Big data processing and analytics

February 5, 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 MapReduce application for Hadoop.

DriverBigData.java

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

public class DriverBigData extends Configured implements Tool {
   @Override
   public int run(String[] args) throws Exception {
     int exitCode;
     Configuration conf = this.getConf();
     Job job = Job.getInstance(conf);
     job.setJobName("Exercise 05/02/2024 - Question 1");

    // Set input and output paths
     FileInputFormat.addInputPath(job, new Path("inputFolder/"));
     FileOutputFormat.setOutputPath(job, new Path("outputFolder/"));

    // Set driver class
     job.setJarByClass(DriverBigData.class);

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

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



    // Set mapper 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,
                                Text value,
                                Context context) throws IOException, InterruptedException {
                // Emit the pair (value, NullWritable)
                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 count
        int count;

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

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

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

Suppose that inputFolder contains the files Cities1.txt and Cities2.txt. Suppose the HDFS block size is 512 MB.

Content of Cities1.txt and Cities2.txt:

Filename (size and number of lines)	Content
Cities1.txt (77 bytes – 10 lines)	Tokyo Delhi Shanghai São Paulo Mexico City Cairo Mumbai Beijing Dhaka

	Osaka
Cities2.txt (56 bytes – 6 lines)	Tokyo Delhi New York City Karachi Buenos Aires Istanbul

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 gen 29 14:01 part-r-00000

-rw-r--r-- 1 paolo paolo 2 gen 29 14:01 part-r-00001

-rw-r--r-- 1 paolo paolo 2 gen 29 14:01 part-r-00002

-rw-r--r-- 1 paolo paolo 0 gen 29 14:01 _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)	7
part-r-00002 (1 line)	4

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

-rw-r--r-- 1 paolo paolo 2 gen 29 14:01 part-r-00000

-rw-r--r-- 1 paolo paolo 2 gen 29 14:01 part-r-00001

-rw-r--r-- 1 paolo paolo 2 gen 29 14:01 part-r-00002

-rw-r--r-- 1 paolo paolo 0 gen 29 14:01 _SUCCESS

The content of the three part files is as follows.

Filename (number of lines)	Content
part-r-00000 (1 line)	5
part-r-00001 (1 line)	7
part-r-00002 (1 line)	4

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

-rw-r--r-- 1 paolo paolo 2 gen 29 14:01 part-r-00000

-rw-r--r-- 1 paolo paolo 2 gen 29 14:01 part-r-00001

-rw-r--r-- 1 paolo paolo 2 gen 29 14:01 part-r-00002

-rw-r--r-- 1 paolo paolo 0 gen 29 14:01 _SUCCESS

The content of the three part files is as follows.

Filename (number of lines)	Content
part-r-00000 (1 line)	10
part-r-00001 (1 line)	6
part-r-00002 (1 line)	0

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

-rw-r--r-- 1 paolo paolo 2 gen 29 14:01 part-r-00000

-rw-r--r-- 1 paolo paolo 2 gen 29 14:01 part-r-00001

-rw-r--r-- 1 paolo paolo 2 gen 29 14:01 part-r-00002

-rw-r--r-- 1 paolo paolo 0 gen 29 14:01 _SUCCESS

The content of the three part files is as follows.

Filename (number of lines)	Content
part-r-00000 (1 line)	16
part-r-00001 (1 line)	16
part-r-00002 (1 line)	16

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

```
package it.polito.bigdata.spark;
import …;

public class SparkDriver {
        public static void main(String[] args) {
                SparkConf conf = new SparkConf().setAppName("Exam 24/02/05");
                JavaSparkContext sc = new JavaSparkContext(conf);


                JavaRDD<String> robotsRDD = sc.textFile("Robots.txt");



                // Compute the number of robots in Robots.txt and print it on stdout
                System.out.println("Robots in Robot.txt: "+robotsRDD.count());

                // Compute the number of distinct countries in Robots.txt and
                // print it on stdout
                System.out.println("Distinct countries in Robot.txt: "+robotsRDD
                                        .map(line -> new String(line.split(",")[1]))
                                        .distinct()
                                        .count());

                JavaRDD<String> countriesRDD = sc.textFile("Countries.txt");

                // Compute the number of countries/lines occurring in Countries.txt
                System.out.println("Countries in Countries.txt:"+countriesRDD.count());

                // Map the content of robotsRDD to (Country, +1)
                JavaPairRDD<String, Integer> CountryOneRobotRDD = robotsRDD
                                .mapToPair(line ->
                                        new Tuple2<String,    Integer>(line.split(",")[1], 1));

                // Map the content of countriesRDD to (Country, +0)
                JavaPairRDD<String, Integer> AllCountriesRDD = countriesRDD
                                .mapToPair(line ->
                                        new Tuple2<String, Integer>(line.split(",")[0], 0));

                // Union CountryOneRobotRDD and AllCountriesRDD
                JavaPairRDD<String, Integer>CountriesValuesRDD =
                                CountryOneRobotRDD.union(AllCountriesRDD);

                // Compute the number of robots per country
                JavaPairRDD<String, Integer> CountriesNumRobotRDD =
                                 CountriesValuesRDD.reduceByKey((v1, v2) -> v1+v2);

                // Compute the maximum number of robots per country and print it on stdout
                int totalRobots = CountriesNumRobotRDD.values()
                                .reduce((v1, v2) -> Math.max(v1, v2));

                System.out.println("Maximum number of robots per country: "+totalRobots);
```

```
        // Close the Spark context
        sc.close();
    }
}
```

Suppose the input files Robots.txt and Countries.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 Robots.txt 1 time.
b) This application reads the content of Robots.txt 3 times.
c) This application reads the content of Robots.txt 4 times.
d) This application reads the content of Robots.txt 5 times.

Part II

AstroPoli is an international organization that gathers observations of Near-Earth Objects (NEOs), which are celestial bodies observed around the planet. The organization provides a set of statistics about observations based on the following input data files, which have been collected in the latest 20 years of activity of the organization.

- Observatories.txt
  - Observatories.txt is a textual file containing the information about the observatories sharing their data with AstroPoli. There is one line for each observatory and the total number of observatories, including also private amateur observatories, is greater than 100,000,000. This file is large and you cannot suppose the content of Observatory.txt can be stored in one in-memory Java variable.
  - Each line in the Observatories.txt has the following format:

    ObservatoryID,Name,Lat,Lon,Country,Continent,Amateur

    where *ObservatoryID* is a unique identifier for the observatory, *Name* is its registered name, *Lat* and *Lon* are the latitude and longitude of the observatory, *Country* and *Continent* are the country and the continent where the observatory is located, respectively, and *Amateur* specifies if the observatory is an amateur one or not (Amateur is a string being either “True” or “False”).
  - For example, the following line

    OB202,Galileo Observatory,41.9028,12.4964,Italy,Europe,False

    means that Observatory **OB202** is named **Galileo Observatory**, it has coordinates **41.9028** and **12.4964**, it is in **Italy** (**Europe**), and it is **not** an amateur observer (**False**).

- NEObjects.txt
  - NEObjects.txt is a textual file containing information about the NEOs monitored by the network observatories. A new line is inserted in NEObjects.txt every time a new NEO is detected. There is only one line for each NEO. NEObjects.txt contains the historical data about the latest 20 years. This file is big and you cannot suppose the content of NEObjects.txt can be stored in one in-memory Java variable.
  - Each line of NEObjects.txt has the following format:

    NEOID,Dimension,MaterialStrength,alreadyFallen

    where *NEOID* is a unique ID associated with the NEO, and *Dimension* and *MaterialStrength* are quantitative information associated with it. *Dimension* and *MaterialStrength* are numerical values ranging from 1 to 10. *alreadyFallen* is a string and indicates whether the object has already fallen (“True”) or not (“False”).
  - For example, the following line

    NEO301,7,9,True

means that the NEO **NEO301** has a dimension of **7**, a strength of **9**, and it has already fallen (**True**).

- Observations.txt
  - Observations.txt is a textual file containing information about the observations gathered by AstroPoli. There is one line for each observation and the total number of observations is greater than 10,000,000,000. This file is large and you cannot suppose the content of Observations.txt can be stored in one in-memory Java variable.
  - Each line of Observations.txt has the following format

    NEOID,ObservatoryID,ObsDateTime,EclipticLat,EclipticLon,EstimatedDistance

  - where *NEOID* is the ID of the observed NEO, *ObservatoryID* is the ID of the observatory in which the observation has been performed, *ObsDateTime* is the UTC timestamp associated with the observation, *EclipticLat* and *EclipticLon* indicate the sky coordinates at which the instruments looked, and *EstimatedDistance* is the estimated distance from the Earth surface in AU (1 Astronomical Unit = conventional distance from Earth to Sun). UTC means that all the timestamps are in the same time-zone reference. The ObdDateTime format is "YYYY-MM-DD**\t**HH:MM:SS".
  - For example, the following line

    NEO301,OB202,2023-07-15 04:30:00,38.9072,-77.0370,1.4

    means that the NEO **NEO301** was observed by observatory **OB202** at an estimated distance from Earth of **1.4 AU** pointing at coordinates **38.9072** and **-77.0370** on July 15, 2023, at 4:30 (***2023-07-15 04:30:00)***.

    One observatory can observe multiple times the same NEO, with a different timestamp, and the same NEO can be observed my multiple observatories simultaneously. An observatory can make more observations associated to different NEOs simultaneously.

Exercise 1 – MapReduce and Hadoop (8 points)

Exercise 1.1

The managers of AstroPoli are interested in performing some analyses about amateur observatories.

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

1. *Countries with the highest number of amateur observatories*. The application considers only the amateur observatories and, for each country, computes the number of amateur observatories. Finally, it finds the country with the highest number of amateur observatories and stores it in the output folder. If more countries are associated with the maximum number of amateur observatories, sort the countries in alphabetical order and select the first. Store the top country and the associated number of amateur observatories in the output HDFS folder (output pair (Country, Number of amateur observatories in that country)).

Suppose that the input is Observatories.txt and it has already been set. Suppose that also the name of the output folder has 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)

You are required to develop a single Spark-based application based on RDDs or Spark SQL to address the following AstroPoli tasks. The application takes the paths of the input files Observatories.txt, NEObjects.txt, Observations.txt, and two output folders (associated with the outputs of the following points 1 and 2, respectively).

1. *Number of observations for the Most Relevant NEOs from 2023*. Considering only the observations starting from 2023, the application aims to calculate the number of observations associated with the Most Relevant NEOs. The Most Relevant NEOs are the NEOs that (i) have not already fallen and (ii) are characterized by a dimension exceeding the average dimension considering all the registered NEOs in the "NEObjects.txt" file. Calculate the number of observations starting from 2023 for each Most Relevant NEO, sort the result by this number in descending order, and store the result in the first HDFS output folder. Consider only the Most Relevant NEOs that have been observed at least one time starting from 2023 in this first part of the application. The first HDFS output folder must contain information in the format "NEOID,Number of observations starting from 2023" for the Most Relevant NEOs (one line for each Most Relevant NEO).

2. *The Most Relevant NEOs observed by few observatories starting from 2023*. The second part of this application considers only the Most Relevant NEOs observed by less than 10 unique observatories starting from the year 2023. For each Most Relevant NEO of that subset, store in the second HDFS output folder its identifier (NEOID) and the identifiers (ObservatoryIDs) of the unique observatories that observed it starting from the year 2023 (one pair (NEOID, ObservatoryID) per output line). For each of the selected Most Relevant NEOs never observed starting from 2023, store the pair (NEOID, "NONE") in the output folder. The output format of each output line is "NEOID, ObservatoryID". Report the string "NONE" instead of the ObservatoryID for each of the selected Most Relevant NEOs never observed starting from 2023.

- 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 all files contains the header information/the name of the attributes. Suppose, instead, there are no header lines if you use RDDs.
- 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".