Experiment 4: Hadoop Map-Reduce

Date of Performance :

Date of Submission :

1. Statement of Problems:

Aim: Write a Program To Implement Word Count Program Using Mapreduce.

Objective:

- To learn the key issues in big data management and its tools and techniques, specifically programming module of Hadoop.
- To understand the working of map-reduce programming.
- To understand the use of map-reduce for big data analytics.

Software Used: Hadoop 2.7.7, Java 8 (OpenJDK 1.8)

2. Theory:

a) MapReduce

MapReduce is a processing technique and a program model for distributed computing based on java. The MapReduce algorithm contains two important tasks, namely Map andReduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller setof tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job.

The major advantage of MapReduce is that it is easy to scale data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called mappers and reducers. Decomposing a data processing application into mappers and reducers is sometimes nontrivial. But, once we write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change.

MapReduce program executes in three stages, namely map stage, shuffle stage, and reducestage.

1. **Map stage**: The map or mapper's job is to process the input data. Generally the inputdata is in the form of file or directory and is stored in the Hadoop file system

(HDFS). The input file is passed to the mapper function line by line. The mapper processes the data and creates several small chunks of data.

2. **Reduce stage**: This stage is the combination of the **Shuffle** stage and the **Reduce** stage. The Reducer's job is to process the data that comes from the mapper. After processing, it produces a new set of output, which will be stored in the HDFS.

During a MapReduce job, Hadoop sends the Map and Reduce tasks to the appropriateservers in the cluster. The framework manages all the details of data-passing such as issuing tasks, verifying task completion, and copying data around the cluster between the nodes. Most of the computing takes place on nodes with data on local disks that reduces the network traffic. After completion of the given tasks, the cluster collects and reduces the data to form an appropriate result, and sends it back to the Hadoop server.

MAP-REDUCE Working

The MapReduce framework operates on <key, value> pairs, that is, the framework views theinput to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types.

The key and the value classes should be in serialized manner by the framework and hence, need to implement the Writable interface. Additionally, the key classes have to implement the Writable-Comparable interface to facilitate sorting by the framework. Input and Output typesof a MapReduce job: (Input) <k1, v1> -> map -
> <k2, v2>-> reduce -> <k3, v3>(Output).

Algorithm

map(key, value):

// key: document name; value: text of

the documentfor each word w in value:

emit(w, 1)

reduce(key,

values):

// key: a word; value: an iterator

over countsresult = 0

for each count v

in values:result

+= v emit(key,

result)

Program:

**********WordCount.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.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 java.util.StringTokenizer;

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> {
   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));
    }
  }

  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);
  }
}
```

3. Implementation:

3.1 Environment Setup:

Step 1 – Install Java & Hadoop

```
!apt-get install -y openjdk-8-jdk-headless
!wget  https://archive.apache.org/dist/hadoop/common/hadoop-2.7.7/hadoop-2.7.7.tar.gz
!tar -xzf hadoop-2.7.7.tar.gz
!mv hadoop-2.7.7 /usr/local/Hadoop
```

```
--2025-08-12 02:50:08--  https://archive.apache.org/dist/hadoop/common/hadoop-2.7.7/hadoop-2.7.7.tar.gz
Resolving archive.apache.org (archive.apache.org)... 65.108.204.189, 2a01:4f9:1a:a084::2
Connecting to archive.apache.org (archive.apache.org)|65.108.204.189|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 218720521 (209M) [application/x-gzip]
Saving to: 'hadoop-2.7.7.tar.gz.1'

hadoop-2.7.7.tar.gz 100%[==================>] 208.59M  8.61MB/s    in 27s

2025-08-12 02:50:35 (7.78 MB/s) - 'hadoop-2.7.7.tar.gz.1' saved [218720521/218720521]
```

Step 2 – Configure Environment Variables

```
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["HADOOP_HOME"] = "/usr/local/hadoop"
os.environ["PATH"] =
```

f"{os.environ['HADOOP_HOME']}/bin:{os.environ['HADOOP_HOME']}/sbin:" + os.environ["PATH"]

```
[4] import os
    os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
    os.environ["HADOOP_HOME"] = "/usr/local/hadoop"
    os.environ["PATH"] = os.environ["HADOOP_HOME"] + "/bin:" + os.environ["JAVA_HOME"] + "/bin:" + os.environ["PATH"]
```

3.2 Explanation of the problem statement:

The Word Count problem in MapReduce is about counting the frequency of words in a large text dataset by dividing the task into smaller parallel jobs. The input text is first processed by the Mapper, which reads each line, splits it into words, and emits key-value pairs in the form (word, 1). Next, during the Shuffle and Sort phase (handled by the framework), all values belonging to the same word are grouped together. Finally, the Reducer takes each word along with its list of counts, sums them up, and produces the final output (word, total_count). This output, stored in HDFS, gives the frequency of every unique word in the dataset, demonstrating the core principle of MapReduce—parallel processing and aggregation.

3.3 Execution and Output:

Step 1 – Create Input Data

!mkdir input1
%%writefile input1/sample.txt
Hadoop is fast. Hadoop is powerful. MapReduce makes Hadoop useful.

```
=== Input file ===
Hadoop is fast. Hadoop is powerful. MapReduce makes Hadoop useful.
```

Step 2 – WordCount Java Program

```
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.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 java.util.StringTokenizer;

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> {
    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));
    }
  }

  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);
  }
}
```

```
Writing WordCount.java
```

Step 3 – Compile and Create JAR

```
!rm -rf wordcount_classes wordcount.jar output_local
!/usr/lib/jvm/java-8-openjdk-amd64/bin/javac -classpath `hadoop classpath` -d
wordcount_classes WordCount.java
!jar -cvf wordcount.jar -C wordcount_classes/ .
```

```
added manifest
adding: WordCount.class(in = 1491) (out= 812)(deflated 45%)
adding: WordCount$IntSumReducer.class(in = 1739) (out= 741)(deflated 57%)
adding: WordCount$TokenizerMapper.class(in = 1736) (out= 753)(deflated 56%)
```

Step 4 – Run the Hadoop Job

!hadoop jar wordcount.jar WordCount input1 output_local

Step 5 – View the Output

!cat output_local/part-r-00000

```
!cat output_local/part-r-00000

Hadoop  3
MapReduce       1
fast.   1
is      2
makes   1
powerful.       1
useful. 1
```

Output:

```
Hadoop  3
MapReduce       1
fast.   1
is      2
makes   1
powerful.       1
useful. 1
```

4.Conclusion:

The implementation of the Word Count Program using MapReduce provides a comprehensive understanding of the Hadoop framework and its distributed data processing capabilities. Through this experiment, we learned how large volumes of unstructured data can be efficiently processed by dividing tasks into smaller sub-tasks and executing them in parallel across multiple nodes. The project demonstrated the complete working of the MapReduce model, including the Map, Shuffle, and Reduce stages, and showed how data is transformed from raw input to meaningful analytical results.

By executing this program in Cloudera's Hadoop environment using Eclipse, we gained hands-on experience in configuring Hadoop jobs, managing input/output through HDFS, and analyzing output data. This practical exposure enhanced our understanding of big data tools and techniques and the importance of distributed computing in handling large-scale datasets.

Overall, the project successfully achieved its objectives—understanding big data management challenges, exploring the working of MapReduce programming, and applying it for data analytics tasks. It laid a strong foundation for further exploration into advanced big data technologies such as Apache Spark, Hive, and Pig, and reinforced the importance of Hadoop as a key tool in modern data-driven applications.

5.References:

1. Tom White, *Hadoop: The Definitive Guide*, 4th Edition, O'Reilly Media, 2015.
2. Apache Hadoop Documentation, "MapReduce Tutorial," https://hadoop.apache.org/docs/
3. Michael G. Noll, "Writing an Hadoop MapReduce Program in Java," https://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-java/
4. Cloudera Documentation, "Getting Started with Hadoop and MapReduce," https://docs.cloudera.com/
5. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, Vol. 51, No. 1, 2008, pp. 107–113.

Sign and Remark:

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