# Hadoop - MapReduce

MapReduce is a framework using which we can write applications to process huge amounts of data, in parallel, on large clusters of commodity hardware in a reliable manner.

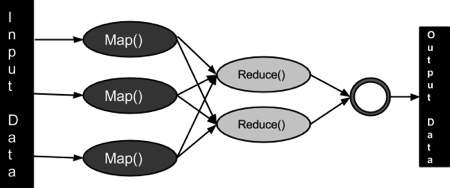
## What is MapReduce?

MapReduce is a processing technique and a program model for distributed computing based on java. The MapReduce algorithm contains two important tasks, namely Map and Reduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job.

The major advantage of MapReduce is that it is easy to scale data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called mappers and reducers. Decomposing a data processing application into mappers and reducers is sometimes nontrivial. But, once we write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change. This simple scalability is what has attracted many programmers to use the MapReduce model.

## The Algorithm

* Generally MapReduce paradigm is based on sending the computer to where the data resides!
* MapReduce program executes in three stages, namely map stage, shuffle stage, and reduce stage.
  + **Map stage** : The map or mapper’s job is to process the input data. Generally the input data is in the form of file or directory and is stored in the Hadoop file system (HDFS). The input file is passed to the mapper function line by line. The mapper processes the data and creates several small chunks of data.
  + **Reduce stage** : This stage is the combination of the **Shuffle**stage and the **Reduce** stage. The Reducer’s job is to process the data that comes from the mapper. After processing, it produces a new set of output, which will be stored in the HDFS.
* During a MapReduce job, Hadoop sends the Map and Reduce tasks to the appropriate servers in the cluster.
* The framework manages all the details of data-passing such as issuing tasks, verifying task completion, and copying data around the cluster between the nodes.
* Most of the computing takes place on nodes with data on local disks that reduces the network traffic.
* After completion of the given tasks, the cluster collects and reduces the data to form an appropriate result, and sends it back to the Hadoop server.



## Inputs and Outputs (Java Perspective)

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

The key and the value classes should be in serialized manner by the framework and hence, need to implement the Writable interface. Additionally, the key classes have to implement the Writable-Comparable interface to facilitate sorting by the framework. Input and Output types of a MapReduce job: (Input) <k1, v1> -> map -> <k2, v2>-> reduce -> <k3, v3>(Output).

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

## Terminology

* **PayLoad** - Applications implement the Map and the Reduce functions, and form the core of the job.
* **Mapper** - Mapper maps the input key/value pairs to a set of intermediate key/value pair.
* **NamedNode** - Node that manages the Hadoop Distributed File System (HDFS).
* **DataNode** - Node where data is presented in advance before any processing takes place.
* **MasterNode** - Node where JobTracker runs and which accepts job requests from clients.
* **SlaveNode** - Node where Map and Reduce program runs.
* **JobTracker** - Schedules jobs and tracks the assign jobs to Task tracker.
* **Task Tracker** - Tracks the task and reports status to JobTracker.
* **Job** - A program is an execution of a Mapper and Reducer across a dataset.
* **Task** - An execution of a Mapper or a Reducer on a slice of data.
* **Task Attempt** - A particular instance of an attempt to execute a task on a SlaveNode.

## Example Scenario

Given below is the data regarding the electrical consumption of an organization. It contains the monthly electrical consumption and the annual average for various years.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Jan** | **Feb** | **Mar** | **Apr** | **May** | **Jun** | **Jul** | **Aug** | **Sep** | **Oct** | **Nov** | **Dec** | **Avg** |
| 1979 | 23 | 23 | 2 | 43 | 24 | 25 | 26 | 26 | 26 | 26 | 25 | 26 | 25 |
| 1980 | 26 | 27 | 28 | 28 | 28 | 30 | 31 | 31 | 31 | 30 | 30 | 30 | 29 |
| 1981 | 31 | 32 | 32 | 32 | 33 | 34 | 35 | 36 | 36 | 34 | 34 | 34 | 34 |
| 1984 | 39 | 38 | 39 | 39 | 39 | 41 | 42 | 43 | 40 | 39 | 38 | 38 | 40 |
| 1985 | 38 | 39 | 39 | 39 | 39 | 41 | 41 | 41 | 00 | 40 | 39 | 39 | 45 |

If the above data is given as input, we have to write applications to process it and produce results such as finding the year of maximum usage, year of minimum usage, and so on. This is a walkover for the programmers with finite number of records. They will simply write the logic to produce the required output, and pass the data to the application written.

But, think of the data representing the electrical consumption of all the largescale industries of a particular state, since its formation.

When we write applications to process such bulk data,

* They will take a lot of time to execute.
* There will be a heavy network traffic when we move data from source to network server and so on.

To solve these problems, we have the MapReduce framework.

### Input Data

The above data is saved as **sample.txt**and given as input. The input file looks as shown below.

1979 23 23 2 43 24 25 26 26 26 26 25 26 25

1980 26 27 28 28 28 30 31 31 31 30 30 30 29

1981 31 32 32 32 33 34 35 36 36 34 34 34 34

1984 39 38 39 39 39 41 42 43 40 39 38 38 40

1985 38 39 39 39 39 41 41 41 00 40 39 39 45

### Example Program

Given below is the program to the sample data using MapReduce framework.

package hadoop;

import java.util.\*;

import java.io.IOException;

import java.io.IOException;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.conf.\*;

import org.apache.hadoop.io.\*;

import org.apache.hadoop.mapred.\*;

import org.apache.hadoop.util.\*;

public class ProcessUnits

{

//Mapper class

public static class E\_EMapper extends MapReduceBase implements

Mapper<LongWritable ,/\*Input key Type \*/

Text, /\*Input value Type\*/

Text, /\*Output key Type\*/

IntWritable> /\*Output value Type\*/

{

//Map function

public void map(LongWritable key, Text value,

OutputCollector<Text, IntWritable> output,

Reporter reporter) throws IOException

{

String line = value.toString();

String lasttoken = null;

StringTokenizer s = new StringTokenizer(line,"\t");

String year = s.nextToken();

while(s.hasMoreTokens())

{

lasttoken=s.nextToken();

}

int avgprice = Integer.parseInt(lasttoken);

output.collect(new Text(year), new IntWritable(avgprice));

}

}

//Reducer class

public static class E\_EReduce extends MapReduceBase implements

Reducer< Text, IntWritable, Text, IntWritable >

{

//Reduce function

public void reduce( Text key, Iterator <IntWritable> values,

OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException

{

int maxavg=30;

int val=Integer.MIN\_VALUE;

while (values.hasNext())

{

if((val=values.next().get())>maxavg)

{

output.collect(key, new IntWritable(val));

}

}

}

}

//Main function

public static void main(String args[])throws Exception

{

JobConf conf = new JobConf(ProcessUnits.class);

conf.setJobName("max\_eletricityunits");

conf.setOutputKeyClass(Text.class);

conf.setOutputValueClass(IntWritable.class);

conf.setMapperClass(E\_EMapper.class);

conf.setCombinerClass(E\_EReduce.class);

conf.setReducerClass(E\_EReduce.class);

conf.setInputFormat(TextInputFormat.class);

conf.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(conf, new Path(args[0]));

FileOutputFormat.setOutputPath(conf, new Path(args[1]));

JobClient.runJob(conf);

}

}

Save the above program as **ProcessUnits.java.** The compilation and execution of the program is explained below.

## Compilation and Execution of Process Units Program

Let us assume we are in the home directory of a Hadoop user (e.g. /home/hadoop).

Follow the steps given below to compile and execute the above program.

### Step 1

The following command is 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. Visit the following link<http://mvnrepository.com/artifact/org.apache.hadoop/hadoop-core/1.2.1> to download the jar. Let us assume the downloaded folder is **/home/hadoop/.**

### Step 3

The following commands are used for compiling the **ProcessUnits.java**program and creating a jar for the program.

$ javac -classpath hadoop-core-1.2.1.jar -d units ProcessUnits.java

$ jar -cvf units.jar -C units/ .

### Step 4

The following command is used to create an input directory in HDFS.

$HADOOP\_HOME/bin/hadoop fs -mkdir input\_dir

### Step 5

The following command is used to copy the input file named **sample.txt**in the input directory of HDFS.

$HADOOP\_HOME/bin/hadoop fs -put /home/hadoop/sample.txt input\_dir

### Step 6

The following command is used to verify the files in the input directory.

$HADOOP\_HOME/bin/hadoop fs -ls input\_dir/

### Step 7

The following command is used to run the Eleunit\_max application by taking the input files from the input directory.

$HADOOP\_HOME/bin/hadoop jar units.jar hadoop.ProcessUnits input\_dir output\_dir

Wait for a while until the file is executed. After execution, as shown below, the output will contain the number of input splits, the number of Map tasks, the number of reducer tasks, etc.

INFO mapreduce.Job: Job job\_1414748220717\_0002

completed successfully

14/10/31 06:02:52

INFO mapreduce.Job: Counters: 49

File System Counters

FILE: Number of bytes read=61

FILE: Number of bytes written=279400

FILE: Number of read operations=0

FILE: Number of large read operations=0

FILE: Number of write operations=0

HDFS: Number of bytes read=546

HDFS: Number of bytes written=40

HDFS: Number of read operations=9

HDFS: Number of large read operations=0

HDFS: Number of write operations=2 Job Counters

Launched map tasks=2

Launched reduce tasks=1

Data-local map tasks=2

Total time spent by all maps in occupied slots (ms)=146137

Total time spent by all reduces in occupied slots (ms)=441

Total time spent by all map tasks (ms)=14613

Total time spent by all reduce tasks (ms)=44120

Total vcore-seconds taken by all map tasks=146137

Total vcore-seconds taken by all reduce tasks=44120

Total megabyte-seconds taken by all map tasks=149644288

Total megabyte-seconds taken by all reduce tasks=45178880

Map-Reduce Framework

Map input records=5

Map output records=5

Map output bytes=45

Map output materialized bytes=67

Input split bytes=208

Combine input records=5

Combine output records=5

Reduce input groups=5

Reduce shuffle bytes=6

Reduce input records=5

Reduce output records=5

Spilled Records=10

Shuffled Maps =2

Failed Shuffles=0

Merged Map outputs=2

GC time elapsed (ms)=948

CPU time spent (ms)=5160

Physical memory (bytes) snapshot=47749120

Virtual memory (bytes) snapshot=2899349504

Total committed heap usage (bytes)=277684224

File Output Format Counters

Bytes Written=40

### Step 8

The following command is used to verify the resultant files in the output folder.

$HADOOP\_HOME/bin/hadoop fs -ls output\_dir/

### Step 9

The following command is used to see the output in **Part-00000**file. This file is generated by HDFS.

$HADOOP\_HOME/bin/hadoop fs -cat output\_dir/part-00000

Below is the output generated by the MapReduce program.

1981 34

1984 40

1985 45

### Step 10

The following command is used to copy the output folder from HDFS to the local file system for analyzing.

$HADOOP\_HOME/bin/hadoop fs -cat output\_dir/part-00000/bin/hadoop dfs get output\_dir /home/hadoop

# MapReduce - API

we will take a close look at the classes and their methods that are involved in the operations of MapReduce programming. We will primarily keep our focus on the following −

* JobContext Interface
* Job Class
* Mapper Class
* Reducer Class

## JobContext Interface

The JobContext interface is the super interface for all the classes, which defines different jobs in MapReduce. It gives you a read-only view of the job that is provided to the tasks while they are running.

The following are the sub-interfaces of JobContext interface.

|  |  |
| --- | --- |
| **S.No.** | **Subinterface Description** |
| 1. | **MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT>**  Defines the context that is given to the Mapper. |
| 2. | **ReduceContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT>**  Defines the context that is passed to the Reducer. |

Job class is the main class that implements the JobContext interface.

## Job Class

The Job class is the most important class in the MapReduce API. It allows the user to configure the job, submit it, control its execution, and query the state. The set methods only work until the job is submitted, afterwards they will throw an IllegalStateException.

Normally, the user creates the application, describes the various facets of the job, and then submits the job and monitors its progress.

Here is an example of how to submit a job −

// Create a new Job

Job job = new Job(new Configuration());

job.setJarByClass(MyJob.class);

// Specify various job-specific parameters

job.setJobName("myjob");

job.setInputPath(new Path("in"));

job.setOutputPath(new Path("out"));

job.setMapperClass(MyJob.MyMapper.class);

job.setReducerClass(MyJob.MyReducer.class);

// Submit the job, then poll for progress until the job is complete

job.waitForCompletion(true);

### Constructors

Following are the constructor summary of Job class.

|  |  |
| --- | --- |
| **S.No** | **Constructor Summary** |
| 1 | **Job**() |
| 2 | **Job**(Configuration conf) |
| 3 | **Job**(Configuration conf, String jobName) |

### Methods

Some of the important methods of Job class are as follows −

|  |  |
| --- | --- |
| **S.No** | **Method Description** |
| 1 | **getJobName()**  User-specified job name. |
| 2 | **getJobState()**  Returns the current state of the Job. |
| 3 | **isComplete()**  Checks if the job is finished or not. |
| 4 | **setInputFormatClass()**  Sets the InputFormat for the job. |
| 5 | **setJobName(String name)**  Sets the user-specified job name. |
| 6 | **setOutputFormatClass()**  Sets the Output Format for the job. |
| 7 | **setMapperClass(Class)**  Sets the Mapper for the job. |
| 8 | **setReducerClass(Class)**  Sets the Reducer for the job. |
| 9 | **setPartitionerClass(Class)**  Sets the Partitioner for the job. |
| 10 | **setCombinerClass(Class)**  Sets the Combiner for the job. |

## Mapper Class

The Mapper class defines the Map job. Maps input key-value pairs to a set of intermediate key-value pairs. Maps are the individual tasks that transform the input records into intermediate records. The transformed intermediate records need not be of the same type as the input records. A given input pair may map to zero or many output pairs.

### Method

**map** is the most prominent method of the Mapper class. The syntax is defined below −

map(KEYIN key, VALUEIN value, org.apache.hadoop.mapreduce.Mapper.Context context)

This method is called once for each key-value pair in the input split.

## Reducer Class

The Reducer class defines the Reduce job in MapReduce. It reduces a set of intermediate values that share a key to a smaller set of values. Reducer implementations can access the Configuration for a job via the JobContext.getConfiguration() method. A Reducer has three primary phases − Shuffle, Sort, and Reduce.

* **Shuffle** − The Reducer copies the sorted output from each Mapper using HTTP across the network.
* **Sort** − The framework merge-sorts the Reducer inputs by keys (since different Mappers may have output the same key). The shuffle and sort phases occur simultaneously, i.e., while outputs are being fetched, they are merged.
* **Reduce** − In this phase the reduce (Object, Iterable, Context) method is called for each <key, (collection of values)> in the sorted inputs.

### Method

**reduce** is the most prominent method of the Reducer class. The syntax is defined below −

**reduce**(KEYIN key, Iterable<VALUEIN> values, org.apache.hadoop.mapreduce.Reducer.Context context)

This method is called once for each key on the collection of key-value pairs.

**Writable** in an interface in Hadoop and types in Hadoop must implement this interface. Hadoop provides these writable wrappers for almost all Java primitive types and some other types,but sometimes we need to pass custom objects and these custom objects should implement Hadoop's Writable interface.Hadoop MapReduce uses implementations of Writables for interacting with user-provided Mappers and Reducers.

To implement the Writable interface we require two methods:

public interface Writable {

void readFields(DataInput in);

void write(DataOutput out);

}

**Why use Hadoop Writable(s)?**

As we already know, data needs to be transmitted between different nodes in a distributed computing environment. This requires serialization and deserialization of data to convert the data that is in structured format to byte stream and vice-versa. Hadoop therefore uses simple and efficient serialization protocol to serialize data between map and reduce phase and these are called Writable(s). Some of the examples of writables as already mentioned before are IntWritable, LongWritable, BooleanWritable and FloatWritable.

**WritableComparable** interface is just a subinterface of the Writable and java.lang.Comparable interfaces. For implementing a WritableComparable we must have compareTo method apart from readFields and write methods, as shown below:

public interface WritableComparable extends Writable, Comparable

{

void readFields(DataInput in);

void write(DataOutput out);

int compareTo(WritableComparable o)

}

Comparison of types is crucial for MapReduce, where there is a sorting phase during which keys are compared with one another.

Implementing a comparator for WritableComparables like the org.apache.hadoop.io.RawComparator interface will definitely help speed up your Map/Reduce (MR) Jobs. As you may recall, a MR Job is composed of receiving and sending key-value pairs. The process looks like the following.

(K1,V1) –> Map –> (K2,V2)

(K2,List[V2]) –> Reduce –> (K3,V3)

The key-value pairs (K2,V2) are called the intermediary key-value pairs. They are passed from the mapper to the reducer. Before these intermediary key-value pairs reach the reducer, a shuffle and sort step is performed.

The shuffle is the assignment of the intermediary keys (K2) to reducers and the sort is the sorting of these keys. In this blog, by implementing the RawComparator to compare the intermediary keys, this extra effort will greatly improve sorting. Sorting is improved because the RawComparator will compare the keys by byte. If we did not use RawComparator, the intermediary keys would have to be completely deserialized to perform a comparison.

**Note (In Short):**

1)WritableComparables can be compared to each other, typically via Comparators. Any type which is to be used as a key in the Hadoop Map-Reduce framework should implement this interface.

2) Any type which is to be used as a value in the Hadoop Map-Reduce framework should implement the Writable interface.

 let us understand some basics and get the motivation behind implementing a custom Writable.

We will discuss the following in this post:

1. What is a Writable in Hadoop?
2. Why does Hadoop use Writable(s)?
3. Limitation of primitive Hadoop Writable classes
4. Custom Writable
5. BigramCount Example
   * Code the custom Writable class
   * Code the Mapper class
   * Code the Reducer class
   * Code the Driver class
6. Setup the input directory in HDFS
7. Run the job

#### **What is a Writable in Hadoop?**

If you have gone through the “Hello World” of MapReduce [post](https://learnhadoopwithme.wordpress.com/2013/08/20/hello-world-of-mapreduce-word-count/), or any other Hadoop program, you must have seen data types different from regular Java defined data types. In wordCount post, you must have seen LongWritable, IntWrtitable and Text. It is fairly easy to understand the relation between them and Java’s primitive types. LongWritable is equivalent to long, IntWritable to int and Text to String.

Any value in Hadoop must be Writable. A Writable in an interface in Hadoop and types in Hadoop must implement this interface. Hadoop provides these writable wrappers for almost all Java primitive types and some other types.

Now the obvious question is why does Hadoop use these types instead of Java types?

#### **Why does Hadoop use Writable(s)?**

As we already know, data needs to be transmitted between different nodes in a distributed computing environment. This requires serialization and deserialization of data to convert the data that is in structured format to byte stream and vice-versa. Hadoop therefore uses simple and efficient serialization protocol to serialize data between map and reduce phase and these are called Writable(s). Some of the examples of writables as already mentioned before are IntWritable, LongWritable, BooleanWritable and FloatWritable. The entire list is in org.apache.hadoop.io package of the Hadoop Source (<http://hadoop.apache.org/docs/current/api/index.html>).

#### **Limitation of primitive Hadoop Writable classes**

In the wordCount example we emit Text as the key and IntWritable as the value from the Mappers and Reducers. Although Hadoop provides many primitive Writable that can be used in simple applications like wordcount, but clearly these cannot serve our purpose all the time.

Consider a scenario where we would like to transmit a 3-D point as a value from the Mappers/Reducers. The structure of the 3D point would be like,

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class point3D  {      public float x;      public float y;      public float z;  } |

Now if you want to still use the primitive Hadoop Writable(s), you would have to convert the value into a string and transmit it. However it gets very messy when you have to deal with string manipulations.

Also, what if you want to transmit this as a key? As we already know Hadoop does the sorting and shuffling automatically, then these point will get sorted based on string values, which would not be correct. So clearly we need to write custom data types that can be used in Hadoop.

#### **Custom Writable**

So any user defined class that implements the Writable interface is a custom writable. So let us first look into the structure of writable interface.

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface Writable  {      void readFields(DataInput in);      void write(DataOutput out);  } |

So the class implementing this interface must provide the implementation of these two method at the very least. So let us now look into these two methods in detail.

write(DataOutput out) – It is used to serialize the fields of the object to ‘out’.  
readFields(DataInput in) – It is used to deserialize the fields of the object from ‘in’.

However, we need a custom Writable comparable if our custom data type is going to be used as key rather that the value. We then need the class to implement WritableComparable interface. The WritableComparable interface extends from the Writable interface and the Compararble interface its structure is as given below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public interface WritableComparable extends Writable, Comparable  {      void readFields(DataInput in);      void write(DataOutput out);      int compareTo(WritableComparable o)  } |

compareTo(WritableComparable o) – It is inherited from Comparable interface and it allows Hadoop to sort the keys in the sort and shuffle phase.

#### **BigramCount Example**

Let us know look into the BigramCount example which will solidify the concepts that we have learnt till now in this post. This example is a good extension to the wordCount example, and will also teach us how to write a custom Writable.

###### **Code the custom Writable class**

In BigramCount we need to count the frequency of the occurrence of two words together in the text. So we are going to define a custom class that is going to hold the two words together.

The code for that is as given below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82 | import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;    import org.apache.hadoop.io.Text;  import org.apache.hadoop.io.WritableComparable;    public class TextPair implements WritableComparable {        private Text first;      private Text second;        public TextPair(Text first, Text second) {          set(first, second);      }        public TextPair() {          set(new Text(), new Text());      }        public TextPair(String first, String second) {          set(new Text(first), new Text(second));      }        public Text getFirst() {          return first;      }        public Text getSecond() {          return second;      }        public void set(Text first, Text second) {          this.first = first;          this.second = second;      }        @Override      public void readFields(DataInput in) throws IOException {          first.readFields(in);          second.readFields(in);      }        @Override      public void write(DataOutput out) throws IOException {          first.write(out);          second.write(out);      }        @Override      public String toString() {          return first + " " + second;      }        @Override      public int compareTo(TextPair tp) {          int cmp = first.compareTo(tp.first);            if (cmp != 0) {              return cmp;          }            return second.compareTo(tp.second);      }        @Override      public int hashCode(){          return first.hashCode()\*163 + second.hashCode();      }        @Override      public boolean equals(Object o)      {          if(o instanceof TextPair)          {              TextPair tp = (TextPair) o;              return first.equals(tp.first) && second.equals(tp.second);          }          return false;      }    } |

We have already seen the explanation of readFields(), write() and compareTo(). And just as you would for any value object you write in Java, you should override the hashCode(), equals(), and toString() methods from java.lang.Object. The hashCode() method is used by the HashPartitioner (the default partitioner in MapReduce) to choose a reduce partition, so you should make sure that you write a good hash function that mixes well to ensure reduce partitions are of a similar size.

###### **Code the Mapper**

The Mapper just as the mapper of the wordCount example, takes the combination to two adjacent words and emits the TextPair and a value of ‘1’.

The code for the Mapper is as given below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | 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.mapreduce.Mapper;    public class BigramCountMapper extends Mapper<LongWritable, Text, TextPair, IntWritable>{        private static Text lastWord = null;      private static TextPair textPair = new TextPair();      private static Text wordText = new Text();      private static IntWritable one = new IntWritable(1);        @Override      public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException      {          String line = value.toString();          line = line.replace(",", "");          line = line.replace(".", "");            for(String word: line.split("\\W+"))          {              if(lastWord == null)              {                  lastWord = new Text(word);              }              else              {                  wordText.set(word);                  textPair.set(lastWord, wordText);                  context.write(textPair, one);                  lastWord.set(wordText.toString());              }          }      }  } |

###### **Code the Reducer**

Hadoop takes all the emitted key-value pair from the Mapper and does the sorting and shuffling. After that all the values that have the same TextPair associated with them is put in the iterable list. This value is then provided to the Reducer. In Reducer we just add the values in the list, just as we had done in case of the wordCount.

The code for the Reducer is as given below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import java.io.IOException;    import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;    public class BigramCountReducer extends Reducer<TextPair, IntWritable, Text, IntWritable>{      private static Text textPairText = new Text();      @Override      public void reduce(TextPair key, Iterable values, Context context) throws IOException, InterruptedException      {          int count=0;          for(IntWritable value: values)          {              count += value.get();          }            textPairText.set(key.toString());          context.write(textPairText, new IntWritable(count));      }  } |

###### **Code the Driver**

Finally, we will code the driver class that controls the job. Here we will need to mention the MapperOutputKey class as TextPair.class, which is the custom writable class.

The code for the Driver is as given below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | import java.io.IOException;    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.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;    public class BigramCount {      public static void main(String args[]) throws IOException, InterruptedException, ClassNotFoundException {          if (args.length != 2) {              System.err.println("Inavlid Command!");              System.err.println("Usage: BigramCount <input type="text" /> <output>");              System.exit(0);          }            Configuration conf = new Configuration();          conf.set("mapreduce.jobtracker.address", "local");          conf.set("fs.defaultFS","file:///");            Job job = new Job(conf);            job.setJarByClass(BigramCount.class);          job.setJobName("Word Count");            FileInputFormat.addInputPath(job, new Path(args[0]));          FileOutputFormat.setOutputPath(job, new Path(args[1]));            job.setMapperClass(BigramCountMapper.class);          job.setReducerClass(BigramCountReducer.class);            job.setMapOutputKeyClass(TextPair.class);          job.setMapOutputValueClass(IntWritable.class);            job.setOutputKeyClass(Text.class);          job.setOutputValueClass(IntWritable.class);            System.exit(job.waitForCompletion(true) ? 0 : 1);      }  } |

###### **Setup the input directory in HDFS**

Download ebooks from Project Gutenberg(<http://www.gutenberg.org/>). Save the ebook as plain text in a directory with the name ‘input’.

Later, we need to move this directory in HDFS. To do that, type the following in the terminal:

|  |  |
| --- | --- |
| 1 | $ hadoop-1.1.2/bin/hadoop fs -put ~/Desktop/input/ . |

This will move the directory in HDFS as seen below.

|  |  |
| --- | --- |
| 1  2  3 | $ hadoop-1.1.2/bin/hadoop fs -ls  Found 1 items  drwxr-xr-x   - hadoop supergroup          0 2013-11-20 23:13 /user/hadoop/input |

###### **Run the job**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | $ hadoop-1.1.2/bin/hadoop jar ~/Desktop/bigramCount.jar BigramCount input output  13/11/20 23:13:28 WARN mapred.JobClient: Use GenericOptionsParser for parsing the arguments. Applications should implement Tool for the same.  13/11/20 23:13:28 INFO input.FileInputFormat: Total input paths to process : 1  13/11/20 23:13:28 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable  13/11/20 23:13:28 WARN snappy.LoadSnappy: Snappy native library not loaded  13/11/20 23:13:28 INFO mapred.JobClient: Running job: job\_201311202308\_0003  13/11/20 23:13:29 INFO mapred.JobClient:  map 0% reduce 0%  13/11/20 23:13:35 INFO mapred.JobClient:  map 100% reduce 0%  13/11/20 23:13:43 INFO mapred.JobClient:  map 100% reduce 33%  13/11/20 23:13:45 INFO mapred.JobClient:  map 100% reduce 100%  13/11/20 23:13:46 INFO mapred.JobClient: Job complete: job\_201311202308\_0003  13/11/20 23:13:46 INFO mapred.JobClient: Counters: 26  13/11/20 23:13:46 INFO mapred.JobClient:   Job Counters  13/11/20 23:13:46 INFO mapred.JobClient:     Launched reduce tasks=1  13/11/20 23:13:46 INFO mapred.JobClient:     SLOTS\_MILLIS\_MAPS=5779  13/11/20 23:13:46 INFO mapred.JobClient:     Total time spent by all reduces waiting after reserving slots (ms)=0  13/11/20 23:13:46 INFO mapred.JobClient:     Total time spent by all maps waiting after reserving slots (ms)=0  13/11/20 23:13:46 INFO mapred.JobClient:     Launched map tasks=1  13/11/20 23:13:46 INFO mapred.JobClient:     Data-local map tasks=1  13/11/20 23:13:46 INFO mapred.JobClient:     SLOTS\_MILLIS\_REDUCES=9545  13/11/20 23:13:46 INFO mapred.JobClient:   File Output Format Counters  13/11/20 23:13:46 INFO mapred.JobClient:     Bytes Written=343198  13/11/20 23:13:46 INFO mapred.JobClient:   FileSystemCounters  13/11/20 23:13:46 INFO mapred.JobClient:     FILE\_BYTES\_READ=803716  13/11/20 23:13:46 INFO mapred.JobClient:     HDFS\_BYTES\_READ=274173  13/11/20 23:13:46 INFO mapred.JobClient:     FILE\_BYTES\_WRITTEN=1711913  13/11/20 23:13:46 INFO mapred.JobClient:     HDFS\_BYTES\_WRITTEN=343198  13/11/20 23:13:46 INFO mapred.JobClient:   File Input Format Counters  13/11/20 23:13:46 INFO mapred.JobClient:     Bytes Read=274059  13/11/20 23:13:46 INFO mapred.JobClient:   Map-Reduce Framework  13/11/20 23:13:46 INFO mapred.JobClient:     Map output materialized bytes=803716  13/11/20 23:13:46 INFO mapred.JobClient:     Map input records=4893  13/11/20 23:13:46 INFO mapred.JobClient:     Reduce shuffle bytes=803716  13/11/20 23:13:46 INFO mapred.JobClient:     Spilled Records=93962  13/11/20 23:13:46 INFO mapred.JobClient:     Map output bytes=709748  13/11/20 23:13:46 INFO mapred.JobClient:     Total committed heap usage (bytes)=269619200  13/11/20 23:13:46 INFO mapred.JobClient:     Combine input records=0  13/11/20 23:13:46 INFO mapred.JobClient:     SPLIT\_RAW\_BYTES=114  13/11/20 23:13:46 INFO mapred.JobClient:     Reduce input records=46981  13/11/20 23:13:46 INFO mapred.JobClient:     Reduce input groups=24292  13/11/20 23:13:46 INFO mapred.JobClient:     Combine output records=0  13/11/20 23:13:46 INFO mapred.JobClient:     Reduce output records=24292  13/11/20 23:13:46 INFO mapred.JobClient:     Map output records=46981 |

You can now view the output from HDFS itself or download the directory on the local hard disk using the get command.

The output would look similar to the following:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | ...  command of  4  command the 1  commanded by    4  commanded the   1  commanded with  2  commander 10    1  commander Colonel   2  commander General   1  commander Prince    1  commander dated 1  commander decided   1  commander hastily   1  commander of    8  commander sent  1 |