Unit 2

Writing MapReduce Programs: A Weather Dataset, Understanding Hadoop API for MapReduce Framework (Old and New), basic programs of Hadoop MapReduce: Driver code, Mapper code, Reducer code, Record Reader, Combiner, Partitioner.

1. Hadoop API for MapReduce Framework (Old and New)

Recently Hadoop new version 2.6.0 has released into Market, Actually Hadoop versions are released in 3 stages 0.x.xx, 1.x.xx and 2.x.x, Up to Hadoop 0.20 All packages are In Old API (Mapred) From Hadoop 0.21 All packages are in New API (MapReduce).

Example of New MapReduce API is ***org.apache.hadoop.mapreduce***
Example of Old MapReduce API is ***org.apache.hadoop.mapred***

Difference	New API	OLD API
Mapper & Reducer	New API uses Mapper and Reducer as Class. So can add a method (with a default implementation) to an abstract class without breaking old implementations of the class	OLD API used Mapper & Reducer as Interface (still exist in New API as well)
Package	new API is in the org.apache.hadoop.mapreduce package	old API can still be found in org.apache.hadoop.mapred.
User Code to commnicate with MapReduce System	use “context” object to communicate with mapReduce system	JobConf, the OutputCollector, and the Reporter object use for communicate with Map reduce System
Control Mapper and Reducer execution	new API allows both mappers and reducers to control the execution flow by overriding the run() method.	Controlling mappers by writing a MapRunnable, but no equivalent exists for reducers.
JOB control	Job control is done through the JOB class in New API	Job Control was done through JobClient (not exists in the new API)
Job Configuration	Job Configuration done through Configuration class via some of the helper methods on Job.	jobconf objet was use for Job configuration.which is extension of Configuration class. java.lang.Object extended by org.apache.hadoop.conf.Configuration extended by org.apache.hadoop.mapred.JobConf

Output file Name	In the new API map outputs are named part-m-nnnnn, and reduce outputs are named part-r-nnnnn (where nnnnn is an integer designating the part number, starting from zero).	in the old API both map and reduce outputs are named part-nnnnn
reduce() method passes values	In the new API, the reduce() method passes values as a java.lang.Iterable	In the Old API, the reduce() method passes values as a java.lang.Iterator

Overview of MapReduce:

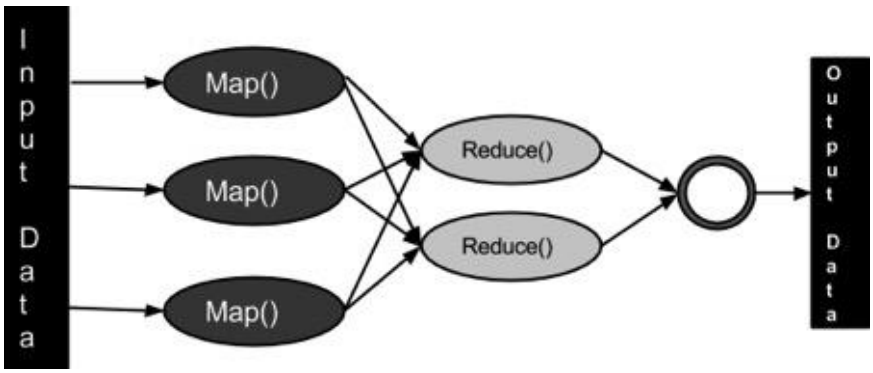
- Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.
- A MapReduce *job* usually splits the input data-set into independent chunks which are processed by the *map tasks* in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the *reduce tasks*. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.
- The MapReduce framework consists of a single master JobTracker and one slave TaskTracker per cluster-node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master.
- The MapReduce framework operates exclusively on `<key, value>` pairs, that is, the framework views the input 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

What is 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 and Reduce.
- ✓ 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 set of 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.

The Algorithm

- Generally MapReduce paradigm is based on sending the computer to where the data resides!
- MapReduce program executes in three stages, namely map stage, shuffle stage, and reduce stage.
    - Map stage : The map or mapper's job is to process the input data. Generally the input data 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.
    - 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 appropriate servers 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.



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

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 types of a MapReduce job: (Input) -> map -> ->reduce -> (Output).

	Input	**Output**
Map	<k1, v1>	list (<k2, v2>)
Reduce	<k2, list(v2)>	list (<k3, v3>)

Hadoop Framework comprises of two main components, namely,

- Hadoop Distributed File System (HDFS) for Data Storage and
- MapReduce for Data Processing.

- A typical Hadoop MapReduce job is divided into a set of Map and Reduce tasks that execute on a Hadoop cluster. The execution flow occurs as follows:
- Input data is split into small subsets of data.
- Map tasks work on these data splits.
- The intermediate input data from Map tasks is then submitted to Reduce task after an intermediate process called 'shuffle'.
- The Reduce task(s) works on this intermediate data to generate the result of a MapReduce Job.

2. Basic programs of Hadoop MapReduce:

1. Driver code,
2. Mapper code,
3. Reducer code,
4. RecordReader,
5. Combiner,
6. Partitioner

MapReduce is the programming model to work on data within the HDFS. The programming language for MapReduce is Java. Hadoop also provides streaming where in other languages could also be used to write MapReduce programs. All data emitted in the flow of a MapReduce Program is in the form of <Key, Value> pairs.

A MapReduce program consists of the following 4 parts:

1. Driver
2. Mapper
3. Reducer
4. Combiner

Driver

The Driver code runs on the client machine and is responsible for building the configuration of the job and submitting it to the Hadoop Cluster. The Driver code will contain the main() method that accepts arguments from the command line.

Some of the common libraries those are included for the Driver class:

1 import ***org.apache.hadoop.fs.Path***;
2 import ***org.apache.hadoop.io.*;***
3 import ***org.apache.hadoop.mapred.*;***

In most cases, the command line parameters passed to the Driver program are the paths to the directory where containing the input files and the path to the output directory. Both these path locations are from the HDFS. The output location should not be present before running the program as it is created after the execution of the program. If the output location already exists the program will exit with an error.

The next step the Driver program should do is to configure the Job that needs to be submitted to the cluster. To do this we create an object of type **JobConf** and pass the name of the Driver class. The JobConf class allows you to configure the different properties for the Mapper, Combiner, Partitioner, Reducer, InputFormat and OutputFormat.

Sample

```
public class MyDriver{
public static void main(String[] args) throws Exception {
```

```
// Create the JobConf object
JobConf conf = new JobConf(MyDriver.class);
// Set the name of the Job
conf.setJobName("SampleJobName");
// Set the output Key type for the Mapper
conf.setMapOutputKeyClass(Text.class);
// Set the output Value type for the Mapper
conf.setMapOutputValueClass(IntWritable.class);
// Set the output Key type for the Reducer
conf.setOutputKeyClass(Text.class);
// Set the output Value type for the Reducer
conf.setOutputValueClass(IntWritable.class);
// Set the Mapper Class
conf.setMapperClass(MyMapper.class);
// Set the Reducer Class
conf.setReducerClass(Reducer.class);
// Set the format of the input that will be provided to the program
conf.setInputFormat(TextInputFormat.class);
// Set the format of the output for the program
conf.setOutputFormat(TextOutputFormat.class);
// Set the location from where the Mapper will read the input
FileInputFormat.setInputPaths(conf, new Path(args[0]));
// Set the location where the Reducer will write the output
FileOutputFormat.setOutputPath(conf, new Path(args[1]));
// Run the job on the cluster
JobClient.runJob(conf);
}
}
```

Mapper

The Mapper code reads the input files as <Key, Value> pairs and emits key value pairs. The Mapper class extends MapReduceBase and implements the Mapper interface. The Mapper interface expects four generics, which define the types of the input and output key/value pairs. The first two parameters define the input key and value types, the second two define the output key and value types.

Some of the common libraries that are included for the Mapper class:

```
public class MyMapper extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable>{
public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>
output, Reporter reporter) throws IOException {
output.collect(key,value);
}
}
}
```

The map() function accepts the key, value, OutputCollector and an Reporter object. The OutputCollector is responsible for writing the intermediate data generated by the Mapper.

Reducer

The Reducer code reads the outputs generated by the different mappers as <Key, Value> pairs and emits key value pairs. The Reducer class extends MapReduceBase and implements the Reducer interface. The Reducer interface expects four generics, which define the types of the input and output key/value pairs. The first two parameters define the intermediate key and value types, the second two define the final output key and value types. The keys are WritableComparables, the values are Writables.

Some of the common libraries that are included for the Reducer class:

```
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
```

Sample

```
public class MyReducer extends MapReduceBase implements Reducer <Text,IntWritable>{
@Override
public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException{
output.collect(key,value);
}
}
}
```

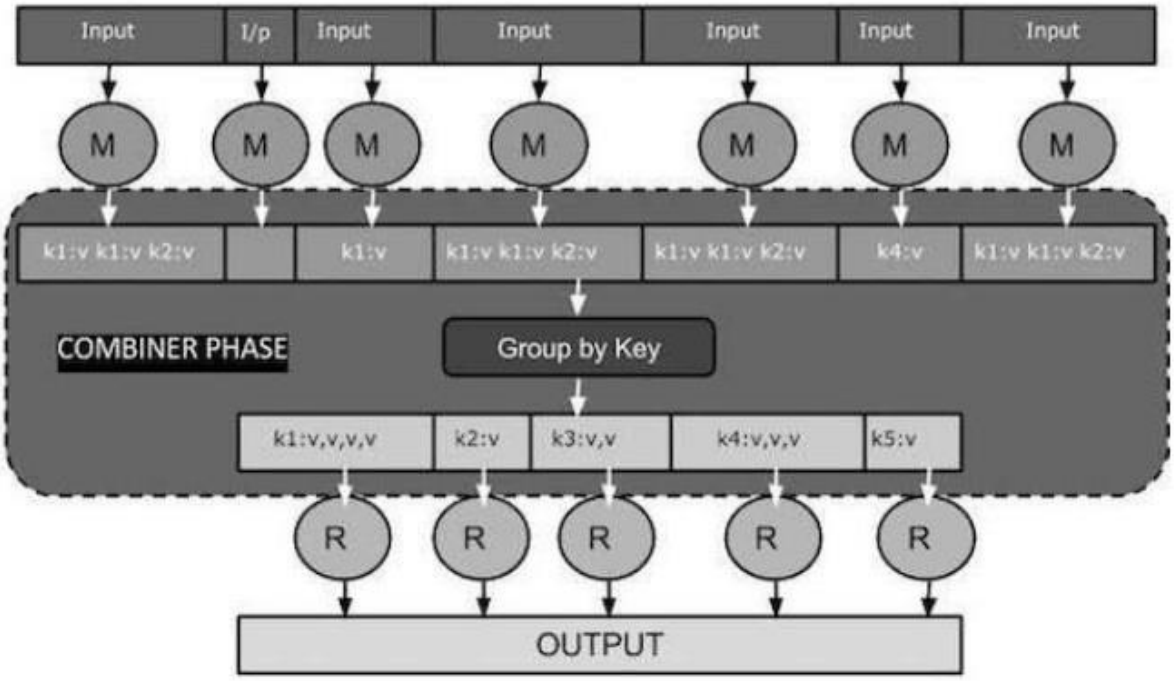
The reduce() function accepts the key, an iterator , OutputCollector and an Reporter object. The OutputCollector is resposible for writing the final output result.

Combiner

A Combiner, also known as a semi-reducer, is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class. The main function of a Combiner is to summarize the map output records with the same key. The output (key-value collection) of the combiner will be sent over the network to the actual Reducer
task as input.

The Combiner class is used in between the Map class and the Reduce class to reduce the volume of data transfer between Map and Reduce. Usually, the output of the map task is large and the data transferred to the reduce task is high.

The following MapReduce task diagram shows the COMBINER PHASE.



A combiner does not have a predefined interface and it must implement the Reducer interface's reduce() method.

A combiner operates on each map output key. It must have the same output key-value types as the Reducer class.

A combiner can produce summary information from a large dataset because it replaces the original Map output.

Although, Combiner is optional yet it helps segregating data into multiple groups for Reduce phase, which makes it easier to process.

MapReduce Combiner Implementation

The following example provides a theoretical idea about combiners. Let us assume we have the following input text file named input.txt for MapReduce.

What do you mean by Object
What do you know about Java
What is Java Virtual Machine
How Java enabled High Performance

The important phases of the MapReduce program with Combiner are discussed below.

Record Reader

This is the first phase of MapReduce where the Record Reader reads every line from the input text file as text and yields output as key-value pairs.

Input − Line by line text from the input file.

Output − Forms the key-value pairs. The following is the set of expected key-value pairs.

<1, What do you mean by Object>
<2, What do you know about Java>
<3, What is Java Virtual Machine>

<4, How Java enabled High Performance>

Map Phase

The Map phase takes input from the Record Reader, processes it, and produces the output as another set of key-value pairs.

Input − The following key-value pair is the input taken from the Record Reader.

<1, What do you mean by Object>
<2, What do you know about Java>
<3, What is Java Virtual Machine>
<4, How Java enabled High Performance>

The Map phase reads each key-value pair, divides each word from the value using StringTokenizer, treats each word as key and the count of that word as value. The following code
snippet shows the Mapper class and the map function.

```
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);
}
}
}
```

Output − The expected output is as follows −

<What,1> <do,1> <you,1> <mean,1> <by,1> <Object,1>
<What,1> <do,1> <you,1> <know,1> <about,1> <Java,1>
<What,1> <is,1> <Java,1> <Virtual,1> <Machine,1>
<How,1> <Java,1> <enabled,1> <High,1> <Performance,1>

Combiner Phase

The Combiner phase takes each key-value pair from the Map phase, processes it, and produces the output as key-value collection pairs.

Input − The following key-value pair is the input taken from the Map phase.

<What,1> <do,1> <you,1> <mean,1> <by,1> <Object,1>
<What,1> <do,1> <you,1> <know,1> <about,1> <Java,1>
<What,1> <is,1> <Java,1> <Virtual,1> <Machine,1>
<How,1> <Java,1> <enabled,1> <High,1> <Performance,1>

The Combiner phase reads each key-value pair, combines the common words as key and values as collection. Usually, the code and operation for a Combiner is similar to that of a Reducer. Following is the code snippet for Mapper, Combiner and Reducer class declaration.

```
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
```

Output − The expected output is as follows –

<What,1,1,1> <do,1,1> <you,1,1> <mean,1> <by,1> <Object,1>
<know,1> <about,1> <Java,1,1,1>
<is,1> <Virtual,1> <Machine,1>
<How,1> <enabled,1> <High,1> <Performance,1>

Reducer Phase

The Reducer phase takes each key-value collection pair from the Combiner phase, processes it, and passes the output as key-value pairs. Note that the Combiner functionality is same as the Reducer.

Input − The following key-value pair is the input taken from the Combiner phase.

<What,1,1,1> <do,1,1> <you,1,1> <mean,1> <by,1> <Object,1>
<know,1> <about,1> <Java,1,1,1>
<is,1> <Virtual,1> <Machine,1>
<How,1> <enabled,1> <High,1> <Performance,1>

The Reducer phase reads each key-value pair. Following is the code snippet for the Combiner.

```
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);
}
}
```

Output − The expected output from the Reducer phase is as follows −

<What,3> <do,2> <you,2> <mean,1> <by,1> <Object,1>
<know,1> <about,1> <Java,3>
<is,1> <Virtual,1> <Machine,1>
<How,1> <enabled,1> <High,1> <Performance,1>

Record Writer

This is the last phase of MapReduce where the Record Writer writes every key-value pair from the Reducer phase and sends the output as text.

Input − Each key-value pair from the Reducer phase along with the Output format.

Output − It gives you the key-value pairs in text format. Following is the expected output.

What 3

do 2
you 2
mean 1
by 1
Object 1
know 1
about 1
Java 3
is 1
Virtual 1
Machine 1
How 1
enabled 1
High 1
Performance 1

MapReduce - Partitioner

A partitioner works like a condition in processing an input dataset. The partition phase takes place after the Map phase and before the Reduce phase.

The number of partitioners is equal to the number of reducers. That means a partitioner will divide the data according to the number of reducers. Therefore, the data passed from a single partitioner is processed by a single Reducer.

Partitioner

A partitioner partitions the key-value pairs of intermediate Map-outputs. It partitions the data using a user-defined condition, which works like a hash function. The total number of partitions is same as the number of Reducer tasks for the job.

Example Program

The following code block counts the number of words in a program.

```
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.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;
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>
{
    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);
    }
}
```