Unit 3

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***

Diffrence	New API	OLD API
Mapper & Reducer	New API useing 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	IN OLD API used Mapper & Reduceer 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 Syaterm	**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*** classin 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***

So This is the Main Differences Between Old and New MR API

2. Basic programs of Hadoop MapReduce:

- Driver code,
- Mapper code,
- Reducer code,
- RecordReader,
- Combiner,
- 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 3 parts :

1. Driver
2. Mapper
3. Reducer

4.

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 that 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 resposible 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,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.

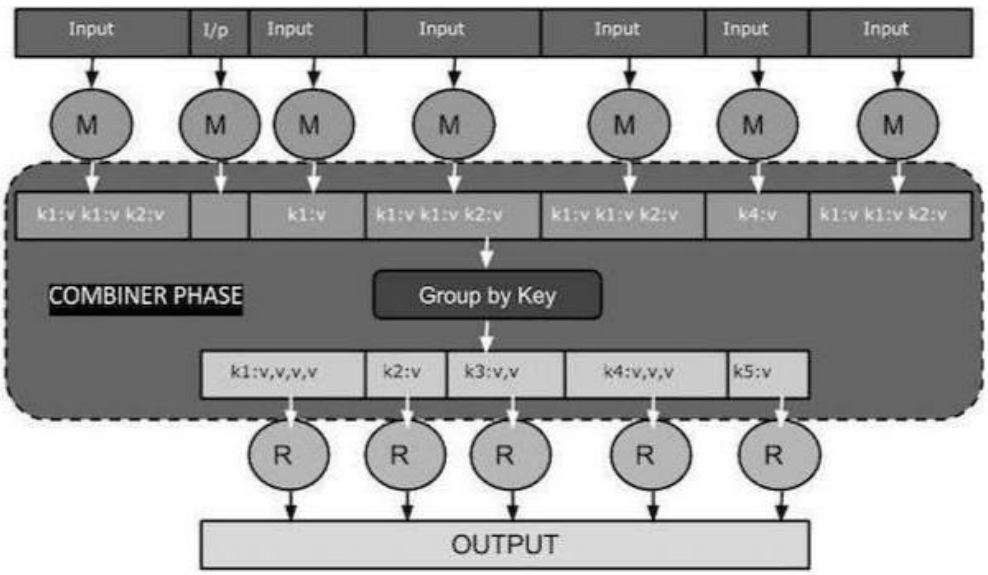
Combiner

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
```

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);
  }
}
```

MapReduce Program: To find average temperature for each year in NCDC data set.

Big data is a framework for storage and processing of data (structured/unstructured). Please check out the program below which draw out results out of semi-structured data from a weather sensor. Its a MapReduce program written in java.

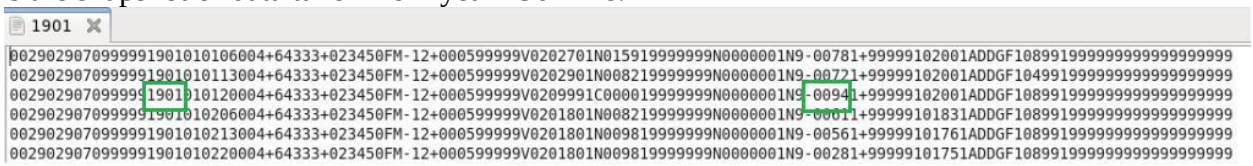
The aim of the program is to find the average temperature in each year of NCDC data.

This program takes a data input of multiple files where each file contains weather data of a particular year. This weather data is shared by NCDC (National Climatic Data Center) and is collected by weather sensors at many locations across the globe. NCDC input data can be downloaded from

https://github.com/tomwhite/hadoop-book/tree/master/input/ncdc/all.

There is a data file for each year. Each data file contains among other things, the year and the temperature information(which is relevant for this program).

Below is the snapshot of the data with year and temperature field highlighted in green box. This is the snapshot of data taken from year 1901 file:



So, in a MapReduce program there are 2 most important phases –

Map Phase and Reduce Phase.

You need to have an understanding of MapReduce concepts so as to understand the intricacies of MapReduce programming. It is one the major component of Hadoop along with HDFS. I will try to include more posts in coming weeks around fundamentals of MapReduce.

Continuing with our current program:

· For writing any MapReduce program, firstly, you need to figure out the data flow, like in this example am taking just the year and temperature information in the map phase and passing it on to the reduce phase. So Map phase in my example is essentially a data preparation phase. Reduce phase on the other hand is more of a data aggregation one.

· Secondly, decide on the types for the key/value pairs—MapReduce program uses lists and (key/value) pairs as its main data primitives. So you need to decide the types for key/value pairs—K1, V1, K2, V2, K3, and V3 for the input, intermediate, and output key/value pairs. In

this example, am taking LongWritable and Text as (K1,V1) for input and Text and IntWritable as both for (K2,V2) and (K3,V3)

Map Phase: I will be pulling out the year and temperature data from the log data that is there in the file, as shown in the above snapshot.

Reduce Phase: The data that is generated by the mapper(s) is fed to the reducer, which is another java program. This program takes all the values associated with a particular key and find the average temperature for that key. So, a key in our case is the year and value is a set of IntWritable objects which represent all the captured temperature information for that year.

I will be writing a java class, each for a Map and Reduce phase and one driver class to create a job with configuration information.

So, in this particular example I will be writing 3 java classes:

AverageMapper.java
AverageReducer.java
AverageDriver.java

Let me share the code of all the 3 classes, along with the explanation of working of each class:

AverageMapper.java

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import java.io.IOException;
public class AverageMapper extends Mapper <LongWritable, Text, Text,
IntWritable> {

public static final int MISSING = 9999;

public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException
        {
                String line = value.toString();
                String year = line.substring(15,19);
                int temperature;
                if (line.charAt(87)=='+')
                        temperature = Integer.parseInt(line.substring(88, 92));
                else
                        temperature = Integer.parseInt(line.substring(87, 92));

                String quality = line.substring(92, 93);
                if(temperature != MISSING && quality.matches("[01459]"))
                context.write(new Text(year),new IntWritable(temperature));
        }
```

```
}
```

Let us get into the details of our AverageMapper class. I need to extend generic class Mapper with four formal data types: input key, input value, output key, output value. The key for the Map phase is the offset of the beginning of the line from the beginning of the file, but as we have no need for it, we can ignore it. The input value would be temperature and output key would be year and output value will be temperature, an integer. The data is fed to the map function one line or record at a time. The map() function converts it into the string and read the year and temperature part from the applicable index value. Also, map() function creates a Context object which is the output object from map(). It contains year value as Text and temperature value as IntWritable.

AverageReducer.java

```
import org.apache.hadoop.mapreduce.*;
import java.io.IOException;

public class AverageReducer extends Reducer <Text, IntWritable,Text, IntWritable >
 {
public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException
        {
        int max_temp = 0;
        int count = 0;
        for (IntWritable value : values)
                {
                        max_temp += value.get();
                        count+=1;
                }
        context.write(key, new IntWritable(max_temp/count));
        }
}
```

Now coming to Reduce Class. Again, four formal data types: input key, input value, output key, output value is specified for this class. The input type and value of reduce function should match output key and value of the map function: Text and IntWritable objects. The reduce() function iterates through all the values and find the sum and count of the values, and finally the average temperature value from that.

AverageDriver.java

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

```
public class AverageDriver
{

        public static void main (String[] args) throws
        Exception {
                if (args.length != 2)
                {
                     System.err.println("Please Enter the input and output
                     parameters"); System.exit(-1);
                }

               Job job = new Job();
               job.setJarByClass(AverageDriver.class);
               job.setJobName("Max temperature");

               FileInputFormat.addInputPath(job,new Path(args[0]));
               FileOutputFormat.setOutputPath(job,new Path (args[1]));

               job.setMapperClass(AverageMapper.class);
               job.setReducerClass(AverageReducer.class);

               job.setOutputKeyClass(Text.class);
               job.setOutputValueClass(IntWritable.class);

               System.exit(job.waitForCompletion(true)?0:1);
        }
}
```

A Job object forms the specification of the job and gives you control over how the job will be run. Hadoop has a special feature of data locality, wherein the code for the program is send to the data instead of other way around. So, Hadoop distributes the jar file of the program across the cluster. we pass the name of the class in setJarByClass() method which hadoop can use to locate the jar file containing this class. We need to specify input and output paths. Input path can specify the file or directory which will be used as an input to the program and output path is a directory which will be created by Reducer. If the directory already exists it leads to an error. Then we specify the map and reduce types to use via setMapperClass() and setReducerClass(). Next we set the output types for the map and reduce functions. waitForCompletion() method submits the job and waits for it to finish. It return 0 or 1, indicating success or failure of the job.