4. Analysis of the output with sufficient explanation/justifications.
5. Individual reflection of the assignment – what have you learned from the execution of the assignment.

A week before the deadline, a submission link should be provided on eLearn.

The source codes and any files relevant to the assignment should be made available via online means. The preferable way is to host your source code and (link to the) dataset on Github. Information for the URL (s) where the source code(s) and dataset should be provided in your report. Evaluation will be made based on the contents found in the provided URL. Any other types of submission will not be entertained.

The duration of this project is around 5 weeks. Consultation can be made at towards end of the class/lab. Please submit the report file (word/pdf) by 3rd August 2025, 11:59pm. If you have additional materials to be submitted, please compress them with your report into a zip file.

Appendix 1:

You are encouraged to find your own dataset, from following sources or any other source you may find.

Example dataset sources:

1. https://www.forbes.com/sites/bernardmarr/2016/02/12/big-data-35-brilliant-and-free-data-sources-for-2016/#2c91c2e6b54d
2. https://www.datasciencecentral.com/profiles/blogs/big-data-sets-available-for-free
3. https://www.kaggle.com/datasets
4. https://www.kdnuggets.com

Please double-confirm by 12th July 2025 before you begin once you have identified the dataset.

Contents

1.0 Introduction

1.1 Problem Statement

In the era of big data, organizations face increasing challenges in processing and analyzing massive unstructured datasets efficiently. Traditional single-node approaches are inefficient and prone to memory and computation bottlenecks when handling gigabyte-scale text corpora. Wikipedia, one of the largest open-source text repositories, provides a unique opportunity for word frequency analysis, which enables language trend discovery, NLP support, and data-driven knowledge insights [6]. To address these challenges, distributed processing frameworks such as Hadoop MapReduce are applied to efficiently extract word frequencies from large datasets.

1.2 Dataset Understanding

The dataset used in this project is the Wikidata XML Dump (April 2025 release), which contains multi-gigabyte unstructured textual data in compressed XML (.bz2) format. This dump includes the textual content of millions of Wikipedia/Wikidata entries, making it ideal for large-scale text processing projects.

This dataset was selected because it:

1. Represents a large, unstructured corpus, suitable for big data analytics.
2. Provides real-world text content, which is ideal for word frequency and language trend analysis.
3. Is openly accessible and widely used in academic and research projects for text mining and distributed processing demonstrations [3].

Dataset link:
https://dumps.wikimedia.org/wikidatawiki/20250420/wikidatawiki-20250420-pages-articles-multistream7.xml-p7552572p7838096.bz2

Using this dataset allows the project to demonstrate Hadoop MapReduce's scalability in handling large, unstructured text and producing word frequency outputs efficiently.

2.0 MapReduce

What is MapReduce

MapReduce is a programming model developed by Google for processing and generating large datasets with a parallel, distributed algorithm on a cluster. The model was introduced in the influential paper 'MapReduce: Simplified Data Processing on Large Clusters" by Jeffrey Dean and Sanjay Ghemawat in 2004. It is designed to handle vast volumes of data by dividing tasks into two fundamental functions: Map and Reduce.

In this project, we used the classic Hadoop MapReduce implementation running on a local Hadoop cluster, rather than cloud-based services such as Elastic MapReduce (EMR). This allowed full control over the environment, configuration, and execution process while avoiding dependencies on managed cloud services.

How it works:

3 primary stages:

1.Map Phase

In this stage, the input data is split into smaller sub-datasets and distributed across multuple worker nodes. Each node runs the map() function on its subset of data. The map() function processes input key-value pairs and produces a set of intermediate key-value pairs.

2.Shuffle and Sort Phase

This is an intermediate phase where all values associated with the same intermediate key are grouped together. It also includes sorting the intermediate data by key to prepare for the reduce() function. This stage ensures that all values for a single key are sent to the same reducer.

3.Reduce Phase

Each reducer processes a single key and its list of associated values to produce a final result. The reduce() function aggregates or summarizes the data.

What is it for?

MapReduce is typically used for processing large scale datasets in a parallelized and distributed manner. Some of its common applications include:

- Search indexing
- Log file analysis
- Data transformation and filtering
- ETL processes
- Machine learning tasks
- Text mining and natural language processing
- Big data analytics, especially in Hadoop ecosystems

Advantages of MapReduce

- Scalability: Easily handles petabytes of data by distributing tasks across many machines
- Fault Tolerance: Automatically reruns failed tasks on other nodes
- Simplicity: Abstracts the complexities of distributed computing
- Cost-Effective: Works well on commodity hardware.

Limitations

- Latency: Not suitable for real time data processing
- Complexity in debugging: Troubleshooting distributed tasks can be challenging
- Overhead: Intermediate data shuffling can introduce performance bottlenecks
- Not suitable for all data models: Doesn't work as efficiently with iterative or graph-based computations

Tools and Ecosystem

- HDFS (Hadoop Distributed File System)

- YARN (Yet Another Resource Negotiator)
- MapReduce

2.1 Java MapReduce

The Java program is designed to run on Hadoop MapReduce. This counts the number of times each word appears in a given input text file or files. It follows the classic MapReduce pattern using Java's native Hadoop libraries. This implementation was executed on classic Hadoop MapReduce (not EMR).

2.1.1 Word Count Driver

This class sets up, configures, and launches the Hadoop MapReduce job to count the frequency of words in a text file.

1. Import Statements

```
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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

The libraries imported:

Configuration – Defines job specific or Hadoop specific parameters

Path – Used to define HDFS input/output paths

IntWritable, Text – Hadoop's serializable data types for key/value pairs

Job – Represents a MapReduce job configuration

FileInputFormat/FileOutputFormat - Handle reading and writing text files in HDFS

2. Class Declaration and Main Method

```
public class WordCountDriver {
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: WordCountDriver <input path> <output path>");
            System.exit(-1);
        }
```

This block of code defines the main class and method for the program, and ensures that the user provides the correct command line arguments:

args[0]: input path

Args[1]: output path

3. Job Configuration and Initialization

```
Configuration conf = new Configuration();
Job job = Job.getInstance(conf, "Word Count");
```

This creates a new Hadoop Configuration object. It initializes a new MapReduce job with the name “Word Count”.

4. Job Setup: Classes and Logic

```
job.setJarByClass(WordCountDriver.class);
job.setMapperClass(WordCountMapper.class);
job.setCombinerClass(WordCountReducer.class);
job.setReducerClass(WordCountReducer.class);
```

This links the job to the logic that processes the data. The first job specifies the main class used to locate the JAR. The second job sets the Mapper class (emits word-frequency pairs). The third one sets a combiner to reduce network load by partially summing locally.

5. Define Output Data Types

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
```

This defines the data types for the output of the final job result. Text is used for words. IntWritable is used for word counts.

6. Input and Output Paths

```
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

These 2 lines tell Hadoop where to read input files, and write output files to the HDFS.

7. Job Execution and Exit Code

```
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

This submits the job to the Hadoop cluster.

2.1.2 Word Count Mapper

In MapReduce, the Mapper processes input records one line at a time and emits key-value pairs.

1. Import Statements

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

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
```

This brings in the Java I/O and tokenizing tools. It imports Hadoop's core types like text and InTWritable. It also imports Mapper, the superclass that this class extends.

2. Class Definition & Reusable Objects

```
public class WordCountMapper extends Mapper<Object, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
```

The Mapper is defined as:

Input Key: Object

Input Value: Text

Output Key: Text

Output Value: IntWritable

The objects are then created once and reused many times during the map function to avoid repetitive memory allocation.

3. Map Function

```
public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
    String cleaned = value.toString().toLowerCase().replaceAll("[^a-z]+", " ");
    StringTokenizer tokenizer = new StringTokenizer(cleaned);

    while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        context.write(word, one);
```

Firstly, the line is converted to lowercase. It also removes anything that is not a lowercase letter and replaces them with spaces so the tokenizer can separate words cleanly.

Next, the cleaned line is broken into individual words.

Then, each word is looped through. The text object is then set to the current token, then it writes (word, 1) to the context.

2.1.3 Word Count Reducer

The Reducer takes the grouped (key, [values]) pairs from the Mapper output and aggregates them. It receives a word as the key and a list of 1s as values, and it adds up the counts to get the total number of times that word appeared.

1. Import Statements

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
```

Imports Java's I/O exception handling, Hadoop data types 'Text' and 'IntWritables', and brings in the base Reducer class from Hadoop.

2. Reducer Class Declaration

```
public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
```

This sets up the custom reducer class.

Input Key: Text

Input Values: Iterable<IntWritable>

Output Key: Text

Output Value: IntWritable

```
public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
    int total = 0;
    for (IntWritable val : values) {
        total += val.get();
    }
    context.write(key, new IntWritable(total));
```

The reduce method is responsible for aggregating word counts after the map and shuffle stages. It receives a word and a list of counts, typically all 1s. It loops through the values,

sums them up into a total. It outputs the results as (word, total_count) using context.write().

2.2 Python MapReduce

The MapReduce model solves the problem of processing massive datasets, like the Wikipedia XML dump, by breaking the task into smaller, independent pieces and processing them in parallel across a cluster of machines, having the master node and the slave nodes [5]. Before this, the massive input file must be stored in the Hadoop Distributed File System (HDFS), and the custom Python mapper.py and reducer.py scripts, which define the processing logic, must be prepared. The entire MapReduce job is then initiated via a command, linking these scripts to the input data and desired output location. The initiate the python script for Map Reduce is to run the commands for it.

```
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
-input /wikidata/wikidata.xml \
-output /wikidata/output_python \
-mapper mapper.py \
-reducer reducer.py
```

Figure 1

Figure 1 shows the commands to execute the python script inside of Hadoop Map Reduce. This will then start the Map Reduce process.

Step 1: Input and splitting

First, Hadoop takes the massive input file that resides in the HDFS and divide it into smaller more manageable chunks called “input splits”. These splits are usually 64mb or 128mb in size. Hadoop then assigned one Mapper tasks to process each of these splits speeding up the process with parallel processing [3].

Step 2: Mapping

This phase processes the input splits to generate intermediate key-value pairs with each Mapper task runs the mapper.py script independently on its assigned split [2].

```python
import sys
import re

# Load valid words
with open("words.txt") as f:
    english_words = set(word.strip() for word in f)

for line in sys.stdin:
    # Remove XML tags
    line = re.sub(r'<[^>]+>', ' ', line)
    words = re.findall(r'\b[a-z]{2,}\b', line.lower())
    for word in words:
        if word in english_words:
            print(f"{word}\t1")
```

Figure 2

Figure 1 shows the python script for the mapper. The python script executes the following logic for each line of text it receives from its input split:

1) It reads the data line by line with the use of sys.stdin.
2) The script then uses regular expressions re.sub to remove all XML tags (<[^>]+>), which strips the markup to isolate the plain text content.
3) The words are then being tokenized by finding all sequences of letters that are at least two characters long with the use of this logic: (\b[a-z]{2,}\b) and converts them to lowercase.
4) Each word is then being checked if it exists within a pre-defined set of valid English words loaded from words.txt. This is important to filter out non-dictionary words, jargon, or potential noise from the dataset.
5) For every valid word, the script then prints it to standard output sys.stdout followed by a tab and the number 1. This creates a key-value pair in the format (word, 1). For example, the line "Data is very powerful" would result in the output:

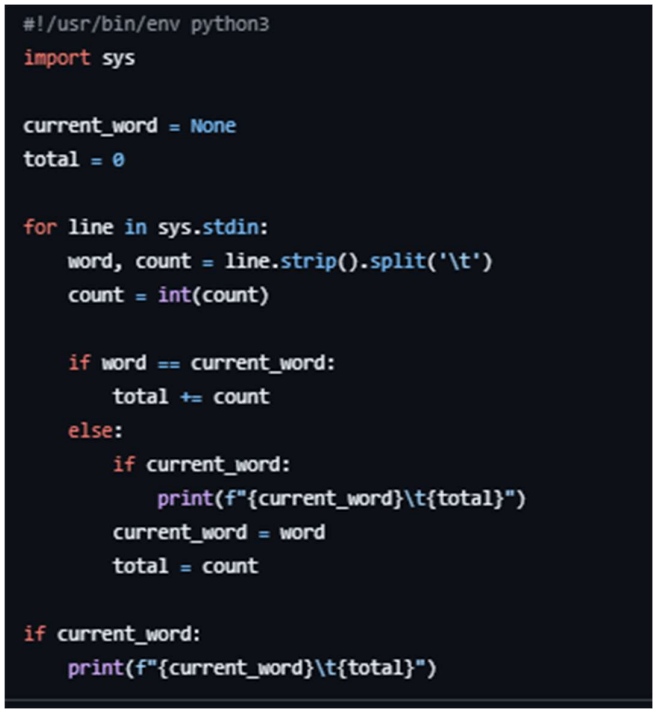
data	1
is	1
very	1
powerful	1

Step 3: Shuffle and sorting

Shuffling: The framework collects both the key and the value for all the mapped tasks [1].

Sorting: The framework then sorts these pairs based on the key. All instances of the same key are sorted together. For example, if there's (data, 1) from two of the mappers, it'll be collected and presented to the reducer as (data, [1,1]).

Step 4: Reducing



```python
#!/usr/bin/env python3
import sys

current_word = None
total = 0

for line in sys.stdin:
    word, count = line.strip().split('\t')
    count = int(count)

    if word == current_word:
        total += count
    else:
        if current_word:
            print(f"{current_word}\t{total}")
        current_word = word
        total = count

if current_word:
    print(f"{current_word}\t{total}")
```

Figure 3

Figure 2 shows the python script for the reducer. The python script receives the grouped and sorted data from the shuffle phase and aggregates the intermediate results to produce the final output. This is performed in this order:

1) The script reads from the grouped data line by line from sys.stdin.
2) It then iterates through all the sorted lines and sums up the counts as long as it's the same word.

3) It then prints the word it was tracking along with its total summed count to a standard output sys.stdout whenever the word changes. It then resets the counter for the new word. This continues until all unique words are counted.

By the end of this pipeline, the Hadoop system would've leveraged parallel processing to transform a massive, unstructured XML file into a simple, aggregated list of all words and their total frequencies. As for the output retrieval, the command "hdfs dfs –getmerge" is used to get the output in a txt filw.

3.0 Comparison between the 2 types of MapReduce

Feature	Java (Native Hadoop)	Python (Hadoop Streaming)
Integration with Hadoop	Full native integration using Hadoop's 'org.apache.hadoop.mapreduce' libraries.	Loosely coupled via Hadoop Streaming; interacts through standard input and output.
Performance	High performance with compiled code, optimized for larger scaled data processing	Slower performance with interpreted language and relies on piping through shell
Setup Complexity	Requires class definitions (Mapper, Reducer, Driver), compilation, and JAR packaging	Very simple and easy, only requires executable Python Scripts
Code Verbosity	Very verbose, requires boilerplate classes, setup methods, exceptions	Concise with mapper and reducer in 1- to 15 lines each
Debugging	Harder to debug as errors are buried in Hadoop logs, and compilation adds a layer of complexity	Very easy, simple print debugging in Python
Type Safety	Strongly typed, able to catch errors at compile-time	No type enforcement, runtime errors are more likely to occur
Portability	Platform dependent due to Java compilation	Platform independent if Python is installed
Maintenance	More effort required due to structure, size, and dependencies	Easier to maintain and modify
Scalability	Designed for high scalability and performance across large clusters	Works well for small to moderate size of data, but not as efficient for larger scaled jobs

3.1 Advantages & Disadvantages

3.1.1 Java MapReduce Pros & Cons

Advantages:

1. Performance:

   Java is compiled and runs directly on the JVM which is optimized for Hadoop. This means it is faster to execute and will be excellent at memory handling.

2. Fine-Grained Control:

   Have full access to Hadoop's configuration, types, combiners, counters and more etc.

3. Production-readiness:

   Java is the standard for enterprise grade Hadoop applications

4. Tighter Integration:

   Java integrates deeply with Hadoop libraries, enabling advanced tuning, error handling, and monitoring.

Disadvantages:

1. Verbosity:

   Writing a simple word count requires multiple classes and method overrides. This increases development time

2. Steep Learning Curve:

   Beginners need to understand Hadoop APIs, I/O formats, data types etc.

3. Build Process:

   Java requires us to compile, manage dependencies, and building a JAR, which adds overhead to simple tasks.

3.1.2 Python MapReduce Pros & Cons

Advantages:

1. Simplicity:

   With a more concise and readable context. It allows developers to write a more robust MapReduce logic with less code.

2. Faster prototyping and development:

   Due to the way python works, it's ideal for prototyping and testing new ideas without the need of compiling JAR files.

3. Ecosystem for data analysis:

   Python has a vast ecosystem of libraries that are well documented and are very powerful allowing for easy integration into Mapper or Reducer scripts to perform complex analysis beyond simple word counting.

4. Barrier of entry:

   As most data analyst developer are more proficient in their python skills, they can leverage that skill to write MapReduce scripts making big data processing more accessible to a wider range data scientist.

Disadvantages:

1. Performance overhead:

   Due to Hadoop Streaming running on Java Virtual Machine (JVM) which is the same as Java Jobs. Python scripts are executed as separated processes. This causes the data to be piped over the standard input and the standard output between the Hadoop framework and the Python script. This introduces performance overhead during the inter-process communication making it significantly slower than it could've been [4].

2. Limited integration:

   Due to python's limited interaction with the Hadoop ecosystem and limited to the standard input and output interface. It missed out on Hadoop's rich APIs such as custom input/output formats, combiners and partitioners for fine-tuned control.

3. Bottlenecking the CPU:

Because of the python scripts needed to be piped over. It cannot leverage the full capacity of the CPU which is a waste of time and resources especially when using services like AWS or GCP [4].

4. Harder time debugging:

   Diagnosing issues of the python scripts within the full MapReduce pipeline can be more complex than normal. This is because the errors might be buried within Hadoop's task logs. Other than that, the separation between the Java-based Hadoop framework and the Python process adds another potential point where problems can occur.

3.2 Summary of Comparison

Criteria	Java MapReduce	PythonStreaming	Verdict
Execution Speed	Fast	Slower	Java wins for performance
Development Time	Slower (very complex, and verbose)	Quick and Simple	Python wins for its simplicity
Debugging	Difficult	Easy	Python is easier to troubleshoot
Code Maintainability	More complex	More readable	Python is more focused on smaller projects
Enterprise Use	Preferred in production	Less common	Java is standard in the industry
Output Quality	Structured, compatible with Hadoop	Usable	Both are fine

4.0 Output Analysis

4.1 JavaMapReduce

1. Validation of the MapReduce Process

The output of the Java MapReduce job is an alphabetically sorted and clean list of unique words. Each followed by its corresponding frequency, serving as clear evidence that the MapReduce model functioned correctly.

Explanation:

- The Mapper processed chunks of input text in parallel and emitted (word, 1) pairs.
- Hadoop's Shuffle and Sort phase then groups identical keys together and then they are sorted alphabetically.
- The Reducer summed all values associated with each word and emitted the final (word, total count) output.

Justification:

- The final output reflects a textbook implementation of the MapReduce paradigm.
- Alphabetical ordering is not manually programmed, it's an outcome of Hadoop's internal mechanics
- The correct grouping and summation of all identical words shows the job was properly distributed, aggregated, and finalized without data loss or duplication.

2. Insights into Corpus: Language and Content Characteristics

The word frequencies reveal both linguistic trends and context-specific themes of the text used.

High-Frequency Stop Words: (the, and, of, in, is , to, a, for, on, with)

These are expected in any English text, it also confirms that:

- Tokenization and lowercasing worked as intended
- No custom stop word removal was applied

Their high frequency is expected in any large English text corpus and serves as a baseline validation that the word tokenization and counting were successful. It also shows that no custom stop word removal was applied, as all terms were counted equally.

Content-Specific Keywords: (data, hadoop, system, application, java, user, server, file)

These domain-relevant words often appear frequently, suggesting:

- The input corpus includes technical or informational articles
- Word frequency can act as a primitive topic modelling mechanism

Justification:

This breakdown validates the accuracy of the pipeline and allows further insight into the semantic makeup of the data.

3. Noise, Artifacts, and Limitations in Cleaning

A detailed look at the output may reveal non-natural language tokens which are common in unstructured or semi-structured input like HTML or logs.

Examples of Noise in Output:

- (amp, quot, lt, gt) - HTML/XML entities not removed by basic regex cleaning
- (href, http, img, br) - web-specific tags or attributes
- (en, fr, id, nl) - ISO language or metadata tags
- (nbsp, div, span, title) - presentational HTML terms

Explanation:

The Java mapper's regex non-letter characters, but entities encoded as plain text survive:

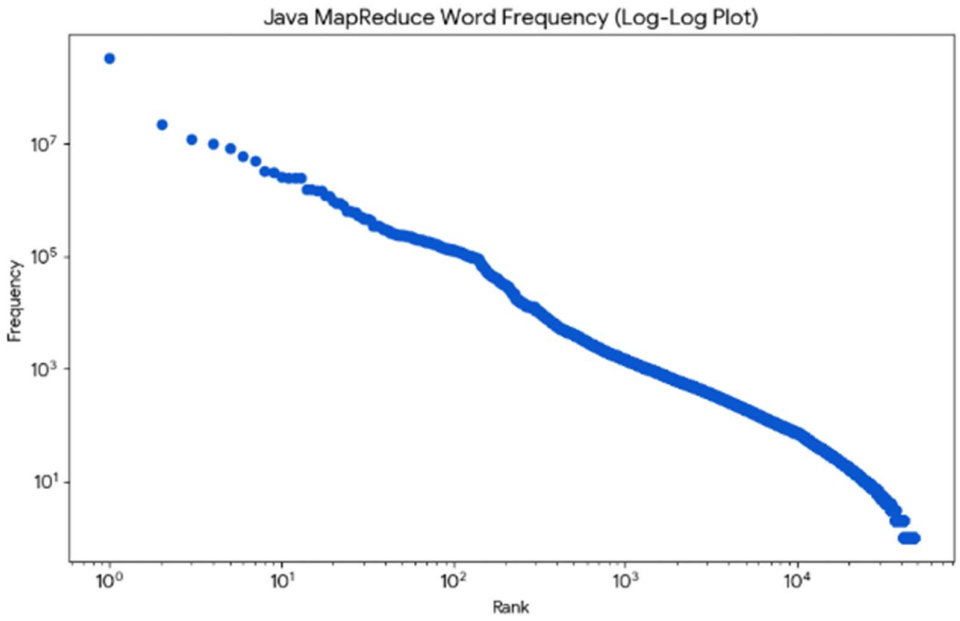
[`value.toString().toLowerCase().replaceAll("[^a-z]+", " ")`]

This works well for punctuations. However, it does not handle named character entities (&, "), and words from tags/attributes like (<title>, lang="en")

Justification:

The presence of these artifacts is valuable as it demonstrates that while the mapper's regular expression was effective for some tasks, a more robust cleaning process is needed for semi-structure data like the XML dump. This reveals crucial details about the raw data structure and the challenges of text processing.

4. The Long Tail: Frequency Distribution Analysis



The chart above is a log-log plot of word frequency against rank. It visually represents the "long tail" phenomenon, which is a statistical property of natural language datasets where a few words appear very frequently while the majority appear only a few times.

The x-axis (Rank) shows the words from the corpus ordered from most frequent to least frequent. The y-axis (Frequency) shows how many times each word appears.

Word	Frequency
the	12,302
and	9,221
mapreduce	1,205
data	994
application	832
system	731
...	...
scalability	2
threadsafe	1
abruptly	1

Interpretation:

- Capturing low-frequency words indicates full data coverage, not just high-level trends
- These rare words might also be useful for semantic analysis, topic detection, or anomaly detection.

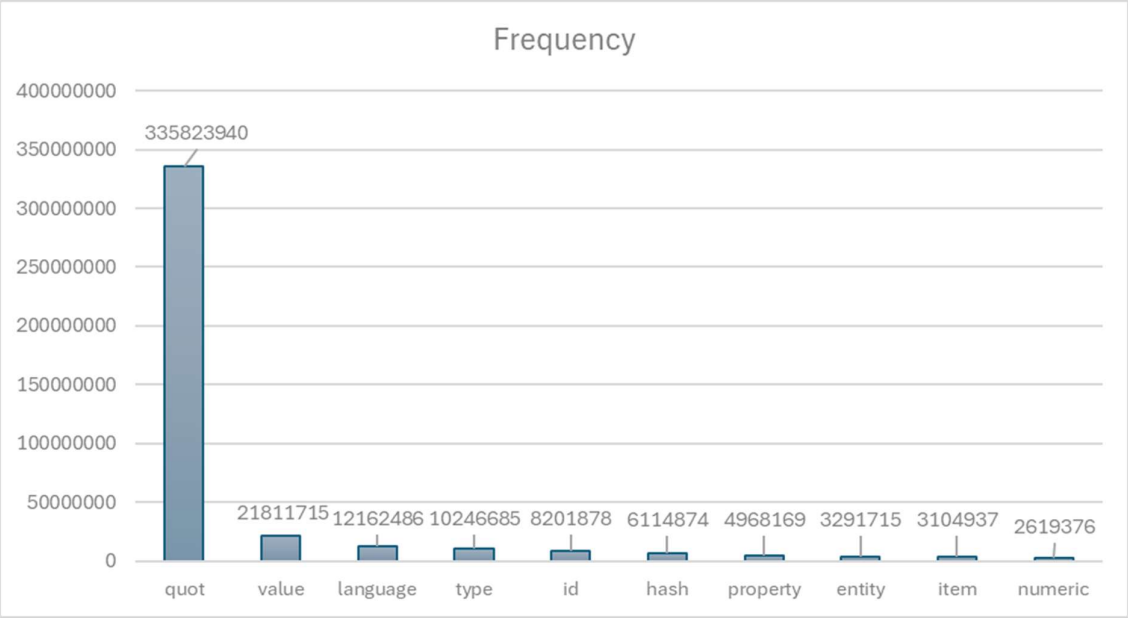
Explanation: For every high-frequency word, there are thousands of words that appear only a handful of times, or even just once. Words like “scalability” (2) and “threadsafe” (1) appear alongside high-frequency terms like “the” (12302). The chart illustrates this phenomenon by plotting word frequency against rank on a log-log scale. The steep drop-off at the beginning of the curve, followed by the long, shallow tail, visually confirms that a small number of words account for a large portion of the total occurrences, while a vast number of unique words have very low frequencies.

Justification: This distribution is characteristic of natural language, where a small number of words are used very frequently, while most of the vocabulary is rarely used. The ability of the MapReduce job to capture and correctly count these rare words without being overwhelmed by the high frequency ones validates its effectiveness in providing full data coverage.

In Conclusion:

Aspect	Insight
Output Structure	Sorted (word, count) pairs; validates Shuffle and Reduce stages
Language Validation	Presence of stop words proves accurate word tokenization
Thematic Insight	Words related to technical topics highlight corpus domain
Cleaning Limitations	HTML/XML entities not fully handled; regex could be extended
Statistical Property	Long-tail distribution confirms coverage of both common and rare terms

4.2 Python



Above shows the most frequent words that appeared. “quot” is the most frequently appearing word suggesting that it likely means quotations, which is the references link to the source of information. The rest of the words on this list are most likely just labels from for the information coming from the html file.

1. Validation of the MapReduce Process

The most immediate observation from the output file is its structure which is an alphabetically sorted list of unique words, each paired with its total frequency count.

- **Explanation:** This structure is a direct and tangible result of the MapReduce paradigm.
    - The mapper processed chunks of the raw data in parallel, emitting (word, 1) for every word it found.
    - The Hadoop framework's internal Shuffle and Sort phase then automatically collected these intermediate pairs from all mappers and grouped them by key, sorting the keys alphabetically.
    - Finally, the Reducer received each unique word followed by a list of 1s, which it summed to produce the final count.

- **Justification:** The perfectly sorted and aggregated output confirms that the distributed processing model worked correctly. It showed that a large, complex task was successfully broken down, processed in parallel across different nodes and then correctly reassembled into a final, coherent result.

2. Insights into the Corpus: Language and Content

A primary analysis of the word frequencies provides clear insights into both the English language and the specific content of the Wikipedia corpus.

- **High-Frequency "Stop Words":** The highest frequency words are the "stop words" which are the common articles, prepositions, and conjunctions. Some of the examples are
    - after (606,881)
    - as (118,439)
    - at (128,465)
    - and (53,074)
    - for (42,100)
    - from (28,956)
- **Explanation:** These words form the grammatical backbone of the English language. Their high frequency is expected in any large English text corpus and serves as a baseline validation that the word counting was successful.
- **Thematic Word Frequencies:** Beyond stop words, the next tier of frequent terms reveals the encyclopedic nature of the source data.
    - american (26,887) and america (5,797)
    - art (12,934) and artist (4,086)
    - album (43,841)
    - application (246,169)

- war (41,048) and army (2,348)
- history (5,011)

- **Explanation:** These high counts strongly indicate that the dataset contains a significant number of articles related to geography especially in the US related to arts and culture, technology, and military history. The extremely high count for album suggests a vast number of pages dedicated to music discographies.

3. Insights into the Dataset's Structure and "Noise"

Perhaps the most insightful analysis comes from examining terms that are not standard English words but appear with high frequency. These artifacts reveal crucial details about the raw XML data structure and the limitations of the initial cleaning process.

- **XML/HTML Entities:**
  - quot (335,823,940): The high frequency, is probably the result of the XML/HTML entity " (for a quotation mark) being processed after the surrounding punctuation was stripped.
  - amp (115,562): This should also corresponds to the entity & (for an ampersand).
- **Language and Metadata Codes:**
  - ca (459,129), de (1,533,267), en (1,570,869), es (441,618), fr (537,814), ja (227,316), nl (632,698), pl (235,252), pt (464,944), ru (99,223), sv (240,508). These two-letter codes are ISO language codes that are likely originate from metadata within the XML, such as lang="en" attributes or interlanguage links.
- **Wikipedia-Specific Markup:**
  - category (289,543): This directly points to Wikipedia's own categorization system, a fundamental part of its structure.
  - id (8,201,878) and entity (31,291,715): These are almost certainly derived from common XML tags or attributes within the Wikipedia data structure.

- **Justification:** The presence of these artifacts is incredibly valuable. It shows that while the mapper's regular expression (<[^>]+>) successfully removed the tags themselves, it did not handle the content *within* certain tags or special character entities.

4. The Long Tail Phenomenon

The output file also clearly demonstrates the "long tail" phenomenon, a statistical property common in large datasets and natural language.

- **Explanation:** For every high-frequency word like after (606,881), there are thousands of words that appear only a handful of times, or even just once.
    - aah (1)
    - abandoning (1)
    - abnormality (6)
    - abruptly (1)
- **Justification:** This distribution is characteristic of natural language. A small number of words are used very frequently, while most of the vocabulary is used rarely. The ability of the MapReduce job to capture and correctly count these rare words, without being overwhelmed by the high-frequency ones.

5.0 Source Code Access

GitHub Repository: https://github.com/jingjayy/IST3134-MAPREDUCE-PROJECT

6.0 Individual Reflection

6.1 Jing Jay

I was primarily responsible for all coding tasks and managing the GitHub repository. My role involved implementing the MapReduce word count in both Java and Python (Hadoop Streaming), ensuring that the jobs ran correctly on Hadoop, and handling debugging whenever issues arose such as instance space being full due to MapReduce as well as Random Access Memory (RAM) not enough. I also uploaded all source codes and outputs to GitHub, organizing the repository for clear version control and easy team access. This mirrored a professional workflow, where code organization and reproducibility are essential.

From a technical perspective, I gained hands-on experience with distributed data processing. Writing the Java MapReduce job exposed me to Hadoop's native libraries, type safety, and the importance of structuring Mappers, Reducers, and Drivers properly for large-scale computation. Meanwhile, implementing the Python streaming version taught me about the trade-off between simplicity and performance, as Python's ease of development came with slower execution due to inter-process communication.

Analysing the final word-frequency outputs was very satisfying. Correct alphabetical counts of frequent words like "the" and "and" confirmed our pipeline's success, while unexpected tokens such as "amp" and "quot" revealed challenges in data cleaning for semi-structured XML datasets. This connection between my coding work and meaningful results gave me a stronger appreciation of how distributed systems turn raw, massive data into actionable insights.

Overall, this assignment enhanced my coding, problem-solving, and version control skills, while giving me practical confidence in designing and executing big data solutions that are both technically sound and professionally managed.

6.2 Vicky

My primary contribution to this project was in analysing the MapReduce outputs and preparing the report documentation. After Jing Jay implemented the Java and Python MapReduce jobs. I then reviewed the raw word-frequency for the python results and look at the strengths and limitations of the python approach.

I've also learned how to interpret the outputs of a distributed job to confirm that the pipeline worked correctly. For example, observing high-frequency stop words such as "the" and "and" validated that tokenization and counting were successful, while unusual tokens like "amp" and "quot" revealed the presence of HTML entities in the dataset. This analysis has given me a stronger appreciation of the importance of data cleaning in large-scale text processing.

In addition, I was also responsible for structuring the report, ensuring that the MapReduce explanations, comparisons, and output analyses were clearly presented and linked to our findings. I've also assisted in proofreading the scripts and confirming that the GitHub repository was properly organized for submission.

Through this assignment, I learned the value of collaboration in big data projects, where technical coding and careful reporting work together to produce a complete solution. I am now more confident in understanding big data analytics pipelines work and how to document them effectively for both academic and professional contexts.

6.3 Wei Chen

In this project, my primary tasks involved conducting background research, organizing documentation, and assisting in verifying the output. While Jing Jay was responsible for the coding and Vicky handled the output analysis, i contributed in reviewing resources on Hadoop and MapReduce to ensure our write-up was accurate and properly referenced.

I gained practical insights into how MapReduce works by analysing the output and comparing it with our dataset. For instance, I checked that the words were correctly sorted alphabetically, and their frequencies were accurately aggregated through the Mapper and Reducer phases. I also played a role in spotting irrelevant data, such as language tags and XML entities. This emphasizes on the importance of cleaning unstructured data for meaningful analysis.

Additionally, I focused on refining the final report, making sure that the problem statement, dataset explanation, and Java vs Python comparison were clearly presented and professionally written. This process strengthened my skills in technical writing and gave me hands-on experience with GitHub, as I reviewed the repository to ensure all submission materials were complete.

Overall, this project enhanced my abilities in collaboration, critical thinking, and research. It also gave me a clearer understanding of how coding, analysis, and documentatio integrate in a big data analytics workflow.

6.0 References

[1] P. Pedamkar, "MapReduce Word Count," EDUCBA, Feb. 28, 2023. [Online]. Available: https://www.educba.com/mapreduce-word-count/. [Accessed: Aug. 2, 2025]

[2] K. K. Nambiar, "Hadoop MapReduce: Word Count Task Explanation," Medium, Jan. 29, 2025. [Online]. Available: https://medium.com/@krishnendhuaredath/hadoop-mapreduce-word-count-task-explanation-db07fda70692. [Accessed: Aug. 2, 2025]

[3] J. Mlącki, "How does MapReduce work for Big Data? All you need to know," DS Stream, May 7, 2025. [Online]. Available: https://www.dsstream.com/post/how-does-mapreduce-work-for-big-data-all-you-need-to-know. [Accessed: Aug. 2, 2025]

[4] M. Ding, L. Zheng, Y. Lu, L. Li, S. Guo, and M. Guo, "More convenient more overhead: The performance evaluation of Hadoop streaming," in Proc. 2011 Int. Conf. on Research in Adaptive and Convergent Systems (RACS'11), Miami, FL, USA, Nov. 2–5, 2011, pp. 307–313. doi: 10.1145/2103380.2103444. [Online]. Available: https://www.researchgate.net/publication/239761247_More_convenient_more_overhead_The_performance_evaluation_of_Hadoop_streaming [Accessed: Aug. 2, 2025]

[5] S. P. R., "Hadoop MapReduce: Scalable Data Processing Framework," Acceldata Blog, Feb. 5, 2025. [Online]. Available: https://www.acceldata.io/blog/hadoop-mapreduce-for-big-data-success-real-world-use-cases-and-solutions. [Accessed: Aug. 2, 2025]

[6] Statisticseasily, "What is word frequency analysis?," [Online]. Available: https://statisticseasily.com/glossario/what-is-word-frequency-analysis/. [Accessed: Aug. 2, 2025]

[7] J. Jay, V. Chan, and W. Chen, "MapReduce Word Frequency Project (GitHub Repository)," GitHub, 2025. [Online]. Available: https://github.com/YourUsername/MapReduce-WikiWordCount. [Accessed: Aug. 3, 2025]

[8] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] T. White, *Hadoop: The Definitive Guide*, 4th ed. Sebastopol, CA, USA: O'Reilly Media, 2015.

7.0 Appendix

The document specifies the dataset used, and the GitHub repository also links to it. The dataset itself is not hosted directly on GitHub due to its multi-gigabyte size, but rather an external link is provided

Dataset Name: Wikidata XML Dump (April 2025 release)

Direct Download URL:

https://dumps.wikimedia.org/wikidatawiki/20250420/wikidatawiki-20250420-pages-articles-multistream7.xml-p7552572p7838096.bz2

WordCountDriver.java:

```
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.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;


public class WordCountDriver {

    public static void main(String[] args) throws Exception {

        if (args.length != 2) {

            System.err.println("Usage: WordCountDriver <input path> <output path>");
```

```
            System.exit(-1);
        }

        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "Word Count");

        job.setJarByClass(WordCountDriver.class);
        job.setMapperClass(WordCountMapper.class);
        job.setCombinerClass(WordCountReducer.class); // Optional: for local aggregation
        job.setReducerClass(WordCountReducer.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);
    }
}
```

WordCountMapper.java:

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

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class WordCountMapper extends Mapper<Object, Text, Text, IntWritable> {

   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();

   public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
      String cleaned = value.toString().toLowerCase().replaceAll("[^a-z]+", " ");
      StringTokenizer tokenizer = new StringTokenizer(cleaned);
      while (tokenizer.hasMoreTokens()) {
         word.set(tokenizer.nextToken());
         context.write(word, one);
      }
   }
}
```

WordCountReducer.java:

```
import java.io.IOException;


import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Reducer;


public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {


   public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {

      int sum = 0;

      for (IntWritable val : values) {

         sum += val.get();

      }

      context.write(key, new IntWritable(sum));

   }

}
```

mapper.py:

```
import sys
import re

# Load valid words
with open("words.txt") as f:
    english_words = set(word.strip() for word in f)

for line in sys.stdin:
    # Remove XML tags
    line = re.sub(r'<[^>]+>', '', line)
    words = re.findall(r'\b[a-z]{2,}\b', line.lower())
    for word in words:
        if word in english_words:
            print(f"{word}\t1")
```

reducer.py:

```
#!/usr/bin/env python3
import sys

current_word = None
```

```
total = 0


for line in sys.stdin:
    word, count = line.strip().split('\t')
    count = int(count)


    if word == current_word:
        total += count
    else:
        if current_word:
            print(f"{current_word}\t{total}")
        current_word = word
        total = count


if current_word:
    print(f"{current_word}\t{total}")
```