Big Data Management Project - Computer Engineering for the Internet of Things Master’s Degree

Francesco Raco

University of Calabria

Gianmarco Marcello

University of Calabria

A project thought as the practical part of the exam of the Big Data Management course. In the following document, we describe our codebase, containing some applications used to extract information from large csv datasets, using different Big Data Management open-source frameworks. We then discuss our choices of technology stacks, languages and databases used to implement the applications and to store the extracted data. The frameworks are mostly language-agnostic so, our choices of languages are dictated by familiarity and personal knowledge.

CCS CONCEPTS • Data management systems • Information storage systems • Distributed computing methodologies

**Additional Keywords and Phrases:** datasets, map-reduce, big data frameworks, information extraction.

ACM Reference Format:

Francesco Raco, Gianmarco Marcello. 2021. Big Data Management Project - Computer Engineering for the Internet of Things Master’s Degree.

1. Introduction

Nowadays one of the most common tasks to execute in the world of computing consists of working on huge datasets to extract information from them. We tried to approach this problem using different technology stacks, applying specific techniques and frameworks, that allowed us to schedule and execute jobs on cluster of computers. The specific frameworks that we chose to work with were Hadoop Map Reduce and Apache Spark, with Apache Spark Streaming as a tool to approach the streamed data case. We also sought to use different data storage solutions, and we specifically chose HDFS and MongoDB.

1. the project

We chose several datasets and tried to extract information from them using the discussed technologies. The datasets were all covid-related and came from kaggle.com. We used every framework and storage technology and came up with six queries.

* 1. Daily province with most COVID cases for each Italian region – Spark Streaming – MongoDB –Scala

With this query we want to extract useful information about the COVID cases in the various Italian regions, in particular for each day the query return for each region the province that had the greatest number of COVID cases. Such query has been implemented in Scala by using the Spark Streaming framework. Data to be processed is sent to the application (listening on port 9999) through a TCP socket in order to simulate a streaming behavior.

*2.1.1 Data gathering and formatting*

In order to get data to be processed, through the Spark context we create an input stream (Dstream) connected on port 9999 of the localhost (in particular we create a socketTextStream which allows to interpret the received data as UTF8-encoded \n delimited lines). Received lines are formatted into a JavaPairDStream (a Java-like class that allows us to treat the DStream as a set of key-value pairs, indeed using methods like *reduceByKey*) having (Date, Region) as keys and (Province, Total Cases) as values.

*2.1.2 Data processing*

The processing simply consists on two operations. First, the reduceByKey function provided by JavaPairDStream is utilized in order to get for each pair (Date, Region) the province with the highest number of total COVID cases. Then, the result is parsed into a list of strings (Date, Region, Province, Total Cases).

*2.1.3 Execution environment and final remarks*

The results are saved on a MongoDB server listen on standard port 27017, instantiated through a Docker image. The application itself creates a connection to such server, and saves results onto a collection “col” in the database “query”.

* 1. Chinese province with highest increase of cumulative deaths, for each month of 2020 – Apache Spark – MongoDB – Scala

The starting point is a dataset covering the period from 22nd January 2020 to 27th February 2021 and providing for each day the cumulative numbers of confirmed cases, deaths and recovered people due to COVID through the globe, mainly focusing on China. The query returns, for each month of 2020, the Chinese province with the highest increase in terms of cumulative deaths with respect to the previous month.

*2.2.1 Data gathering, formatting and processing*

Data is taken through the Spark context from the local file system, where it is stored into a CVS file. Such tuples are then parsed into Scala Tuples (Month, Province, Deaths): each tuple represents the number of cumulative deaths of a given province registered in the last day of a given month. This set of tuples is the input of another last parsing before the processing phase starts. In particular, by means of the flatMap function, each tuple (M, P, D) is converted into two tuples: ((M, M+1, P), D) and ((M-1, M, P), D). such parsing is necessary in order to achieve, in the subsequent reduce, the increases in terms of cumulative deaths: the dataset after the previous parsing will consist on a set of pairs, with each pair differing only in the last vale, the number of cumulative deaths. For better understanding, assume to have the tuples ((2, 3, P), X) and ((2, 3, P), Y). The meaning of such tuples is the following: one represents the cumulative cases on province “p” in the last day of February (2) and the other one (that with the higher value of deaths) represents the cumulative deaths for the same province in the last day of March (3). Since deaths values are cumulative, in the reduce phase by subtracting x and y and by taking the absolute value of the result, we obtain the increase of cumulative deaths in the province p in March with respect to February. Finally, after parsing the dataset into a JavaPairRDD with Month as key and (Province, Deaths) as values, a final reduce is performed in order to take for each month the province with the highest increase.

*2.2.2 Execution environment and final remarks*

The result is stored on MongoDB, with a server listening on standard port 27017 instantiated through a Docker image. as in the previous query, also here a Mongo Client is instantiated and a connection to the proper database is created, then the results are stored into the proper collection.

* 1. Italian region with the highest percentage of intensive care patients, for each day – Hadoop Map Reduce – HDFS – Java

This query operates on a dataset providing daily COVID data regarding Italy’s regions, from 24th February to 6th December of 2020. For each day, such query returns the region with the highest percentage of intensive care patients over total hospitalized patients. The processing consists of only one stage, so just one Mapper and one Reducer have been implemented for carrying out the Map Reduce computation. In order to improve performances, the Reducer class has been set as Combiner class.

2.3.1 Mapper Implementation

The mapper performs on each row of the dataset the following operations. First, it converts such row into an array of strings using “,” as argument of the split operation. Then, it takes from such array the number of intensive care patients and the number of total hospitalized patients and calculates the percentage. Finally, it calculates its final result: both the key and the value are Text objects, with the key being the Date and the value the couple (Region, Percentage).

2.3.2 Reducer implementation

The Reducer work is quite simple: it checks all the values related to a given key (which is a date) and extracts the couple (Region, Percentage) containing, for that date, the name of the region and the percentage of intensive care patients. During the processing, the Reducer checks also the presence of some random “%” that has been found in the dataset and that creates problems during the computation of the result. The reducer finally both the key and the value as Text Objects.

2.3.3 Execution environment and final remarks

In this query HDFS has been used for both retrieving data and to store the result. In particular, through the effeerre/Hadoop Docker image a Docker Network of one master node and three slave nodes have been created, following all the operations we saw in the practical lectures like, for example, formatting the Namenode and starting YARN.

* 1. Months with most COVID cases by US county – Hadoop Map Reduce – HDFS – Java

We try to extrapolate the list of the months with most COVID cases for each US county. This is done by starting from a dataset containing data about each US county during the pandemic period. The dataset is loaded onto HDFS and is then passed through the Map Reduce procedure; its results are stored on the HDFS partition too. The language that was chosen for the actual implementation is Java: the project was set up as a Maven project, with a dependency towards Apache Hadoop and the capability of generating an executable Jar file as an output.

When using Java as the implementation language, two classes, a Mapper class and a Reducer class must be designed, that will be the entities implementing the actual Map Reduce logic.

* + 1. Mapper implementation

The mapper is implemented as a generic class that specifies the types that it takes as inputs, with reference to the dataset. In this case, we have no use for the input key type, but we care about the value type and we want it to be an instance of the Text support class. Each row belonging to the dataset which is passed as input to the map method is split with respect to the ',' character. The county is then set as an output key, while the month and the death count are joined together as a comma separated string and passed as a value to the next step. Both key and values are manipulated as strings through the Text class. Moreover, two private instances of the Text class are used within the mapper to optimize the Map Reduce application, so that the same instances are used every time instead of new ones being allocated at each map call.

* + 1. Reducer implementation

In a similar fashion with respect to the mapper class, a Reducer generic class must be implemented, specifying the types of the input key and value iterator, which in this specific case are both Text instances. The same approach towards the optimization of the Text class usage is applied here too. The reducer splits the values that were mapped as comma separated strings and parses the death count as a number so that it can count the whole number of occurrences. It then computes the max count by month and writes the results.

* + 1. Execution environment and final remarks

Within the main method of the application, the mapper output key/value classes were explicitly specified to let the framework know not to use the default values.

The environment in which the application was run is as follows: we used docker-compose to set up a Docker stack containing a network with four containers based on the effeerre/hadoop image. This made it possible to realize a cluster of one hadoop master node and three slave nodes. A bash script was then realized to be executed within the master node, to correctly initialize it, formatting the namenode, starting the distributed file system, loading the dataset and putting the application in execution through the maven-generated jar.

* 1. Day with most tweets about COVID by location – Apache Spark – MongoDB – Python

This application exploits the distributed capabilities of the Apache Spark framework to analyze a huge dataset about COVID related tweets, writing the extracted information on a MongoDB collection. The dataset used as input for the application is read from the local filesystem or it is copied onto the filesystem of a Docker container through a Dockerfile. The language that was chosen for the actual implementation is Python, that offers a way to write Spark applications using the pyspark library. This library provides Python APIs to interface with the Spark engine. Python also offers a nice and easy to use interface to interact with MongoDB, by using the pymongo library. While I’s possible to use database models through ORM-like libraries, pymongo is simple to use and lets us write data easily by inserting it into a collection as a Python dictionary, which is akin to a Javascript object.

* + 1. Initialization and formatting: SparkSession and SparkContext

SparkContext is the legacy standard entry point for a Spark app. In this implementation, a SparkSession is used instead of SparkContext because it provides native csv parsing capabilities which are needed because the csv used for this application uses commas in the fields, escaping them with double quotes. A SparkSession may be also used to work with DataFrames, and it encapsulates a SparkContext: its RDD manipulation capabilities are recovered through the rdd member parameter.

* + 1. Spark data processing

After the initialization step is completed, we recover the RDD reference and map the underlying data onto a Python tuple. We then apply the algorithm used to extract the information we are seeking for. The first step is the date validation, to ensure that the parsed dates are well formatted, followed by two rounds of map/reduceByKey. These first round is needed to count the occurrences of a tweet in a location and a specific date; the second one is used to find the max number of occurrences of a tweet in a location and in a specific day. One of the strong points of using Python as a language for this kind of operation is the fact that we can exploit its inbuilt possibility to use lambda expressions through the lambda keyword. This feature is widely used throughout this application.

* + 1. Execution environment and MongoDB interactions

Once we finished processing the data, each partition of the RDD is targeted through a call to the foreachPartition method. Since Python functions are first-class citizens – Python supports higher-order functions – it is possible to pass a function to the foreachPartition method, that can handle the partition in an iterable fashion. In our case, the write\_to\_mongo function writes every row of an RDD partition onto the MongoDB collection.The execution environment is realized using a docker-compose stack, through a Docker network where a Spark node and a MongoDB node are connected and exposed together. The Python Spark environment is set up through a requirements.txt file, using the bde2020/spark-master:2.4.5-hadoop2.7 image as a base. Once everything is set up, the application is submitted to the Spark engine through the spark-submit script included in Spark.

* 1. Months with the highest increase in hospitalized patients, by state – Spark Streaming – Python

This last application uses the streaming capabilities of Spark Streaming to extract the right information from a stream of data, which, in this case, originates from a dataset that is sent through a TCP socket, simulating data arriving in streams. The language chosen for the implementation is once again Python: we also use the same libraries to interact with Spark and with MongoDB.

* + 1. Processing incoming data from a StreamingContext

A StreamingContext object can be used to obtain data through a call to its socketTextStream API, which returns the main Spark Streaming abstraction: a DStream, which is essentially a flow of data organized in RDDs. Data incoming from this process is split to be interpreted from a csv row. After being split, it is organized in the following fashion {(State, Date), HospitalizedCaseCumulative}. This formatting allows us to easily calculate the sum of the number of cases by month through a reduceByKey.

* + 1. Computing the increase in hospitalized numbers

The increase in hospitalized patients is computed by exploiting a particular feature of the Spark framework, which is DataFrame Windowing. We can iterate through RDDs thanks to the foreachRDD method exposed by a DStream, passing a function that will be applied to each RDD, thanks to the Python features that were being described in the previous application details. Once an RDD is being processed, we force a DataFrame view on it, specifying a column model through the createDataFrame spark API. We can then force an ordering in the windows that we are going to apply onto the streamed data through the orderBy method of the WindowSpec class. Once we have a DataFrame view, we can calculate the increase that we want to know about via the lag and when/otherwise functions available in the pyspark.sql.functions package. Through the lag function, we create an additional column in the DataFrame, containing the hospitalized count lagged by one entry, which is the one from the previous month. We then calculate the one-month increment by subtracting two values from adjacent months, putting the result into the increment column. It is important to consider the corner case of the first element of the RDD, which has no previous month, by using the isnull function.

Once everything is correctly executed, it is easy to regain the RDD capabilities by using the rdd member parameter of the DataFrame class, computing the max increase with a simple mapping of the data into a {State: Increment} structure, followed by a reduceByKey that uses Python inbuilt max function.

* + 1. Execution environment and a TCP streaming server

The processed data is saved onto a MongoDB collection in the same way as for the previous application. The whole environment is once again based on a docker-compose stack containing a Docker network where three containers are created: a MongoDB server, a Spark node that runs the application and is based on the the bde2020/spark-master:2.4.5-hadoop2.7 image, and a sender server. This last one is a simple Python TCP server that loads the dataset and accepts external connections on a configurable port. Once a connection is received, the server starts streaming the dataset data row by row towards the client, emulating a data streaming service that may be fed into a Spark Streaming application.