UNIVERSITY OF PISA

MSc in Artificial Intelligence and Data Engineering

Cloud Computing

MapReduce alghoritm for letters frequency analysis through Hadoop

Group Members:
Fabiani Martina
Falaschi Tommaso
Sacco Rossana Antonella

ACADEMIC YEAR 2023/2024

Indice

Capitolo 1

Design

This project focuses on implementing a MapReduce program that calculates the frequency of each letter in a text file, through the use of Hadoop. The main objective of the study was to use Hadoop to perform a distributed analysis of the frequency of letters in large texts, applying a mapreduce algorithm to obtain accurate and scalable results on different input sizes.
The general workflow is composed by two different job:

- The first job counts the total number of letters in the text file given by input
- The second job calculates each letter frequency, using first job output

In letter counting, a single Reduce suffices due to the presence of a unique key. However, in frequency computing, where multiple different keys are present, the number of Reduce can be varied to distribute the workload and improve the efficiency of the process. The default partitioner is employed to distribute workload among the Reducers when multiple reducers are used.

This project involves developing and comparing two different workflow implementations: MapReduce with Combiner and In-Mapping Combining of which we find the pseudocidice later.

For a better efficiency and simplicity in code execution and analysis of results, some scripts are implemented automating some processes.

1.1 Combiner

1.1.1 Pseudocode for the total letter count program

Algorithm: LetterCount with Combiner
Require: Txt file
Ensure: Total number of characters

Class MAPPER

```
1:  method Map(docid a, doc value)
2:     str ← CleanText(value)     ▷ Remove non-letters, accents and set lowercase
3:     for each character c in str do
4:             EMIT(id⊥, count 1)
```

Class REDUCER

```
1:  method Reduce(id⊥, counts[n1,n2...])
2:     sum ← 0
3:     for each count n in counts[n1, n2,...] do
4:         sum ← sum + n
5:     EMIT(id⊥, count sum)
```

1.1.2 Pseudocode for the program to calculate the frequency of each letter

Algorithm: LetterFrequency with Combiner
Require: Txt file, Total number of characters in the document
Ensure: Frequency of each letter in the input file

Class MAPPER

```
1:  method Map(docid a, doc value)
2:     str ← CleanText(value)     ▷ Remove non-letters,accents and set lowercase
3:     for each character c in str do
4:             EMIT(character c, count 1)
```

Class COMBINER

```
1:  method Combine(character c, counts[n1,n2...])
2:     sum ← 0
3:     for each count n in counts[n1, n2,...] do
4:         sum ← sum + n
5:     EMIT(character c, count sum)
```

Class REDUCER

```
1:  method Initialize(Context context)
2:       TOTAL_LETTERS ← GetTotalLetters(context) ▷ Retrieve configuration
```

```
3: method Reduce(character c, counts [n1, n2,...])
4:     sum ← 0
5:     for each count n in counts [n1, n2,...] do
6:         sum ← sum + n
7:     freq ← sum / TOTAL_LETTERS
8:     EMIT(character c, frequency freq)
```

1.2 In-Mapping Combining

1.2.1 Pseudocode for the total letter count program

```
Algorithm: LetterCount with In-Mapper Combining
Require: Txt file
Ensure: Total number of characters
Class MAPPER
1: method Initialize
2:     sum ← 0
3: method Map(docid a, doc value)
4:     str ← CleanText(value)     ▷ Remove non-letters,accents and set lowercase
5:     for each character c in str do
6:         sum ← sum + 1
7: method Close
8:     EMIT(id⊥, count sum)

Class REDUCER
1: method Reduce(id⊥, counts[c1,c2...])
2:     sum ← 0
3:     for each count c in counts[c1, c2,...] do
4:         sum ← sum + c
5:     EMIT(id⊥, count sum)
```

1.2.2 Pseudocode for the program to calculate the frequency of each letter

```
Algorithm: LetterFrequency with In-Mapper Combining
Require: Txt file, Total number of characters in the document
Ensure: Frequency of each letter in the input file
Class MAPPER
1: method Initialize
2:     map ← new ASSOCIATIVEARRAY
3: method Map(docid a, doc value)
4:     str ← CleanText(value)     ▷ Remove non-letters,accents and set lowercase
5:     for each character c in str do
```

```
6:          map{c} ← map{c} + 1
7:  method Close
8:      for each character c ∈ map do
9:          EMIT(c, count map{c})
```

Class REDUCER

```
1:  method Initialize(Context context)
2:      TOTAL_LETTERS ← GetTotalLetters(context) ▷ Retrieve configuration
3:  method Reduce(character c, counts [n1, n2,...])
4:      sum ← 0
5:      for each count n in counts [n1, n2,...] do
6:          sum ← sum + n
7:      freq ← sum / TOTAL_LETTERS
8:      EMIT(character c, frequency freq)
```

Capitolo 2

Results

2.1 Analyses performed

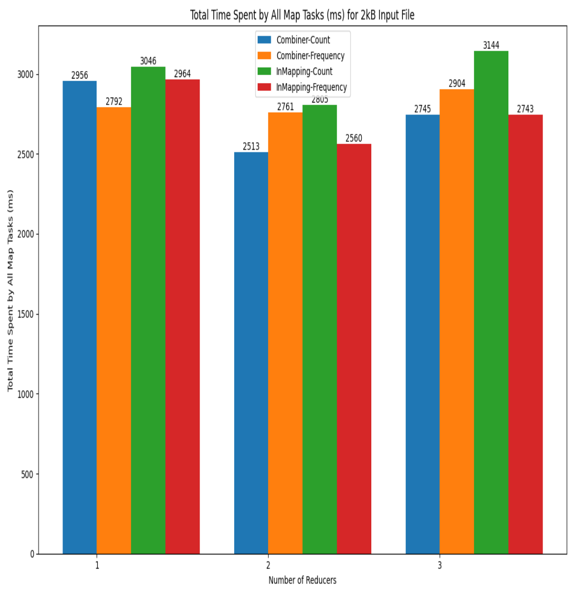
Two type of analyses were carried out:

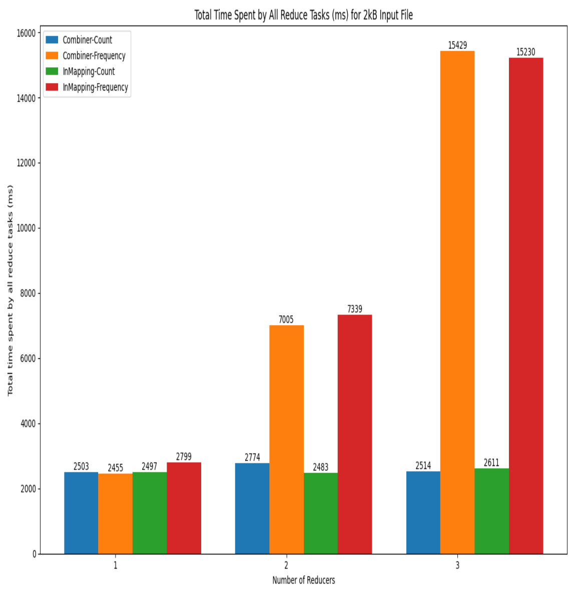
- **Performance Analysis:** MapReduce alghoritm executed with different configurations in order to meseaure its performance and understand its behaviour as settings change.
  The following configurations were used:
  - *Optimization technique:* Combiner, In-Mapper Combining.
  - *Input:* text files randomly generated with dimensions of 2.09 kB, 2.14 MB, 2.15 GB.
  - *Number of Reducers:* 1, 2, 3.
- **Statistic Analysis:** MapReduce alghoritm executed to analyse letter frequency in different languages in order to extract similarites and differences between them. The following configurations were used:
  - *Optimization technique:*Combiner.
  - *Input:*The Adventures of Pinocchio.
  - *Languages:*Dutch, English, Finnish, French, German, Italian, Portuguese, Spanish .
  - *Number of Reducers:* 1.

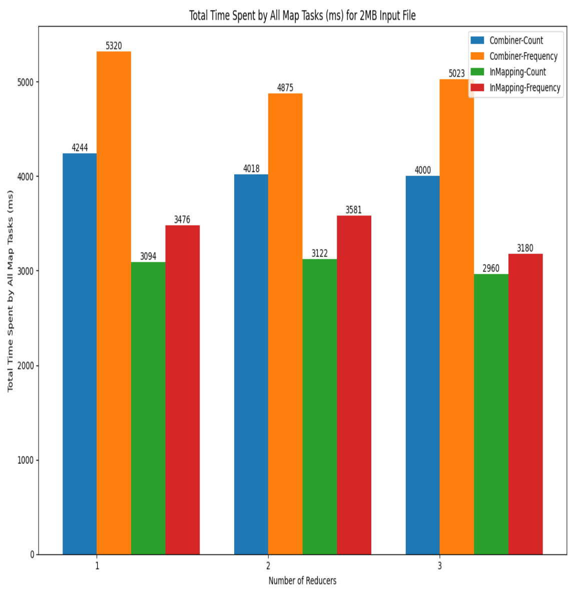
2.2 Performance Analysis

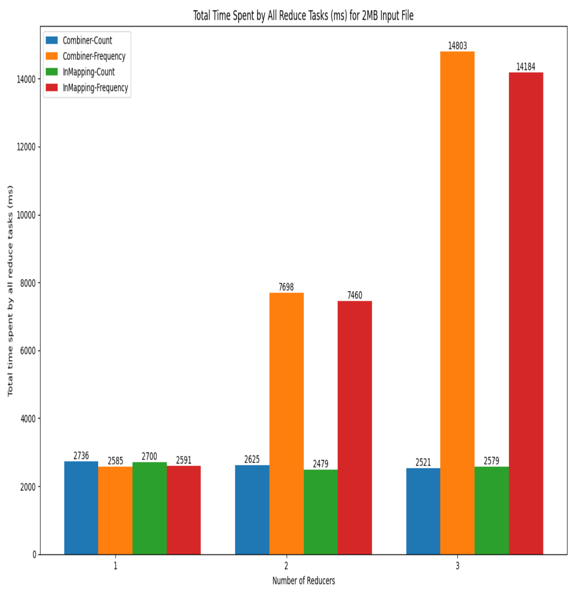
The performance analysis was carried out by taking into consideration six parameters obtainable at the end of the job execution:

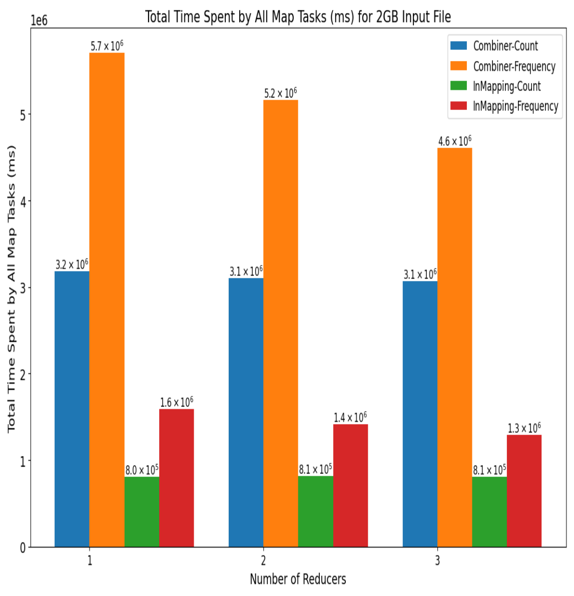
- **Total time spent by all Map tasks (ms):** provides a direct measure of the total time spent on Map tasks to understand how long the job Map phase takes.
- **Total time spent by all Reduce tasks (ms):** provides a direct measure of the total time spent on Reduce tasks to understand how long the job Reduce phase takes.
- **CPU time spent (ms):** total CPU time used by all tasks.
- **Garbage Collection (GC) time elapsed (ms):** Indicates the time spent in garbage collection to retrieve unused memory, which may affect the overall efficiency of the execution.
- **Reduce shuffle bytes:** measures the amount of data transferred during the shuffle phase.
- **Physical memory (bytes) snapshot:** total amount of physical memory used during job execution to get an overview of memory consumption.

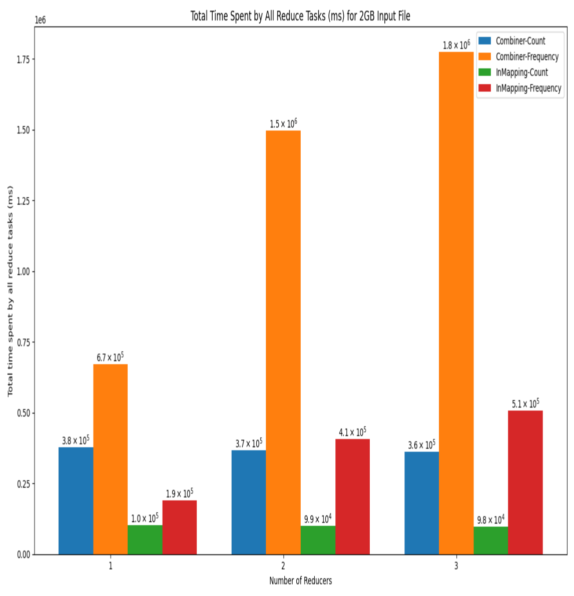

Total Time Spent by All Map Tasks (ms) for 2kB Input File
Total Time Spent by All Map Tasks (ms)
Number of Reducers
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency
2956
2792
3046
2964
2513
2761
2805
2560
2745
2904
3144
2743

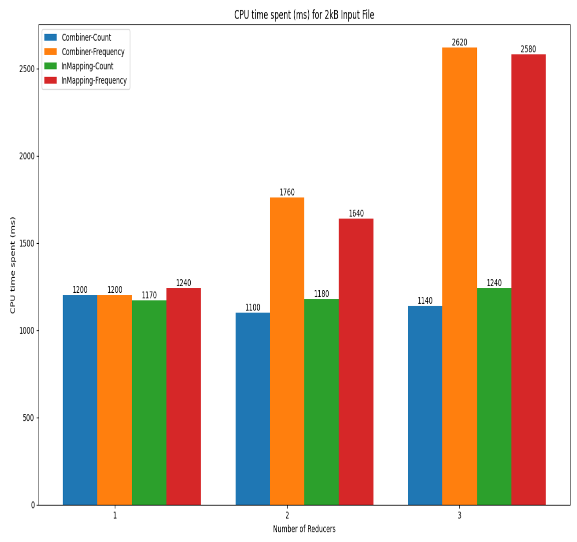

Total Time Spent by All Reduce Tasks (ms) for 2kB Input File
Total time spent by all reduce tasks (ms)
Number of Reducers
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency
2503
2455
2497
2799
2774
7005
2483
7339
2514
15429
2611
15230

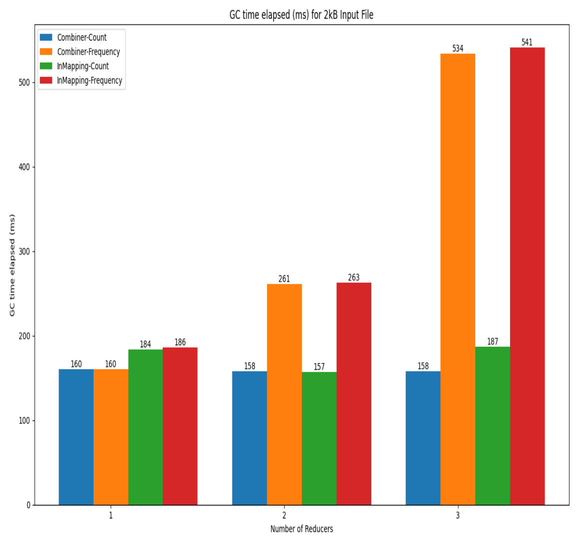

Total Time Spent by All Map Tasks (ms) for 2MB Input File
Total Time Spent by All Map Tasks (ms)
Number of Reducers
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency
4244
5320
3094
3476
4018
4875
3122
3581
4000
5023
2960
3180

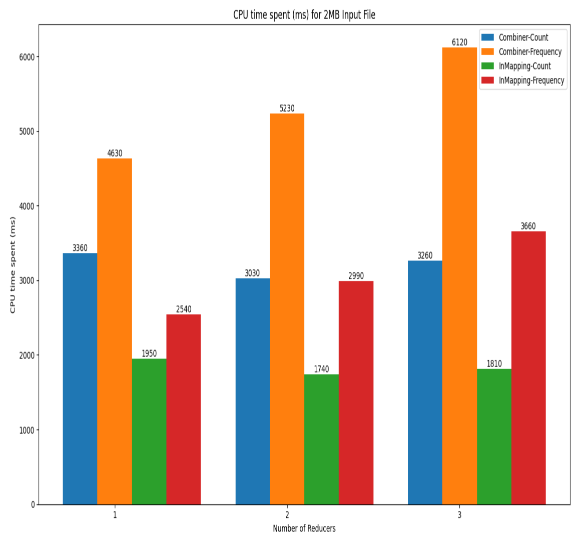

Total Time Spent by All Reduce Tasks (ms) for 2MB Input File
Total time spent by all reduce tasks (ms)
Number of Reducers
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency
2736
2585
2700
2591
2625
7698
2479
7460
2521
14803
2579
14184

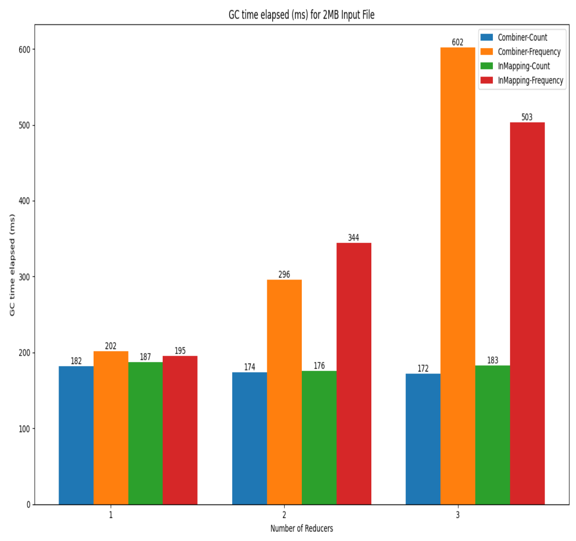

Total Time Spent by All Map Tasks (ms) for 2GB Input File
1e6
Total Time Spent by All Map Tasks (ms)
Number of Reducers
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency
3.2 x 10^6
5.7 x 10^6
8.0 x 10^5
1.6 x 10^6
3.1 x 10^6
5.2 x 10^6
8.1 x 10^5
1.4 x 10^6
3.1 x 10^6
4.6 x 10^6
8.1 x 10^5
1.3 x 10^6

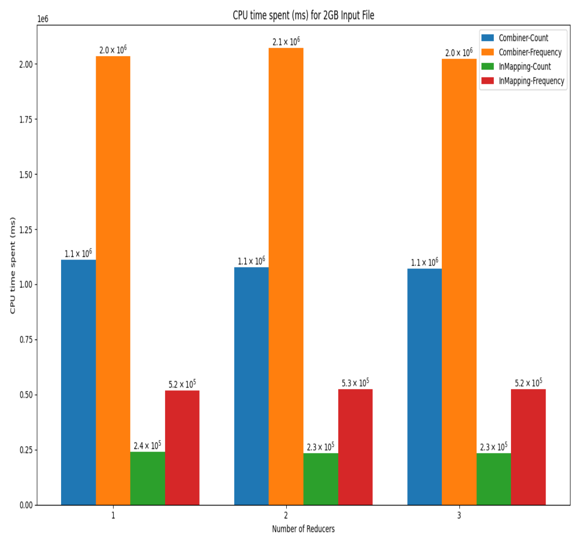

Total Time Spent by All Reduce Tasks (ms) for 2GB Input File
1e6
Total time spent by all reduce tasks (ms)
Number of Reducers
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency
3.8 x 10^5
6.7 x 10^5
1.0 x 10^5
1.9 x 10^5
3.7 x 10^5
1.5 x 10^6
9.9 x 10^4
4.1 x 10^5
3.6 x 10^5
1.8 x 10^6
9.8 x 10^4
5.1 x 10^5

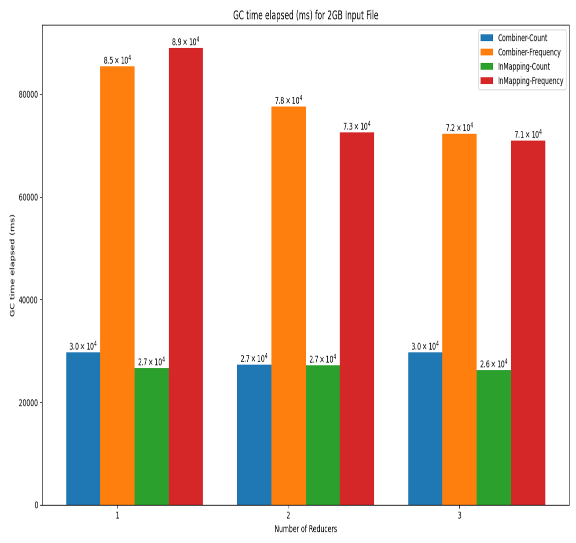

CPU time spent (ms) for 2kB Input File
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency
CPU time spent (ms)
Number of Reducers
1200
1200
1170
1240
1100
1760
1180
1640
1140
2620
1240
2580

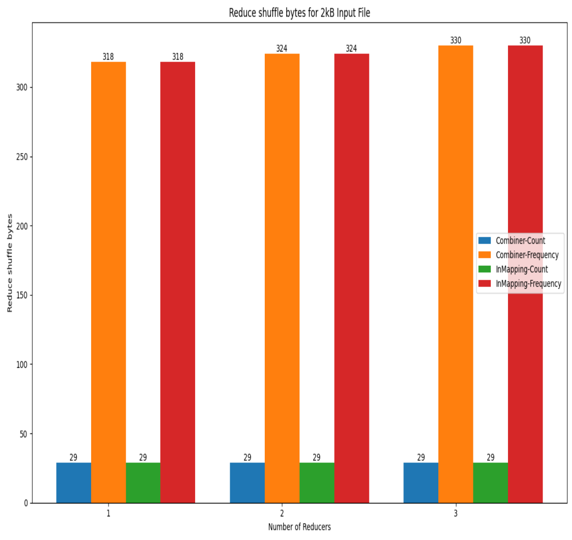

GC time elapsed (ms) for 2kB Input File
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency
GC time elapsed (ms)
Number of Reducers
160
160
184
186
158
261
157
263
158
534
187
541

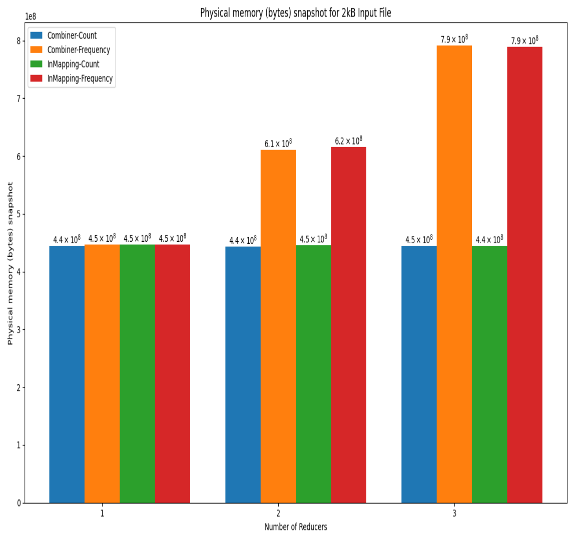

CPU time spent (ms) for 2MB Input File
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency
CPU time spent (ms)
Number of Reducers
3360
4630
1950
2540
3030
5230
1740
2990
3260
6120
1810
3660

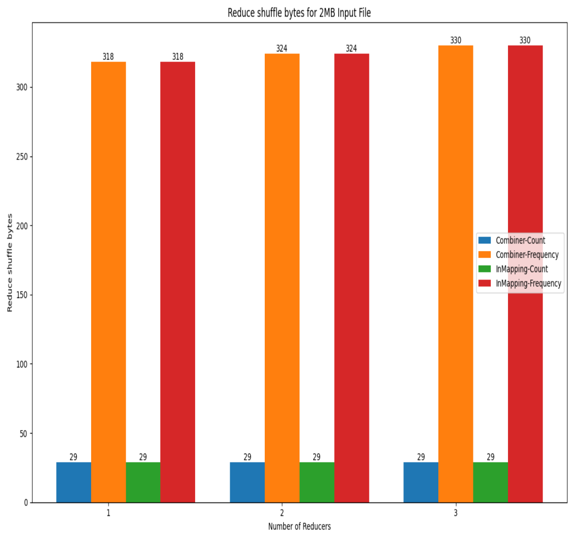

GC time elapsed (ms) for 2MB Input File
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency
GC time elapsed (ms)
Number of Reducers
182
202
187
195
174
296
176
344
172
602
183
503

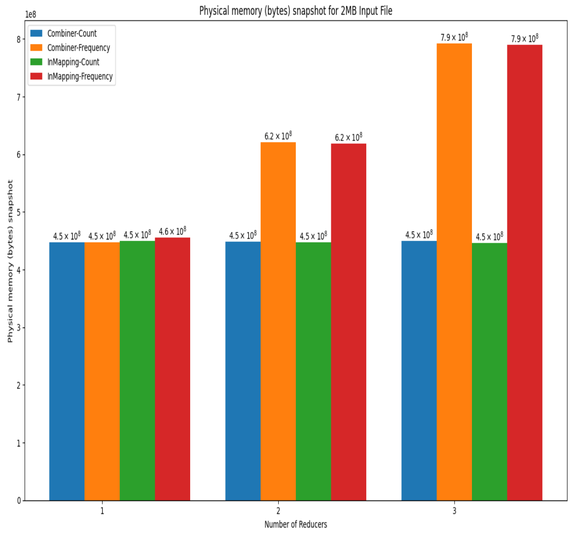

CPU time spent (ms) for 2GB Input File
1e6
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency
CPU time spent (ms)
Number of Reducers
1.1×10^6
2.0×10^6
2.4×10^5
5.2×10^5
1.1×10^6
2.1×10^6
2.3×10^5
5.3×10^5
1.1×10^6
2.0×10^6
2.3×10^5
5.2×10^5

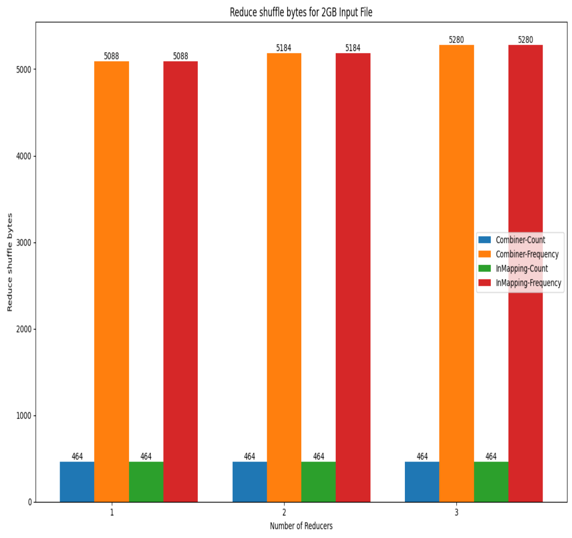

GC time elapsed (ms) for 2GB Input File
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency
GC time elapsed (ms)
Number of Reducers
3.0×10^4
8.5×10^4
2.7×10^4
8.9×10^4
2.7×10^4
7.8×10^4
2.7×10^4
7.3×10^4
3.0×10^4
7.2×10^4
2.6×10^4
7.1×10^4

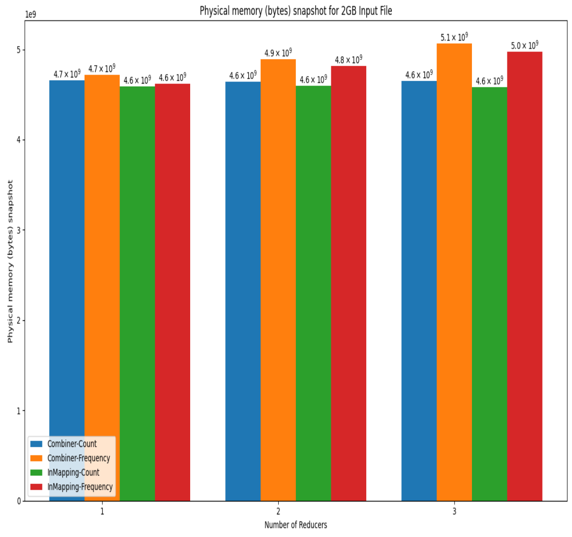

Reduce shuffle bytes for 2kB Input File
Reduce shuffle bytes
Number of Reducers
1
2
3
0
50
100
150
200
250
300
29
318
29
318
29
324
29
324
29
330
29
330
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency


Physical memory (bytes) snapshot for 2kB Input File
1e8
Physical memory (bytes) snapshot
Number of Reducers
1
2
3
0
1
2
3
4
5
6
7
8
4.4×10^8
4.5×10^8
4.5×10^8
4.5×10^8
4.4×10^8
6.1×10^8
4.5×10^8
6.2×10^8
4.5×10^8
7.9×10^8
4.4×10^8
7.9×10^8
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency


Reduce shuffle bytes for 2MB Input File
Reduce shuffle bytes
Number of Reducers
1
2
3
0
50
100
150
200
250
300
29
318
29
318
29
324
29
324
29
330
29
330
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency


Physical memory (bytes) snapshot for 2MB Input File
1e8
Physical memory (bytes) snapshot
Number of Reducers
1
2
3
0
1
2
3
4
5
6
7
8
4.5×10^8
4.5×10^8
4.5×10^8
4.6×10^8
4.5×10^8
6.2×10^8
4.5×10^8
6.2×10^8
4.5×10^8
7.9×10^8
4.5×10^8
7.9×10^8
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency


Reduce shuffle bytes for 2GB Input File
Reduce shuffle bytes
Number of Reducers
1
2
3
0
1000
2000
3000
4000
5000
464
5088
464
5088
464
5184
464
5184
464
5280
464
5280
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency


Physical memory (bytes) snapshot for 2GB Input File
1e9
Physical memory (bytes) snapshot
Number of Reducers
1
2
3
0
1
2
3
4
5
4.7×10^9
4.7×10^9
4.6×10^9
4.6×10^9
4.6×10^9
4.9×10^9
4.6×10^9
4.8×10^9
4.6×10^9
5.1×10^9
4.6×10^9
5.0×10^9
Combiner-Count
Combiner-Frequency
InMapping-Count
InMapping-Frequency

Looking at the results obtained, some considerations can be made:

- **Input file size:**We can see how the use of In-Mapper Combining significantly reduces execution times, especially as the size of the input file increases. For the memory, the behaviour of the two approaches is very similar and there are no substantial differences.
- **Number of Reducers:**Increasing the number of reducers does not lead to performance improvements; on the contrary, performance very often deteriorates, both in terms of execution time and memory consumption, as this parameter increases.

We can say that system performance depends mainly on the size of the input file and the approach chosen. With large files, it is best to use In-Mapping Combining, otherwise better performance is achieved using a Combiner while it is almost always better to have a number of reducers equals to 1.

2.3 Statistic Analysis

The analysis was carried out on the book *The Adventures of Pinocchio* by Carlo Collodi, one of the most widely translated books in the world.
For the study of letter frequency, the languages has been divided into three specific language groups:

- **Romance languages:** French, Italian, Portuguese and Spanish.
- **Germanic languages:** Dutch, English and German.
- **Finno-Ugric languages:** Finnish.

The objective was to find if there were similarities in the same language group and differences between different language groups.
The results obtained are shown in the following plots.

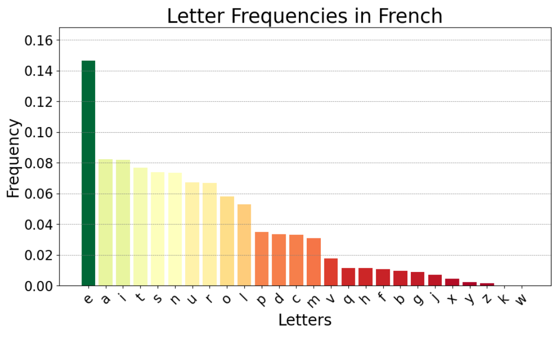


Figura 2.4: French Language Frequency Plot

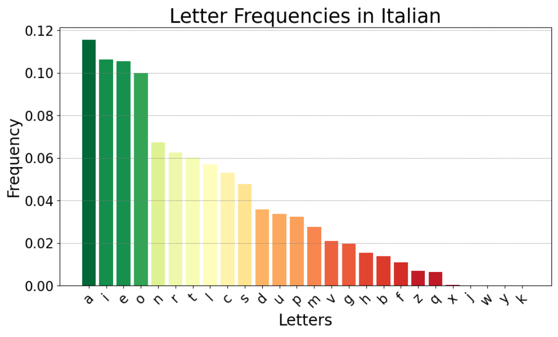


Figura 2.5: Italian Language Frequency Plot

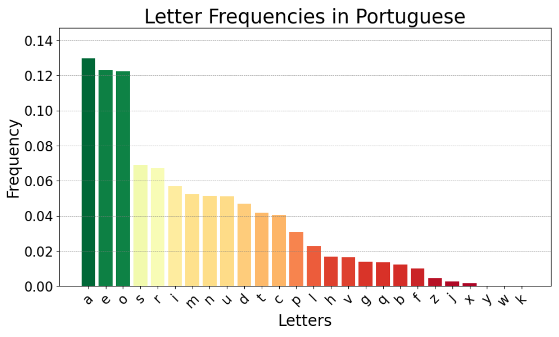


Figura 2.6: Portuguese Language Frequency Plot

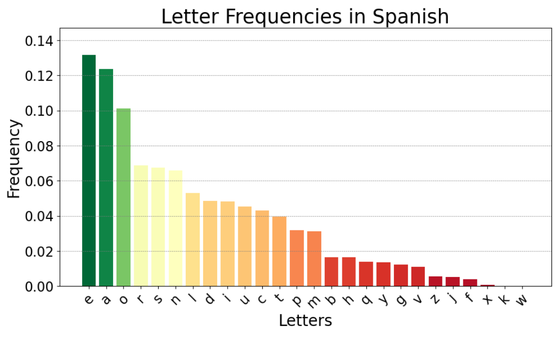


Figura 2.7: Spanish Language Frequency Plot

Figura 2.8: Frequency plots for different Romance languages.

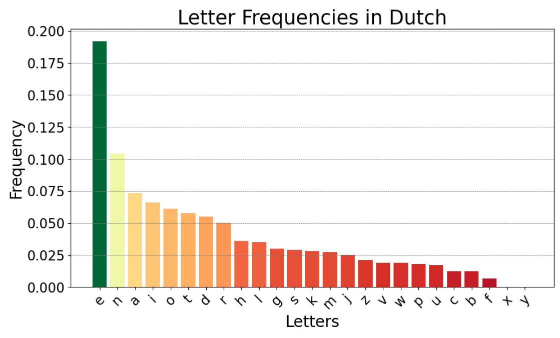


Figura 2.9: Dutch Language Frequency Plot

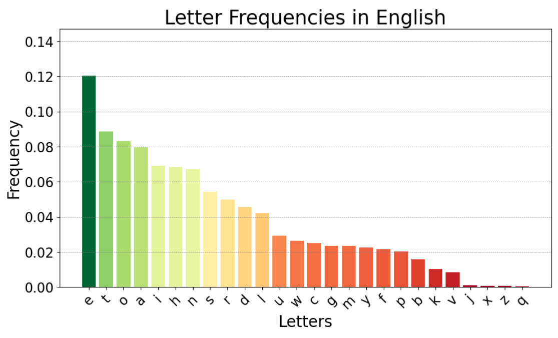


Figura 2.10: English Language Frequency Plot

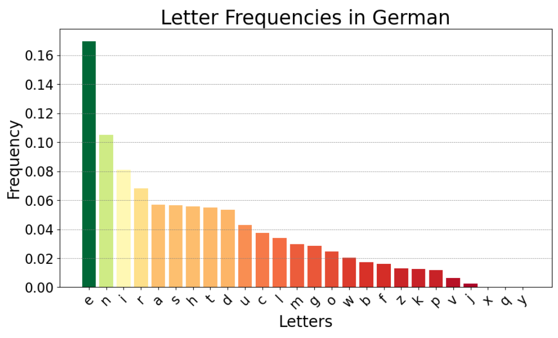


Figura 2.11: German Language Frequency Plot

Figura 2.12: Frequency plots for different Germanic languages.

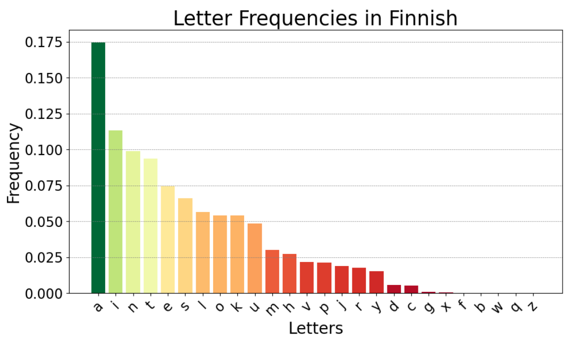


Figura 2.13: Finnish Language Frequency Plot

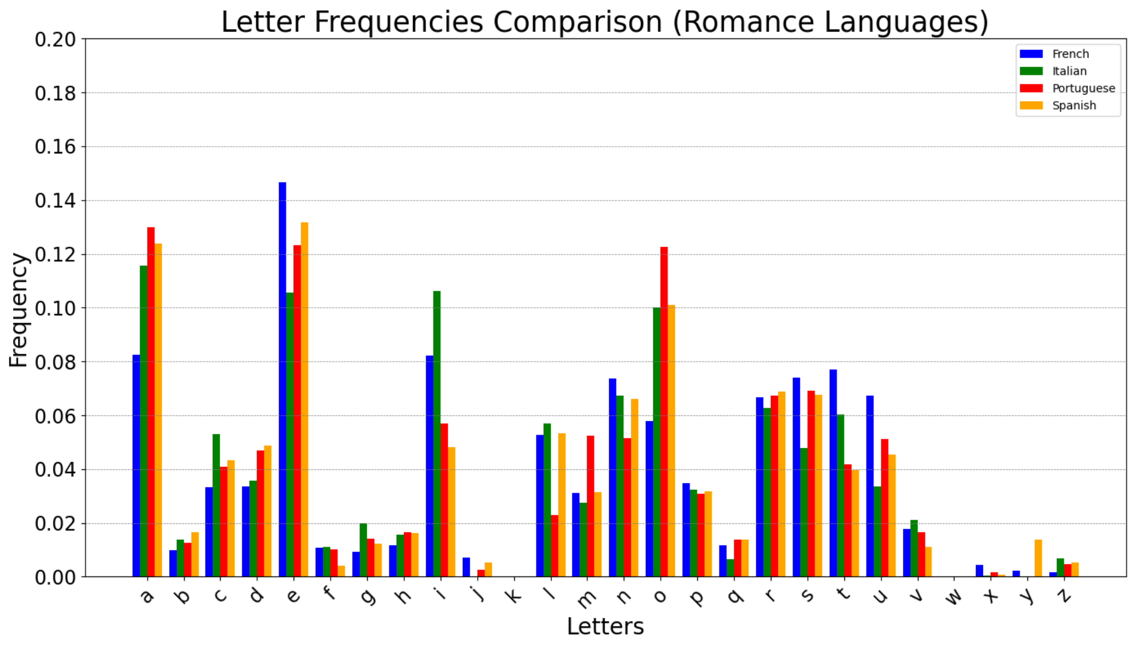


Figura 2.14: Romance Languages Comparison Plot

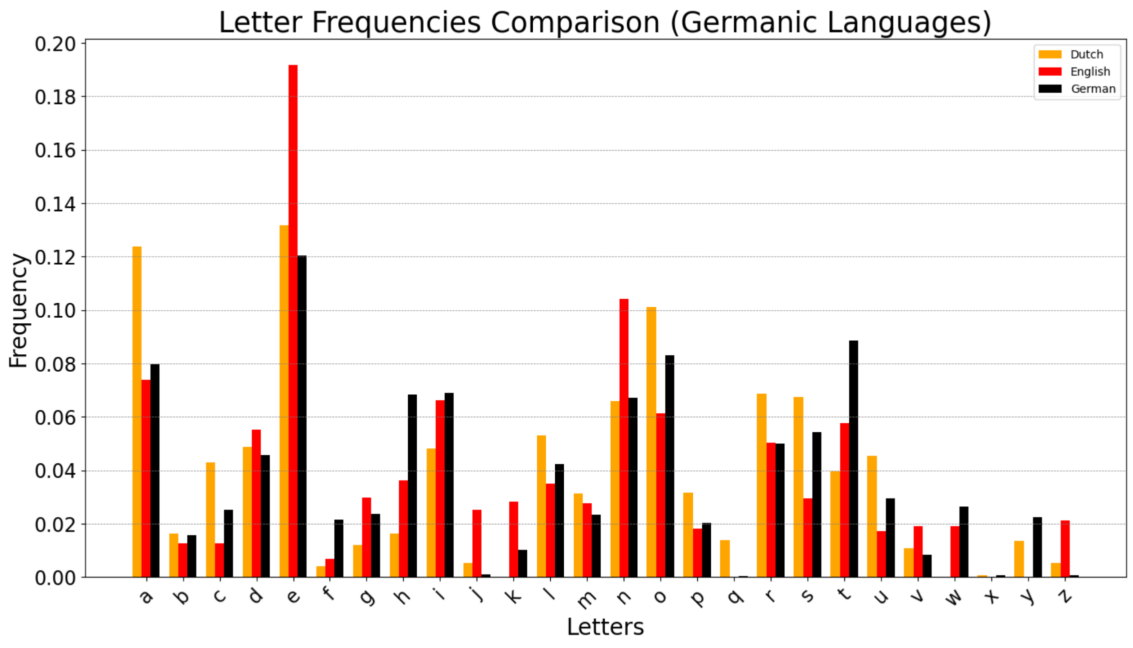


Figura 2.15: Germanic Languages Comparison Plot

Figura 2.16: Comparison plots for different language groups.

From these results we can get interesting conclusions:

- **Romance languages:**
  - Vowels such as **a, e, o** are the **most common letters** in Romance languages. The vowel **i** has a high frequency only in Italian, medium frequency in French and low frequency in Portuguese and Spanish. The vowel **u** has the lowest frequency in almost all Romance languages.
  - The *least used* letters in all Romance languages are consonants **j, k, w, x, z**. A special case is **y** letter which is almost absent in French, Italian and Portuguese, but used in Spanish although rarely.
  - We can see that the frequency of letters is almost the same in most cases, which could be the consequence of the same linguistic roots.
- **Germanic languages:**
  - In the three Germanic languages the **most frequently used letter** is **e**. Other vowels are significatally less used., giving way to a greater use of consonants suchas the letter **n**, in Dutch and German, or **t** in English.
  - The **least used letter** in the three languages is **x**. Other letters are rarely used, such as the letters **j** or **z** used in English and Dutch but not in german, the letters **k** and **w** used in English and German but not in Dutch and the letter **y** not used in English.
- **Finno-Ugric languages:**
  - The **most commonly used letter** in Finnish is **a** followed by other letters such as **i**, **n** and **t**. Compared to other languages, we see less use of the letter **e**.
  - There are **many almost unused** letters such as **x, f, b, w, q** and **z**.
  - The particular thing we can notice is the low use of letters that in other alphabets are used more frequently, such as the c. This leads to the Finnish language having a limited alphabet than other languages, mainly using only 17 of the 26 letters available.

We can conclude that Romance languages use vowels more frequently than Germanic languages, which make greater use of certain consonants with the exception of English, which has a letter frequency distribution more similar to a Romance language.

2.4 Python local execution

A possible comparison could be the local execution of a Python code that calculates the frequency of letters in a text.
To do this, we took the same texts used in the performance analysis (2kB, 2MB and 2GB input files) and used CPU time spent as the benchmark, which is the most indicative parameter concerning the state of execution and its time.
As far as MapReduce is concerned, for each dimension, the lowest CPU time spent was taken, obtained by summing the cpu time spent for job count and that for job frequency. The result obtained is shown in the plot 2.17 below: As we can see, as

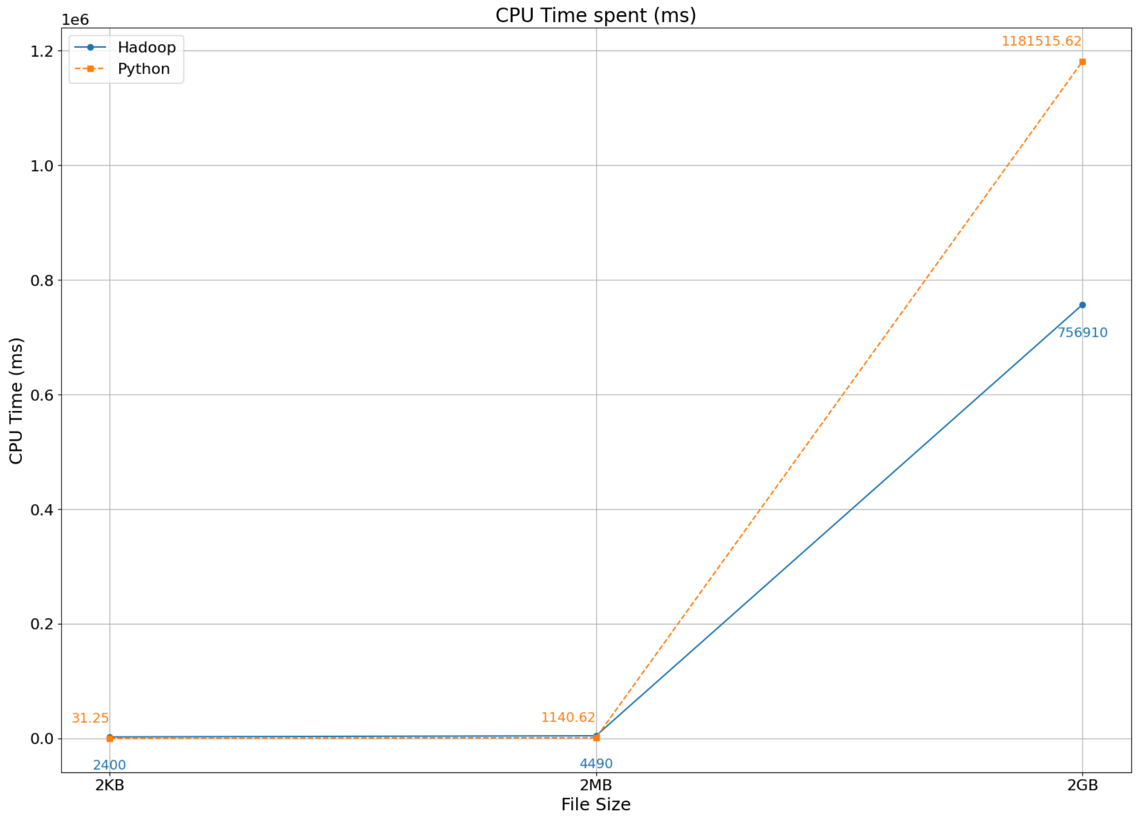


Figura 2.17: Comparisono between Hadoop execution and local Python execution

we had previously noted, CPU time spent depends on the size of the input file.
We can therefore state that for small files (2kB and 2MB), local execution with Python is, even if only slightly, more efficient, while for large files (2GB) there is a reversal, with Hadoop execution having a better CPU time spent than local execution with Python.