

UNIVERSITY OF PISA



*Artificial Intelligence and Data Engineering*

## Cloud Computing

*Hadoop Letter Frequency*

**Authors: Gemelli M. Namaki D. Nocella F.**

Academic Year 2023/2024

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Algorithm Design</b>	<b>3</b>
2.1	MapReduce with Combiner . . . . .	3
2.1.1	Pseudocode . . . . .	3
2.2	MapReduce with In-Mapper Combiner . . . . .	4
2.2.1	Pseudocode . . . . .	5
<b>3</b>	<b>Results</b>	<b>7</b>
3.1	Experimental Setup . . . . .	7
3.2	Performance Analysis . . . . .	7
3.3	Letter Frequency Analysis . . . . .	11
<b>4</b>	<b>Conclusions</b>	<b>14</b>
4.1	Performance . . . . .	14

## Introduction

The objective of this project is to implement a data processing pipeline that can handle substantial data sets, ensuring efficient computation and meaningful data insights. By exploiting the MapReduce paradigm, data processing tasks is split into two main functions: the **Mapper** and the **Reducer**.

The Mapper function processes and filters the input data, emitting **key-value pairs**, while the Reducer function aggregates and processes these pairs to produce the final output.

The project was developed using the Hadoop framework, which provides an open-source implementation of the MapReduce paradigm. Hadoop allows for the distributed processing of large data sets across clusters of computers using simple programming models. The Hadoop ecosystem also includes other tools, such as HDFS (Hadoop Distributed File System) for distributed storage, and YARN (Yet Another Resource Negotiator) for cluster resource management.

Given a text document as input, it is aimed to extract the frequency of each letter composing such document. In Order to achieve this, two differnt jobs are required: the first job is responsible for counting the number of letters in the document, while the second job is responsible for evaluating the frequency of each letter. Finally, the results obtained from the processing pipeline will be shown.

## Algorithm Design

The objective of this project is to analyze letter frequency in text documents utilizing Hadoop's MapReduce framework. Specifically, two distinct approaches were implemented to optimize the MapReduce task: the use of a Combiner and the implementation of an In-Mapper Combiner. These methods aim to enhance the efficiency of the MapReduce process by reducing the amount of data transferred between the Mapper and Reducer stages.

### MapReduce with Combiner

The Combiner is a mini-reducer that processes the output of the Mapper tasks before passing it to the Reducer. By aggregating the intermediate data locally on the mapper nodes, the Combiner reduces the volume of data shuffled across the network, thus improving the performance of the MapReduce job. It performs its operations on the same node where the mapper is running.

### Pseudocode

---

**Algorithm 1** Letter Count with Combiner

---

**Require:** Txt file

**Ensure:** Total count of each letter in the input file

#### Mapper

```
1: procedure SETUP(context)
2:   normalize  $\leftarrow$  context.getConfiguration().get("normalize")
3: end procedure
4: procedure MAP(Object key, Text value)
5:   line  $\leftarrow$  Normalize(value.toString(), normalize)  $\triangleright$  Remove accents and set lowercase
6:   for each character  $c$  in line do
7:     EmitIntermediate(LETTER_COUNT_KEY, 1)
8:   end for
9: end procedure
```

#### Combiner & Reducer

```
10: procedure REDUCE(Text key, Iterable<LongWritable> values)
11:   sum  $\leftarrow$  0
12:   for each LongWritable val in values do
13:     sum  $\leftarrow$  sum + val.get()
14:   end for
15:   Emit(key, new LongWritable(sum))
16: end procedure
```

---

---

**Algorithm 2** Letter Frequency with Combiner

---

**Require:** Txt file, Total number of characters in the txt file

**Ensure:** Frequency of each letter in the input file

**Mapper**

```
1: procedure SETUP(context)
2:   normalize  $\leftarrow$  context.getConfiguration().get("normalize")
3: end procedure
4: procedure MAP(Object key, Text value)
5:   line  $\leftarrow$  Normalize(value.toString(), normalize)     $\triangleright$  Remove accents and set lowercase
6:   for each character  $c$  in line do
7:     EmitIntermediate(String.valueOf( $c$ ), 1)
8:   end for
9: end procedure
```

**Combiner**

```
10: procedure REDUCE(Text key, Iterable<LongWritable> values)
11:   sum  $\leftarrow$  0
12:   for each LongWritable val in values do
13:     sum  $\leftarrow$  sum + val.get()
14:   end for
15:   Emit(key, new LongWritable(sum))
16: end procedure
```

**Reducer**

```
17: procedure SETUP(context)
18:   letterCount  $\leftarrow$  context.getConfiguration().getLong("letterCountValue", 1)
19: end procedure
20: procedure REDUCE(Text key, Iterable<LongWritable> values)
21:   sum  $\leftarrow$  0
22:   for each LongWritable val in values do
23:     sum  $\leftarrow$  sum + val.get()
24:   end for
25:   freq  $\leftarrow$  (double) sum / (double) letterCount
26:   Emit(key, new DoubleWritable(freq))
27: end procedure
```

---

## MapReduce with In-Mapper Combiner

The In-Mapper Combiner combines the mapping and combining steps within the Mapper itself. This method involves accumulating the results in a data structure within the Mapper, which is then emitted at the end of the mapping phase. This approach minimizes the overhead of multiple data passes by efficiently combining intermediate results within the Mapper, reducing the need for external Combiner steps and further optimizing network usage and processing time.

---

**Algorithm 3** Letter Count with In-Mapper Combiner

---

**Require:** Txt file

**Ensure:** Total count of each letter in the input file

**CountMapper**

```
1: private map  $\leftarrow \{\}$ 
2: private normalize
3: procedure SETUP(Context context)
4:   normalize  $\leftarrow$  context.getConfiguration().get("normalize")
5: end procedure
6: procedure MAP(Object key, Text value)
7:   line  $\leftarrow$  Normalize(value.toString(), normalize)
8:   for each character  $c$  in line do
9:     if  $c$  is a letter then
10:      if map contains  $c$  then
11:        map[ $c$ ]  $\leftarrow$  map[ $c$ ] + 1
12:      else
13:        map[ $c$ ]  $\leftarrow$  1
14:      end if
15:    end if
16:  end for
17: end procedure
18: procedure CLEANUP(Context context)
19:   for each entry  $\langle k, v \rangle$  in map do
20:     Emit( $k, v$ )
21:   end for
22: end procedure
```

**CountReducer**

```
23: procedure REDUCE(Text key, Iterable<LongWritable> values)
24:   sum  $\leftarrow$  0
25:   for each value in values do
26:     sum  $\leftarrow$  sum + value
27:   end for
28:   Emit(key, sum)
29: end procedure
```

---

---

**Algorithm 4** Letter Frequency with In-Mapper Combiner

---

**Require:** Txt file, Total number of characters in the txt file

**Ensure:** Frequency of each letter in the input file

**CountMapper**

```
1: private map  $\leftarrow \{\}$ 
2: private normalize
3: procedure SETUP(Context context)
4:   normalize  $\leftarrow$  context.getConfiguration().get("normalize")
5: end procedure
6: procedure MAP(Object key, Text value)
7:   line  $\leftarrow$  Normalize(value.toString(), normalize)
8:   for each character  $c$  in line do
9:     if  $c$  is a letter then
10:      if map contains  $c$  then
11:        map[ $c$ ]  $\leftarrow$  map[ $c$ ] + 1
12:      else
13:        map[ $c$ ]  $\leftarrow$  1
14:      end if
15:    end if
16:  end for
17: end procedure
18: procedure CLEANUP(Context context)
19:   for each entry  $\langle k, v \rangle$  in map do
20:     Emit( $k, v$ )
21:   end for
22: end procedure
```

**CountReducer**

```
23: private letterCount
24: procedure SETUP(Context context)
25:   letterCount  $\leftarrow$  context.getConfiguration().getLong("letterCountValue", 1)
26: end procedure
27: procedure REDUCE(Text key, Iterable<LongWritable> values)
28:   sum  $\leftarrow$  0
29:   for each value in values do
30:     sum  $\leftarrow$  sum + value
31:   end for
32:   freq  $\leftarrow$  (double) sum / (double) letterCount
33:   Emit(key, freq)
34: end procedure
```

---

# Results

## Experimental Setup

Executing the MapReduce workflow with different inputs, and configurations:

- **Input size:** the size of the input file is varied to evaluate the performance of the MapReduce workflow.
  - **Paradise Lost** ~ 310 kb
  - **Moby Dick** ~ 421kb
  - **Frankenstein** ~ 440 kb
  - **Divina Commedia** ~ 600 kb
  - **Gerusalemme Liberata** ~ 691 kb
  - **Promessi Sposi** ~ 1.440 kb
  - **Test file** (random generated sequence of char) of ~ 800 Mb.
- **Number of mappers:** Hadoop handles this step
- **Number of reducers:** From one up to three reducers are used for letter frequency

## Performance Analysis

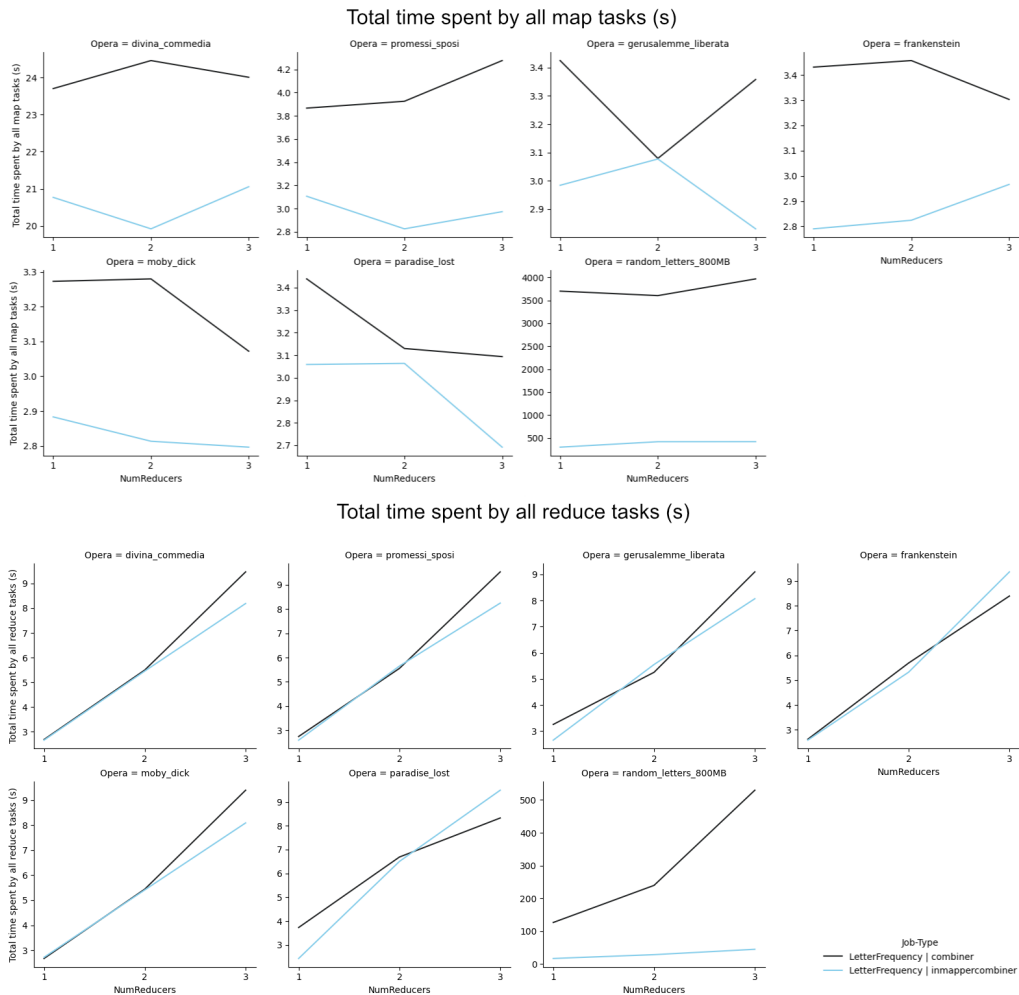


Figure 1: TotalTimeMapReduce



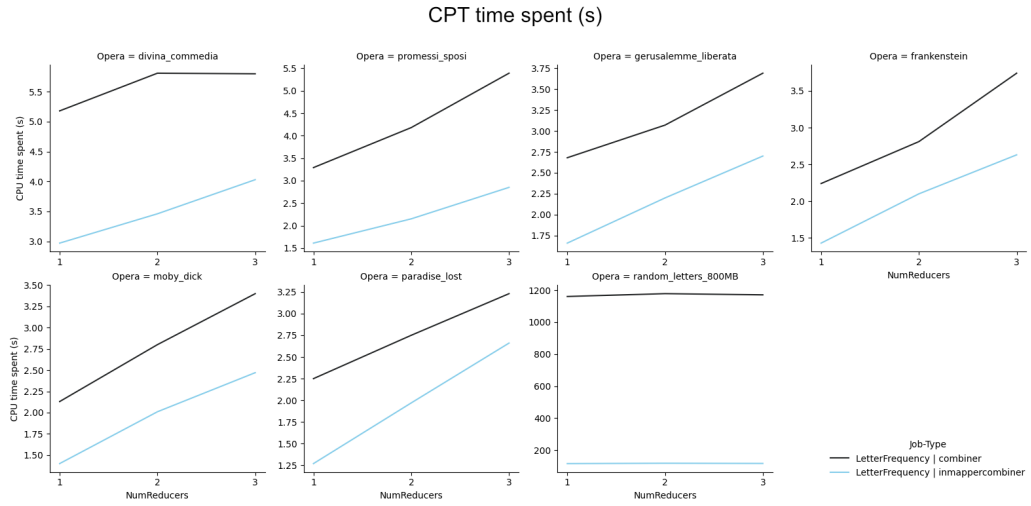


Figure 2: Cpu Time

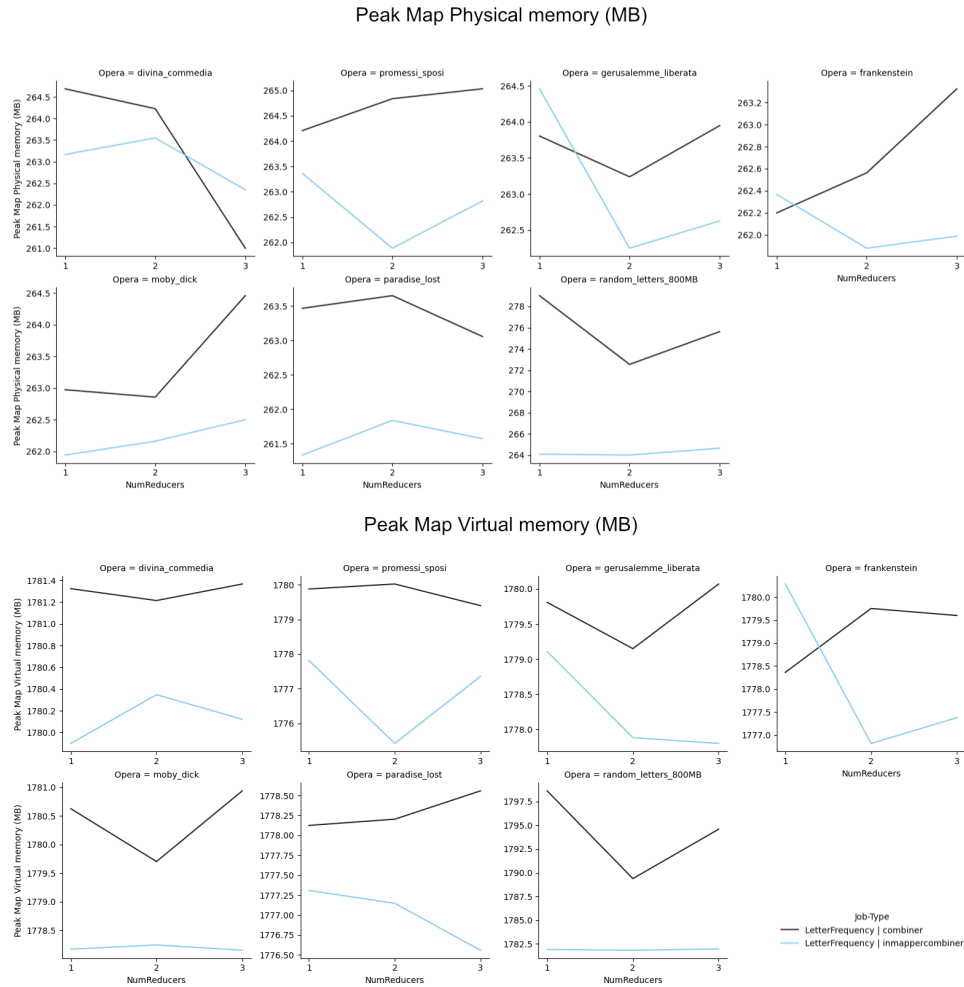


Figure 3: Peak Map Memory

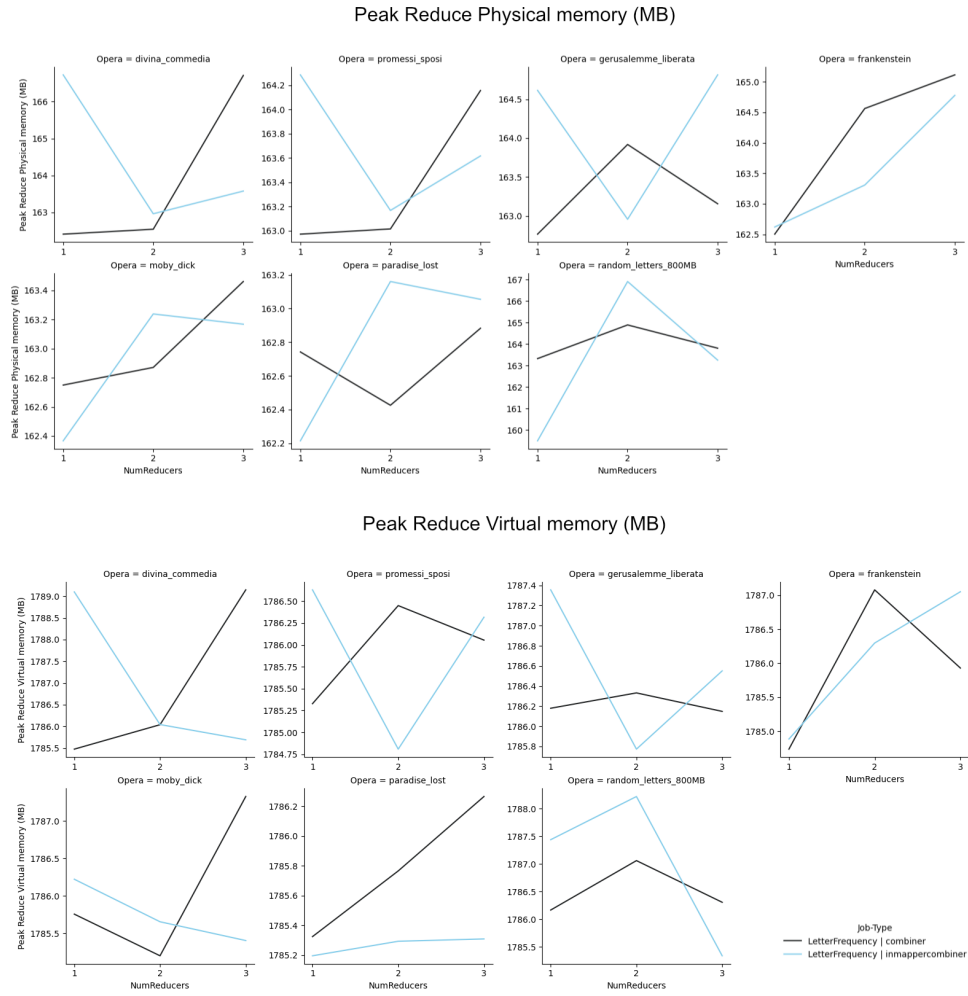


Figure 4: Peak Reduce Memory

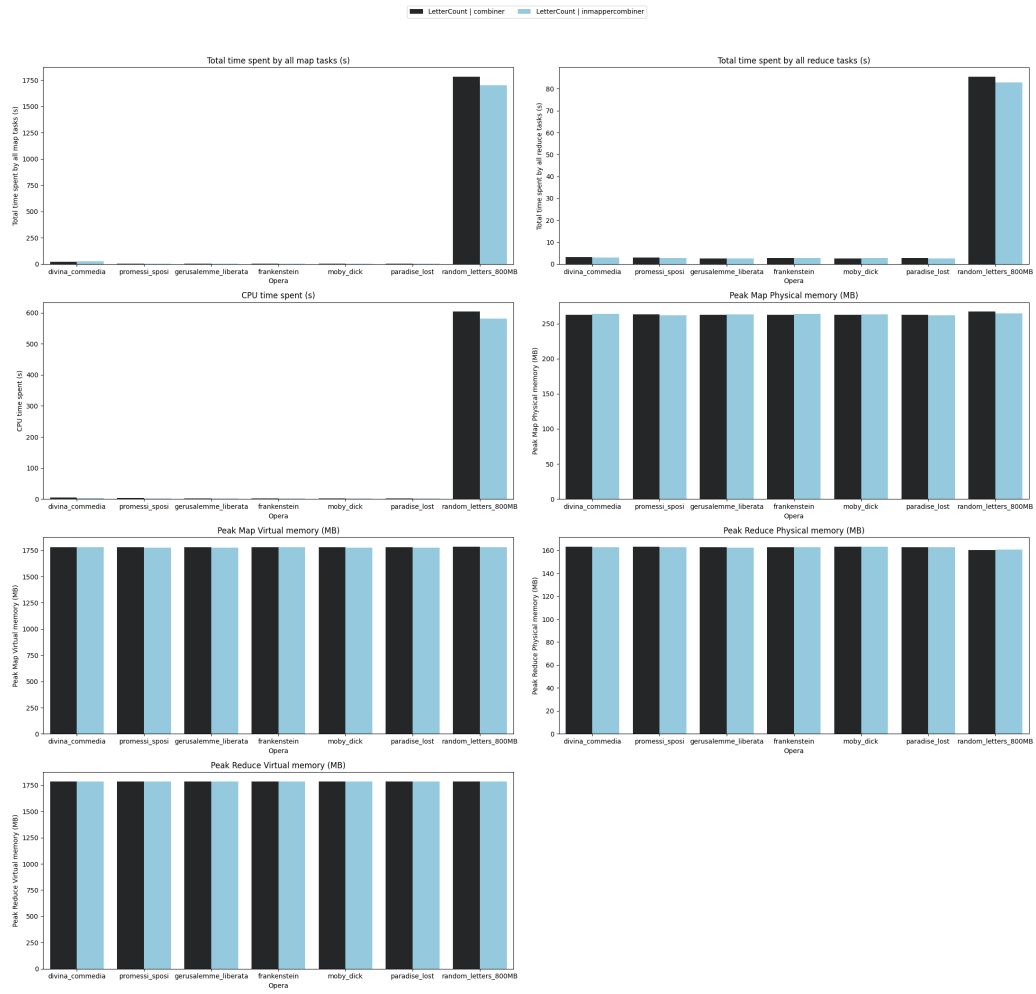


Figure 5: Performance Letter Count

# Letter Frequency Analysis

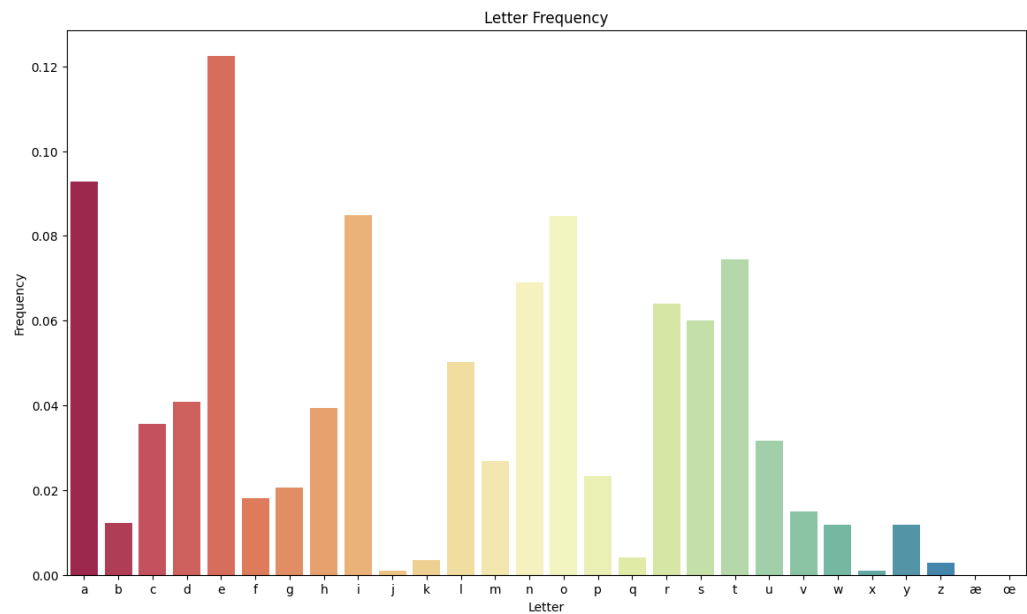


Figure 6: Letter Frequency Distribution

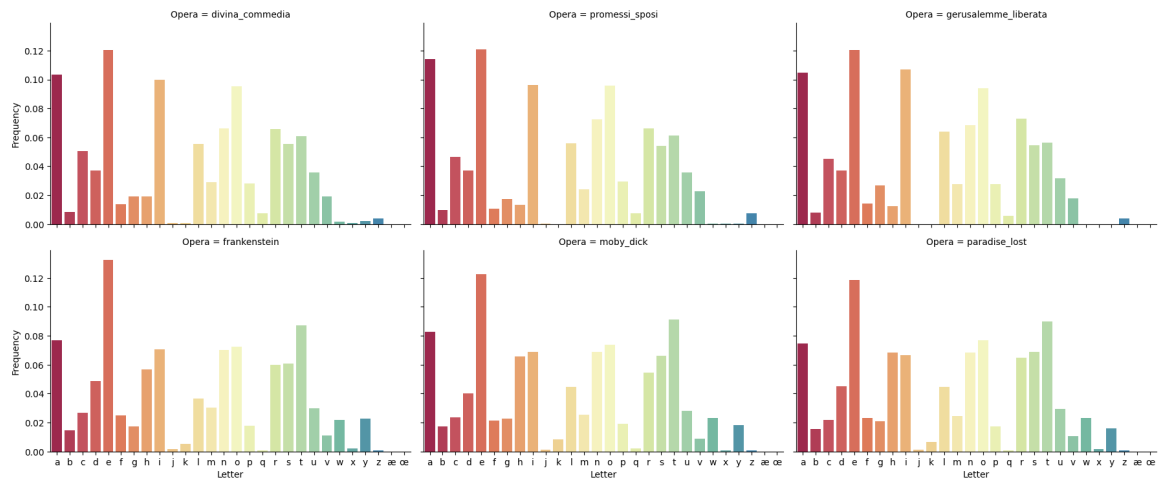


Figure 7: Letter Frequency Distribution for different Operas

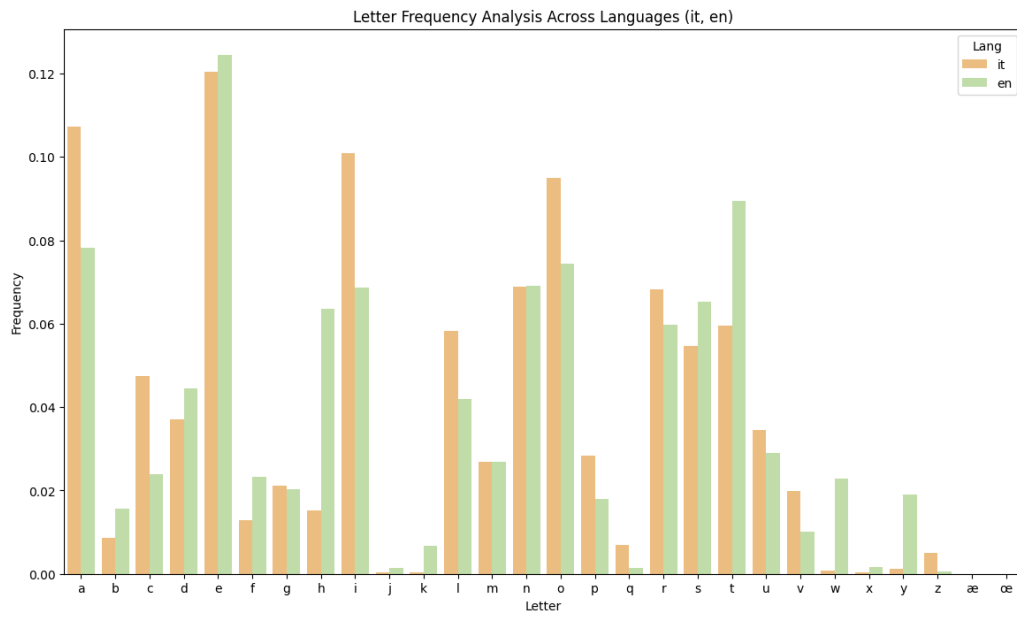


Figure 8: Letter Frequency Distribution for italian and english

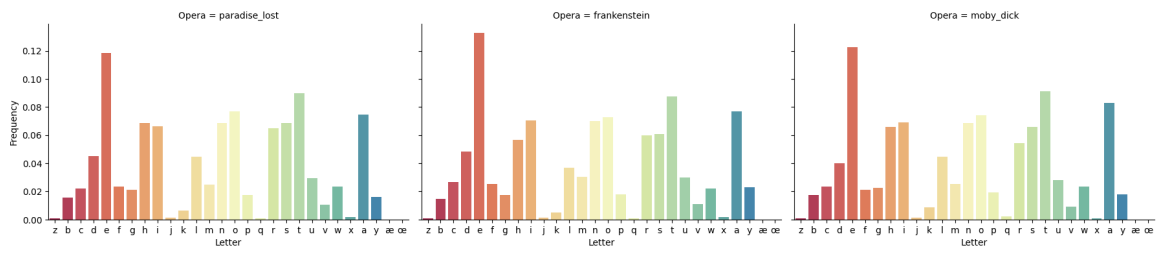


Figure 9: Letter Frequency Distribution for english

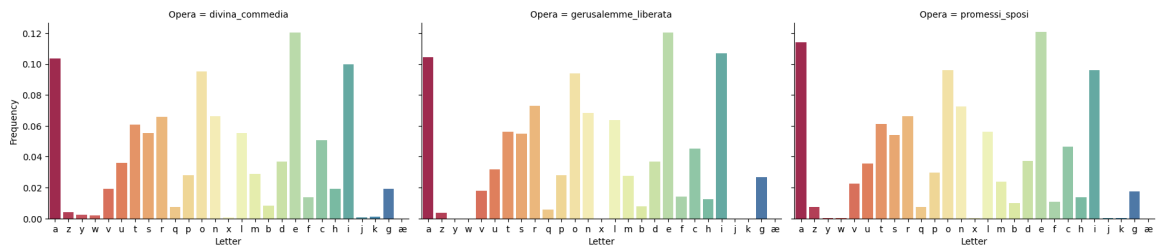


Figure 10: Letter Frequency Distribution for italian

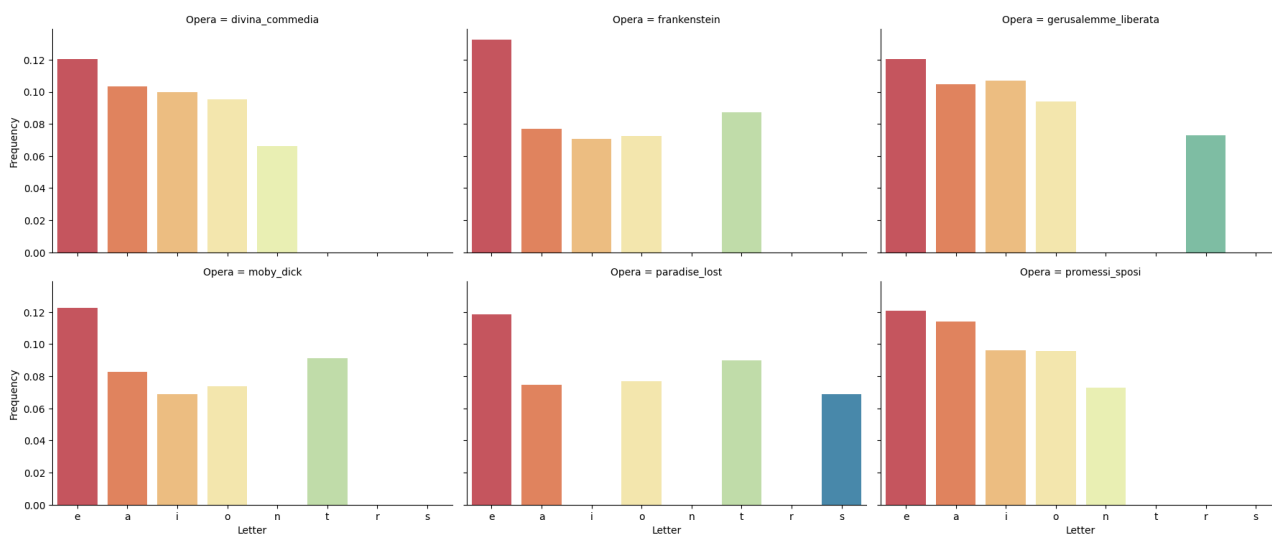


Figure 11: Top 5 letters

## Conclusions

### Performance

The performance evaluation of the letter frequency job and letter count job shows the impact of input size and number of reducers. Moreover, the comparison between the Combiner and In-Mapper Combiner implementations highlights the better performances obtained by using the *In-Mapper Combining* design pattern.

- **Impact of Input Size:** Larger input sizes, such as the 800 MB test file, grater the total processing time. This effect is evident across both the Combiner and In-Mapper Combiner implementations.
- **Number of Reducers:** It is shown that increasing the number of reducers do not lead to an overall significant improvement. This is due to the overhead of managing multiple reducers and the limited amount of data to process. In most cases, the performance is optimal with just two reducers.
- **Combiner vs In-Mapper Combiner:** The In-Mapper Combiner approach consistently outperforms the traditional Combiner method. The In-Mapper Combiner design pattern reduces the volume of intermediate data and minimizes data transfer overhead, leading to a more efficient MapReduce workflow.