Systems for Big Data Processing - Project 1 Natural Disasters Analysis

1st Rafaela Cruz
nº 56926
rc.cruz@campus.fct.unl.pt

2nd André Bastos
nº 56969
afn.bastos@campus.fct.unl.pt

I. Introduction

In this project, we extracted some information on natural disasters, using three different technologies: Map-Reduce, Spark and Spark DataFrame. In this report, we describe our rational for extracting the required information, as well as the time it took to process the complete data set.

II. Map-Reduce

A. Number of disasters per continent

Our first goal was to determine how many disasters occurred in each continent. To do that, we created a mapper function that receives as input our CVS file and emits intermediate tuples (key, value). The keys and values chosen for these tuples were, respectively, the continent and the number of occurrences of some natural disaster.

Between the map and the reduce, the tuples are shuffled and sorted by the keys, so that each pair with the same key is sent to the same reduce. [1] The sorted tuples are then sent to the reducer, which is a function that counts the occurrences of natural disasters in each continent. Our reducer function consists in iterating through each key and checking if the current key is equal to the last key checked. If the key is the same as the last one, we increment the count of occurrences. If the keys are different, we change the last key to the one being currently checked. Note that this solution only works because the tuples are grouped by the keys.

B. Regions with disasters of type X

To create an index with every type of disaster and the regions in which each type occurred, we started by creating a mapper function that emits a tuple with the type of disaster as key and the region as value.

The tuples are then sorted by their keys, so the keys that are equal are grouped together. [1] As we did before, we created a reducer function that iterates through each key and checks if each key is equal to the last one. If the key is the same as the last one, we check if the region for that key is already in a list of regions that we had created before, and add that region to the list otherwise.

The output for this program was an index with each type of disaster and a list of regions in which that disaster occurred.

C. Probabilities of getting injured or dying

In this exercise, the objective was to compute the probability of getting injured (number of injured divided by the total affected) and the probability of dying (number of deaths divided by the total affected) in a disaster, for every combination of decade, disaster type and continent.

The first thing we did in the mapper function was to filter the data rows missing some of the required information, so that we wouldn't have empty probabilities.

Since we need to compute the probabilities for every disaster type, continent and decade, we decided to use these three columns as the key. Note that the column with the date contains years, so the last digit was replaced by a 0 to represent a decade. This means that all rows that fit a specific combination of these features are going to end up together when sorted and be used as input for the same reduce. For the value, the columns chosen were the necessary information for computing the probabilities (the number of deaths, injured and total affected).

In the reducer function, each tuple is parsed into key and value, and then the values are parsed into deaths, injured and total affected. Then, we iterate through each key and check if the current key is equal to the last key checked. If the key is the same as the last one, we increment the death, injured and total affected variables with the information received in the value. If the keys are different, we compute the probabilities for the old key and print them. Furthermore, we change the last key to the one being currently checked and set the values of the variables to the ones corresponding to the new key.

III. Spark

A. Number of disasters per continent

Using Spark, the first task to do was to import the input CVS file and then separate every line by its columns. Then, we limited the information to only what we needed. In this case, only the continent and the occurrences were required, thus they were retrieved together in a key value fashion. After that, we applied the reduceByKey operation provided by Spark, that aggregates the keys (continents) and applies a user defined function to the respective values, to sum the occurrences of disasters per continent. [2] This results in a RDD with the continents and the total number of occurrences per continent. The only thing left to do is to print every row.

B. Regions with disasters of type X

In this case, the parsing was done in the same way as in the exercise before, but we chose to retrieve the columns corresponding to the disaster type and the region. The list of regions we want to obtain requires only unique values, so we used the distinct operation to discard repetitions. [2] Then, using a groupByKey operation, we aggregated all values (regions) that had the same key (types), turning this set of regions into an iteration spark object. We turned these objects into regular lists, using the map operation, and, for every disaster type, we printed the list of regions obtained. [3]

C. Probabilities of getting injured or dying

This exercise requires the probability of getting injured and the probability of dying in a disaster for each combination of decade, disaster type and continent. First of all, we applied a filter to discard every row that did not contain the desired information (rows with empty columns).

To operate with the deaths information, we isolated it as values in an RDD with key, value rows. The key chosen was a string containing the decade (year column with last digit as 0), disaster type and continent. The same was done for the injured and total affected columns. This is useful to sum the number of deaths, injured and affected for every combination of keys, which was done using the reduceByKey operation.

To obtain the probability of dying in a natural disaster, we joined the RDD containing the summed deaths with the RDD containing the summed affected. The resulting RDD had the same keys as before, but had as values a tuple with the total deaths and total affected. We applied a map operation to this RDD, leaving as values the division of the first element of the tuple by the second element of the tuple. The same was done for the injured and total affected to obtain the probability of getting injured in a disaster.

Using the same key, we joined both RDDs containing the probabilities with a join operation, and printed every row of this new RDD.

IV. Spark DataFrame

A. Number of disasters per continent

To determine the number of disasters for each continent using Spark DataFrame, we imported the input CVS file and separated every line by its columns. After that, we created a data frame with the columns corresponding to the continents and the occurrences of disasters.

Using this data frame, we grouped together the rows with the same continent and applied a sum operation to the rows that were grouped (i.e. to the rows that had the same continent). We added this new column with the summed occurrences by continent to the column with the continents and printed the resulting table. [4]

B. Regions with disasters of type X

In this exercise, we parsed the CVS file as before, and created a data frame with the columns corresponding to the disaster type and the region. Using this data frame, we grouped together the rows with the same disaster type and collected to a list the regions in which that disaster type had occurred. To do that, we used the operation collect_set, that collects only the unique values to a list. [5] We then added these lists of regions for each disaster type to the column with the types of disasters and printed the resulting table.

C. Probabilities of getting injured or dying

As we did in the previous technologies, in this exercise, we applied a filter to discard every row that did not contain the desired information (rows with empty columns). We then created a data frame with the columns corresponding to decade (year column with last digit as 0), continent, disaster type, number of deaths, number of injured and number of total affected.

With this new data frame, we grouped together all the rows with the same combination of decade, disaster type and continent, and summed the deaths, the injured and the total affected columns of these rows. After that, we created a new column, corresponding to the summed deaths divided by the summed total affected, i.e., with the probability of dying for each combination of decade, disaster type and continent. We did the same for the probability of getting injured. [5]

In the end, we dropped the old columns with the deaths, injured and total affected, printing a table with the decade, disaster type, continent, probability of dying and probability of getting injured.

V. Times

The time it took to execute our solutions for each technology and each exercise was recorded and is shown in Table I.

TABLE I
TIMES OBTAINED FOR EACH TECHNOLOGY IN EACH EXERCISE

	Disasters per continent		**Region by disaster type**		**Probabilities**	
Time (s)	*real*	*user + sys*	*real*	*user + sys*	*real*	*user + sys*
Map-Reduce	15.741	26.700	15.510	26.826	13.200	24.197
Spark	13.602	0.736	13.191	0.754	24.210	0.846
Spark DataFrame	24.151	1.392	23.371	1.196	19.871	1.250

Observing Table I, we see that, for the first two exercises, Spark is the technology with the shortest *real* time. This is because Spark provides in-memory processing whereas Map-Reduce uses the local disk. [2] Furthermore, in comparison to Spark DataFrame, Spark is a lower-level framework and thus, if the optimizer in Spark DataFrame is not applied, Spark will be more efficient. [2] [4] In the last exercise, Map-Reduce has the shortest *real* time.

Note that the *user + sys* time should be higher than the *real* time, which was not the case for neither Spark nor Spark DataFrame. We had a talk with the teacher about it, but we could not find an answer for this problem.

VI. Optional Exercise

For the optional exercise, we decided to study the damage caused by natural disasters. For that purpose, we computed the mean and the maximum damage caused by each disaster subgroup in every country.

A. Map-Reduce

Being the relevant information the expanses of the disasters, we applied a filter to ignore all rows that did not contain this information. The key chosen for our exercise was the country and the disaster subgroup and the values were the total damage and the occurrences.

In the reducer function we iterate through each key and check if it is the same as the last one. If they are the same, the variables that hold the damage and occurrences are increased by the new iteration's values. Since we also want to know the maximum a country wasted on a disaster subgroup, we check if the damage in the new value is higher than in the last one and, if this is the case, we update the maximum with this new value.

If the new key is not the same as the last one, the mean damage for the previous iteration is computed and then printed with the maximum damage. Afterwards, the values are refreshed by the new iteration. Finally, we need to compute the mean and print the results for the last iteration.

The times obtained for this exercise were 10.999 s for *real* and 19.366 s for *user + sys*.

B. Spark

Using Spark, the first thing we did was to split each row by its columns and filter the rows that had the column corresponding to the total damage empty. Then, we created two RDDs: the first one using the country and the disaster subgroup columns as key and the total damage column as value; and the second one using the same key and with the occurrences column as value.

To obtain the total damage for every combination of country and disaster subgroup, we applied a reduceByKey operation to the first RDD to sum its values by key. Then, we did the same with the second RDD. Afterwards, to compute the mean damage, we joined the two RDDs and applied a map operation to divide the summed total damage by the summed occurrences.

To compute the maximum damage caused by a disaster subgroup in a given country, we applied a reduceByKey operation to the RDD containing the total damage to find the maximum value for each key. In the end, we joined the RDD containing the mean damage to the one containing the maximum damage and printed the results.

The times obtained for this exercise were 21.687 s for *real* and 0.832 s for *user + sys*.

C. Spark DataFrame

In Spark DataFrame, we applied the same filter for the rows with an empty damage column, and created a data frame holding only the relevant columns: country, disaster subgroup, total damage and occurrences.

Then, we grouped the data by their country and disaster subgroup, applying a sum operation to the total damage and to the occurrences and also creating a new column (max_damage) with the function max, that will hold the maximum value of the damage for each country and disaster subgroup.

Afterwards, we created a new column with the objective of holding the average damage caused by each disaster subgroup, applying a division between the summed total damage and the summed occurrences columns.

Before we showed the final result, we dropped the now useless columns with the total damage and the occurrences.

The times obtained for this exercise were 17.557 s for *real* and 1,167 s for *user + sys*.

Acknowledgements

We would like to thank the teacher for restlessly answering our relevant questions and for creating useful exercises in the practical classes that allowed us to understand and solve this more complex project.

References

[1] Lourenço, João. "Map-reduce" Systems for Big Data Processing, Oct 3, 2019, FCT-UNL.

[2] Preguiça, Nuno, Lourenço, João. "Spark" Systems for Big Data Processing, Oct 11, 2019, FCT-UNL.

[3] Stack Overflow: PySpark groupByKey returning pyspark.resultiterable.ResultIterable. Retrieved from https://stackoverflow.com/questions/29717257/pyspark-groupbykey-returning-pyspark-resultiterable-resultiterable. Last accessed in Oct 27, 2019.

[4] Preguiça, Nuno, Lourenço, João. "Spark SQL" Systems for Big Data Processing, Oct 11, 2019, FCT-UNL.

[5] Spark.apache.org: pyspark.sql module - PySpark 2.4.4 documentation. Retrieved from https://spark.apache.org/docs/latest/api/python/pyspark.sql.html. Last accessed in Nov 9, 2019.