# Project 1
# Big Data Programming Paradigms

**Arthur Souto Lima**

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

arthursl@ufmg.br

## 1. Introdução

Nowadays solutions and tools for handling large amounts of data are being developed and created each and every day. Moreover, these new approaches involve as well distributed databases, so they can handle more ubiquitous problems. However, even among these many different tools, their paradigms for problem solving are similar, and getting used to this mindset is essential to being a more active professional. This is the aim of this project, to get in touch with two very famous tools to handle Big Data problems: MapReduce and Spark, both running in a cloud environment.

## 2. Project Structure

The project submitted is organized in folders, one for each task. It follows the structure below:

```
TP1
├── task1
│   ├── classes
│   ├── Hashtags.java
│   ├── IntTextArrayWritable.java
│   ├── TextArrayWritable.java
│   ├── run.sh
│   ├── json-simple-1.1.1.jar
│   └── task1_plots.ipynb
├── task2
│   ├── task2.py
│   └── run.sh
├── task3
│   ├── task3.py
│   └── run.sh
├── results
│   ├── task1.csv
│   ├── task2.csv
│   ├── task3.csv
│   ├── task1-2.png
│   ├── task1-3.png
│   ├── task3.png
│   └── task1-2.pdf
└── minidoc.pdf
```

The first task has the most files and folders inside. The main program is inside *Hashtags.java* and the other two *.java* files are auxiliary classes. *TextArrayWritable.java* is the one provided on Moodle, with an added method for printing. *IntTextArray-Writable.java* is an similar class with defines an writable object composed by an integer and an *TextArray*. The library JSON Simple was chosen for reading the input files and its code for linking is also available. Considering the amount of files and also some minor configurations in order to compile and run the task correctly, a *run.sh* bash script to do it all automatically is provided. It even gets the results from HDFS back to the current folder. As of now, this script uses the user *arthurlima*, however it may be changed by altering the variable *USER* in the script to work any other user as well. There is also a python notebook which was used to generate the graphs for this task, using *matplotlib* and the suggested Word Cloud[1] library. This notebook is run automatically by the script to generate the plots.

The second and third tasks only have a python file and its correspondent bash script for running them and recovering its results automatically. To run both these tasks one should change the *USER* variable in each python script and in each *run.sh* so they can access the correct HDFS folders.

Lastly, there is a folder with all CSV's, graphs and reports asked in the specification. Each file is labelled after its task. The code submitted generates all these artifacts, however, they will have different names and will be found in each task's folder. They have been moved to this special folder and renamed for submission purposes only.

## 3. Implementation Overview

In all code written, we aimed to comment everywhere possible to clarify the algorithm's intention. We also, as asked by the specification, plotted the graphs with labels and titles to increase its readability. Below, we describe the general ideas and main difficulties when implementing each of the tasks.

### 3.1. Task 1

The first task was the hardest to implement, mainly due to some inexperience with the JAVA language itself and, on top of that, with the Hadoop and MapReduce environment. In order to complete the task, there are two steps, i.e., two map-reduce jobs, which are both implemented in *Hashtags.java*. The first one is similar to the *WordCount* example provided, in which we aim to find the most popular hashtags, i.e., those which appear in more than 1000 tweets.

The second job, then, uses this information to find the related hashtags only between this set of hashtags. The first output is saved to a file an subfolder of the folder passed to the program and it is made available to all mappers and reducers (from the second job) via the Distributed Cache provided by Hadoop, so all entities can read the file. In the code, all mappers and reducers had a setup method which opened this file and read the contents into a Hash Map.

The second job mapper still goes through the tweets' text, but now, before sending anything, it selects only the popular hashtags in the tweet and then send to the reducer each

[1]`https://amueller.github.io/word_cloud/index.html`

pair of tags that appear in the text field. The reducer adds all related hashtags from a key (an hashtag) into an set, so we can avoid repetitions, and this set is the base for writing all the related hashtags into the output.

Lastly, in order to limit the final result, we sort the CSV using *sort* from GNU Core Utils and select only the first 500 lines, using *head* also from this set of tools. As commented in a previous section, the compilation, running and result extraction is done automatically using the bash script provided. It includes sorting and limiting the output.

With this cleaned output, creating the graph from task 1.3 was almost direct. The word cloud was also simple to do, after some quick reading of the library documentation.

Running the script, we generate two word clouds, one as asked with only the 50 most popular hashtags and one with all the 500. In terms of plots, we also generate two of them, they only differ in the order of the labels.

### 3.2. Task 2

As suggested by the specifications, the data provided is well structured, so it could be read into a Pyspark Dataframe. After that, solving the task was a question of choosing the correct queries and operations, which look very similar to SQL ones, so the databases courses proved themselves useful here.

The main idea is to find the oldest and the newest tweet for each verified account, which can be done by grouping tweets by user and selecting only the minimum or maximum date. With this information, we can find the initial and final values of followers and then compute the increase by subtracting them.

### 3.3. Task 3

The third task was also facilitated by the prior background on Python and databases. The hardest part in this activity was casting correctly the date attribute and finding the correct function to group by day. The moving average's implementation was also quick after the specs mentioned the Window functions.

The main query must find, for each day, how many users (not tweets), used a specific tag. To avoid counting an user more than once per day, we used the function *countDistinct* on the column *screen_name*, so when we grouped the tweets, we would only count users, no matter how many tweets he/she did that day using that hashtag.