# Distributed architectures for big data processing and analytics

July 5, 2021

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 following piece of a Spark application:

   ```
   from pyspark.streaming import StreamingContext
   ssc = StreamingContext(sc, 10)
   lines = ssc.textFileStream("StreamingFolder")
   counts = lines.map(lambda l: int(l)).reduce(lambda a, b: a+b)
   counts.saveAsTextFiles("StreamingOutput", "txt");
   ssc.start()
   ssc.awaitTermination()
   ```

   Which one of the following statements is true?

   a) The code is counting the number of words in each batch of the DStream “lines”

   b) A new batch of the Dstream “lines” is created every 5 seconds

   c) The output of each batch does not depend on the content of the previous batches

   d) The output of each batch depends on the content of the previous batches

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

   **DRIVER**
   import …

```
public class DriverBigData extends Configured implements Tool {

	public int run(String[] args) throws Exception {
		Path inputPath, outputDir;
		inputPath = new Path(args[0]);
		outputDir = new Path(args[1]);

		Configuration conf = this.getConf();
		Job job = Job.getInstance(conf);
		job.setJobName("Exam");
		FileInputFormat.addInputPath(job, inputPath);
		FileOutputFormat.setOutputPath(job, outputDir);

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



		// Set map class
		job.setMapperClass(MapperBigData.class);
		// Set map output key and value classes
		job.setMapOutputKeyClass(NullWritable.class);
		job.setMapOutputValueClass(IntWritable.class);

		// Set combiner class
		job.setCombinerClass(CombinerBigData.class);

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

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

		// Set number of reducers
		job.setNumReduceTasks(1);

		// Execute the job and wait for completion
		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, // Input key type
                Text, // Input value type
                NullWritable, // Output key type
                IntWritable> {// Output value type

        protected void map(LongWritable key, // Input key type
                        Text value, // Input value type
                        Context context) throws IOException, InterruptedException {
                // Read one integer from each input line
                int number = Integer.parseInt(value.toString());
                // emit the pair (NullWritable, number)
                context.write(NullWritable.get(), new IntWritable(number));
        }
}
```

**COMBINER**

```
import ….

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

        protected void reduce(NullWritable key, // Input key type
                        Iterable<IntWritable> values, // Input value type
                        Context context) throws IOException, InterruptedException {
                // Iterate over the set of values and emit them.
                for (IntWritable number : values) {
                        // Emits pair (NullWritable, number)
                        context.write(NullWritable.get(), new IntWritable(number.get()));
                }
        }
}
```

**REDUCER**

```
import …
```

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

        protected void reduce(NullWritable key, // Input key type
                        Iterable<IntWritable> values, // Input value type
                        Context context) throws IOException, InterruptedException {
                int sum = 0;
                // Iterate over the set of values and sum them.
                for (IntWritable number : values) {
                        sum = sum + number.get();
                }
                // Emits pair (NullWritable, sum)
                context.write(NullWritable.get(), new IntWritable(sum));
        }
}
```

Which one of the following statements is true?

a) The application works properly and returns the sum of the input values but the combiner is useless because it does not reduce the amount of data that is sent on the network.

b) The application works properly, returns the sum of the input values and the combiner is useful because it reduces the amount of data that is sent on the network.

c) The application works properly and returns the sum of the input values only if the combiner is executed but the combiner does not reduce the amount of data that is sent on the network.

d) The application does not work properly and raises an error at runtime also if the combiner is instantiated.

# Part II

PoliMarket is an international company that sells items online. To improve the number of sales and revenue of PoliMarket, a set of statistics about the managed items and customers are computed based on the following input data sets/files.

- Items.txt
  - Items.txt is a textual file containing the information about the items that are sold by PoliMarket. There is one line for each item and the total number of items is greater than 1,000,000. This file is large and you cannot suppose the content of Items.txt can be stored in one in-memory python variable.
  - Each line of Items.txt has the following format
    - ItemID,Name,RecommendedPrice,Category

      where *ItemID* is the item unique identifier, *Name* is the name of *ItemID*, *RecommendedPrice* is the recommended sale price of *ItemID* and *Category* is its category (i.e., the item category).

    - For example, the following line

      *ID1,t-shirt-winter,10.5,Clothing*

      means that the item with id ***ID1*** is characterized by the name ***t-shirt-winter***, its recommended price is ***10.5*** euro and it belongs to the ***Clothing*** category.

- Customers.txt
  - Customers.txt is a textual file containing the information about the customers who are registered on the web site of PoliMarket. There is one line for each customer and the total number of customers is greater than 10,000,000. This file is large and you cannot suppose the content of Customers.txt can be stored in one in-memory python variable.
  - Each line of Customers.txt has the following format
    - Username,Name,Surname,DateOfBirth

      where *Username* is the customer unique identifier, *Name* and *Surname* are his/her name and surname, respectively, and DateOfBirth is his/her date of birth. The DateOfBirth format is "YYYY/MM/DD".

    - For example, the following line

      *User20,Paolo,Garza,1976/03/01*

      means that the name and surname of customer ***User20*** are ***Paolo*** and ***Garza***, respectively, and that the customer was born on March 1, 1976.

- Ads_Purchases.txt
  - Ads_Purchases.txt is a textual file containing the information about which items, and when, were proposed through advertisements to each customer and how many of those advertisements were converted in a purchase of the advertised items. A new line is inserted in Ads_Purchases.txt every time a new advertisement is shown to a customer. Ads_Purchases.txt contains the historical data about the last 10 years. This file is big and you cannot suppose the content of Ads_Purchases.txt can be stored in one in-memory python variable.
  - Each line of Ads_Purchases.txt has the following format
    - Timestamp,Username,ItemID,Purchased,SalePrice

      where *Timestamp* is the timestamp at which an advertisement about the item *ItemID* has been shown to customer *Username*. *Purchased* is a Boolean variable that is set to "true" if the customer *Username* purchased the item *ItemID* after the visualization of the advertisement. Otherwise, *Purchased* is set to "false". *SalePrice* is the price at which *Username* bought I*temID.* SalePrice is set to 0 if *Purchased* is "false"*.*

    - For example, the following line

      *2019/02/02-09:15:01,User20,ID1*,*true,50.99*

      means that on ***February 2, 2019***, at ***09:15:01*** an advertisement about the item identified by ***ID1*** was shown to the customer ***User20***, and then ***User20*** bought that item at ***50.99*** euro. The format of timestamp is "YYYY/MM/DD-HH:MM:SS".

      Note that the same advertisement can be shown to different users at the same timestamp, the same advertisement can be shown to the same user in many different timestamps, and many different advertisements can be shown to the same customer, also at the same timestamps.

**Exercise 1 – MapReduce and Hadoop** (7 points)

The managers of PoliMarket are interested in performing some analyses about the effectiveness of their advertisements.

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

1. *Advertised items with a high percentage of conversions in the year 2020*. The application considers only the advertisements related to the year 2020 (i.e., those with timestamps related to the year 2020) and selects the items associated with a conversion rate greater than 0.1%. The conversion rate of an item is given by the ratio between the number of advertisements related to that item which are also associated with a purchase (i.e., the number of lines in Ads_Purchases.txt associated with that item having *Purchased* equal to “true”) divided by the number of advertisements related to that item (i.e., the number of lines in Ads_Purchases.txt associated with that item). Pay attention that you must compute the conversion rate considering only the data related to the year 2020. Store in the output HDFS folder the identifiers (ItemIDs) of the selected items and their conversion rate in the year 2020 (i.e., the output contains one line for each of the selected items and the format is ItemID\t percentage of conversions in the year 2020).

Suppose that the input is Ads_Purchases.txt and it has been already set and also the name of the output folder has been already 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 next 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:

    - 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 get and set methods. I suppose they are "automatically defined"

**Exercise 1 - Number of instances of the reducer - Job 1 - MapReduce and Hadoop** (0.5 points)

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

(a) 0

(b) exactly 1

(c) any number >=1

**Exercise 1 - Number of instances of the reducer - Job 2 - MapReduce and Hadoop** (0.5 points)

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

(a) One single job is needed for this MapReduce application

(b) 0

(c) exactly 1

(d) any number >=1

**Exercise 2 – Spark** (19 points)

The managers of PoliMarket are interested in performing some analyses related to their items, customers, and the effectiveness of their advertisements.

The managers of PoliMarket asked you to develop one single application to address all the analyses they are interested in. The application has five arguments: the three input files Items.txt, Customers.txt, Ads_Purchases.txt and two output folders, "outPart1/" and "outPart2/", which are associated with the outputs of the following Points 1 and 2, respectively.

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

1. *Items that are frequently sold at a price greater than the recommended one.* The application selects the items that in more than 90% of the associated purchases (lines of "Ads_Purchases.txt" with Purchased equal to "true") are characterized by a price of sale (SalePrice) greater than the recommended price (RecommendedPrice). The application stores in the first HDFS output folder the identifiers (ItemIDs) of the selected items and their categories (one combination ItemID, Category per output line).

2. *Categories with many unadvertised items and many low-profit items.* The application considers all the data about the displayed advertisements that are in "Ads_Purchases.txt" and considers only the items that are either unadvertised or low-profit items. An item is categorized as an unadvertised item if it was never advertised (i.e., it does not occur in "Ads_Purchases.txt"). An item is categorized as a low-profit item if the profit associated with its advertisements is greater than 0 and below 100 euro. The profit associated with each item is the sum of the values of SalePrice of its advertisements (remember that SalePrice is equal to 0 when Purchased is equal to "false"). The application selects the categories (category is a feature of items.txt) that are associated with (i) at least 10 unadvertised items and (ii) at least 10 low-profit items. The categories that satisfy the reported constraints are stored in the second output folder. For each of the selected categories, the following information is stored in the second output folder: category, number of unadvertised items, number of low-profit items (one of the selected categories and the associated information per output line).

   **Some examples related to Point 2 (second part of the exercise)**

   - Suppose that the category Books includes 15 unadvertised items and 12 low-profit items. The category Books **is selected** because the number of unadvertised items is greater than or equal to 10 and also the number of low-profit items of that category is greater than or equal to 10. The line "Books,15,12" **is stored** in the second output folder.

   - Suppose that the category Clothing includes 4 unadvertised items and 30 low-profit items. Clothing **is not selected** because the number of unadvertised items is less than 10. Clothing **is not stored** in the second output folder.

   - Suppose that the category "Home & Garden" includes 30 unadvertised items and 0 low-profit items. "Home & Garden" **is not selected** because the number

of low-profit items is less than 10. “Home & Garden” **is not stored** in the second output folder.

**Suppose sc (Spark Context) and spark (Spark Session) have been already set.**

```
itemsPath= 'Items.txt'
customersPath= 'Customers.txt'
adsPath= 'Ads_Purchases.txt'

output1 = 'out1'
output2 = 'out2'
```