Experiment - 4

Lab Manual

Aim : "Write a map-reduce program to count the frequencies of words from distributed storage source and understand the phases involved in map-reduce programming."

WordCount Program with Map-Reduce

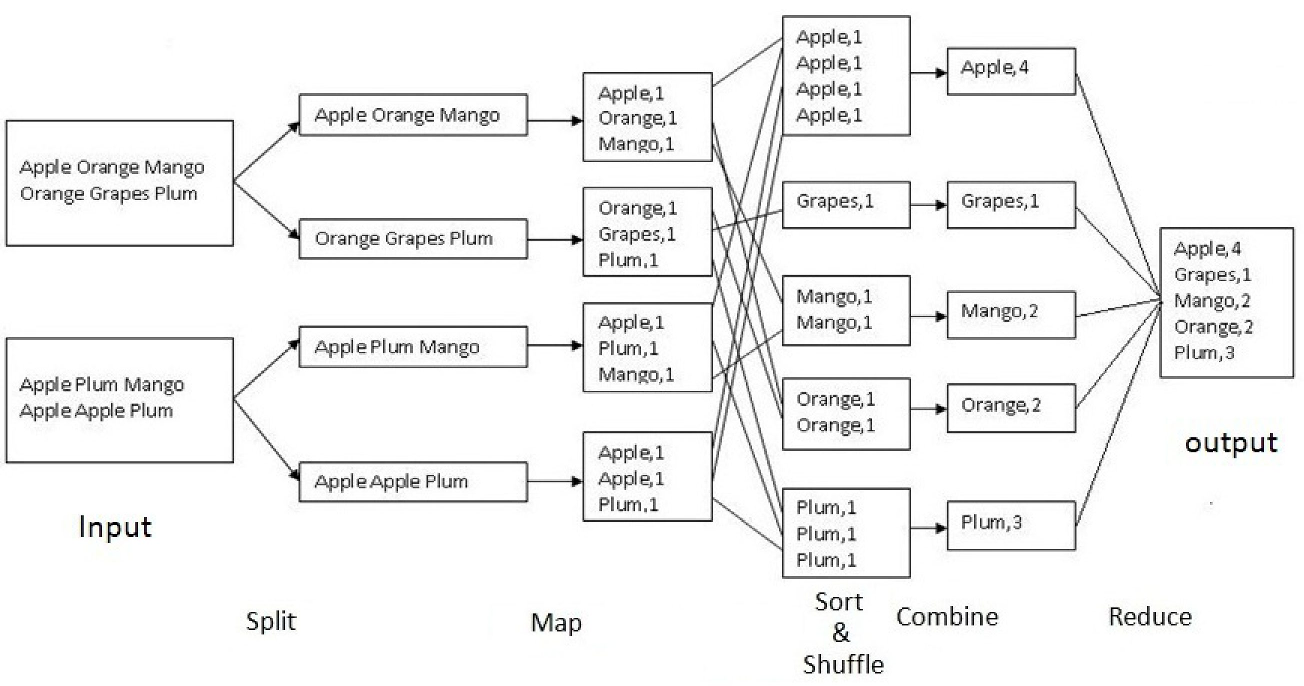
The MapReduce algorithm contains two important tasks, namely Map and Reduce.

The Map task takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key-value pairs).

The Reduce task takes the output from the Map as an input and combines those data tuples (key-value pairs) into a smaller set of tuples.

The reduce task is always performed after the map job.

WordCount Example (Work Flow)



WordCount Program: 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.*;
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 < 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
                {
                        StringTokenizer tokenizer = new StringTokenizer(value.toString());
                        while           (tokenizer.hasMoreTokens()) {
                                word.set(tokenizer.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);
    }
}
```

Explanation
The program consist of 3 classes:

1. Map (i.e. TokenizerMapper) class which extends public class Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT> and implements the Map function.
2. Reduce (i.e. IntSumReducer) class which extends public class Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT> and implements the Reduce function.
3. Driver (i.e. WordCount) class (public static void main() - the entry point)

Mapper Class
Mapper <LongWritable, Text, Text, IntWritable>

The data types provided here are Hadoop specific data types designed for operational efficiency suited for massive parallel and lightning fast read-write operations.

All these data types are based out of java data types itself, for example LongWritable is the equivalent for long in java, IntWritable for int and Text for String.

When we use it as Mapper<LongWritable, Text, Text, IntWritable> , it refers to the data type of input and output key value pairs specific to the mapper or rather the map method, ie Mapper<Input Key Type, Input Value Type, Output Key Type, Output Value Type>.

In our example the input to a mapper is a single line, so this Text (one input line) forms the input value.
The input key would be a long value assigned by default based on the position of Text in input file.

Our output from the mapper is of the format “Word, 1“ hence the data type of our output key value pair is <Text(String), IntWritable(int)>

The functionality of the map method is as follows
1. Create a IntWritable variable ‘one’ with value as 1
2. Convert the input line in Text type to a String

3. Use a StringTokenizer to split the line into words
4. Iterate through each word and a form key value pairs as
a. Assign each word from the tokenizer(of String type) to a Text ‘word’
b. Form key value pairs for each word as <word,one> and write it to the context

Reducer Class
Reducer<Text, IntWritable, Text, IntWritable>

The first two refers to data type of Input Key and Value to the reducer and the last two refers to data type of output key and value.

Our mapper emits output as <apple,1> , <grapes,1> , <apple,1> etc. This is the input for reducer so here the data types of key and value in java would be String and int, the equivalent in Hadoop
would be Text and IntWritable.

Also we get the output as<word, no of occurrences> so the data type of output Key Value would be <Text, IntWritable>

The functionality of the reduce method is as follows
1. Initialize a variable ‘sum’ as 0
2. Iterate through all the values with respect to a key and sum up all of them
3. Write the final result (number of occurrence) for each word (key) to the context.

Start hadoop entities.

start-dfs.sh
start-yarn.sh
start-historyserver.sh

Run jps and verify as below:

```
Jigar-Pandyas-MacBook:BookAnalytics JigarPandya$ jps
70482 NameNode
70837 NodeManager
70646 SecondaryNameNode
70758 ResourceManager
73369 Jps
70554 DataNode
70922 JobHistoryServer
Jigar-Pandyas-MacBook:BookAnalytics JigarPandya$
```

Execute the Program
Assuming environment variables are set appropriately:

Compile WordCount.java and create a jar:

```
$ hadoop com.sun.tools.javac.Main WordCount.java
$ jar cf wc.jar WordCount*.class
```

As hadoop provides an option to generate and provide hadoop classpath as output runtime. We can utilize following command also on linux like platforms to substitute the output and handle classpath.

```
$javac -cp `hadoop classpath` WordCount.java
$jar cf wc.jar WordCount*.class
```

Run the application:

Assuming that:
/projects/wordcount/input - input directory in HDFS contains input text-files

Create using below commands :
hadoop dfs -mkdir -p /projects/wordcount/input

Put some text files within input folder to validate the result later in output folder:
hadoop dfs -put *.txt /projects/wordcount/input
hadoop dfs -chmod -R 775 /projects

or create dummy files as below:
echo "Hello World of Hadoop" > file01.txt
echo "Hello Again. Learning MapReduce programming with Hadoop. Hadoop enables horizontal scalling." > file02.txt

hadoop dfs -put file* .txt /projects/wordcount/input

/projects/wordcount/output - output directory in HDFS. (output directory will be created by program itself). If you are running multiple times, either provide different name of output directory or remove using below command.

hadoop dfs -rm -r /projects/wordcount/output

Run following command to execute program

$ hadoop jar wc.jar WordCount /projects/wordcount/input /projects/wordcount/output

Notice the progress on CLI as well as using GUI (http://localhost:8088/cluster).

When completed the MapReduce task, run the following command to see the output:

$ hadoop dfs –cat /projects/wordcount/output/part-r-00000

```
Jigar-Pandyas-MacBook:WordCount JigarPandya$ hadoop dfs -cat /projects/wordcount/output/part-r-00000
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.

19/06/03 19:10:48 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform..
plicable
Again.	1
Hadoop	2
Hadoop.	1
Hello	2
Learning	1
MapReduce	1
World	1
enables	1
horizontal	1
of	1
programming	1
scalling.	1
with	1
Jigar-Pandyas-MacBook:WordCount JigarPandya$
```

References :

1.https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example:_WordCount_v1.0

Exercise:

1. To practice MapReduce programming in Hadoop using Java Coding.

Step. 1.1: Create a file named 'pages.txt' in local file system. Store line by line content as shown below. Each line data represents number of pages of a sample book.

350
250
150
450
120

Step. 1.2: Put the file from local file system to hdfs with folder named 'input'. Confirm the presence of above data.

Step. 1.3: Write map and reduce functions to split the books into the following two categories:

(a) Big Books

(b) Small Books

Books which have more than 300 pages should be in the big book category. Books which have less than 300 pages should be in the small book category.

Count the number of books in each category.

Store the output as follow as result file within hdfs 'output' folder.

Book Category	Count of the books
"Big Books"	2
"Small Books"	3