Name-Nirmal Avhad

Roll No:- 71

Subject :- CL3 ( CI)

Practical 3 Title:- Distributed Application using MapReduce for Text Analysis : a) Character counting in a given text file.

b) Counting no. of occurrences of every word in a given text file.

Problem Statement:

Objective:

1. Implement a MapReduce program under Hadoop for character counting in a given text file.
2. Develop a MapReduce program under Hadoop for counting the number of occurrences of every word in a given text file.

Outcome-

1. Successfully count the number of characters in the given text file using MapReduce.

1. Count the occurrences of every word in the given text file using MapReduce.

.

Software Requirement

* Python (3.x recommended)
* Jupyter Notebook or any Python IDE

Hardware Requirement

A machine with sufficient RAM and processing power for model training (8GB RAM recommended)

Prerequisities-

Counting the number of words in any language is a piece of cake like in C, C++, Python, Java, etc. MapReduce

also uses Java but it is very easy if you know the syntax on how to write it.

It is the basic of MapReduce. You will first learn how to execute this code similar to “Hello World” program in other languages. So here are the steps which show how to write a MapReduce code for Word Count.

What is MapReduce?

MapReduce is a programming model and framework within the Hadoop ecosystem that enables efficient

processing of big data by automatically distributing and parallelizing the computation. It consists of two fundamental tasks: Map and Reduce.

In the Map phase, the input data is divided into smaller chunks and processed independently in parallel across multiple nodes in a distributed computing environment. Each chunk is transformed or “mapped” into key-value pairs by applying a user-defined function. The output of the Map phase is a set of intermediate key-value pairs. The Reduce phase follows the Map phase. It gathers the intermediate key-value pairs generated by the Map tasks, performs data shuffling to group together pairs with the same key, and then applies a user-defined reduction function to aggregate and process the data. The output of the Reduce phase is the final result of the computation.

Map Reduce example allows for efficient processing of large-scale datasets by leveraging parallelism and distributing the workload across a cluster of machines. It simplifies the development of distributed data processing applications by abstracting away the complexities of parallelization, data distribution, and fault tolerance, making it an essential tool for big data processing in the Hadoop ecosystem.

Advantages of using MapReduce

The advantages of using MapReduce are as follows:

* MapReduce can define mapper and reducer in several different languages using Hadoop streaming. • MapReduce facilitates automatic parallelization and distribution, reducing the time required to run programs.
* MapReduce provides fault tolerance by re-executing, writing map output to a distributed file system,

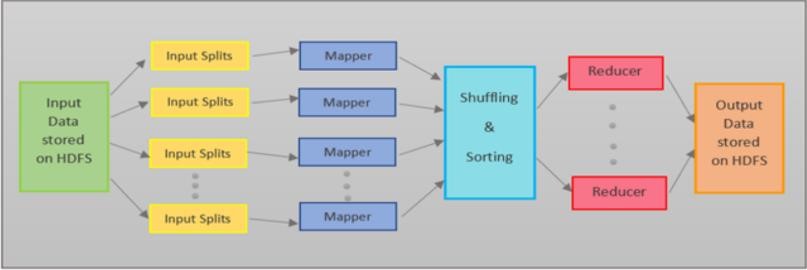
and restarting failed map or reducer tasks.

* Processing of data using MapReduce is a cost-effective solution.
* MapReduce processes large volumes of unstructured data very quickly.
* Using HDFS and HBase security, Map Reduce ensures data security by allowing only approved users to access data stored in the system.
* MapReduce programming utilizes a simple programming model to handle tasks more efficiently and quickly and is easy to learn.
* MapReduce is flexible and works with several Hadoop languages to handle and store data.

MapReduce Architecture

Map Reduce example process has the following phases:

1. Input Splits
2. Mapping
3. Shuffling
4. Sorting
5. Reducing



Input Splits

MapReduce splits the input into smaller chunks called input splits, representing a block of work with a single mapper task. Mapping

The input data is processed and divided into smaller segments in the mapper phase, where the number of mappers is equal to the number of input splits. RecordReader produces a key-value pair of the input splits using TextFormat, which Reducer later uses as input. The mapper then processes these key-value pairs using coding logic to produce an output of the same form. Shuffling

In the shuffling phase, the output of the mapper phase is passed to the reducer phase by removing duplicate values and grouping the values. The output remains in the form of keys and values in the mapper phase. Since shuffling can begin even before the mapper phase is complete, it saves time.

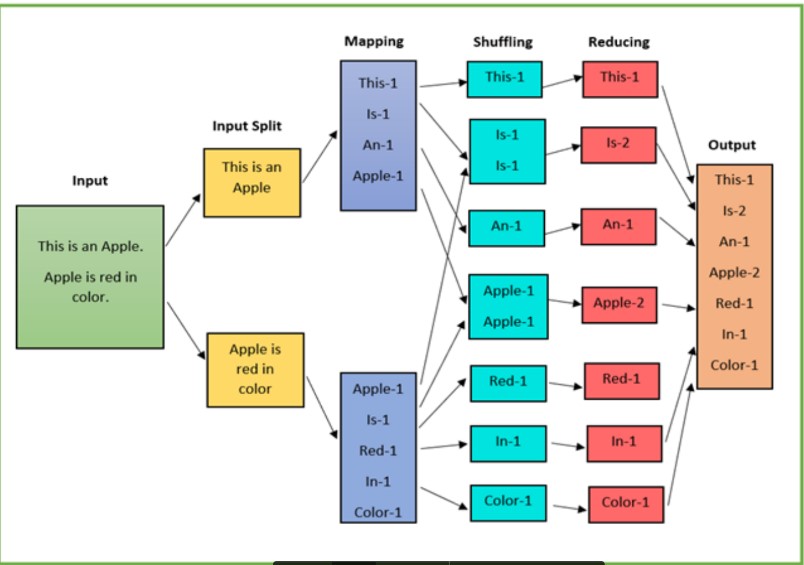
Sorting

Sorting is performed simultaneously with shuffling. The Sorting phase involves merging and sorting the output generated by the mapper. The intermediate key-value pairs are sorted by key before starting the reducer phase, and the values can take any order. Sorting by value is done by secondary sorting.

Reducing

In the reducer phase, the intermediate values from the shuffling phase are reduced to produce a single output value that summarizes the entire dataset. HDFS is then used to store the final output.

Here’s an Map Reduce example to count the frequency of each word in an input text. The text is, “This is an apple. Apple is red in color.”



* The input data is divided into multiple segments, then processed in parallel to reduce processing time. In this case, the input data will be divided into two input splits so that work can be distributed over all the map nodes.
* The Mapper counts the number of times each word occurs from input splits in the form of key-value pairs where the key is the word, and the value is the frequency.
* For the first input split, it generates 4 key-value pairs: This, 1; is, 1; an, 1; apple, 1; and for the second, it generates 5 key-value pairs: Apple, 1; is, 1; red, 1; in, 1; color.
* It is followed by the shuffle phase, in which the values are grouped by keys in the form of key-value pairs. Here we get a total of 6 groups of key-value pairs.
* The same reducer is used for all key-value pairs with the same key.
* All the words present in the data are combined into a single output in the reducer phase. The output shows the frequency of each word.
* Here in the example, we get the final output of key-value pairs as This, 1; is, 2; an, 1; apple, 2; red, 1; in, 1; color, 1.
* The record writer writes the output key-value pairs from the reducer into the output files, and the final output data is by default stored on HDFS.

**Limitations of MapReduce**

Map Reduce example also faces some limitations, and they are as follows:

* MapReduce is a low-level programming model which involves a lot of writing code.
* The batch-based processing nature of MapReduce makes it unsuitable for real-time processing.
* It does not support data pipelining or overlapping of Map and Reduce phases.
* Task initialization, coordination, monitoring, and scheduling take up a large chunk of MapReduce’s execution time and reduce its performance.
* MapReduce cannot cache the intermediate data in memory, thereby diminishing Hadoop’s performance. Conclusion: Thus designed Distributed Application using MapReduce for Text Analysis.

**Code:-**

WC\_Mapper.java

package org.code; import java.io.IOException; import java.util.StringTokenizer; import org.apache.hadoop.io.IntWritable; import org.apache.hadoop.io.LongWritable; import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.MapReduceBase; import org.apache.hadoop.mapred.Mapper; import org.apache.hadoop.mapred.OutputCollector; import org.apache.hadoop.mapred.Reporter; public class WC\_Mapper extends MapReduceBase implements 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,OutputCollector<Text,IntWritable> output, Reporter reporter) throws IOException{

String line = value.toString();

StringTokenizer tokenizer = new StringTokenizer(line); while (tokenizer.hasMoreTokens()){ word.set(tokenizer.nextToken()); output.collect(word, one);}}}

WC\_Runner.java

package org.code; import java.io.IOException; import org.apache.hadoop.fs.Path; import org.apache.hadoop.io.IntWritable; import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.FileInputFormat; import org.apache.hadoop.mapred.FileOutputFormat; import org.apache.hadoop.mapred.JobClient; import org.apache.hadoop.mapred.JobConf; import org.apache.hadoop.mapred.TextInputFormat; import org.apache.hadoop.mapred.TextOutputFormat; public class WC\_Runner {

public static void main(String[] args) throws IOException{ JobConf conf = new JobConf(WC\_Runner.class); conf.setJobName("WordCount"); conf.setOutputKeyClass(Text.class); conf.setOutputValueClass(IntWritable.class); conf.setMapperClass(WC\_Mapper.class); conf.setCombinerClass(WC\_Reducer.class); conf.setReducerClass(WC\_Reducer.class); conf.setInputFormat(TextInputFormat.class); conf.setOutputFormat(TextOutputFormat.class); FileInputFormat.setInputPaths(conf,new Path(args[0]));

FileOutputFormat.setOutputPath(conf,new Path(args[1]));

JobClient.runJob(conf);}}

WC\_Reducer.java

package org.code; import java.io.IOException; import java.util.Iterator;

import org.apache.hadoop.io.IntWritable; import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.MapReduceBase; import org.apache.hadoop.mapred.OutputCollector; import org.apache.hadoop.mapred.Reducer;

import org.apache.hadoop.mapred.Reporter;

public class WC\_Reducer extends MapReduceBase implements Reducer<Text,IntWritable,Text,IntWritable> { public void reduce(Text key, Iterator<IntWritable> values,OutputCollector<Text,IntWritable> output, Reporter reporter) throws IOException { int sum=0;

while (values.hasNext()) {

sum+=values.next().get();

}

output.collect(key,new IntWritable(sum));}}

Output:-

