Analysing Letter Frequency through Hadoop

Tsegay Gebrelibanos (683925), Gabriele Giudici (530344), Gabriele Billi Ciani (615560)
University of Pisa
Cloud Computing Course

A.Y. 2023-2024

1 Introduction

The aim of this project was to exploit a fully distributed Hadoop cluster to analyse letter frequency in various aspects of the English language. By defining a MapReduce algorithm, we processed text documents ranging from a few kilobytes to several gigabytes, considering only standard alphabetic characters.

In the following sections, we will delve into the pseudocode for the MapReduce algorithm, compare the Hadoop distributed execution with a local Python implementation, and explore the differences between adopting In-Mapper combining, not adopting it, and using a Combiner class. Furthermore, we will discuss the impact of varying the number of Reducer tasks, compare the performance of the Hadoop implementation with an equivalent Spark implementation, and provide both a synchronic and diachronic analysis of letter frequency in English[1].

2 Hadoop Statistics

2.1 Pseudocode

Pseudocode 1 Mapper

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all character l in d do
4:             if l is letter then
5:                 l ← lowercase(l)
6:                 Emit(letter l, count 1)
7:             end if
8:         end for
```

For more efficient executions, we used a Combiner or In-Mapper combining: see paragraph 2.3.

Pseudocode 2 Reducer*

```
1: class Reducer
2:     method Initialize
3:         totalLetters ← 0
4:         H ← new AssociativeArray
5:     method Reduce(letter l, counts [c1, c2, ...])
6:         sum ← 0
7:         for all count c in counts do
8:             sum ← sum + c
9:         end for
10:        H{l} ← sum
11:        totalLetters ← totalLetters + sum
12:    method Close
13:        for all l in H do
14:            frequency ← H{l}/totalLetters
15:            Emit(letter l, frequency)
16:        end for
```

* Note that this Reducer is suitable only when the number of reducers is equal to 1. Refer to paragraph 2.4 for variations when modifying the number of reducers.

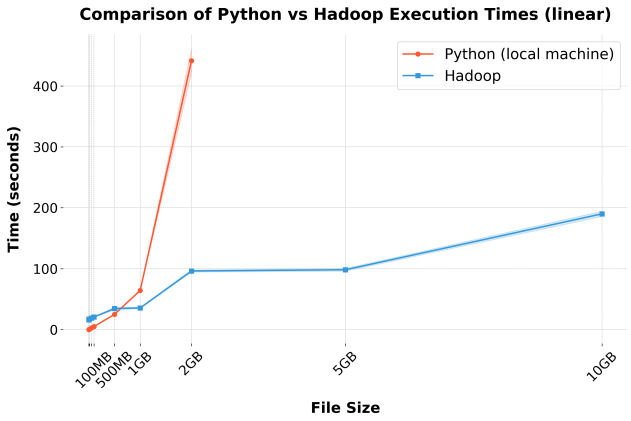
2.2 Hadoop (cluster) vs. Python (local machine)

Initially, we compared the performance of Hadoop with that of a Python program executed on a local machine to determine the file sizes for which Hadoop is advantageous. Our analysis revealed that for input files up to approximately one hundred megabytes, Python is significantly faster (see Figure 1 for more details). This is due to the substantial overhead introduced by Hadoop, which, for smaller file sizes, outweighs the speed-up benefits provided by parallelisation. It should be noted that for these tests, a Hadoop program with In-Mapper Combining and a single Job was used, as this configuration was found to be the most efficient for the input file sizes tested.

Given these results, we have decided to compare the various versions of the Hadoop programs using input files of at least 500MB in the subsequent tests.

[1] In linguistics, a synchronic analysis studies a language at a specific point in time, while a diachronic analysis examines the changes and development of a language over time.

[2] In all executions, the input dataset consistently comprised a single file. Our experimental observations indicated that datasets of equivalent size, but composed of many smaller files, significantly increase execution times. Conversely, merging these smaller files into a single file prior to execution is a very rapid operation.



Input File Size[2]	Hadoop	Python (local machine)
1KB	16.40 ± 0.55	0.004 ± 0.0049
10KB	16.00 ± 0.71	0.008 ± 0.004
100KB	16.40 ± 0.55	0.006 ± 0.0049
1MB	17.00 ± 1.00	0.05 ± 0
10MB	17.00 ± 1.00	0.486 ± 0.0049
50MB	18.80 ± 0.84	2.466 ± 0.0408
100MB	20.60 ± 0.55	4.862 ± 0.0859
500MB	34.50 ± 2.26	25.058 ± 0.5235
1GB	35.50 ± 1.22	64.216 ± 1.0611
2GB	96.20 ± 1.79	441.498 ± 19.8922
5GB	98.20 ± 2.28	-
10GB	190.00 ± 4.43	-

Figure 1: Comparison of the Mean and Standard Deviation of Processing Times (each based on five executions) for Hadoop and Python. All times are expressed in seconds.

2.3 In-Mapper Combining vs. Combiner

Pseudocode 3 Mapper (In-Mapper Combining)

```
1:  class Mapper
2:    method Initialize
3:      H ← new AssociativeArray
4:    method map(docid a, doc d)
5:      for all character l in d do
6:        if l is letter then
7:          l ← lowercase(l)
8:          H{l} ← H{l} + 1
9:        end if
10:     end for
11:   method Close
12:     for all l in H do
13:       Emit(letter l, count H{l})
14:     end for
```

Pseudocode 4 Combiner

```
1: class Combiner
2:   method Reduce(letter l, counts [c1, c2, ...])
3:     sum ← 0
4:     for all count c in counts do
5:       sum ← sum + c
6:     end for
7:       Emit(letter l, count sum)
```

This Combiner is designed for use with the Mapper described in Pseudocode 1. We chose not to use the Reducer from Pseudocode 2 as Combiner because its `cleanup()` method computes frequencies. However, the combiner's logic resembles that of the Reducer.

Configuration	500MB Execution Time (s)
Mapper without combining logic	551.0 ± 20.51
Mapper + Combiner	187.0 ± 14.27
In-Mapper Combining	34.5 ± 2.26

Table 1: Execution times for different configurations with a 500MB input file. The times are reported as Mean ± Standard Deviation (each based on five executions). The use of a Combiner mitigates the impact of the shuffle phase but does not reduce the intermediate data emitted by the Mapper. In contrast, In-Mapper Combining drastically reduces both the intermediate data and the shuffle phase impact. Some other parameters on this are shown in Table 4.

2.4 Modifying the Number of Reducer Tasks

Pseudocode 5 Reducer (Job 2)

```
1:  class Reducer
2:    method Initialize
3:      totalLetters ← 0
4:      H ← new AssociativeArray
5:    method Reduce(letter l, count c)
6:      H{l} ← c
7:      totalLetters ← totalLetters + c
8:    method Close
9:      for all l in H do
10:       frequency ← H{l}/totalLetters
11:       Emit(letter l, frequency)
12:     end for
```

In order to be able to increase the number of Reducer tasks, we require the `totalLetters` and `frequency` calculation to be performed in a separate MapReduce job. We therefore implemented another version of the program.

The Mapper for the first job remains identical to Pseudocode 1, even incorporating either a Combiner or In-Mapper Combining as recommended in paragraph 2.3. The Reducer, again, mirrors the structure outlined in Pseudocode 2, with two exceptions: the `cleanup()` method's logic and the `totalLetters` computation are transferred to the second job.

Regarding the second job, which requires only aggregation logic, no Mapper has been explicitly specified. In this scenario, Hadoop defaults to using the `IdentityMapper`, which maps inputs directly to outputs[3]. The logic for the second Reducer is shown in Pseudocode 5.

[3]See Hadoop documentation for the `IdentityMapper` here

With this two-job version of the program, we have conducted experiments varying the number of Reducer tasks in the first job (the second job must have only one Reducer task since it performs the `totalLetters` computation[4]).

Number of Reducer Tasks	500MB Execution Time (s)	10GB Execution Time (s)
1	56.6 ± 0.8	278.75 ± 2.38
2	58.0 ± 1.0	289.25 ± 3.63
5	65.25 ± 2.05	297.25 ± 6.14
13	85.25 ± 4.32	308.0 ± 10.95
26	119.75 ± 1.64	344.67 ± 7.72

Table 2: Execution times for different numbers of reducers using two input file sizes: 500MB and 10GB. The times are reported as Mean ± Standard Deviation (each based on five executions).

Experimental data reported in Table 2 demonstrate that as the number of Reducer tasks increases, the execution times increase for both chosen file sizes. Our hypothesis is that employing multiple Reducer tasks offers no benefit in this application, where the number of distinct keys is 26 (the letters of the alphabet), due to the greater overhead introduced by managing additional tasks compared to the advantage gained from parallelising this low number of keys.

2.5 MapReduce Executions Statistics

File Size	# Input Splits	CPU Time (s, average)	Cluster CPU Utilisation (average)
500MB	4	25.475	0.7384
1GB	8	52.368	1.4752
2GB	16	110.052	2.0457
5GB	40	271.240	2.7621
10GB	80	539.990	2.8421

Table 3: Number of input splits, CPU time, and Cluster CPU utilisation (CPU time/Execution time, using the same executions as in Figure 1) for different file sizes. Notice that the cluster CPU utilisation approaches 3 (the number of nodes in our cluster) as the input file size increases.

Metric	10GB	500MB		
		In-Mapper Combining	Combiner	No Combining
Physical Memory (MB)	22108	1267	1286	1246
Virtual Memory (MB)	151197	9338	9360	9373
Peak Map Physical Memory (MB)	277	275	280	282
Peak Map Virtual Memory (MB)	1868	1867	1873	1887
Peak Reduce Physical Memory (MB)	155	172	172	168
Peak Reduce Virtual Memory (MB)	1873	1873	1873	1875
Total Time Spent by All Map Tasks (s)	2483.056	84.633	695.613	1383.608
Total Time Spent by All Reduce Tasks (s)	140.541	3.461	4.475	171.12

Table 4: Hadoop memory usage and time statistics for different configurations and file sizes (all numbers are averages each based on five executions). A comparison between In-Mapper Combining, Combiner and No Combining logic is provided, in addition to Paragraph 2.3.

2.6 Hadoop vs. Spark

We implemented a program with the same logic in Spark. We compared its execution times with those of the Hadoop MapReduce program, taking into account the limitations of MapReduce (it excels at one-pass computation, while it is inefficient for multi-pass algorithms). Indeed, the single-job Hadoop MapReduce program proved to be significantly faster than the Spark one[5].

From our simulations, we observed that the Spark program performs significantly better than the Hadoop one when the input dataset consists of many small files, as shown in Table 5.

[4]It is also possible to design a two-job Hadoop program where `totalLetters` is computed in the first job, allowing the number of Reducer tasks to be varied in the second job instead.

[5]With a single 10GB input file, Hadoop took an average of 190 seconds, whereas Spark took over 2500 seconds.

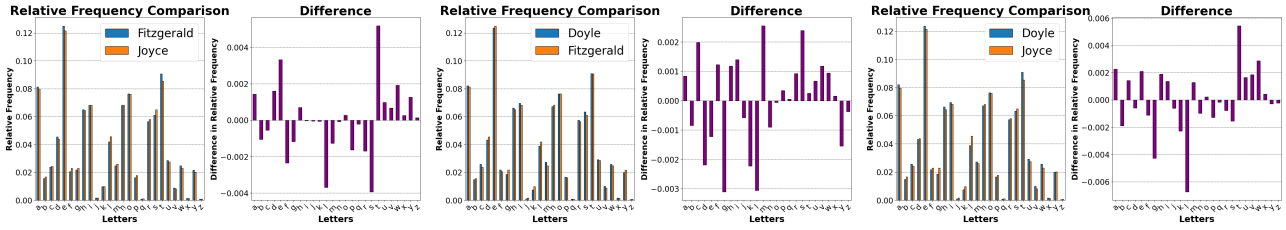
Input Dataset	Hadoop Execution Time (s)	Spark Execution Time (s)
500MB (50 files)	94.0 ± 1.79	183.6 ± 9.96
100MB (100 files)	123.2 ± 23.87	84.25 ± 18.93
500MB (500 files)	501.4 ± 2.42	207.0 ± 4.56
500MB (5000 files)	6453	1931

Table 5: Execution times for different input datasets using Hadoop and Spark. The times are reported as the Mean ± Standard Deviation, where applicable.

3 Analysis of Letter Frequency in English

3.1 Synchronic Analysis

As previously mentioned, a synchronic analysis studies a language at a specific point in time. We compared British English, American English, and Irish English by examining the works of A. C. Doyle, F. S. Fitzgerald, and J. Joyce, respectively. These authors, all born in the second half of the 19th century and dead in the first half of the 20th, represent distinct regional variations of English from the same historical period. Their literary works are available in open-source formats[6].



(a) Fitzgerald vs Joyce (b) Doyle vs Fitzgerald (c) Doyle vs Joyce

Figure 2: Comparison of letter frequency in different English variations. The graph on the left shows the difference in letter frequencies between the first mentioned author and the second one.

Looking at Figure 2b, as one would expect, the letter 's' is more frequent in British English (BrE) than in American English (AmE), while the letter 'z' is more common in AmE. This can be attributed to the different usage patterns in the two varieties (consider words like 'realise' vs. 'realize' and 'analyse' vs. 'analyze'). Furthermore, the letter 'l' is more prevalent in AmE, reflecting its different usage in the two varieties (e.g. 'skilful' vs. 'skillful', 'fulfil' vs. 'fulfill'). Additionally, the letter 'u' is more common in BrE, as exemplified by words like 'labour' vs. 'labor', 'favour' vs. 'favor', 'colour' vs. 'color'.[7]

These differences in letter frequency reflect the distinct orthographic conventions of BrE and AmE, which have evolved over time due to various historical and linguistic factors.

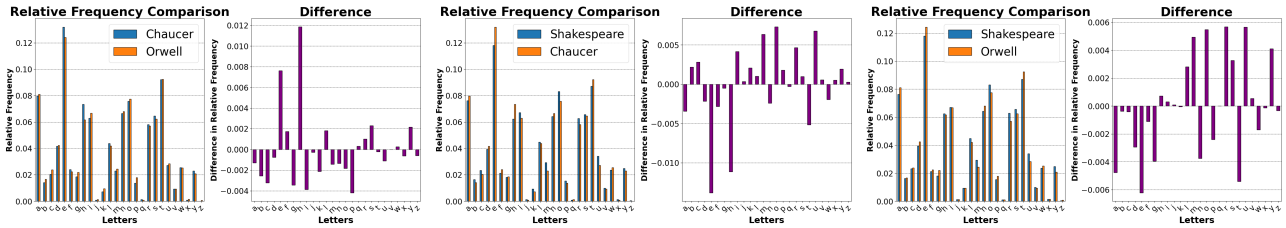
3.2 Diachronic Analysis

As previously mentioned, a diachronic analysis examines the changes and development of a language over time. We compared British English texts from different historical periods: G. Orwell (1903-1950), G. Chaucer (1343-1400), and W. Shakespeare (1564-1616). These authors were selected to represent distinct stages in the evolution of the English language. The texts for this analysis were sourced from the same repositories mentioned the synchronic analysis.

Among the various observed differences, a higher frequency of the letters 'e' and 'h' stands out in Chaucerian English compared to Shakespearean English (Figure 3b). This could be attributed to the continued use of verb endings in 'eth' and 'es' in Chaucerian English, such as 'maketh' instead of 'make' and 'loveth' instead of 'loves'.

[6]All texts were sourced from the Gutenberg Project and the Gutenberg Project Australia. The texts compared consist of collections of works by each author, totalling several megabytes of plain text each.

[7]All mentioned differences were found to be statistically significant by computing the Chi-squared test using the absolute frequencies for each letter.



(a) Chaucer vs Orwell (b) Shakespeare vs Chaucer (c) Shakespeare vs Orwell

Figure 3: Comparison of letter frequency in different historical periods of British English. The graph on the left shows the difference in letter frequencies between the first mentioned author and the second one.

These differences reflect the linguistic evolution of English between the times of Chaucer and Shakespeare.[8]

[8]All mentioned differences were found to be statistically significant by computing the Chi-squared test using the absolute frequencies for each letter.