Final Work for CDLE 2023

Duarte Domingues, Miguel Távora
Instituto Superior de Engenharia de Lisboa, a45102, a45140@alunos.isel.pt

***Abstract* – This work uses the MapReduce programming model to solve large-scale computational problems (Big Data). The MapReduce model is supported by the Apache Hadoop platform. This work presents a word count project using Apache Hadoop for analyzing a set of documents by counting the frequency of n-grams. The project produces relevant statistical information, including a table of n-gram frequencies and the percentage of n-grams that occur only once (also known as singletons). The project uses the Hadoop Distributed File System (HFDS) and allows for configuring the application through properties specified at application submission or through configuration files. The project also utilizes compressed data, collects statistical data using counters, and make use of a distributed cache. For the results we have achieved a program that can count words according to the configuration files, such as n-grams, including a table of the frequency of n-grams, relevant statistical information, and calculation of the statistical measure TF-IDF. In this work various statistical information can be extracted using the different features of the Apache Hadoop platform.**

INTRODUCTION

This work intends to apply the MapReduce programming model to solve large-scale computational problems (Big Data). To implement the MapReduce model, the Apache Hadoop platform was used. The problem to be addressed is left to the students to decide, but it must include the mechanisms and features of the MapReduce supported by the Apache Hadoop platform. The problem chosen in this work was counting words of dimension n-grams in a chosen set of documents. The implementation has the goal of producing relevant statistic information, such as frequency tables of n-grams, percentage of n-grams that occurs once (*singletons*) and calculate the statistical metric of TF-IDF. In the implementation the HDFS file system must be used and there must be a possibility to configure the application through configuration files. The application must use compressed data and utilize distributed caching. The results intended to obtain is statistical data using counters and the results must be sorted.

RELATED WORK

“Big Data Analytics with Apache Hadoop MapReduce Framework” by L. Greeshma and G. Pradeepini is a research paper focused on the usage of Apache Hadoop MapReduce for big data analytics. This article provides an overview of the framework most important components, and the most important challenges related to Big Data. It describes the use of Hadoop Distributed File System (HDFS) and MapReduce. The paper describes advantages of using Hadoop MapReduce, for example its scalability, cost-effectiveness, and fault tolerance. The paper presents a way of developing a simple word count application using MapReduce. It first divides input files into tokens. Next the Mapper function initiates the key value pairs that will be represented in the Output file. The Reducer function gathers input splits from the map and generates intermediate values. Thereby applying shuffling and sorting operations. Finally, the output is denoted as (<key>, n) where n is the number of times a token is presented in the input slit file.

The conclusion of the paper points that the Hadoop MapReduce framework is a powerful tool for big data analytics and has many usages for various applications.

This paper was useful to understand basics of Apache Hadoop, its advantages and usage in big data analytics. With our project we aimed to develop a more complex word count application utilizing some of the information found in this paper.

IMPLEMENTATION

I. Apache Hadoop

To implement the work, it's necessary to first understand Apache's Hadoop technology. Apache Hadoop is an open-source software framework widely used for distributed storage and processing of large data sets on clusters of commodity hardware. It was developed by the Apache Software Foundation and is widely adopted by organizations for big data processing, data analysis and machine learning tasks.

Hadoop's architecture is designed to handle large amounts of data, either structured or unstructured. It can process data from a variety of sources such as newspapers, sensor data, social media feeds, images and videos. Hadoop's distributed storage and parallel processing capabilities make it suitable for tasks like data storage, data mining, data analysis and machine learning.

Hadoop includes two main parts: the Hadoop Distributed File System (HDFS) and the MapReduce programming model. The HDFS is a distributed file system that allows data to be stored across multiple machines in a cluster, offering a high degree of fault tolerance and scalability. The MapReduce is a programming model that allows to process large datasets in parallel through the nodes of a cluster.

II. Implementation of the main classes

For the implementation of our application, the Java programming language was chosen. For the application to take advantage of the Hadoop technology it's necessary to have three classes, which are:

- The class to run the program, which contains the `main`
- The mapper class, which extends from the class `Mapper` from Hadoop
- The reducer class, which extends from the `Reducer` class from Hadoop

The main function of a mapper class is to process the input data and transform it into a set of key-value pairs. It applies a user-defined function, known as the "map-function". The output of the mapper is sorted by key and grouped together by key, this way all the values associated with the same key are passed to the same reducer. The mapper class contains a set of methods that can be used to implement the desired behavior, such as `setup()`, `map()`, `run()` and others. The method `setup()` is used to add information to the map function, for example: to detect only the red color on an image. In all those methods the one that must be always implemented is the `map()` method. The map method is used to define which information is going to define the key and the value on the map. In the case of the project is going to set the key as the word and value as 1.

The reduce function is to process the intermediate key-value pairs generated by the mapper class and calculate the output of the MapReduce job. The reducer applies a user-defined reduce function to each group of values associated with the same key. In the case of the project the reduce function is going to count the number of occurrences of each word. The output is a set of key-value pairs where the key is the word, and the value is number of occurrences. The reduce class works in a similar way as the mapper, having the methods: `setup()`, `reduce()`, `cleanup()` and others.

The class to run the program has the responsibility to create the configuration of the program and the job that is going to execute the MapReduce. The configuration and the class to run was separated in two classes to make it easier to read the code. The class to execute only has the `main` method. It uses the class `ToolRunner`, which provides a simple way to configure and run MapReduce jobs with command-line options. It can specify configurations like input/output paths, number of reducers, memory limits and so on. To use `ToolRunner` some class must implement `Tool`. The class that implements Tool is the configuration class.

III. Implementation of the configuration class

As mentioned before there is a class that is responsible for various configurations such as set up and run a MapReduce job. By extending `Configured`, the class inherits the ability to access and set Hadoop configurations. By implementing the class `Tool` it can run using the `ToolRunner` class. It will be on this class that is going to be called the class that uses configuration files and where it will be displayed the output result.

The name given to the class is `WordCountConfig`. To create the MapReduce firstly it's needed to instantiate the `Job` class. This class needs a group of setters to work, where the first ones is: the set, sets the class where the Jar is going to be executed and the name of the job. Without those two methods the Job will not be execute and will raise an exception. The other important part is to define the `FileInputFormat` and `FileOutputFormat`.

The `FileInputFormat` specifies the input format for the MapReduce of the `Job` and splits the data into chunks that will be processed by the map tasks. To use the class, first the path to the input files must be defined. The `FileOutputFormat` is the class that specifies the output format. In the same way as the `FileInputFormat` it must

define the output path and provides methods to specify the output data compression, although the compression was specified on the configuration file.

The next step necessary on the Job configuration is to set the Mapper and Reducer classes. Those classes will decide which part of the files will be the key and which part the value. The result of the reducer will be the grouped by the key and the value will be the number of occurrences of the key. The next part is just defining the type of class that will be the key and value for both the input and output. At the end the following method must be called:

$$job.waitForCompletion(false)$$

This method waits for the Job to complete the MapReduce. Only this way will the desired Output be generated; without this method the program will just end without doing any job.

IV. Implementation of the configuration files class

In the program, most of the time we need specific configurations to be applied without needing to compile all the code again. In that matter the necessity of configurations files emerges.

First of all, a configuration file can be used to configure properties of a Hadoop cluster, such as: location of the Hadoop cluster, the location of the input and output data, number of reducers and other settings related to the execution of the program.

To add the configuration file to the settings of the job, it's only necessary the following line of code:

$$conf.addResource(new\ Path(PATH, FILENAME))$$

In the previous line of code, the `conf` represents the instance of the class `Configuration`. In total the program has two configuration files, one to configure the MapReduce functions, namely: the compression type used during the output phase of MapReduce, number of reducers used, and if it will compress the output or not. The second is used to configure the word count variables, namely: number of n-grams used, if the words will be case-sensitive or not and if it's going to skip some patterns or not.

To verify that the configurations are being applied, methods to check if the properties are being applied or not were created. The first one is to check if both the configuration files exist if it's true it applies the file to the `Configuration` if not it doesn't apply. The second is the number of reducers used, if it's not being applied the number of reducers should be 1.

$$job.getNumReduceTasks()$$

Secondly the output file format was checked, which can be done with the following line of code:

$$conf.get("mapreduce.output.fileoutputformat.compress.codec")$$

The last configuration added was to compress or not the output. To verify this, only the file extension of the output was checked.

V. Data compression

The data compression used on the configuration file is going to be explained is this chapter.

Data compression is a method of reducing the necessary amount of data needed to represent a specific information. This leads to more efficient storage and faster data transfer.

Apache Hadoop supports various types of compression codecs. The codecs, compression / decompression algorithms, used in this project were Gzip, Bzip2, Lz4 and Snappy.

- **Gzip** uses the Lempel-Ziv-Markov chain algorithm (LZ77) for data compression and sits in the middle of the space/time trade-off.
- **Bzip2** uses a combination of the Burrows-Wheeler transform and Huffman coding for data compression, it compresses data more efficiently than Gzip but is not as fast.
- **Lz4** is a data compression library that is optimized to be fast at compressing and decompressing data, but less effective.
- **Snappy** like Lz4 is designed for optimal compression and decompression speed but has lower compression rates than Gzip and Bzip2.

On the project, compression was applied to the files before they were stored in the HDFS. In the moment a file is read from the HDFS, the chosen codec is used to decompress the data stream.

The configuration for the compression codec to be used in the output files of the MapReduce job is stored inside an XML file. In these files the value of the codec to be used is defined and if the output files of the MapReduce job are going to be compressed. This is achieved using the following configuration properties:

- `mapreduce.output.fileoutputformat.compression.codec`

- `mapreduce.output.fileoutputformat.compress`

VI. N-Grams

N-Grams refers to a sequence of N words. It's a popular method in Natural Language Processing to represent words or tokens in an input text. This method divides the input text into smaller groups of N words, which is useful to obtain the context of words in a text.

In the project, n-grams were calculated in the MapReduce application. More specifically on the mapper method. The Mapper method calls a method called `getAllNGrams()`. This method execution follows the following state diagram:

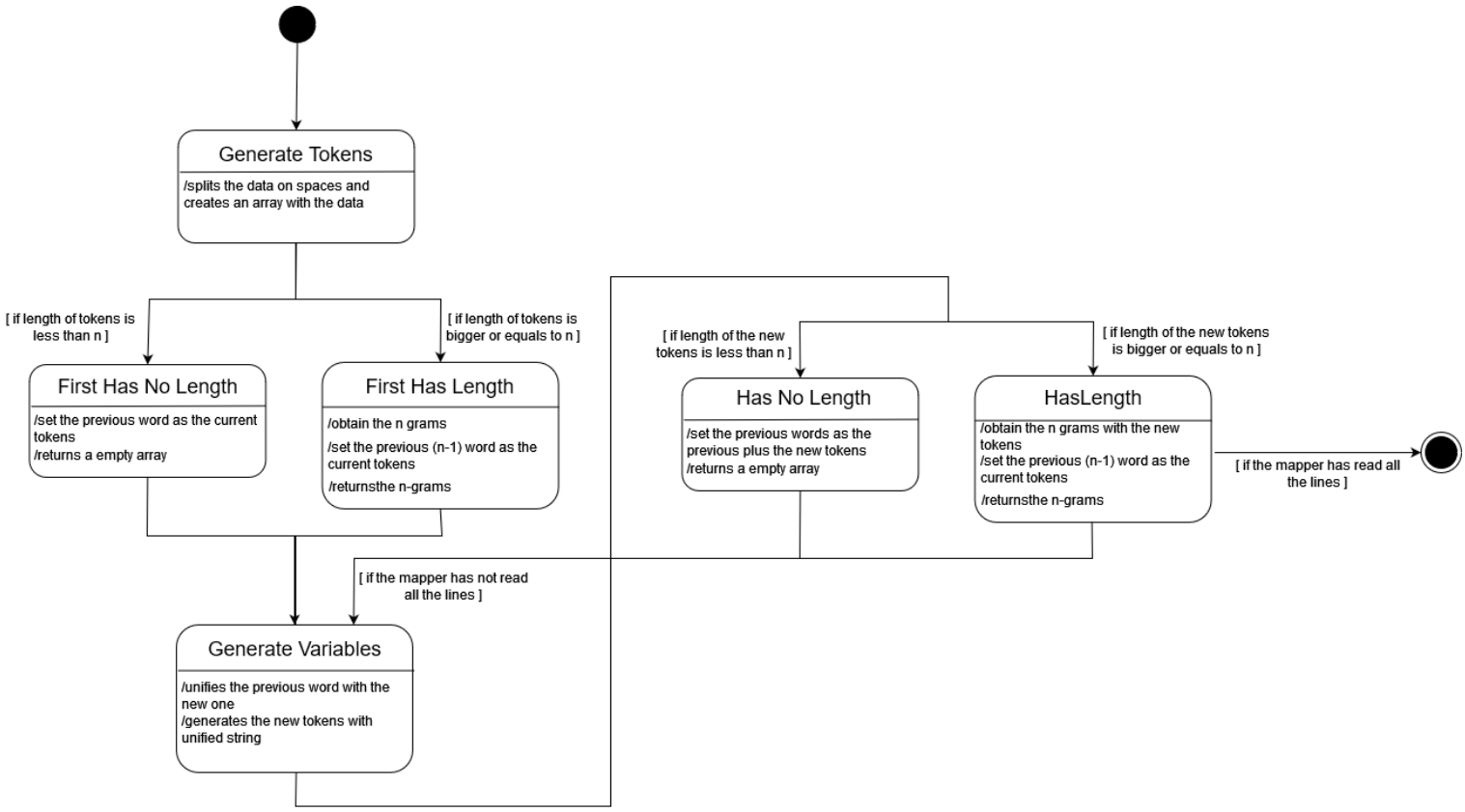

Figure 1 - state machine diagram of the n-grams method

The method firstly splits the first line read obtaining the tokens of this line. After that it checks if the size of the tokens have enough length to build an n-gram, if they do the program creates the n-grams with the method `getNGrams()` and sets the new previous words and returns the n-grams. The previous words are the last words used on the last line; this follow the rule where the number of words is n-1. If the tokens don't have enough size, the previous words are set as the tokens obtained and the method returns an empty array. After the first time it generates tokens with always the previous words plus the new ones. The following states are equal to the previous one, the only difference is that they use the previous plus the new one's tokens to build the words.

The method `getNGrams()` basically creates an array of strings where each index is an n-gram word united by a white space.

VII. Collecting statistical data using counter

Counters are a mechanism of Hadoop MapReduce that allow to track distinct information of the MapReduce job's execution. They are important to assure the quality of a job and gather statistical data for the application.

In this application, built-in counters were defined to collect specific statistical data. The counters are defined in the job configuration and can be incremented in the mapper or reducer function.

The statistical counters were used in the context of the word count applications. The counters were used to keep track of the following information:

- Number of total n-grams
- Number of distinct n-grams
- Number of singletons n-grams

To gather information from word count, firstly it's necessary an enumerator where each entry defines a value that will be incremented by the MapReduce. This enumerator is on the class `WordCountUtils` and the entries is the same as mentioned before.

The number of total n-grams is the only counter that is incremented in the mapper function. The mapper iterates through the n-grams created and sets the pair <key, value>, where key is the word and value is 1. The count of the words will be done on the reducer method. So, each pair created is a word and it also increments one on the total words. The increment is written in the following line:

$$context.getCounter(WordCountUtils.Statistics.TotalWords).increment(1);$$

The other two counters, number of distinct n-grams and singletons n-grams are incremented in the reducer function.

The number of distinct n-grams is obtained through a counter which is incremented in the reduce method for each key. The counter of number of singletons n-grams works in a similar way but is only incremented when the sum of values is equal to one. The increment of the distinct word, is written in the following way:

$$context.getCounter(WordCountUtils.Statistics.Distincts).increment(1);$$

The results for the statistical counters were the following:

	N-Grams				
	1	2	3	4	5
distinct	4432	14073	17786	18786	19237
singletons	123	475	697	843	969
total	44820	44811	44802	44793	44784

Table 1 - number of occurrences of words with corpus: gutenberg-mixed

	N-Grams				
	1	2	3	4	5
distinct	12	14	12	10	8
singletons	10	14	12	10	8
total	16	14	12	10	8

Table 2 - number of occurrences of words with corpus: gutenberg-small

	N-Grams				
	1	2	3	4	5
distinct	4432	14073	17786	18786	19237
singletons	123	475	697	843	969
total	44820	44811	44802	44793	44784

Table 3 - number of occurrences of words with corpus: Gutenberg

The previous tests were made with case-sensitive letters and ignoring patterns such as: "'\\.", "\\,", "\\!", "the'".

The following results differ from the ones in the examples provided by the professor because it was ignored the word "the". To split the string, it was used the class `StringTokenizer`, and after that we converted it into an array of strings. Another possibility was using the method `split()` with the a regular expression "\\s+". The class `StringTokenizer` splits on spaces, tabs and newlines, while the regular expression splits on one or more white spaces.

VIII. Singletons

In the context of word count applications an important concept is singletons. Singletons are words or tokens that appear only one time on the input texts. The calculation of singletons in the input texts allow the extraction of important statistical information about the input data.

In the word count MapReduce application singletons are the distinct keys calculated in the reducer function that have a value of one in the key-value pairs of the input data. Those words normally are the most relevant to obtain information from the text.

Singletons were calculated on the reducer function as mentioned before through counters. To do that we created an "if" statement where the singleton counter is only incremented when the value of the key-value pair is 1. The line of code that increases the count of the singletons is the following:

$$context.getCounter(WordCountUtils.Statistics.Singletons).increment(1);$$

IX. Patterns and case-sensitive

Two important parts of the word count problem is the patterns and the case-sensitive words. The problem with the patterns is that sometimes when counting the words, some random characters appear that are not relevant to extract information from the texts.

For the reasons mentioned previously it's necessary to delete those words. To do that first it must be defined which words are going to be deleted. This was defined on the class Patterns, but this could also be done with a text file. In the configuration file it's defined if the elimination of the patterns should be done or not.

Those patters were removed on it on the mapper function using a "for loop" replacing all the patterns on the string of the file.

The second matter is if the words should contain capital letters or not. The question on this case is if the capital letters contain more information than just lowercase letters. The same way as patterns, it can be defined on the configuration file. The usage or not of capital letters, was also defined on the mapper function where it uses the function `toLowerCase()`, to convert all the characters to lowercase.

X. Ordering the results

By default, the output results of the word count application are alphabetically ordered based on the words in the input files. This sort of ordering of results is useful in certain scenarios, for example in a search algorithm based on a given word.

Another useful way of ordering the output results is by ordering based on the number of occurrences of the word. This is useful in cases where the knowledge of the most important words in the input texts is necessary, for example if it's necessary to know the word that occurs the most in a document.

This sort of ordering must be done after all the words have been counted. Therefore, a secondary job that sorts the output of the first job is needed, now based on the frequency of the words. The output result will be a key-value pair of (number of occurrences of a word, word).

The ordering Job is created also on the class that creates the first Job, which is `WordCountConfig`. The second job also needs to set the jar and the job name. In the case of this job the input path will be the output path of the first job. The output will be a directory inside the output of the first job. Since the sorting only needs the mapper, it's set the mapper and the type of the key and the value for the mapper. At the end the method to wait for the Job to complete is also called.

The mapper class, `SortWordMapper`, of this job takes the output files of the word count job, although the key obtained is the length of the line. So, the key obtained was ignored. In the value both the word and number of occurrences are presented. In this matter, we split the value on the white spaces and the last value of the array is the number of occurrences. In the pair key-values now the key corresponds to the number of occurrences and the value to the word.

Given this type of pair key-value the Hadoop will automatically order the file by number of occurrences. The result files are grouped by number of occurrences, where the words that occurs once appear on the file 01, the ones that occurs twice appears on the file 02, and so on.

XI. Hadoop Distributed Filesystem

HDFS is a distributed file system designed to be executed on commodity hardware. The HDFS is the primary storage used in the Apache Hadoop application. In this file system when a file is stored, it's divided into data blocks.

The HDFS architecture is master / slave. The master node is the Namenode, it's responsible for managing the file system namespace and the access to files by clients. The slave nodes are the Datanodes. The Datanodes store the data blocks of the files. They are responsible for serving read and write requests from clients, block creation, deletion, and replication based on instructions from the Namenode.

The HDFS block size is configurated based on the needs of the application. A smaller data block has a higher seek time then a larger one due to the file being split into smaller blocks, so it's harder for the Namenode to search for the location of each block. Although with a smaller block size the transfer time is lower.

The Command Line Interface (CLI) was used to interact with HDFS and perform different operations in the file system. This was done through various commands, for example:

- `hadoop fs -mkdir` a command used to create a new directory in HDFS

To access HDFS resources the "hdfs://" URL scheme is used.

To use this filesystem on the project the input and output of the files must be on the HDFS. When running the program it removes the previous output of the program, because Hadoop can't generate a new output when the directory already exists. All the processes are the same as when running on the local computer. The only difference is that HDFS uses a different path, like the one referred before *"hdfs://"*. To specify which filesystem will be used it was created a bash file. This file is used to run the program to count the words and it receives one argument which defines which file system to use between the local and HDFS filesystem. The written arguments are the following:

- local – uses the local filesystem to run the program.
- HDFS – uses HDFS filesystem to run the program.

XII. Read ZIP files

When using the different input files, we attempted to read Zip files. We found that it's possible to read ZIP files through a set of configurations on the Configuration file. This code is on the ZipFile class and the configurations referred before are in the `setCompressionZip()` method. This class also checks if on the input directory exists a zip file. Since we have two different possible filesystems, it must check both the local and HDFS filesystem. To check the HDFS we used a `FileSystem` class from Hadoop. With this class it's possible to obtain an array of file status where it contains the files on the directory. Then it's checked for files with ".zip" extension, if there are files with this extension, they are added to the configuration. In case there aren't any files with the extension in HDFS it checks if there are files with the "zip" extension in the local filesystem. To check on the local filesystem it's used a conventional approach using Java.

Although the program can read Zip files, it must convert the characters read to plain text and this is where the problem lies. We tried several approaches, where the first one was using the Tika library. This library was imported on the Maven project but was not recognized when imported on the Java class. The second approach was using a Java Scanner, but this approach was also unsuccessful.

So, even though the program can read Zip files, it can't convert it to plain text. This means that reading zip files was not implemented with success.

XIII. Usage of distributed cache

In Hadoop distributed cache is a mechanism for caching used files, like jars and configuration files. This allows the files to be more accessible to the tasks running on the nodes of the cluster, preventing it to be sent over the network. This can improve the performance of jobs that uses the cached files.

In the application this was done on the class `WordCountConfig`. The files that needed to be stored were the input files. To do that, it was used the following line of code:

$$job.addCacheFile(new\ URI(args[0]));$$

On the previous line, the job represents the `Job` class that will run the MapReduce. The `args[0]` represents the path to the input either on the local or HDFS filesystem. To check if the files were really added to the cache it was created two methods. The first one is on the `WordCountConfig` itself. The method consists in calling the following line of code:

$$URI[]\ cacheFiles\ =\ job.getCacheFiles();$$

The program iterates through the array of URI's and prints the files stored on the cache. Although we had to be certain if the files were being used on the MapReduce. To do that we created a method on the mapper class which has the following line of code:

$$Path[]\ localCacheFiles =$$
$$context.getLocalCacheFiles();$$

The previous code returns the files stored on the MapReduce. This the program also iterates through the array and prints the files cached on the MapReduce. This way we were certain that the cache was being used.

RESULTS

Analyzing the results of the application, the output of the program was checked multiple times, both the word count and the sorting of the words. The word count seems correct, to verify this we used the Gutenberg-small file. The results are the following:

Word	Occurrences
World	4
Hello	2
Bye	1
In	1
Hadoop	1
to	1
The	1
Goodbye	1
is	1
blue	1
again	1

Table 4 - results obtained with 1 n-gram with gutenberg-small dataset

An occurrence of the word "the" with lowercase letters is missing on the previous table. This is because on the patterns specification this word was removed. After counting the words by hand, the result was verified as correct.

The part where the results are ordered is also correct. The sorted results are always inside a directory on the output of the word count. This way it's easier to find the sorted results.

The output for the rest of the n-grams with gutenber-small dataset is the following:

Word	Occurrences
Hello Hadoop	1
In World	1
World again	1
World is	1
to hadoop	1
The World	1
World The	1
blue In	1
Bye World	1
Goodbye to	1
Hadoop Goodbye	1
Hello World	1
World Bye	1
is blue	1

Table 5 - results obtained with 2 n-grams with gutenberg-small

Word	Occurrences
Bye World The	1
Goodbye to hadoop	1
Hadoop Goodbye to	1
Hello Hadoop Goodbye	1
Hello World Bye	1
In World again	1
The World is	1
World Bye World	1
World The World	1
World is blue	1
blue In World	1
is blue In	1

Table 6 - results obtained with 3 n-grams with gutenberg-small

Word	Occurrences
Bye World The World	1
Hadoop Goodbye to hadoop	1
Hello Hadoop Goodbye to	1
Hello World Bye World	1
The World is blue	1
World Bye World The	1
World The World is	1
World is blue In	1
blue In World again	1
is blue In World	1

Table 7 - results obtained with 4 n-grams with gutenberg-small

Word	Occurrences
Bye World The World is	1
Hello Hadoop Goodbye to hadoop	1

Hello World Bye World The	1
The World is blue In	1
World Bye World The World	1
World The World is blue	1
World is blue In World	1
is blue In World again	1

Table 8 - results obtained with 5 n-grams with gutenberg-small

CONCLUSION AND FUTURE WORK

In conclusion, this work aims to solve a large-scale computational problem using MapReduce problem. To do this we learned how to use the framework Apache Hadoop. Hadoop has a lot of functionalities, which can be used to produce the resulting key-value pairs pretended. This framework problem can handle problems like for example: counting the words of files, image or video processing, form processing, and extraction of information from a bill.

In this project we choose to do the word counting with n-grams. In this matter we learned how to implement a lot of functionalities to obtain the best information extraction. Overall, we learned:

- What is a MapReduce job.
- What is HDFS filesystem.
- How to create a program that can extract information from strings with MapReduce.
- The necessity to use distributed cache and to do it.
- Why are configuration files so important, and to add it to a program.
- How to collect statistical data from a MapReduce job.
- How to compress the data during the MapReduce job.
- Why it's necessary to sometimes order the data of the output, and how to create a new job to do that.

The future work that was intended to do is to implement the TF-IDF statistical metric. This is a way to prevent the words that appear more of smashing the words that appears less. This is done by giving more relevancy to the words that appears less and less importance to the ones that appears the most. Unfortunately, it wasn't possible to implement this because of the lack of time.

REFERENCES

[1] – Carlos Gonçalves, Computação de Dados em Larga Escala, 04-HadoopIO, 2023

[2] – Carlos Gonçalves, Computação de Dados em Larga Escala, 05-HadoopConfigurations, 2023

[3] – Carlos Gonçalves, Computação de Dados em Larga Escala, 06-MapReduceApplications, 2023

[4] - Greeshma, L., and G. Pradeepini. "Big Data Analytics with Apache Hadoop: MapReduce Framework."

AUTHOR INFORMATION

Duarte Domingues and Miguel Távora, Student, Engenharia Eletrónica e Telecomunicações e de Computadores, Instituto Superior de Engenharia Informática.

APPENDIXES

```
package cdle.wordcount.mr;

import org.apache.hadoop.util.ToolRunner;


public class WordCountApplication {


	public static void main(String[] args) throws Exception {

		if ( args.length < 2 ) {
			System.err.println( "hadoop ... <input path> <output path>" );
			System.exit(-1);
		}

		System.exit( ToolRunner.run( new WordCountConfig(), args) );

	}

}
```

```
package cdle.wordcount.mr;

import java.io.File;
import java.io.IOException;

import java.util.*;


import cdle.wordcount.lib.WordCountUtils;
import cdle.wordcount.patterns.Patterns;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class WordCountMapper
                extends Mapper<Object, Text, Text, IntWritable> {

        private final static IntWritable one = new IntWritable(1);
   private final Text word = new Text();

        // variable to change the content of the file
        private boolean caseSensitive = false;
        private Set<String> patternsToSkip = new HashSet<String>();
        private int nGrams = 1;

        private String[] previousContent = null;


        @Override
        public void setup(Context context) throws IOException {

                Configuration conf = context.getConfiguration();

                this.caseSensitive = conf.getBoolean("wordcount.case.sensitive", true);

                boolean skipPatterns = conf.getBoolean( "wordcount.skip.patterns", false);

                this.nGrams = conf.getInt( "wordcount.ngrams", 1);

                // only obtain the patterns to skip if configured that way
                if(skipPatterns) {

        patternsToSkip.addAll(Arrays.asList(Patterns.PATTERNS));
                }
        }


        @Override
        public void map(Object key, Text value, Context context)
                        throws IOException, InterruptedException {

                // content in lower or higher case
                String content = ( this.caseSensitive) ? value.toString() : value.toString().toLowerCase();

                for (String pattern : this.patternsToSkip ) {
                        content = content.replaceAll(pattern, "" );
                }

                // variable to print if has used cache or not
                String result = getCacheFiles(context);
                System.out.println("Cache used: "+result);

                //String[] grams = getNGrams(content, this.nGrams);
                String[] grams = getAllNGrams(content, this.nGrams);

                for (String gram : grams ) {
                        word.set(gram);
                        context.write(word, one);


        context.getCounter(WordCountUtils.Statistics.TotalWords).increment(1);
                }
        }


        private String[] getAllNGrams(String input, int n) {
                //String[] tokens = input.split("\\s+");
                String[] tokens = getTokens(input);

                // first state, when ot does not have any data
                if (previousContent == null) {
                        // when it has not enough length just set the values
                        if (tokens.length < n) {
                                this.previousContent = tokens;
                                return new String[0];
                        }
                        // when has enough size, generate n-grams and copy the last words
```

```
                else {
                        String[] nGrams = getNGrams(input, n);
                        this.previousContent = copyPrevious(tokens, n);
                        return nGrams;
                }
        }

        // generate a string with all the previous content
        String previous =getUnifiedString(this.previousContent);

        // adds to the new one
        String fullStrg = previous+input;

        // must create a new tokens because has new content added
        String[] newTokens = fullStrg.split("\\s+");

        // when it's not enough on the second or more to full a gram
        if(newTokens.length < n) {
                // just copy the words that exists
                this.previousContent = copyPrevious(newTokens, n);
                return new String[0];
        }

        // when it has enough content, gets the previous and the new
        else {
                // calculates the n-grams
                String[] nGrams = getNGrams(fullStrg, n);
                // copy the new previous
                this.previousContent = copyPrevious(newTokens, n);
                return nGrams;
        }

    }

    private String[] getNGrams(String input, int n) {
        //String[] tokens = input.split("\\s+");
        String[] tokens = getTokens(input);

        String[] grams = new String[tokens.length - n + 1];

        for (int i = 0; i < tokens.length - n + 1; i++) {
                String gram = "";
                for (int j = 0; j < n; j++) {
                        gram += tokens[i + j] + " ";
                }
                grams[i] = gram.trim();
        }
        return grams;
    }

    private String[] copyPrevious(String[] tokens, int n) {
        // when size is 0, just return empty
        if(tokens.length == 0) return new String[0];

        // when size is less than n - 1 it's just the tokens
        if(tokens.length < n - 1) return tokens;

        // when has enough size
        String[] previous = new String[n - 1];
        int count = 0;
        for(int i = n - 1; i > 0; i--) {
                previous[count] = tokens[tokens.length-i];
                count++;
        }
        return previous;

    }

    private String getUnifiedString(String[] previous) {
        String result = "";
        for(int i = 0; i < previous.length; i++) {
                result += previous[i]+" ";
        }
        return result;
    }

    private String[] getTokens(String input) {
        System.out.println("Using new TOkens!!!");
        StringTokenizer st = new StringTokenizer(input);
        String[] tokens = new String[st.countTokens()];

        int i = 0;
        while (st.hasMoreTokens()) {
                tokens[i] = st.nextToken();
                i++;
        }
        return tokens;
    }

    @SuppressWarnings("deprecation")
    private String getCacheFiles(Context context) throws IOException {
```

```
		// used to obtain all the directories used in cache
		String result = "";
		Path[] localCacheFiles = context.getLocalCacheFiles();
		if (localCacheFiles != null) {

			for (Path localCacheFile : localCacheFiles) {

	System.out.println(localCacheFile);
				result += " " + localCacheFile;
			}
		}
		else
			result = "--------NÃO_USADO----------";

		return result;
	}
}
```

```
package cdle.wordcount.mr;

import java.io.IOException;

import cdle.wordcount.lib.WordCountUtils;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class WordCountReducer
		extends Reducer<Text, IntWritable, Text, IntWritable> {

	private IntWritable result = new IntWritable();

	@Override
	public void reduce(Text key, Iterable<IntWritable> values, Context context)
			throws IOException, InterruptedException {

		int sum = 0;
		for (IntWritable val : values) {
			sum += val.get();
		}

		result.set(sum);
		context.write(key, result);

		// count the distincts

	context.getCounter(WordCountUtils.Statistics.Distincts).increment(1);

		// count the singletons
		if ( sum == 1 )

	context.getCounter(WordCountUtils.Statistics.Singletons).increment(1);

	}

}
```

```
package cdle.wordcount.mr;

import cdle.wordcount.lib.WordCountUtils;
import cdle.wordcount.lib.ZipFile;
import cdle.wordcount.sort.SortWordMapper;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Counter;
import org.apache.hadoop.mapreduce.Job;
import
org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import
org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.Tool;

import java.io.IOException;
import java.net.URI;

public class WordCountConfig extends Configured
      implements Tool {

   private final String DIR_SORTED = "/sorted";


   @Override
   public int run(String[] args) throws Exception {

      // set the compression out type
      Codecs codecs = new Codecs();
      codecs.setCodec( getConf() );


      // zip file to test if the directory has zip files
      ZipFile zipFile = new ZipFile();
      zipFile.convertToZip(getConf(), args[0]);

      // create the job
      Job job = Job.getInstance( getConf() );

      // set jar and name
      job.setJarByClass( WordCountApplication.class );
      job.setJobName( "Word Count" );

      // command to cache the different files
      job.addCacheFile(new URI(args[0]));

      // check the files stored in cache
      getCacheFiles(job);

      // check the number of reducers used
      int numReducers = job.getNumReduceTasks();
      System.out.println("Number of reducers: " +
numReducers);

      FileInputFormat.addInputPath(job, new Path(args[0])
);
      FileOutputFormat.setOutputPath(job, new
Path(args[1]) );


      job.setMapperClass( WordCountMapper.class );
      job.setReducerClass( WordCountReducer.class );


      System.out.println("----------------------------------------
----------------");

      // Output types of map function
      job.setMapOutputKeyClass( Text.class );
      job.setMapOutputValueClass( IntWritable.class );


      // Output types of reduce function
      job.setOutputKeyClass( Text.class );
      job.setOutputValueClass( IntWritable.class );


      boolean result = job.waitForCompletion( false );

      if ( result == true ) {

         Counter distinctsCounter =
job.getCounters().findCounter(
WordCountUtils.Statistics.Distincts );
         Counter singletonsCounter =
job.getCounters().findCounter(
WordCountUtils.Statistics.Singletons );
         Counter totalCounter =
job.getCounters().findCounter(
WordCountUtils.Statistics.TotalWords );

         System.out.printf( "Number of distinct n-grams:
%d\n", distinctsCounter.getValue() );
         System.out.printf( "Number of singletons n-grams:
%d\n", singletonsCounter.getValue() );
         System.out.printf( "Number of total n-grams: %d\n",
totalCounter.getValue() );


WordCountUtils.printPercentageSingletons(totalCounter.get
Value(),singletonsCounter.getValue());
      }
      else {
         System.out.printf( "Job failed!\n" );
      }

      boolean resultSort = jobSortByOccurrences(args);

      return result && resultSort ? 0 : 1;
```

```
    }

    public boolean jobSortByOccurrences(String[] args) throws IOException, InterruptedException, ClassNotFoundException {

        Job job = Job.getInstance( getConf() );
        job.setJarByClass( WordCountApplication.class );
        job.setJobName( "Sort Word" );

        // check the number of reducers used
        int numReducers = job.getNumReduceTasks();
        System.out.println("Number of reducers: " + numReducers);


        FileInputFormat.addInputPath(job, new Path(args[1]) );
        FileOutputFormat.setOutputPath(job, new Path(args[1]+DIR_SORTED) );

        job.setMapperClass( SortWordMapper.class );

        System.out.println("----------------------------------------------------------");


        job.setMapOutputKeyClass( LongWritable.class );
        job.setMapOutputValueClass( Text.class );


        boolean result = job.waitForCompletion( false );

        return result;
    }

    // check the files stored in cache
    public static void getCacheFiles(Job job) throws IOException {

        URI[] cacheFiles = job.getCacheFiles();
        if (cacheFiles != null) {
            System.out.println("Cache files:");
            for (URI cacheFile : cacheFiles) {
                System.out.println(cacheFile);
            }
        }
        else
            System.out.println("Não tem métodos em cache!!!!!");
    }
}
```

```
package cdle.wordcount.mr;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import java.io.File;


public class Codecs {

    private final String LOCAL_PATH = "./";

    private final String CONFIG_FILE_MAPREDUCE = "configuration-mapreduce.xml";
    private final String CONFIG_FILE_WORDCOUNT = "configuration-wordcount.xml";


    public void setCodec(Configuration conf) {

        System.out.println( "Adding resource from first configuration..." );

        // obtains the configuration to the map reduce
        if(checkLocalFile(LOCAL_PATH, CONFIG_FILE_MAPREDUCE)) {
            conf.addResource( new Path( LOCAL_PATH, CONFIG_FILE_MAPREDUCE ) );
        }

        // obtains the config to the word count
        if(checkLocalFile(LOCAL_PATH, CONFIG_FILE_WORDCOUNT)) {
            conf.addResource( new Path( LOCAL_PATH, CONFIG_FILE_WORDCOUNT ) );
        }

        String codedUsed2 = conf.get("mapreduce.output.fileoutputformat.compress.codec");
        //String codedUsed2 = conf.get("io.compression.codecs");
        //String codedUsed3 = conf.get("mapred.output.compression.codec");

        System.out.println("Codec used: "+ codedUsed2);
        System.out.println("Sensitive case: " + conf.getBoolean("wordcount.case.sensitive", true));
        System.out.println("N-grams: " + conf.getInt( "wordcount.ngrams", 1));
        System.out.println("Skip patterns: "+conf.getBoolean( "wordcount.skip.patterns", false));
    }
```

```
  public boolean checkLocalFile(String path, String fileName) {

      File file = new File(path +"/" +fileName);
      if (file.exists()) {
         System.out.println("Ficheiro de configuração existe !!");
         return true;
      } else {
         System.out.println("Ficheiro de configuração NÃOOOOO existe !!!!!!");
         return false;
      }
   }

}
```

```
package cdle.wordcount.lib;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;


public class WordCountUtils {

	private static final Log log = LogFactory.getLog( WordCountUtils.class );

	public static enum Statistics {
		TotalWords,
		Distincts,
		Singletons };

	public static void printPercentageSingletons(long totalWord, long singletons) {
		double result = (double) singletons/totalWord;
		result = result * 100; // this is in percentage
		System.out.println("Percentage of n-grams that occurs once: "+result+"%");
	}
}
```

```
package cdle.wordcount.lib;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;

import java.io.File;
import java.io.FileFilter;
import java.io.IOException;


public class ZipFile {


   public void convertToZip(Configuration conf, String input) throws IOException {

      boolean foundZip = hasZipFiles(conf, input, "zip");

      if (foundZip) {
         this.setCompressionZip(conf, input);
      }
   }

   public void setCompressionZip(Configuration conf, String input) {

      System.out.println("Conveteu para ZIP!!!!");
      // to zip files

conf.set("mapreduce.input.fileinputformat.input.dir.recursive", "true");
      conf.set("mapreduce.input.fileinputformat.input.dir", input);

conf.set("mapreduce.input.fileinputformat.input.dir.filter","*.zip");
      conf.set("mapreduce.compress.map.output","true");

conf.set("mapreduce.map.output.compress.codec","org.apache.hadoop.io.compress.ZipCodec");
   }

   public boolean hasZipFiles(Configuration conf, String directoryPath, String fileType) throws IOException {

      try {
         // first he tries on the HDFS file system
         System.out.println("TRY!!!!!");

         FileSystem fs = FileSystem.get(conf);
         Path path = new Path(directoryPath);

         FileStatus[] fileStatuses = fs.listStatus(path);

         for (FileStatus fileStatus : fileStatuses) {
            if (fileStatus.getPath().getName().endsWith(".zip")) {
               return true;
            }
         }
         return false;

      } catch (Exception e) {
         // if he can't find the dir it goes to the local file system
         System.out.println("CATCH!!!!!");
         return this.checkFileExistsWithExtension(directoryPath, fileType);
      }
   }

   public boolean checkFileExistsWithExtension(String directoryPath, String extension) {
      directoryPath = directoryPath.substring(6)+"/"; // remove file://

      File dir = new File(directoryPath);

      FileFilter fileFilter = new FileFilter() {
         public boolean accept(File file) {
            return file.getName().endsWith("." + extension);
         }
      };

      File[] files = dir.listFiles(fileFilter);
      return files != null && files.length > 0;
   }


   /*public static String convertToPlainText(Configuration conf) throws IOException {

      checkFile("", "");

      //Configuration conf = new Configuration();
      FileSystem fs = FileSystem.get(conf);
      Path path = new Path("/home/usermr/examples/input//wikipedia/enwiki-2019-04-0000000243.zip");
      FSDataInputStream inputStream = fs.open(path);
      Scanner scanner = new Scanner(inputStream, "UTF-8").useDelimiter("\\A");
      String text = scanner.hasNext() ? scanner.next() : "";
      scanner.close();
      inputStream.close();
      return text;
   }*/

   /*public static String convertToPlainText(Configuration conf) throws IOException {
      /*Configuration conf = new Configuration();
      FileSystem fs = FileSystem.get(conf);
      //Path hdfsFilePath = new Path();
```

```
        File localFile = new File("/home/usermr/examples/input//wikipedia/enwiki-2019-04-0000000243.zip");
        //FileUtil.copy(fs, hdfsFilePath, localFile, false, conf);

        Scanner scanner = new Scanner(localFile, "UTF-8").useDelimiter(" ");
        String text = scanner.hasNext() ? scanner.next() : "";
        System.out.println("RESULTADO OBTIDO::::::: "+ );
        scanner.close();
        return text;
    }*/
}
```

```
package cdle.wordcount.sort;

import java.io.IOException;
import java.util.Arrays;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class SortWordMapper extends Mapper<LongWritable, Text, LongWritable, Text> {

  private final static LongWritable occurrences = new LongWritable();


  @Override
  public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {


      System.out.println("value: "+ value);

      String[] elements = value.toString().split("\\s+");
      System.out.println("Elements: "+ Arrays.toString(elements));

      String[] wordAndCount = obtainWordAndCount(elements);
      System.out.println("wordAndCount: "+ Arrays.toString(wordAndCount));

      long count = Long.parseLong(wordAndCount[1]);


      occurrences.set(count);
      context.write(new LongWritable(count), new Text(wordAndCount[0]));
  }

  // returns an array where first element is the word and second is the number of occurrences
  private String[] obtainWordAndCount(String[] elements) {
      String wordResult = "";
      for(int i = 0; i < elements.length-1; i++) {
         if (i != elements.length-2)
            wordResult += elements[i]+" ";
         else
            wordResult += elements[i];
      }
      String[] result = {wordResult, elements[elements.length-1]};
      return result;
  }
}
```

```
package cdle.wordcount.patterns;

public class Patterns {

   public static String[] PATTERNS = {"\\.", "\\,", "\\!",
"the"};
}
```