Distributed architectures for big data processing and analytics

September 6, 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 applications.

(Application A)

```
from pyspark.streaming import StreamingContext

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 10)

# Create a (Receiver) DStream that will connect to localhost:9999
inputDStream = ssc.socketTextStream("localhost", 9999)

resADStream = inputDStream\
.map(lambda value: int(value))
.filter(lambda value : value>5)
.window(30, 10)
.reduce(lambda v1,v2: max(v1, v2) )
.filter(lambda value : value>10)


# Print the result
resADStream.pprint()

ssc.start()
ssc.awaitTerminationOrTimeout(360)
ssc.stop(stopSparkContext=False)
```

(Application B)

```
from pyspark.streaming import StreamingContext

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 10)

# Create a (Receiver) DStream that will connect to localhost:9999
inputDStream = ssc.socketTextStream("localhost", 9999)

resBDStream = inputDStream\
.map(lambda value: int(value))
.reduce(lambda v1,v2: max(v1, v2) )
.filter(lambda value : value>5)
```

```
.window(30, 10)
.reduce(lambda v1,v2: max(v1, v2) )
.filter(lambda value : value>10)


# Print the result
resBDStream.pprint()

ssc.start()
ssc.awaitTerminationOrTimeout(360)
ssc.stop(stopSparkContext=False)
```

(Application C)

```
from pyspark.streaming import StreamingContext

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 10)

# Create a (Receiver) DStream that will connect to localhost:9999
inputDStream = ssc.socketTextStream("localhost", 9999)

resCDStream = inputDStream\
.map(lambda value: int(value))
.filter(lambda value : value>5)
.reduce(lambda v1,v2: max(v1, v2) )
.filter(lambda value : value>10)
.window(30, 10)
.reduce(lambda v1,v2: max(v1, v2) )
.filter(lambda value : value>10)


# Print the result
resCDStream.pprint()

ssc.start()
ssc.awaitTerminationOrTimeout(360)
ssc.stop(stopSparkContext=False)
```

Which one of the following statements is true?

a) Applications A, B, and C are equivalent in terms of the returned result (i.e., they always return the same result independently of the content of the input).

b) Applications A and B are equivalent in terms of the returned result (i.e., A and B always return the same result independently of the content of the input), while C is not equivalent to the other two applications (C can return a different result depending on the content of the input).

c) Applications A and C are equivalent in terms of the returned result (i.e., A and C always return the same result independently of the content of the input), while B is not equivalent to the other two applications (B can return a different result depending on the content of the input).

d) Applications B and C are equivalent in terms of the returned result (i.e., B and C always return the same result independently of the content of the input), while A is not equivalent to the other two applications (A can return a different result depending on the content of the input).

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 06/09/2024 - 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 the name of the city starts with "D"
                if (value.toString().startsWith("D") == 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 numCitiesD
        int numCitiesD;

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

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

        protected void cleanup(Context context) throws IOException, InterruptedException {
                // Emit the pair (numCitiesD, NullWritable))
                context.write(new IntWritable(numCitiesD), 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 (80 bytes – 10 lines)	Beijing Cairo **Delhi** **Dhaka** **Dortmund** Mexico City Mumbai São Paulo Shanghai Tokyo

Cities2.txt (54 bytes – 6 lines)	Buenos Aires Chongqing **Delhi** Istanbul Karachi 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)	1
part-r-00001 (1 line)	2
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)	3
part-r-00001 (1 line)	1
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 (2 lines)	2
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 (3 lines)	4
part-r-00001 (0 line – empty file)	
part-r-00002 (0 line – empty file)	

Part II

PoliOnline is an international company that sells items online. To improve the sales and revenue of PoliOnline, a data warehouse has been created to store information about prices and daily sales. Specifically, the following input data sets/files are available for supporting the analyses of interest of the company.

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

      where *ItemID* is the unique identifier of the item, *Name* is the name of ItemID, and *Category* is its category.

    - For example, the following line

      ID1,t-shirt-winter,Clothing

      means that the item with ItemID ***ID1*** is characterized by the name ***t-shirt-winter*** and belongs to the ***Clothing*** category.

    Note that many items can be associated with the same category.

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

      where *ItemID* is the identifier of an item, and *StartingDate* and *EndingDate* are the beginning and end of the period of validity of the price reported in the attribute *Price* for item *ItemID.* The format of StartingDate and EndingDate is “YYYY/MM/DD”.

    - For example, the following line

      ID1,2019/03/01,2020/06/15,10.8

      means the price associated with item ***ID1*** from ***March 1, 2019***, to ***June 15, 2020***, was ***10.8*** euros.

Note that the price of each item 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 item is associated with one single price in each period.

- DailySales.txt
  - DailySales.txt is a textual file containing information about daily sales for each item. DailySales.txt contains historical data about the last 40 years. This file is big and its content cannot be stored in one in-memory Java/Python variable. There is one line for each combination (ItemID, Date) for the last 40 years.
  - Each line of DailySales.txt has the following format
    - ItemID,Date,NumberOfSales

      where *ItemID* is the identifier of an item, *Date* is a date, and *NumberOfSales* is an integer value representing the number of times item *ItemID* was sold on the date *Date*. The format of Date is "YYYY/MM/DD".

    - For example, the following line

      ID1,2019/05/02,1151

      means that on ***May 2, 2019***, the item identified by ***ID1*** was sold ***1151*** times.

Note that there is a many-to-many relationship between items and dates (i.e., the combination of attributes (ItemID, Date) is the "primary key" of DailySales.txt). Each item is associated with all the dates of the last 40 years, and each date of the last 40 years is associated with all items. Even if an item was not sold on a specific date, there is a line for that combination in DailySales.txt anyway, with NumberOfSales set to 0.

Exercise 1 – MapReduce and Hadoop (8 points)

Exercise 1.1

The managers of PoliOnline are interested in performing some analyses about the prices of the items over time.

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

1. *Maximum and minimum price for each item from 2015 to 2023.* The application computes the maximum and minimum prices of each item considering the prices

valid in the period 2015-2023. Store the result in the output HDFS folder (one item per output line associated with its maximum and minimum prices from 2015). Output format: *ItemID, Maximum price from 2015 to 2023, Minimum price from 2015 to 2023.*

Suppose that the input is Prices.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 PoliOnline 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. *Categories with a higher number of sales in 2023 than in 2022*. The first part of this application considers only the years 2022 and 2023. It selects the categories with a total number of sales in 2023 greater than the total number of sales in 2022. Store the selected categories in the first output folder (one category per output line).

2. *For each item, select the dates with an increasing income compared to the previous date*. The second part of this application considers all 40 years of data. It selects, for each item, the dates on which the daily income associated with the item is greater than the daily income of the previous date. The daily income of an item on a specific date is given by the number of sales of that item on that date multiplied by the item's price on that date. Store the result in the second output folder (one pair (ItemID, date with an increasing income compared to the previous date) per output line).

   Suppose the function **nextDate(date)** is provided. Given a date in the format 'YYYY/MM/DD', nextDate(date) returns the next date (again in the format 'YYYY/MM/DD').

   For example, nextDate('2018/04/05') returns the date '2018/04/06'.

   Example Part 2
   Consider a toy example with a few records/lines.

   Suppose there are only two items: ID1 and ID2.

   Suppose the toy example version of DailySales.txt contains the following 10 lines
   - ID1,2019/05/01,500
   - ID1,2019/05/02,1100
   - ID1,2019/05/03,0
   - ID1,2019/05/04,0
   - ID1,2019/05/05,10
   - ID2,2019/05/01,1000
   - ID2,2019/05/02,1000
   - ID2,2019/05/03,1000
   - ID2,2019/05/04,500
   - ID2,2019/05/05,0

Suppose the toy example version of Prices.txt contains the following 3 lines

- ID1,2019/05/01,2019/12/31,10.0
- ID2,2019/05/01,2019/05/02,5.0
- ID2,2019/05/03,2019/12/31,5.5

It follows that the daily incomes associated with items ID1 and ID2 on the five dates of this toy example are

- ID1,2019/05/01,5000
- ID1,2019/05/02,**11000**
- ID1,2019/05/03,0
- ID1,2019/05/04,0
- ID1,2019/05/05,**100**

- ID2,2019/05/01,5000
- ID2,2019/05/02,5000
- ID2,2019/05/03,**5500**
- ID2,2019/05/04,2750
- ID2,2019/05/05,0

In this case, the dates for the two items with an increasing income compared to the previous date are as follows (this is the output of the second part of this application):

- ID1,2019/05/02
- ID1,2019/05/05
- ID2,2019/05/03

- You do not need to write imports. Focus on the content of the main method.
- Suppose both **SparkContext sc** and **SparkSession ss** have already been set.
- **Only if you use Spark SQL**, suppose the first line of all files contains the header information/the name of the attributes. Suppose, instead, there are no header lines if you use RDDs.