# Supercomputing for Big Data ET4310 (2016)
## Assignment 1 (Using Spark for In-Memory Computation)

Lorenzo Gasparini (4609905)

October 9, 2016

## 1 Introduction

This assignment is about the usage of *Apache Spark* for large-scale in-memory data processing.

Apache Spark is an open-source framework for efficiently processing large amounts of data in parallel. It is gradually replacing *Apache Hadoop* for big data applications, since it does a better job at leveraging the RAM gaining 10-100x speedups for certain applications. Moreover, Spark integrates in a single framework the ability to do Batch processing, Streaming, Machine Learning and Graph Computation.

Spark is flexible as it can run on *Hadoop YARN*, *Apache Mesos* or in *standalone mode*, accessing data from a variety of sources including *HDFS (Hadoop Distributed File System)*, *Apache HBase*, *Amazon S3* and others. Furthermore, it offers a wide range of APIs for different programming languages, like *Python*, *Scala*, *Java* and *R*.

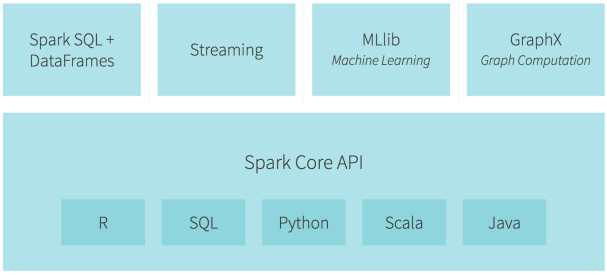


Figure 1: Spark ecosystem

Nonetheless, Scala is often considered the best choice since it's the language in which Spark itself is programmed, it's the API which gets updated first and the functional nature of this language allows for greater expressiveness.

Scala is a general-purpose multi-paradigm programming language, designed to integrate both the characteristics of functional and object-oriented programming languages. Compiling Scala programs results in *Java bytecode* that can run in a *JVM (Java Virtual Machine)*.

The assignment is divided in 3 exercises:

1. *Data exploration using Spark*: using Spark to obtain a set of statistics about page views of Wikipedia starting from raw pagecounts;

2. *Spark streaming*: reading a continuous stream of tweets from Twitter and computing statistics in real-time on them;

3. *More data exploration*: processing the output of exercise 2 to gain additional insight on the recorded tweets.

The next sections are organised as follows: *Background* contains an introduction to Spark and the specific components of Spark that were used to complete this assignment. In *Implementation* the technical implementation details are discussed. In *Results* it's discussed whether the initial goals were achieved successfully. Finally, *Conclusion* contains a summary of the report with additional remarks on what was learnt during the assignment.

# 2 Background

## 2.1 Batch processing

The primary abstraction provided by Spark is the *resilient distributed dataset (RDD)*. A RDD is a distributed and resilient (fault-tolerant) collection of records. The records can be either primitive values (like Strings or Integers) or more sophisticated objects, and they represent the data Spark is working with. RDDs are immutable (read-only), in-memory (as much and as long as possible), lazily evaluated (available only after an action triggers the transformations) and partitioned (residing in different nodes). Furthermore, RDDs are cacheable, meaning the data can be held in a persistent memory like RAM or disk.

The motivation behind the creation of RDDs is the enabling of efficient reuse of intermediate results between different processive-intensive tasks (in-memory processing). RDDs are best suited for batch applications applying the same operation on all the elements of a dataset, since this enables efficient recovering of lost partitions.

Currently the Spark API is going through a major overhaul and RDDs are being superseded by *DataFrames* and *Datasets*. These are higher level abstractions that are designed to make processing of large datasets even easier.

Since Spark 2.0, the preferred data structures are Datasets, which are strongly-typed, and DataFrames, which are untyped. A `DataFrame` is only an alias for `Dataset[Row]`, where `Row` is an untyped object. The advantages of the new API are: static-typing and run-time safety, expression of domain-specific operations with high-level structured constructs, performance and optimization.

Nonetheless, RDDs are not being abandoned and still the preferred way of applying low-level transformations on the dataset, working with unstructured data and not imposing a schema on it. The conversion between RDDs, DataFrames and Datasets is seamless and is just a matter of a method call.

## 2.2 Streaming

Spark Streaming is a component of the Spark ecosystem that enables continuous processing of live data, in a fault-tolerant and scalable way. The sources for the live data can be different, from *Kafka* to *TCP Sockets*.


Figure 2: DStream

The basic abstraction for live data provided by Spark is the *DStream*, which stands for *discretized stream*. A DStream is a continuous series of RDDs which supports operations similar to the ones that can be applied to RDDs, but hides the complexity of dealing with a continuous stream providing a higher-level API to the developer.

In Spark 2.0 the concept of *Structured Streaming* was introduced. It's a new computation model built on top of Datasets that offers a high-level declarative API. This, however, is still considered at an alpha stage of development and is not recommended for production use.

# 3 Implementation

## 3.1 Data exploration using Spark

The first assignment was about the processing of raw Wikipedia pagecounts to compute statistics on the most visited pages for each language.

The assignment has been done in two ways: the class `PagecountsRDD` uses RDDs while the class `PagecountsDataset` uses Datasets. This was done to gain an insight on the differences between the two APIs, in terms of performance and easiness of programming. In both the assignments the case class `PageViewData` was used, which represents an input object and enables referencing the fields with names instead of indexes. Furthermore, since the properties of the class are statically typed, we have the assurance that the input data is conforming to our expectations, type-wise.

#### 3.1.1 PagecountsRDD

For what concerns `PagecountsRDD`, only one reduction - using the language code as key - was necessary to compute the output. The transformations applied on the input are, in sequence

1. `map`: to split the lines on the spaces
2. `map`: to convert the rows into `PageViewData` objects
3. `filter`: to filter objects for which the `Language_code` is the same as the `PageTitle`
4. `map`: to create the `(K, V)` pairs for reduction, where the key is the language code
5. `reduceByKey`: to reduce the dataset using the language code as key
6. `sortByKey`: to order the output by the total number of views for each language
7. `map`: to format the output for visualisation

#### 3.1.2 PagecountsDataset

In this case, the first step is to create the `Dataset[PageViewData]`, which is done through reading the input, splitting and filtering the rows in a way similar to the RDDs version. Once it's created, it's possible to operate on it using the high-level API.

Unfortunately this specific exercise is not easily expressible with this API so the length of the code turns out to be longer than the version with RDDs. The main culprit is the computation of `MostVisitedPageInThatLang` and `ViewsOfThatPage`, that is a so-called *greatest-n-per-group* task. Because of the nature of the API, this values have to be computed initially for each row of the input dataset, using a window.

After computing those two values, the dataset is "grouped by" the language code, and the aggregate sum of each groups' view count is computed. Then the output is sorted and the column with the name of the language is added, using a UDF. This is similar to the way it would be done in SQL.

### 3.2 Spark streaming

This task involved dealing with continuous streams of data. To get the stream of tweets, the Twitter streaming extension of Spark is used. Unfortunately, this extension was removed in Spark 2.0 so the exercise is solved in Spark 1.5.2 instead.

The first step, after configuring the stream with a batch interval of 5 seconds and setting Twitter access credentials, is to filter the tweets stream so that only retweets are kept. Then, since we want to operate on a sliding window of data, the `Window` method is called with a window length of 60 seconds and a sliding interval of 5 seconds (implicit, same as batch interval). What the window method does is to generate a sequence of RDDs, one each 5 seconds, containing each one the previous 60 seconds of the stream. Each one of these RDDs is mapped to get only the information that we need from the retweets, and the result constitutes `retweets`, the primary dataset used in the program. All the operations listed below are applied to the 60-seconds window.

Next, the `retweets` dataset is is reduced using ID of the original tweet as a key, to get the minimum, maximum and total retweet count of each original tweet in the window. The output is then joined with the primary dataset and becomes `retweetsWithCounts`.

Afterwards, we reduce `retweetsWithCounts` by the language code to get the total retweet count for each language. The output dataset is then joined with `retweetsWithCounts` to get the final output stream.

The final output stream is then mapped to flatten the fields and sorted so that the retweets whose language has a higher retweet count appear first, and within the same language the tweets with the highest retweet count appear first. To sort the DStream it's necessary to use the `transform` method, to operate on every single RDD, since DStreams don't support sorting. Next, the objects composing the dataset are converted to the `TwitterStatsData` class, so that we have the type safety and we can refer to the fields by name.

The final step is writing the results to the log and this is done calling the `write2Log` function on each RDD in the output stream. This function checks that 60 seconds have passed - so the window has reached the full size - an then logs the results to a text file.

### 3.3 More data exploration

The last exercise was solved using the Dataset API and it uses, as the input, the output of the previous exercise. The input file is read as a CSV file, and the rows are mapped to `TwitterAnalysisData` objects, keeping only the fields that we need.

Then, the input dataset is reduced by the ID of the original tweets, to get the minimum and maximum retweet counts. The difference between these and the previous counts is that while the previous were relative to the 60-seconds windows, these are relative to all the running time of the previous script. We are then able to calculate the retweet count for each tweet.

The newly computed retweet counts are joined then with the input dataset and the result is de-duplicated using `distinct`.

Then the `TotalRetweetsInThatLang` column is added, which is computed using a window, similarly to what was done in the first exercise. Afterwards we sort the results by the language and inside the same language by the retweet count. The output is filtered so that only retweets with 2 or more tweets are kept.

## 4 Results

### 4.1 Data exploration using Spark

For what concerns the fist exercise, the `run.sh` script was modified so that it's possible to choose which variant of the solution to run (with RDDs or Datasets).

The applications were tested on a 2012 Macbook Pro with 4GB of RAM. This is the output of the first version, cut for readability:

```
$ ./run.sh PagecountsRDD pagecounts-20160801-040000.gz
Language,Language-code,TotalViewsInThatLang,MostVisitedPageInThatLang,ViewsOfThatPage
English,en,7464709,Main_Page,1732807
[...]
Done!
Time taken = 0 mins 9 secs
```

and this is the output of the second version:

```
$ ./run.sh PagecountsDataset pagecounts-20160801-040000.gz
Language,Language-code,TotalViewsInThatLang,MostVisitedPageInThatLang,ViewsOfThatPage
English,en,7464709,Main_Page,1732807
[...]
Done!
Time taken = 0 mins 50 secs
```

The execution time difference between the RDD and Dataset version is quite noticeable. This has to be attributed to the fact that the computations required by this task cannot be easily expressed using the Dataset API and thus require additional work by this variant, which results in longer execution times. In particular, the computation of `TotalViewsInThatLang` and `MostVisitedPageInThatLang` through the use of a window is done for every row of the input dataset and this is not efficient.

```
$ du -h pagecounts-20160801-040000.gz
75M pagecounts-20160801-040000.gz
```

The size of the input dataset is 75M, which is not really considered *Big data* since it can fit entirely into the computer memory. Usually Big data means datasets that cannot fit into a single computer disk, let alone its memory.

### 4.2 Spark streaming

For the second exercise, it's not possible to get the execution time of the transformations applied to each window due to the nature of the streaming application. Nonetheless, the computer is able to keep up seamlessly with the stream processing with a batch interval of 5 seconds.

### 4.3 More data exploration

The third exercise has the following execution time:

```
$ ./run.sh part2.txt
Language,Language-code,TotalRetweetsInThatLang,IDOfTweet,RetweetCount,Text
Korean,ko,9625,780367792717475840,511,RT @evgeniyaa0ale77: [text]
[...]
Done!
Time taken = 0 mins 18 secs
```

with the following input size:

```
$ du -h part2.txt
22M part2.txt
```

## 5 Conclusion

To conclude, this assignment have given me a good insight on big data processing using Spark, with both Batch and Streaming modes. I learnt the differences between RDDs and Datasets, and understood the limitations of each approach.

It would be interesting to test the programs with larger datasets, to see how well this approach can scale, which should be Spark's strong suit. It may even be that for big datasets the differences between the RDD and Dataset approaches, in terms of execution time, would reduce significantly.