Master's degree in Computer Engineering

Cloud Computing

Letter Count

Alessio Di Ricco

Academic Year 2023/2024

1. Introduction

The purpose of this project was to implement a letter count algorithm exploiting the functionality of the Hadoop Framework to obtain a MapReduce program that receives input files of different sizes and outputs the number of occurrences of each letter.

After the code implementation several tests have been done to evaluate the performance of the algorithm in all its different implementations.

The input dataset combines several .txt file included in the snippets folder provided in the course.
To test the behavior of the program I decided to run different tests using a single file, a partition of the full dataset and the complete dataset.

2. Implementation

2.1 Code execution

The project can be executed using the .jar file, with the following command

hadoop jar file.Jar it.unipi.hadoop.Start snippets output_file reducer_number version

Where:

• reducer_number: specifies the reducers number.

• version: can be 0 (No-Combiner), 1 (In-Mapper combiner (optional request)) and 2 (Combiner).

2.2 Output retrieval

The output can be seen with the following command:

*hadoop fs -cat output_file/part-r**

2.3 Developed classes

To implement the letter count algorithm three different classes have been developed to implement the different operational modes: combiner, in-mapper combiner and no-combiner

- LetterCount
- LetterCountNoCombiner
- LetterCountInMapper

2.3.1 LetterCount

Mapper Class: LetterCountMapper

Analyzes each line of input text and emits key-value pairs for every letter found.

The behavior of the Mapper is the following: firstly it converts the text to lowercase using toLowerCase(). It then Iterates through each character in the line emitting a key-value pair where Key is the letter (e.g., a, b, ...) and value is 1 (represents one occurrence of the letter).

Combiner Class: LetterCountCombiner

Performs a preliminary count of letters within each mapper node to reduce the data sent to the reducers.

The role of the combiner is to aggregate results produced by the mappers before sending them to the reducer in order to decrease the amount of data transferred over the network.

In this case the Combiner receives key-value pairs emitted by the Mapper (e.g., ("a", 1), ("a", 1)). It aggregates the values for each key (e.g., ("a", [1, 1]) → ("a", 2)) and then it emits the aggregated key-value pairs.

Reducer Class: LetterCountReducer

It Aggregates all occurrences of each letter received from the combiners and produces the final output.

Receives as input a key (letter) and a list of values (partial counts). It then sums all values associated with that key and emits a key-value pair where key is the letter and value is the total number of occurrences of the letter in the input file

```
Class LetterCount
  Class LetterCountMapper extends Mapper
    Variables:
      one ← 1
      letter ← empty text

    Method map(key, value, context):
      text ← convert value to lowercase string
      for each character c in text:
        if c is a letter:
          letter ← c
          write (letter, one) to context

    Class LetterCountCombiner extends Reducer
      Variables:
        result ← 0

      Method reduce(key, values, context):
        sum ← 0
        for each val in values:
          sum ← sum + val
        Set result to sum
        write (key, result) to context

    Class LetterCountReducer extends Reducer
      Variables:
        result ← 0

      Method reduce(key, values, context):
        sum ← 0
        for each val in values:
          sum ← sum + val
        Set result to sum
        write (key, result) to context
```

2.3.2 LetterCountNoCombiner

The implementation of this part is exactly the same as the LetterCount with Combiner for the mapper and reducer part. The combiner part has not been implemented in this section.

This implementation has been made in order to verify the importance of combining in the application, in the following chapter several tests have been done to analyze the execution time between those implementations.

2.3.3 LetterCountInMapper (optional part)

In the In-Mapper implementation the reducer code is the same as the Mapper one.

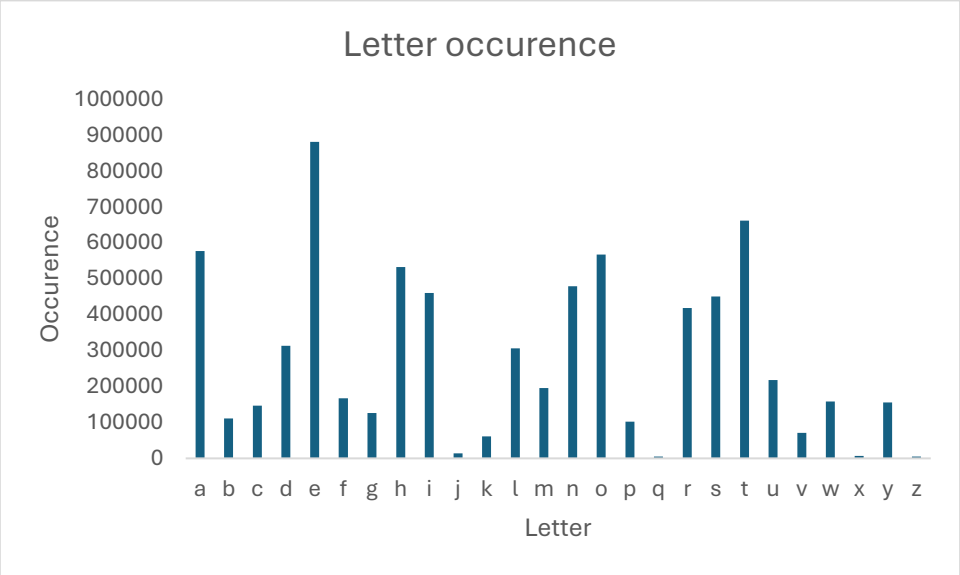
The Mapper part include also the combiner. In particular, the mapper uses an hashmap to store the occurrence of the letters in the input dataset. If the letter is not already in the map a new record is inserted, otherwise if it is already in the corresponding counter is incremented.

```
Class LetterCountInMapper
  Class LetterCountMapper extends Mapper
    Variables:
      lettersMap ← empty map

    Method setup(context):
      Initialize lettersMap as an empty map

    Method map(key, value, context):
      text ← convert value to lowercase string
      For each character c in text:
        If c is a letter:
          letter ← c
          Increment the count of letter by 1 or add letter to lettersMap with value 1

    Method setup(context):
      Write the contents of lettersMap to context
```

3. Analysis

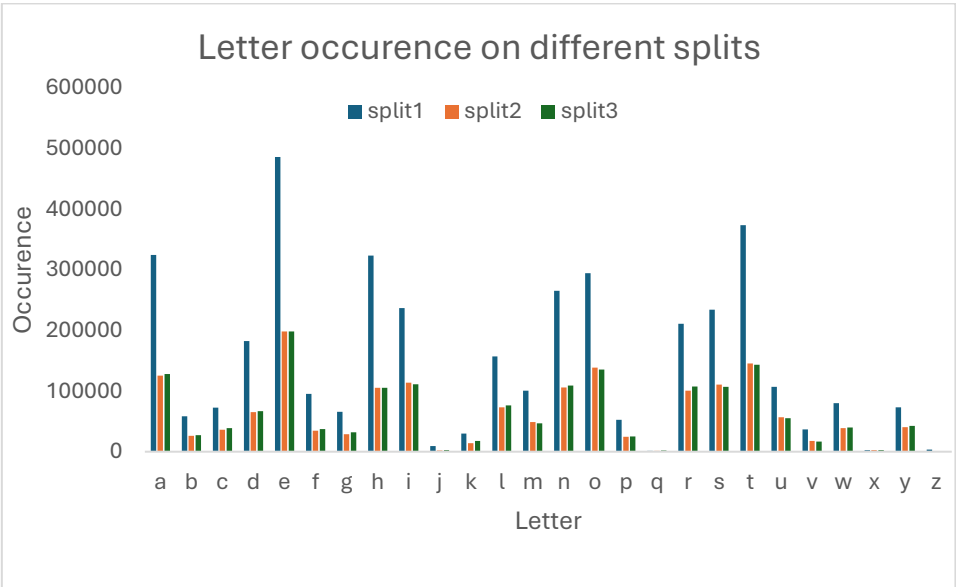
3.1 Letter occurrence report

In the plot below we can see the total letter occurrence of the txt files given in input, the result confirms the expected trend seen in the English vocabulary where the most frequent letter are the e and t followed by the rest of the vowels.



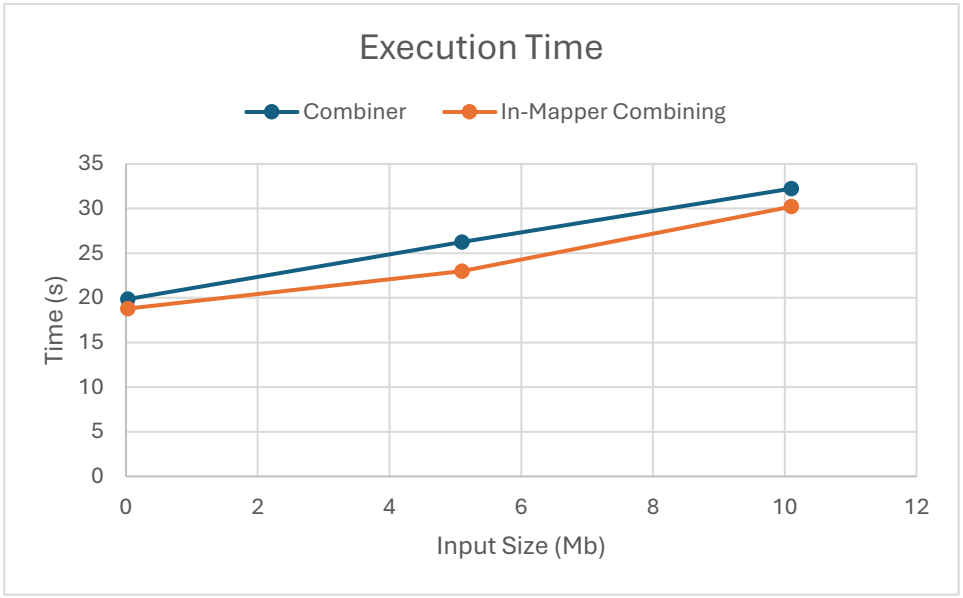
In the following graph I repeat the same analysis three different times splitting the dataset: the first split represents the first third of the txt files, the second one the second third and so on.
We can see a similar trend between split2 and split3 which corresponds in a similar distribution of the letters in all the words.

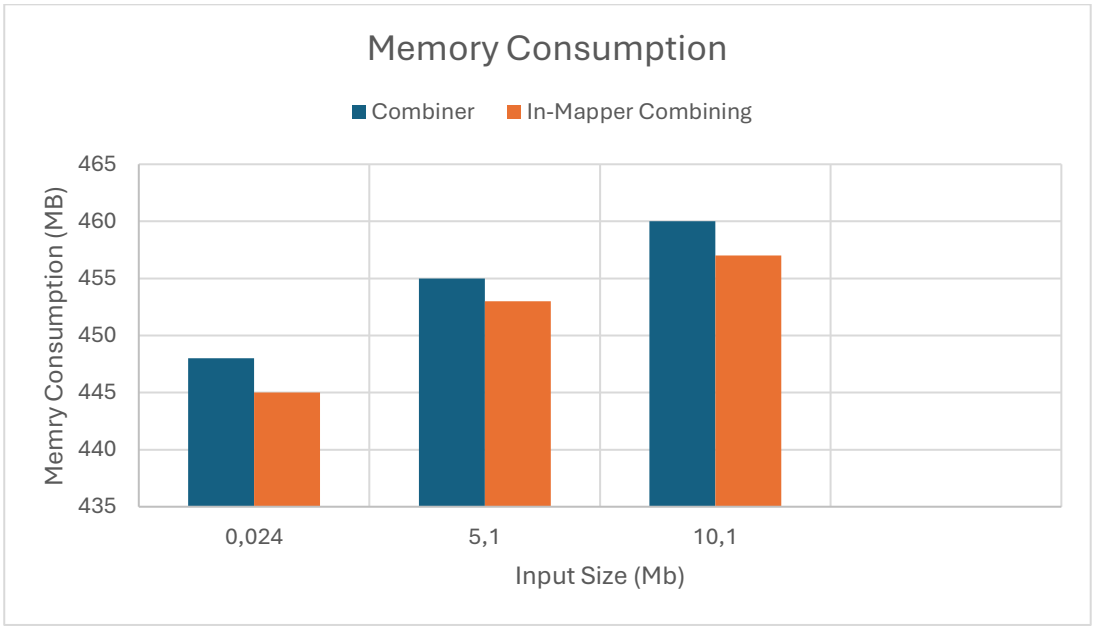
The huge difference between split1 and the other two splits is simply because the first third files are almost three times bigger respect to the others.



3.2 Execution time and memory consumption

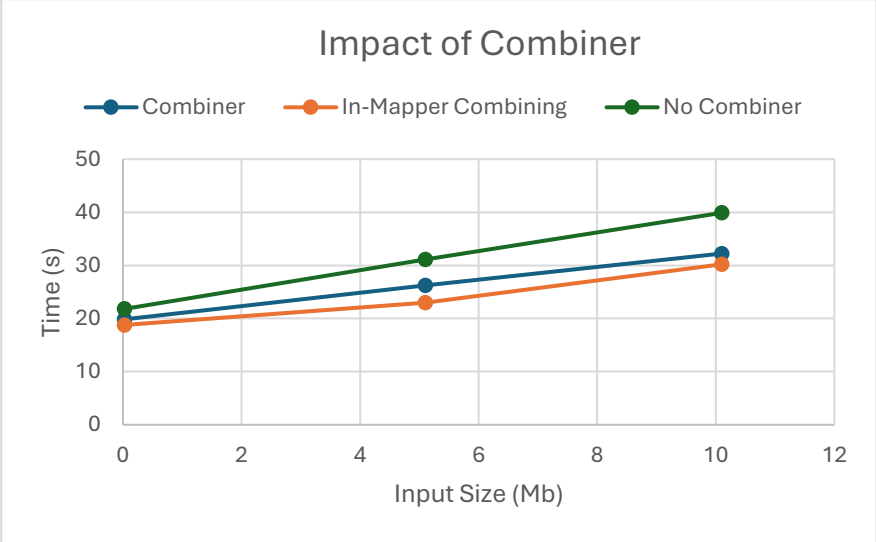
The combiner implementation has a slower execution time respect to the In-Mapper one as we expected. However the advantages are not so visible mainly because the input dataset size is very short.





3.4 Impact of combining

In this section an analysis with the No combiner implementation has been made, we clearly see an increase in the execution time without using the combiner even with a small dataset.



3.5 Increased number of reducers (optional analysis)

In the graph below we can see different executions with different numbers of reducers. The execution time of the application tends to increase instead of having benefits using such numbers.

This is mainly due to the input size given to the application. Because of such limitations, just a single reducer can complete its task in a limited time, the overhead introduced by the other reducers is higher than the benefits that they produce. A higher dataset might get better benefits using multiple reducers