Big data processing and analytics

February 21, 2025

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 application.

```
tempRDD = sc.textFile("Temperature.txt") \
                    .map(lambda s: int(s))


# Computes the number of lines of Temperature.txt
numLinesTemp = tempRDD.count()


# Select medium temperature values
mediumTempRDD = tempRDD.filter(lambda v: v>16)\
                       .filter(lambda v: v<25)

# Cache mediumTempRDD
mediumTempRDDCached = mediumTempRDD.cache()


# Computes the number of medium temperature values
numMediumTemp = mediumTempRDDCached.count()



# Select the maximum medium temperature value
maxValue = mediumTempRDDCached.reduce(lambda v1, v2: max(v1, v2))


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


# Print on the standard output the computed values
print("Num lines: " + str(numLinesTemp))
print("Num medium temperatures: " + str(numMediumTemp))
print("Max medium temperature: " + str(maxValue))
```

Suppose the input file Temperature.txt is read from HDFS. Suppose this Spark application is executed only 1 time. Suppose mediumTempRDD is small enough to be completely cached into mediumTempRDDCached. Which one of the following statements is **true**?

a) This application reads the content of Temperature.txt 1 time.
b) This application reads the content of Temperature.txt 2 times.
c) This application reads the content of Temperature.txt 3 times.
d) This application reads the content of Temperature.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("MapReduce - 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 2
    job.setNumReduceTasks(2);



    // 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)
            context.write(new Text(value), NullWritable.get());
    }
}
```

ReducerBigData.java

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

    // Define count
    int count;

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

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

            // Consider only the keys starting with "B"
            if (key.toString().startsWith("B")) {
                    sum = 0;

                    for (NullWritable value : values) {
                            sum++;
                    }

                    if (sum >=3) {
                            // 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 (67 bytes – 10 lines)	**Bari** **Bari** **Bari** **Bari** **Beijing** Cairo Mexico City Mumbai Mumbai Mumbai
Cities2.txt (69 bytes – 8 lines)	**Beijing** **Beijing** **Buenos Aires** Chongqing Delhi Dortmund Dortmund Milan

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 0 set 3 14:00 _SUCCESS

The content of the two part files is as follows.

Filename (number of lines)	Content
part-r-00000 (1 line)	2
part-r-00001 (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 0 set 3 14:00 _SUCCESS

The content of the two part files is as follows.

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

part-r-00001 (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 0 set 3 14:00 _SUCCESS

The content of the two part files is as follows.

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

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

The content of the two part files is as follows.

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

Part II

PolitoTV is an international company operating worldwide. PolitoTV is specialized in streaming television series on demand. It manages several television series and has millions of users. Statistics about the television series and the users are computed based on the following input data files, which have been collected in the company's latest twenty years of activity.

- Users.txt
  - Users.txt is a textual file containing information about the users of PolitoTV. There is one line for each user. The total number of users is greater than 200,000,000. Users.txt is large. Its content cannot be stored in one in-memory Java/Python variable.
  - Each line of Users.txt has the following format
    - Username,Name,Surname,City,Country,PricingPlan

      where *Username* is the user's unique identifier, *Name* and *Surname* are his/her name and surname, respectively, *City* and *Country* are the city and country where he/she lives, respectively, and *PricingPlan* is the pricing plan this user subscribed to.
    - For example, the following line

PG1976,Paolo,Garza,Carmagnola,Italy,Standard

means that the name and surname of the user with the username ***PG1976*** are ***Paolo*** and ***Garza***, respectively, he lives in ***Carmagnola*** (***Italy***), and he subscribed to a ***Standard*** pricing plan.

- TVSeries.txt
  - TVSeries.txt is a textual file containing information about the television series (TV series) streamed on PolitoTV. There is one line for each TV series. The total number of television series stored in TVSeries.txt is greater than 200,000. TVSeries.txt is large. Its content cannot be stored in one in-memory Java/Python variable.
  - Each line of TVSeries.txt has the following format
    - TVSID,Title,TVGenre

      where *TVSID* is the TV series's unique identifier, *Title* is its title, and *TVGenre* is its TV series genre.
    - For example, the following line

      TVS10,Friends,Comedy

      means that the TV series with TVSID ***TVS10*** is titled ***Friends*** and is a ***Comedy*** television series.

  Note that each television series is associated with one single TV series genre.

- Episodes.txt
  - Episodes.txt is a textual file containing information about the episodes of the television series. There is one line for each episode. The total number of episodes stored in Episodes.txt is greater than 4,000,000. Episodes.txt is large. Its content cannot be stored in one in-memory Java/Python variable.
  - Each line of Episodes.txt has the following format
    - TVSID,SeasonNumber,EpisodeNumber,Title

      where *TVSID* is the identifier of the TV series the episode belongs to and *SeasonNumber* is the number of the season this episode is part of. *EpisodeNumber* is the number of this episode in the season *SeasonNumber* of the TV series identified by *TVSID*. Each episode is uniquely identified by the triplet (*TVSID, SeasonNumber, EpisodeNumber*), i.e., the triplet (*TVSID, SeasonNumber, EpisodeNumber*) is the "primary key" of this file. Finally, *Title* is the episode's title.
    - For example, the following line

      TVS10,2,7,The One with the Blackout

      means that the ***7th*** episode of the ***2nd*** season of the television series with TVSID ***TVS10*** is titled ***"The One with the Blackout"***.

- UserWatched.txt

- UserWatched.txt is a textual file containing information about who watched which episodes. A new line is inserted in UserWatched.txt every time a user watches an episode. UserWatched.txt contains the historical data about the last 15 years. UserWatched.txt is large and cannot be stored in one in-memory Java/Python variable.
- Each line of UserWatched.txt has the following format
  - Username,StartTimestamp,TVSID,SeasonNumber,EpisodeNumber
    where *Username* is the identifier of the user who started watching the episode identified by the triplet (*TVSID*, *SeasonNumber*, *EpisodeNumber*) at the time *StartTimestamp*. *StartTimestamp* is a timestamp in the format YYYY/MM/DD-HH:MM.
  - For example, the following line
    PG1976,2022/11/07-21:40,TVS10,1,7

    means that the user ***PG1976*** watched the episode identified by the triplet (***TVS10,1,7***) on ***November 7, 2022*** at ***21:40***.

Note that each user can watch many episodes in different timestamps, and each episode can be watched by many users. Moreover, **the same user can watch each episode several times** (a new line associated with a different StartTimestamp is inserted in UserWatched.txt for each visualization of the same episode by the same user). Note that each pair (Username, StartTimestamp) occurs at most one time in UserWatched.txt.

Exercise 1 – MapReduce and Hadoop (8 points)

Exercise 1.1

The managers of PolitoTV are interested in performing some analyses about the pricing plans in France.

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

1. *French cities with more premium than standard users.* The application considers only the French cities and selects the French cities where the number of users who have subscribed to a premium pricing plan (PricingPlan='Premium') is greater than those who have subscribed to a standard pricing plan (PricingPlan='Standard'). Please pay attention that there are several pricing plans (specifically, there are five different pricing plans, and Premium and Standard are only two of those pricing plans). The selected French cities are stored in the output HDFS folder.

   Output format (one line per each selected French city):
   city

Suppose that the input is Users.txt and 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 (**Exercises 1.2 and 1.3)** to specify the number of instances of the reducer class for each 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"

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 (19 points)
The managers of PolitoTV 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 input files and two output folders (associated with the outputs of the following parts 1 and 2, respectively).

1. *Number of long TV series for each TV series genre*. The first part of this application computes the number of long TV series for each TV series genre. A TV series is classified as a long TV series if it lasts more than 10 seasons (i.e. if it is characterized by more than 10 seasons). Store the result in the first HDFS output folder. Specifically, store one TV series genre per output line with its number of long TV series. **Consider only the TV series genre with at least one long TV series**.

   Output format of each output line (first part):

   TV series genre, Number of long TV series for this TV series genre

2. *The longest TV series each user has started watching.* For each user, the second part of this application selects the longest TV series in terms of number of seasons among those the user has started watching. A user has started watching a TV series if *he/she has watched the first episode of the first season of that TV series* (i.e., the episode of TV series with SeasonNumber=1 and EpisodeNumber=1). Analogously to the first part, the "length" of a TV series corresponds to the number of seasons of that TV series. Store the result in the second HDFS output folder. Specifically, there is one output line for each combination (username, TVSID of the longest TV series username has started watching).

   Output format of each output line (second part):

   Username, TVSID of the longest TV series username has started watching

   Note. Pay attention that there can be more than one TV series associated with the same username (i.e., the user can have started watching more TV series associated with the maximum number of seasons associated with that user).

   Note. Consider only those users who have started watching at least one TV series.

   Example for the second part.

   In this small example, suppose there are only three users. The usernames of these users are User1, User2, and User3.

   Suppose that User1 has started watching the following three TV series
   - TVS10. TVS10 is **10 seasons** long.
   - TVS20. TVS20 is 3 seasons long.
   - TVS25. TVS25 is 2 seasons long.

   Suppose that User2 has started watching the following four TV series
   - TVS9. TVS9 is **6 seasons** long.
   - TVS11. TVS11 is 3 seasons long.
   - TVS45. TVS45 is **6 seasons** long.
   - TVS100. TV100 is 5 seasons long.

   Suppose that User3 has not started watching TV series so far.

   The second output folder must contain the following lines in this case:
   - User1,TVS10
   - User2,TVS9
   - User2,TVS45

   Note that User2 is associated with two output lines because there are two TV series User2 has started watching associated with 6 (and 6 is the number of seasons of the longest TV series User2 has started watching).

Note that User3 is not part of the output of the second part of this application because User3 has not started watching TV series so far.

- You do not need to write imports. Focus on the content of the main method.
- **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.
- Suppose both Spark Context ***sc*** and SparkSession ***spark*** have already been set.
- 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)