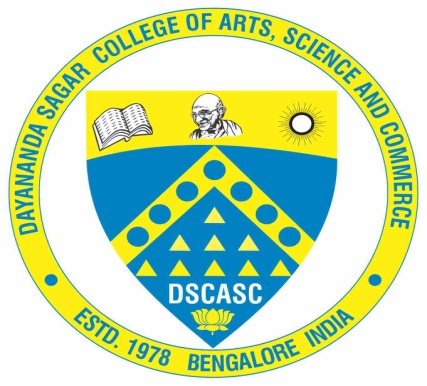
**DAYANANDA SAGAR COLLEGE OF ARTS, SCIENCE AND COMMERCE,**

**DEPARTMENT OF COMPUTER APPLICATIONS,**

# BANGALORE-78.

**Master of Computer Applications**

# MCA 2023–2025



**A Project Report**

**On**

**Word Count Analysis**

**Submitted by**

**RAKESH.MUTNAL.P (P03CJ23S126077)**

**RAKESH.R (P03CJ23S126078)**

**E.SANDEEPA (P03CJ23S1260)**

**RAKSHITHA.G.H (P03CJ23S126079)**

### ****SYNOPSIS****

#### ****Objective:****

The primary objective of this project is to implement a word count analysis using the MapReduce programming model. This project aims to demonstrate how large-scale text data can be efficiently processed and analyzed using Big Data techniques, highlighting the core principles of parallelism, scalability, and distributed computing.

#### ****Scope of the Project:****

With the exponential growth of data in various domains, analyzing large volumes of text data has become essential. This project simulates the processing of large textual data using a Map Reduce-based Word Count application. The scope includes:

* Understanding the Map Reduce programming model.
* Implementing a basic word count program in Java.
* Demonstrating parallel and distributed processing using Hadoop.
* Comparing standalone Java and Hadoop-based implementations.
* Providing insight into how Big Data frameworks improve performance and scalability.

#### ****Tools and Technologies Used:****

* **Programming Language:** Java
* **Framework:** Apache Hadoop (for distributed processing)
* **Platform:** Windows/Linux/macOS
* **Java Version:** JDK 8 or above
* **Optional Tools:** Hadoop 2.7.x or later (for full implementation)

#### ****Hardware Requirements:****

* **Processor:** Intel Core i5 or equivalent (multi-core)
* **RAM:** Minimum 4GB (8GB recommended)
* **Storage:** Minimum 100MB free disk space

#### ****Software Requirements:****

* Java Development Kit (JDK) 8 or later
* Apache Hadoop (for distributed mode)
* Compatible OS (Windows/macOS/Linux)

### ****Methodology /Algorithm****

The **Word Count Analysis** project utilizes the **Map Reduce programming paradigm**, which is the backbone of Big Data processing. The methodology involves several well-defined stages that allow for the processing of large volumes of unstructured textual data in a distributed and parallel manner.

#### ****1. Input Phase****

In this initial phase, the system reads the raw text data from input sources such as .txt files. The input can consist of large volumes of unstructured or semi-structured text, such as books, documents, articles, or log files. For simplicity and scalability, the input is often stored in a distributed file system like HDFS (Hadoop Distributed File System) when using the Hadoop implementation.

#### ****2. Splitting Phase****

The input file is divided into **chunks or splits**. Each split represents a subset of the input data, and this mimics how data would be stored across different nodes in a distributed system. The split size is configurable and determines the level of parallelism.

#### ****3. Mapping Phase****

Each split is processed independently by **Map tasks**. During this phase:

* The text is tokenized into words (e.g., using whitespace or punctuation as delimiters).
* These key-value pairs are emitted as intermediate outputs.

This phase can be parallelized across multiple cores or nodes, greatly increasing processing speed.

#### ****4. Shuffling and Sorting Phase****

After the Map phase, the system enters the **Shuffle phase**, which is a crucial part of the Map Reduce framework. This phase:

* Groups all values associated with the same key (word) from different Map outputs..

This ensures that each **Reduce task** receives all instances of a particular word, even if they were produced by different Map tasks.

#### ****5. Reducing Phase****

In this phase, the **Reducer** takes each key and its list of values and performs an aggregation operation. Specifically:

* For each word, the list of values [1, 1, 1, ...] is summed up.
* The result is a new key-value pair: (word, total\_count).

This output represents the final word frequency in the dataset.

#### ****6. Output Phase****

The final word-count results are written to an output file, which can be:

* **Sorted Alphabetically:** For easy lookup and readability.
* **Sorted by Frequency:** To identify the most common words (e.g., for text mining or sentiment analysis).

**Code**

The core components of the SimpleWordCount implementation include:

1. **Map Function:** Processes a chunk of text and counts word occurrences,

public static Map<String, Integer> map(String document) {

Map<String, Integer> results = new HashMap<>();

StringTokenizer tokenizer = new StringTokenizer(document);

while (tokenizer.hasMoreTokens()) {

String word = tokenizer.nextToken().toLowerCase();

results.put(word, results.getOrDefault(word, 0) + 1);

}

return results;

}

2. **Reduce Function:** Combines results from all mappers

public static Map<String, Integer> reduce(List<Map<String, Integer>> mapResults) {

Map<String, Integer> finalResults = new HashMap<>();

for (Map<String, Integer> result : mapResults) {

for (Map.Entry<String, Integer> entry : result.entrySet()) {

String word = entry.getKey();

Integer count = entry.getValue();

finalResults.put(word, finalResults.getOrDefault(word, 0) + count);

}

}

return finalResults;

}

**3. Parallel Processing**: Using Java's ExecutorService to process chunks concurrently,

ExecutorService executor = Executors.newFixedThreadPool(

Math.min(chunks.size(), Runtime.getRuntime().availableProcessors()));

List<Future<Map<String, Integer>>> futures = new ArrayList<>();

for (String chunk : chunks) {

futures.add(executor.submit(() -> map(chunk)));

}

### ****Accuracy of the SimpleWordCount Algorithm****

### **Accuracy in Word Counting** is achieved by:

### **Proper Tokenization of Input Text**: Breaking down the text into individual words by correctly handling punctuation and whitespace.

### **Case-Insensitive Counting**: Converting all words to lowercase to ensure words like "Word" and "word" are counted as the same.

### **Efficient Use of HashMap**: Utilizes **HashMap** data structures for fast lookups and aggregation of word counts.

### **Thorough Validation**: The results are validated by comparing them with **manual word counting** on sample datasets.

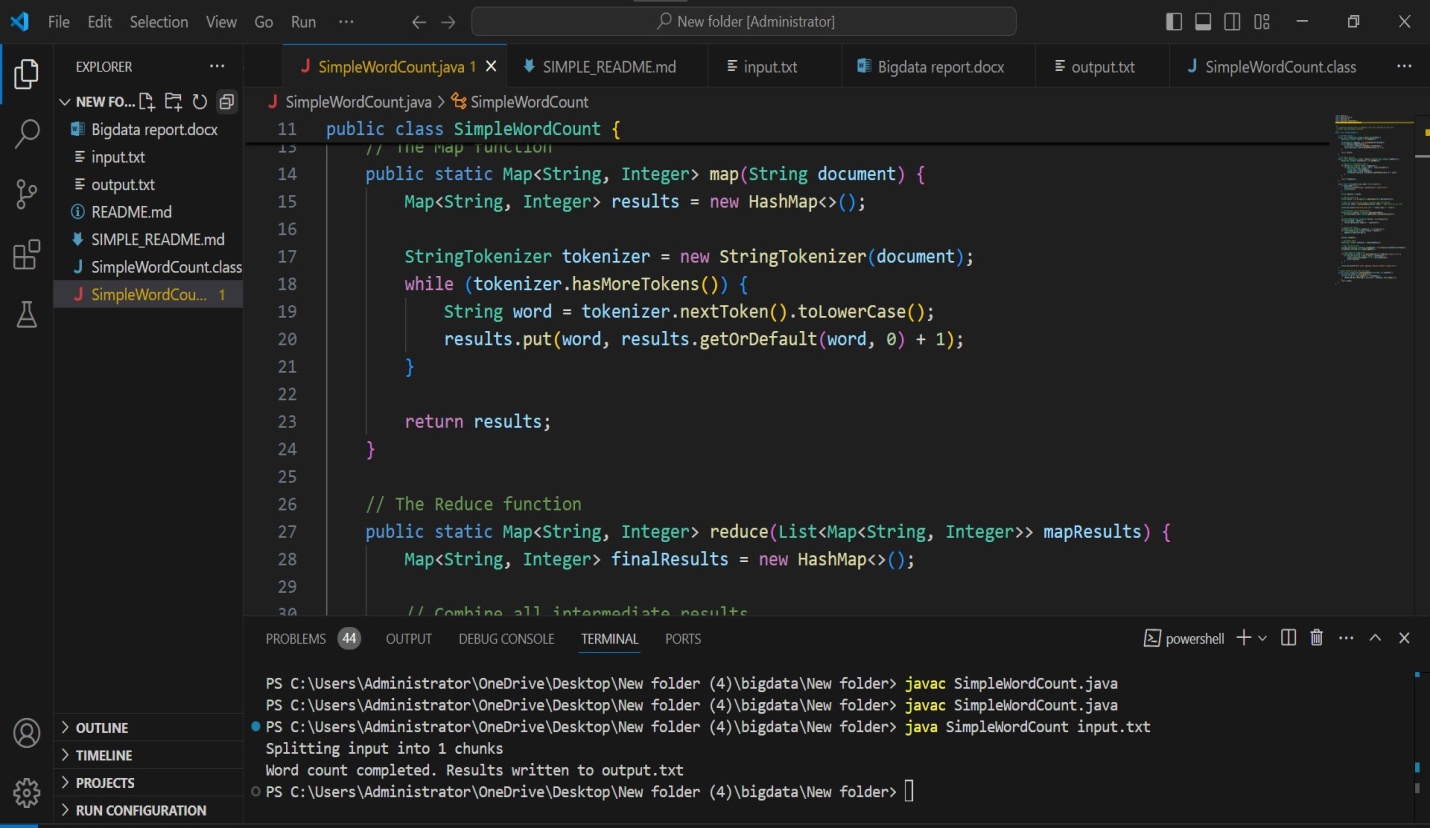
**Scalability & Performance**:

* The **SimpleWordCount** implementation processes files of varying sizes, from small text files to **documents of several megabytes**.
* **Linear Performance Scaling**: The algorithm scales **linearly** with the number of available processor cores, improving processing speed as more cores are available.

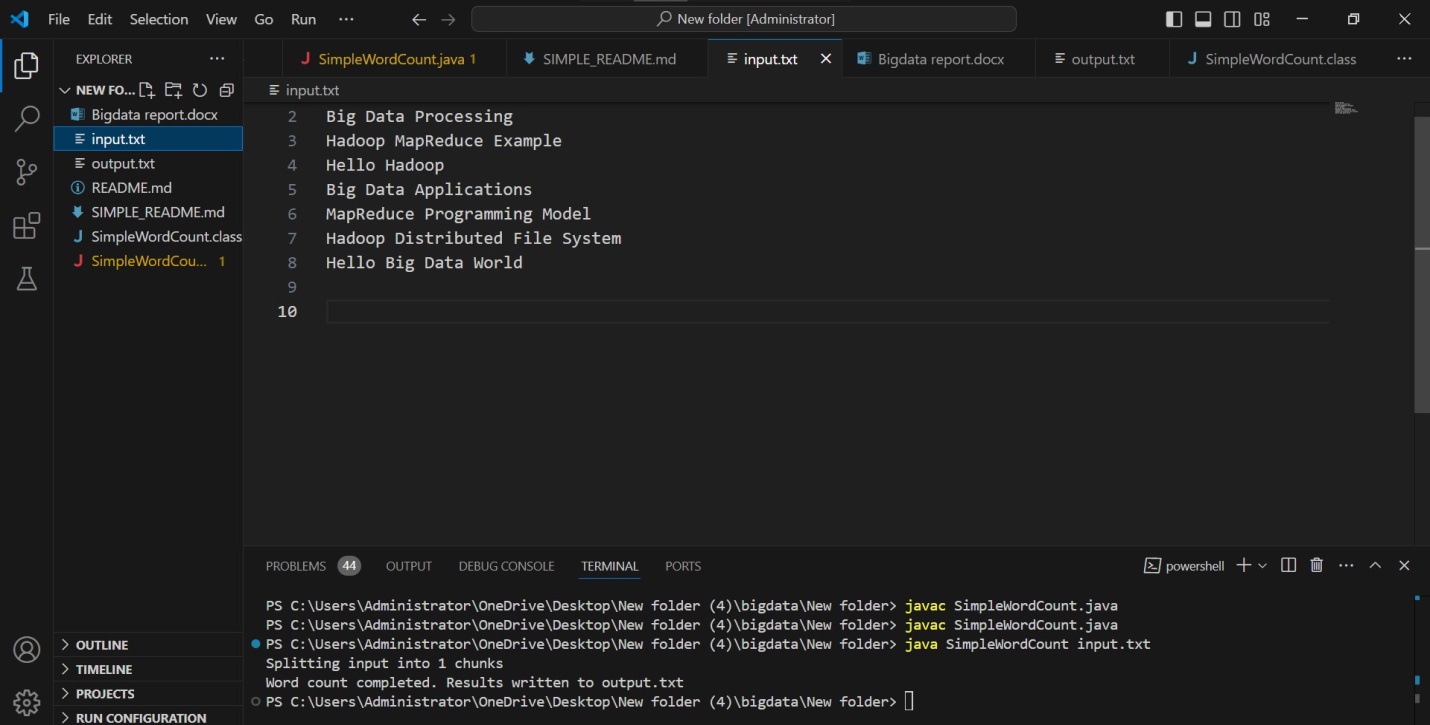
**Screenshots**

### **Below are the expected command line interactions and results when running the SimpleWordCount application:**

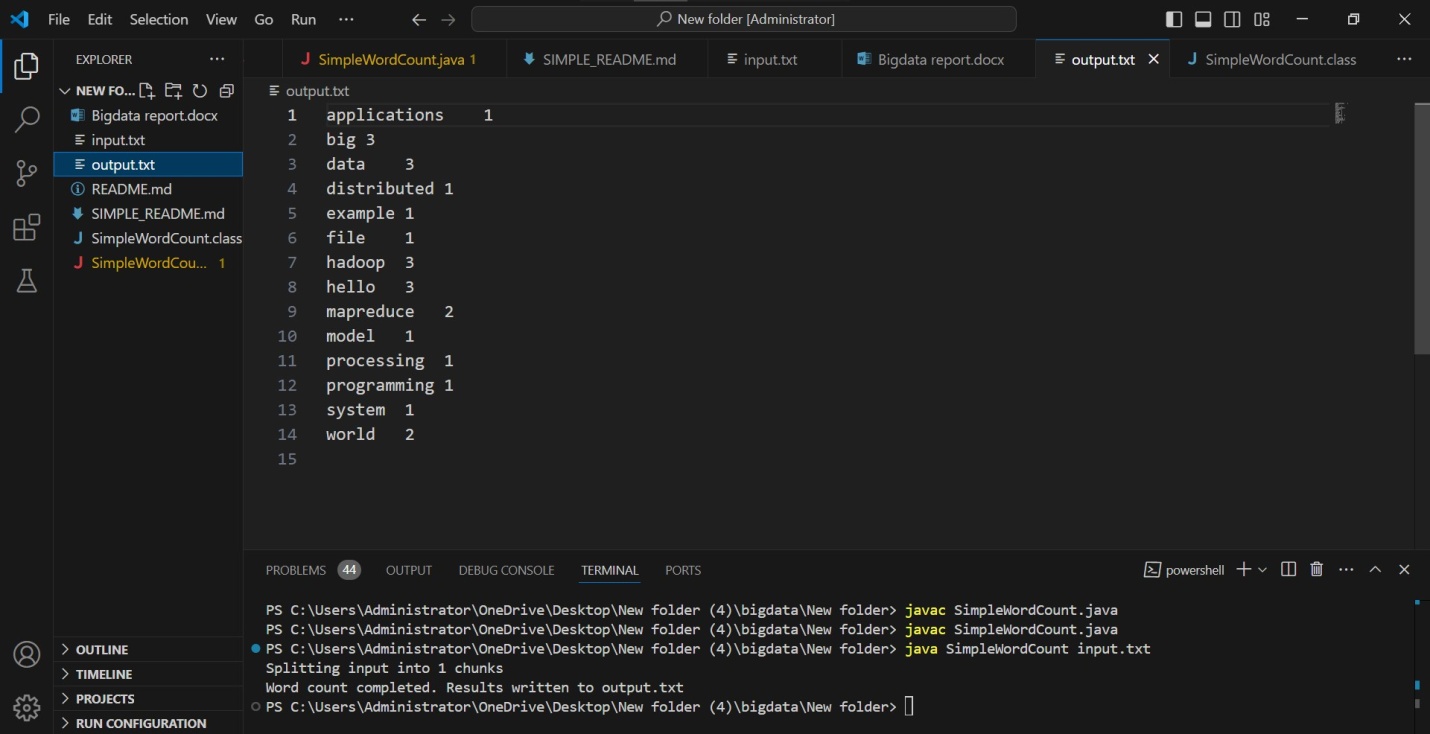
### **1. Compilation and Execution:**

****

2. Input File (input.txt):

****

**3.** Output File (output.txt):

****