Programming Assignment 2

100 Most Popular Words

Justin Li & Kyle Mondina

Introduction

The objective of this analysis is to design and implement an efficient MapReduce code to determine the top 100 most frequent words in a given dataset, while also analyzing the impact of input size and MapReduce efficiency on performance. The analysis will be conducted on the 16GB dataset, which was used in programming assignment 1. In this programming assignment, we will be utilizing the MapReduce paradigm to solve the problem of finding the 100 most frequent words in each dataset, as well as the 100 most frequent words with six or more letters. By employing the MapReduce framework, we aim to distribute the computational workload across multiple mappers and leverage parallel processing capabilities to enhance the overall efficiency of the solution.

System Specification

Linux Machine
CPU Cores - 8
RAM - 16GB

Approach:

In our assignment, we implemented the MapReduce paradigm to identify the 100 most frequent words in a large text file. As part of the preprocessing step, we needed to remove stopwords from a given list and convert all words to lowercase.

In the Map phase of the MapReduce paradigm, the input data is divided into smaller chunks and processed independently in parallel. In our code, we read the input data line by line from the standard input. Each line is treated as a separate unit of data.

The code begins by performing a map operation. It splits each line into individual words using the split() function and stores them in a list called "words." Then, using a loop, each word is transformed to lowercase using the lower() function. This step ensures that the words are in a consistent case for later comparisons.

After the transformation, the code checks if the lowercase word is present in the stopwords list. This check acts as a filtering step, excluding common stopwords from further processing. Stopwords are typically words that do not carry significant meaning in text analysis. By removing them, we can improve the accuracy and efficiency of subsequent analyses.

For each non-stopword word, the code generates a key-value pair, where the word is the key and the value is set to 1. These key-value pairs are printed using the print() function, following a format commonly used in MapReduce. This format allows the subsequent Reduce phase to process the data.

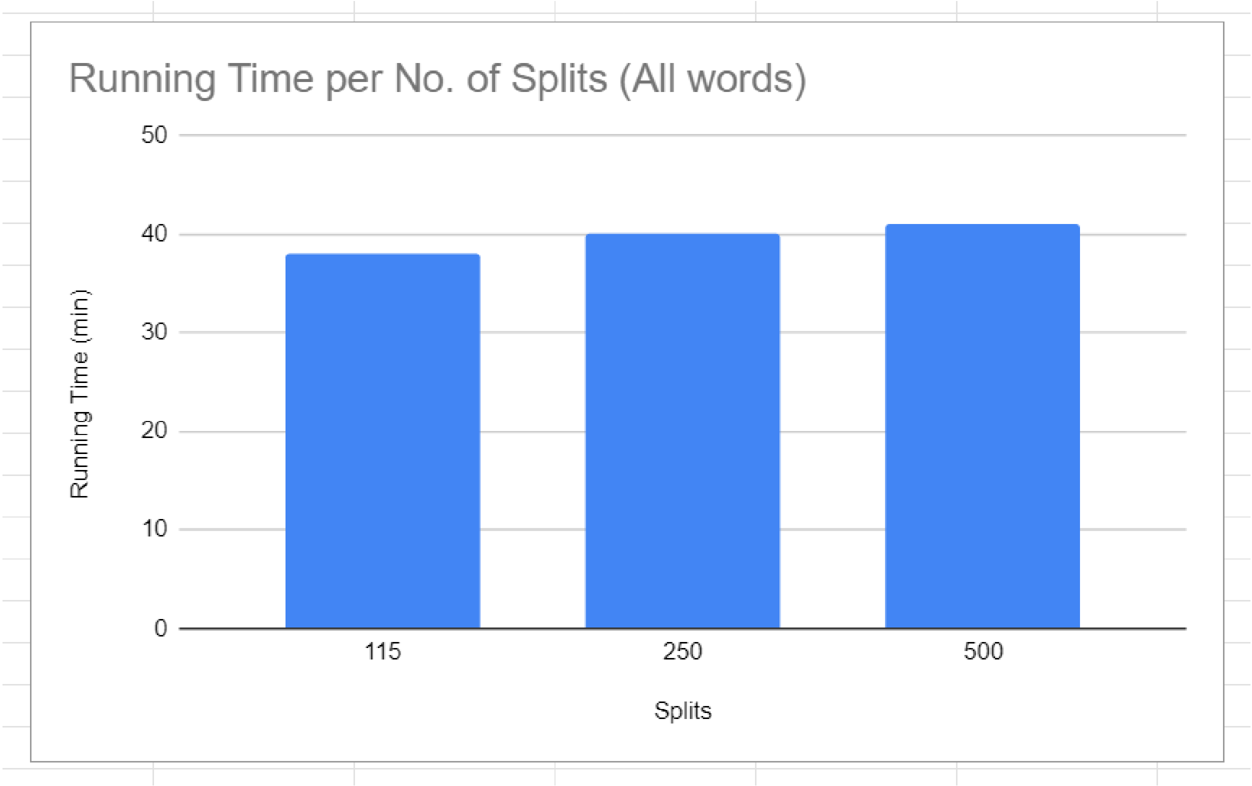
In the reducer.py script, the input is read from the standard input (stdin), with each line expected to be in the format "word\tcount". The reducer aggregates the counts for each word using a dictionary-based approach. By summing the counts of each word encountered, the reducer combines the partial results generated by multiple mapper tasks.

To ensure efficiency, the reducer employs a sorted output mechanism. It iterates over the dictionary items, sorting them in descending order of count and ascending order of the word itself. The code then prints the word and its count, limiting the output to the top 100 most frequent words.

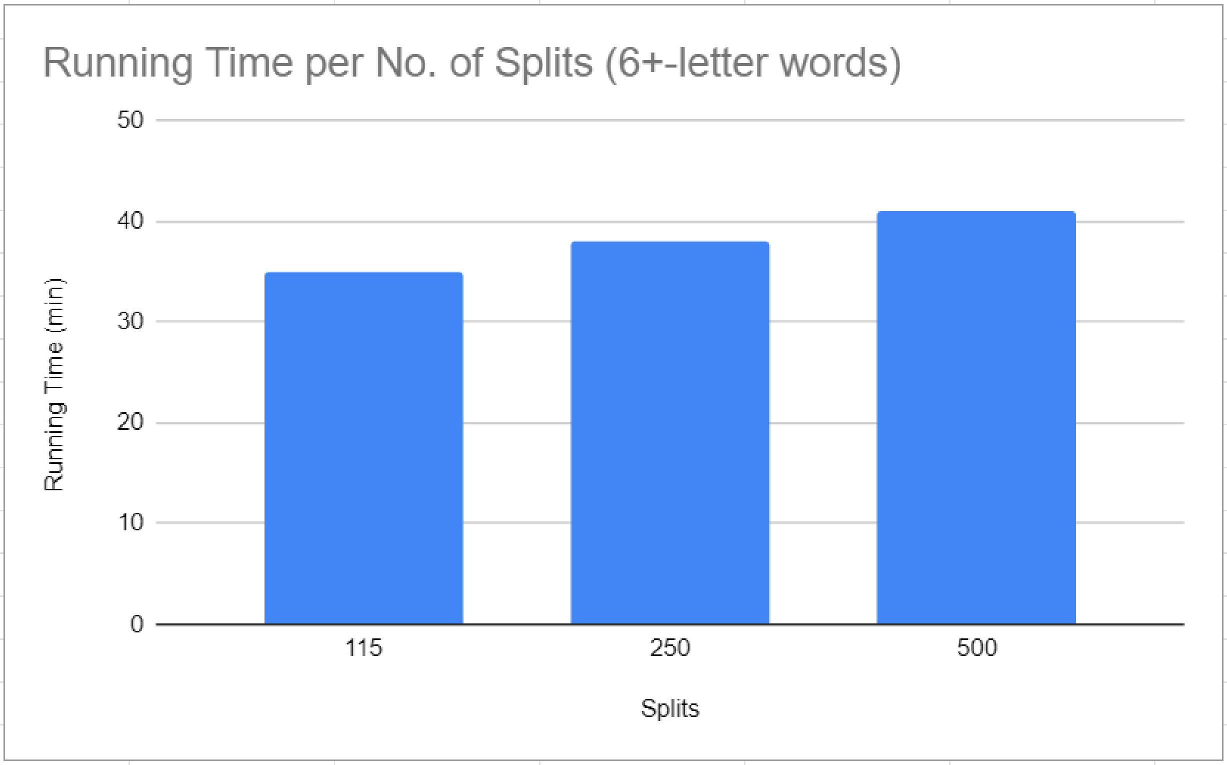
During our experiments, we investigated how the number of splits in mapping the data affected the efficiency of our algorithms. For very large datasets, a higher number of splits can increase parallelization and improve execution speed. However, for smaller datasets, a high number of splits may introduce inefficiencies due to the overhead involved in the splitting process. In our experiment, we measured the running time with different numbers of mappers to analyze their impact on performance.

Results and Analysis

16GB Dataset (All words)



16GB Dataset (6+-letter words)



Analysis

- The execution time of 35 minutes for 115 splits demonstrates a relatively efficient processing time. This suggests that the number of mappers is well-matched to the dataset size, allowing for efficient parallelization and distribution of the workload.
- As the number of splits increased to 250 and 500, the execution times significantly increased to 38 minutes and 41 minutes, respectively. These longer execution times indicate a decrease in efficiency as the number of mappers exceeded the optimal range for the given dataset size.
- When we only considered 6+ letter words
- The increased execution times for higher numbers of splits could be attributed to the overhead involved in the splitting process. With a larger number of splits, the data distribution and coordination between mappers become more complex, resulting in longer processing times.
- Our findings suggest that the scalability of the MapReduce approach may have limitations when applied to smaller datasets. In our experiments, a higher number of splits led to decreased efficiency for the 16GB dataset.
- This highlights the importance of considering the dataset size when determining the optimal number of splits. While a larger number of splits may improve performance for

very large datasets, it can introduce inefficiencies for smaller datasets due to the increased coordination overhead.
- To achieve better scalability, it may be beneficial to dynamically adjust the number of splits based on the dataset size, optimizing the parallel processing capabilities without sacrificing efficiency.

Conclusion

The execution times varied based on the number of splits (mappers) used in processing the 16GB dataset.Optimal efficiency was achieved with 115 splits, resulting in an 35-minute execution time. As the number of splits increased to 250 and 500, the execution times rose to 38 minutes and 41 minutes, respectively, indicating decreased efficiency. Regarding running with 6-letter words, we can also see the execution times increase from 38 minutes on the 115 split run to 40 and 41 minutes on the consecutive runs. The scalability of the MapReduce approach exhibited limitations for smaller datasets, highlighting the need to carefully consider the number of splits based on dataset size. There are opportunities for optimization in the code, particularly in the preprocessing steps and the sorting mechanism during the reducer phase. To improve overall performance, dynamic adjustment of the number of splits, optimization of preprocessing steps, and exploration of alternative sorting approaches can be considered.