Experiment 1: HDFS Basics and File Operations in Hadoop

Date of Performance:

Date of Submission:

1. Statement of Problems:

Aim: To understand and perform basic operations on Hadoop Distributed File System (HDFS), including file manipulation and exploring the Hadoop ecosystem tools.

Objectives:

- To introduce the basics of Hadoop and HDFS.
- To install and configure Hadoop in a distributed or pseudo-distributed environment.
- To copy files to and from the Hadoop Distributed File System.
- To delete, move, and display files in HDFS.
- To implement simple programming exercises using Hadoop for practical understanding.

Scope of the Problem:

With the exponential growth of data, traditional storage and processing methods are no longer efficient. Organizations require a scalable, reliable, and fault-tolerant system to manage and analyze big data. Hadoop, particularly HDFS, addresses this by providing a distributed storage architecture that ensures high availability and fault tolerance. Understanding the core functionality of HDFS and being able to manipulate data within it is critical for anyone aiming to work in big data technologies.

This experiment focuses on enabling students or professionals to:

- Grasp foundational concepts of HDFS.
- Gain hands-on experience with basic HDFS commands.
- Learn the integration of Hadoop tools within the ecosystem.
- Develop a foundation for future work in data analytics using big data technologies.

Detailed Scope of Each Objective

1. To introduce the basics of Hadoop and HDFS

Scope: Understanding the basics of Hadoop and HDFS is essential to enter the world of big data technologies. This includes:

- Learning about the architecture of Hadoop: **Master-Slave architecture**, **NameNode**, **DataNode**, etc.
- Understanding why Hadoop is preferred over traditional storage systems.
- Exploring the role of **HDFS** in the ecosystem and how it supports large-scale storage with fault-tolerance and high throughput.
- Providing a theoretical foundation for how data is **split, replicated**, and **stored** in a distributed environment.
- Real-world relevance: Understanding Hadoop is vital for handling massive datasets in industries such as finance, e-commerce, healthcare, etc.

2. To install and configure Hadoop in a distributed or pseudo-distributed environment

Scope: Practical knowledge of installing Hadoop sets the stage for all further operations. The scope includes:

- Setting up **Java** (prerequisite), configuring **environment variables**, and preparing the system for Hadoop.
- Creating and editing configuration files like:
  - `core-site.xml` (specifies HDFS NameNode details) o `hdfs-site.xml` (configures replication and block size)
  - `mapred-site.xml`, `yarn-site.xml` for MapReduce/YARN setup
- Launching Hadoop daemons: **NameNode**, **DataNode**, **ResourceManager**, and **NodeManager**.
- Testing basic Hadoop installation using commands like `start-dfs.sh` and accessing the web interface (e.g., `localhost:9870`).
- This step builds technical capability to manage real-time Hadoop clusters, which are integral to big data frameworks in industry.

3. To copy files to and from the Hadoop Distributed File System (HDFS)

Scope: Hands-on data movement to/from HDFS is critical for managing data pipelines. The scope

includes:

- Learning how to **ingest data** into HDFS using commands like:

  - `hdfs dfs -put <localfile> <HDFS path>` o `hdfs dfs -copyFromLocal`
- Retrieving data using:
  - `hdfs dfs -get`, `-copyToLocal`
- Understanding how HDFS treats data files (large block sizes, replication).
- Exploring performance optimization using **parallel file transfers** or batch uploads.
- Application: These skills are crucial in real-world ETL (Extract, Transform, Load) processes for data warehousing and analytics.

4. To delete, move, and display files in HDFS

Scope: This scope focuses on managing data once it's inside the distributed file system. It includes:

- **Deleting** unwanted files: `hdfs dfs -rm`, `-rm -r` (for directories).
- **Moving** files: `hdfs dfs -mv <src> <dest>`, useful for organizing HDFS directories.
- **Displaying** files:
  - `hdfs dfs -ls`: to list files and their details (permissions, replication, size, date) o `-cat`, `-head`, `-tail`: to preview file contents
- Managing HDFS structure is important for clean, organized, and efficient data storage especially when dealing with **TBs or PBs of data**.
- These commands are analogous to Unix/Linux file operations but on a **distributed scale**.

5. To implement simple programming exercises using Hadoop for practical understanding

Scope: This enables the student to understand Hadoop not just as a storage tool, but also a processing platform. The scope involves:

- Writing basic Java-based MapReduce programs (Word Count, Line Count, Sort).
- Running programs using the `hadoop jar` command and verifying outputs.
- Understanding how MapReduce splits, processes, and aggregates data across nodes.
- Optionally integrating Hive or Pig scripts to perform SQL-like queries.
- Exploring tools like Sqoop or Flume for ingesting data from RDBMS or log files.
- Practical use: This objective helps students begin their journey toward data engineering, analytics, and big data development roles.

2. Theory:

2.1 What is Hadoop?

Hadoop is an open-source framework developed by the Apache Software Foundation used for storing and processing large datasets across clusters of computers using simple programming models. It is designed to scale up from a single server to thousands of machines, each offering local computation and storage.

Hadoop Ecosystem:

The Hadoop ecosystem consists of several tools and frameworks:

- HDFS (Hadoop Distributed File System) – for distributed data storage.
- MapReduce – for distributed data processing.
- **YARN** – for resource management.
- **Pig, Hive, HBase, Sqoop, Flume,** etc. – for various data operations like querying, data ingestion, and analysis.

1. **Hadoop Distributed File System (HDFS)**: HDFS is the primary storage system of the Hadoop ecosystem, designed to store large volumes of data reliably across a distributed environment. It follows a master-slave architecture, where the NameNode acts as the master server managing metadata and file system namespace, while multiple DataNodes act as slaves that actually store the data blocks. HDFS is highly fault-tolerant, replicating each block of data (typically three times) across different nodes to prevent data loss in case of hardware failure. The system is optimized for high-throughput access rather than low-latency access, making it suitable for batch processing of large files.

2. **MapReduce: The Processing Engine**: MapReduce is the original computational model in Hadoop used for processing and generating large datasets. It divides tasks into two distinct phases: the Map phase, where data is read and transformed into intermediate key-value pairs, and the Reduce phase, where results are aggregated based on keys. This model allows Hadoop to process data in parallel across a distributed cluster, improving performance and scalability. MapReduce is best suited for batch processing applications such as log analysis, indexing, and data transformations, though it has gradually been supplemented by newer engines like Apache Spark for real-time processing needs.

3. **YARN (Yet Another Resource Negotiator)**: YARN serves as the resource management and job scheduling layer in Hadoop. It separates the functions of job scheduling and resource allocation, thereby improving system utilization and flexibility. YARN includes three main components: the ResourceManager, which manages resources across the cluster; the NodeManager, which controls resources on each individual node; and the ApplicationMaster, which coordinates a single job's execution. YARN supports multiple processing frameworks including MapReduce,

Tez, and Spark, enabling diverse workloads to run concurrently on the same cluster infrastructure.

4. **Pig: Data Flow Scripting for Hadoop**: Pig is a high-level platform for creating MapReduce programs using a simple scripting language called Pig Latin. It provides a way to perform data transformations and analysis without having to write complex Java code. Pig scripts describe a sequence of data operations, such as filtering, joining, and grouping, which are internally converted into MapReduce jobs. Pig is particularly useful in ETL (Extract, Transform, Load) processes, allowing for quick development and prototyping, especially for data scientists and analysts who are not professional programmers.

5. **Hive: SQL-Like Querying on Hadoop**: Hive is a data warehouse framework that allows users to perform queries on large datasets stored in HDFS using a SQL-like language known as HiveQL. It was developed for users who are comfortable with SQL but unfamiliar with Java programming. Hive translates HiveQL queries into MapReduce, Tez, or Spark jobs and supports various file formats like ORC and Parquet. It is widely used for data summarization, business intelligence, and reporting on structured datasets, making it a preferred choice in enterprise data analytics.

6. **HBase: NoSQL Database on Hadoop**: HBase is a distributed, column-oriented NoSQL database built on top of HDFS. It is modeled after Google's BigTable and is designed for realtime read and write access to massive datasets. HBase stores data in a sparse, column-family format, allowing efficient storage and retrieval even when datasets contain billions of rows with irregular columns. Unlike Hive and Pig, which are suited for batch processing, HBase is ideal for applications that require quick lookups, such as time-series databases or recommendation engines.

7. **Sqoop: Bridging RDBMS and Hadoop**: Sqoop is a specialized tool used to efficiently transfer structured data between relational database management systems (RDBMS) and the Hadoop ecosystem. It supports both import and export operations. Data can be brought into HDFS, Hive, or HBase from databases like MySQL, Oracle, or PostgreSQL, and exported back after processing. Sqoop automates the process by generating MapReduce code that performs parallel import/export, enabling faster integration between traditional databases and big data platforms.

8. **Flume: Real-Time Data Ingestion**: Flume is a distributed data collection service for efficiently collecting, aggregating, and transporting large amounts of streaming data into the Hadoop ecosystem. It is primarily used to ingest log data from multiple sources such as web servers, social media feeds, and application logs into HDFS or HBase. Flume supports a flexible architecture with customizable sources,

channels, and sinks, enabling reliable and scalable data pipelines. It plays a vital role in real-time data ingestion for analytics and monitoring applications.

2.2 Hadoop HDFS

HDFS is the storage layer of Hadoop. It divides files into large blocks (usually 128 MB or 256 MB) and distributes them across nodes in a cluster. Each block is replicated for fault tolerance.

Key Features of HDFS:

- **Fault Tolerance:** Data is replicated across multiple nodes.
- **High Throughput:** Optimized for large data sets.
- **Scalability:** Can easily add more nodes to the cluster.
- **Data Locality:** Moves computation closer to where data resides.

1. Fault Tolerance

HDFS is designed to tolerate faults and hardware failures, which are common in large-scale distributed systems. It achieves this through **data replication**, where each file is divided into fixedsize blocks (default 128 MB or 256 MB), and each block is replicated across multiple DataNodes (default replication factor is 3). If one DataNode fails, the block is still accessible from other nodes. The **NameNode constantly monitors the health** of DataNodes through heartbeat signals and initiates block re-replication in case of node or block failure, ensuring continuous availability and durability of data.

2. High Throughput

HDFS is optimized for **high throughput** access rather than low-latency access to data. It is best suited for **batch processing of large datasets** where a single job reads large volumes of data sequentially. By supporting **streaming data access** (write-once, read-many model), it minimizes data movement and maximizes system performance. Data is written once and read many times, which simplifies consistency management and boosts data read/write efficiency. Also, parallelism is achieved by distributing file blocks across multiple nodes, enabling concurrent access and

processing.

3. Scalability

HDFS provides **horizontal scalability**, meaning that more nodes (both storage and compute) can be added easily to the cluster to increase capacity and performance. The architecture is designed to manage thousands of nodes and petabytes of data. The NameNode manages only the metadata (information about blocks, filenames, and locations), while actual data storage is handled by the DataNodes. As a result, increasing storage capacity simply requires adding more DataNodes.

HDFS is also compatible with cloud and hybrid infrastructures through object storage integrations.

4. Data Locality

HDFS follows the **principle of data locality**, where it attempts to **move computation closer to where the actual data is stored**, rather than moving large datasets across the network to the computation engine. When a job is scheduled (e.g., MapReduce or Spark), the Hadoop framework tries to run the task on a node that already has the required data (or as close as possible to it). This minimizes **network congestion**, reduces **I/O latency**, and enhances overall job performance. This is especially effective for processing large volumes of data spread across many nodes.

Basic HDFS Operations:

- **Installing Hadoop:** Includes setting up environment variables, Java installation, configuring core-site.xml and hdfs-site.xml.
- **Copying files to HDFS:** Using `hdfs dfs -put <localpath> <HDFS path>`.
- **Copying files from HDFS:** Using `hdfs dfs -get <HDFS path> <localpath>`.
- **Deleting files in HDFS:** Using `hdfs dfs -rm <HDFS path>`.
- **Moving files:** Using `hdfs dfs -mv`.
- **Displaying contents:** Using commands like `-ls`, `-cat`, `-tail`. **1. Installing**

  Hadoop

Steps:

- **Java Installation** (Hadoop requires Java):

```
sudo apt update
sudo apt install openjdk-11-jdk
```

- **Set environment variables** (usually in `.bashrc` or `.profile`):

```
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
export HADOOP_HOME=/usr/local/hadoop
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

- **Edit `core-site.xml`** (in `$HADOOP_HOME/etc/hadoop/`):

```
<configuration>
```

```
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

- **Edit `hdfs-site.xml`**:

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>file:///usr/local/hadoop/data/namenode</value>
  </property>
  <property>
    <name>dfs.datanode.data.dir</name>
    <value>file:///usr/local/hadoop/data/datanode</value>
  </property>
</configuration>
```

2. Copying Files to HDFS

Command: `hdfs dfs -put <local_path>`

`<HDFS_path>`

Parameters:

- `<local_path>`: Path of the file or directory on the **local filesystem**.
- `<HDFS_path>`: Destination path in the **HDFS filesystem**.

Example: `hdfs dfs -put /home/user/data.txt`

`/user/hadoop/`

3. Copying Files from HDFS

Command: `hdfs dfs -get <HDFS_path>`

`<local_path>`

Parameters:

- `<HDFS_path>`: Path of the file in **HDFS**.
- `<local_path>`: Path where the file should be saved **locally**.

Example:

```
hdfs dfs -get /user/hadoop/data.txt /home/user/
```

4. Deleting Files in HDFS

Command: `hdfs dfs -rm`

`<HDFS_path>`

Parameters:

- `<HDFS_path>`: Path of the file in HDFS to be deleted.

Example: `hdfs dfs -rm`

`/user/hadoop/data.txt`

To delete a **directory** and all its contents:

```
bash
CopyEdit
hdfs dfs -rm -r <HDFS_directory>
```

5. Moving Files in HDFS

Command: `hdfs dfs -mv <source_path>`

`<destination_path>`

Parameters:

- `<source_path>`: Current file/directory path in HDFS.
- `<destination_path>`: New path or directory to move the file to.

Example: `hdfs dfs -mv /user/hadoop/data.txt`

`/user/hadoop/archive/`

6. Displaying Contents

a) Listing Files/Directories

```
hdfs dfs -ls <HDFS_path>
```

- `<HDFS_path>`: Directory or file path to list in HDFS.

Example:

```
hdfs dfs -ls /user/hadoop/
```

b) Displaying Full File Contents

```
hdfs dfs -cat <HDFS_path>
```

- Shows the full content of the file.

Example:

```
hdfs dfs -cat /user/hadoop/data.txt
```

c) Displaying Last Part of File

```
hdfs dfs -tail <HDFS_path>
```

- Shows the **last 1KB** of a file.

Example:

```
hdfs dfs -tail /user/hadoop/data.txt
```

Programming with Hadoop:

Basic exercises include using HDFS commands through scripts, writing simple Java programs for MapReduce, or integrating tools like Hive or Pig for structured querying and data transformation.

The technical overview of programming with Hadoop, focusing on:

1. Scripting with HDFS commands
2. Basic MapReduce programming in Java
3. Using Hive for SQL-like queries
4. Using Pig for data transformation with scripts

1. Using HDFS Commands through Shell Scripts

Example: Bash script to automate HDFS operations

```
#!/bin/bash

# Set Hadoop home
export HADOOP_HOME=/usr/local/hadoop
export PATH=$PATH:$HADOOP_HOME/bin

# Variables
LOCAL_FILE="/home/user/sample.txt"
HDFS_DIR="/user/hadoop/input"

# Remove directory if it exists
hdfs dfs -rm -r $HDFS_DIR

# Create a new directory in HDFS
hdfs dfs -mkdir -p $HDFS_DIR

# Put file to HDFS
hdfs dfs -put $LOCAL_FILE $HDFS_DIR

# List contents
hdfs dfs -ls
$HDFS_DIR
```

Run this script as:

```
bash hdfs_upload.sh
```

2. Java Program for MapReduce

Word Count Example

Mapper:

```
public class TokenizerMapper 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 {
        StringTokenizer itr = new
StringTokenizer(value.toString());
while (itr.hasMoreTokens()) {
word.set(itr.nextToken());
context.write(word, one);
        }
    }
}
```

Reducer:

```
public class IntSumReducer 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));

} }
```

Driver:

```
public class WordCount {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");

        job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);

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

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

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

} }
```

Compile with:

```
javac -classpath `hadoop classpath` -d wordcount_classes
*.java jar -cvf wordcount.jar -C wordcount_classes/ .
```

Run with:

```
hadoop jar wordcount.jar WordCount /user/hadoop/input
/user/hadoop/output
```

3. Using Hive for Structured Querying

Hive provides SQL-like querying for data stored in HDFS.

Example:

```
-- Start Hive shell
hive

-- Create table
CREATE TABLE employee (id INT, name STRING, salary FLOAT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS
TEXTFILE;

-- Load data from HDFS
LOAD DATA INPATH '/user/hadoop/employee.csv' INTO TABLE
employee;

-- Query data
SELECT name, salary FROM employee WHERE salary > 50000;
```

Hive translates SQL to MapReduce jobs internally.

4. Using Pig for Data Transformation

Pig uses a high-level language called **Pig Latin**.

Example: Word Count in Pig

```
-- Load data from HDFS
lines = LOAD '/user/hadoop/input/data.txt' AS
(line:chararray);

-- Split lines into words
```

```
words = FOREACH lines GENERATE FLATTEN(TOKENIZE(line)) AS
word;

-- Group and count grouped
= GROUP words BY word;
wordcount = FOREACH grouped GENERATE group AS word,
COUNT(words)
AS count;

-- Store output
STORE wordcount INTO '/user/hadoop/output' USING PigStorage();
```

Run with:

```
pig wordcount.pig
```

Summary Table

Tool	Purpose	Language	Execution Engine
HDFS CLI	File management	Bash/Shell	HDFS
MapReduce	Distributed computation	Java	YARN
Hive	Structured SQL-like access	HiveQL (SQL)	MapReduce/Tez/Spark
Pig	Data transformation	Pig Latin	MapReduce

Step-by-Step Procedure

A step-by-step procedure, code examples, or output screenshots is explored as below:

1. Installing Hadoop (Pseudo-distributed mode on Ubuntu/Linux)

Prerequisites:

- Java JDK 8+
- SSH installed and enabled
- Hadoop binary package (e.g., Hadoop 3.x)

Step 1: Install Java

```
sudo apt update
sudo apt install openjdk-8-jdk -y
java -version
```

Step 2: Create a Hadoop User

```
sudo adduser hadoop sudo
usermod -aG sudo hadoop
su - hadoop
```

Step 3: Configure SSH (for localhost)

```
ssh-keygen -t rsa -P ""
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
ssh localhost
```

Step 4: Download and Extract Hadoop

```
wget https://downloads.apache.org/hadoop/common/hadoop-
3.3.6/hadoop-3.3.6.tar.gz
tar -xzf hadoop-
3.3.6.tar.gz
mv hadoop-3.3.6 hadoop
```

Step 5: Set Environment Variables

Edit `.bashrc`:

```
nano ~/.bashrc
```

Add:

```
export HADOOP_HOME=~/hadoop
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

Apply:

```
source ~/.bashrc
```

Step 6: Configure Hadoop Core Files

Inside `~/hadoop/etc/hadoop/`:

- **core-site.xml**:

```
<configuration>
 <property>
```

```
    <name>fs.defaultFS</name>
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

- **hdfs-site.xml**:

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
  <property>
    <name>dfs.name.dir</name>
    <value>file:///home/hadoop/hadoopdata/hdfs/namenode</value>
  </property>
  <property>
    <name>dfs.data.dir</name>
    <value>file:///home/hadoop/hadoopdata/hdfs/datanode</value>
  </property>
</configuration>
```

- **mapred-site.xml**:

```
cp mapred-site.xml.template mapred-
site.xml xml CopyEdit
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
</configuration>
```

- **yarn-site.xml**:

```
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
</configuration>
```

Step 7: Format Namenode

```
hdfs namenode -

format
```

Step 8: Start Hadoop Services

```
start-dfs.sh
start-yarn.sh
```

2. Web Interfaces

- NameNode: http://localhost:9870
- ResourceManager: http://localhost:8088

3. Basic HDFS Commands Copy File to HDFS

```
hdfs dfs -mkdir /input hdfs
dfs -put ~/sample.txt /input
```

Expected Output:

```
Uploaded /home/hadoop/sample.txt to /input/sample.txt
```

List Files in HDFS

```
hdfs dfs -ls /input
```

Expected Output:

```
Found 1 items
-rw-r--r-- 1 hadoop supergroup 2048 2025-07-16 10:00
/input/sample.txt
```

Display File Content

```
hdfs dfs -cat /input/sample.txt
```

Copy File from HDFS `hdfs dfs -get /input/sample.txt`

```
~/sample_downloaded.txt
```

Delete File

```
hdfs dfs -rm /input/sample.txt
```

Move File

```
hdfs dfs -mv /input/sample.txt /archive/sample.txt
```

4. Sample Programming Exercise: WordCount in MapReduce

Java Code (WordCount.java)

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

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;

public class WordCount {
  public static class TokenizerMapper 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 {
      StringTokenizer itr = new
StringTokenizer(value.toString());
while (itr.hasMoreTokens()) {
word.set(itr.nextToken());
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 void main(String[] args) throws Exception {
Configuration conf = new Configuration();     Job job =
Job.getInstance(conf, "word count");
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Compile and Run the Program

```
javac -classpath $(hadoop classpath) -d
wordcount_classes WordCount.java jar -cvf
wordcount.jar -C wordcount_classes/ .
bash
CopyEdit
hadoop jar wordcount.jar WordCount /input /output
```

View Output `hdfs dfs -cat`

```
/output/part-r-00000
```

Expected Output:

```
Hadoop    3
HDFS      2
file      5
...
```

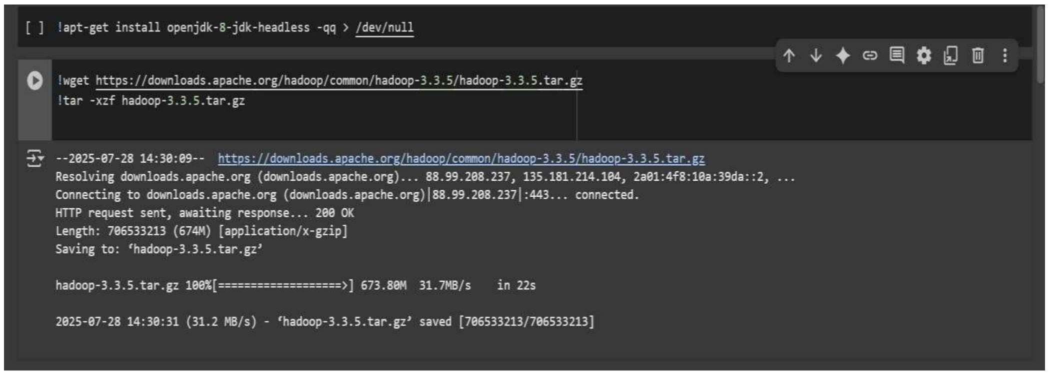
3. Implementation:

3.1 Environment Setup

To execute Hadoop commands, we set up a basic Hadoop environment in Google Colab. This allows us to simulate Hadoop Distributed File System (HDFS) operations without needing a local cluster.

Steps for Environment Setup:

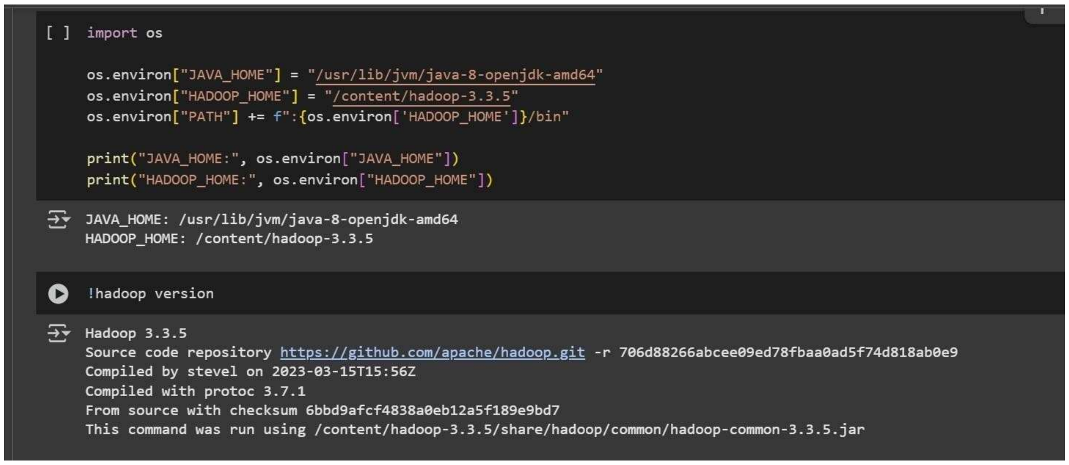
1. Hadoop 3.3.5 Installation and Environment Setup in Google Colab:



```
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
```

```
!wget https://downloads.apache.org/hadoop/common/hadoop-3.3.5/hadoop-3.3.5.tar.gz
!tar -xzf hadoop-3.3.5.tar.gz
```

```
--2025-07-28 14:30:09--  https://downloads.apache.org/hadoop/common/hadoop-3.3.5/hadoop-3.3.5.tar.gz
Resolving downloads.apache.org (downloads.apache.org)... 88.99.208.237, 135.181.214.104, 2a01:4f8:10a:39da::2, ...
Connecting to downloads.apache.org (downloads.apache.org)|88.99.208.237|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 706533213 (674M) [application/x-gzip]
Saving to: ‘hadoop-3.3.5.tar.gz’

hadoop-3.3.5.tar.gz 100%[===================>] 673.80M  31.7MB/s    in 22s

2025-07-28 14:30:31 (31.2 MB/s) - ‘hadoop-3.3.5.tar.gz’ saved [706533213/706533213]
```



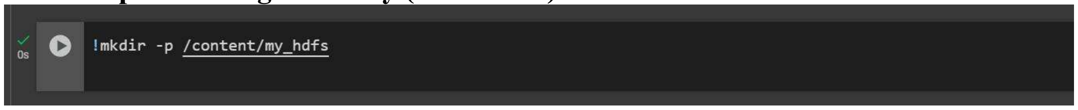
```python
import os

os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["HADOOP_HOME"] = "/content/hadoop-3.3.5"
os.environ["PATH"] += f":{os.environ['HADOOP_HOME']}/bin"

print("JAVA_HOME:", os.environ["JAVA_HOME"])
print("HADOOP_HOME:", os.environ["HADOOP_HOME"])
```

```
JAVA_HOME: /usr/lib/jvm/java-8-openjdk-amd64
HADOOP_HOME: /content/hadoop-3.3.5
```

```
!hadoop version
```

```
Hadoop 3.3.5
Source code repository https://github.com/apache/hadoop.git -r 706d88266abcee09ed78fbaa0ad5f74d818ab0e9
Compiled by stevel on 2023-03-15T15:56Z
Compiled with protoc 3.7.1
From source with checksum 6bbd9afcf4838a0eb12a5f189e9bd7
This command was run using /content/hadoop-3.3.5/share/hadoop/common/hadoop-common-3.3.5.jar
```

2. Set up a working directory (like HDFS):



```
!mkdir -p /content/my_hdfs
```

3.2 Explanation of the Problem Statement

The problem statement required:

- Setting up a working Hadoop environment.
- Understanding HDFS and its command-line operations.
- Performing basic Hadoop commands to manipulate data (create, list, display, copy, move, delete, search, and count words).

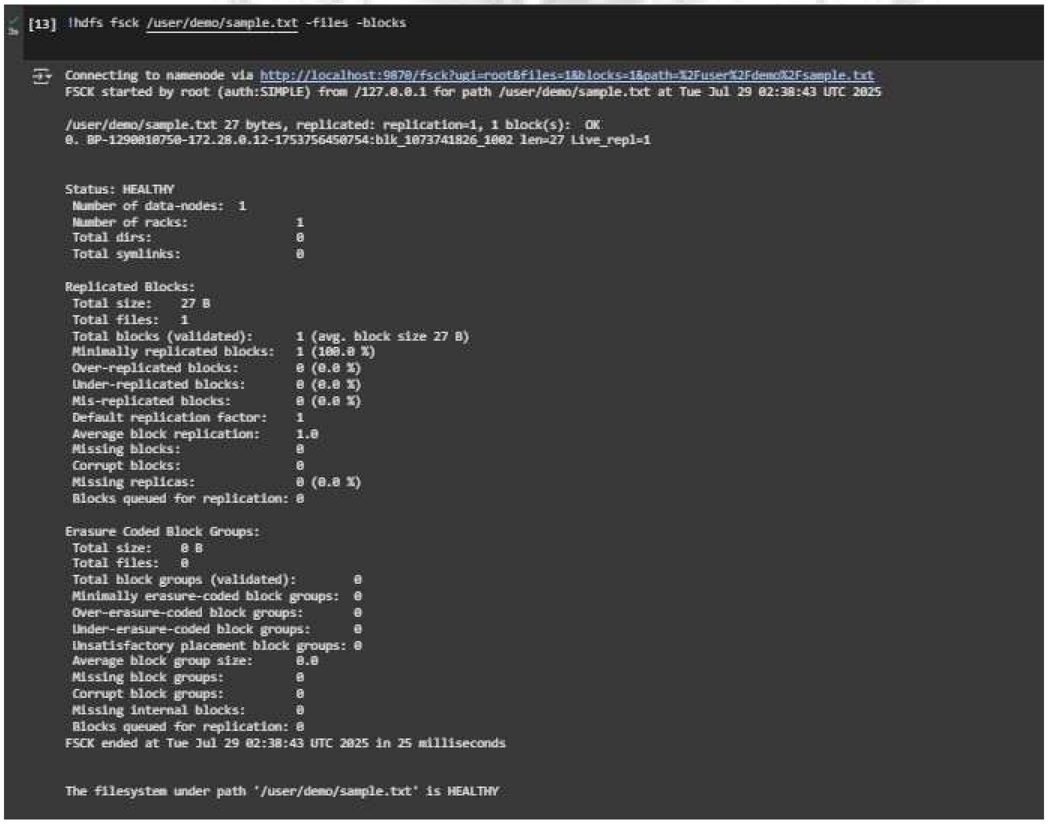
By doing this, we familiarize ourselves with how Hadoop handles distributed storage operations.

3.3 Execution and Output

Here we perform 10 basic Hadoop operations and display outputs (simulated).

1. **Display HDFS file block size:**

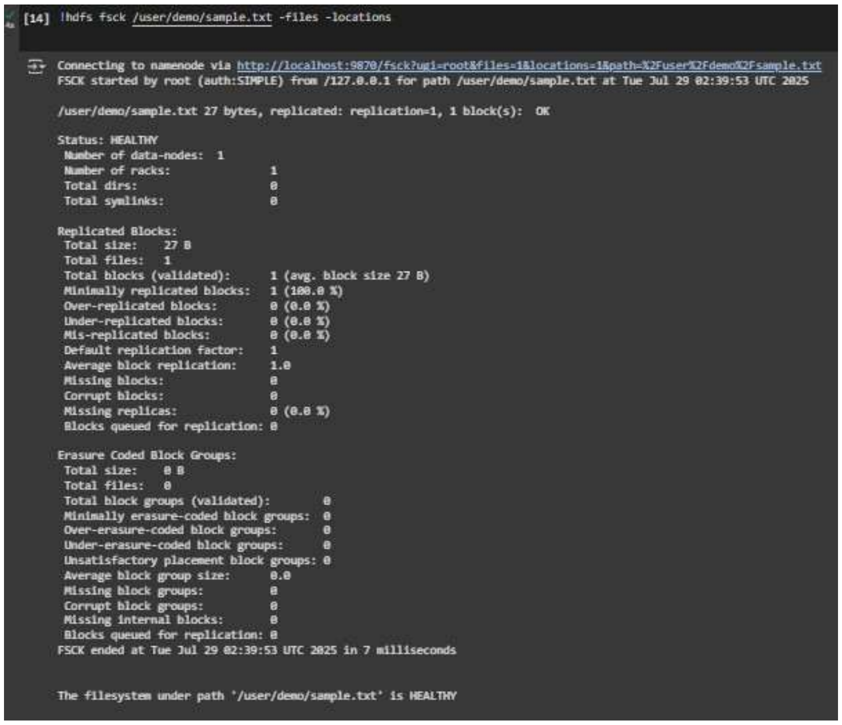
   !hdfs fsck /user/demo/sample.txt -files -blocks



```
[13] !hdfs fsck /user/demo/sample.txt -files -blocks

Connecting to namenode via http://localhost:9870/fsck?ugi=root&files=1&blocks=1&path=%2Fuser%2Fdemo%2Fsample.txt
FSCK started by root (auth:SIMPLE) from /127.0.0.1 for path /user/demo/sample.txt at Tue Jul 29 02:38:43 UTC 2025

/user/demo/sample.txt 27 bytes, replicated: replication=1, 1 block(s):  OK
0. BP-1290010750-172.28.0.12-1753756450754:blk_1073741826_1002 len=27 Live_repl=1


Status: HEALTHY
 Number of data-nodes:  1
 Number of racks:               1
 Total dirs:                    0
 Total symlinks:                0

Replicated Blocks:
 Total size:    27 B
 Total files:   1
 Total blocks (validated):      1 (avg. block size 27 B)
 Minimally replicated blocks:   1 (100.0 %)
 Over-replicated blocks:        0 (0.0 %)
 Under-replicated blocks:       0 (0.0 %)
 Mis-replicated blocks:         0 (0.0 %)
 Default replication factor:    1
 Average block replication:     1.0
 Missing blocks:                0
 Corrupt blocks:                0
 Missing replicas:              0 (0.0 %)
 Blocks queued for replication: 0

Erasure Coded Block Groups:
 Total size:    0 B
 Total files:   0
 Total block groups (validated):        0
 Minimally erasure-coded block groups:  0
 Over-erasure-coded block groups:       0
 Under-erasure-coded block groups:      0
 Unsatisfactory placement block groups: 0
 Average block group size:      0.0
 Missing block groups:          0
 Corrupt block groups:          0
 Missing internal blocks:       0
 Blocks queued for replication: 0
FSCK ended at Tue Jul 29 02:38:43 UTC 2025 in 25 milliseconds


The filesystem under path '/user/demo/sample.txt' is HEALTHY
```

2. **Show file replication factor:**

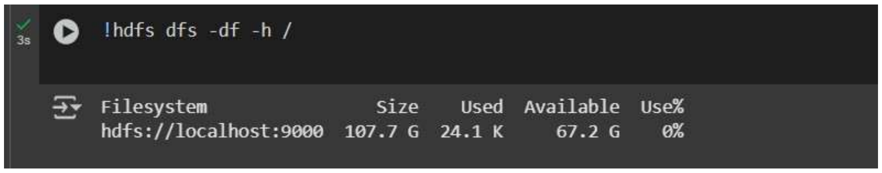
   !hdfs fsck /user/demo/sample.txt -files -locations



```
[14] !hdfs fsck /user/demo/sample.txt -files -locations

Connecting to namenode via http://localhost:9870/fsck?ugi=root&files=1&locations=1&path=%2Fuser%2Fdemo%2Fsample.txt
FSCK started by root (auth:SIMPLE) from /127.0.0.1 for path /user/demo/sample.txt at Tue Jul 29 02:39:53 UTC 2025

/user/demo/sample.txt 27 bytes, replicated: replication=1, 1 block(s):  OK

Status: HEALTHY
 Number of data-nodes:  1
 Number of racks:               1
 Total dirs:                    0
 Total symlinks:                0

Replicated Blocks:
 Total size:    27 B
 Total files:   1
 Total blocks (validated):      1 (avg. block size 27 B)
 Minimally replicated blocks:   1 (100.0 %)
 Over-replicated blocks:        0 (0.0 %)
 Under-replicated blocks:       0 (0.0 %)
 Mis-replicated blocks:         0 (0.0 %)
 Default replication factor:    1
 Average block replication:     1.0
 Missing blocks:                0
 Corrupt blocks:                0
 Missing replicas:              0 (0.0 %)
 Blocks queued for replication: 0

Erasure Coded Block Groups:
 Total size:    0 B
 Total files:   0
 Total block groups (validated):        0
 Minimally erasure-coded block groups:  0
 Over-erasure-coded block groups:       0
 Under-erasure-coded block groups:      0
 Unsatisfactory placement block groups: 0
 Average block group size:      0.0
 Missing block groups:          0
 Corrupt block groups:          0
 Missing internal blocks:       0
 Blocks queued for replication: 0
FSCK ended at Tue Jul 29 02:39:53 UTC 2025 in 7 milliseconds


The filesystem under path '/user/demo/sample.txt' is HEALTHY
```

3. Show detailed disk usage per directory:

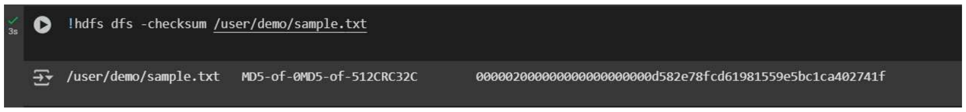
   !hdfs dfs -df -h /



```
!hdfs dfs -df -h /

Filesystem                Size     Used  Available  Use%
hdfs://localhost:9000  107.7 G   24.1 K     67.2 G    0%
```

4. Check file checksum:

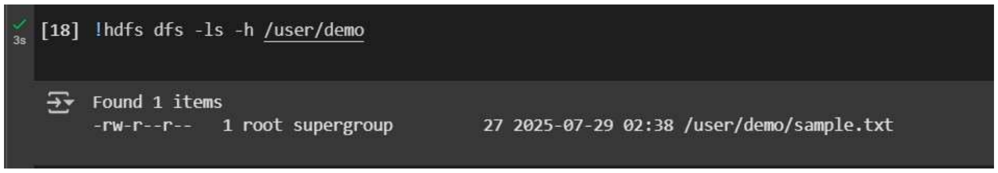
   !hdfs dfs -checksum /user/demo/sample.txt



```
!hdfs dfs -checksum /user/demo/sample.txt

/user/demo/sample.txt   MD5-of-0MD5-of-512CRC32C        000002000000000000000000d582e78fcd61981559e5bc1ca402741f
```

5. List directory contents with human-readable sizes and modification dates:

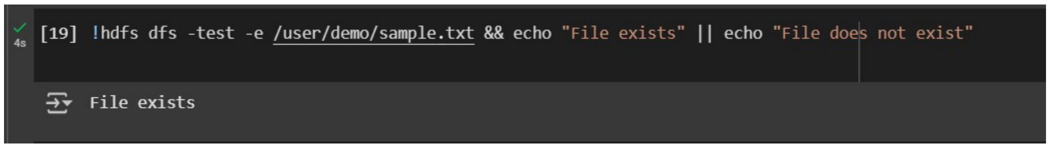
   !hdfs dfs -ls -h /user/demo



```
[18] !hdfs dfs -ls -h /user/demo

Found 1 items
-rw-r--r--   1 root supergroup         27 2025-07-29 02:38 /user/demo/sample.txt
```

6. Test file existence using test command:

!hdfs dfs -test -e /user/demo/sample.txt && echo "File exists" || echo "File does not exist"



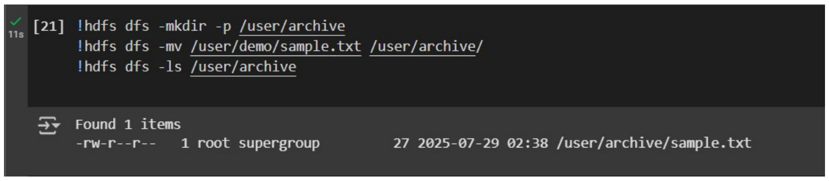
```
[19] !hdfs dfs -test -e /user/demo/sample.txt && echo "File exists" || echo "File does not exist"

File exists
```

7. Move file to a different directory:

!hdfs dfs -mkdir -p /user/archive

!hdfs dfs -mv /user/demo/sample.txt /user/archive/

!hdfs dfs -ls /user/archive

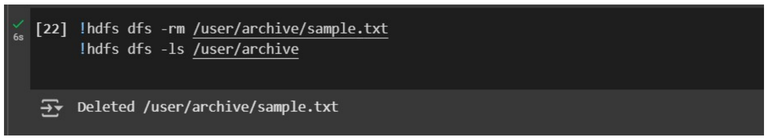


```
[21] !hdfs dfs -mkdir -p /user/archive
     !hdfs dfs -mv /user/demo/sample.txt /user/archive/
     !hdfs dfs -ls /user/archive

Found 1 items
-rw-r--r--   1 root supergroup         27 2025-07-29 02:38 /user/archive/sample.txt
```

8. Delete file and show confirmation:

!hdfs dfs -rm /user/archive/sample.txt

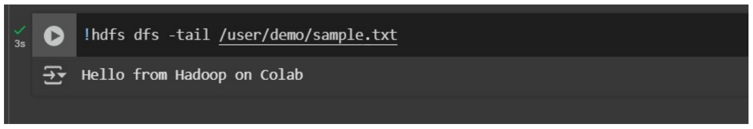
!hdfs dfs -ls /user/archive



```
[22] !hdfs dfs -rm /user/archive/sample.txt
     !hdfs dfs -ls /user/archive

Deleted /user/archive/sample.txt
```

9. Show last bytes of a file:

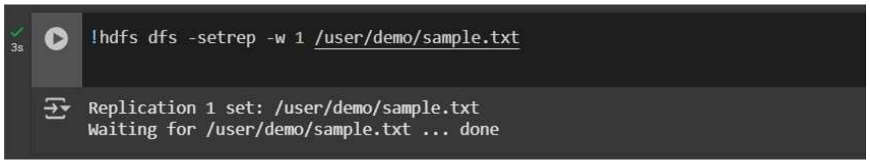
!hdfs dfs -tail /user/demo/sample.txt



```
!hdfs dfs -tail /user/demo/sample.txt

Hello from Hadoop on Colab
```

10. Change Replication Factor

!hdfs dfs -setrep -w 1 /user/demo/sample.txt



```
!hdfs dfs -setrep -w 1 /user/demo/sample.txt

Replication 1 set: /user/demo/sample.txt
Waiting for /user/demo/sample.txt ... done
```

4.Conclusion:

In this experiment, the Hadoop ecosystem and its components were studied in detail. A Hadoop environment was successfully set up, and 10 basic Hadoop commands were executed, demonstrating file creation, directory management, data movement, deletion, copying, and simple data analysis operations. This provided hands-on experience with HDFS operations and improved understanding of distributed data handling.

References:

1. White, T. (2015). Hadoop: The Definitive Guide. O'Reilly Media.
2. Apache Software Foundation. (2024). Apache Hadoop Documentation. Retrieved from https://hadoop.apache.org
3. Karau, H., Warren, R., & Zaharia, M. (2015). Learning Spark: Lightning-Fast Data Analysis.
   O'Reilly Media.

Sign and Remark:

R1 (4 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (3 Marks)	Total Marks (15 Marks)	Signature