - Products.txt is a textual file containing information about the Products that are sold by PoliSales. There is one line for each Product and the total number of distinct Products is greater than 5,000,000. This file is large, and you cannot suppose the content of Products.txt can be stored in one in-memory Java variable.
  - Each line of Products.txt has the following format
    - ProductID,Name,Category

      where *ProductID* is the unique identifier of the Product, *Name* is the name of ProductID, and *Category* is its category.
    - For example, the following line

      ID1,Galaxy S24 256GB Black,Smartphone

      means that the Product with ProductID ***ID1*** is characterized by the name ***Galaxy S24 256GB Black*** and belongs to the ***Smartphone*** category.

    Note that many Products can be associated to the same category.

- **Prices.txt**
  - Prices.txt is a textual file containing information about the prices of the Products. The price of each Product varies over time. There are potentially multiple lines for each Product. This file is large, and you cannot suppose the content of Prices.txt can be stored in one in-memory Java variable.
  - Each line of Prices.txt has the following format
    - ProductID,StartingDate,EndingDate,Price

      where *ProductID* is the identifier of a Product, and *StartingDate* and *EndingDate* are the beginning and end of the period of validity of the price reported in the attribute *Price* for Product *ProductID.*
    - For example, the following line

      ID1,2021/01/31,2022/21/31,98.7

      means the price associated with Product ***ID1*** from ***January 31, 2021***, to ***December 31, 2022***, was ***98.70*** euros. The format of StartingDate and EndingDate is “YYYY/MM/DD”.

Note that the price of each Product varies over time. Every time there is a price variation, a new line is inserted in Prices.txt with information about the new price and its validity period. Each Product is associated with a single price in each period.

- **Sales.txt**
  - Sales.txt is a textual file containing information about daily sales for each Product. Sales.txt contains historical data about the last 30 years. This file is big and its content cannot be stored in one in-memory Java variable. There is one line for each combination (ProductID, Date) for the last 30 years.
  - Each line of Sales.txt has the following format
    - ProductID,Date,NumberOfProductsSold

      where *ProductID* is the identifier of an Product, *Date* is a date, and *NumberOfProductsSold* is an integer value representing the number of times *ProductID* was sold on that *Date*.
    - For example, the following line

      ID1,2021/04/30,1234

      means that on ***April 30, 2021***, the Product identified by ***ID1*** was sold ***1234*** times. The format of Date is "YYYY/MM/DD".

  Note that there is a many-to-many relationship between Products and Dates (i.e., the combination of attributes (ProductID, Date) is the "primary key" of Sales.txt). Each Product is associated with all the dates of the last 30 years, and each date of the last 30 years is associated with all Products. Even if a Product was not sold on a specific date, there is a line for that combination in Sales.txt anyway, with NumberOfProductsSold set to 0 (zero).

Exercise 1 – MapReduce and Hadoop (8 points)

Exercise 1.1

The managers of PoliSales are interested in performing some analyses about the Product.

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

- *Categories with the largest number of products.* The application computes the number of products associated with each Category and outputs only the Categories with the largest number of products. Store the result in the output HDFS folder (one Category per output line associated with its number of products). Output format: *Category, Number of products*

Suppose that the number of distinct Categories is limited and they can fit in main memory.

Suppose that the input is Products.txt and it has already been set.

Suppose that the name of the output folder has also already been set.

- Write only the content of the Mapper and Reducer classes (map and reduce methods. setup and cleanup if needed). The content of the Driver must not be reported.
- Use the following two specific multiple-choice questions to specify the number of instances of the reducer class for each job.
- If your application is based on two jobs, specify which methods are associated with the first job and which are associated with the second job.
- If you need personalized classes, report for each of them:
    - the name of the class,
    - attributes/fields of the class (data type and name),
    - personalized methods (if any), e.g., the content of the toString() method if you override it,
    - do not report the get and set methods. Suppose they are "automatically defined".

Answer the following two questions to specify the number of jobs (one or two) and the number of instances of the reducer classes.

Exercise 1.2 - Number of instances of the reducer - Job 1

Select the number of instances of the reducer class of the first Job

(a) 0

(b) exactly 1

(c) any number >=1 (i.e., the reduce phase can be parallelized)

Exercise 1.3 - Number of instances of the reducer - Job 2

Select the number of instances of the reducer class of the second Job

(a) One single job is needed

(b) 0

(c) exactly 1

(d) any number >=1 (i.e., the reduce phase can be parallelized)

Exercise 2 – Spark and RDDs (19 points)

The managers of PoliSales asked you to develop a single Spark-based application based either on RDDs or Spark SQL to address the following tasks. The application takes the paths of the three input files and two output folders (associated with the outputs of the following points 1 and 2, respectively).

1. *Products that decreased their total sales in 2021 with respect to the sales in 2019*. The first part of this application considers only the years 2019 and 2021. It selects the Products with a total number of products sold in 2021 lower than the total number of products sold in 2019. Store the selected Products in the first output folder (one product per output line).

2. *Most sold product(s) for each year*. The second part of this application considers all input data. It selects, for each year, the product(s) associated with the highest total annual sales for that year. For each product and year, the total annual sales for that product is given by the sum of the number of products sold in all days of that year. Store the result in the second output folder, as pairs (year, ID of the most sold product for that year).

   Note: For each year, many products might be associated with the highest annual sales. Moreover, pay attention to the fact that the maximum annual sales can differ each year.

- You do not need to write Java imports. Focus on the content of the main method.
- Suppose both **JavaSparkContext sc** and **SparkSession ss** have already been set.
- **Only if you use Spark SQL**, suppose the first line of each file contains the header information/the name of the attributes. Suppose, instead, there are no header lines if you use RDDs.
- Please **comment** your solution by stating the meaning of the fields you intend to process with each instruction, e.g., key=(product id, date), value=(category, year)
- If you need personalized classes, report for each of them:
  - the name of the class,
  - attributes/fields of the class (data type and name),
  - personalized methods (if any), e.g., the content of the toString() method if you override it,
  - do not report the get and set methods. Suppose they are "automatically defined".