

CS6220 Big Data System and Analytics

Assignment 1

Sathvik Karatattu Padmanabha

September 2024

Student Session: cs6220-A
GT ID: 904032361

1 Problem Chosen

Option 1.1

Suitable for students who are the beginner of Hadoop/Spark MapReduce

- Install HDFS and Hadoop MapReduce on your laptop.
- Run the word count map-reduce program, and report the runtime for three different sizes of datasets, two of the datasets should be larger and need multiple reducers and all three will be larger and need multiple mappers.
- Using MapReduce to solve another problem. You may choose one of the following.

o Consider a dataset of 100 files, print the top 30 words occurring in the most files. Scaling your solution to 1000 files, 5000 and optionally 10,000 files. Make sure to meet deliverable requirements (e) and (f).

You are encouraged to use excel file to generate your runtime statistics plot or organize the performance measurement data in a tabular format.

You are encouraged to learn by observing the runtime performance of Hadoop / Spark MapReduce program through different ways of programming the same problem and show their impact on the runtime performance of the MapReduce job.

2 Introduction and Readme

The source code is present in the zip file. The source code is written in Java and is organized as a maven project. To modify, compile and package, java-8 and maven is required and an IDE such as eclipse is recommended.

There are 2 different approaches taken to solve the word count and print the top 30 words. Both the approaches are coded and are present in 'approach1' and 'approach2' directories. The approach 1 is simple word count and sorting

by decreasing order of frequency. Approach 2 uses the inverted index method followed by counting and sorting. Details will be mentioned later.

The excel sheet “Big Data Assignment Final.xlsx” contains additional runtime statistics and plots for the various experiments and tasks. I have put the analysis in this report, but interesting extra statistics are present in this excel sheet.

The terminal log outputs for all experiments are present in the “terminal_log_files” folder.

The screenshots for certain tasks are present in the ‘screenshots’ directory and are also included in this pdf.

The 2 different jars for 2 different methods are also included. The instructions to run are provided later.

Trivial examples for input and output for the 2 different methods are present in the ‘Examples’ file. It shows the output for each mapper and reducer phases.

The outputs (word counts) for all the execution are present in the ‘assignment_1_output_final’ directory.

Github repo containing my code:

`"https://github.com/SathvikKP/hadoop_wordcount_final"`

The repository is private, so you need to request me to allow access.

3 Dataset Used

I used plain text files from project gutenber. It consists of large number of english articles (<https://www.gutenberg.org/>).

To get a large number of files, I used a helper script to automate downloads (auxillary_scripts/downloader.py)

I split the downloaded files into different directories using bash scripts (auxillary_scripts/*.sh)

Then, I moved them into the hdfs file system using hdfs commands

```
hdfs dfs -put books_v1 /input/project_gutenberg/books_v1.
```

Figure 1 shows a screenshot of the directory structure after I divided the files using bash scripts and the output of a randomly chosen file.

```
sathkp@sathvik:~/BigData/Programs/project_gutenberg/parts$ ls -d */
books_v1/  books_v1_189/  books_v1_388/  books_v1_combined/  books_v2/  books_v4/  books_v6/
books_v1_10/  books_v1_10_combined/  books_v1_500/  books_v1_split/  books_v3/  books_v5/
sathkp@sathvik:~/BigData/Programs/project_gutenberg/parts$ ls -lt books_v1 | head -10
10706.txt
10701.txt
10702.txt
10704.txt
10705.txt
10713-8.txt
11329-8.txt
11330-8.txt
11331-8.txt
11332-8.txt
sathkp@sathvik:~/BigData/Programs/project_gutenberg/parts$ head -n 20 books_v1/10706.txt
The Project Gutenberg eBook, The History of Rome (Volumes 1-5), by Theodor
Mommsen, Translated by William Purdie Dickson

This eBook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever. You may copy it, give it away or
re-use it under the terms of the Project Gutenberg License included
with this eBook or online at www.gutenberg.org

Title: The History of Rome (Volumes 1-5)
Author: Theodor Mommsen
Release Date: March 16, 2005 [eBook #10706]
Language: English
sathkp@sathvik:~/BigData/Programs/project_gutenberg/parts$ |
```

Figure 1: Dataset directory and contents of a random sample file.

4 Source Code and Deliverables

4.1 Source Code

The source code is present in the zip file. I have also put the main source code logic at the end of this document. I have also uploaded the source code to my private Github repository. Please request me if you need access to the github repo.

https://github.com/SathvikKP/hadoop_wordcount_final.

4.2 Deliverables

WordCountAndSort_Final.jar and WordCountAndSort_InvertedFinal.jar are the two jars for the two approaches.

5 Experiment Details

5.1 Component 1

Component 1: • Install HDFS and Hadoop MapReduce on your laptop.

I have referred the official documentation for installation and setup. I used Windows Subsystem for Linux and installed Ubuntu 22.04.4 LTS.

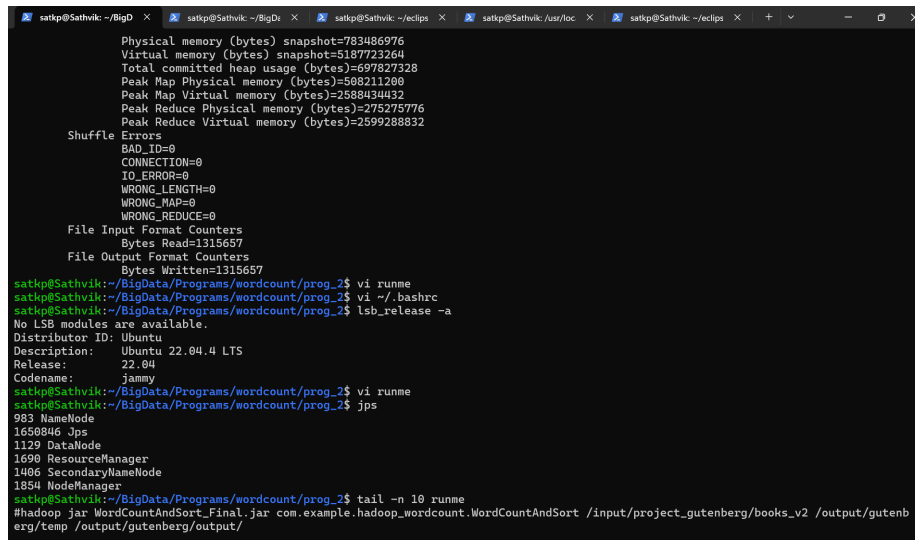
To run successfully, I need to properly set the environment variables and Hadoop

settings such as core-site.xml, mapred.xml. I referred the official Hadoop documentation for instructions.

<https://hadoop.apache.org/releases.html>

<https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html>

<https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html>



```
Physical memory (bytes) snapshot=783486976
Virtual memory (bytes) snapshot=5187723264
Total committed heap usage (bytes)=697827328
Peak Map Physical memory (bytes)=588211200
Peak Map Virtual memory (bytes)=2588434432
Peak Reduce Physical memory (bytes)=275275776
Peak Reduce Virtual memory (bytes)=2599288832

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=1315657
File Output Format Counters
  Bytes Written=1315657

satkp@Sathvik: ~/BigData/Programs/wordcount/prog_2$ vi runme
satkp@Sathvik:~/BigData/Programs/wordcount/prog_2$ vi ~/.bashrc
satkp@Sathvik:~/BigData/Programs/wordcount/prog_2$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description: Ubuntu 22.04.4 LTS
Release: 22.04
Codename: jammy
satkp@Sathvik:~/BigData/Programs/wordcount/prog_2$ vi runme
satkp@Sathvik:~/BigData/Programs/wordcount/prog_2$ jps
983 NameNode
1650846 Jps
1129 DataNode
1690 ResourceManager
1406 SecondaryNameNode
1854 NodeManager
satkp@Sathvik:~/BigData/Programs/wordcount/prog_2$ tail -n 10 runme
#hadoop jar WordCountAndSort_Final.jar com.example.hadoop_wordcount.WordCountAndSort /input/project_gutenberg/books_v2 /output/gutenb
erg/temp /output/gutenberg/output/
```

Figure 2: Execution of word count map-reduce program, OS info and Hadoop processes.

5.2 Component 2

- Run the word count map-reduce program and report the runtime for three different sizes of datasets, two of the datasets should be larger and need multiple reducers and all three will be larger and need multiple mappers.

I used the eclipse IDE and used the maven build tool for compiling and packaging.

To run the program, use the following command templates. For the first approach, there are 3 arguments – input, temp and output.

```
hadoop jar WordCountAndSort_Final.jar com.example.hadoop_wordcount.
WordCountAndSort /input/project_gutenberg/books_v1
/output/gutenberg/temp /output/gutenberg/output/
```

```

Combine output records=0
Reduce input groups=4281
Reduce shuffle bytes=9727541
Reduce input records=604940
Reduce output records=604940
Spilled Records=1209880
Shuffled Maps =1
Failed Shuffles=0
Merged Map outputs=1
GC time elapsed (ms)=367
CPU time spent (ms)=5060
Physical memory (bytes) snapshot=1008050176
Virtual memory (bytes) snapshot=519478864
Total committed heap usage (bytes)=866648064
Peak Map Physical memory (bytes)=616587264
Peak Map Virtual memory (bytes)=2592264192
Peak Reduce Physical memory (bytes)=391462912
Peak Reduce Virtual memory (bytes)=2602524672

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=7416040
File Output Format Counters
Bytes Written=7416040
satkp@Sathvik:~/BigData/Programs/wordcount/prog_2$

```

Figure 3: Successful execution of Hadoop MapReduce word count program.

ID	User	Name	Application Type	Application Tags	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated C	VC
application_1725454718410_0024	satkp	sorting	MAPREDUCE		root.default	0	Wed Sep 4 19:23:12 -0400 2024	Wed Sep 4 19:23:13 -0400 2024	Wed Sep 4 19:23:28 -0400 2024	FINISHED	SUCCEEDED	N/A	N/A	
application_1725454718410_0023	satkp	frequency count	MAPREDUCE		root.default	0	Wed Sep 4 19:22:49 -0400 2024	Wed Sep 4 19:22:50 -0400 2024	Wed Sep 4 19:23:06 -0400 2024	FINISHED	SUCCEEDED	N/A	N/A	
application_1725454718410_0022	satkp	inverted index	MAPREDUCE		root.default	0	Wed Sep 4 19:21:21 -0400 2024	Wed Sep 4 19:21:22 -0400 2024	Wed Sep 4 19:22:43 -0400 2024	FINISHED	SUCCEEDED	N/A	N/A	
application_1725454718410_0021	satkp	sorting	MAPREDUCE		root.default	0	Wed Sep 4 19:15:48 -0400 2024	Wed Sep 4 19:15:50 -0400 2024	Wed Sep 4 19:16:06 -0400 2024	FINISHED	SUCCEEDED	N/A	N/A	
application_1725454718410_0020	satkp	frequency count	MAPREDUCE		root.default	0	Wed Sep 4 19:15:24 -0400 2024	Wed Sep 4 19:15:26 -0400 2024	Wed Sep 4 19:15:43 -0400 2024	FINISHED	SUCCEEDED	N/A	N/A	
application_1725454718410_0019	satkp	inverted index	MAPREDUCE		root.default	0	Wed Sep 4 19:11:30 -0400 2024	Wed Sep 4 19:11:30 -0400 2024	Wed Sep 4 19:15:19 -0400 2024	FINISHED	SUCCEEDED	N/A	N/A	
application_1725454718410_0018	satkp	sorting	MAPREDUCE		root.default	0	Wed Sep 4 18:55:53 -0400 2024	Wed Sep 4 18:55:55 -0400 2024	Wed Sep 4 18:56:11 -0400 2024	FINISHED	SUCCEEDED	N/A	N/A	
application_1725454718410_0017	satkp	frequency count	MAPREDUCE		root.default	0	Wed Sep 4 18:55:27 -0400 2024	Wed Sep 4 18:55:29 -0400 2024	Wed Sep 4 18:55:47 -0400 2024	FINISHED	SUCCEEDED	N/A	N/A	

Figure 4: Resource Manager UI showcasing successful tasks for various map-reduce programs.

On running the program, the word counts are present in the hdfs output location ('/output/gutenberg/output' in this case). I have chosen 3 different

configurations for relatively large number of files. The experiment details are shown in the next section.

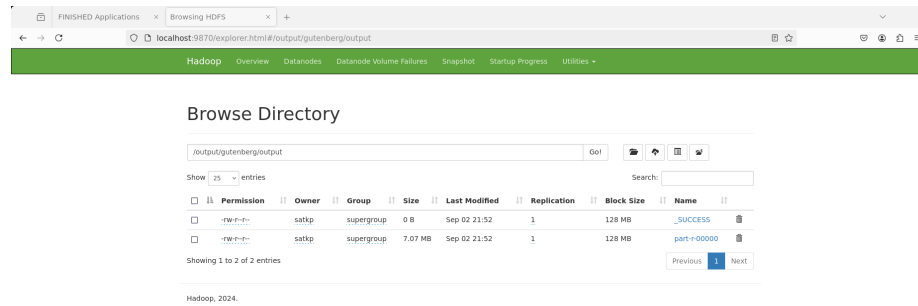


Figure 5: The output directory, part-r-00000 contains the word counts.

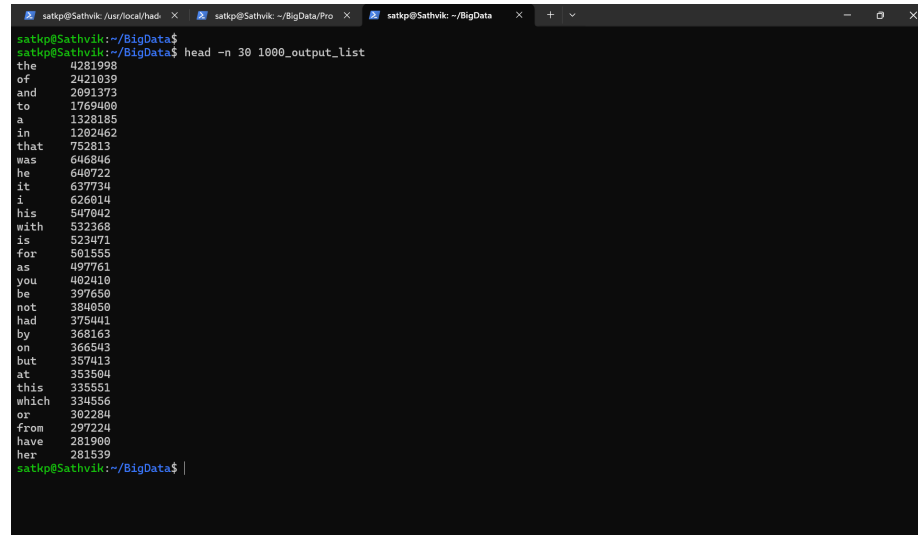


Figure 6: Printing of top 30 words (for 1000 randomly chosen files) in descending order.

6 Experiments on Word-Count program to meet deliverable (e)

6.1 Experiment 1: Vary the size of the input dataset

I have increased the number of files (increasing size of dataset) and recorded the execution time for the word count problem. Figure 7 shows the statistics.

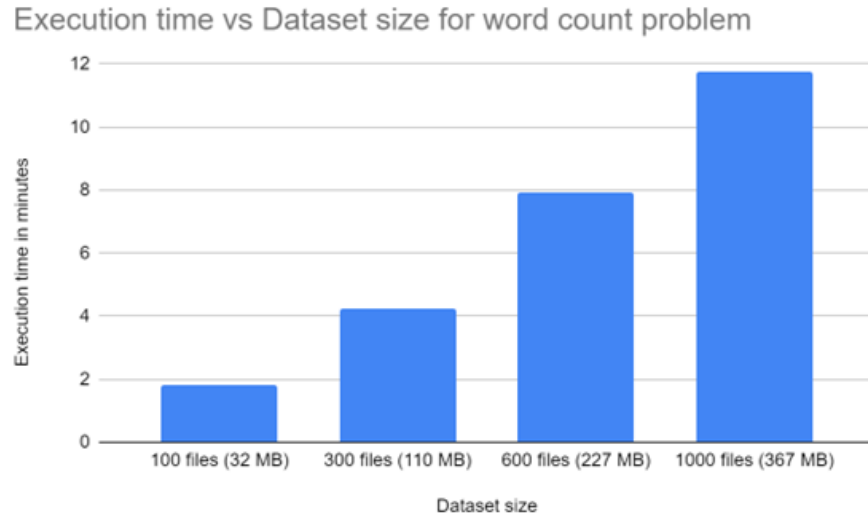


Figure 7: Execution time vs number of files for word count problem.

Analysis: On increasing dataset size, I observed that the execution time increased almost linearly. This is the expected result given all other parameters are kept constant.

6.2 Experiment 2: Vary the number of mappers for constant dataset size.

To vary the number of mappers, we can set the block size in Hadoop mapred settings. However this was inconsistent and hence, I chose to keep the same dataset size but to vary the file count. I first ran the experiment for 1000 files. Next, I split each file into 2 halves doubling the number of files and ran the experiment. Finally, I combined every 2 files to halve the number of files and re-ran the experiment. The Figure 8 shows the statistics for this experiment.

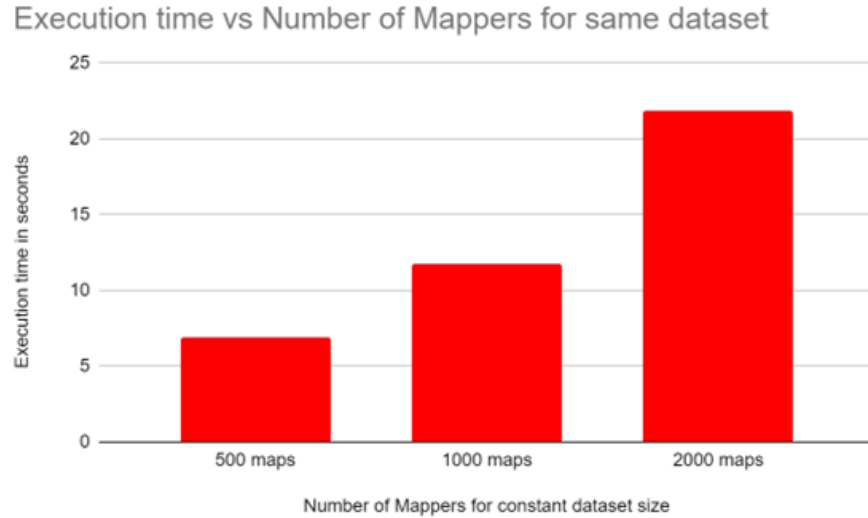


Figure 8: Execution time vs Number of mappers

Analysis: On increasing mappers, the execution time increased. This is not the expected result as increasing number of mappers should ideally reduce execution time. However, as I executed this program on my laptop with very limited resources and just a single node, there is a substantial overhead due to resource contention. Each mapper task needs to initialize, process the split, and clean up. This slows down the whole process due to limited resources and hence this output is ideally not expected.

6.3 Experiment 3: Vary the number of reducers for constant dataset size

We can vary the number of reducers by configuring via code or by command line arguments. For a constant dataset of 100 files, I increased the number of reducers from 1 to 3 to 5 and recorded the execution results. Figure 9 shows the statistics.

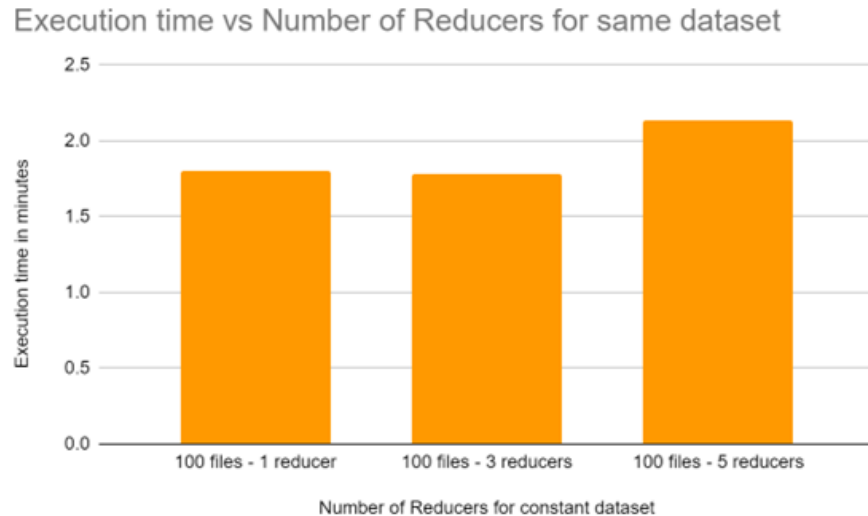


Figure 9: Execution time vs Number of reducers.

Analysis: The execution time reduced barely for 3 reducers and increased for 5 reducers. This is not the expected output as ideally, increasing number of reducers should decrease the runtime. Again, the same reason for experiment 2 applies here. I executed this program on my laptop with very limited resources and just a single node, there is a substantial overhead due to resource contention. This slows down the whole process due to limited resources and hence the output is ideally not expected.

For all further experiments, I have kept the reducer size as 3 as it gave the best results.

7 Component 3 - Most Popular Words Problem

- Using MapReduce to solve another problem. You may choose one of the following.
 - o Consider a dataset of 100 files, print the top 30 words occurring in the most files. Scaling your solution to 1000 files, 5000 and optionally 10,000 files. Make sure to meet deliverable requirements (e) and (f).

To achieve this task, we can use 2 map-reduce jobs to achieve this task. The first job will count the number of words, and the second job will sort the number of words in the decreasing order of frequency. By this way, we can get the top 30 words occurring in the files. We must use 1 reducer in the second job so that the final output will be present in a single file. An example is present in the 'Examples.txt' text file showcasing the input and output at each stage. The

‘WordCountAndSort_Final.jar’ is the deliverable for this approach. To run the program, use the following command templates. For the first approach, there are 3 arguments – input, temp and output.

```
hadoop jar WordCountAndSort_Final.jar com.example.hadoop_wordcount.  
WordCountAndSort /input/project_gutenberg/books_v1  
/output/gutenberg/temp /output/gutenberg/output/
```

The outputs for all the executions for all experiments are present in ‘terminal_log_files’ directory.

7.1 Experiment 4: Execution time for Most Popular Words

I have executed this program by varying the number of files thereby increasing the dataset size. The number of mappers vary for each experiment as Hadoop framework decides the number of mappers optimally based on number of files. The number of reducers are kept to 3 for the first job and 1 for the second job (to get results in a single file). Figure 10 shows the statistics for 100 files, 300 files, 600 files, 1000 files and 5000 files.

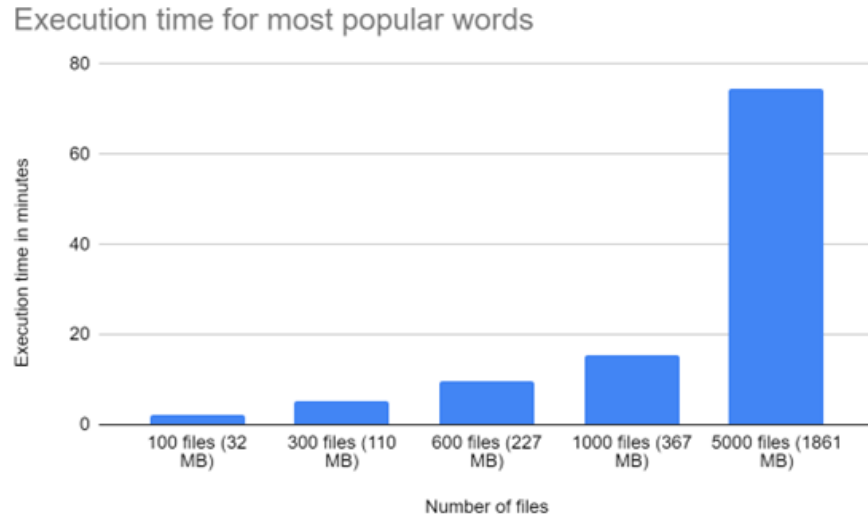


Figure 10: Execution time vs Number of files.

Analysis: As expected, the execution time increases almost linearly as we increase the number of files / dataset size. It is ‘almost’ linear and can sometimes vary as there can be possibilities of execution overhead due to resource contention and constraints, node availability and various other environment related factors.

7.2 Experiment 5: Execution time for Most Popular Words for different dataset with similar total size

To meet the requirement (e), I have considered one more option. I have kept the number of files constant (1000 files), but the files are completely different. I took 3000 files and split them into 3 parts, then ran the Most popular words program for the 3 parts separately. Figure 11 shows the statistics.

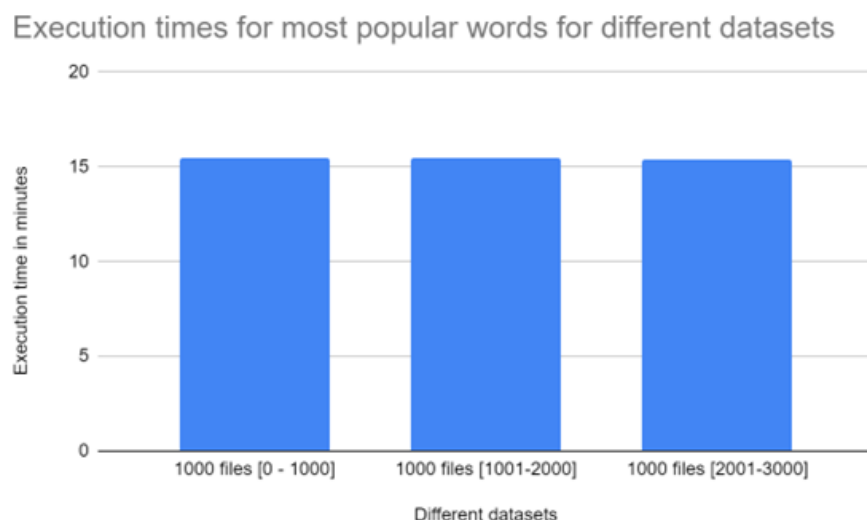


Figure 11: Execution time vs Different datasets.

Analysis: The runtime is almost the same for all the cases. This is because the file sizes are also same, and all other configurations are kept the same. This is the expected output.

8 Most Popular Words using Inverted Index Method

To meet the requirement (f), I have solved the Most popular words program using a different approach (approach 2). The approach used is the inverted index method. In this method, we use 3 different MapReduce jobs to solve the problem.

1. First job: For each word, output the files/document in which it is present.
2. Second job: For each word, count the number of times it occurs in all documents
3. Third job: Sort the words in decreasing order of frequency.

An example is present in the 'Examples.txt' text file showcasing the input and

output at each stage. I have used 3 reducers in the first 2 jobs and 1 reducer in the final job so that the final output will be present in a single file. The number of mappers is equal to the number of input files.

The ‘WordCountAndSort_InvertedFinal.jar’ is the deliverable for this approach. To run the program, use the following command templates. For this approach (approach2), there are 4 arguments, input, temp1, temp2 and output.

```
hadoop jar WordCountAndSort_InvertedFinal.jar
com.example.inverted_method.WordCountAndSortInverted
/input/project_gutenberg/books_v1_100 /output/gutenberg/temp1
/output/gutenberg/temp2 /output/gutenberg/output/
```

The outputs for all the executions for all experiments are present in ‘terminal_log_files’ directory.

8.1 Experiment 6: Execution Time for Most popular words using Inverted index method

I have run the Most popular words using the inverted index method for various number of files (different dataset size) and compared their execution times. Figure 12 shows the statistics.

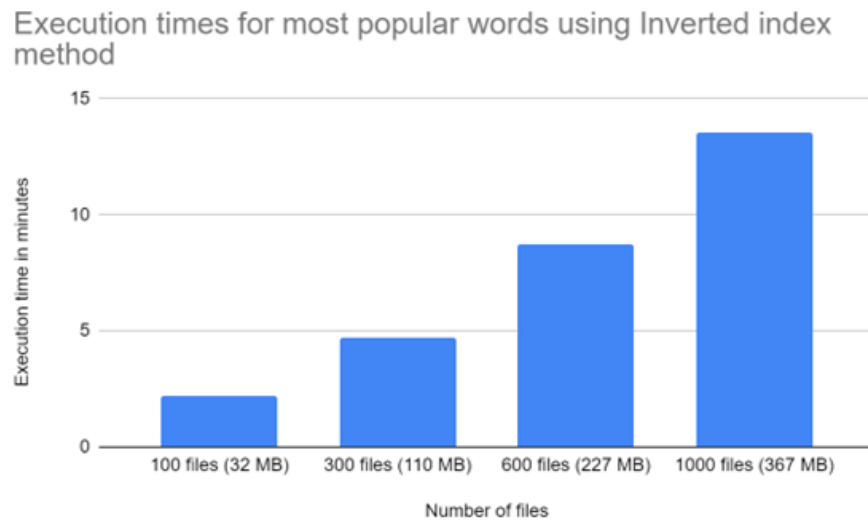


Figure 12: Execution time for most popular words using Inverted index method vs number of files

Analysis: The execution time increases almost linearly as we increase the number of files (dataset size). This is the expected output. This approach is useful if additional information (e.g., document frequency) is needed.

8.2 Experiment 7: Comparison of two methods

I have performed a comparison of Approach 1 (normal word count) and Approach 2 (inverted index method). The environment settings and the dataset settings are the same for both the experiments. Figure 13 shows the statistics.

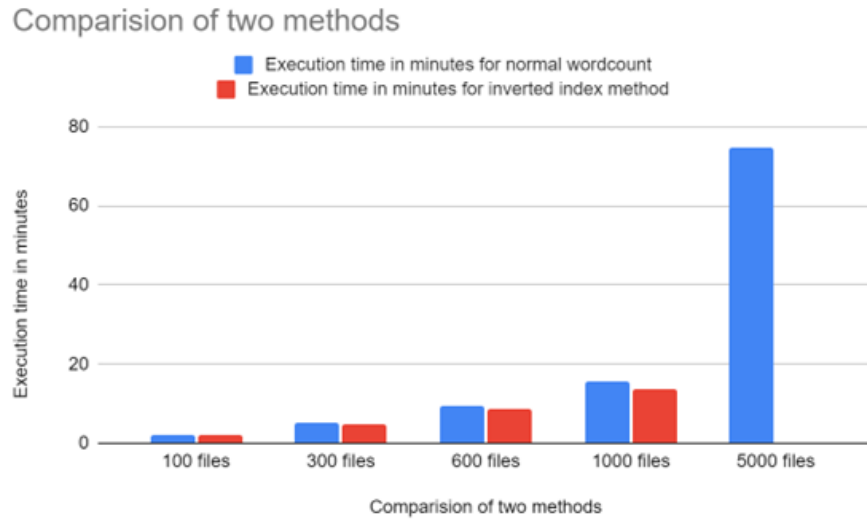


Figure 13: Comparison of two methods.

Analysis: We can see that the inverted index method is slightly faster than the normal word count method. However, my system failed to run the inverted index method for 5000 files due to resource limitations (out-of-memory error). Hence, the inverted index method is faster but takes more resources (memory). Hence, depending on the resource availability and dataset size, we can choose the appropriate approach. If we have less resources, we can choose the normal word count method. If we have plentiful resources, we can choose the inverted index method. If the dataset size is small, we can safely choose the inverted index method.

The Inverted index is faster for large datasets as sometimes, a single word can appear many times in a document which is emitted in the first phase, which simplifies counting in the second phase. In the normal method, this is not possible and hence, it may take a longer time for large data sizes.

For small datasizes, the normal wordcount is faster due to less number of MapReduce jobs and simpler implementation.

9 Conclusion

Hadoop MapReduce can be used to solve tasks in parallel for large amounts of data. By solving this assignment, I learnt a lot about the framework, installing it and using it to solve the Most Popular Words problem using 2 different approaches and drew interesting analytical conclusions.

I have observed that increasing the size of dataset will increase the execution time. Increasing the number of files will also have a similar impact if the total size of all files also increases.

I could not successfully replicate the number of mappers vs execution time scenario as my laptop could not handle such a scenario, but I learnt about it by going through official documentation and resources online.

I also observed that increasing reducers will slightly reduce execution time, but we need to be careful about the task at hand. For instance, in the most popular words problem, we need the final output in a single file and hence, we only need 1 reducer at the last stage.

I observed that to solve the most popular words problem, the Inverted Index approach will take a shorter execution time for a large dataset when compared to the Normal word count approach, but the Inverted Index method will consume more resources (memory), and my laptop was unable to handle it for large dataset sizes (5000 files / 1.8 GB).

The Inverted Index method is useful if additional information (e.g., document frequency) is needed.

10 References

1. CS6220 Big Data Systems and Analytics Class lecture slides.
2. <https://hadoop.apache.org/releases.html>
3. <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html>
4. <https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html>
5. <https://www.gutenberg.org/>
6. <https://cognitivedemons.wordpress.com/2017/07/10/downloading-all-english-books-from-gutenberg-org-with-python/>.
7. Numerous Stackoverflow posts for Hadoop environment issues

11 Source Code Snippets

The directory structure is present in the zip file.

11.1 Approach 1

WordCountAndSort.java

```
package com.example.hadoop_wordcount;

import java.io.IOException;
import java.util.StringTokenizer;
import java.util.regex.Pattern;
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.Reducer.Context;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class WordCountAndSort {

    public static class TokenizerMapper
        extends Mapper<Object, Text, Text, IntWritable> {

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

        // Regular expression pattern to match words and numbers
        private static final Pattern WORD_BOUNDARY = Pattern.compile("[^a-zA-Z0-9]+");

        public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                String token = itr.nextToken();
                // Remove punctuation by replacing non-alphanumeric characters with an empty
                // string
                token = WORD_BOUNDARY.matcher(token).replaceAll("");
                // word.set(itr.nextToken());
                // context.write(word, one);
            }
        }
    }
}
```

```

        if (!token.isEmpty()) {
            word.set(token.toLowerCase()); // Convert to lowercase for case-insensitive counting
            context.write(word, one);
        }
    }
}

public static class IntSumReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();

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

public static class SortMapper
    extends Mapper<Object, Text, IntWritable, Text> {
    private IntWritable count = new IntWritable();
    private Text word = new Text();

    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        String[] parts = value.toString().split("\\t");
        if (parts.length == 2) {
            word.set(parts[0]);
            count.set(Integer.parseInt(parts[1]));
            context.write(count, word);
        }
    }
}

public static class SortReducer
    extends Reducer<IntWritable, Text, Text, IntWritable> {
    public void reduce(IntWritable key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
        for (Text val : values) {
            context.write(val, key);
        }
    }
}

```



```
}
```

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

    System.out.println("\nargs[0] = " + args[0]+ " , args[1] = " +args[1]+ " , args[2] = "+args[2]);

    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCountAndSort.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class); // sort on key - alphabetical
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    job.setNumReduceTasks(3);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    // System.exit(job.waitForCompletion(true) ? 0 : 1);
    if (!job.waitForCompletion(true)) {
        System.exit(1);
    }

    Job sortJob = Job.getInstance(conf, "soft by frequency");
    sortJob.setJarByClass(WordCountAndSort.class);
    sortJob.setMapperClass(SortMapper.class);
    sortJob.setReducerClass(SortReducer.class);
    // Sort in ascending order by default
    //sortJob.setSortComparatorClass(IntWritable.Comparator.class);
    // Set custom comparator for sorting keys in descending order
        sortJob.setSortComparatorClass(DescendingIntWritableComparator.class);
    sortJob.setOutputKeyClass(IntWritable.class);
    sortJob.setOutputValueClass(Text.class);
    sortJob.setNumReduceTasks(1);
    FileInputFormat.addInputPath(sortJob, new Path(args[1]));
    FileOutputFormat.setOutputPath(sortJob, new Path(args[2]));
    // System.exit(sortJob.waitForCompletion(true) ? 0 : 1);
    if (!sortJob.waitForCompletion(true)) {
        System.exit(1);
    }

}
}
```

DescendingIntWritableComparator.java

```
package com.example.hadoop_wordcount;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;
import java.io.DataInputStream;
import java.io.ByteArrayInputStream;
import java.io.IOException;

public class DescendingIntWritableComparator extends WritableComparator {

    public DescendingIntWritableComparator() {
        super(IntWritable.class);
    }

    @Override
    public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
        IntWritable int1 = new IntWritable();
        IntWritable int2 = new IntWritable();
        try {
            int1.readFields(new DataInputStream(new ByteArrayInputStream(b1, s1, l1)));
            int2.readFields(new DataInputStream(new ByteArrayInputStream(b2, s2, l2)));
        } catch (IOException e) {
            e.printStackTrace();
        }

        return -int1.compareTo(int2); // Reverse the comparison for descending order
    }
}
```

11.2 Approach 2

WordCountAndSortInverted.java

```
package com.example.inverted_method;

import java.io.IOException;
import java.util.HashSet;
import java.util.Map;
import java.util.HashMap;
import java.util.StringTokenizer;
import java.util.regex.Pattern;
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;
import org.apache.hadoop.io.WritableComparator;

// Main class
public class WordCountAndSortInverted {

    // Phase 1: Inverted Index Mapper
    public static class InvertedIndexMapper extends Mapper<Object, Text, Text, Text> {
        private Text word = new Text();
        private Text documentId = new Text();
        private static final Pattern WORD_BOUNDARY = Pattern.compile("[^a-zA-Z0-9]+");

        @Override
        public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            documentId.set(key.toString());

            while (itr.hasMoreTokens()) {
                String token = itr.nextToken();
                token = WORD_BOUNDARY.matcher(token).replaceAll("");
                if (!token.isEmpty()) {
                    word.set(token.toLowerCase());
                    context.write(word, documentId);
                }
            }
        }
    }

    /*
    // Phase 1: Inverted Index Reducer
    public static class InvertedIndexReducer extends Reducer<Text, Text, Text, Text> {
        @Override
        public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
            HashSet<String> uniqueDocs = new HashSet<>();
            for (Text val : values) {
                uniqueDocs.add(val.toString());
            }
            context.write(key, new Text(String.join(" ", uniqueDocs)));
        }
    }
    */
}

```

```
*/
```

```
// Phase 1: Inverted Index Reducer
```

```
public static class InvertedIndexReducer extends Reducer<Text, Text, Text, Text> {
    private IntWritable result = new IntWritable();

    @Override
    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException {
        // Use a map to keep track of document IDs and their counts
        Map<String, Integer> docCountMap = new HashMap<>();

        for (Text val : values) {
            String docId = val.toString();
            docCountMap.put(docId, docCountMap.getOrDefault(docId, 0) + 1);
        }

        // Build the result string in the format: word doc1 doc1 doc2 doc3 doc3 ...
        StringBuilder result = new StringBuilder();
        for (Map.Entry<String, Integer> entry : docCountMap.entrySet()) {
            String docId = entry.getKey();
            int count = entry.getValue();
            for (int i = 0; i < count; i++) {
                result.append(docId).append(" ");
            }
        }

        context.write(key, new Text(result.toString().trim()));
    }
}
```

```
// Phase 2: Frequency Count Mapper
```

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

    @Override
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        String[] parts = value.toString().split("\\s+");
        if (parts.length > 1) {
            word.set(parts[0]); // The word itself
            int count = parts.length - 1; // Number of document IDs
            context.write(word, new IntWritable(count));
        }
    }
}
```

```

// Phase 2: Frequency Count Reducer
public static class FrequencyCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();

    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        int count = 0;
        for (IntWritable val : values) {
            count += val.get(); // Sum up the counts for each word
        }
        result.set(count);
        context.write(key, result);
    }
}

// Phase 3: Sorting Mapper
public static class SortingMapper extends Mapper<Object, Text, IntWritable, Text> {
    private IntWritable count = new IntWritable();
    private Text word = new Text();

    @Override
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        String[] parts = value.toString().split("\\s+");
        if (parts.length == 2) {
            word.set(parts[0]);
            count.set(Integer.parseInt(parts[1]));
            context.write(count, word);
        }
    }
}

// Custom Comparator for Sorting in Descending Order
public static class IntWritableDecreasingComparator extends WritableComparator {
    public IntWritableDecreasingComparator() {
        super(IntWritable.class, true);
    }

    @Override
    public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
        int int1 = readInt(b1, s1);
        int int2 = readInt(b2, s2);
        return Integer.compare(int2, int1); // Descending order
    }
}

// Phase 3: Sorting Reducer

```

```

public static class SortingReducer extends Reducer<IntWritable, Text, Text, IntWritable> {
    private IntWritable result = new IntWritable();
    private Text word = new Text();

    @Override
    public void reduce(IntWritable key, Iterable<Text> values, Context context) throws IOException {
        for (Text val : values) {
            word.set(val);
            result.set(key.get());
            context.write(word, result);
        }
    }
}

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

    // Phase 1: Inverted Index
    Job job1 = Job.getInstance(conf, "inverted index");
    job1.setJarByClass(WordCountAndSortInverted.class);
    job1.setMapperClass(InvertedIndexMapper.class);
    job1.setReducerClass(InvertedIndexReducer.class);
    job1.setOutputKeyClass(Text.class);
    job1.setOutputValueClass(Text.class);
    job1.setNumReduceTasks(3);
    FileInputFormat.addInputPath(job1, new Path(args[0]));
    FileOutputFormat.setOutputPath(job1, new Path(args[1]));
    job1.waitForCompletion(true);

    // Phase 2: Frequency Count
    Job job2 = Job.getInstance(conf, "frequency count");
    job2.setJarByClass(WordCountAndSortInverted.class);
    job2.setMapperClass(FrequencyCountMapper.class);
    job2.setReducerClass(FrequencyCountReducer.class);
    job2.setOutputKeyClass(Text.class);
    job2.setOutputValueClass(IntWritable.class);
    job2.setNumReduceTasks(3);
    FileInputFormat.addInputPath(job2, new Path(args[1]));
    FileOutputFormat.setOutputPath(job2, new Path(args[2]));
    job2.waitForCompletion(true);

    // Phase 3: Sorting
    Job job3 = Job.getInstance(conf, "sorting");
    job3.setJarByClass(WordCountAndSortInverted.class);
    job3.setMapperClass(SortingMapper.class);
    job3.setReducerClass(SortingReducer.class);

```

```
        job3.setOutputKeyClass(IntWritable.class);
        job3.setOutputValueClass(Text.class);
        job3.setSortComparatorClass(IntWritableDecreasingComparator.class); // Set custom c
        FileInputFormat.addInputPath(job3, new Path(args[2]));
        FileOutputFormat.setOutputPath(job3, new Path(args[3]));
        System.exit(job3.waitForCompletion(true) ? 0 : 1);
    }
}
```