ET4310: SuperComputing For Big Data
Resit Data exploration using Spark

Georgios Dimitropoulos: 4727657
Emmanouil Manousogiannis: 4727517
Group 29

November 2018

Introduction

In this assignment we are experimenting with the Apache Spark tool for processing big data analytics. More specifically, we are focusing on how to perform data exploration using Spark on a very large text file. Our task is about counting the frequency of each word in combination with the frequency of their preceding word. In order to be able to do so, we handled our large text data using the RDD data abstraction of Spark and performing a set of narrow and wide transformations. More details about RDDs, Spark and our implementation are mentioned in the following sections.

The rest of the paper is organized as follows: In the background section, we present all the relevant theoretical information about Spark and RDDs . In the implementation section, we describe the implementation of our code. In the sequel, in the results section we mention information related to the results that we managed to achieve like the execution time of our implementation, followed by the conclusion section where we sum up our work and our findings. Finally, our results can be found in the output file “freq.txt”.

Background

Apache Spark is an open-source framework which can be used in order to process a large amount of data in an efficient and effective way in parallel. In comparison to its main "competitor" Apache Hadoop has a better performance in terms of gaining RAM speed-ups of order of 10-100x for applications related to large amount of data. Furthermore, Spark consolidates into its ecosystem a variety of capabilities like Batch processing, Stream Processing, Machine Learning and Graph Computations. However, in streaming applications its performance is sometimes questionable due to the fact that Spark’s main processing technique of data is the batch processing.

In addition, Spark is quite flexible with respect to the mode that it can run, as there is the possibility to run on Hadoop YARN, on Apache Mesos or on standalone mode and accessing data through many different sources. Finally, there are a lot of different APIs in Spark for different programming languages including Scala (which is considered the default and best choice), Python, Java and R, something that enables Spark to be used by a variety of software developers with different programming backgrounds. An overview of the Spark ecosystem can be depicted in Figure 1.

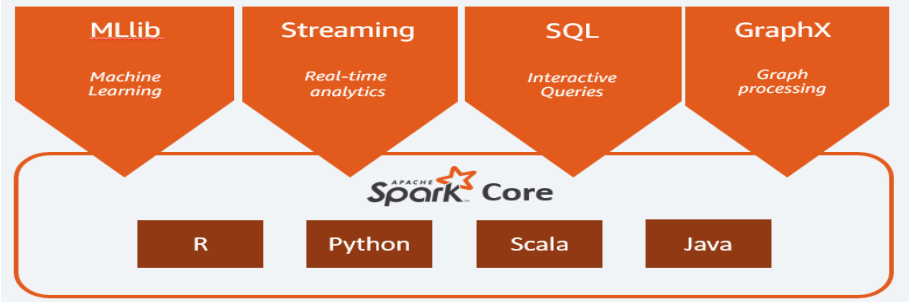

Figure 1: Overview of Spark Ecosystem

The main abstraction that Spark provides is the resilient distributed dataset (RDD). RDDs are considered as distributed and resilient with respect to fault-tolerance collection of data. RDDs are immutable in the sense that permit read-only operations, are partitioned in different nodes and are lazily evaluated in the sense that can be reached in case that a transformation has been triggered by some actions. Moreover, there is the possibility for the the data in RDDs to be saved in a persistent memory (i.e. RAM/disk).

The main advantage that RDDs offer is that of allowing an efficient reusing of the intermediate results that have been calculated in intermediate tasks (i.e in-memory processing) and also allowing reducing computational costs through applying the same operation on the whole dataset.

The other form of data abstraction that exist in Apache Spark are the Data-frames (untyped) and Datasets (strongly-typed). More specifically, Data-frames are an alias for Dataset[Row], where Row is an untyped object. The advantages of Data-frames and datasets in comparison to the RDDs are the Static-typing and run-time type-safety, the high-level of abstraction, the view into structured and semi-structured data and their performance and optimization. An overview of Data-frames and Datasets can be delineated in Figure 2.

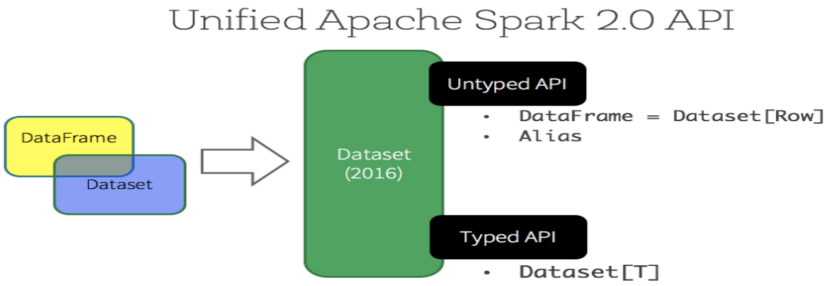

Figure 2: Overview of Data-frames and Datasets

However, as a final remark it is worth mentioning that RDDs are still the preferred way of applying low-level transformations on the data and dealing with unstructured data without enforcing a schema on it in most of the use cases.

Implementation

As mentioned above, we used RDD , the main data abstraction of Spark, in order to perform the necessary data exploration of our large text file. As a first step, we created an RDD with this text file, using *SparkContext*. We split the whole string with white-space as a separator and converted all the resulted tokens (words) to lowercase, in order to ensure the case insensitive requirement. As a next step, we created tuples of all the words that occurred next to each other in the text in order to also be able to identify the preceding words. After performing, some narrow and wide transformations in this data we managed to count the frequency of all words and their preceded words in each case.

It is worth mentioning here that we tried to perform as few wide transformations as possible, as they are more expensive and reduce the performance and the scaling ability of our application.

Wide transformations, usually consist of *'join', 'groupByKey()' or 'reduceByKey'* transformations which are in general computationally expensive compared to single 'map' or 'filter' operations. Our initial implementation, consisted of two RDDs, where the first one was used to count the word frequencies and the second one was used for counting the preceding words of each word. As a final step, the two RDDs were joined and the final result was saved in a text file. However, this was computationally expensive, and made our application a bit slower than we expected for this amount of data. For this reason, we tried to perform this using one simple rdd. This made our 'map' and 'filter' transformations a bit more complex, but the performance was improved. For the same reason, we tried to reduce the use of 'groupByKey()" operations when possible. At the final stage, we preferred not to use 'coaelsce(1)' operation, as we initially did before storing to our text file, as this would re-partition our RDD to one partition and this was quite expensive.

Results

The final results of our implementation were quite good in terms of performance, as well as in terms of results' quality. All non-word tokens are not taken into account as requested, and all words that come after a comma or another deli-meter are stored without any preceding words. For this reason, it is reasonable that the sum of the preceding word frequencies, does not always sum up to the frequency of the main word itself.
In terms of performance, we finally managed to process the given text file of 6.31 MB (128,456 lines of text), in 11 seconds running in our personal computer with an Intel i7 processor and 8 GB of RAM memory.

Conclusion

As a conclusion, we can say that the most important thing that we learned from this assignment is that Spark is a very efficient tool to handle big data tasks, especially when they are related to batch processing. However its capabilities of quick and parallel processing, are only fully utilized when our implementation is able to take advantage of them. If we do not manage to take advantage of the parallel processing capabilities of Spark, then our code will not be able to scale sufficiently in case that the amount of data increases, even if our computational power is large enough. This requires deep understanding of the Spark architecture and its data abstractions in order to optimize our code.