**CENG 790: Big Data Analytics, Fall 2020**

**Report of Assignment 1: Introduction to Spark**

**Part 1: Processing data using the DataFrame API**

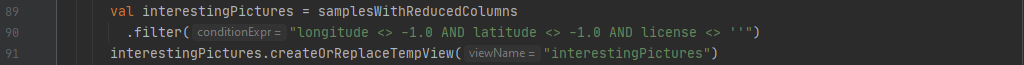
1. Using the Spark SQL API (accessible with spark.sql("...")), select fields containing the identifier, GPS coordinates, and type of license of each picture.





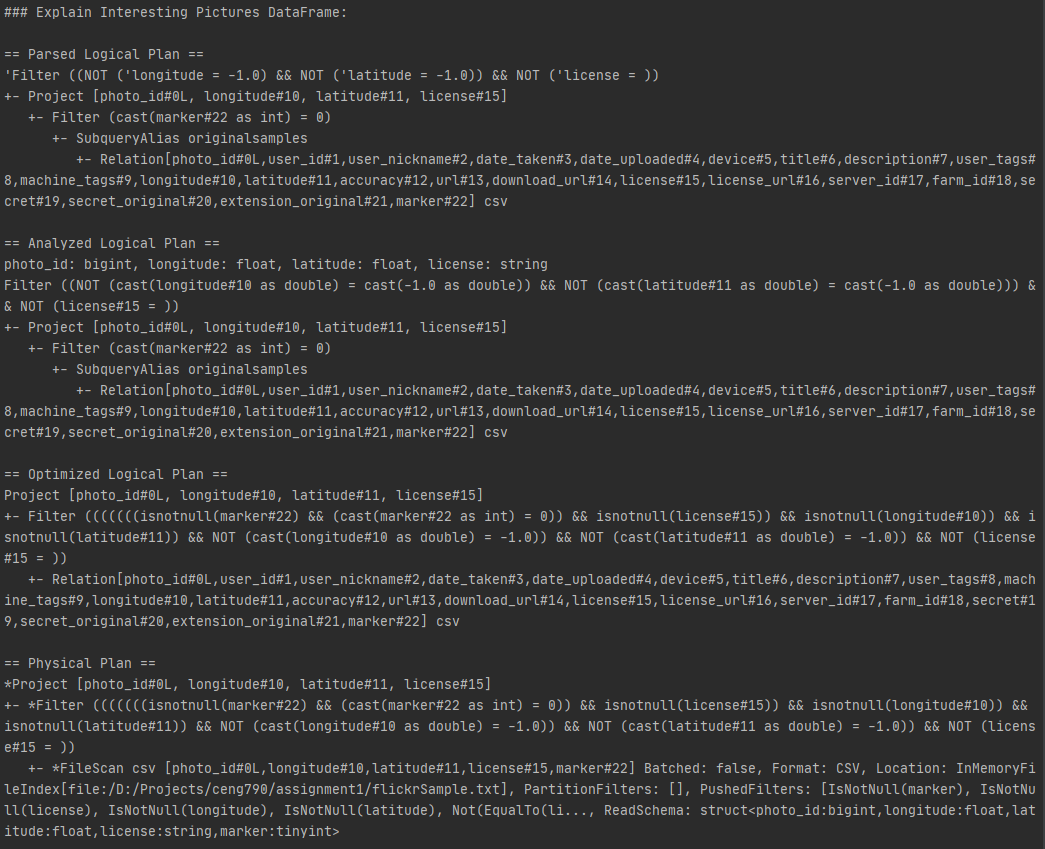
In order to get only pictures, *marker* field must be 0.

1. Create a DataFrame containing only data of interesting pictures, i.e. pictures for which the license information is not null, and GPS coordinates are valid (not -1.0).



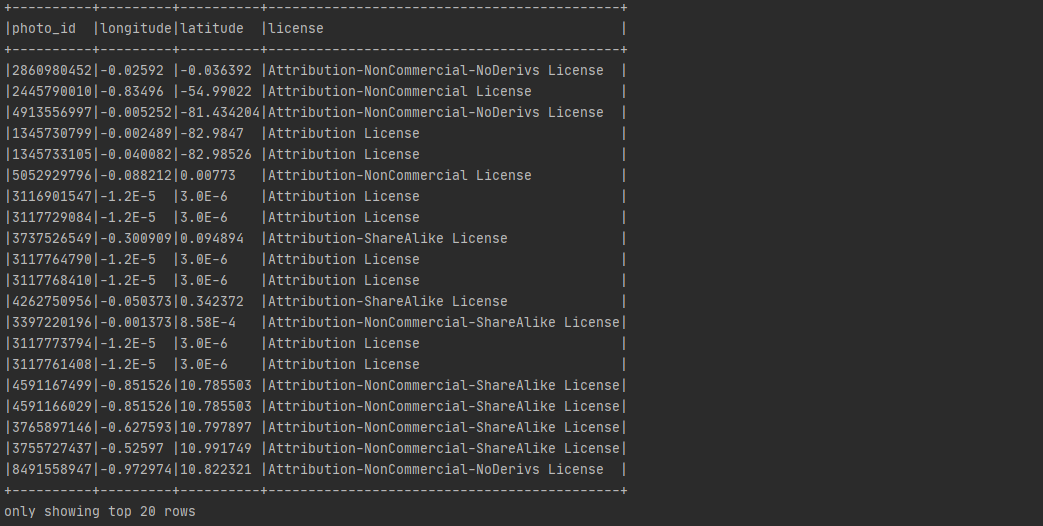
1. Display the execution plan used by Spark to compute the content of this DataFrame(explain()).





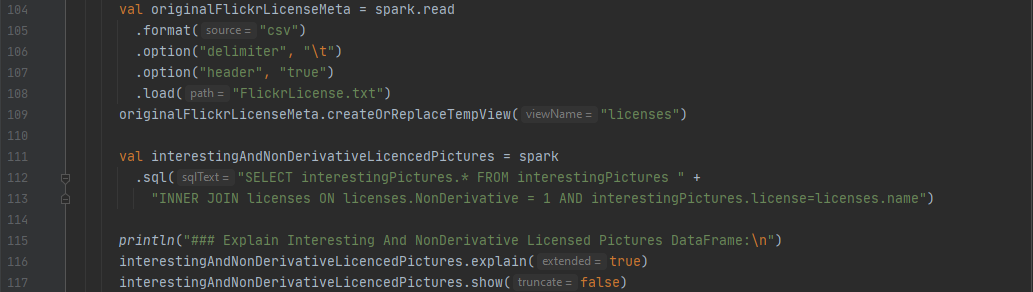
1. Display the data of this pictures (show()). Keep in mind that Spark uses lazy execution, so as long as we do not perform any action, the transformations are not executed.

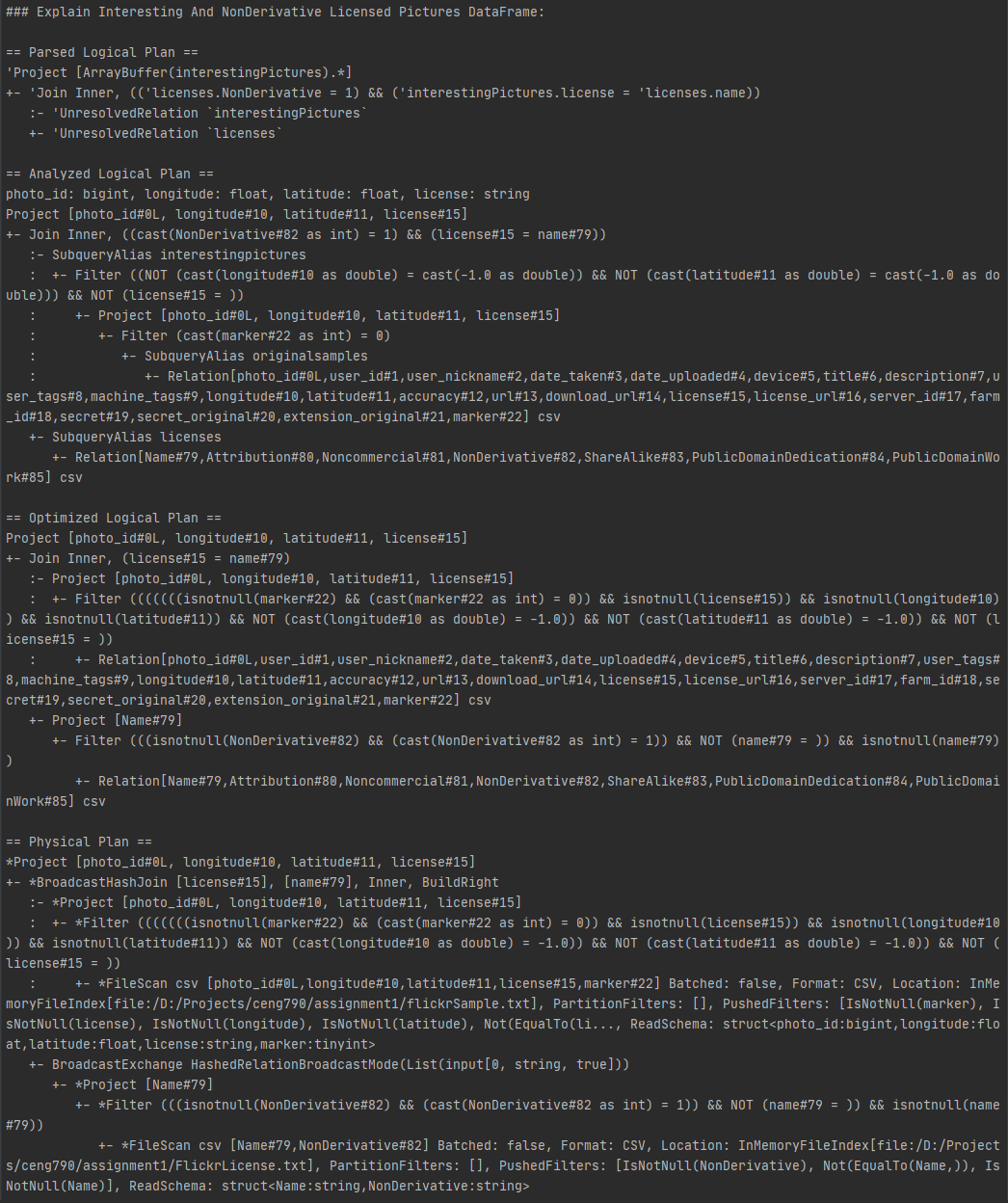


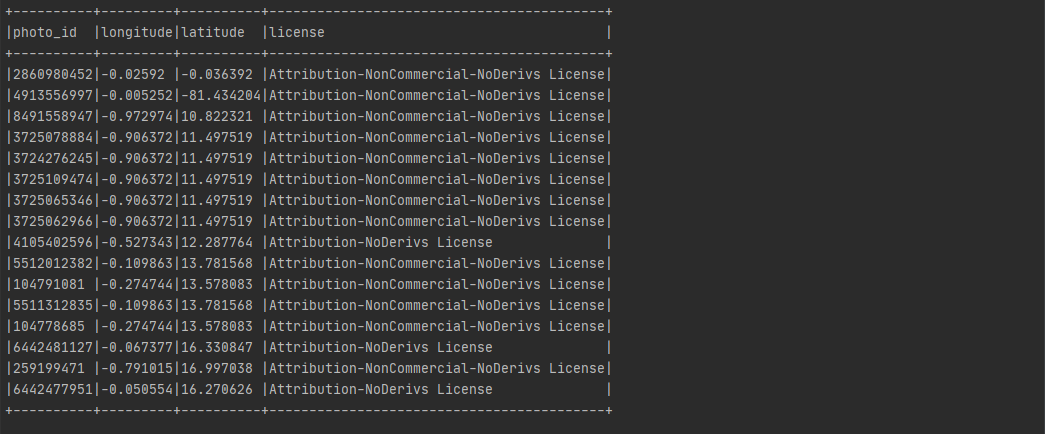


Displayed rows are truncated to 20 rows even though I used *df*.show(truncate = false) to display every row.

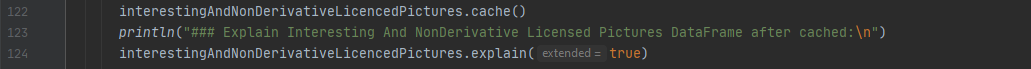
1. Our goal is now to select the pictures whose license is NonDerivative. To this end we will use a second file containing the properties of each license. Load this file in a DataFrame and do a join operation to identify pictures that are both interesting and NonDerivative. Examine the execution plan and display the results.

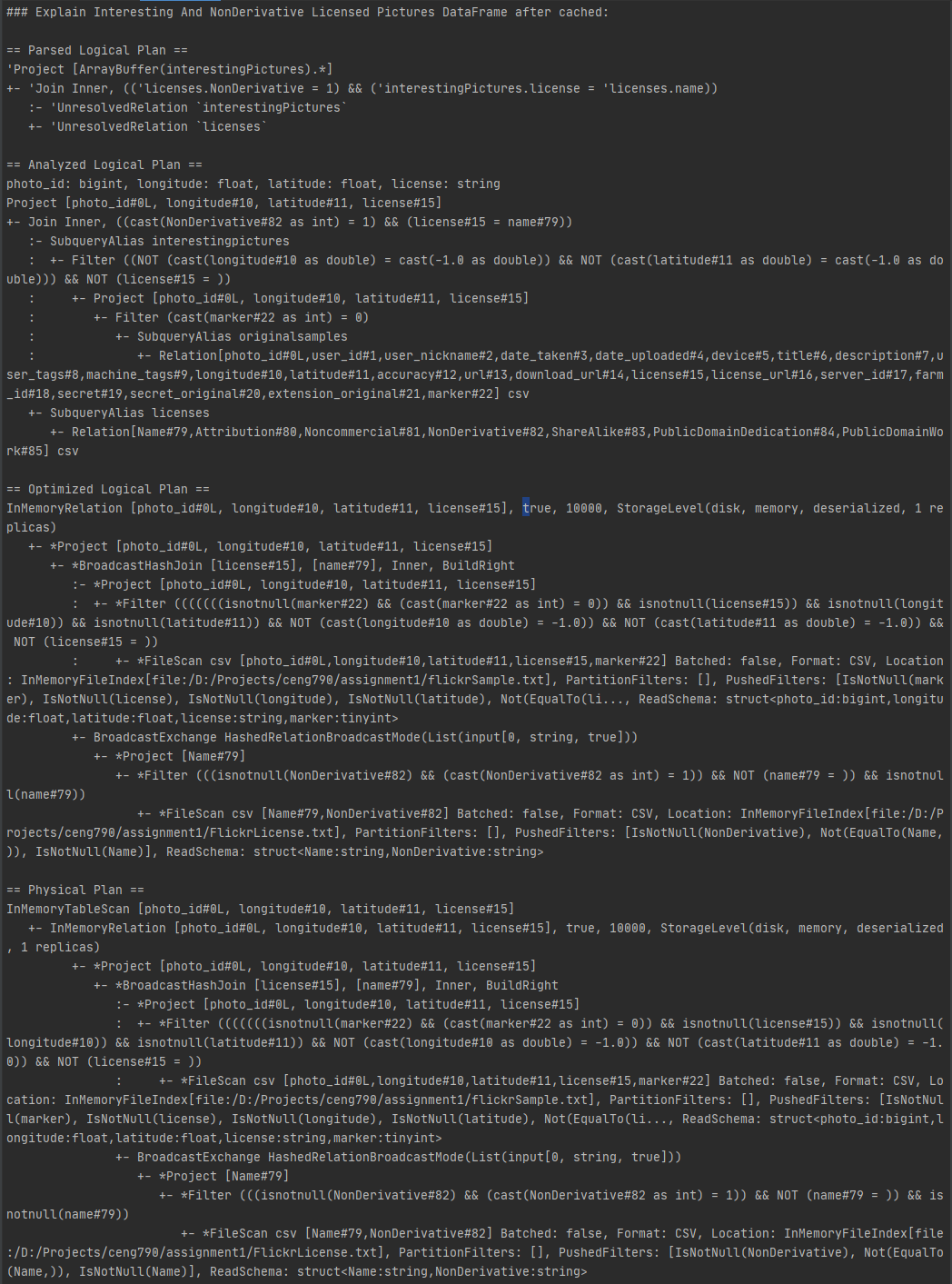






1. During a work session, it is likely that we reuse multiple time the DataFrame of interesting pictures. I would be a good idea to cache it to avoid recomputing it from the file each time we use it. Do this, and examine the execution plan of the join operation again. What do you notice?

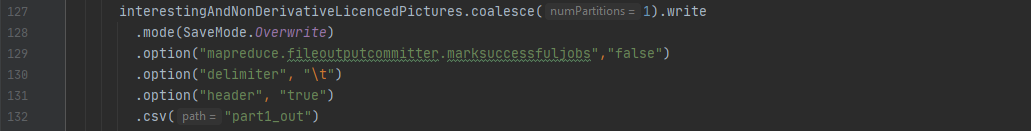




Parsed Logical Plan and Analyzed Logical Plan are the same in both DataFrames’ execution plans. However, Optimized Logical Plans and Physical Plans of cached and uncached DataFrames have differences such as:

* Cached DataFrame have “**\***” characters (i.e. \*Project, \*Filter) in Optimized Logical Plan. I am not sure but “\*” character might mean that the transformations are already executed.
* Cached DataFrame’s Optimized Logical Plan and Physical Plan has lines starting with “*InMemoryRelation*” and “*InMemoryTableScan*” which can be interpreted as the table and the relation is retrieved from the memory.
* Cached DataFrame’s Optimized Logical Plan have “*BroadcastHashJoin*” and “*BroadcastExchange*” instead of “*Join*”, “*FileScan*” instead of “*Relation*”.
* Cached DataFrame’s Physical Plan is almost identical to the uncached DataFrame’s plan. Only difference is the use of “InMemoryTableScan” and “InMemoryRelation”.

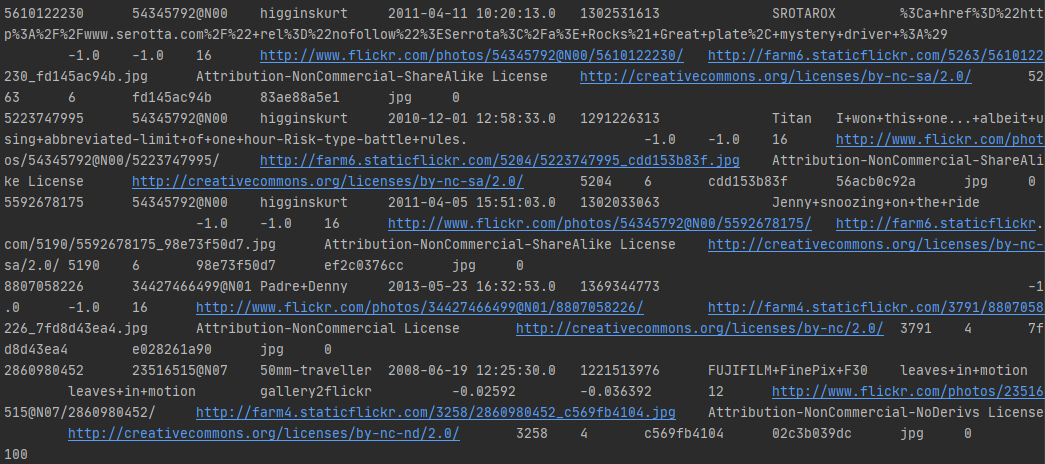
1. Save the final result in a csv file (write). Don’t forget to add a header to reuse it more easily.



**Part 2: Processing data using RDDs**

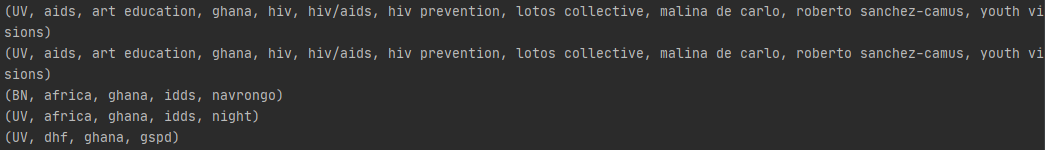
1. Display the 5 lines of the RDD (take(5)) and display the number of elements in the RDD (count()).



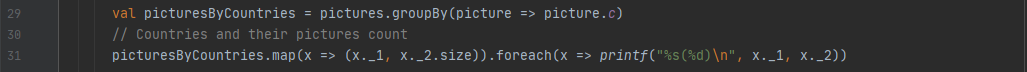


1. Transform the RDD[String] in RDD[Picture] using the Picture class. Only keep interesting pictures having a valid country and tags. To check your program, display 5 elements.





1. Now group these images by country (groupBy). Print the list of images corresponding to the first country. What is the type of this RDD?

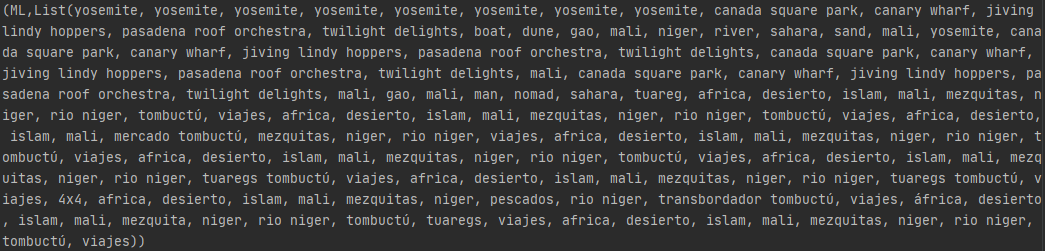




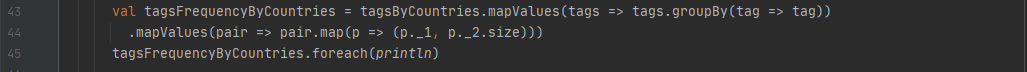
The type of picturesByCountries is RDD[(Country, Iterable[Picture])]. Additionally, I printed picture counts of countries.

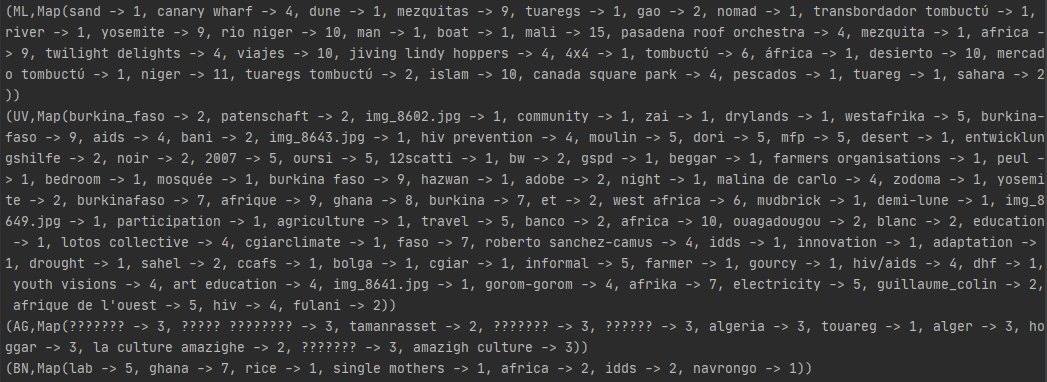
1. We wish to avoid repetitions in the list of tags, and would rather like to have each tag associated to its frequency. Hence, we want to build a RDD of type RDD[(Country, Map[String, Int])]. The groupBy(identity) function, equivalent to groupBy(x=>x) could be useful.





1. We wish to avoid repetitions in the list of tags, and would rather like to have each tag associated to its frequency. Hence, we want to build a RDD of type RDD[(Country, Map[String, Int])]. The groupBy(identity) function, equivalent to groupBy(x=>x) could be useful.





1. There are often several ways to obtain a result. The method we used to compute the frequency of tags in each country quickly reaches a state in which the size of the RDD is the number of countries. This can limit the parallelism of the execution as the number of countries is often quite small. Can you propose another way to reach the same result without reducing the size of the RDD until the very end?

Instead of Creating RDD[(Country, Map[String, Int])], we could create RDD[(Country, String)] where String represents the tag then we could group by each country, tag pair or simply RDD[(Country, String, Int)] where String still represents the tag and Int is the frequency of that tag in the country.