Problem statement:

The objective of this report is to determine the 100 most frequent/repeated words in a given dataset of 16GB text using the MapReduce framework. Additionally, we aim to identify the 100 most frequent/repeated words among those that have more than 6 characters. The report will discuss the configuration, performance analysis, and findings of these MapReduce experiments.

Initial configurations:

In our Hadoop 3.3.5 environment, the default value for `mapreduce.input.fileinputformat.split.maxsize` is typically determined by the `fs.local.block.size` property, which represents the block size of the local file system. Since we didn't explicitly configure either property in the `mapred-site.xml` or `hdfs-site.xml` files, the input size split was set to the default value of 128 MB.
Initially, our Hadoop cluster was configured as a single-node setup, indicating either a standalone or pseudo-distributed configuration where Hadoop runs on a single machine. In this case, the available cores in the cluster correspond to the resources (CPU cores) of that machine. We had a total of 8 cores available. By default, our dataset was processed using 115 mappers and 1 reducer in this single-node configuration.

EXP 0. TopK along with Map Reduce

Mapper Phase: The mapper component reads the stop word file, stores the stop words in a set, and processes the input dataset. It ignores stop words and adds the non-stop words to a default dictionary, with the word as the key and its count as the value.
Reducer Phase: The reducer component receives key-value pairs from the mapper, where the key is a word, and the value is its count. It aggregates the counts for each word in a dictionary. Then, it uses a heap data structure to keep track of the top k words based on their counts. The reducer's final output is the top k words and their respective counts.
Result: The experiment was deemed a failure as the program had to be forcibly stopped after running for 37 minutes.
Next Steps: Considering the failed experiment, it was decided to separate the reducer and topK functions, likely to improve the efficiency and performance of the MapReduce process.

EXP 1. Stop words in Mapper |#Mapper 115 | #Reducer 1

Mapper Phase: The mapper reads the input file, filters out stop words and using a default dictionary, and emits key-value pairs representing words and their occurrences.
Reducer Phase: The reducer receives the word counts from the mappers and aggregates them by summing up the occurrences for each word using a default dictionary.
Merging and TopK Phase: The output from the reducer is stored in the Hadoop directory and then merged into a single local file. The TopK component processes the merged file using a min heap of size 100 to extract the top 100 words with the highest counts. The result is obtained from the TopK component.

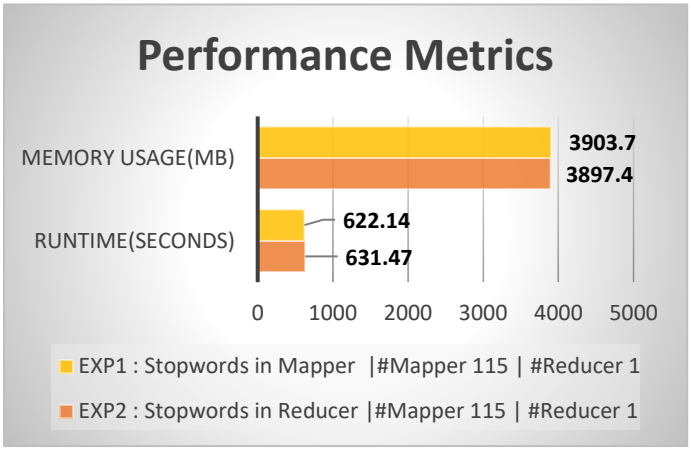
Performance Metrics			
EXP1: Stop words in Mapper |#Mapper 115 | #Reducer 1	Runtime (seconds)	Memory usage (MB)	Average CPU Utilization %
Mappers	521.39	3633.13	3.23
Reducer	8.92	254.86	3.3
Top100	91.83	15.71	3.9

EXP 2. Stop words in Reducer |#Mapper 115 | #Reducer 1

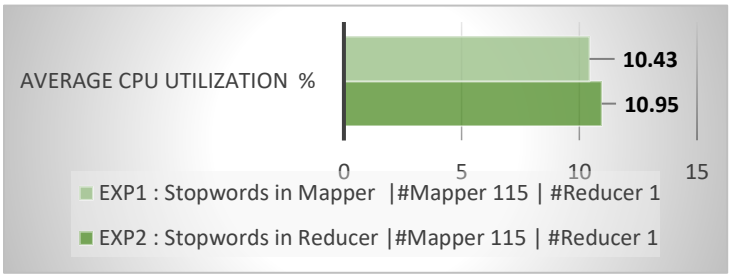
Mapper Phase: The mapper reads the input file and emits key-value pairs representing word occurrences, excluding stop words.
Reducer Phase: The reducer filters out stop words and aggregates the remaining words by summing up their occurrences.
Merging and TopK Phase: Reducer output is merged into a single file, processed by TopK to extract top 100 words.

Performance Metrics			
EXP2: Stop words in Reducer |#Mapper 115 | #Reducer 1	Runtime (seconds)	Memory usage (MB)	Average CPU Utilization %
Mappers	530.48	3627.96	2.65
Reducer	9.16	253.73	4.4
Top100	91.83	15.71	3.9

Performance analysis for EXP 1 and EXP 2:



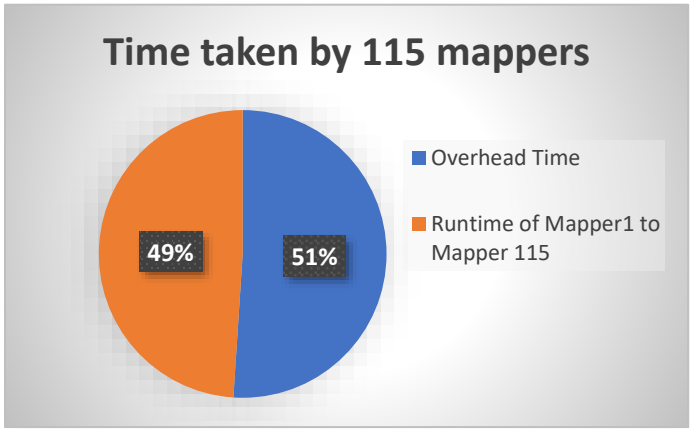
After conducting a thorough analysis of the performance metrics between experiment 1 and 2, we observed that there was **no significant difference** in the overall map-reduce performance based on whether the stop words were read by the reducer or the mapper.



Next Steps: Considering the potential impact of a larger number of stop words in the future, we made a reasoned decision to send preprocessed data to the reducer in experiment 1. This approach ensures that the stop words are filtered out early in the pipeline, reducing the amount of data that needs to be processed by the reducer and potentially improving performance in scenarios with a substantial number of stop words.

Log analysis for EXP 1 and EXP 2:

Furthermore, we conducted a detailed analysis of the logs from the 115 mappers for both experiments. During this analysis, we identified a common factor present in both sets of logs, which involved sequential execution and a time discrepancy. In the following sections, we provide a comprehensive overview of our analysis.



Execution Sequence and Time Discrepancy

Upon examining the timestamps, it was observed that the mappers were executed sequentially, with the first mapper starting at 9:55 and the last mapper finishing at 10:13 with the total time elapsed (1083.24 seconds or 18 minutes). The total runtime of all mappers was calculated to be 530.48 seconds, which equates to approximately 8 minutes and 50 seconds.

Factors Contributing to Elapsed Time

The observed 18-minute time span likely includes the time required for Hadoop to distribute and schedule the tasks across the cluster, as well as other overhead associated with job execution. These factors introduce additional time overhead beyond the actual processing time of the mappers. Consequently, the sum of the individual mapper runtimes (530.48 seconds) does not directly align with the total elapsed time.

The 18-minute elapsed time for the mappers includes the time required for Hadoop to distribute and schedule tasks, as well as other overhead associated with job execution. Therefore, the actual processing time of the mappers (530.48 seconds) does not directly align with the total elapsed time. The difference between the mapper runtimes and the total elapsed time is due to the overhead of distributing data, scheduling tasks, and handling coordination within the Hadoop environment.

EXP 3. Reducer Modification |#Mapper 115 | #Reducer 8

In this experiment, we utilized the mapper and reducer components from experiment 1, which involved filtering out stop words and generating word occurrence key-value pairs. However, we introduced a configuration change by increasing the number of reducers to 8 to evaluate its effect on performance and results.

Analysis:

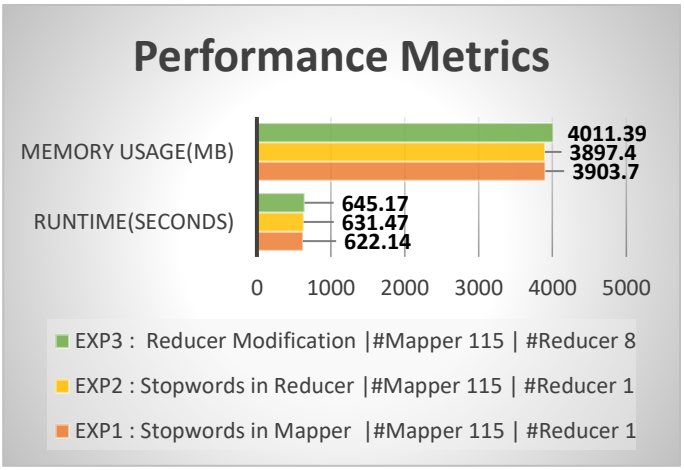
- Increasing the number of reducers from 1 to 8 resulted in a performance decrease.

Performance Metrics			
EXP3: Reducer Modification |#Mapper 115 | #Reducer 8	Runtime (seconds)	Memory usage (MB)	Average CPU Utilization %
Mappers	544.59	3628.72	3.32
Reducer	8.75	366.96	5.31
Top100	91.83	15.71	3.9

- The runtime increased by approximately 23.03 seconds, and memory usage increased by over 107.69MB.
- The performance degradation is attributed to the additional overhead of communication and coordination among the multiple reducers.
- In a multi-reducer setup, partitioning and distributing intermediate data introduce extra network communication and coordination overhead.

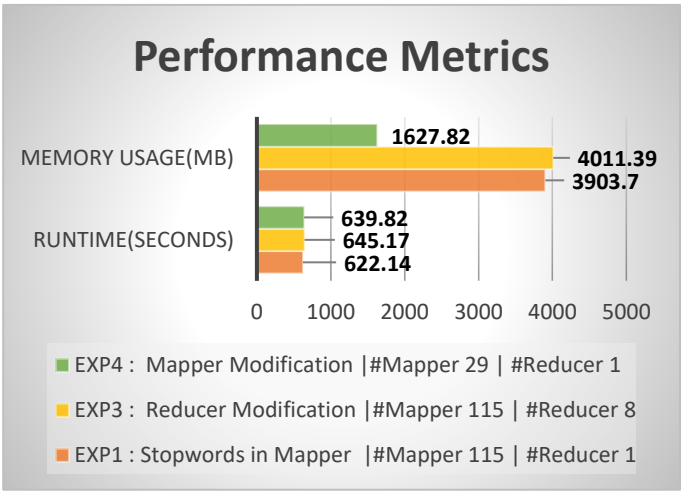
Next Steps: To optimize the MapReduce process, we decided to stick with a single reducer to minimize performance overhead.

However, we aimed to improve efficiency and performance by reducing the number of mapper tasks. This was accomplished by increasing the data split size, leading to better resource utilization and overall performance enhancement.



EXP 4. Mapper Modification |#Mapper 29 | #Reducer 1

In this experiment, we utilized the mapper and reducer components from experiment 1, with stop word filtering and word occurrence aggregation. However, we changed the input split size from 128MB to 512MB, reducing the number of mappers from 115 to 29, to evaluate its effect on performance.



Performance Metrics			
EXP4: Reducer Modification |#Mapper 115 | #Reducer 8	Runtime (seconds)	Memory usage (MB)	Average CPU Utilization %
Mappers	542.49	1357.67	4.52
Reducer	5.5	254.44	4.9
Top100	91.83	15.71	3.9

Analysis:

EXP4 (29 mappers and 1 reducer) compared to EXP1 (115 mappers and 1 reducer)

- The runtime increased by 17.32 seconds and Memory usage decreased significantly by 2275.88 MB signifying the trade-off.
- Decreased memory usage suggests efficient memory utilization.
- Fewer mappers resulted in faster execution.
- Reduced coordination and processing overhead

Next Steps: Based on the efficient memory usage and runtime tradeoff observed in EXP4, we will continue using the configuration with a 512 MB input data split size and 29 mappers. This setup offers improved memory utilization and reduced coordination overhead, leading to enhanced performance in terms of memory usage and runtime.

EXP 5. Adding Combiners |#Mapper 15+ | #Combiners 2184+ | #Reducer?

Mapper Phase: The mapper reads input, converts it to lowercase, and emits non-stop words as key-value pairs.

Combiner Phase: The combiner performs partial aggregation, reducing data transfer by combining counts of the same word.

Reducer Phase: The reducer aggregates word counts, summing up occurrences.

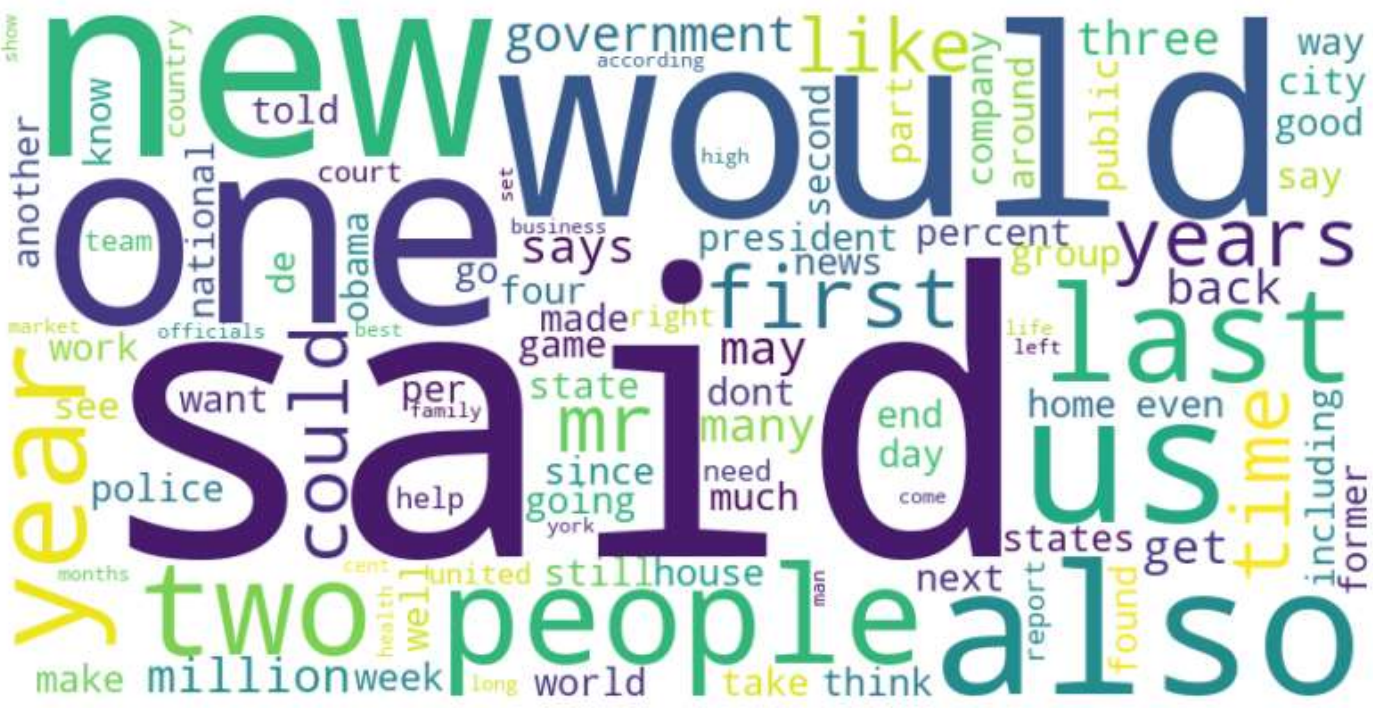
Analysis: EXP5, which involved adding combiners to the configuration, did not yield the expected performance improvement. With a data split size of 512MB and a goal of 29 mappers, the experiment was expected to be completed in under 18 minutes. However, after running for 32 minutes, it exceeded the average runtime for MapReduce jobs.

Results: In a MapReduce framework, the number of combiners is automatically determined based on factors like the availability of combiner functions and the data volume. In this experiment, we observed that approximately 144 combiners were executed sequentially after each mapper, with a 2-minute delay between consecutive mappers. During the 32-minute timeframe, 15 mappers and 2184 combiners were generated.

Next Steps: Given the extended runtime and the observed execution pattern, we decided to terminate the experiment. It became evident that the inclusion of combiners did not enhance processing efficiency as expected.

Results: Top 100 words

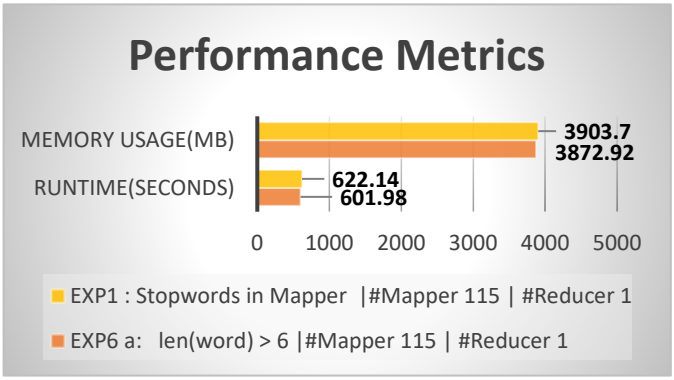
Here is the word cloud representing the top 100 most frequent words in the given dataset. The word cloud visualizes the popularity of each word based on its frequency of occurrence. The size and positioning of each word in the cloud correspond to its relative frequency, with larger and more prominent words indicating higher frequency. This visualization provides an intuitive and visually appealing representation of the dataset, highlighting the most prevalent words within it.



EXP 6. Top100 with length of words more than Six characters

In EXP6, we extended the configurations from EXP1 and EXP4. The mapper filtered out stop words and generated key-value pairs for word occurrences. Additionally, a new filter excluded words with less than 6 characters. By implementing this filter in the mapper phase, we aimed to reduce data processed by the reducer, potentially improving performance. This modification aims to optimize processing efficiency and will be evaluated by comparing performance metrics with EXP1 and EXP4 results.

a. #Mapper 115 | #Reducer 1



In the first part of the experiment, we replicated the configurations of EXP1, utilizing 115 mappers and 1 reducer. This configuration allowed us to compare the performance metrics between EXP 1 & EXP 6

Performance Metrics			
EXP6a: length(word) > 6 |#Mapper 115 | #Reducer 1	Runtime (seconds)	Memory usage (MB)	Average CPU Utilization %
Mappers	446.52	3602.76	3.59
Reducer	6.61	254.55	3.9
Top100	148.85	15.61	4.3

Analysis:

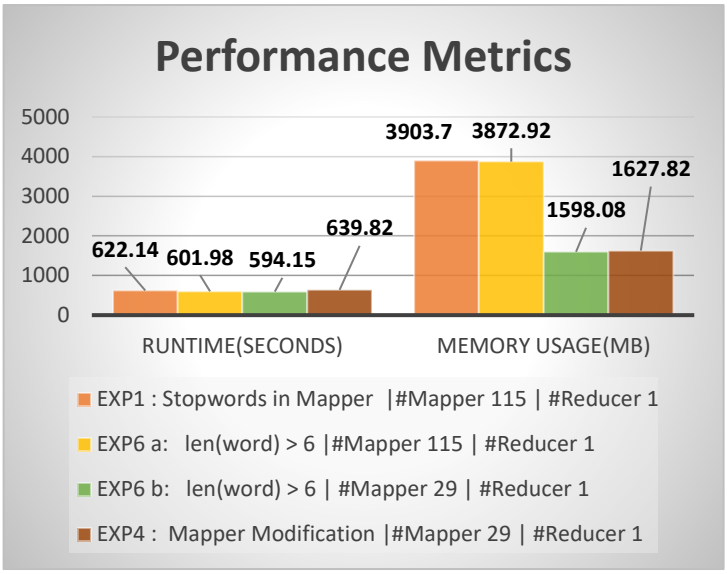
- EXP6 showed a slight improvement in runtime and memory usage compared to EXP1.
- Runtime for EXP6 was 601.98 seconds, while EXP1 took 622.14 seconds. Memory usage in EXP6 decreased to 3872.92 MB from 3903.7 MB in EXP1. Top 100 function took almost twice the time to process than without 6 char filter.
- Excluding shorter words in the mapper phase resulted in more efficient memory utilization. The additional filter in EXP6 reduced data processed by the reducer, improving performance.

Overall, EXP6 demonstrated the effectiveness of the additional filter, leading to better runtime and memory usage.

b. #Mapper 29 | #Reducer 1

Performance Metrics			
EXP6: length(word) > 6 |#Mapper 29 | #Reducer 1	Runtime (seconds)	Memory usage (MB)	Average CPU Utilization %
Mappers	440.99	1328.13	1.87
Reducer	4.31	254.34	4.7
Top100	148.85	15.61	4.3

In the second part of the experiment, we replicated the configurations of Experiment 4, using 29 mappers and 1 reducer. This configuration aimed to assess the impact of reducing the number of mappers on the processing efficiency and performance.



Analysis: EXP6b showed further improvement in runtime and memory usage compared to EXP1, EXP6a, and EXP4.

- The runtime of EXP6b (594.15 seconds) was shorter than EXP6a (601.98 seconds) and EXP4 (639.82 seconds).
- EXP6b exhibited lower memory usage (1598.08 MB) compared to EXP6a (3872.92 MB) and EXP4 (1627.82 MB).
- The reduction in the number of mappers from 115 to 29 in EXP6b contributed to improved performance by reducing coordination overhead and optimizing resource allocation.
- The additional filtering in EXP6b further enhanced performance compared to EXP4, which lacked the filter condition.
- EXP6b demonstrated faster runtime and slightly lower memory usage compared to both EXP1 and EXP4.

Overall, EXP6b's reduced mapper count and inclusion of the filter condition in the mapper phase significantly improved runtime and memory usage, making it an optimal configuration for MapReduce optimization.

Results: Top 100 words having more than 6 characters

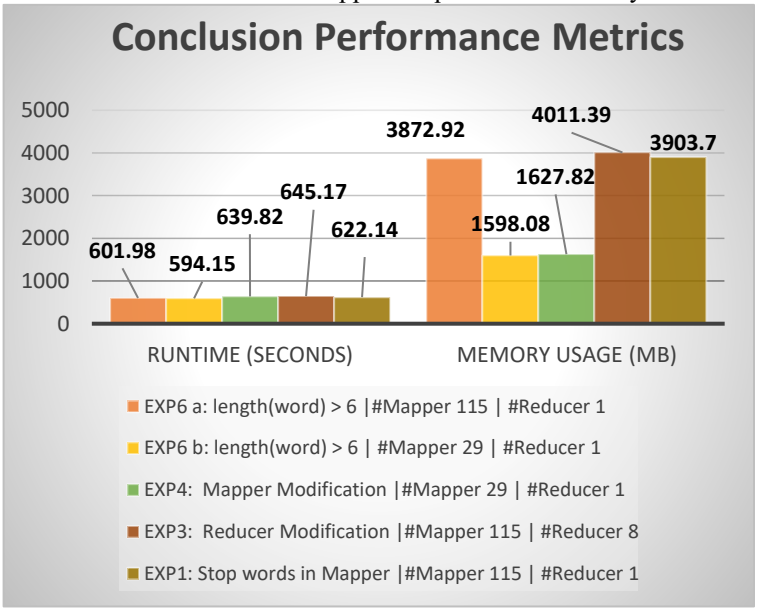
Here is the word cloud representing the top 100 most frequent words in the given dataset considering only words having more than 6 characters.

Conclusion

Based on the analysis of the experiments conducted, several conclusions can be drawn regarding the performance and efficiency of different configurations in the MapReduce process.

Conclusion Performance Metrics	Runtime (seconds)	Memory usage (MB)	Avg CPU Utilization %
EXP6 a: length(word) > 6 |#Mapper 115 | #Reducer 1	601.98	3872.92	11.79
EXP6 b: length(word) > 6 | #Mapper 29 | #Reducer 1	594.15	1598.08	10.86
EXP4: Mapper Modification |#Mapper 29 | #Reducer 1	639.82	1627.82	13.32
EXP3: Reducer Modification |#Mapper 115 | #Reducer 8	645.17	4011.39	12.53
EXP1: Stop words in Mapper |#Mapper 115 | #Reducer 1	622.14	3903.7	10.43

1. **EXP0:** It was observed that calculating the top 100 results separately from the MapReduce process is the most efficient approach. This indicates that certain operations may be better suited for separate processing rather than incorporating them into the MapReduce framework.
2. **EXP1 & EXP2:** Configurations with 115 mappers and 1 reducer, where stop words are filtered out early in the pipeline, showed potential for performance improvement. By reducing the amount of data that needs to be processed by the reducer, these configurations can enhance efficiency, particularly in scenarios with a significant number of stop words.
3. **EXP3:** Increasing the number of reducers to 8 resulted in a performance decrease. This decline can be attributed to the additional overhead of communication and coordination among multiple reducers. Therefore, it is important to consider the trade-off between parallel processing and the associated overhead when determining the number of reducers.
4. **EXP4:** With 29 mappers and 1 reducer, EXP4 demonstrated improved memory utilization and slightly faster execution compared to EXP1. The reduced coordination and processing overhead associated with fewer mappers contributed to these improvements. The decreased memory usage indicates that the reduced number of mappers required less memory for data processing tasks.
5. **EXP5:** The addition of combiners to the configuration did not yield the expected performance improvement. This suggests that combiners may not be effective in enhancing processing efficiency in this scenario.
6. **Part 2 | EXP 6:** The extended experiments, EXP6a and EXP6b, introduced an additional filter to exclude words with a length less than 6 characters. EXP6a, with 115 mappers and 1 reducer, exhibited a reduction in processing time compared to EXP1. The decrease in runtime indicates that filtering out shorter words early in the pipeline improved processing efficiency. EXP6b, with the same filter condition but only 29 mappers, showed a shorter runtime and lower memory usage compared to EXP6a. This suggests that the additional filtering step in EXP6b further contributed to performance improvement.



Key Findings

1. The algorithm's performance is more affected by disk I/O rather than computational factors.
2. Hadoop automatically spawns a read thread for each mapper task, as observed in the console logs.
3. Increasing the block size of the HDFS file system from 128MB to 51MB resulted in a decrease in overall runtime.
4. Fewer mappers were required in the 512MB configuration, leading to fewer read/write queries to the disk and improved efficiency.
5. Reducing the number of mappers from 115 to 29 significantly reduced the overhead associated with mapper management.
6. Changing the block size also contributed to a reduction in total memory usage by efficiently utilizing the default Hadoop threads for each mapper.

In conclusion, the experiments revealed that filtering out irrelevant data in the mapper phase, utilizing 115 mappers, and employing a single reducer without combiners resulted in the most efficient performance. This configuration showcased improved processing efficiency, reduced memory usage, and faster execution time. The findings highlight the importance of considering the specific characteristics of the data and the computational requirements when configuring the MapReduce process.