20CYS402 – Distributed Systems & Cloud Computing

Lab 5 - Hadoop Cluster and MapReduce

Name: Chitra Harini **Date:** 22/08/2025

Roll no: CH.EN.U4CYS22010 **Lab - 5**

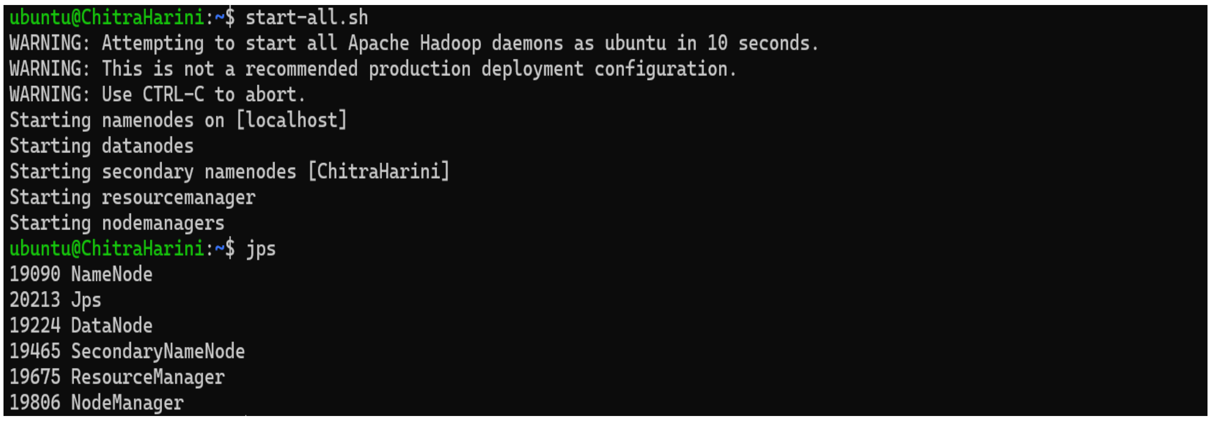
Github Link: [20CYS402-Distributed-Systems-Cloud-Computing/LAB5 at main · Harini-chitra/20CYS402-Distributed-Systems-Cloud-Computing]

Task 1: Hadoop Service Startup and HDFS Interaction

- **Objective**: The primary goal of this task is to start the Hadoop cluster, verify that all its essential background services (daemons) are running correctly, and then perform fundamental file system operations—such as creating a directory and uploading a file—on the Hadoop Distributed File System (HDFS).

Service Startup: Start all essential Hadoop services: NameNode, DataNode, ResourceManager, NodeManager, and SecondaryNameNode. Provide the command line instructions used to start each service.
Service Verification: Use the jps command to list and verify that all core Hadoop services are running. Capture a screenshot of the terminal output showing the running services and describe the function of each service:
• NameNode: Manages the HDFS filesystem namespace.
• DataNode: Stores the actual data in HDFS.
• ResourceManager: Manages resource allocation for MapReduce jobs.
• NodeManager: Manages the execution of containers on worker nodes.
• SecondaryNameNode: Periodically checkpoints the HDFS metadata.

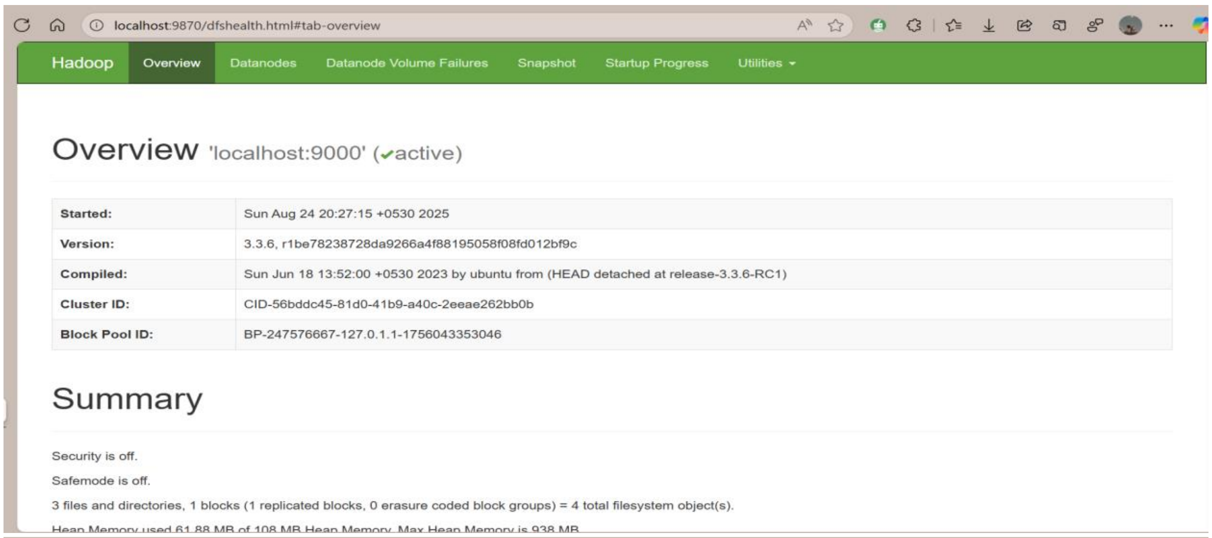
1. *start-all.sh* - This script initiates all the necessary Hadoop daemons for both HDFS and YARN.
2. *jps* - This command lists all running Java processes, which allows you to verify that the Hadoop daemons have started.



```
ubuntu@ChitraHarini:~$ start-all.sh
WARNING: Attempting to start all Apache Hadoop daemons as ubuntu 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 [ChitraHarini]
Starting resourcemanager
Starting nodemanagers
ubuntu@ChitraHarini:~$ jps
19090 NameNode
20213 Jps
19224 DataNode
19465 SecondaryNameNode
19675 ResourceManager
19806 NodeManager
```

Hadoop Service Testing via Browser: Use the following URLs to verify the status of your Hadoop services through a web browser:

1. NameNode Web UI: http://localhost:9870/



Overview 'localhost:9000' (✓active)

Started:	Sun Aug 24 20:27:15 +0530 2025
Version:	3.3.6, r1be78238728da9266a4f88195058f08fd012bf9c
Compiled:	Sun Jun 18 13:52:00 +0530 2023 by ubuntu from (HEAD detached at release-3.3.6-RC1)
Cluster ID:	CID-56bddc45-81d0-41b9-a40c-2eeae262bb0b
Block Pool ID:	BP-247576667-127.0.1.1-1756043353046

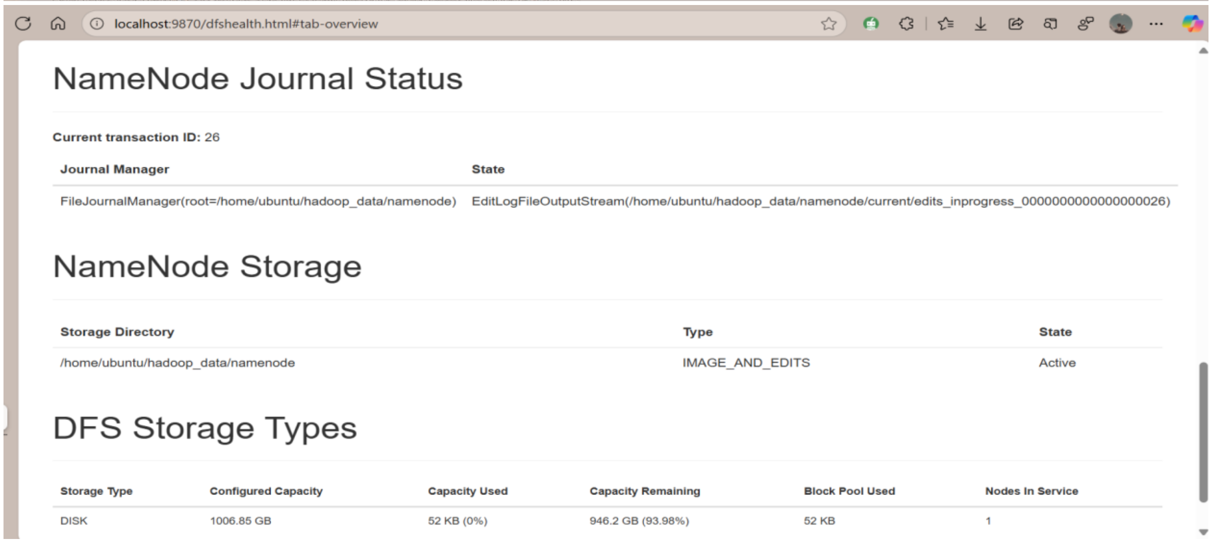
Summary

Security is off.

Safemode is off.

3 files and directories, 1 blocks (1 replicated blocks, 0 erasure coded block groups) = 4 total filesystem object(s).

Heap Memory used 61.88 MB of 108 MB Heap Memory. Max Heap Memory is 938 MB.



NameNode Journal Status

Current transaction ID: 26

Journal Manager	State
FileJournalManager(root=/home/ubuntu/hadoop_data/namenode)	EditLogFileOutputStream(/home/ubuntu/hadoop_data/namenode/current/edits_inprogress_0000000000000000026)

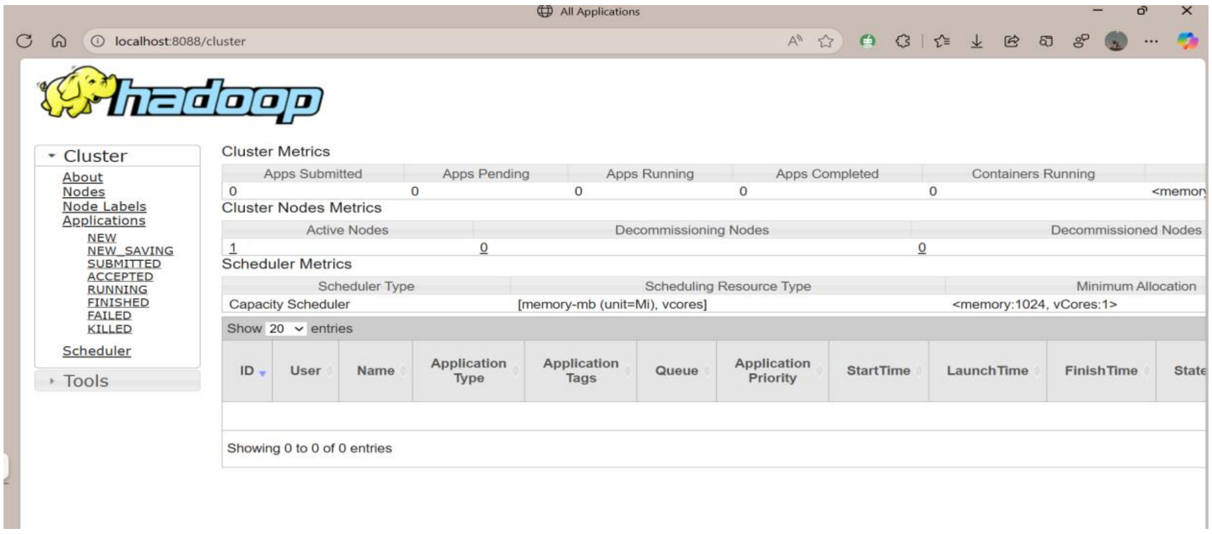
NameNode Storage

Storage Directory	Type	State
/home/ubuntu/hadoop_data/namenode	IMAGE_AND_EDITS	Active

DFS Storage Types

Storage Type	Configured Capacity	Capacity Used	Capacity Remaining	Block Pool Used	Nodes In Service
DISK	1006.85 GB	52 KB (0%)	946.2 GB (93.98%)	52 KB	1

2. ResourceManager Web UI: http://localhost:8088/



Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	
0	0	0	0	0	<memor

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes
1	0	0

Scheduler Metrics

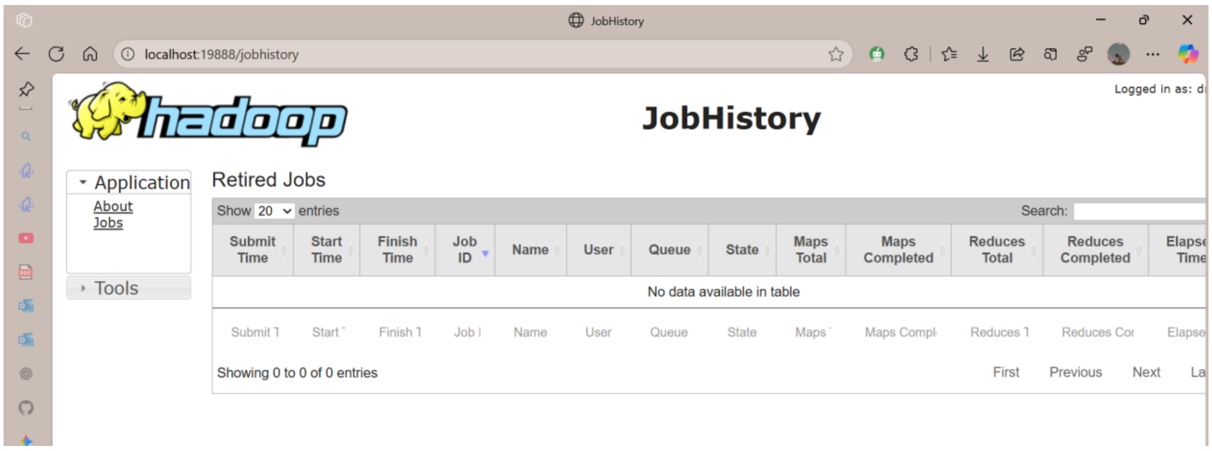
Scheduler Type	Scheduling Resource Type	Minimum Allocation
Capacity Scheduler	[memory-mb (unit=Mi), vcores]	<memory:1024, vCores:1>

Show 20 entries

ID	User	Name	Application Type	Application Tags	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State

Showing 0 to 0 of 0 entries

3. JobHistory Server Web UI: http://localhost:19888/
This server tracks the status of completed MapReduce jobs. It often needs to be started with a separate command *mapred --daemon start historyserver*



JobHistory

Retired Jobs

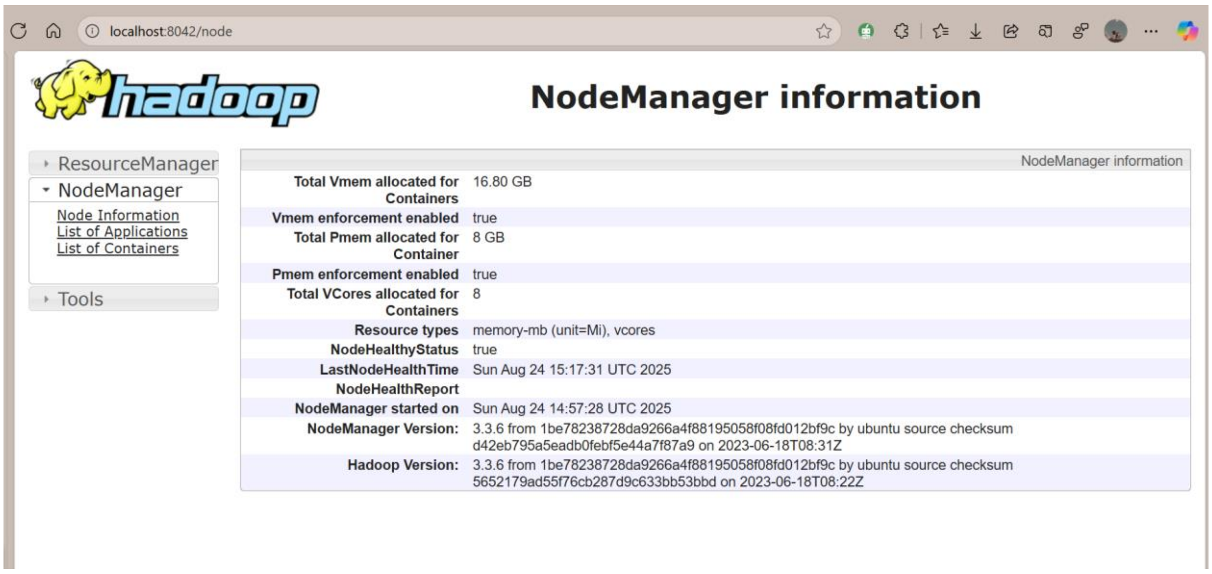
Show 20 entries Search:

Submit Time	Start Time	Finish Time	Job ID	Name	User	Queue	State	Maps Total	Maps Completed	Reduces Total	Reduces Completed	Elapsed Time
No data available in table												

Showing 0 to 0 of 0 entries

First Previous Next La

4. YARN NodeManager Web UI: http://localhost:8042/

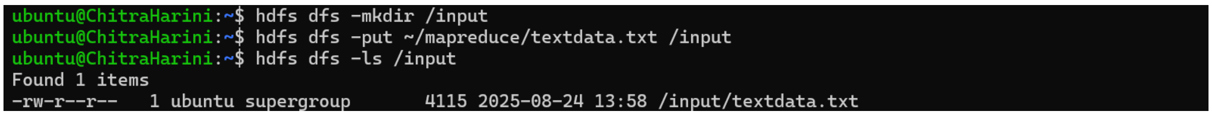


NodeManager information

	NodeManager information
Total Vmem allocated for Containers	16.80 GB
Vmem enforcement enabled	true
Total Pmem allocated for Container	8 GB
Pmem enforcement enabled	true
Total VCores allocated for Containers	8
Resource types	memory-mb (unit=Mi), vcores
NodeHealthyStatus	true
LastNodeHealthTime	Sun Aug 24 15:17:31 UTC 2025
NodeHealthReport	
NodeManager started on	Sun Aug 24 14:57:28 UTC 2025
NodeManager Version:	3.3.6 from 1be78238728da9266a4f88195058f08fd012bf9c by ubuntu source checksum d42eb795a5eadb0febf5e44a7f87a9 on 2023-06-18T08:31Z
Hadoop Version:	3.3.6 from 1be78238728da9266a4f88195058f08fd012bf9c by ubuntu source checksum 5652179ad55f76cb287d9c633bb53bbd on 2023-06-18T08:22Z

HDFS Interaction:

3. *hdfs dfs -mkdir /input* - Creates a new directory named input within HDFS.
4. *hdfs dfs -put textdata.txt /input* - Copies the local textdata.txt file into the /input directory on HDFS.
5. *hdfs dfs -ls /input* - Lists the contents of the /input directory to confirm the file was uploaded.



```
ubuntu@ChitraHarini:~$ hdfs dfs -mkdir /input
ubuntu@ChitraHarini:~$ hdfs dfs -put ~/mapreduce/textdata.txt /input
ubuntu@ChitraHarini:~$ hdfs dfs -ls /input
Found 1 items
-rw-r--r--   1 ubuntu supergroup       4115 2025-08-24 13:58 /input/textdata.txt
```

- **Explanation**: This task is the foundation for all Hadoop operations. The start-all.sh script brings the entire cluster online. HDFS provides the distributed storage layer, managed by the **NameNode** (which organizes the file system) and **DataNodes** (which store the data). YARN provides the processing framework, managed by the **ResourceManager** (the master that allocates resources) and **NodeManagers** (workers on each machine). The hdfs dfs commands are your tools for interacting with HDFS, much like using mkdir and cp in a standard Linux environment.

- **Conclusion**: Successfully completing this task results in a fully operational single-node Hadoop cluster. It confirms that the storage and processing layers are correctly configured and ready to accept data and execute jobs. The textdata.txt file is now stored within HDFS, prepared for the MapReduce tasks.

Task 2: Implement Word Count Using MapReduce

Write a MapReduce program in Java to count the occurrences of each word in a given document. The program should map each word to a key-value pair and reduce the data by summing the counts. Run the job on your Hadoop cluster and submit the Java code, a sample input file, the output of the job, and a brief description of the steps taken.

Objective

The objective of this task is to execute a fundamental MapReduce program called **WordCount**. This program reads a text file (textdata.txt) from HDFS, processes it in a distributed manner to count the occurrences of each unique word, and writes the final counts back into a new file on HDFS. This task validates that your entire Hadoop cluster—both HDFS for storage and YARN for processing—is working correctly.

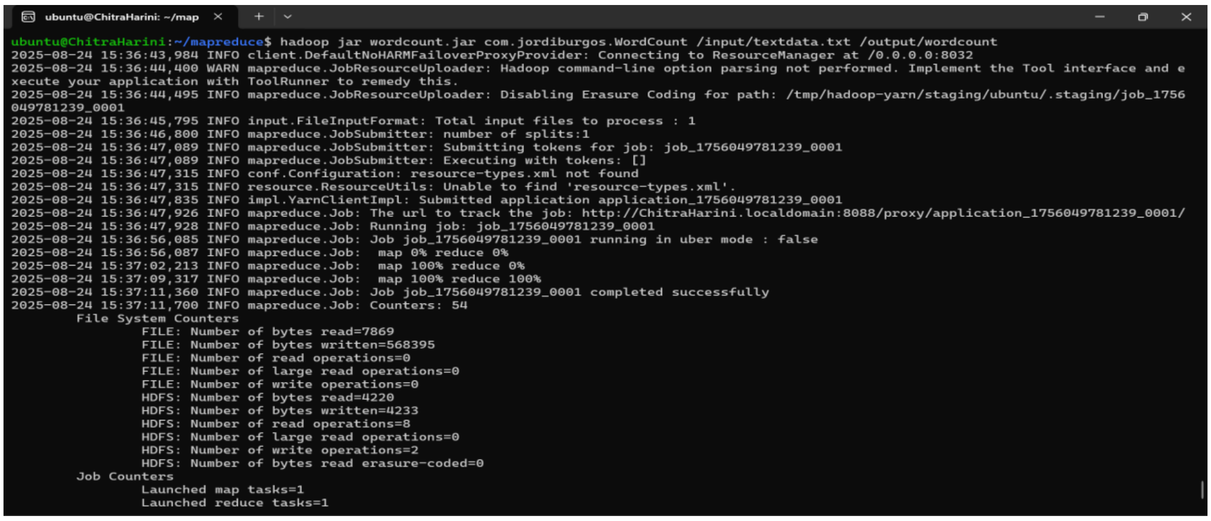
Commands

1. **Run the WordCount Job**

Command: *hadoop jar wordcount.jar com.jordiburgos.WordCount /input/textdata.txt /output/wordcount*

Explanation: This command tells the YARN ResourceManager to start a new application.

- hadoop jar wordcount.jar: Specifies the Java program to run.
- WordCount: The main class inside the JAR that contains the job configuration.
- /input/textdata.txt: The input file on HDFS that the job will read.
- /output/wordcount: The new directory on HDFS where the results will be stored. **This directory must not exist before you run the command.**

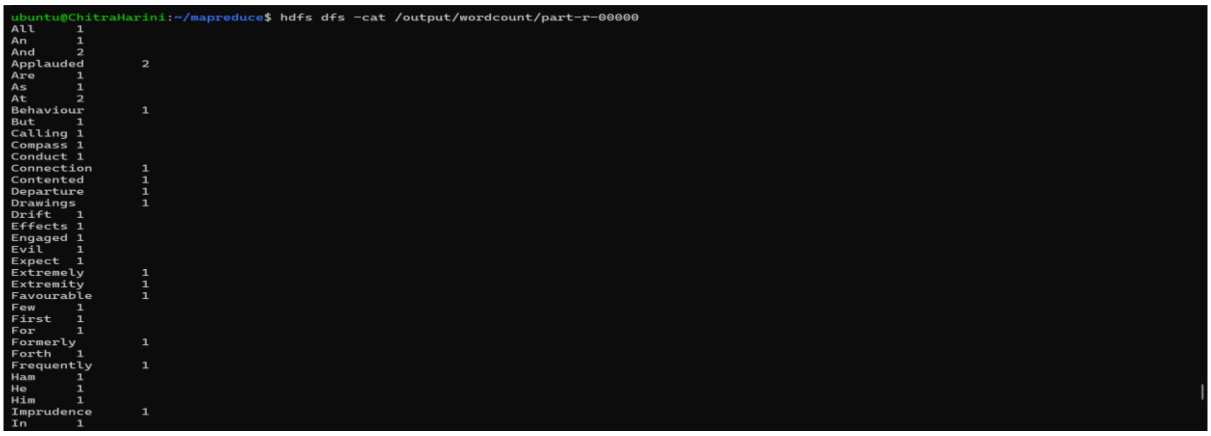


```
ubuntu@ChitraHarini:~/mapreduce$ hadoop jar wordcount.jar com.jordiburgos.WordCount /input/textdata.txt /output/wordcount
2025-08-24 15:36:43,984 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at /0.0.0.0:8032
2025-08-24 15:36:44,400 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and e
xecute your application with ToolRunner to remedy this.
2025-08-24 15:36:44,495 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/ubuntu/.staging/job_1756
049781239_0001
2025-08-24 15:36:45,795 INFO input.FileInputFormat: Total input files to process : 1
2025-08-24 15:36:46,800 INFO mapreduce.JobSubmitter: number of splits:1
2025-08-24 15:36:47,089 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1756049781239_0001
2025-08-24 15:36:47,089 INFO mapreduce.JobSubmitter: Executing with tokens: []
2025-08-24 15:36:47,315 INFO conf.Configuration: resource-types.xml not found
2025-08-24 15:36:47,315 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2025-08-24 15:36:47,835 INFO impl.YarnClientImpl: Submitted application application_1756049781239_0001
2025-08-24 15:36:47,926 INFO mapreduce.Job: The url to track the job: http://ChitraHarini.localdomain:8088/proxy/application_1756049781239_0001/
2025-08-24 15:36:47,928 INFO mapreduce.Job: Running job: job_1756049781239_0001
2025-08-24 15:36:56,085 INFO mapreduce.Job: Job job_1756049781239_0001 running in uber mode : false
2025-08-24 15:36:56,087 INFO mapreduce.Job:  map 0% reduce 0%
2025-08-24 15:37:02,213 INFO mapreduce.Job:  map 100% reduce 0%
2025-08-24 15:37:09,317 INFO mapreduce.Job:  map 100% reduce 100%
2025-08-24 15:37:11,360 INFO mapreduce.Job: Job job_1756049781239_0001 completed successfully
2025-08-24 15:37:11,700 INFO mapreduce.Job: Counters: 54
        File System Counters
                FILE: Number of bytes read=7869
                FILE: Number of bytes written=568395
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=4220
                HDFS: Number of bytes written=4233
                HDFS: Number of read operations=8
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
                HDFS: Number of bytes read erasure-coded=0
        Job Counters
                Launched map tasks=1
                Launched reduce tasks=1
```

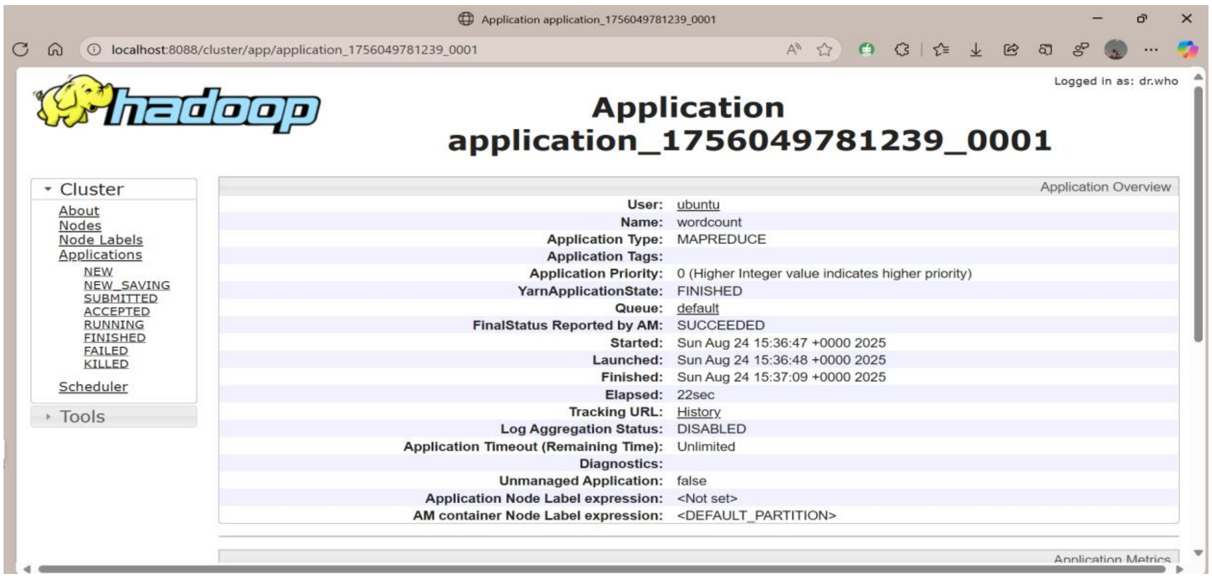
2. **View the Output**

   - **Command**: *hdfs dfs -cat /output/wordcount/part-r-00000*

- **Explanation**: After the job completes, this command prints the contents of the output file. The results are typically in a file named part-r-00000 (the 'r' stands for Reducer).



```
ubuntu@ChitraHarini:~/mapreduce$ hdfs dfs -cat /output/wordcount/part-r-00000
All     1
An      1
And     2
Applauded       2
Are     1
As      1
At      2
Behaviour       1
But     1
Calling 1
Compass 1
Conduct 1
Connection      1
Contented       1
Departure       1
Drawings        1
Drift   1
Effects 1
Engaged 1
Evil    1
Expect  1
Extremely       1
Extremity       1
Favourable      1
Few     1
First   1
For     1
Formerly        1
Forth   1
Frequently      1
Ham     1
He      1
Him     1
Imprudence      1
In      1
```



Application application_1756049781239_0001

localhost:8088/cluster/app/application_1756049781239_0001

Logged in as: dr.who

Application application_1756049781239_0001

Application Overview

User:	ubuntu
Name:	wordcount
Application Type:	MAPREDUCE
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Sun Aug 24 15:36:47 +0000 2025
Launched:	Sun Aug 24 15:36:48 +0000 2025
Finished:	Sun Aug 24 15:37:09 +0000 2025
Elapsed:	22sec
Tracking URL:	History
Log Aggregation Status:	DISABLED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Code Explanation

The Java code for WordCount, as seen in your mapreduce_Help.docx, has three main parts:

1. **The Driver (main method)**
   - **Purpose**: This is the orchestrator. It sets up and configures the MapReduce job.
   - **What it does**:
     - It takes the input path (/input/textdata.txt) and output path (/output/wordcount) as command-line arguments.
     - It tells Hadoop which class to use for the Mapper (MapForWordCount.class) and which to use for the Reducer (ReduceForWordCount.class).
     - It defines the data types for the final output key (Text for the word) and value (IntWritable for the count).
     - Finally, it submits the job to the YARN cluster and waits for it to complete.
2. **The Mapper (MapForWordCount class)**
   - **Purpose**: To process the raw input data and transform it into standardized key-value pairs.
   - **What it does**:
     - The map function receives one line of the textdata.txt file at a time.

        - It converts the line to a string and splits it into an array of words using spaces as the delimiter.
        - It then loops through each word. For every single word, it emits a key-value pair of **(word, 1)**. For example, the line "the dog saw the cat" results in five outputs: (THE, 1), (DOG, 1), (SAW, 1), (THE, 1), (CAT, 1).
3. **The Reducer (ReduceForWordCount class)**
    - **Purpose**: To aggregate the intermediate results from the Mapper.
    - **What it does**:
        - After the map phase, Hadoop's "shuffle and sort" process groups all the emitted pairs by their key. The Reducer receives one key at a time, along with a list of all its associated values.
        - For example, it will receive (THE, [1, 1]).
        - The reduce function then simply iterates through the list of values (the ones) and sums them up.
        - Finally, it writes the word and its final calculated sum to the output file in HDFS.

Conclusion
Successfully completing this task demonstrates a complete, end-to-end execution of a distributed data processing job. It proves that your Hadoop cluster can take input data from HDFS, process it in parallel across the cluster using the MapReduce framework managed by YARN, and write the final, aggregated results back to HDFS. It is the "Hello, World!" of Big Data and confirms your setup is ready for more complex analysis.

Task 3: Implement Line Count Using MapReduce

Write a MapReduce program in Java to count the number of lines in a text document. The program should map each line to a key-value pair with the line count (1 for each line) and reduce the data by summing the counts. Run the job on your Hadoop cluster and submit the Java code, a sample input file, the output of the job, and a brief description of the steps taken.

Objective
The objective of this task is to develop, compile, and execute a custom MapReduce program from scratch. This program, **LineCount**, will read the textdata.txt file from HDFS and calculate the total number of lines in the document. This demonstrates a deeper understanding of the MapReduce paradigm by applying it to a new problem.

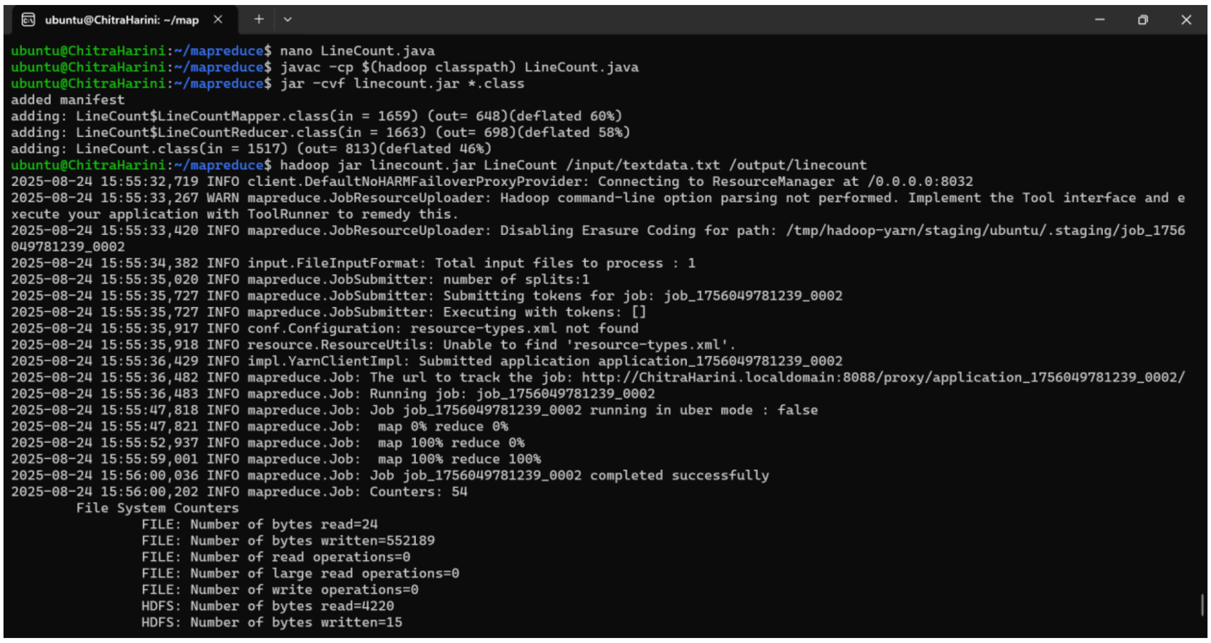
Java Code (LineCount.java)
```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import 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;
```

```
public class LineCount {
 // The Mapper Class
 public static class LineCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
   private final static IntWritable one = new IntWritable(1);
   private Text constantKey = new Text("Total Lines");
   public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
     // For every line we read, we emit the same key ("Total Lines") and the value 1.
     // We don't care what the line's content is.
     context.write(constantKey, one);
   }
 }
 // The Reducer Class
 public static class LineCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
   public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
     int sum = 0;
     // The reducer receives the key "Total Lines" and a list of 1s ([1, 1, 1, ...]).
     // We just need to sum them up to get the total count.
     for (IntWritable val : values) {
       sum += val.get();
     }
     context.write(key, new IntWritable(sum));
   }
 }
 // The Driver
 public static void main(String[] args) throws Exception {
   Configuration conf = new Configuration();
   Job job = Job.getInstance(conf, "line count");
   job.setJarByClass(LineCount.class);
   job.setMapperClass(LineCountMapper.class);
   // Combiner is the same as the Reducer in this case, which is a common optimization.
   job.setCombinerClass(LineCountReducer.class);
   job.setReducerClass(LineCountReducer.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);
 }
}
```

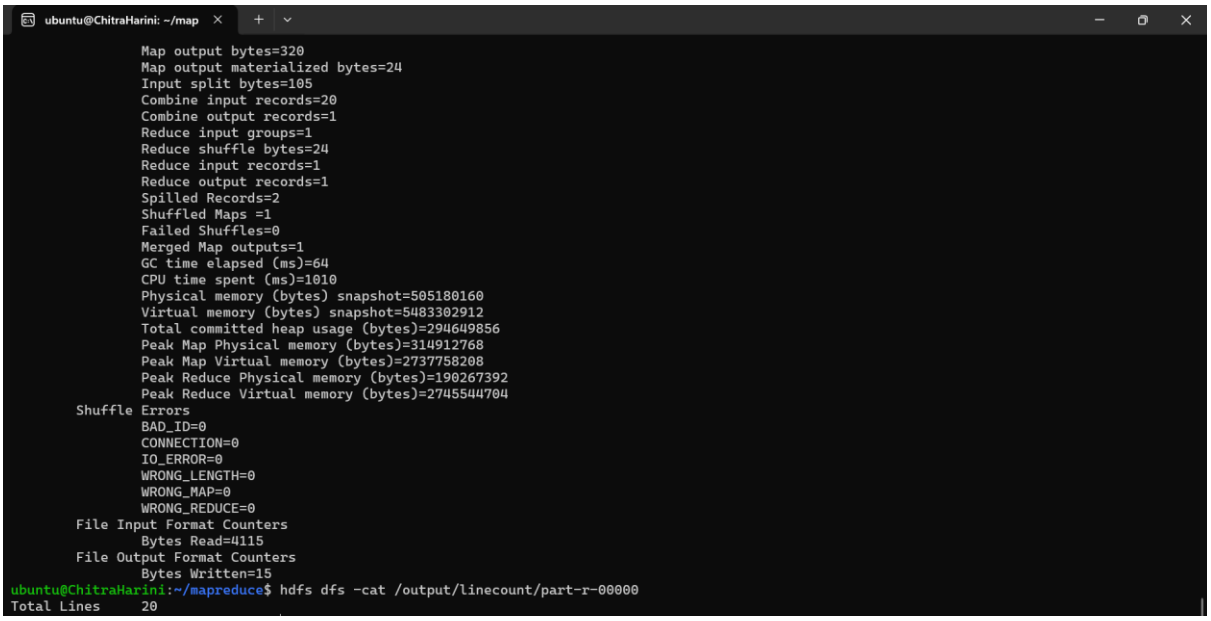
Commands:

Follow these steps in your terminal to create, compile, package, and run your program.

1. **Create the Java File**: Use nano to create the file and paste the code from above into it.
   ```
   nano LineCount.java
   ```
2. **Compile the Code**: This command uses javac to compile your program. The -cp flag tells the compiler where to find all the necessary Hadoop library files.
   ```
   javac -cp $(hadoop classpath) LineCount.java
   ```
3. **Create the JAR File**: This command packages your compiled .class files into a single linecount.jar file that you can submit to Hadoop.
   ```
   jar -cvf linecount.jar *.class
   ```
4. **Run the MapReduce Job**: Execute your custom JAR on the cluster. We'll use the same input file and create a new output directory.
   ```
   hadoop jar linecount.jar LineCount /input/textdata.txt /output/linecount
   ```
5. **View the Output**: Check the result file to see the final line count.
   ```
   hdfs dfs -cat /output/linecount/part-r-00000
   ```



```
ubuntu@ChitraHarini:~/mapreduce$ nano LineCount.java
ubuntu@ChitraHarini:~/mapreduce$ javac -cp $(hadoop classpath) LineCount.java
ubuntu@ChitraHarini:~/mapreduce$ jar -cvf linecount.jar *.class
added manifest
adding: LineCount$LineCountMapper.class(in = 1659) (out= 648)(deflated 60%)
adding: LineCount$LineCountReducer.class(in = 1663) (out= 698)(deflated 58%)
adding: LineCount.class(in = 1517) (out= 813)(deflated 46%)
ubuntu@ChitraHarini:~/mapreduce$ hadoop jar linecount.jar LineCount /input/textdata.txt /output/linecount
2025-08-24 15:55:32,719 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at /0.0.0.0:8032
2025-08-24 15:55:33,267 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and e
xecute your application with ToolRunner to remedy this.
2025-08-24 15:55:33,420 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/ubuntu/.staging/job_1756
049781239_0002
2025-08-24 15:55:34,382 INFO input.FileInputFormat: Total input files to process : 1
2025-08-24 15:55:35,020 INFO mapreduce.JobSubmitter: number of splits:1
2025-08-24 15:55:35,727 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1756049781239_0002
2025-08-24 15:55:35,727 INFO mapreduce.JobSubmitter: Executing with tokens: []
2025-08-24 15:55:35,917 INFO conf.Configuration: resource-types.xml not found
2025-08-24 15:55:35,918 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2025-08-24 15:55:36,429 INFO impl.YarnClientImpl: Submitted application application_1756049781239_0002
2025-08-24 15:55:36,482 INFO mapreduce.Job: The url to track the job: http://ChitraHarini.localdomain:8088/proxy/application_1756049781239_0002/
2025-08-24 15:55:36,483 INFO mapreduce.Job: Running job: job_1756049781239_0002
2025-08-24 15:55:47,818 INFO mapreduce.Job: Job job_1756049781239_0002 running in uber mode : false
2025-08-24 15:55:47,821 INFO mapreduce.Job:  map 0% reduce 0%
2025-08-24 15:55:52,937 INFO mapreduce.Job:  map 100% reduce 0%
2025-08-24 15:55:59,001 INFO mapreduce.Job:  map 100% reduce 100%
2025-08-24 15:56:00,036 INFO mapreduce.Job: Job job_1756049781239_0002 completed successfully
2025-08-24 15:56:00,202 INFO mapreduce.Job: Counters: 54
        File System Counters
                FILE: Number of bytes read=24
                FILE: Number of bytes written=552189
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=4220
                HDFS: Number of bytes written=15
```



```
                Map output bytes=320
                Map output materialized bytes=24
                Input split bytes=105
                Combine input records=20
                Combine output records=1
                Reduce input groups=1
                Reduce shuffle bytes=24
                Reduce input records=1
                Reduce output records=1
                Spilled Records=2
                Shuffled Maps =1
                Failed Shuffles=0
                Merged Map outputs=1
                GC time elapsed (ms)=64
                CPU time spent (ms)=1010
                Physical memory (bytes) snapshot=505180160
                Virtual memory (bytes) snapshot=5483302912
                Total committed heap usage (bytes)=294649856
                Peak Map Physical memory (bytes)=314912768
                Peak Map Virtual memory (bytes)=2737758208
                Peak Reduce Physical memory (bytes)=190267392
                Peak Reduce Virtual memory (bytes)=2745544704
        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=4115
        File Output Format Counters
                Bytes Written=15
ubuntu@ChitraHarini:~/mapreduce$ hdfs dfs -cat /output/linecount/part-r-00000
Total Lines     20
```

Code Explanation

- **The Mapper (LineCountMapper)**: This is the simplest possible Mapper. For every single line it reads from the input file, it ignores the content of the line entirely and just outputs a static key-value pair: ("Total Lines", 1).
- **The Reducer (LineCountReducer)**: After the map phase, the shuffle and sort process gathers all the emitted pairs. Since every key was identical, the Reducer receives just one group: the key "Total Lines" and a very long list of 1s, where the length of the list is equal to the number of lines in the original file. The Reducer's job is to simply iterate through this list and sum all the 1s to get the final total.
- **The Driver (main method)**: Just like in WordCount, the driver configures the job. It specifies the Mapper and Reducer classes, sets the input and output paths from the command line, and defines the data types for the final output.

Conclusion

By successfully writing, compiling, and running the LineCount program, you demonstrate a solid grasp of the MapReduce programming model. You have proven that you can analyze a problem (counting lines) and break it down into a parallelizable solution using the map and reduce paradigm. The final output, a single file containing the total line count, is the result of your own custom-built, distributed data processing application.