Experiment 5: Hadoop Map-Reduce

Date of Performance:

Date of Submission:

1. Statement of Problem:

Implementation simple algorithim in Map-Reduce: Matrix Multiplication.

Aim: To implement matrix multiplication using the MapReduce programming model on Hadoop.

Objectives:

- To learn the key issues in big data management and its tools and techniques, specifically programming module of Hadoop.
- To understand need of multiple mappers and reducers in analytics.

Scope of the Problem:

The scope of this experiment is to implement matrix multiplication using the **MapReduce programming model in Hadoop**. It focuses on demonstrating how **big data computation** can be efficiently handled by dividing a large task into smaller sub-tasks executed in parallel using multiple mappers and reducers. This implementation helps in understanding the **concept of distributed data processing** and how MapReduce simplifies large-scale analytical computations.

Detailed Scope of Each Objective

1. To learn the key issues in big data management and its tools and techniques, specifically programming module of Hadoop

Scope: This objective focuses on understanding how Hadoop handles large-scale data management and processing through the MapReduce programming framework. It helps learners explore how data is divided, processed, and aggregated efficiently across distributed systems.

- Exploring the concept of **Big Data** and challenges in storing and processing massive datasets.
- Understanding **Hadoop architecture** — NameNode, DataNode, JobTracker, and TaskTracker roles.
- Learning how **MapReduce programming** helps in breaking down complex computations into smaller, parallel tasks.
- Gaining practical exposure to **data flow**, **job execution**, and **fault tolerance** in a distributed environment.

- Real-world relevance: Hadoop MapReduce forms the backbone of many **big data analytics**, **machine learning**, and **ETL** applications in enterprises.

2. To understand the need of multiple mappers and reducers in analytics

Scope: This objective emphasizes the importance of parallelism and task distribution in analytical computations. By using multiple mappers and reducers, Hadoop achieves efficient and scalable data processing.

- Understanding how **mappers** divide input data into smaller chunks for parallel processing.
- Exploring how **reducers** aggregate intermediate results to produce the final output.
- Learning how data is **shuffled and sorted** between the map and reduce phases for optimization.
- Demonstrating **matrix multiplication** as a practical example of parallel data processing using MapReduce.
- Real-world relevance: Parallel computation using multiple mappers and reducers enhances **performance**, **scalability**, and **fault tolerance** in analytics workflows across industries.

2. Theory:

2.1 Definiation?

Apache M is a matrix with element mij in row i and column j.N is a matrix with element njk in row j and column k. P is a matrix = MN with element pik in row i and column k, where pik =∑j mijnjk

Mapper function does not have access to the i, j, and k values directly. An extra MapReduceJob has to be run initially in order to retrieve the values.

The Map Function:

For each element mij of M, emit a key-value pair (i, k), (M, j, mij) for k = 1, 2, . . .number of columns of N. For each element njk of N, emit a key-value pair (i, k), (N, j, njk) for i = 1, 2, . . . number ofrows of M.

The Reduce Function:

For each key (i, k), emit the key-value pair (i, k), pik where, Pik = ∑jmij * njk

The product MN is almost a natural join followed by grouping and aggregation. That is, the natural join of M(I, J, V) and N(J,K,W), having only attribute J in common, would produce tuples (i, j, k, v,w) from each tuple(i, j, v) in M and tuple (j, k,w) in N.

This five-component tuple represents the pair of matrix elements (mij ,njk).What we want instead is the product of these elements, that is, the four-component tuple (i, j, k, v × w), because that represents the product mijnjk. Once we have this relation as the result of one MapReduce operation, we can perform grouping and aggregation, with me and K as the grouping attributes and the sum of V × W as the aggregation. That is, we can implement matrix multiplication as the cascade of two MapReduce operations, as follows.

2.2 Flow of entire matrix multiplication using map-reduce

The input file contains two matrices M and N. The entire logic is divided into two parts:

Step1: Find the product.

Step2: Find sum of the products

Algorithm 1: The Map Function

for *each element* m_{ij} *of* M **do**
  produce $(key, value)$ pairs as $((i,k),(M,j,m_{ij}))$, for $k = 1,2,3,..$ up to the number of columns of N
for *each element* n_{jk} *of* N **do**
  produce $(key, value)$ pairs as $((i,k),(N,j,n_{jk}))$, for $i = 1,2,3,...$ up to the number of rows of M
return *Set of (key, value) pairs that each key,* (i,k)*, has a list with values* (M,j,m_{ij}) *and* (N,j,n_{jk}) *for all possible values of* j

Algorithm 2: The Reduce Function

for *each key (i,k)* **do**
  sort values begin with M by j in $list_M$
  sort values begin with N by j in $list_N$
  multiply m_{ij} and n_{jk} for j_{th} value of each list
  sum up $m_{ij} * n_{jk}$
return $(i,k), \sum_{j=1} m_{ij} * n_{jk}$

3. Implementation

3.1 Environment Setup

To execute the matrix multiplication program using Hadoop MapReduce, the following setup was used:

- **Software Requirements:**
  - Hadoop 2.x version
  - Eclipse IDE
  - Java JDK 8 or above
- **Environment Configuration Steps:**

1. Install and configure **Hadoop** on a Linux-based system.
2. Set up **Java** and configure environment variables (JAVA_HOME, HADOOP_HOME, PATH).
3. Create input and output directories in **HDFS**.
4. Compile and package the Java program using the Hadoop libraries.
5. Run the MapReduce job using Hadoop commands.

3.2 Problem Explanation

Matrix multiplication is a computationally heavy operation that can be parallelized using the **MapReduce** model in Hadoop.

Let:

- **M** be a matrix with element *m_{ij}* (row *i*, column *j*)
- **N** be a matrix with element *n_{jk}* (row *j*, column *k*)

The resultant matrix **P = M × N**, where each element is calculated as: *p_{ik} = Σ_j (m_{ij} × n_{jk})*

In MapReduce:

- The **Mapper** reads elements of both matrices and emits intermediate key-value pairs representing partial products.
- The **Reducer** aggregates values for each key (i, k) and computes the sum of products to generate the final result.

This approach demonstrates **distributed computing**, where multiple mappers and reducers process matrix elements in parallel to achieve scalability and efficiency.

3.3 Execution and Output

Program

matrixmultiplication.java

```
import java.io.IOException; import java.util.*; import java.io.*;import
java.lang.*;

import org.apache.hadoop.fs.Path; import org.apache.hadoop.conf.*; 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 main
```

```
{
public static class Map extends Mapper<LongWritable, Text, Text, Text>
{
public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException
{try
{Configuration conf = context.getConfiguration();
 int m = Integer.parseInt(conf.get("m"));int p = Integer.parseInt(conf.get("p")); String line =
value.toString(); String[] indicesAndValue = line.split(",");Text outputKey = new Text();
Text outputValue = new Text();
if (indicesAndValue[0].equals("A")) {for (int k
= 0; k < p; k++) {
outputKey.set(indicesAndValue[1] + "," + k);
outputValue.set("A," + indicesAndValue[2] + "," + indicesAndValue[3]); context.write(outputKey,
outputValue);
}
} else
{
for (int i = 0; i < m; i++) {
outputKey.set(i + "," + indicesAndValue[2]);
outputValue.set("B," + indicesAndValue[1] + "," + indicesAndValue[3]); context.write(outputKey,
outputValue);
}
}
}
catch(ArrayIndexOutOfBoundsException  e)
{
}
}
}
public static class Reduce extends Reducer<Text, Text, Text, Text> {
public void reduce(Text key, Iterable<Text> values, Context context)
throws IOException, InterruptedException {
String[] value;
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("A")) {
```

```
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 a_ij; float b_jk;
for (int j = 0; j < n; j++) {

 a_ij = hashA.containsKey(j) ? hashA.get(j) : 0.0f; b_jk = hashB.containsKey(j) ? hashB.get(j) : 0.0f;
result += a_ij * b_jk;
    }
if (result != 0.0f) {
context.write(null, new Text(key.toString() + "," + Float.toString(result)));
    }
    }
    }
public static void main(String[] args) throws Exception { Configuration conf = new Configuration();
// A is an m-by-n matrix; B is an n-by-p matrix.
conf.set("m", "2"); conf.set("n", "5");
conf.set("p", "3");
  Job job = new Job(conf, "MatrixMatrixMultiplicationOneStep"); job.setJarByClass(mm.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);
    }
    }
hadoop fs -rm -
r /input hadoop
fs -rm -r /output
hadoop fs -
mkdir /input
hadoop fs -put
input.txt /input
javac    -Xlint    -classpath    /usr/local/hadoop/share/hadoop/common/hadoop-common-
3.21.jar:/usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-core-3.2.1.jar
mm.java jar cvf  mm.jar *.class
hadoop jar mm.jar mm /input/output hadoop fs -cat/output/part-r-00000
```

Input:

k=1 i=1 j=1 ((1, 1), (A, 1, 1))
j=2 ((1, 1), (A, 2, 2))
i=2 j=1 ((2, 1), (A, 1, 3))
j=2 ((2, 1), (A, 2, 4))
k=2 i=1 j=1 ((1, 2), (A, 1, 1))
j=2 ((1, 2), (A, 2, 2))
i=2 j=1 ((2, 2), (A, 1, 3))
j=2 ((2, 2), (A, 2, 4))
i=1 j=1 k=1 ((1, 1), (B, 1, 5))
k=2 ((1, 2), (B, 1, 6))
j=2 k=1 ((1, 1), (B, 2, 7))
k=2 ((1, 2), (B, 2, 8))
i=2 j=1 k=1 ((2, 1), (B, 1, 5))
k=2 ((2, 2), (B, 1, 6))
j=2 k=1 ((2, 1), (B, 2, 7))
k=2 ((2, 2), (B, 2, 8))

Output:

$$\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

4. Conclusion

Thus, the Matrix Multiplication using MapReduce program was successfully implemented. The experiment demonstrated how Hadoop's MapReduce framework distributes data and computations across multiple nodes to perform large-scale data operations efficiently.

It provided insights into parallel data processing, key-value pair transformations, and the coordination of mapper and reducer tasks for producing analytical results in a distributed environment.

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/docs/
3. Lin, J., & Dyer, C. (2010). *Data-Intensive Text Processing with MapReduce.* Morgan & Claypool Publishers.
4. Dean, J., & Ghemawat, S. (2008). *MapReduce: Simplified Data Processing on Large Clusters.* Communications of the ACM, 51(1), 107–113.
5. TutorialsPoint. (2024). *Hadoop MapReduce – Matrix Multiplication Example.* Retrieved from https://www.tutorialspoint.com/hadoop/
6. GeeksforGeeks. (2024). *Matrix Multiplication using Hadoop MapReduce.* Retrieved from https://www.geeksforgeeks.org/

Sign and Remark:

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