1. Setting up NameNode:

//Initialize the NameNode by running:

hdfs namenode -format

//Start the HDFS service:

start-dfs.sh

2. Verifying Hadoop DFS (Distributed File System):

//To verify the HDFS setup, you can run various commands:

//List the contents of the root directory in HDFS:

hdfs dfs -ls /

//Create a directory in HDFS:

hdfs dfs -mkdir /test

//Upload a file to HDFS:

hdfs dfs -copyFromLocal localfile /test/

3. Verifying YARN Script:

//To verify YARN, you can use the following commands:

//Check the YARN ResourceManager status by visiting the web interface at http://localhost:8088/.

//List the nodes in the YARN cluster:

yarn node -list

4. Accessing Hadoop on Web Browser:

//Hadoop services through web browsers:

HDFS NameNode web interface: http://localhost:9870/

YARN ResourceManager web interface: http://localhost:8088/

5. Verify All Applications of Cluster:

1. Make sure your Hadoop cluster is up and running, including the ResourceManager service.

2. Open a web browser on your local machine.

3. Access the YARN ResourceManager web interface by navigating to the following URL in your web browser:

http://<ResourceManager-Host>:8088/

//Replace <ResourceManager-Host> with the hostname or IP address of your YARN ResourceManager node

1. **Add Files and Directories to HDFS:**

//To add files and directories to the Hadoop Distributed File System (HDFS), you can use the hadoop fs copyFromLocal command:

//To copy a local file to HDFS:

hadoop fs -copyFromLocal /path/to/local/file /path/in/hdfs/

//To copy a local directory to HDFS:

hadoop fs -copyFromLocal /path/to/local/directory /path/in/hdfs/

2. **Retrieve Files from HDFS:**

//To retrieve files from HDFS to your local machine, you can use the hadoop fs -copyToLocal command:

//To copy a file from HDFS to your local filesystem:

hadoop fs -copyToLocal /path/in/hdfs/file /path/in/local/

//To copy a directory from HDFS to your local filesystem:

hadoop fs -copyToLocal /path/in/hdfs/directory /path/in/local/

3. **Delete Files from HDFS:**

//To delete files and directories from HDFS, you can use the hadoop fs -rm command:

//To delete a file from HDFS:

hadoop fs -rm /path/in/hdfs/file

//To delete an empty directory from HDFS:

hadoop fs -rmdir /path/in/hdfs/directory

// To delete a directory and its contents from HDFS:

hadoop fs -rm -r /path/in/hdfs/directory

4. **Copy Data from NFS to HDFS:**

//To copy data from NFS to HDFS:

hadoop distcp nfs://nfs-server/path/to/source /path/in/hdfs/

Note - Replace nfs-server with the address of your NFS server and /path/to/source with the source path on NFS. /path/in/hdfs/ should be the destination path in HDFS.

//Uploading the sample1, sample2.txt file which contains the matrix multiplication data to HDFS

hadoop fs-mkdir /path/in/hdfs/directory

//Uploading sample1.txt

hadoop fs -copyFromLocal sample1 /path/in/hdfs/directory

//Uploading sample2.txt

hadoop fs -copyFromLocal sample2 /path/in/hdfs/directory

//To find items list

hadoop fs -ls / path/in/hdfs/director

hadoop dfs -cat / path/in/hdfs/directory/sample1.txt

hadoop dfs -cat / path/in/hdfs/directory/sample2.txt

File1: Sample1.txt

M,0,0,1
M,0,1,2
M,1,0,3
M,1,1,4

File2: Sample2.txt

N,0,0,5
N,0,1,6
N,1,0,7
N,1,1,8

//Creating Jar file for the Matrix Multiplication

Jar -cvf MatrixMultiply.jar

//To display implementation of matrix multiplication with hadoop map reduce.

hadoop dfs -cat /opdir/*

MatrixMultiply.java

```
package com.mapreduce.wc;

import org.apache.hadoop.conf.*; import org.apache.hadoop.fs.Path; import
org.apache.hadoop.io.*; import org.apache.hadoop.mapreduce.*; import
org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import
org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  import
org.apache.hadoop.mapreduce.lib.output.FileOutputFormat; import
org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;  public class MatrixMultiply {

public static void main(String[] args) throws Exception { if (args.length != 2) {
System.err.println("Usage: MatrixMultiply <in_dir> <out_dir>");
System.exit(2);
}
Configuration conf = new Configuration(); conf.set("m", "1000"); conf.set("n", "100");
conf.set("p", "1000");

@SuppressWarnings("deprecation") Job job = new Job(conf, "MatrixMultiply");

job.setJarByClass(MatrixMultiply.class);  job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);  job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);  job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);  FileInputFormat.addInputPath(job,
new Path(args[0])); FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.waitForCompletion(true);
}
}
```

Map.java

```
package com.mapreduce.wc;

import org.apache.hadoop.conf.*; import org.apache.hadoop.io.LongWritable; import
org.apache.hadoop.io.Text;
//import org.apache.hadoop.mapreduce.Mapper; import java.io.IOException;

public class Map extends org.apache.hadoop.mapreduce.Mapper<LongWritable, Text,
Text, Text> {
	@Override

public void map(LongWritable key, Text value, Context context) throws IOException,

InterruptedException {
		Configuration conf = context.getConfiguration();		int m =
Integer.parseInt(conf.get("m"));		int p = Integer.parseInt(conf.get("p"));
String line = value.toString();
// (M, i, j, Mij);
```

```
String[] indicesAndValue = line.split(",");

Text outputKey = new Text(); Text outputValue = new Text(); if
(indicesAndValue[0].equals("M")) {
for (int k = 0; k < p; k++) {
outputKey.set(indicesAndValue[1] + "," + k);
// outputKey.set(i,k);
outputValue.set(indicesAndValue[0] + "," + indicesAndValue[2]
+ "," + indicesAndValue[3]); // outputValue.set(M,j,Mij);
context.write(outputKey, outputValue);
}
} else {
        // (N, j, k, Njk);              for (int i = 0; i < m; i++) {
        outputKey.set(i + "," + indicesAndValue[2]); outputValue.set("N," +
indicesAndValue[1] + ","    + indicesAndValue[3]); context.write(outputKey, outputValue);
        }
        }
        }
}
```

Reduce.java

```
package com.mapreduce.wc;

import org.apache.hadoop.io.Text;
// import org.apache.hadoop.mapreduce.Reducer; import java.io.IOException; import
java.util.HashMap; public class Reduce extends
org.apache.hadoop.mapreduce.Reducer<Text, Text, Text, Text> { @Override

public void reduce(Text key, Iterable<Text> values, Context context)

throws IOException, InterruptedException {
String[] value;
//key=(i,k),
//Values = [(M/N,j,V/W),..]
HashMap<Integer, Float> hashA = new HashMap<Integer, Float>(); HashMap<Integer,
Float> hashB = new
HashMap<Integer, Float>(); for (Text val : values) {
value = val.toString().split(","); if (value[0].equals("M")) {
hashA.put(Integer.parseInt(value[1]), Float.parseFloat(value[2])); } else {
hashB.put(Integer.parseInt(value[1]), Float.parseFloat(value[2]));
} }

int n = Integer.parseInt(context.getConfiguration().get("n")); float result = 0.0f; float m_ij;
float n_jk;
for (int j = 0; j < n; j++) {
m_ij = hashA.containsKey(j) ? hashA.get(j) : 0.0f; n_jk = hashB.containsKey(j) ?
hashB.get(j) : 0.0f; result += m_ij * n_jk;
```

```
} if (result != 0.0f) { context.write(null,
new Text(key.toString() + "," + Float.toString(result))); }
}
}
```

1. **Create a Text File on Your Local Machine:**

//Create a text file named data.txt on your local machine and write some text into it.

For example: data.txt
Data Science is like being a detective in the digital age. It's about collecting, analyzing, and making sense of enormous amounts of data - the kind of data you find in everything from social media posts to scientific experiments.

AI, on the other hand, is like having a digital brain, a brain that can learn and think, almost like we humans do. But here's where it gets truly fascinating - Data Science is the key that unlocks AI's potential.

2. **Check the Text in the data.txt File:**

Make sure to verify that the text is written correctly in the data.txt file.

3. **Create a Directory in HDFS:**

//Create a directory in HDFS where you want to store the data.txt file. You can use the following Hadoop command to create a directory:

hdfs dfs -mkdir /user/yourusername/wordcount

Note - Replace /user/yourusername/wordcount with the desired directory path in HDFS.

4. **Upload the data.txt File to HDFS:**

//Hadoop command to copy the file to HDFS:

hdfs dfs -copyFromLocal /path/to/local/data.txt /user/yourusername/wordcount/

5. **Write the MapReduce Program:**

Word Count Mapper and Reducer in Java:

// Mapper Class

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

        public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
        String line = value.toString();
```

```
        StringTokenizer tokenizer = new StringTokenizer(line);

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

// Reducer Class

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

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

Compile your Java code and package it into a JAR file.

6. Run the Word Count Job:

```
hadoop jar YourWordCountJar.jar WordCountDriver -input
/user/yourusername/wordcount/data.txt -output /user/yourusername/wordcount/output/
```

// You can use the hdfs dfs -cat command to view the output on the command line or download the output file to your local machine.

```
$ start-all sh
$ cd hive
$ cd bin

hive> show databases;
Ok
default
hive> create database dbl;
ok
hive> use dbl;
Ok
hive> create table tb1 (name string);
Ok
hive> describe tb1;
name string
hive > LOAD Data Local inpath “input data.txt”
    → overwrite into table tb1:
        1, 09-10-23, 1001, 36000
        2, 10-09-33, 100 , 3000
        3, 10-08-23, 1003, 40000
        4, 08-07-23, 1004, 50000
hive> select count (*) from tb1;
ok
4
hive> select & from tb1;
ok
hive > Altter table tb1 rename to kum;
Ok
hive > Drop Table if exists record;
Ok
```

OUTPUT

default
retail
4

1	09-10-23	1001	25000
2	10-09-23	1002	3000
3	10-09-23	10003	40000
4	08-07-23	1004	50000

Table renamed successfully.

Create:

Create 'education', 'guru'

List:

list TABLE

user
Crawldata
edu
education
emp
fest
test1
Veera

Describe:

decribe 'eduation'

Disable :

disable’ education'

Disable-all:

disable-all <" matching region">

Enable

enable 'education"

Show filters:

Show filters;

Drop:

Drop 'education'

Drop-all:

drop all <"region">

Alter:

alter “education" name = 'guru 99-1', version => 5

OUTPUT

0 rows in 2.3250 seconds
8 rows in 0.8620 Secors
Table eduation enabled
education
colurn FAMFILES DESCRIPTION
{ Name = 'guougg', Bloom FILTER - "row", replication Scope => 'o', version => '1'}
1 row in 0.1010 seconds
0 row In 2.3760
O rows in 1.8360
Column Pre Filter
Time stamp Filter

Step 1: open mysql database
Step 2: Create a table
Step 3 Insert columns
Step 4: Insert values
step 6 : Relate tables K values

```
mysql> create Create Database db1;
mysql> use db1;
mysql> create table Student ( Rollno int, Name string , Marks float);
mysql> Insest into student values (1, 'Kumaran', 93);
mysql> insert into student values (2, 'Sudarshan', 97);
mysql> Alter Table tb1 add column percentage float (4, 2);
mysql> Alter Table th1 drop column percentage;
mysql> Drop table tb1;
```

OUTPUT

Roll no	Name	Marks
1	Kumaran	93
2	Sudarshan	97
3	Jobesh	95