UNIT-III
WRITING MAPREDUCE PROGRAMMING

MapReduce is a programming model for data processing. The model is simple, yet not too simple to express useful programs in. Hadoop can run MapReduce programs written in various languages; Most important, MapReduce programs are inherently parallel, thus putting very large-scale data analysis into the hands of anyone with enough machines at their disposal. MapReduce comes into its own for large datasets, so let's start by looking at one.

A Weather Dataset

For our example, we will write a program that mines weather data. Weather sensors collecting data every hour at many locations across the globe gather a large volume of log data, which is a good candidate for analysis with MapReduce, since it is semistructured and record-oriented.

Data Format

The data we will use is from the National Climatic Data Center (NCDC, *http://www.ncdc.noaa.gov/*). The data is stored using a line-oriented ASCII format, in which each line is a record. The format supports a rich set of meteorological elements, many of which are optional or with variable data lengths. For simplicity, we shall focus on the basic elements, such as temperature, which are always present and are of fixed width.

Example shows a sample line with some of the salient fields highlighted. The line has been split into multiple lines to show each field: in the real file, fields are packed into one line with no delimiters.

Data files are organized by date and weather station. There is a directory for each year from 1901 to 2001, each containing a gzipped file for each weather station with its readings for that year. For example, here are the first entries for 1990:

```
% ls raw/1990 | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
```

Since there are tens of thousands of weather stations, the whole dataset is made up of a large number of relatively small files. It's generally easier and more efficient to process a smaller number of relatively large files, so the data was preprocessed so that each year's readings were concatenated into a single file.

Analyzing the Data with Hadoop

To take advantage of the parallel processing that Hadoop provides, we need to express our query as a MapReduce job. After some local, small-scale testing, we will be able to run it on a cluster of machines.

Map and Reduce

MapReduce works by breaking the processing into **two phases**: the **map phase** and the **reduce phase**. Each phase has **key-value pairs as input and output**, the types of which

may be chosen by the programmer. The programmer also specifies two functions: the **map function** and the **reduce function**.

The input to our map phase is the raw NCDC data. We choose a text input format that gives us each line in the dataset as a text value. The key is the offset of the beginning of the line from the beginning of the file, but as we have no need for this, we ignore it.

Our map function is simple. We pull out the year and the air temperature, since these are the only fields we are interested in. In this case, the map function is just a data preparation phase, setting up the data in such a way that the reducer function can do its work on it: finding the maximum temperature for each year. The map function is also a good place to drop bad records: here we filter out temperatures that are missing, suspect, or erroneous.

To visualize the way the map works, consider the following sample lines of input data (some unused columns have been dropped to fit the page, indicated by ellipses):

```
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
```

These lines are presented to the map function as the key-value pairs:

```
(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

The keys are the line offsets within the file, which we ignore in our map function. The map function merely extracts the year and the air temperature (indicated in bold text), and emits them as its output (the temperature values have been interpreted as integers):

```
(1950, 0)
(1950, 22)
(1950, −11)
(1949, 111)
(1949, 78)
```

The output from the map function is processed by the MapReduce framework before being sent to the reduce function. This processing **sorts and groups the key-value pairs** by key. So, continuing the example, our reduce function sees the following input:

```
(1949, [111, 78])
(1950, [0, 22, −11])
```

Each year appears with a list of all its air temperature readings. All the reduce function has to do now is iterate through the list and pick up the maximum reading:

```
(1949, 111)
(1950, 22)
```

This is the final output: the maximum global temperature recorded in each year.

The whole data flow is illustrated in the bellow Figure. At the bottom of the diagram is a Unix pipeline

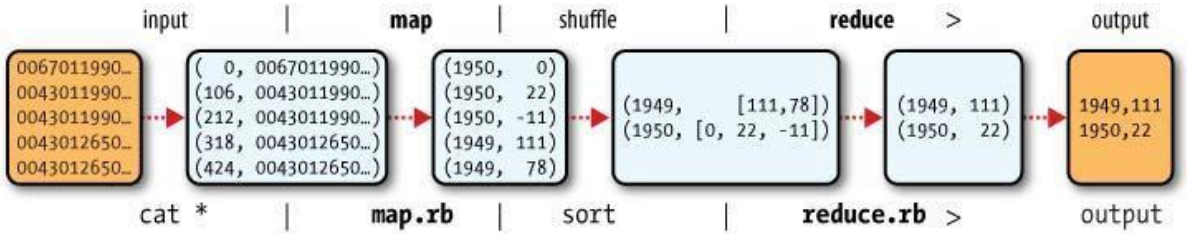


Figure 2-1. MapReduce logical data flow

Java MapReduce

Having run through how the MapReduce program works, the next step is to express it in code. We **need three things: a map function, a reduce function, and some code to run the job**. The map function is represented by an implementation of the Mapper interface, which declares a map() method. Example -1 shows the implementation of our map function.

Example - 1. Mapper for maximum temperature example

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
public class MaxTemperatureMapper extends MapReduceBase implements
                                        Mapper <LongWritable, Text, Text, IntWritable>
{
private static final int MISSING = 9999;
public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
Reporter reporter) throws IOException
{
String line = value.toString();
String year = line.substring(15, 19);
int airTemperature;
if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
airTemperature = Integer.parseInt(line.substring(88, 92));
} else {
airTemperature = Integer.parseInt(line.substring(87, 92));
}
String quality = line.substring(92, 93);
if (airTemperature != MISSING && quality.matches("[01459]")) {
output.collect(new Text(year), new IntWritable(airTemperature));
}
}
}
```

The Mapper interface is a generic type, with four formal type parameters that specify the input key, input value, output key, and output value types of the map function. For the present example, the input key is a long integer offset, the input value is a line of text, the output key is a year, and the output value is an air temperature (an integer). Rather than use built-in Java types, Hadoop provides its own set of basic types that are optimized for network serialization. These are found in the **org.apache.hadoop.io package**.

Here we use **LongWritable**, which corresponds to a Java Long, **Text** (like Java String), **and IntWritable** (like Java Integer).

The map() method is passed a key and a value. We convert the Text value containing the line of input into a Java String, then use its substring() method to extract the columns we are interested in.

The map() method also provides an instance of OutputCollector to write the output to. In this case, we write the year as a Text object (since we are just using it as a key), and the temperature is wrapped in an IntWritable. We write an output record only if the temperature is present and the quality code indicates the temperature reading is OK.

The reduce function is similarly defined using a Reducer, as illustrated in Example - 2.

Example - 2. Reducer for maximum temperature example

```
import java.io.IOException;
import java.util.Iterator;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import  org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
public class MaxTemperatureReducer extends MapReduceBase
                                        implements Reducer<Text, IntWritable, Text, IntWritable>
{
public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException
{
int maxValue = Integer.MIN_VALUE;
while (values.hasNext()) {
maxValue = Math.max(maxValue, values.next().get());
}
output.collect(key, new IntWritable(maxValue));
}
}
```

Again, four formal type parameters are used to specify the input and output types, this time for the reduce function. The input types of the reduce function must match the output types of the map function: Text and IntWritable. And in this case, the output types of the reduce function are Text and IntWritable, for a year and its maximum temperature, which we find by iterating through the temperatures and comparing each with a record of the highest found so far.

The third piece of code runs the MapReduce job (see Example -3).

Example -3. Application to find the maximum temperature in the weather dataset

```
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
public class MaxTemperature
{
public static void main(String[] args) throws IOException
{
if (args.length != 2)
{
System.err.println("Usage: MaxTemperature <input path> <output path>");
System.exit(-1);
}
JobConf conf = new JobConf(MaxTemperature.class);
conf.setJobName("Max temperature");
FileInputFormat.addInputPath(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));
conf.setMapperClass(MaxTemperatureMapper.class);
conf.setReducerClass(MaxTemperatureReducer.class);
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);
JobClient.runJob(conf);
}
}
```

A **JobConf** object forms the specification of the job. It gives you control over how the job is run. When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute around the cluster). Rather than explicitly specify the name of the JAR file, we can pass a class in the JobConf constructor, which Hadoop will use to locate the relevant JAR file by looking for the JAR file containing this class.

Having constructed a JobConf object, we specify the input and output paths. An input path is specified by calling the static **addInputPath()** method on **FileInputFormat**, and it can be a single file, a directory (in which case, the input forms all the files in that directory), or a file pattern. As the name suggests, **addInputPath()** can be called more than once to use input from multiple paths.

The output path (of which there is only one) is specified by the static **setOutput Path()** method on **FileOutputFormat**. It specifies a directory where the output files from the reducer functions are written. **The directory shouldn't exist before running the job**, as Hadoop will complain and not run the job. This precaution is to prevent data loss (it can be very annoying to accidentally overwrite the output of a long job with another).

Next, we specify the map and reduce types to use via the **setMapperClass()** and **setReducerClass()** methods. The **setOutputKeyClass()** and **setOutputValueClass()** methods control the output types for the map and the reduce functions, which are often the same, as they are in our case.

If they are different, then the map output types can be set using the methods setMapOutputKeyClass() and setMapOutputValueClass(). The input types are controlled via the input format, which we have not explicitly set since we are using the default TextInputFormat. After setting the classes that define the map and reduce functions, we are ready to run the job.

The static **runJob()** method on **JobClient** submits the job and waits for it to finish, writing information about its progress to the console.

The output was written to the *output* directory, which contains one output file per reducer. The job had a single reducer, so we find a single file, named ***part-00000***:

% **cat output/part-00000**
1949 111
1950 22

This result is the same as when we went through it by hand earlier. We interpret this as saying that the maximum temperature recorded in 1949 was 11.1°C, and in 1950 it was 2.2°C.

The new Java MapReduce API:

Release 0.20.0 of Hadoop included a new Java MapReduce API, sometimes referred to as "Context Objects," designed to make the API easier to evolve in the future. The new API is type-incompatible with the old, however, so applications need to be rewritten to take advantage of it.

There are several notable differences between the two APIs:

- The new API favours abstract classes over interfaces, since these are easier to evolve. For example, you can add a method (with a default implementation) to an abstract class without breaking old implementations of the class. In the new API, the Mapper and Reducer interfaces are now abstract classes.
- The new API is in the **org.apache.hadoop.mapreduce** package (and subpackages). The old API can still be found in **org.apache.hadoop.mapred**.
- The new API makes extensive use of context objects that allow the user code to communicate with the MapReduce system. The MapContext, for example, essentially unifies the role of the JobConf, the OutputCollector, and the Reporter.
- The new API supports both a "push" and a "pull" style of iteration. In both APIs, key-value record pairs are pushed to the mapper, but in addition, the new API allows a mapper to pull records from within the map() method. The same goes for the reducer. An example of how the "pull" style can be useful is processing records in batches, rather than one by one.
- Configuration has been unified. The old API has a special JobConf object for job configuration, which is an extension of Hadoop's vanilla Configuration object. In the new API, this distinction is dropped, so job configuration is done through a Configuration.
- Job control is performed through the Job class, rather than JobClient, which no longer exists in the new API.

- Output files are named slightly differently: ***part-m-nnnnn*** for map outputs, and ***partr-nnnnn*** for reduce outputs (where ***nnnnn*** is an integer designating the part number, starting from **zero**).

When converting your Mapper and Reducer classes to the new API, don't forget to change the signature of the map() and reduce() methods to the new form. Just changing your class to extend the new Mapper or Reducer classes will *not* produce a compilation error or warning, since these classes provide an identity form of the map() or reduce() method (respectively). Your mapper or reducer code, however, will not be invoked, which can lead to some hard-to-diagnose errors.

Combiner

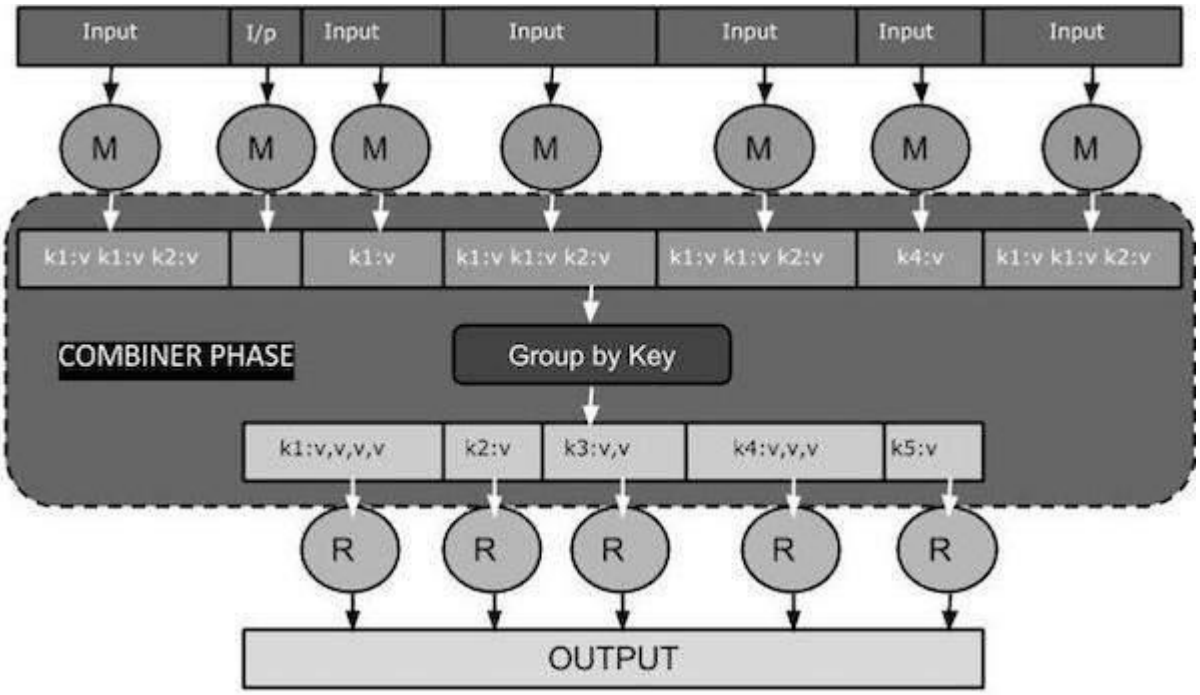
A **Combiner**, also known as a **semi-reducer,** is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class.

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.

Combiner class

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**, and 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 WordCountMapper 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(WordCountMapper.class);
job.setCombinerClass(WordCountReducer.class);
job.setReducerClass(WordCountReducer.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>
```

Partitioner Phase

The partitioning phase takes place after the **map phase** and before the **reduce phase**. The number of **partitions is equal to the number of reducers**. The data gets partitioned across the reducers according to the partitioning function.

The difference between a partitioner and a combiner is that **the partitioner** divides the data according to the number of reducers so that all the data in a single partition gets executed by a single reducer. However, **the combiner** functions similar to the reducer and processes the data in each partition. The combiner is an optimization to the reducer.

The default partitioning function is the hash partitioning function where the hashing is done on the key. However it might be useful to partition the data according to some other function of the key or the value.

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 WordCountReducer 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>
```

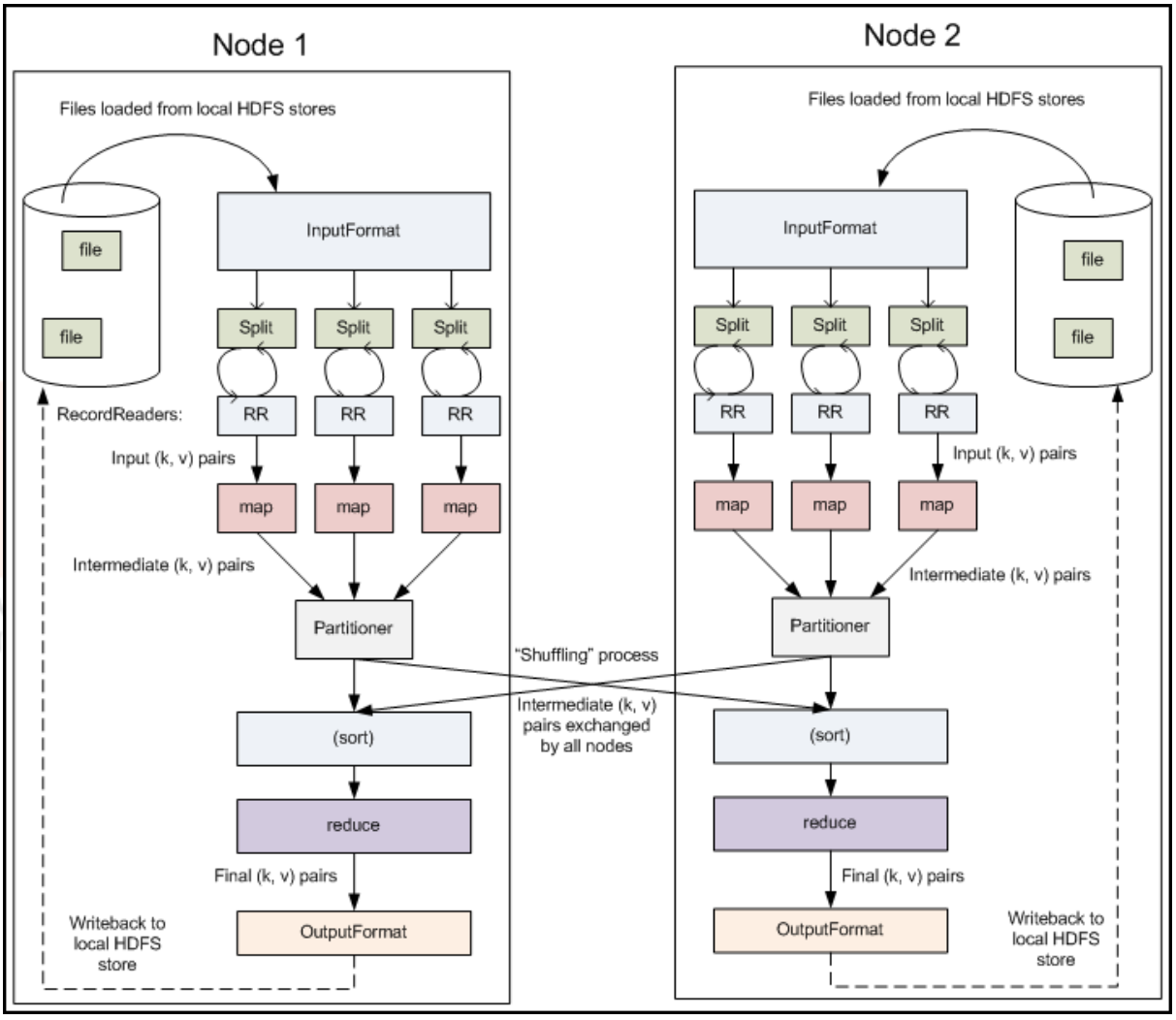
Record Writer

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
```


Node 1
Node 2
Files loaded from local HDFS stores
Files loaded from local HDFS stores
file
file
file
file
InputFormat
InputFormat
Split
Split
Split
Split
Split
Split
RecordReaders:
RR
RR
RR
RR
RR
RR
Input (k, v) pairs
Input (k, v) pairs
map
map
map
map
map
map
Intermediate (k, v) pairs
Intermediate (k, v) pairs
Partitioner
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes
(sort)
(sort)
reduce
reduce
Final (k, v) pairs
Final (k, v) pairs
Writeback to local HDFS store
OutputFormat
OutputFormat
Writeback to local HDFS store