ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING – SITE

Department of Artificial Intelligence

Course: Distributed Computing for AI ITSC-2112

Assignment A4

MapReduce: WordCount

Submitted By: Mintesnot Fikir ID GSR/1669/15

Section: Regular

Submitted to: Dr. Beakal Gizachew

Submission date: January, 2024

1. Introduction

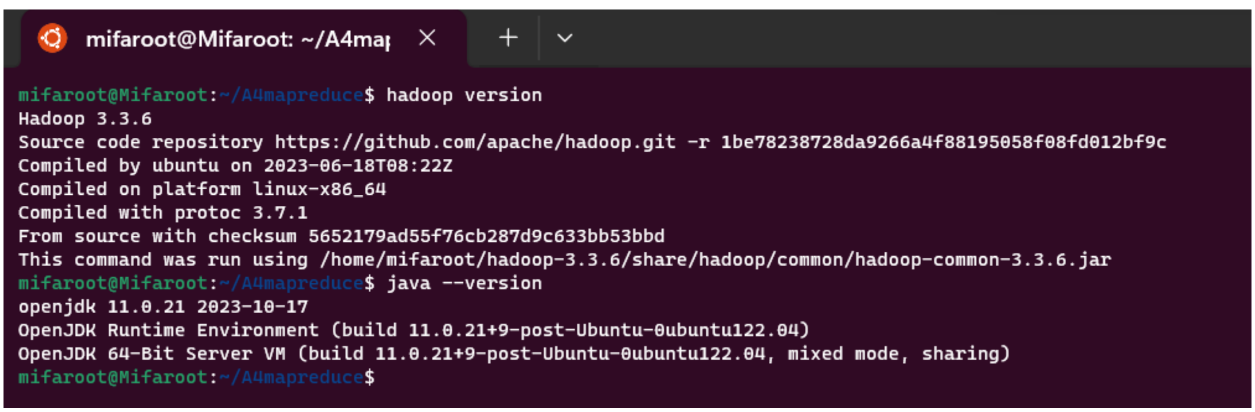
In this assignment, I used the MapReduce programming model within the Hadoop ecosystem to tackle text analysis challenges, particularly focusing on word frequency counts across various text files downloaded from Project Gutenberg. First, I set up a MapReduce job to count words in these texts. Then, I improved my analysis by ignoring common words(stopwords), like "the" or "and," to find more meaningful words. This experience helped me learn more about how Hadoop can handle big data, showing its power to work through large amounts of text and provide useful insights.

This assignment not only allowed me to delve deep into the practical applications and capabilities of Hadoop in managing and processing large-scale datasets but given me the chance to explore how MapReduce marketecture implemented in distributed systems also illuminated the textual characteristics and themes prevalent within the corpus of literature available in Project Gutenberg. Through this analytical exercise, I demonstrated the scalability, efficiency, and adaptability of the MapReduce model in handling complex data-intensive tasks, offering a glimpse into the potential of big data technologies in extracting valuable insights from vast repositories of textual information.

2. Project Setup

The setup for this project:

- **Operating System**: Ubuntu 20.04.3 desktop version provided a stable and versatile foundation for the development and execution of MapReduce tasks.
- **Hadoop Installation**: Version 3.3.6 of Hadoop was installed
- **Java Environment**: OpenJDK 11.0.21 was selected for its compatibility with Hadoop, facilitating the compilation and execution of Java-based MapReduce programs.



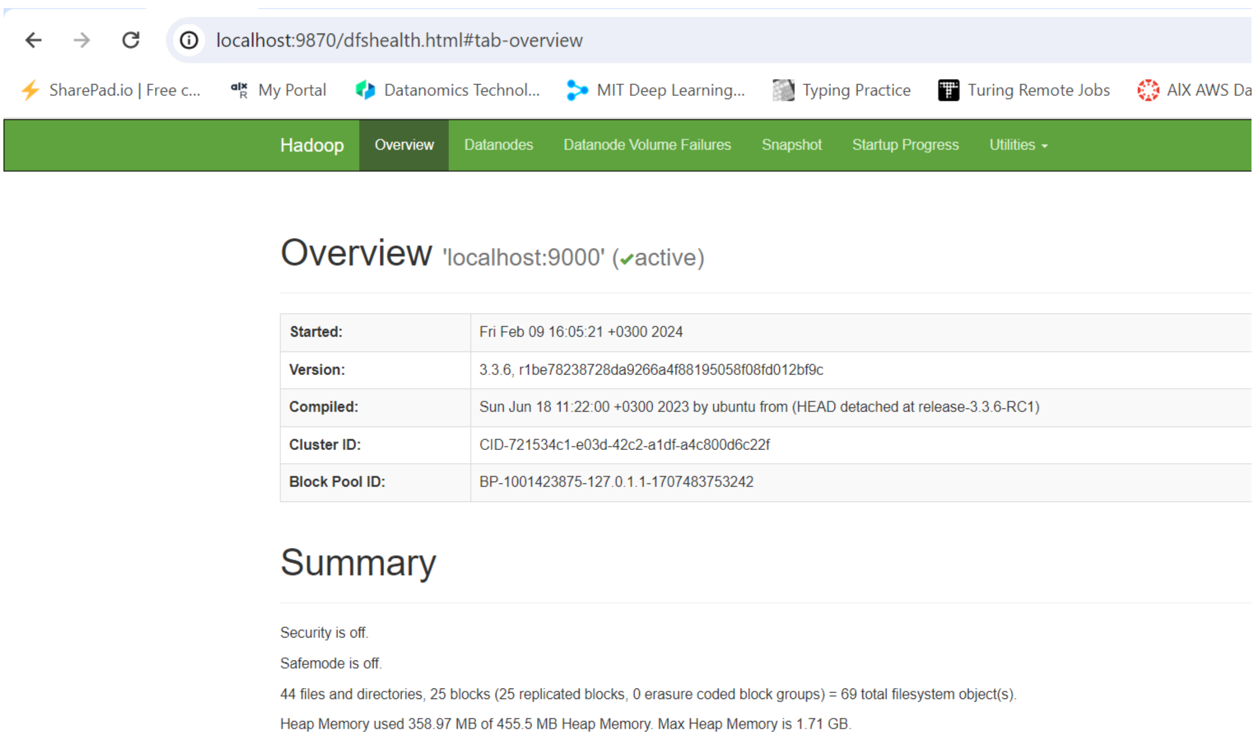
3. Setting up hadoop environment

After installing Hadoop, setting up the environment involves configuring Hadoop's core components for optimal performance and functionality. This typically includes editing the $hadoop - env.sh$, core-site.xml, hdfs-site.xml, $mapred - site.xml$, and $yarn - site.xml$ configuration files to specify the file system, resource management, and job submission parameters. Additionally, formatting the Hadoop Distributed File System (HDFS) and starting Hadoop's $NameNode$ and $DataNode$ services are crucial steps. It's important to verify the setup by running test jobs to ensure that the Hadoop cluster is functioning as expected.

```
mifaroot@Mifaroot:~$ hadoop namenode -format
WARNING: Use of this script to execute namenode is deprecated.
WARNING: Attempting to execute replacement "hdfs namenode" instead.

2024-02-09 16:02:30,218 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = Mifaroot/127.0.1.1
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 3.3.6
STARTUP_MSG:   classpath = /home/mifaroot/hadoop-3.3.6//etc/hadoop:/home/mifaroot/hadoop-3.3.6//share/hadoop/common/lib/kerb
```

```
mifaroot@Mifaroot:~$ start-all.sh
WARNING: Attempting to start all Apache Hadoop daemons as mifaroot in 10 seconds.
WARNING: This is not a recommended production deployment configuration.
WARNING: Use CTRL-C to abort.
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [Mifaroot]
Starting resourcemanager
Starting nodemanagers
```

Then check the web UI interface using http://localhost:9870/



4. Dataset (Books)

For this assignment, I selected ten diverse books from Project Gutenberg, ensuring each text file exceeded the minimum size requirement of 100kB to provide a rich dataset for analysis. The books vary in genre, length, and historical period, offering a broad spectrum for word count examination. Below is a table detailing the names, file sizes, and character counts of the downloaded books:

Name of the Book	File Name(.txt format)	Size (KB)	Number of Characters
A Visit to the Roman Catacombs	A_visit_to_the_Roman_catacombs_pg71927	257	256,593
Elements of Metaphysics	elements_of_meta_physics_pg71885	1,182	1,181,041
History For Ready Reference	history_For_Ready_Reference_pg71897	5,159	5,150,599
Little Women	Little_Women_pg37106	648	648,914
Middlemarch	Middlemarch_pg5197	1,823	1,799,413
The Australian Aboriginal	The_Australian_aboriginal_pg71940	840	835,784
The Green Hat	The_Green_hat_pg71913	529	517,486
The Iliad	The_Iliad_pg6130	223	220,426
The Romance of Lust	The_Romance_of_Lust_pg30254	234	233,569
The Tempest	The_Tempest_pg23042	102	101,171

Project Gutenberg is a library of over 70,000 free eBooks: https://www.gutenberg.org/

5. Basic Word Count

I developed a basic Word Count application using Hadoop MapReduce. The purpose was to process a dataset consisting of text files, count the occurrences of each word, and output the results in a key-value pair format, with words as keys and their counts as frequencies. This task involved implementing a Mapper class to tokenize text into words while filtering out punctuation and making the process case-insensitive. A Reducer class aggregated these counts across all text inputs. The application was tested on a selection of books, demonstrating Hadoop's capability to handle large-scale data processing efficiently. The initial results provided a straightforward count of all words appearing in the texts, serving as a foundational step for more complex analysis that followed, such as filtering out common stop words to refine the output.

In this phase of the project, the focus was on preparing and transferring the dataset for the word count task in Hadoop. Operating within a Windows Subsystem for Linux (WSL) environment, I needed to move the dataset, comprised of 10 books sourced from Project Gutenberg, into Hadoop's Distributed File System (HDFS). To achieve this, I initially created a dedicated input directory in HDFS using the command ***hadoop fs -mkdir -p /input***. Subsequently, I used the ***hadoop fs -copyFromLocal ~/A4mapreduce/data/books /input*** command to transfer the entire folder containing the books from the local filesystem to the specified HDFS location. This command ensured that all 10 books, each exceeding the 100kb size requirement, were successfully uploaded to HDFS, ready for processing. This step was crucial for setting the groundwork for the MapReduce word count operation, demonstrating an essential Hadoop file management operation—transferring data from the local file system to HDFS.

```
mifaroot@Mifaroot: ~/A4maŗ

mifaroot@Mifaroot:~$ hadoop fs -mkdir -p /input
mifaroot@Mifaroot:~$ ls
A4mapreduce  card  demo.csv  dist  hadoop-3.3.6  hadoop-3.3.6.tar.gz  stream  xai
mifaroot@Mifaroot:~$ cd A4mapreduce
mifaroot@Mifaroot:~/A4mapreduce$ ls
CountWithoutStop.java  SortWord.java  WordCount.java  data  local_output  stopwords.txt
```

```
mifaroot@Mifaroot: ~/A4maŗ

mifaroot@Mifaroot:~/A4mapreduce$ hadoop fs -copyFromLocal ~/A4mapreduce/data/books /input
mifaroot@Mifaroot:~/A4mapreduce$ hadoop fs -ls /input/myfile.txt
ls: `/input/myfile.txt': No such file or directory
mifaroot@Mifaroot:~/A4mapreduce$ hadoop fs -ls /input/
Found 1 items
drwxr-xr-x   - mifaroot supergroup          0 2024-02-09 23:26 /input/books
mifaroot@Mifaroot:~/A4mapreduce$ hadoop fs -ls /input/books
Found 10 items
-rw-r--r--   1 mifaroot supergroup     262343 2024-02-09 23:26 /input/books/A_visit_to_the_Roman_catacombs_pg71927.txt
-rw-r--r--   1 mifaroot supergroup    5281878 2024-02-09 23:26 /input/books/History_For_Ready_Reference_pg71897.txt
-rw-r--r--   1 mifaroot supergroup     662750 2024-02-09 23:26 /input/books/Little_Women_pg37106.txt
-rw-r--r--   1 mifaroot supergroup    1865775 2024-02-09 23:26 /input/books/Middlemarch_pg145.txt
-rw-r--r--   1 mifaroot supergroup     859228 2024-02-09 23:26 /input/books/The_Australian_aboriginal_pg71940.txt
-rw-r--r--   1 mifaroot supergroup     541195 2024-02-09 23:26 /input/books/The_Green_hat_pg71913.txt
-rw-r--r--   1 mifaroot supergroup     227414 2024-02-09 23:26 /input/books/The_Iliad_pg6130.txt
-rw-r--r--   1 mifaroot supergroup     239450 2024-02-09 23:26 /input/books/The_Romance_of_Lust_pg30254.txt
-rw-r--r--   1 mifaroot supergroup     103995 2024-02-09 23:26 /input/books/The_Tempest_pg23042.txt
-rw-r--r--   1 mifaroot supergroup    1210107 2024-02-09 23:26 /input/books/elements_of_meta_physics_pg71885.txt
mifaroot@Mifaroot:~/A4mapreduce$
```

Hadoop   Overview   Datanodes   Datanode Volume Failures   Snapshot   Startup Progress   Utilities

Browse Directory

/input/books Go!

Show 25 entries Search:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	mifaroot	supergroup	256.19 KB	Feb 09 23:26	1	128 MB	A_visit_to_the_Roman_catacombs_pg71927.txt
-rw-r--r--	mifaroot	supergroup	5.04 MB	Feb 09 23:26	1	128 MB	History_For_Ready_Reference_pg71897.txt
-rw-r--r--	mifaroot	supergroup	647.22 KB	Feb 09 23:26	1	128 MB	Little_Women_pg37106.txt
-rw-r--r--	mifaroot	supergroup	1.78 MB	Feb 09 23:26	1	128 MB	Middlemarch_pg145.txt
-rw-r--r--	mifaroot	supergroup	839.09 KB	Feb 09 23:26	1	128 MB	The_Australian_aboriginal_pg71940.txt
-rw-r--r--	mifaroot	supergroup	528.51 KB	Feb 09 23:26	1	128 MB	The_Green_hat_pg71913.txt
-rw-r--r--	mifaroot	supergroup	222.08 KB	Feb 09 23:26	1	128 MB	The_Iliad_pg6130.txt
-rw-r--r--	mifaroot	supergroup	233.84 KB	Feb 09 23:26	1	128 MB	The_Romance_of_Lust_pg30254.txt
-rw-r--r--	mifaroot	supergroup	101.56 KB	Feb 09 23:26	1	128 MB	The_Tempest_pg23042.txt
-rw-r--r--	mifaroot	supergroup	1.15 MB	Feb 09 23:26	1	128 MB	elements_of_meta_physics_pg71885.txt

5.1. Basic Word Count implementation

Now let's move on to identifying the most frequently occurring words in a book, I decided to use a MapReduce tutorial example to accurately count words, eliminating punctuation impacts and ensuring case insensitivity, thus equating "Dog" and "dog." This adjustment required importing necessary Hadoop libraries, pivotal for executing MapReduce tasks. These libraries provide the infrastructure to dissect and process large text datasets, enabling me to implement custom logic for text analysis. By focusing on these modifications, I aim to enhance data processing accuracy, ensuring that the word count reflects true text usage by treating variations of a word as the same entity, regardless of case or surrounding punctuation.

WordCount.java | SortWord.java | CountWithoutStop.java

Ubuntu-22.04 > home > mifaroot > A4mapreduce > WordCount.java

```java
import java.io.IOException;
import java.util.StringTokenizer;

// Hadoop configuration and data types for map and reduce functions
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

```

The next section is, ***WordMapper*** class, which extends the class provided by Mapper, ***WordMapper*** class is at the heart of the word processing operation. It's designed to tokenize the input text, breaking it down into individual words. The critical modification here involves using a regular expression ***([^a-zA-Z])*** to filter out any non-alphabetic characters, effectively removing punctuation from the tokens. This ensures that the analysis focuses solely on words, disregarding symbols or numbers that could skew the word count. Moreover, converting tokens to lowercase addresses the case sensitivity issue, treating words like "***Dog***" and "***dog***" as identical. This preprocessing step is crucial for achieving accurate and meaningful word count results, particularly when analyzing texts with varied capitalization and punctuation usage.

```java
// Mapper class that tokenizes each line of input text into words
public static class WordMapper extends Mapper<Object, Text, Text, IntWritable>{

  // A constant to represent a count of one for each word encountered
  private final static IntWritable one = new IntWritable(1);
  // A Text object to store each word processed
  private Text processedWord = new Text();

  // The map method processes text input, one line at a time
  public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
    // Tokenizes the input text
    StringTokenizer words = new StringTokenizer(value.toString());
    while (words.hasMoreTokens()) {
      // Cleans each token by removing non-alphabetic characters and converting to lowercase
      String cleanWord = words.nextToken().replaceAll("[^a-zA-Z]", "").toLowerCase();
      // Skips any empty tokens
      if (!cleanWord.isEmpty()) {
        // Sets the cleaned word into the Text object
        processedWord.set(cleanWord);
        // Writes the word as key and one as value to the context
        context.write(processedWord, one);
      }
    }
  }
}
```

Following the mapping phase, the ***CountReducer*** class plays a crucial role in aggregating the counts of each word. For each unique word identified by the mapper, this reducer sums up all occurrences found across the dataset. This aggregation phase is where the counts of words, now normalized for case and punctuation, are compiled into their final tallies. The process of iterating through each word's counts and summing them is straightforward yet powerful, enabling the synthesis of data from potentially vast and disparate text sources into a coherent count per word.

In the ***CountReducer*** class's reduce method, the process iterates over each unique word's occurrence counts provided by the mapper. It aggregates these counts to determine the total frequency of each word. The sum of occurrences is stored in an ***IntWritable*** variable, representing the consolidated count for each word. This total count is then output alongside the word (as the key) using ***context.write(key, totalcount)***, effectively finalizing the word count operation by listing each word with its corresponding total count. This method ensures a comprehensive tally of all words processed by the job.

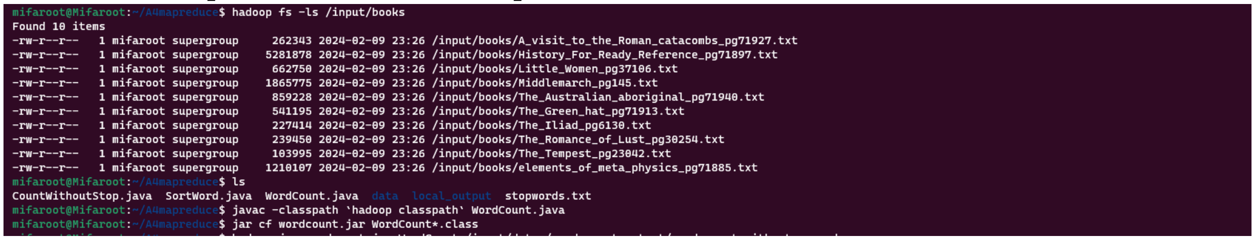
```java
// Reducer class that sums up the counts for each word
public static class CountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
  // A writable integer to store the sum of counts for a word
  private IntWritable totalCount = new IntWritable();

  // The reduce method processes each key (word) and its list of counts
  public void reduce(Text key, Iterable<IntWritable> counts, Context context) throws IOException, InterruptedException {
    int sum = 0;
    // Iterates over the counts and sums them
    for (IntWritable count : counts) {
      sum += count.get();
    }
    // Sets the sum as the total count for the word
    totalCount.set(sum);
    // Writes the word and its total count to the context
    context.write(key, totalCount);
  }
}
```

The main method serves as the orchestrator for setting up and executing the MapReduce job. It configures the job, specifying the job name, mapper, reducer, and combiner classes, and defines the input and output formats. This setup is critical for ensuring that the Hadoop framework correctly understands the job's requirements, from the data it will process to the format of the output results. By establishing these parameters, the main method facilitates the smooth execution of the word count task, leveraging Hadoop's distributed computing capabilities to efficiently process large volumes of text data.

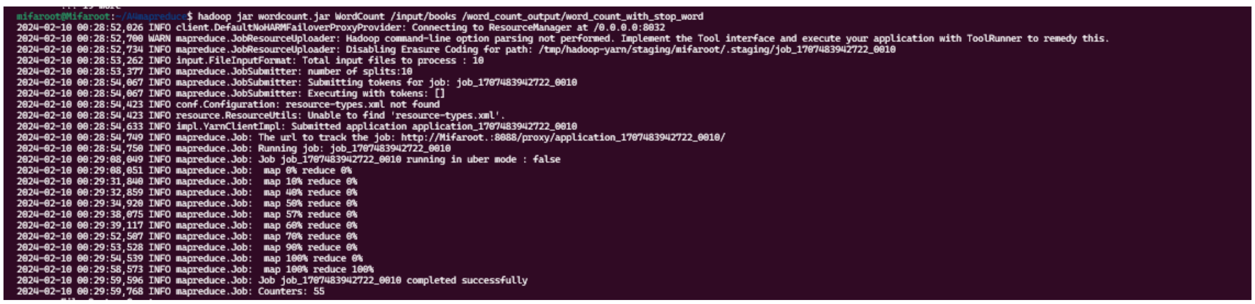
```java
  // The main method to set up and start the MapReduce job
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    // Defines a new job named "Enhanced Word Count"
    Job job = Job.getInstance(conf, "Enhanced Word Count");
    // Sets the jar by the class
    job.setJarByClass(WordCount.class);
    // Sets mapper, combiner, and reducer classes
    job.setMapperClass(WordMapper.class);
    job.setCombinerClass(CountReducer.class);
    job.setReducerClass(CountReducer.class);
    // Sets the types of output key and value
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    // Sets the paths to input and output
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    // Exits with the job completion status
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

5.2. Executing the MapReduce Program and Analyzing Output

Now we can move into the execution and analysis of a MapReduce word count program. Initially, the dataset containing books as a texts file was transferred to HDFS under ***/input/books.*** Compilation and packaging of the **WordCount.java** program into a JAR file (**wordcount.jar**) followed. Compiled the WordCount.java program without errors, generating class files necessary for packaging, packaged the compiled classes into wordcount.jar, making the MapReduce program executable on Hadoop. Successful execution processed 10 books as a text file, generating word count outputs stored in ***/word_count_output/word_count_with_stop_word***.

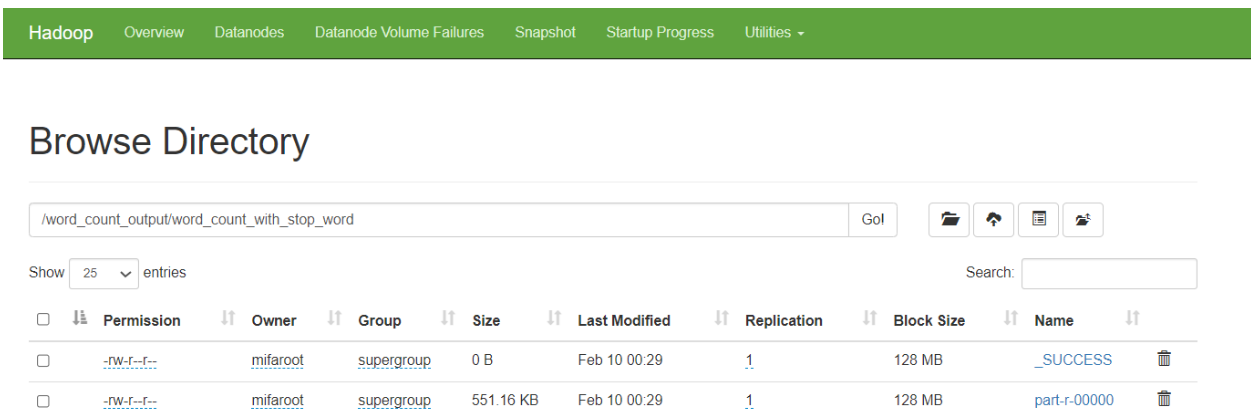


Execution of the *WordCount* Program: successfully executed the Word Count program. This process analyzed the content of 10 books, demonstrating the MapReduce framework's capability to distribute and process large datasets efficiently.



Here is the result:

```
        File System Counters
                FILE: Number of bytes read=20711573
                FILE: Number of bytes written=44462038
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=11255423
                HDFS: Number of bytes written=564383
                HDFS: Number of read operations=35
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
                HDFS: Number of bytes read erasure-coded=0
        Job Counters
                Killed map tasks=2
                Launched map tasks=11
                Launched reduce tasks=1
                Data-local map tasks=11
                Total time spent by all maps in occupied slots (ms)=223265
                Total time spent by all reduces in occupied slots (ms)=21772
                Total time spent by all map tasks (ms)=223265
                Total time spent by all reduce tasks (ms)=21772
                Total vcore-milliseconds taken by all map tasks=223265
                Total vcore-milliseconds taken by all reduce tasks=21772
                Total megabyte-milliseconds taken by all map tasks=228623360
                Total megabyte-milliseconds taken by all reduce tasks=22294528
        Map-Reduce Framework
                Map input records=233454
                Map output records=1768774
                Map output bytes=17174019
                Map output materialized bytes=20711627
                Input split bytes=1288
                Combine input records=0
                Combine output records=0
                Reduce input groups=48963
                Reduce shuffle bytes=20711627
                Reduce input records=1768774
                Reduce output records=48963
                Spilled Records=3537548
                Shuffled Maps =10
                Failed Shuffles=0
                Merged Map outputs=10
                GC time elapsed (ms)=3571
                CPU time spent (ms)=91980
                Physical memory (bytes) snapshot=3747192832
                Virtual memory (bytes) snapshot=30215315456
                Total committed heap usage (bytes)=2413821952
                Peak Map Physical memory (bytes)=382685184
                Peak Map Virtual memory (bytes)=2764210176
                Peak Reduce Physical memory (bytes)=248832000
                Peak Reduce Virtual memory (bytes)=2739150848
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=11254135
        File Output Format Counters
                Bytes Written=564383
mifaroot@Mifaroot:~/A4mapreduce$
```

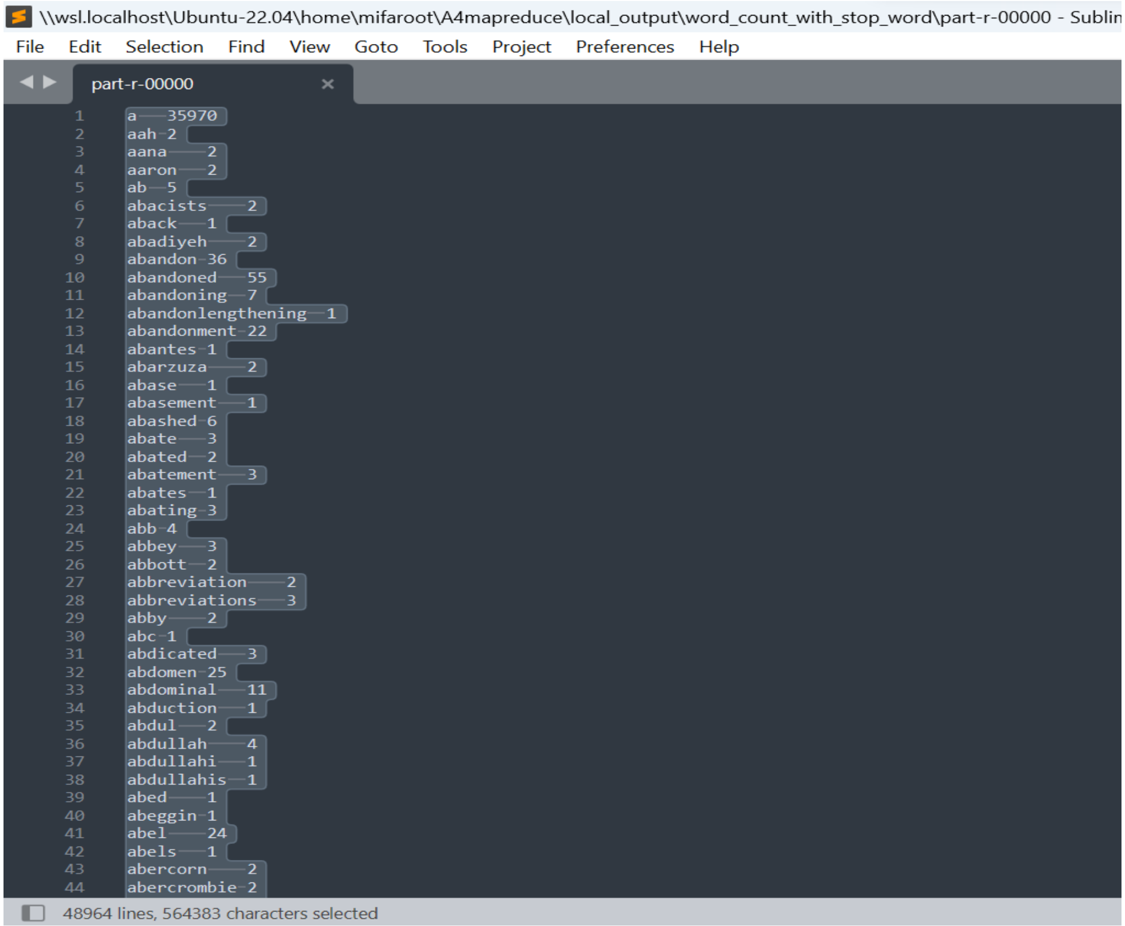



```
mifaroot@Mifaroot:~/A4mapreduce$ hadoop fs -ls /word_count_output
Found 2 items
-rw-r--r--   1 mifaroot supergroup          0 2024-02-09 17:04 /word_count_output/_SUCCESS
-rw-r--r--   1 mifaroot supergroup     433133 2024-02-09 17:04 /word_count_output/part-r-00000
mifaroot@Mifaroot:~/A4mapreduce$
```

Now let's copy the output files form the Hadoop to local

```
mifaroot@Mifaroot:~/A4mapreduce$ hadoop fs -get /word_count_output/word_count_with_stop_word ~/A4mapreduce/local_output
mifaroot@Mifaroot:~/A4mapreduce$ hadoop fs -ls /word_count_output/word_count_with_stop_word
Found 2 items
-rw-r--r--   1 mifaroot supergroup          0 2024-02-10 00:29 /word_count_output/word_count_with_stop_word/_SUCCESS
-rw-r--r--   1 mifaroot supergroup     564383 2024-02-10 00:29 /word_count_output/word_count_with_stop_word/part-r-00000
mifaroot@Mifaroot:~/A4mapreduce$ ls
 CountWithoutStop.java   SortWord.java  'WordCount$IntSumReducer.class'  'WordCount$TokenizerMapper.class'   WordCount.class   WordCount.java   data   local_output   stopwords.txt   wordcount.jar
mifaroot@Mifaroot:~/A4mapreduce$ cd local_output
mifaroot@Mifaroot:~/A4mapreduce/local_output$ ls
word_count_with_stop_word
mifaroot@Mifaroot:~/A4mapreduce/local_output$ cd word_count_with_stop_word
mifaroot@Mifaroot:~/A4mapreduce/local_output/word_count_with_stop_word$ ls
_SUCCESS  part-r-00000
mifaroot@Mifaroot:~/A4mapreduce/local_output/word_count_with_stop_word$
```

Wordcount results



As we see, from the above result, those ten books collectively encompass a total of 48,964 words. The data currently is organized alphabetically. To align with the assignment's requirements, which inquire about words with the highest frequency, a Java program named SortWord.java was devised. This program reorders the output, prioritizing words by their occurrence rates, thereby facilitating the identification of the most frequent terms within the dataset. This approach is essential for addressing specific queries.

```java
public class SortWord {
    public static void main(String[] args)
        if (args.length != 1) {
            System.exit(1);
        }
        String filePath = args[0];
        try (BufferedReader fileReader = new BufferedReader(new FileReader(filePath))) {
            String currentLine;
            Map<String, Integer> wordFrequencies = new HashMap<>();
            while ((currentLine = fileReader.readLine()) != null) {
                String[] wordAndCount = currentLine.split("\t");
                if (wordAndCount.length == 2) {
                    String word = wordAndCount[0];
                    int count = Integer.parseInt(wordAndCount[1].trim());
                    wordFrequencies.put(word, count);
                }
            }
            List<Map.Entry<String, Integer>> entriesSortedByCount = new ArrayList<>(wordFrequencies.entrySet());
            entriesSortedByCount.sort(Comparator.comparing(Map.Entry::getValue, Comparator.reverseOrder()));
            List<Map.Entry<String, Integer>> top25Words = entriesSortedByCount.subList(0, Math.min(25, entriesSortedByCount.size()));
            System.out.println("The top 25 words with highest counts:");
            for (Map.Entry<String, Integer> entry : top25Words) {
                System.out.println(entry.getKey() + " - " + entry.getValue());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

To analyze the frequency of words in a dataset, the SortWord program was executed, designed to identify and rank the top 25 words based on their occurrence. Upon running this program with the specified dataset, it revealed a predominance of common words, often referred to as "stop words," such as "the," "of," "and," "to," among others. These words, while frequent, often add minimal contextual value for analytical purposes, leading to an inflated representation in the word count.

```
mifaroot@Mifaroot:~/A4mapreduce$ ls
 CountWithoutStop.java   SortWord.java  'WordCount$IntSumReducer.class'  'WordCount$TokenizerMapper.class'   WordCount.class   WordCount.java   data   local_output   stopwords.txt   wordcount.jar
mifaroot@Mifaroot:~/A4mapreduce$ javac SortWord.java
mifaroot@Mifaroot:~/A4mapreduce$ java SortWord local_output\word_count_with_stop_word/part-r-00000
java.io.FileNotFoundException: local_outputword_count_with_stop_word/part-r-00000 (No such file or directory)
        at java.base/java.io.FileInputStream.open0(Native Method)
        at java.base/java.io.FileInputStream.open(FileInputStream.java:219)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:157)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:112)
        at java.base/java.io.FileReader.<init>(FileReader.java:60)
        at SortWord.main(SortWord.java:20)
mifaroot@Mifaroot:~/A4mapreduce$ java SortWord local_output/word_count_with_stop_word/part-r-00000
The top 25 words with highest counts
the - 128046
of - 79598
and - 52442
to - 49129
in - 36484
a - 35970
that - 20586
is - 16232
it - 16069
as - 15305
was - 14943
for - 13702
with - 13671
be - 13627
i - 13118
by - 12471
which - 11385
on - 10351
he - 9929
not - 9851
this - 9561
at - 9555
his - 9111
her - 9091
had - 8508
mifaroot@Mifaroot:~/A4mapreduce$
```

6. Extending the basic word count program (Exclude Stop words)

To ensure the Word Count program for more meaningful analysis, a significant modification was implemented on the previous code which is the exclusion of stop words. This improvement involved downloading a list of 5,489 stop words and word forms, saved in ***/home/mifaroot/A4mapreduce/stopwords.txt***. The modified code introduces a mechanism to load these stop words into the program, effectively ignoring them during the word count process. This approach ensures that common but less informative words are filtered out, allowing for a focused analysis on more relevant text content.

```java
// A Hadoop MapReduce program that counts words in input files excluding stop words.
public class WordCountWithoutStopWords {

  // Mapper that tokenizes input text and filters out stop words.
  public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    // Set to hold stop words for filtering.
    private Set<String> stopWords = new HashSet<>();

    // Setup method to load stop words from a file specified in the job configuration.
    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
      Configuration conf = context.getConfiguration();
      String stopWordsPath = conf.get("stopwords.file");
      if (stopWordsPath != null && !stopWordsPath.isEmpty()) {
        try (BufferedReader reader = new BufferedReader(new FileReader(stopWordsPath))) {
          String line;
          while ((line = reader.readLine()) != null) {
            stopWords.add(line.trim());
          }
        }
      }
```

And finally, in the main function config object sets a configuration parameter, stopwords. File, pointing to the location of a stop words file. This ensures the Mapper class has access to a predefined list of stop words to exclude from the count. The job is then defined with its name, input and output formats, Mapper and Reducer classes, and the output key and value classes, before being executed.

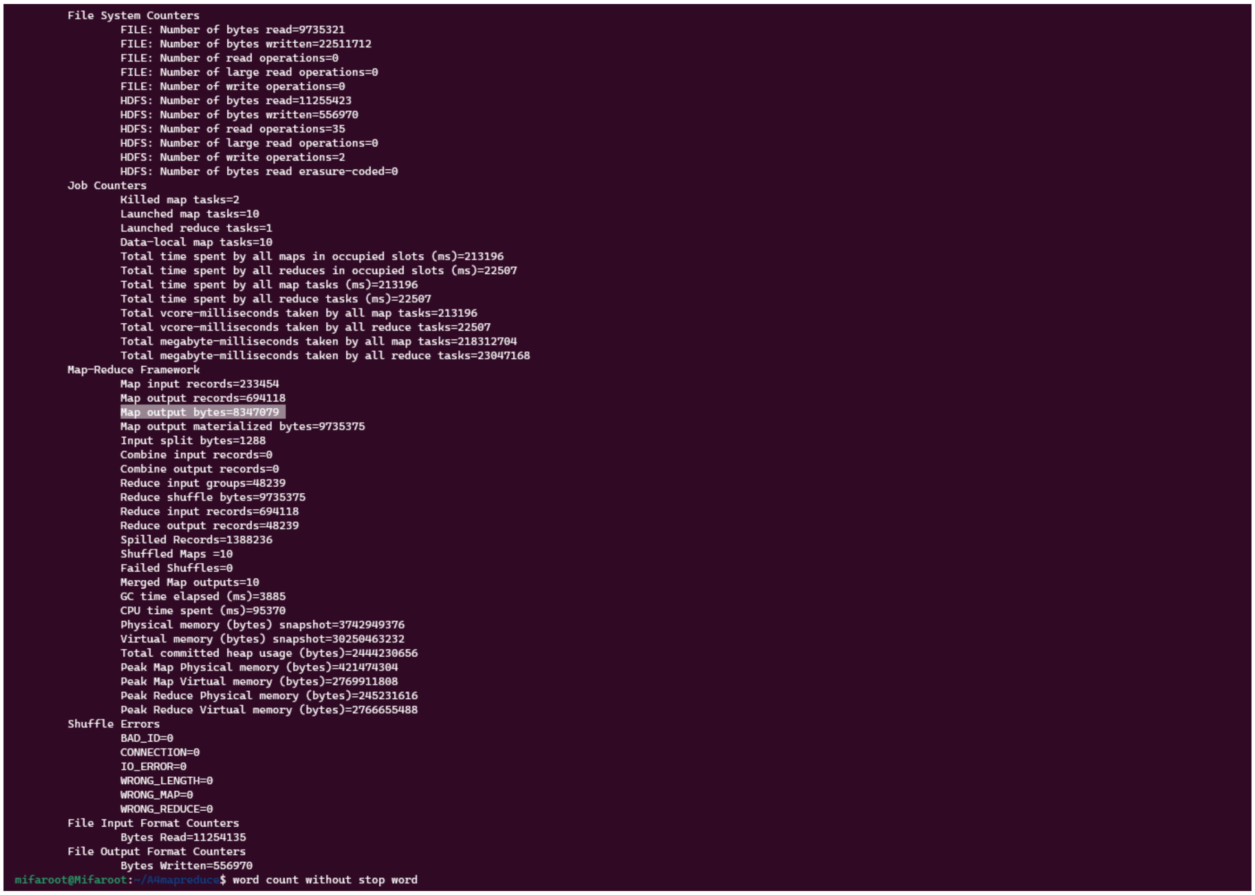
```java
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  conf.set("stopwords.file", "/home/mifaroot/A4mapreduce/stopwords.txt");
  Job job = Job.getInstance(conf, "Word Count Excluding Stop Words");
  job.setJarByClass(WordCountWithoutStopWords.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

6.1. Executing the MapReduce Program and Analyzing Output

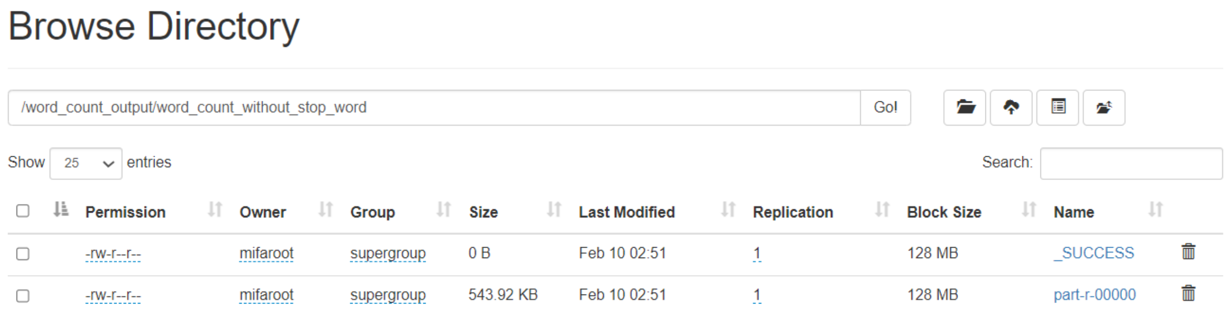
The execution of the ***WordCountWithoutStopWords*** program commenced with the compilation of the Java code, utilizing Hadoop's libraries for MapReduce tasks. This step prepared the program by compiling the **CountWithoutStop.java** file, followed by packaging the compiled classes into a JAR file named word_count_without_stop_word.jar. The program was then executed against a dataset of books stored in /input/books, aiming to count words while excluding common stop words.

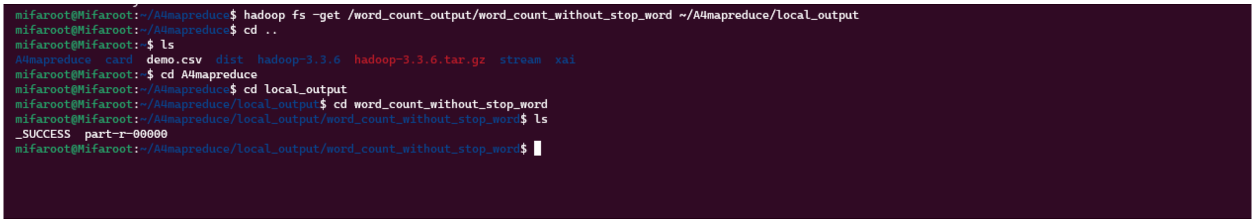
```
javac -classpath `hadoop classpath` -d . CountWithoutStop.java
```

```
mifaroot@Mifaroot:~/A4mapreduce$ ls
 CountWithoutStop.java   SortWord.class   SortWord.java  'WordCount$IntSumReducer.class'  'WordCount$TokenizerMapper.class'   WordCount.class   WordCount.java   data   local_output   stopwords.txt   wordcount.jar
mifaroot@Mifaroot:~/A4mapreduce$ javac -classpath `hadoop classpath` -d . CountWithoutStop.java
mifaroot@Mifaroot:~/A4mapreduce$ jar cf word_count_without_stop_word.jar  CountWithoutStop*.class
mifaroot@Mifaroot:~/A4mapreduce$ hadoop jar word_count_without_stop_word.jar CountWithoutStop /input/books /word_count_output/word_count_without_stop_word
2024-02-10 02:50:47,595 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at /0.0.0.0:8032
2024-02-10 02:50:48,458 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
2024-02-10 02:50:48,508 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/mifaroot/.staging/job_1707483942722_0011
2024-02-10 02:50:49,530 INFO input.FileInputFormat: Total input files to process : 10
2024-02-10 02:50:49,653 INFO mapreduce.JobSubmitter: number of splits:10
2024-02-10 02:50:49,962 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1707483942722_0011
2024-02-10 02:50:49,963 INFO mapreduce.JobSubmitter: Executing with tokens: []
2024-02-10 02:50:50,370 INFO conf.Configuration: resource-types.xml not found
2024-02-10 02:50:50,371 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2024-02-10 02:50:50,577 INFO impl.YarnClientImpl: Submitted application application_1707483942722_0011
2024-02-10 02:50:50,680 INFO mapreduce.Job: The url to track the job: http://Mifaroot.:8088/proxy/application_1707483942722_0011/
2024-02-10 02:50:50,681 INFO mapreduce.Job: Running job: job_1707483942722_0011
2024-02-10 02:51:03,046 INFO mapreduce.Job: Job job_1707483942722_0011 running in uber mode : false
2024-02-10 02:51:03,049 INFO mapreduce.Job:  map 0% reduce 0%
2024-02-10 02:51:26,942 INFO mapreduce.Job:  map 20% reduce 0%
2024-02-10 02:51:27,981 INFO mapreduce.Job:  map 30% reduce 0%
2024-02-10 02:51:29,010 INFO mapreduce.Job:  map 50% reduce 0%
2024-02-10 02:51:33,203 INFO mapreduce.Job:  map 60% reduce 0%
2024-02-10 02:51:47,645 INFO mapreduce.Job:  map 70% reduce 0%
2024-02-10 02:51:48,671 INFO mapreduce.Job:  map 90% reduce 0%
2024-02-10 02:51:49,698 INFO mapreduce.Job:  map 100% reduce 0%
2024-02-10 02:51:52,733 INFO mapreduce.Job:  map 100% reduce 100%
2024-02-10 02:51:53,762 INFO mapreduce.Job: Job job_1707483942722_0011 completed successfully
2024-02-10 02:51:53,981 INFO mapreduce.Job: Counters: 55
```



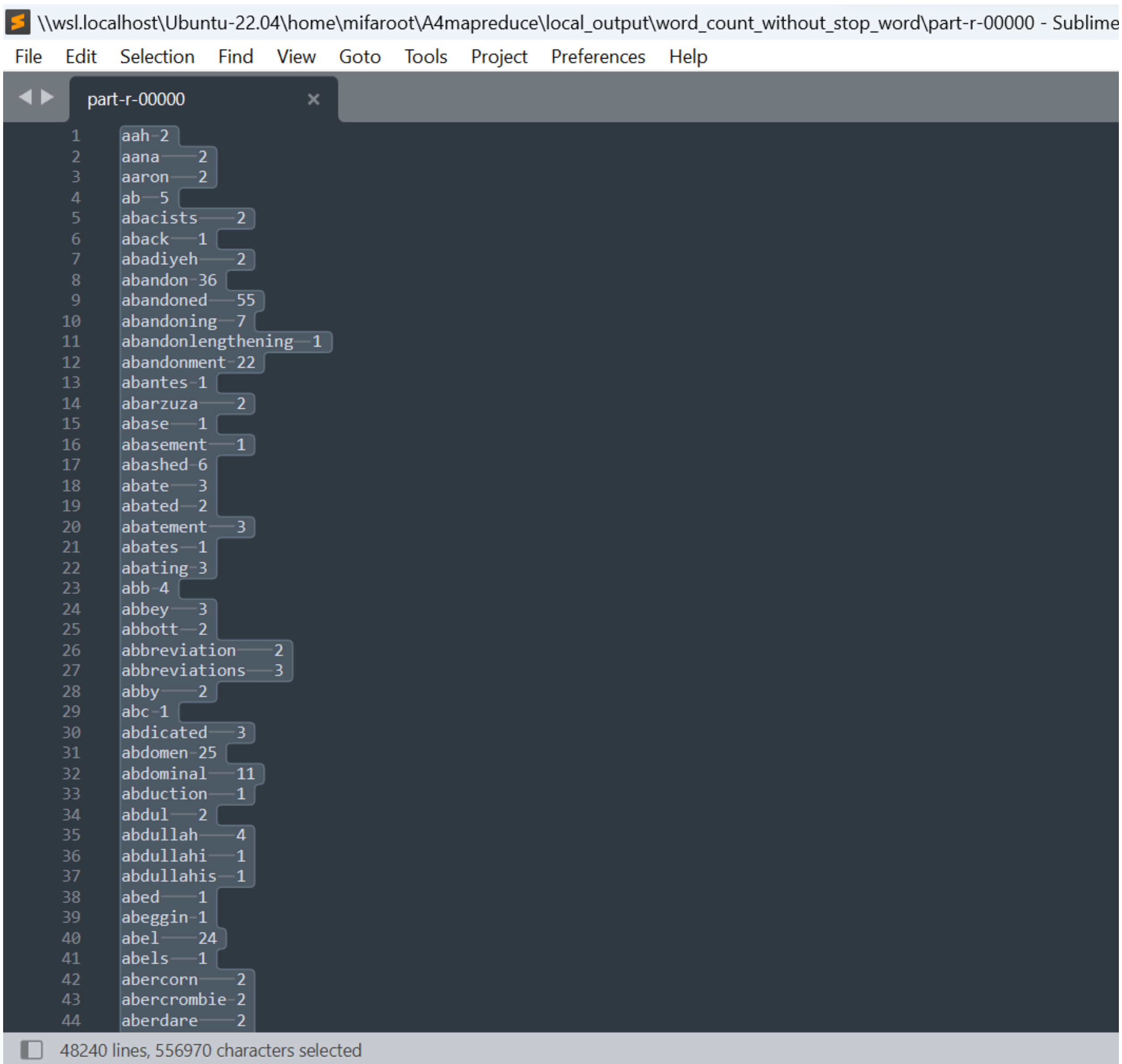
Now let's get the output files form the Hadoop to local ***hadoop fs -get /word_count_output/word_count_without_stop_word ~/A4mapreduce/local_output***





Execution of this JAR via Hadoop processed ten books, yielding insightful output that underscored the prevalence of stop words in raw text data. By comparing the initial and modified program results, the substantial impact of excluding stop words became evident, setting the stage for further iterations to explore the visible differences this exclusion presents.

After running the MapReduce job with the modified ***WordCountWithoutStopWords*** program, the results showcased a more refined analysis by excluding common stop words, leading to the identification of 48,239 unique words across the dataset of 10 books. This approach significantly improved the relevance of the word frequency data, providing insights that are more aligned with the content's context rather than being overshadowed by frequent but less informative words. The successful compilation, execution, and output retrieval process not only demonstrated the program's efficiency in processing large datasets but also highlighted the importance of excluding stop words for a more meaningful textual analysis.



The output is currently organized alphabetically, but for a comprehensive analysis of the ten-book dataset, it's essential to reorder the data by the frequency of word occurrences. To achieve this, a Java program, previously developed as WordSort.java, will be utilized. This step is crucial as the upcoming assignment questions rely heavily on identifying words with the highest counts.

```
mifaroot@Mifaroot:~/A4mapreduce$ java SortWord local_output/word_count_without_stop_word/part-r-00000
The top 25 words with highest counts
government - 2723
united - 2559
time - 2488
great - 2161
volume - 1995
people - 1610
general - 1596
men - 1579
man - 1576
good - 1395
work - 1274
life - 1175
thought - 1162
years - 1148
south - 1132
war - 1129
long - 1099
british - 1078
order - 1054
day - 1036
american - 980
number - 977
hand - 955
president - 950
country - 942
mifaroot@Mifaroot:~/A4mapreduce$
```

After refining the word count analysis by excluding stop words, the results highlight more contextually significant words such as "government," "united," and "time" as the top frequencies. This method effectively surfaces key themes and topics within the texts, demonstrating the impact of removing common stop words for deeper textual insights.

7. Analysis and answer for assignment questions [Questions 1 - 6]

1. What are the 25 most common words and the number of occurrences of each when you do not remove stop words? here are 25 most common words and the number of occurrences of each when you do not remove stopwords:

Answer:

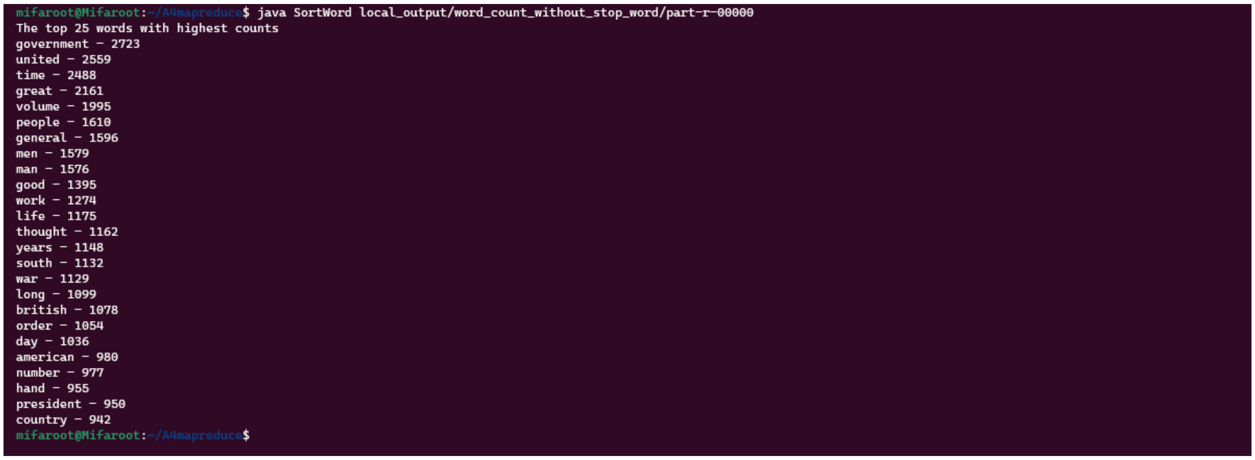
```
mifaroot@Mifaroot:~/A4mapreduce$ java SortWord local_output/word_count_with_stop_word/part-r-00000
The top 25 words with highest counts
the - 128046
of - 79598
and - 52442
to - 49129
in - 36484
a - 35970
that - 20586
is - 16232
it - 16069
as - 15305
was - 14943
for - 13702
with - 13671
be - 13627
i - 13118
by - 12471
which - 11385
on - 10351
he - 9929
not - 9851
this - 9561
at - 9555
his - 9111
her - 9091
had - 8508
mifaroot@Mifaroot:~/A4mapreduce$
```

1. the - 128046
2. of - 79598
3. and - 52442
4. to - 49129
5. in - 36484
6. a - 35970
7. that - 20586
8. is - 16232
9. it - 16069
10. as - 15305
11. was - 14943
12. for - 13702
13. with - 13671
14. be - 13627
15. i - 13118
16. by - 12471
17. which - 11385
18. on - 10351
19. he - 9929
20. not - 9851
21. this - 9561
22. at - 9555
23. his - 9111
24. her - 9091
25. had - 8508

2. What are the 25 most common words and the number of occurrences of each when you do remove stopwords?

Answer:

Here are 25 most common words and the number of occurrences of each when we remove stopwords:



1. government - 2723
2. united - 2559
3. time - 2488
4. great - 2161
5. volume - 1995
6. people - 1610
7. general - 1596
8. men - 1579
9. man - 1576
10. good - 1395
11. work - 1274
12. life - 1175
13. thought - 1162
14. years - 1148
15. south - 1132
16. war - 1129
17. long - 1099
18. british - 1078
19. order - 1054
20. day - 1036
21. american - 980
22. number - 977
23. hand - 955
24. president - 950
25. country - 942

3. Based on the output of your application, how does removing stop words affect the total amount of bytes output by your mappers? Name one concrete way that this would affect the performance of your application.

Answer:

Excluding stop words decreases the overall byte output from mappers due to fewer key-value pairs being generated. This variation in byte reduction largely depends on the frequency of stop words present in the input data. By minimizing intermediate data, this strategy enhances performance, notably reducing the necessary file I/O operations and the volume of data needing to be transferred over the network, thereby streamlining the processing efficiency.

Data with stop words:

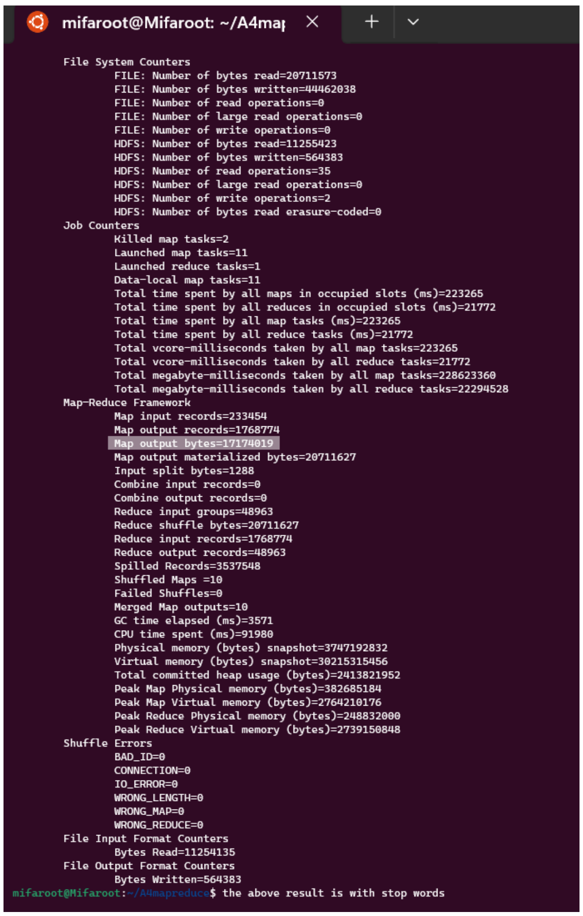
- Map input records=233454
- Total map output bytes: 17,174,019 bytes
- File Output Bytes Written: 564,383 bytes

Without stop words:

- Map input records=233454
- Total map output bytes: 8,347,079 bytes
- File Output Bytes Written: 556,970 bytes

Eliminating stopwords significantly enhances system performance by reducing the amount of data transferred from mappers to reducers. For instance, data sent by mappers drops from 17,174,019 bytes to 8,347,079 bytes without stopwords, leading to quicker processing and improved scalability. Moreover, the output size decreases from 564,383 bytes with stopwords to 556,970 bytes without, underscoring the reduction in data volume and subsequent resource usage. Therefore, excluding stopwords not only speeds up processing but also optimizes resource utilization, contributing to the overall system efficiency.

With stop words



Without Stop Word

4. Based on the output of your application, what is the size of your keyspace with and without removing stopwords? How does this correspond to the number of stopwords you have chosen to remove?

- Number of stopwords used in stopwords.txt: 851
- Data with stop words:
  - Total reduce input groups (keyspace size): 48,963
- Data without stop words:
  - Total reduce input groups (keyspace size): 48,239

Difference=48963-48239=724

With stop word

```
Map-Reduce Framework
        Map input records=233454
        Map output records=1768774
        Map output bytes=17174019
        Map output materialized bytes=20711627
        Input split bytes=1288
        Combine input records=0
        Combine output records=0
        Reduce input groups=48963
        Reduce shuffle bytes=20711627
        Reduce input records=1768774
        Reduce output records=48963
        Spilled Records=3537548
        Shuffled Maps =10
        Failed Shuffles=0
        Merged Map outputs=10
        GC time elapsed (ms)=3571
        CPU time spent (ms)=91980
        Physical memory (bytes) snapshot=3747192832
        Virtual memory (bytes) snapshot=30215315456
        Total committed heap usage (bytes)=2413821952
        Peak Map Physical memory (bytes)=382685184
        Peak Map Virtual memory (bytes)=2764210176
        Peak Reduce Physical memory (bytes)=248832000
        Peak Reduce Virtual memory (bytes)=2739150848
```

Without Stopwords

```
Map-Reduce Framework
        Map input records=233454
        Map output records=694118
        Map output bytes=8347079
        Map output materialized bytes=9735375
        Input split bytes=1288
        Combine input records=0
        Combine output records=0
        Reduce input groups=48239
        Reduce shuffle bytes=9735375
        Reduce input records=694118
        Reduce output records=48239
        Spilled Records=1388236
        Shuffled Maps =10
        Failed Shuffles=0
        Merged Map outputs=10
        GC time elapsed (ms)=3885
        CPU time spent (ms)=95370
        Physical memory (bytes) snapshot=3742949376
        Virtual memory (bytes) snapshot=30250463232
        Total committed heap usage (bytes)=2444230656
        Peak Map Physical memory (bytes)=421474304
        Peak Map Virtual memory (bytes)=2769911808
        Peak Reduce Physical memory (bytes)=245231616
        Peak Reduce Virtual memory (bytes)=2766655488
```

Excluding stopwords reduced the dataset's unique word count from 48,963 to 48,239, indicating a removal of 724 stopwords. So, the keys-pace size is probably related to the number of unique words in the books. So total number of stop words should reduce to: 48963-48,239=724.

5. Let's now assume you were going to run your application on the entirety of Project Gutenberg. For this question, assume that there are 100TB of input data, the data is spread over 10 sites, and each site has 20 mappers. Assume you ignore all but the 25 most common words that you listed in question 2. Furthermore, assume that your combiners have been run optimally, so that each combiner will output at most 1 key-value pair per key.

a) How much data will each mapper have to parse?
b) What is the size of your keyspace?
c) What is the maximum number of key-value pairs that could be communicated during the barrier between mapping and reducing?
d) Assume you are running one reducer per site. On average, how many key-value pairs will each reducer have to handle?

Answer

a. How much data will each mapper have to parse?

We have:

- 100 TB of input data
- The input data is spread to sites = 10 sites
- Number of mappers = 20

Data per site $=\frac{Total\ imput\ data}{Numbesr\ of\ sites} = \frac{100TB}{10sites} = 500GB\ data\ per\ site$

$$The\ data\ each\ maper\ needs\ to\ parse = \frac{Data\ per\ site}{Number\ of\ mapper} = \frac{10TB}{20}$$

So, each mapper will have **0.5T** or 500GB or 500000MB of data to parse.

b. What is the size of your key-space?

0 25 as all the words are excluded.

c. What is the maximum number of key-value pairs that could be communicated during the barrier between the mapping and reducing?

- **Keys per Mapper**: Each mapper can output up to 25 keys.
- **Mappers per Site**: There are 20 mappers operating at each site.
- **Total Sites**: The operation spans across 10 different sites.

To calculate the total key-value pairs:

- First, multiply the number of keys each mapper can produce (25) by the number of mappers per site (20), giving the total keys per site.
- Then, multiply this result by the total number of sites (10) to find the overall total key-value pairs.

$$\mathbf{25\ keys\ per\ mapper \times 20\ mappers\ per\ site \times 10\ sites = 5,000}$$

So, we have 5,000 key-value pairs.

d. Assume you are running one reducer per site. On average, how many key- Value pairs will each reducer have to handle?

- So, since there are 5000 key value pairs for and 10 sites, so each reducer will handle **500** key value pairs worth of data.

The answer is **5000** key value pairs.

6. Draw the data flow diagram for question 5. The diagram should be similar to the diagram shown in lecture. On your diagram, label the specific quantities you got for 5 a, b, c, and d.

Answer:

20 splits, 20 mappers, 20 combiners per site

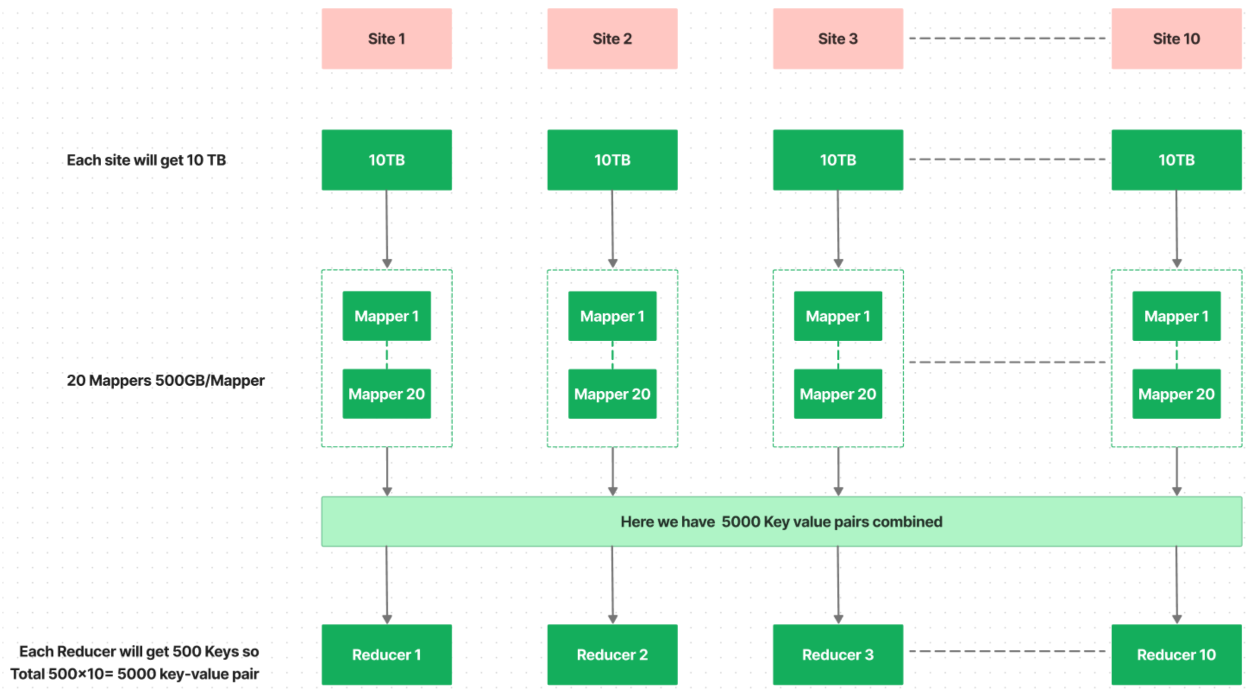


Diagram-link: https://www.figma.com/@mintesnotfikir

Each Reducer will get 500 Keys so total 500x10= 5000 key-value pair