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

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 ← context.getConfiguration().get("normalize")
 3: end procedure
 4: procedure MAP(Object key, Text value)
 5:     line ← Normalize(value.toString(), normalize)        ▷ 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 ← 0
12:     for each LongWritable val in values do
13:         sum ← 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 ← context.getConfiguration().get("normalize")
 3: end procedure
 4: procedure MAP(Object key, Text value)
 5:     line ← Normalize(value.toString(), normalize)      ▷ 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 ← 0
12:     for each LongWritable val in values do
13:         sum ← sum + val.get()
14:     end for
15:     Emit(key, new LongWritable(sum))
16: end procedure

    Reducer
17: procedure SETUP(context)
18:     letterCount ← context.getConfiguration().getLong("letterCountValue", 1)
19: end procedure
20: procedure REDUCE(Text key, Iterable<LongWritable> values)
21:     sum ← 0
22:     for each LongWritable val in values do
23:         sum ← sum + val.get()
24:     end for
25:     freq ← (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.

Pseudocode

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 ← {}
 2: private normalize
 3: procedure SETUP(Context context)
 4:     normalize ← context.getConfiguration().get(”normalize”)
 5: end procedure
 6: procedure MAP(Object key, Text value)
 7:     line ← 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[$c$] ← map[$c$] + 1
12:             else
13:                 map[$c$] ← 1
14:             end if
15:         end if
16:     end for
17: end procedure
18: procedure CLEANUP(Context context)
19:     for each entry $< k, v >$ in map do
20:         Emit($k$, $v$)
21:     end for
22: end procedure

    CountReducer
23: procedure REDUCE(Text key, Iterable<LongWritable> values)
24:     sum ← 0
25:     for each value in values do
26:         sum ← 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
private map ← {}
private normalize
procedure Setup(Context context)
    normalize ← context.getConfiguration().get("normalize")
end procedure
procedure Map(Object key, Text value)
    line ← Normalize(value.toString(), normalize)
    for each character $c$ in line do
        if $c$ is a letter then
            if map contains $c$ then
                map[$c$] ← map[$c$] + 1
            else
                map[$c$] ← 1
            end if
        end if
    end for
end procedure
procedure Cleanup(Context context)
    for each entry $< k, v >$ in map do
        Emit($k$, $v$)
    end for
end procedure

CountReducer
private letterCount
procedure Setup(Context context)
    letterCount ← context.getConfiguration().getLong("letterCountValue", 1)
end procedure
procedure Reduce(Text key, Iterable<LongWritable> values)
    sum ← 0
    for each value in values do
        sum ← sum + value
    end for
    freq ← (double) sum / (double) letterCount
    Emit(key, freq)
end procedure
```

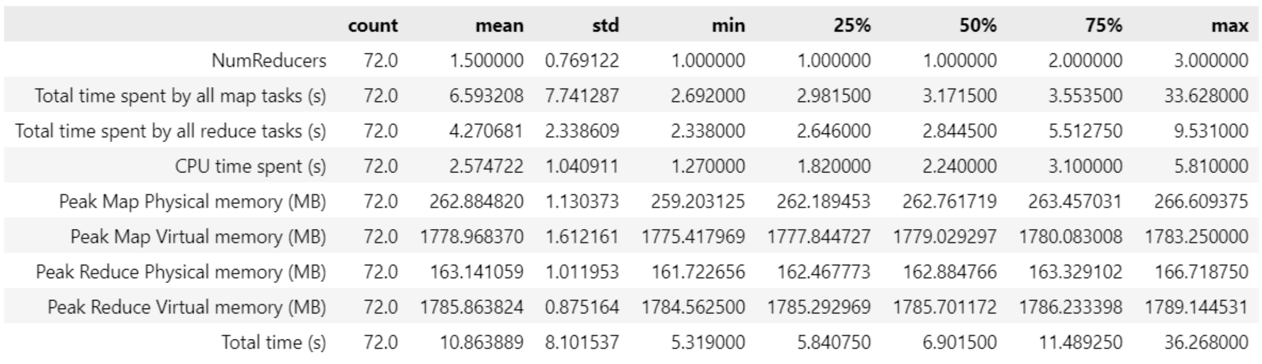
Results

Experimental Setup

Executing the MapReduce workflow with different inputs, and configurations:

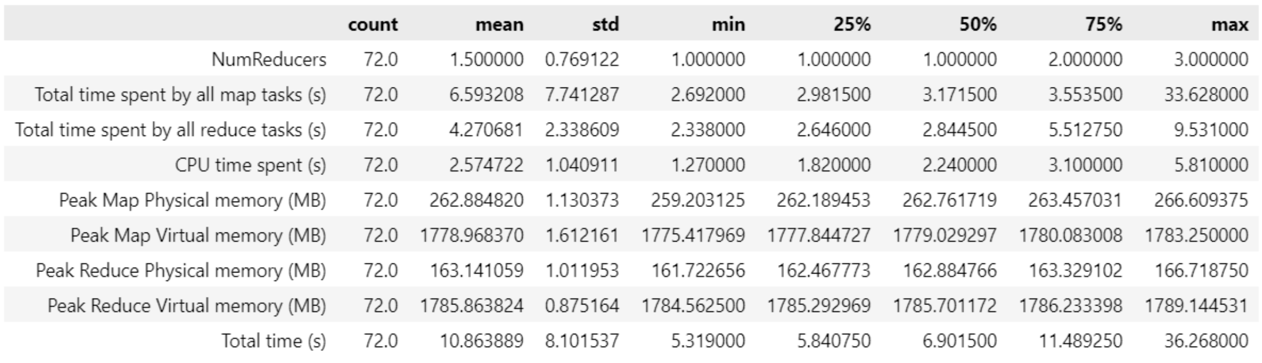
- **Input size**: the size of the input file is varied to evaluate the performance of the MapReduce workflow.
  - **Paradise Lost** ∼ 310 kb
  - **Moby Dick** ∼ 421kb
  - **Frankenstein** ∼ 440 kb
  - **Divina Commedia** ∼ 600 kb
  - **Gerusalemme Liberata** ∼ 691 kb
  - **Promessi Sposi** ∼ 1.440 kb
  - **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 Evaluation



	count	mean	std	min	25%	50%	75%	max
NumReducers	72.0	1.500000	0.769122	1.000000	1.000000	1.000000	2.000000	3.000000
Total time spent by all map tasks (s)	72.0	6.593208	7.741287	2.692000	2.981500	3.171500	3.553500	33.628000
Total time spent by all reduce tasks (s)	72.0	4.270681	2.338609	2.338000	2.646000	2.844500	5.512750	9.531000
CPU time spent (s)	72.0	2.574722	1.040911	1.270000	1.820000	2.240000	3.100000	5.810000
Peak Map Physical memory (MB)	72.0	262.884820	1.130373	259.203125	262.189453	262.761719	263.457031	266.609375
Peak Map Virtual memory (MB)	72.0	1778.968370	1.612161	1775.417969	1777.844727	1779.029297	1780.083008	1783.250000
Peak Reduce Physical memory (MB)	72.0	163.141059	1.011953	161.722656	162.467773	162.884766	163.329102	166.718750
Peak Reduce Virtual memory (MB)	72.0	1785.863824	0.875164	1784.562500	1785.292969	1785.701172	1786.233398	1789.144531
Total time (s)	72.0	10.863889	8.101537	5.319000	5.840750	6.901500	11.489250	36.268000

Figure 1: Statics' insights on Operas



	count	mean	std	min	25%	50%	75%	max
NumReducers	72.0	1.500000	0.769122	1.000000	1.000000	1.000000	2.000000	3.000000
Total time spent by all map tasks (s)	72.0	6.593208	7.741287	2.692000	2.981500	3.171500	3.553500	33.628000
Total time spent by all reduce tasks (s)	72.0	4.270681	2.338609	2.338000	2.646000	2.844500	5.512750	9.531000
CPU time spent (s)	72.0	2.574722	1.040911	1.270000	1.820000	2.240000	3.100000	5.810000
Peak Map Physical memory (MB)	72.0	262.884820	1.130373	259.203125	262.189453	262.761719	263.457031	266.609375
Peak Map Virtual memory (MB)	72.0	1778.968370	1.612161	1775.417969	1777.844727	1779.029297	1780.083008	1783.250000
Peak Reduce Physical memory (MB)	72.0	163.141059	1.011953	161.722656	162.467773	162.884766	163.329102	166.718750
Peak Reduce Virtual memory (MB)	72.0	1785.863824	0.875164	1784.562500	1785.292969	1785.701172	1786.233398	1789.144531
Total time (s)	72.0	10.863889	8.101537	5.319000	5.840750	6.901500	11.489250	36.268000

Figure 2: Statics' insights on Test

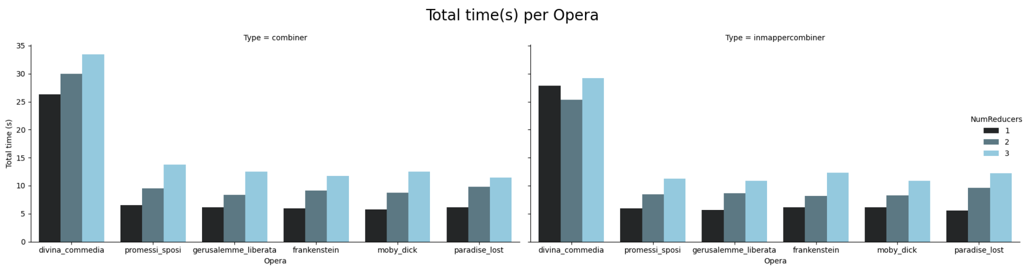


Figure 3: Total time for each opera with different reducers

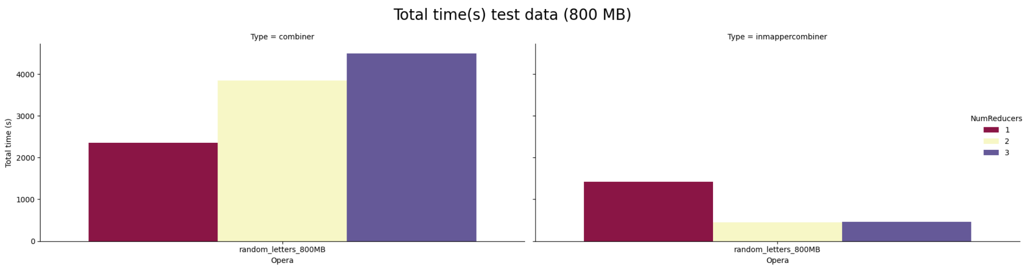


Figure 4: Total time for Test with different reducers

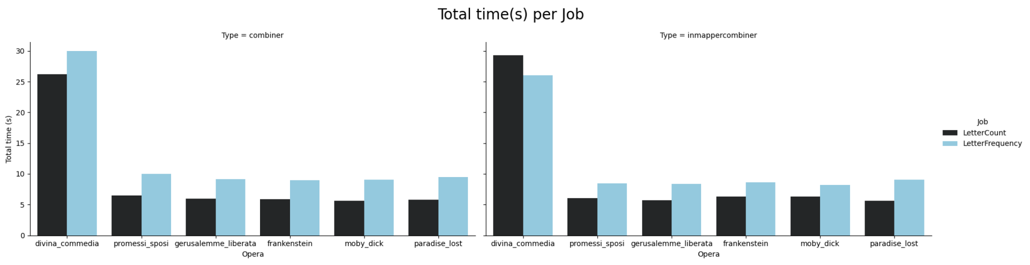


Figure 5: Total time for each job with different operas

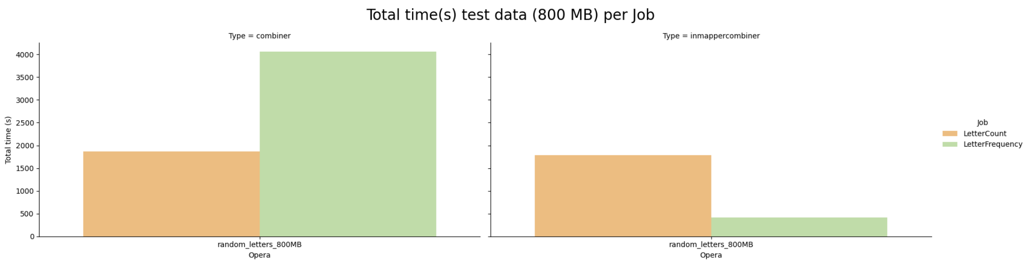


Figure 6: Total time for each job with Test

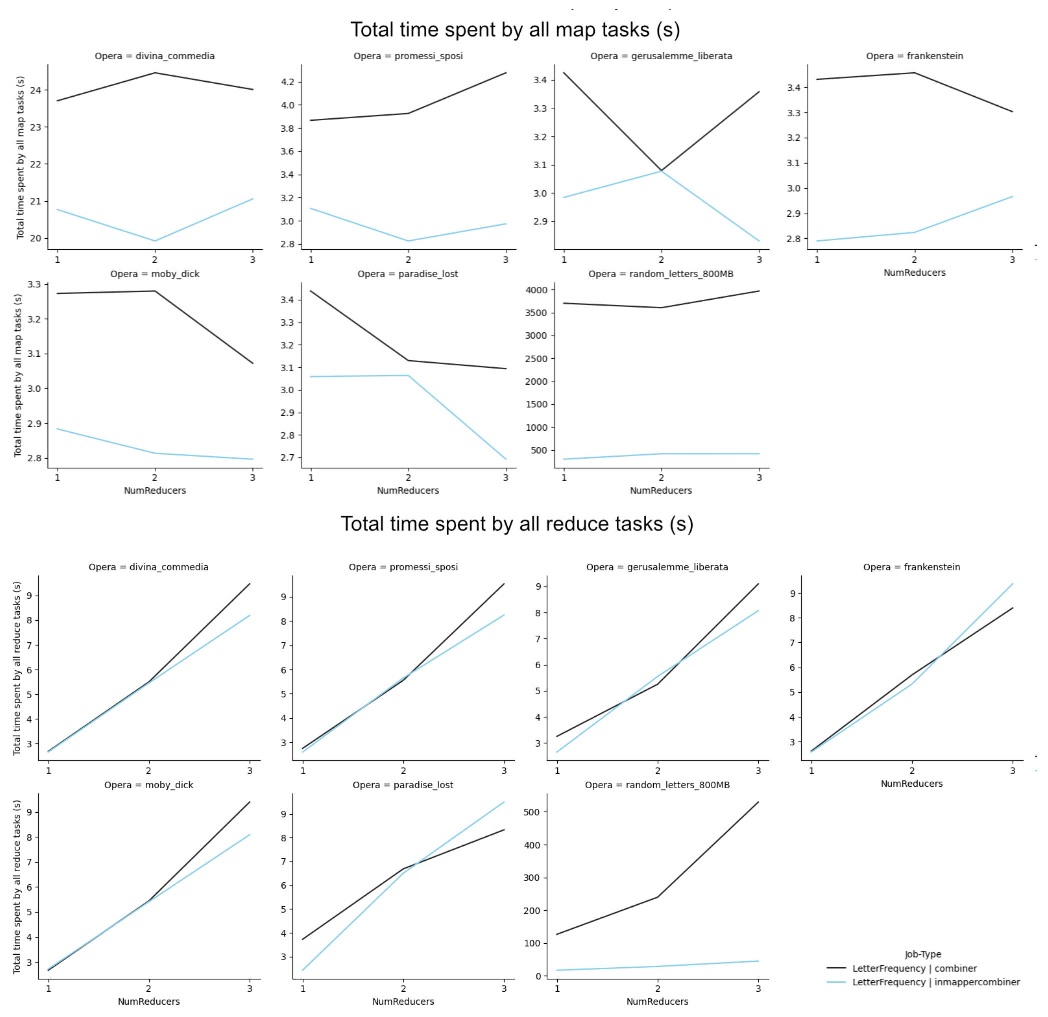


Figure 7: TotalTimeMapReduce

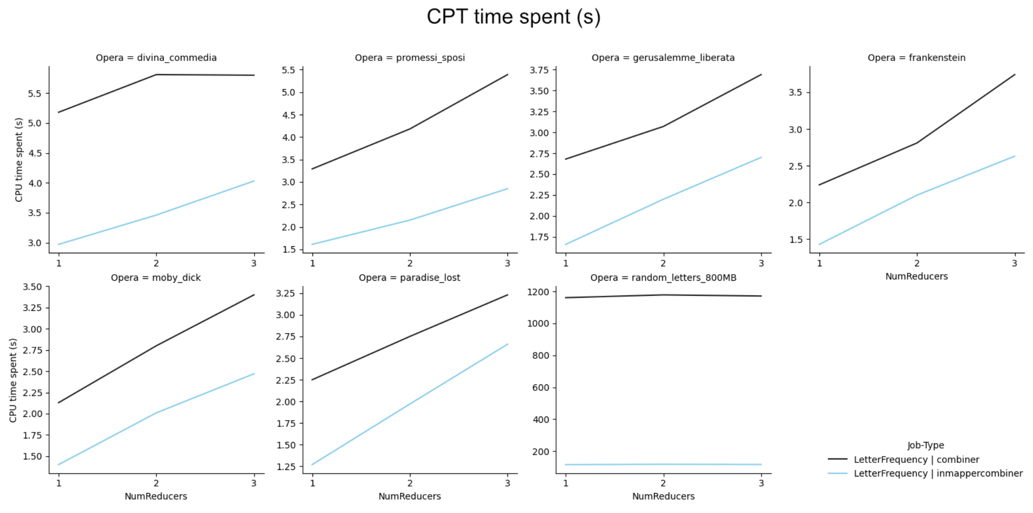


Figure 8: Cpu Time

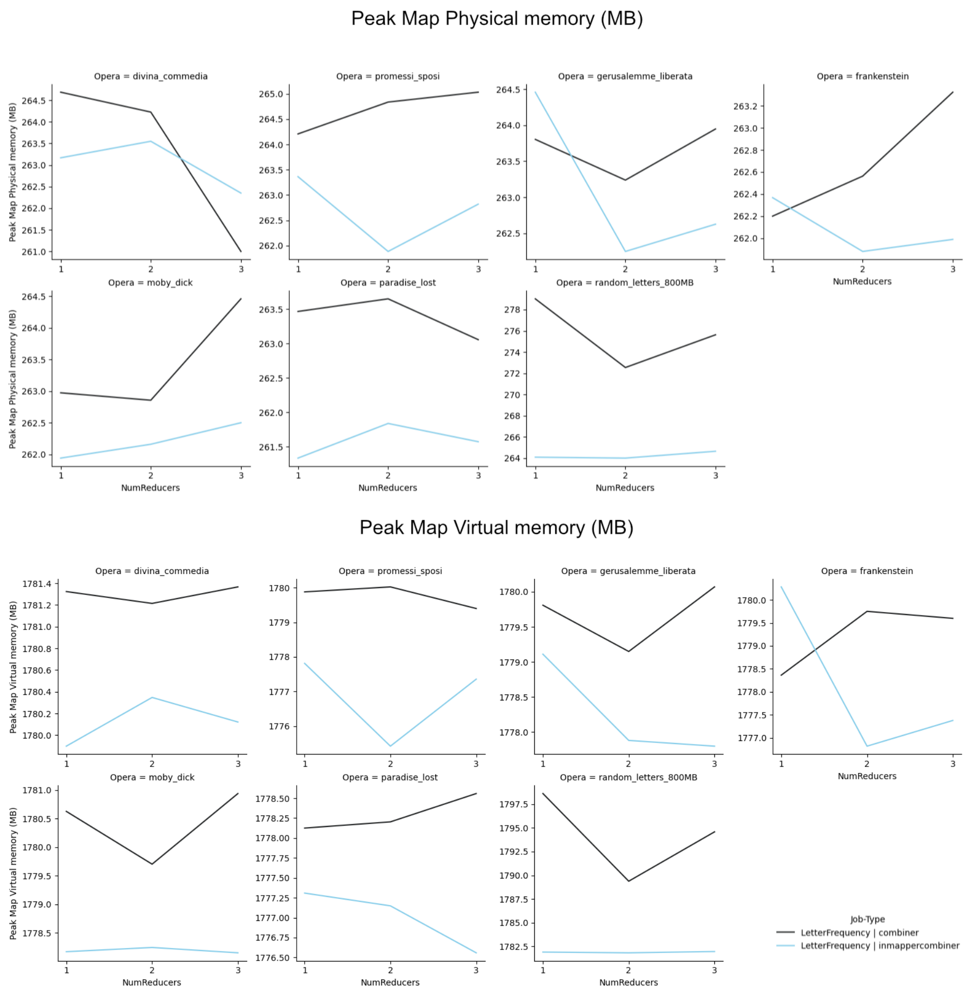


Figure 9: Peak Map Memory

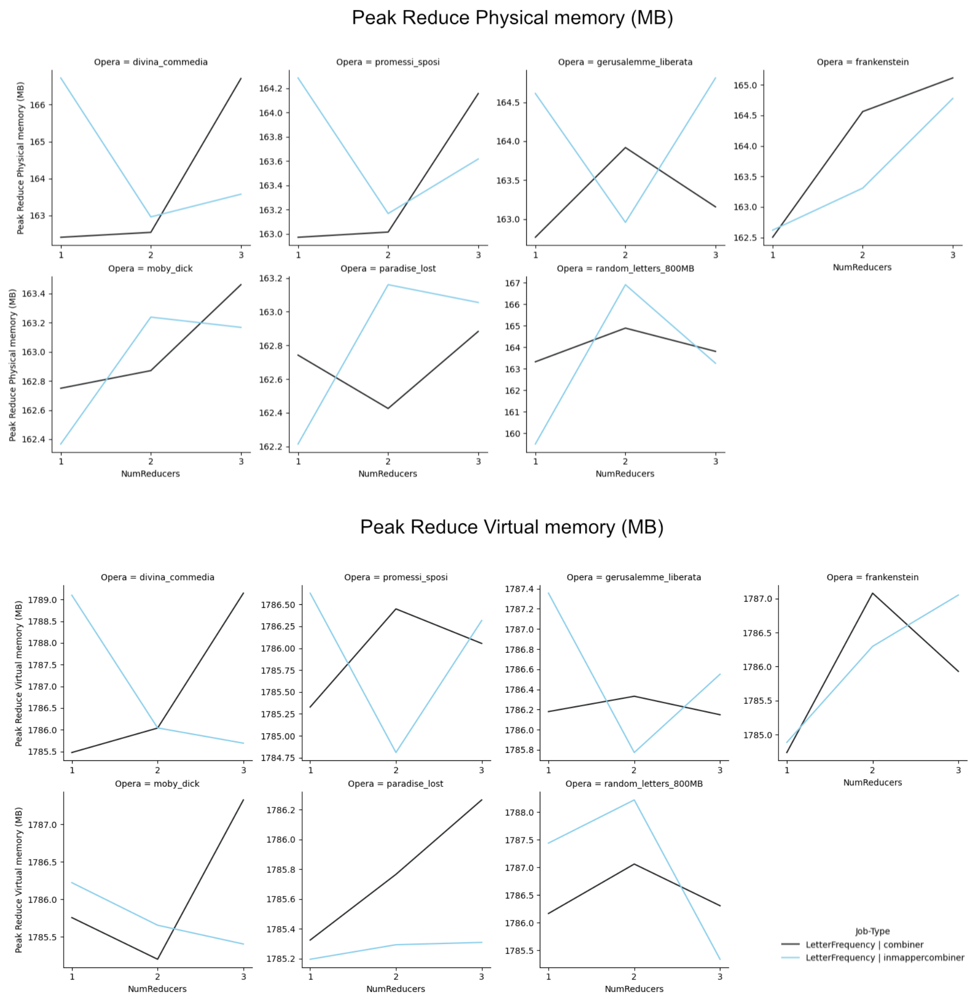


Figure 10: Peak Reduce Memory

Local Python script comparison

As expected, the execution of a local Python script showed excellent performance with small text sizes. In these cases, the absence of the map and reduce phases, typical of the Hadoop framework, proved advantageous. For smaller files, these phases can even be counterproductive, increasing execution time due to the overhead introduced by the distribution and management of tasks among the cluster nodes.
However, for large files, Hadoop demonstrated significantly better performance compared to the local Python script. Hadoop's ability to distribute the workload across multiple cluster nodes and efficiently manage large volumes of data resulted in reduced execution times even for substantial datasets, confirming its effectiveness in handling big data.

Conclusions

The conducted experiments demonstrate the effectiveness of both the Combiner and In-Mapper Combiner approaches in optimizing MapReduce performance. Several key findings emerged from the analysis:

- **Impact of Input Size**: Larger input sizes, such as the 800 MB test file, significantly increased the total processing time. This effect was evident across both the Combiner and In-Mapper Combiner implementations, although the In-Mapper Combiner consistently showed better performance due to reduced data shuffling.
- **Number of Reducers**: Increasing the number of reducers generally decreased the total processing time for the letter frequency job. This is attributed to the parallel processing capabilities of Hadoop, which effectively distributes the workload across multiple reducers.
- **Memory Usage**: The In-Mapper Combiner approach showed a higher peak in memory usage for both map and reduce tasks compared to the Combiner. This is expected as the In-Mapper Combiner holds intermediate results in memory.
- **CPU Time**: The CPU time for the In-Mapper Combiner approach was slightly higher due to the additional overhead of managing intermediate data structures within the mapper. However, the reduced data shuffling led to an overall decrease in total processing time.

In conclusion, the experiments validate the efficiency of using combiners in optimizing MapReduce tasks. The In-Mapper Combiner approach, despite its higher memory usage, consistently outperformed the traditional Combiner method by reducing the volume of intermediate data and minimizing data transfer overhead.