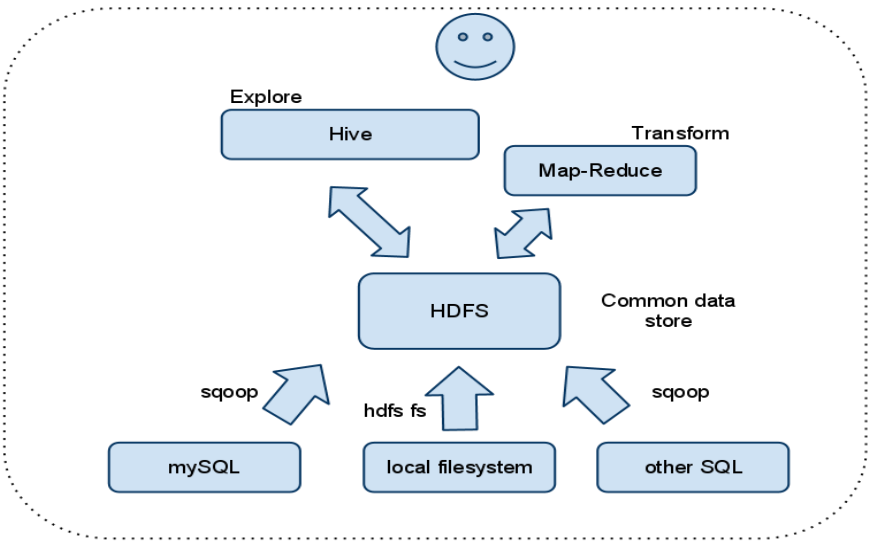
Unit III – HADOOP MAPREDUCE FRAMEWORK

Relationship between Mapreduce Framework and HDFS:

We compare the hadoop software framework as a computer, the mapreduce is the same as software, and the hdfs is the same as hardware.
MapReduce is a framework that is used by Hadoop to process the data residing with HDFS.
HDFS store data in each block which size is 64MB or 128MB. And MapReduce can interaction with HDFS and operates the data in HDFS.
The configuration which we set when we set the environment of hadoop.

The below diagram will show the structure of hadoop.



Map Task (HDFS data localization):

The unit of input for a map task is an HDFS data block of the input file. The map task functions most efficiently if the data block it has to process is available locally on the node on which the task is scheduled. This approach is called HDFS data localization.

An HDFS data locality miss occurs if the data needed by the map task is not available locally. In such a case, the map task will request the data from another node in the cluster: an operation that is expensive and time consuming, leading to inefficiencies and, hence, delay in job completion.

Clients, Data Nodes, and HDFS Storage:

Input data is uploaded to the HDFS file system in either of following two ways:

1. An HDFS client has a large amount of data to place into HDFS.

2. An HDFS client is constantly streaming data into HDFS.

Both these scenarios have the same interaction with HDFS, except that in the streaming case, the client waits for enough data to fill a data block before writing to HDFS. Data is stored in HDFS in large blocks, generally 64 to 128 MB or more in size. This storage approach allows easy parallel processing of data.

HDFS-SITE.XML

<property>

<name>dfs.block.size</name>

<value>134217728</value> ç 128MB Block size

</property>

OR

<property>

<name>dfs.block.size</name>

<value>67108864</value> ç 64MB Block size (Default is this value is not set)

</property>

Block Replication Factor:

During the process of writing to HDFS, the blocks are generally replicated to multiple data nodes for redundancy. The number of copies, or the replication factor, is set to a default of 3 and can be modified by the cluster administrator as below:

HDFS-SITE.XML

<property>

<name>dfs.replication</name>

<value>3</value>

</property>

<property>

When the replication factor is three, HDFS's placement policy is to:

- Put one replica on one node in the local rack,
- Another on a node in a different (remote) rack,
- Last on a different node in the same remote rack.

When a new data block is stored on a data node, the data node initiates a replication process to replicate the data onto a second data node. The second data node, in turn, replicates the block to a third data node, completing the replication of the block.

With this policy, the replicas of a file do not evenly distribute across the racks. One third of replicas are on one node, two thirds of replicas are on one rack, and the other third are evenly distributed across the remaining racks. This policy improves write performance without compromising data reliability or read performance.

Hadoop Data Types Hadoop Online Tutorials

Serialization:

Serialization is the process of converting object data into byte stream data for transmission over a network across different nodes in a cluster or for persistent data storage.

Deserialization:

Deserialization is the reverse process of serialization and converts byte stream data into object data for reading data from HDFS. Hadoop provides *Writables* for serialization and eserialization purpose.Writable and WritableComparable Interfaces To provide mechanisms for serialization and deserialization of data, Hadoop provided two important interfaces Writable and WritableComparable. Writable interface specification is as follows:

```
package org.apache.hadoop.io;
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
public interface Writable
{
void write(DataOutput out) throws IOException;
void readFields(DataInput in) throws IOException;
}
```

WritableComparable interface is subinterface of Hadoop's Writable and Java's Comparable interfaces. and its specification is shown below:

```
1
2
3
public interface WritableComparable extends Writable, Comparable
{}
```

The standard java.lang.*Comparable* Interface contains single method compareTo() method for comparing the operators passed to it.

```
public interface Comparable
{
public int compareTo(Object obj);
}
```

The compareTo() method returns 1, 0 , or 1 depending on whether the compared object is less than, equal to, or greater than the current object.

The above two interfaces are provided in org.apache.hadoop.io package

Constraints on Keyvalues in Mapreduce

Hadoop data types used in Mapreduce for key or value fields must satisfy two constraints.
Any data type used for a Value field in mapper or reducer input/output must implement

Writable Interface.
Any data type used for a Key field in mapper or reducer input/output must implement WritableComparable interface along with Writable interface to compare the keys of this type with each other for sorting purposes
.
Writable Classes – Hadoop Data Types
Hadoop provides classes that wrap the Java primitive types and implement the *WritableComparable* and *Writable* Interfaces. They are provided in the org.apache.hadoop.io package.
All the Writable wrapper classes have a get() and a set() method for retrieving and storing the wrapped value.

Primitive Writable Classes
These are Writable Wrappers for Java primitive data types and they hold a single primitive value that can be set either at construction or via a setter method.
All these primitive writable wrappers have get() and set() methods to read or write the wrapped value. Below is the list of primitive writable data types available in Hadoop.

- BooleanWritable
- ByteWritable
- IntWritable
- VIntWritable
- FloatWritable
- LongWritable
- VLongWritable
- DoubleWritable

In the above list VIntWritable and VLongWritable are used for variable length Integer types and variable length long types respectively.
Serialized sizes of the above primitive writable data types are same as the size of actual java data type. So, the size of IntWritable is 4 bytes and LongWritable is 8 bytes.

Array Writable Classes
Hadoop provided two types of array writable classes, one for *singledimensional* and another for *twodimensional* arrays. But the elements of these arrays must be other writable objects like IntWritable or LongWritable only but not the java native data types like int or float.

- ArrayWritable
- TwoDArrayWritable

Map Writable Classes

Hadoop provided below MapWritable data types which implement java.util.Map interface

- AbstractMapWritable – This is abstract or base class for other MapWritable classes.
- MapWritable – This is a general purpose map mapping Writable keys to Writable values.
- SortedMapWritable – This is a specialization of the MapWritable class that also implements the SortedMap interface.

Other Writable Classes

- NullWritable

NullWritable is a special type of Writable representing a null value. No bytes are read or written when a data type is specified as NullWritable. So, in Mapreduce, a key or a value can be declared as a NullWritable when we don't need to use that field.

- ObjectWritable

This is a generalpurpose
generic object wrapper which can store any objects like Java primitives,
String, Enum, Writable, null, or arrays.

- Text

Text can be used as the Writable equivalent of java.lang.String and It's max size is 2 GB. Unlike java's String data type, Text is mutable in Hadoop.

- BytesWritable

BytesWritable is a wrapper for an array of binary data.

- GenericWritable

It is similar to ObjectWritable but supports only a few types. User need to subclass this GenericWritable class and need to specify the types to support. Example Program to Test Writables

Lets write a WritablesTest.java program to test most of the data types mentioned above in this post with get(), set(), getBytes(), getLength(), put(), containsKey(), keySet() methods.

```
import org.apache.hadoop.io.* ;
import java.util.* ;
public class WritablesTest
{
public static class TextArrayWritable extends ArrayWritable
{
public TextArrayWritable()
{
super(Text.class) ;
}
}
public static class IntArrayWritable extends ArrayWritable
{
public IntArrayWritable()
{
super(IntWritable.class) ;
}
}
public static void main(String[] args)
{
IntWritable i1 = new IntWritable(2) ;
IntWritable i2 = new IntWritable() ;
i2.set(5);
IntWritable i3 = new IntWritable();
i3.set(i2.get());
System.out.printf("Int Writables Test I1:%d , I2:%d , I3:%d", i1.get(), i2.get(), i3.get()) ;
BooleanWritable bool1 = new BooleanWritable() ;
bool1.set(true);
ByteWritable byte1 = new ByteWritable( (byte)7) ;
System.out.printf("\n Boolean Value:%s Byte Value:%d", bool1.get(), byte1.get()) ;
Text t = new Text("hadoop");
Text t2 = new Text();
t2.set("pig");
System.out.printf("\n t: %s, t.legth: %d, t2: %s, t2.length: %d \n", t.toString(), t.getLength(),
t2.getBytes(), t2.getBytes().length);
ArrayWritable a = new ArrayWritable(IntWritable.class) ;
a.set( new IntWritable[]{ new IntWritable(10), new IntWritable(20), new IntWritable(30)}) ;
ArrayWritable b = new ArrayWritable(Text.class) ;
b.set( new Text[]{ new Text("Hello"), new Text("Writables"), new Text("World !!!")}) ;
for (IntWritable i: (IntWritable[])a.get())
System.out.println(i) ;
for (Text i: (Text[])b.get())
```

```
System.out.println(i) ;
IntArrayWritable ia = new IntArrayWritable() ;
ia.set( new IntWritable[]{ new IntWritable(100), new IntWritable(300), new IntWritable(500)}) ;
IntWritable[] ivalues = (IntWritable[])ia.get() ;
for (IntWritable i : ivalues)
System.out.println(i);
MapWritable m = new MapWritable() ;
IntWritable key1 = new IntWritable(1) ;
NullWritable value1 = NullWritable.get() ;
m.put(key1, value1) ;
m.put(new VIntWritable(2), new LongWritable(163));
m.put(new VIntWritable(3), new Text("Mapreduce"));
System.out.println(m.containsKey(key1)) ;
System.out.println(m.get(new VIntWritable(3))) ;
m.put(new LongWritable(1000000000), key1) ;
Set<Writable> keys = m.keySet() ;
for(Writable w: keys)
System.out.println(m.get(w)) ;
}
}
```

MAPREDUCE - COMBINERS

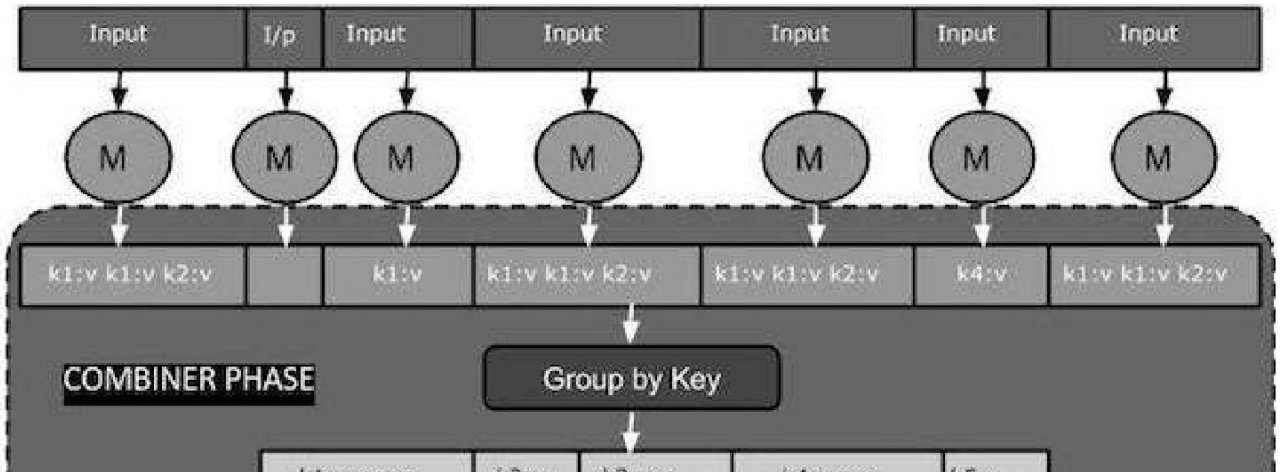
A Combiner, also known as a semi-reducer, is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class.

The main function of a Combiner is to summarize the map output records with the same key. The output key –valuecollection of the combiner will be sent over the network to the actual Reducer task as input.

Combiner

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.



How Combiner Works?

Here is a brief summary on how MapReduce Combiner works –

- 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);
   }
}
```

Save the above program as WordCount.java. The compilation and execution of the program is given below.

Compilation and Execution

Let us assume we are in the home directory of Hadoop user forexample ,/home hadoop/. Follow the steps given below to compile and execute the above program.

Step 1 – Use the following command to create a directory to store the compiled java classes.

```
$ mkdir units
```

Step 2 – Download Hadoop-core-1.2.1.jar, which is used to compile and execute the MapReduce program.

You can download the jar from mvnrepository.com.

Let us assume the downloaded folder is /home/hadoop/.

Step 3 – Use the following commands to compile the WordCount.java program and to create a jar for the program.

```
$ javac -classpath hadoop-core-1.2.1.jar -d units
WordCount.java $ jar -cvf units.jar -C units/ .
```

Step 4 – Use the following command to create an input directory in HDFS.

```
$HADOOP_HOME/bin/hadoop fs -mkdir input_dir
```

Step 5 – Use the following command to copy the input file named input.txt in the input directory of HDFS.

```
$HADOOP_HOME/bin/hadoop fs -put /home/hadoop/input.txt input_dir
```

Step 6 – Use the following command to verify the files in the input directory.

```
$HADOOP_HOME/bin/hadoop fs -ls input_dir/
```

Step 7 – Use the following command to run the Word count application by taking input files from the input directory.

```
$HADOOP_HOME/bin/hadoop jar units.jar hadoop.ProcessUnits input_dir output_dir
```

Wait for a while till the file gets executed. After execution, the output contains a number of input splits, Map tasks, and Reducer tasks.

Step 8 – Use the following command to verify the resultant files in the output folder.

```
$HADOOP_HOME/bin/hadoop fs -ls output_dir/
```

Step 9 – Use the following command to see the output in Part-00000 file. This file is generated by HDFS.

```
$HADOOP_HOME/bin/hadoop fs -cat output_dir/part-00000
```

Following is the output generated by the MapReduce program.

```
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. Let us take an example to understand how the partitioner works.

MapReduce Partitioner Implementation

For the sake of convenience, let us assume we have a small table called Employee with the following data. We will use this sample data as our input dataset to demonstrate how the partitioner works.

Id	Name	Age	Gender	Salary
1201	gopal	45	Male	50,000
1202	manisha	40	Female	50,000
1203	khalil	34	Male	30,000
1204	prasanth	30	Male	30,000
1205	kiran	20	Male	40,000
1206	laxmi	25	Female	35,000

1207	bhavya	20	Female	15,000
1208	reshma	19	Female	15,000
1209	kranthi	22	Male	22,000
1210	Satish	24	Male	25,000
1211	Krishna	25	Male	25,000

1212	Arshad	28	Male	20,000
1213	lavanya	18	Female	8,000

We have to write an application to process the input dataset to find the highest salaried employee by gender in different age groups (for example, below 20, between 21 to 30, above 30).

Input Data

The above data is saved as input.txt in the "/home/hadoop/hadoopPartitioner" directory and given as input.

1201	gopal	45	Male	50000
1202	manisha	40	Female	51000
1203	khaleel	34	Male	30000
1204	prasanth	30	Male	31000
1205	kiran	20	Male	40000
1206	laxmi	25	Female	35000
1207	bhavya	20	Female	15000
1208	reshma	19	Female	14000
1209	kranthi	22	Male	22000
1210	Satish	24	Male	25000
1211	Krishna	25	Male	26000
1212	Arshad	28	Male	20000
1213	lavanya	18	Female	8000

Based on the given input, following is the algorithmic explanation of the program.

Map Tasks

The map task accepts the key-value pairs as input while we have the text data in a text file. The input for this

map task is as follows –

Input – The key would be a pattern such as “any special key + filename + line number” (example: key = @input1) and the value would be the data in that line (example: value = 1201 \t gopal \t 45 \t Male \t 50000).

Method – The operation of this map task is as follows –

- Read the value (record data), which comes as input value from the argument list in a string.
- Using the split function, separate the gender and store in a string variable.

```
String[] str = value.toString().split("\t", -3);
String gender=str[3];
```

- Send the gender information and the record data value as output key-value pair from the map task to the partition task.

```
context.write(new Text(gender), new Text(value));
```

- Repeat all the above steps for all the records in the text file.

Output – You will get the gender data and the record data value as key-value pairs.

Partitioner Task

The partitioner task accepts the key-value pairs from the map task as its input. Partition implies dividing the data into segments. According to the given conditional criteria of partitions, the input key-value paired data can be divided into three parts based on the age criteria.

Input – The whole data in a collection of key-value pairs.

key = Gender field value in the record.

value = Whole record data value of that gender.

Method – The process of partition logic runs as follows.

- Read the age field value from the input key-value pair.

```
String[] str = value.toString().split("\t");
int age = Integer.parseInt(str[2]);
```

- Check the age value with the following conditions.
  - Age less than or equal to 20
  - Age Greater than 20 and Less than or equal to 30.
  - Age Greater than 30.

```
if(age<=20)
{
   return 0;
}
else if(age>20 && age<=30)
{
   return 1 % numReduceTasks;
}
else
{
   return 2 % numReduceTasks;
}
```

Output – The whole data of key-value pairs are segmented into three collections of key-value pairs. The Reducer works individually on each collection.

Reduce Tasks

The number of partitioner tasks is equal to the number of reducer tasks. Here we have three partitioner tasks and hence we have three Reducer tasks to be executed.

Input – The Reducer will execute three times with different collection of key-value pairs.

key = gender field value in the record.

value = the whole record data of that gender.

Method – The following logic will be applied on each collection.

- Read the Salary field value of each record.

```
String [] str = val.toString().split("\t", -3);
Note: str[4] have the salary field value.
```

- Check the salary with the max variable. If str[4] is the max salary, then assign str[4] to max, otherwise skip the step.

```
if(Integer.parseInt(str[4])>max)
{
   max=Integer.parseInt(str[4]);
}
```

- Repeat Steps 1 and 2 for each key collection (Male & Female are the key collections). After executing these three steps, you will find one max salary from the Male key collection and one max salary from the Female key collection.

```
context.write(new Text(key), new IntWritable(max));
```

Output – Finally, you will get a set of key-value pair data in three collections of different age groups. It contains the max salary from the Male collection and the max salary from the Female collection in each age

group respectively.

After executing the Map, the Partitioner, and the Reduce tasks, the three collections of key-value pair data are stored in three different files as the output.

All the three tasks are treated as MapReduce jobs. The following requirements and specifications of these jobs should be specified in the Configurations –

- Job name
- Input and Output formats of keys and values
- Individual classes for Map, Reduce, and Partitioner tasks

```
Configuration conf = getConf();

//Create Job
Job job = new Job(conf, "topsal");
job.setJarByClass(PartitionerExample.class);

// File Input and Output paths
FileInputFormat.setInputPaths(job, new Path(arg[0]));
FileOutputFormat.setOutputPath(job,new Path(arg[1]));

//Set Mapper class and Output format for key-value
pair. job.setMapperClass(MapClass.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(Text.class);

//set partitioner statement
job.setPartitionerClass(CaderPartitioner.class);

//Set Reducer class and Input/Output format for key-value
pair. job.setReducerClass(ReduceClass.class);

//Number of Reducer tasks.
job.setNumReduceTasks(3);

//Input and Output format for data
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
```

Example Program

The following program shows how to implement the partitioners for the given criteria in a MapReduce program.

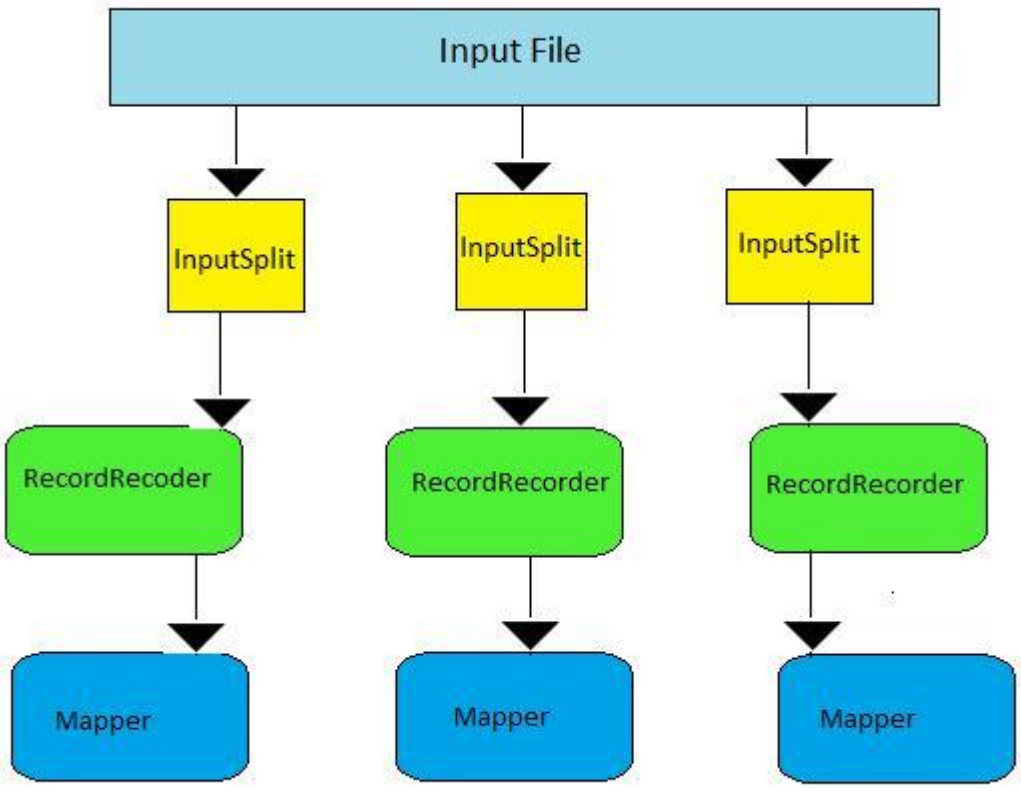
```
package partitionerexample;

import java.io.*;
```

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.fs.*;

import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.mapreduce.lib.output.*;

import org.apache.hadoop.util.*;

public class PartitionerExample extends Configured implements Tool
{
   //Map class

   public static class MapClass extends Mapper<LongWritable,Text,Text,Text>
   {
      public void map(LongWritable key, Text value, Context context)
      {
         try{
            String[] str = value.toString().split("\t", -
            3); String gender=str[3];
            context.write(new Text(gender), new Text(value));
         }
         catch(Exception e)
         {
            System.out.println(e.getMessage());
         }
      }
   }

   //Reducer class

   public static class ReduceClass extends Reducer<Text,Text,Text,IntWritable>
   {
       public int max = -1;
       public void reduce(Text key, Iterable <Text> values, Context context) throws
 IOException, InterruptedException
       {
          max = -1;

         for (Text val : values)
         {
            String [] str = val.toString().split("\t", -
            3); if(Integer.parseInt(str[4])>max)
            max=Integer.parseInt(str[4]);
         }

         context.write(new Text(key), new IntWritable(max));
      }
   }

   //Partitioner class
```

```
public static class CaderPartitioner
extends Partitioner < Text, Text >
{
   @Override
   public int getPartition(Text key, Text value, int numReduceTasks)
   {
      String[] str = value.toString().split("\t");
      int age = Integer.parseInt(str[2]);

      if(numReduceTasks == 0)
      {
         return 0;
      }

      if(age<=20)
      {
         return 0;
      }
      else if(age>20 && age<=30)
      {
         return 1 % numReduceTasks;
      }
      else
      {
         return 2 % numReduceTasks;
      }
   }
}

@Override
public int run(String[] arg) throws Exception
{
   Configuration conf = getConf();

   Job job = new Job(conf, "topsal");
   job.setJarByClass(PartitionerExample.class);

   FileInputFormat.setInputPaths(job, new Path(arg[0]));
   FileOutputFormat.setOutputPath(job,new Path(arg[1]));

   job.setMapperClass(MapClass.class);

   job.setMapOutputKeyClass(Text.class);
   job.setMapOutputValueClass(Text.class);

   //set partitioner statement

   job.setPartitionerClass(CaderPartitioner.class);
   job.setReducerClass(ReduceClass.class);
   job.setNumReduceTasks(3);
   job.setInputFormatClass(TextInputFormat.class);

   job.setOutputFormatClass(TextOutputFormat.class);
```

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

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

    public static void main(String ar[]) throws Exception
    {
        int res = ToolRunner.run(new Configuration(), new
        PartitionerExample(),ar); System.exit(0);
    }
}
```

Hadoop Input Format



Input Splits, Input Formats and Record Reader:

On job startup, each input file is broken into splits and each map processes a single split. Each split is further divided into records of key/value pairs which are processed by map tasks one record at a time.

To get split details of an input file, Hadoop provides an InputSplit class in org.apache.hadoop.mapreduce package and its implementation is as follows.

```
public abstract class InputSplit
{
public abstract long getLength() throws IOException, InterruptedException; public
abstract String[] getLocations() throws IOException, InterruptedException;
}
```

From the above two methods, programmer can get length of a split and storage locations. A good input split size is equal to the HDFS block size. But if the splits are too smaller than the default HDFS block size, then managing splits and creation of map tasks becomes an overhead than the job execution time.

But these file splits need not be taken care by Mapreduce programmer because Hadoop provides InputFormat class in org.apache.hadoop.mapreduce package for the below two responsibilities.

To provide details on how to split an input file into the splits.

To create a RecordReader class that will generate the series of key/value pairs from a split.

To meet these two requirements, Hadoop provides below implementation for InputFormat class with two methods.

```
public abstract class InputFormat<K, V>
{
public abstract List<InputSplit> getSplits(JobContext context) throws IOException,
InterruptedException;
public abstract RecordReader<K, V> createRecordReader(InputSplit split,
TaskAttemptContext context) throws IOException, InterruptedException;
}

public abstract class RecordReader<Key, Value> implements Closeable
{
public abstract void initialize(InputSplit split, TaskAttemptContext context) ;
public abstract boolean nextKeyValue() throws IOException, InterruptedException ;
public abstract Key getCurrentKey() throws IOException, InterruptedException ;
public abstract Value getCurrentValue() throws IOException, InterruptedException ;
public abstract float getProgress() throws IOException, InterruptedException ;
public abstract close() throws IOException ;
}
```

Thus, Record reader creates key/value pairs from input splits and writes on to Context, which will be shared with Mapper class. Mapper class's run() method retrieves these key/value pairs from context by calling getCurrentKey() and getCurrentValue() methods and passes onto map() method for further processing of the record.

Mapper's run() method:

```
public void run(Context context) throws IOException, InterruptedException
{
  setup(context);
  while (context.nextKeyValue())
 {
  map(context.getCurrentKey(), context.getCurrentValue(), context);
 }
 cleanup(context);
}
```

Builtin Hadoop Input Formats:

Hadoop provided some built in InputFormat implementations in the org.apache.hadoop.mapreduce.lib.input package:

FileInputFormat: Base class for all filebased InputFormat implementations.

Some of the important sub classes of the FileInputFormat class are:

TextInputFormat :

The default InputFormat class when no other class is specified. It treats the input files as text files.

KeyValueTextInputFormat :

An InputFormat for plain text files. Files are broken into lines. Each line is divided into key and value parts by a separator byte. If no such a byte exists, the key will be the entire line and value will be empty.

FixedLengthInputFormat :

An input format to read input files with fixed length records. These need not be text files and can be binary files. Users must configure the record length property by calling:

FixedLengthInputFormat.setRecordLength(conf, recordLength);

NLineInputFormat :

It splits N lines of input as one split which will be fed to a single map task. It can be used in applications, that splits the input file such that by default, one line is fed as a value to one map task, and key is the offset. i.e. (k,v) is (LongWritable, Text).

CombineFileInputFormat :

This input file format is suitable for processing huge number of small files. CombineFileInputFormat packs many small files into each split so that each mapper has more to process. Thus it can improve the efficiency of mapreduce job by making less number of map tasks to process huge number of small files.

MultiFileInputFormat :

An abstract InputFormat class that returns MultiFileSplit's in getSplits() method from the files under the input paths.

SequenceFileInputFormat :

Hadoop specific Binary file format for efficient file processing.

SequenceFileAsTextInputFormat :

SequenceFileAsTextInputFormat is a sub class of SequenceFileInputFormat. This class is similar to SequenceFileInputFormat, except it generates SequenceFileAsTextRecordReader which converts the input keys and values to their String forms by calling toString() method.

SequenceFileAsBinaryInputFormat :

SequenceFileAsBinaryInputFormat is another sub class of SequenceFileInputFormat. It is an input format for reading keys, values from Sequence Files in binary (raw) format.

MultipleInputs :

DBInputFormat :

A InputFormat that reads input data from an SQL table. DBInputFormat emits LongWritables containing the record number as key and DBWritables as value. The SQL query, and input class can be using one of the two setInput() methods.

Hadoop Output Format:

Hadoop provides output formats that corresponding to each input format. All hadoop output formats must implement the interface org.apache.hadoop.mapreduce.OutputFormat.

OutputFormat describes the outputspecification for a MapReduce job. Based on Output specification,

1.Mapreduce job checks that the output directory doesn't already exist.

2.OutputFormat provides the RecordWriter implementation to be used to write out the output files of the job.

These two requirements of the OutputFormat are accomplished with below two methods in the interface.

1.public abstract void checkOutputSpecs(JobContext context) throws IOException, InterruptedException{}

This method checks that output directory doesn't exist already and throws an exception when it already exists, so that output is not overwritten.

2.public abstract RecordWriter<K,V> getRecordWriter(TaskAttemptContext context) throws IOException, InterruptedException{}

This method Gets the RecordWriter for the given task.

org.apache.hadoop.mapreduce.RecordWriter<K,V> class implementations are used to write the output <key, value> pairs to an output file.

BuiltIn Hadoop Output Formats

Hadoop provided some built in InputFormat implementations in the org.apache.hadoop.mapreduce.lib.output package:

FileOutputFormat Base class for all file based OutputFormat implementations.

Some of the important sub classes of the FileOutputFormat class are:

TextOutputFormat

The default output format provided by hadoop is TextOuputFormat and it writes records as lines of text. If file output format is not specified explicitly, then text files are created as output files. Output Keyvalue pairs can be of any format because *TextOutputFormat* converts these into strings with toString() method. Output keyvalue pairs are tab delimited by default.

For reading these output text files as input, KeyValueTextInputFormat is best suitable, since it breaks input lines into key value pairs based on a separator character.

SequenceFileOutputFormat

This output format class is useful to write out sequence files which is a best option when the output files need to be fed into another mapreduce jobs as input files, since these are compressed and compact.

SequenceFileAsBinaryOutputFormat

SequenceFileAsBinaryOutputFormat is a direct subclass of SequenceFileOutputFormat and it is counter part for SequenceFileAsBinaryInputFormat. It writes keys and values to sequence Files in binary format.

MapFileOutputFormat

It is also a direct subclass of FileOutputFormat and it is used to write output as Map files.

MultipleOutputs

The MultipleOutputs class is used to write output data to multiple outputs. Below are the two main use cases of MultipleOutputs.

1. Job output can be written to additional outputs other than the default output. Each additional output, or named output, may be configured with its own *OutputFormat*, with its own key class and value class.

2. Write data to different files provided by user

MultipleOutputs supports counters to count the number records written to each output name. But these are disabled by default.

MultipleOutputFormat

It is an abstract class which is extended by *MultipleTextOutputFormat* and *MultipleSequenceFileOutputFormat*. This abstract class extends the FileOutputFormat,

The main advantage of this format is the ability to write the output data to different output files. Below are the three scenarios where output file names can be changed.

1.If there is at least one reducer in the mapreduce job, output can be written to different files depending on the actual keys.

2.If the mapreduce job is a map only job, then job can use the output file name that is either a part of the input file name or any name derived from it.

3.If it is a map only job, job can use the output file name that depends on both the keys and the input file name.

MultipleSequenceFileOutputFormat

This is also a sub class of MultipleOutputFormat class. Using this format the output data can be written to different output files in Sequence file format.

MultipleTextOutputFormat

This is also a sub class of MultipleOutputFormat class. Using this format the output data can be written to different output files in Text output format.

LazyOutputFormat

By Default FileOutputFormat creates the output files even if a single output record is not emitted from reducers. Thus Mapreduce jobs create empty output files some times. This can be avoided with LazyOutputFormat in which output files are created only when the first output is emitted from the reducers. It is used in conjunction with org.apache.hadoop.mapreduce.lib.output.MultipleOutputs to recreate the behavior of *org.apache.hadoop.mapred.lib.MultipleTextOutputFormat* of the old Hadoop API.

DBOutputFormat

This output format is used to write an output into SQL tables using mapreduce jobs.

DBOutputFormat accepts <key,value> pairs, where key has a type extending DBWritable. Returned DBOutputFormat. DBRecordWriter writes only the key to the database with a batch SQL query.

NullOutputFormat

NullOutputFormat writes nothing to output directory. It Consumes all outputs and put them in /dev/null. We can suppress the key or value in the output using a NullWritable type. Both can be suppressed if the output file format is NullOutputFormat.

Using Advanced Hadoop MapReduce Features

Counters

Counters help in quantitative analysis of the job. It provides aggregated statistics at the end and hence can be referred to validate the output. Hadoop provides some built in as well as user defined counters. We can analysis them using apis in Driver class or all counters are listed in output logs at last.

The best thing about counters are that they work at the cluster level i.e., provides aggregated information about all the mappers and reducers.

Built-in Counters

Hadoop provides some built in counter to provide information about each process of hadoop for a particular job.

Few important ones for debugging and testing perspective: MAP_INPUT_RECORDS — number of input records consumed by all the maps. MAP_OUTPUT_RECORDS — number of output records produced by all the maps.

REDUCE_INPUT_RECORDS — number of input reocords consumed by all the reducers.
REDUCE_OUTPUT_RECORDS — number of output records produced by all the reducers.

User-Defined Java Counters

We can have our own counters to report the state of job. These provide output in form of a map. There are two ways to create and access the counters viz., enums and Strings. Enum is more easy and is type safe. It should be used in case we know all the output states in advance. Enum based counters are best suited for case where we want to calculate the number of requests based on HttpResponseCode. String based counters are dynamic and can be used where we don't have visibility in advance. This can be used when we want to do count based on domain.

Counters are incremented through the Reporter.incrCounter() method. The names of the counters are defined as Java enum's. The following example demonstrates how to count the number of "A" vs. "B" records seen by the mapper:

Custom Writables

Custom Types (Data)

For user provided Mapper and Reducer, the Hadoop MapReduce framework always uses typed data. The data which passes through Mappers and Reducers is stored in Java objects.

Writable Interface: The Writable interface is one of the most important interfaces. The objects which can be marshaled to/from files and over the network use this interface. Hadoop also uses this interface to transmit data in a serialized form. Some of the classes that implement Writable interface are mentioned below:

1. Text class(It stores String data)
2. LongWritable
3. FloatWritable
4. IntWritable
5. BooleanWritable

Custom data type can also be created by implementing the *Writable* interface. Hadoop is capable of transmitting any custom data type (which fits your requirement) that implements Writable interface.

The following is the Writable interface that has two methods readFields and write. The first method (readFields) initializes the data of the object from the data contained in the 'in' binary stream. The second method (write) is used to reconstruct the object to the binary stream 'out'. The most important contract of the entire process is that the order of read and write to the binary stream is same.

```
public interface Writable {
  void readFields(DataInput in);
  void write(DataOutput out);
}
```

Custom Types (Key)

In the previous section we discussed custom data types to meet an application specific data requirement. It manages the value part only. Now we will discuss the custom key type. In Hadoop MapReduce, the Reducer processes the key in sorted order. So the custom key type needs to implement the interface called *WritableComparable*. The key types should also implement hashCode ().

The following shows *WritableComparable* interface. It represents a Writable that is also *Comparable*.

```
public interface WritableComparable<T>
extends Writable, Comparable<T>
```

How to use Custom Types

We have already discussed the custom value and key types that can be processed by Hadoop. Now we explore the mechanism that Hadoop uses to understand it. The JobConf object (which defines the job) has two methods called *setOutputKeyClass ()* and *setOutputValueClass ()* and these methods are used to control the value and key data types. If the Mapper produces different types that do not match the Reducer then JobConf's *setMapOutputKeyClass ()* and *setMapOutputValueClass ()* methods can be used to set the input type as expected by the Reducer.

public class WritableComparator extends Object implements RawComparator

Custom data and key types allow us to use a higher level data structure in the Hadoop framework. In a practical Hadoop application, the custom data type is one of the most important requirements. So this feature allows the use of custom writable types and provides a significant performance improvement.

Data Partitioning

Partitioning can be defined as a process that determines which Reducer instance will receive which intermediate key/value pair. Each Mapper should determine the destination Reducer for all its output key/value pairs. The most important point is that for any key, regardless of its Mapper instance, the destination partition is the same. For performance reasons Mappers never communicate with each other to the partition of a particular key.

The *Partitioner* interface is used by the Hadoop system to determine the destination partition for a key/value pair. The number of partitions should match with the number of reduce tasks. The MapReduce framework determines the number of partitions when a job starts.

The following is the signature of Partitioner interface.

public interface Partitioner<K2,V2>
extends JobConfigurable

Unit Testing Framework for Map reduce Job

Being a parallel programming framework it becomes a bit difficult to properly unit test and validate map reduce jobs from a developer's scope let alone the Test Driven Development.

We will focus on various ways to do unit testing for map reduce jobs.

In the post we will discuss how to validate map reduce output using :

1. JUnit framework to test mappers and reducers using mocking (Mockito)
2. MRUnit framework to completely test the flow but in a single JVM.
3. mini-HDFS and a mini-MapReduce cluster to perform Integration Testing.
4. Hadoop Inbuilt Counters
5. LocalJobRunner to debug jobs using local filesystem.

1. JUnit framework to test mappers and reducers using mocking (Mockito)

Junit tests can be easily executed for Map Reduce jobs provided we test map function and reduce function in isolation. We can also test Driver function but with SpringData - Hadoop[http://www.springsource.org/spring-data/hadoop] project, driver configuration can be moved out of code. Using springData beans can further ease testing. If we execute the map and reduce function in isolation then there is dependency only on context object. We can easily clone context object using Mockito[http://code.google.com/p/mockito/].

All the tests can be executed from IDE. We just need to hadoop distribution jars and Mockito and junit jars in classpath.

Example to test the WordCount Mapper. It works with hadoop 1.0.3 and junit 4.1.

```
public class WordCountTest {

    private TokenizerMapper
    mapper; private Context context;
    final Map<Object,Object> test = new HashMap(); final
    AtomicInteger counter = new AtomicInteger(0);

    @Before
    public void setUp() throws Exception {
        mapper = new TokenizerMapper();
        context = mock(Context.class);
    }

    @Test
    public void testMethod() throws IOException, InterruptedException {

        doAnswer(new Answer<Object>() {
            public Object answer(InvocationOnMock invocation)
                { Object[] args = invocation.getArguments();
```

```
            test.put(args[0].toString(), args[1].toString());
            counter.incrementAndGet();
            return "called with arguments: " + args;
          }
        }).when(context).write(any(Text.class),any(IntWritable.class));

        mapper.map(new LongWritable(1L), new Text("counter counter counter" +
        " test test test"), context);
        Map<String,String> actualMap = new HashMap<String,
        String>(); actualMap.put("counter", "1");
        actualMap.put("test", "1");
        assertEquals(6,counter.get());
        assertEquals(actualMap, test);
    }
}
```

On the similiar lines reducer can be tested.

Key to use this strategy effectively is to refactor the code properly.Business logic related code should be moved out of map and reduce methods. It helps in effectively testing the business logic. We should also think about moving the mapper and reducer is separate classes. This follows strategy pattern and better reusability.

Junit tests with Mockito are very easy to use. Only problem is that we can not test the solution as a whole. It at max certifies the business logic. We should consider other testing strategy to test the complete solution.

2. MRUnit framework to completely test the flow but in a single JVM.

MRUnit is testing framework which provides support structure to test map reduce jobs. It provides mocking support which can be helpful in testing Mapper, Reducer, Mapper+Reducer and Driver as well. http://mrunit.apache.org/ is a top level apache project now. It takes JUnit mocking a level up for map reduce job testing.

Example

We require mrunit and mockito jars and hadoop supporting jars. Test has been executed on hadoop 0.20.203 and junit4. We are testing the PiEstimator example provided with hadoop distribution. public class TestExample {

```
    MapDriver<LongWritable, LongWritable, BooleanWritable,
    LongWritable> mapDriver;
    ReduceDriver<BooleanWritable, LongWritable, WritableComparable<?>,
    Writable> reduceDriver;
    MapReduceDriver<LongWritable, LongWritable, BooleanWritable,
    LongWritable, WritableComparable<?>, Writable> mapReduceDriver;

    @Before
    public void setUp() {
        PiEstimator.PiMapper mapper = new PiEstimator.PiMapper();
        PiEstimator.PiReducer reducer = new PiEstimator.PiReducer();
```

```
        mapDriver = new MapDriver<LongWritable, LongWritable,
        BooleanWritable, LongWritable>();
        mapDriver.setMapper(mapper);
        reduceDriver = new ReduceDriver<BooleanWritable,
        LongWritable, WritableComparable<?>, Writable>();
        reduceDriver.setReducer(reducer);
        mapReduceDriver = new MapReduceDriver<LongWritable, LongWritable,
        BooleanWritable,LongWritable,
        WritableComparable<?>, Writable>();
        mapReduceDriver.setMapper(mapper);
        mapReduceDriver.setReducer(reducer);
    }

    @Test
    public void testMapper() {
        mapDriver.withInput(new LongWritable(10), new LongWritable(10));
        mapDriver.withOutput(new BooleanWritable(true), new LongWritable(10));
        mapDriver.addOutput(new BooleanWritable(false), new LongWritable(0));
        mapDriver.runTest();
    }

    @Test
    public void testReducer() {
        List<LongWritable> values = new ArrayList<LongWritable>();
        values.add(new LongWritable(10)); reduceDriver.withInput(new
        BooleanWritable(true), values);

        reduceDriver.runTest();
    }
}
```

These tests are extemely fast as we dont require any interaction with filesystem. This are very good but lacks support to test code in distributed environment. Please check http://mrunit.apache.org/documentation/javadocs/0.9.0-incubating/org/apache/hadoop/mrunit/mock/package-summary.html for other useful support classes.

These tests can be sufficient to test code in isolation but doesn't test interaction with HDFS and test execution on cluster.

Hadoop Error Handling

Hadoop Error Handling! How can we make this work for us? The problem is that when errors happen in the mappers and/or reducers it is really hard to get the meaningful error information back to client.

The biggest reason for this is because a job would have been submitted at one node in cluster and then the job gets processed on other remote data nodes. Sure errors will be written into the log files of those data nodes, but that really does not allow for some nice neat error message to be displayed for someone to review.

The ability to present users with meaningful error messages is becoming more and more important. As the user base expands beyond the group of highly technical Hadoop nerds and moves into the

mainstream, with business analysts, market researchers, etc… using Hadoop. These less technical users need a different standard of interaction with the cluster.

Generally, they will need easy to use graphical interfaces that provide understandable error messages, when things go wrong. Thus effective Hadoop Error Handling will become a necessity.

The three approaches are:

- Handling non-fatal errors that need to be tracked
- Fatal errors that occurs that you want to define explicitly and occur only once
- Fatal errors that occur during the processing and can use a simple message

The Hadoop Error Handling for non-fatal errors that need to be tracked can be done by using counters. The basic idea is to create a counter with the Context.getCounter() and then increment it with each erorr.

Finally, on the client side after the job has completed to read the counter to get the total. For a very simplified example, see below.

In the mapper:

```
if(some_error_confidtion){
    context.getCounter(COUNTER_GROUP, COUNTER).increment(1);
}
```

In the client:

```
boolean okay =
job.waitForCompletion(true); if (okay){
    Counters counters = job.getCounters();
    Counter bwc = counters.findCounter(COUNTER_GROUP, COUNTER);
    System.out.println("Errors" + bwc.getDisplayName()+":" + bwc.getValue());
}
```

For the Hadoop Error Handling of fatal errors that you want to explicitly define and happen once, I have found that using an Enum is an excellent way to go. When you encounter the error condition, you would create the counter and use the Enum parameter with the error code value as the value of the counter:

```
catch (NullPointerException
    npe){ npe.printStackTrace();
    context.getCounter(ExceptionTypeEnum.PLACEHOLDER).setValue(
```

```
            ExceptionTypeEnum.NULL_POINTER.getErrorCode());
        needToStop++;
}
```

Then on the client side, you would need to retrieve the counter and then convert the PLACEHOLDER into the actual error message for presentation:

```
Counters counters = job.getCounters();
Counter bwc = counters.findCounter(ExceptionTypeEnum.PLACEHOLDER);
ExceptionTypeEnum err = ExceptionTypeEnum.get(bwc.getValue());
System.out.println();
System.out.println("Terminal error, " + err.getName());
System.out.println("With message: " + err.getMessage());
```

The only real drawback of this approach is that you need to ensure that you do not terminate the mapper when the error occurs. It needs to be allowed to complete at least one iteration, because otherwise the counter will be zeroed out.

The final Hadoop Error Handling approach is to reach into the data returned by the TaskTracker and then processing it. The challenge here is that the data is not readily available and there will be some duplication of the error messages. The first thing to know is that you will need to use the MR API to get the report.

The first thing you need to do is to take you JobID and downgrade it to the old API object. Then you need to create a JobClient() instance. With the JobClient, you can then get a RunningJob instance which gives you status information for the entire job. Next, you will want to get the specific error messages from the TaskTrackers.

You can use the JobClient.getMapTaskReports() and getReduceTaskReports() methods to get all the error messages from all of the failed tasks. This is where you will need to de-duplicate the error messages to ensure there is only one.

```
boolean okay = job.waitForCompletion(true);
JobID jobId = job.getJobID();
org.apache.hadoop.mapred.JobID oldJobId = org.apache.hadoop.mapred.JobID.downgrade(jobId);
RunningJob runningJob = jobClient.getJob(oldJobId);
JobStatus jobStatus = runningJob.getJobStatus();
TaskReport[] mapReports = jobClient.getMapTaskReports(oldJobId);
Map<String,String> errMap = new HashMap<String,String>();
```

```
for (TaskReport report : mapReports){
    TIPStatus status = report.getCurrentStatus();
    if (status.compareTo(TIPStatus.COMPLETE) ==
        0){ // not interested in tasks that completed
        continue;
    }
    for (String err : report.getDiagnostics()){
        // only need first line, can get the rest from logs
        // and don't want to get duplicates
        String[] errArr = err.split("\n");
        errMap.put(errArr[0],errArr[0]);
    }
}
```

Tuning of Mapreduce program

- Most important process of mapreduce program is shuffling of outputs produced by map function .
- So we need to concentrate mainly on map phase for better optimization of mapreduce programs

. hence , We should do the following things for optimization :-

1. We should give as much as more memory to shuffle process , but also need to keep in mind that we also give sufficient memory to map and reduce function .

2. Amount of memory given to JVM for map reduce tasks is set by :- mapred.child.java.opts , we need to make this value more as much as we can.

3. Map Side optimization :-

   a) optimization can be done by minimizing the multi spills ,, which can be controlled by io.sort.*

   b) We can use counters to check about the count of spill records

   c) We should increaseio.sort.mb :- which is used as amount of memory buffer used while sorting out the map output

   d) io.sort.spill.percent :- threshold value for using memory buffer , afterwards records started to spill.Default value :- 0.80

e) io.sort.factor :- property which help to merge output streams by map. We should increase this upto 100 for optimization .

f)mapred.compress.map.output :- we should compress the ouptput of mapper phase , it saves space as ell increase transfer of data between tasks.