Map Reduce Programming

A Weather Dataset, Understanding Hadoop API for MapReduce Framework (Old and New), Basic Concepts Hadoop MapReduce: Driver code, Mapper code, Reducer code, Record Reader, Combiner, Partitioner.

Introduction

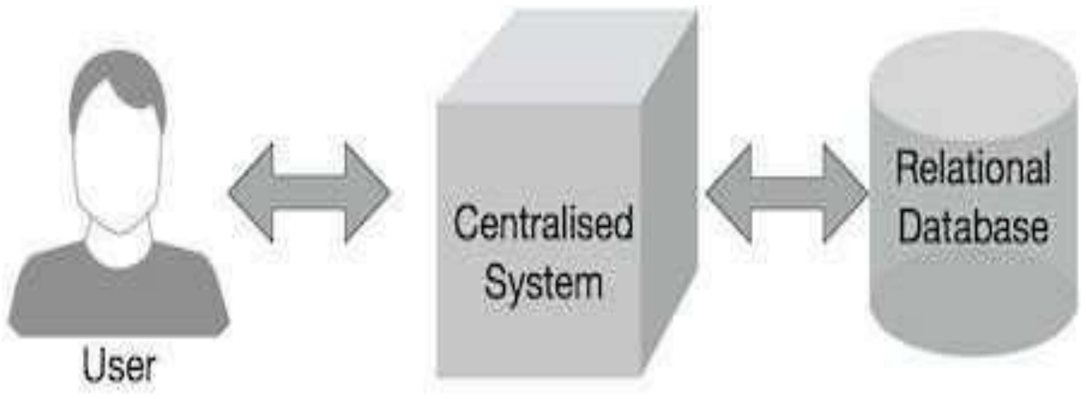
- **MapReduce** is the data processing layer (processing technique) of Hadoop
- **MapReduce(MR)** is a software framework and programming model that allows to perform distributed and parallel processing on large data sets stored in Hadoop Distributed File System(HDFS)
- In MapReduce, a job is divided among multiple nodes and each node works with a part of the job simultaneously. So, Mapreduce is based on **Divide and conquer** paradigm that helps us process the data in a quick manner.
- The core idea behind **MapReduce** is mapping your data set into a collection of <key, value> pairs, and then reducing over all pairs with the same key.

Cont..

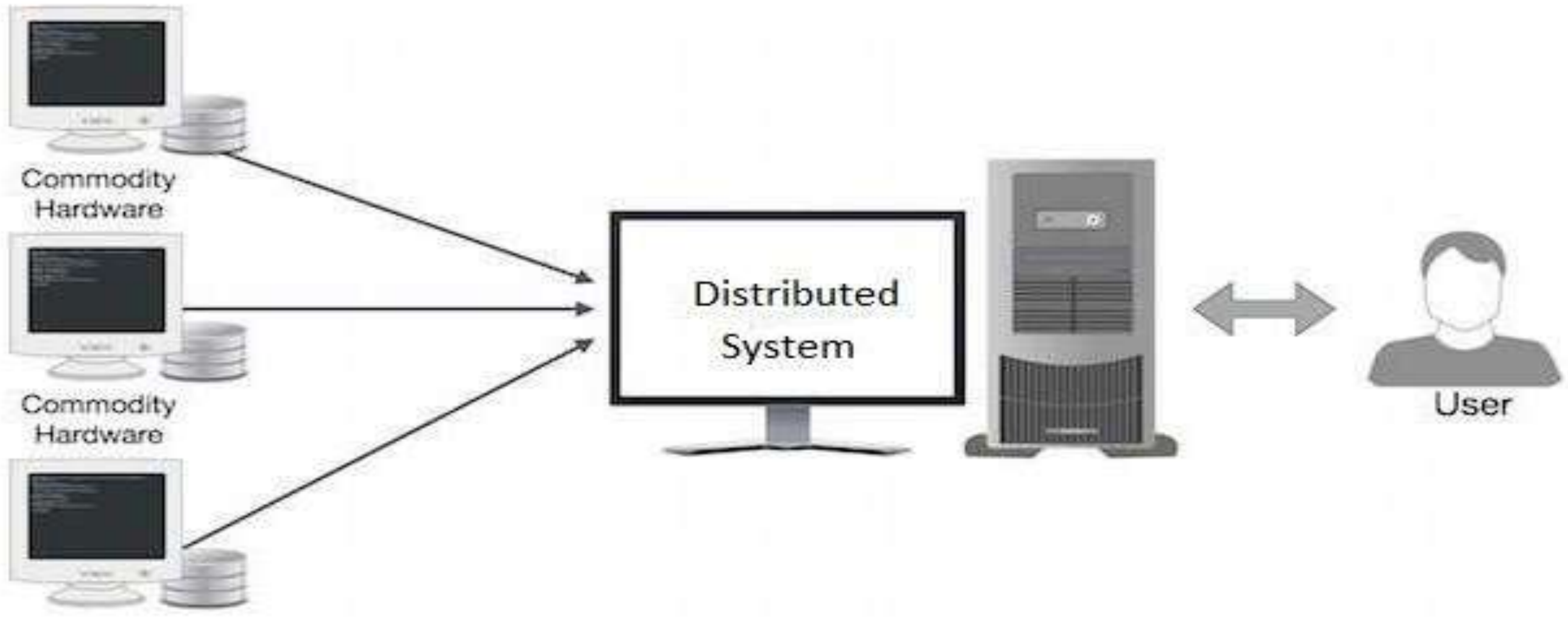
- In Hadoop, MapReduce works by breaking the data processing **into two distinct phases**
  1. Map
  2. Reduce
- **Map tasks deal with splitting and mapping of data while Reduce tasks shuffle and reduce the data**
- A MapReduce Program process data by manipulating (key/value) pairs in the general form:
  - ✓ **Map:** (K1,V1) -----∟ List(K2,V2)
  - ✓ **Reduce:** (K2,V2) -----∟ (K3,V3)
- Hadoop is capable of running MapReduce programs written in various languages: Java, Ruby, Python, and C++.
- MapReduce programs are parallel in nature, thus are very useful for performing large-scale data analysis using multiple machines in the cluster

Why MapReduce

- Traditional Enterprise Systems normally have a centralized server to store and process data.
- The following illustration depicts a schematic view of a traditional enterprise system. Traditional model is certainly not suitable to process huge volumes of scalable data and cannot be accommodated by standard database servers. Moreover, the centralized system creates too much of a bottleneck while processing multiple files simultaneously



- This bottleneck issue can be solved by using an algorithm called MapReduce. MapReduce divides a task into small parts and assigns them to many computers. Later, the results are collected at one place and integrated to form the result dataset



Advantage of using MapReduce

- 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 non-trivial (not an easy job).
- 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 only a configuration change.
- This simple scalability is what has attracted many programmers to use the MapReduce model.

How MapReduce Works

- The **MapReduce algorithm** contains two important tasks, namely Map and Reduce.

Map-

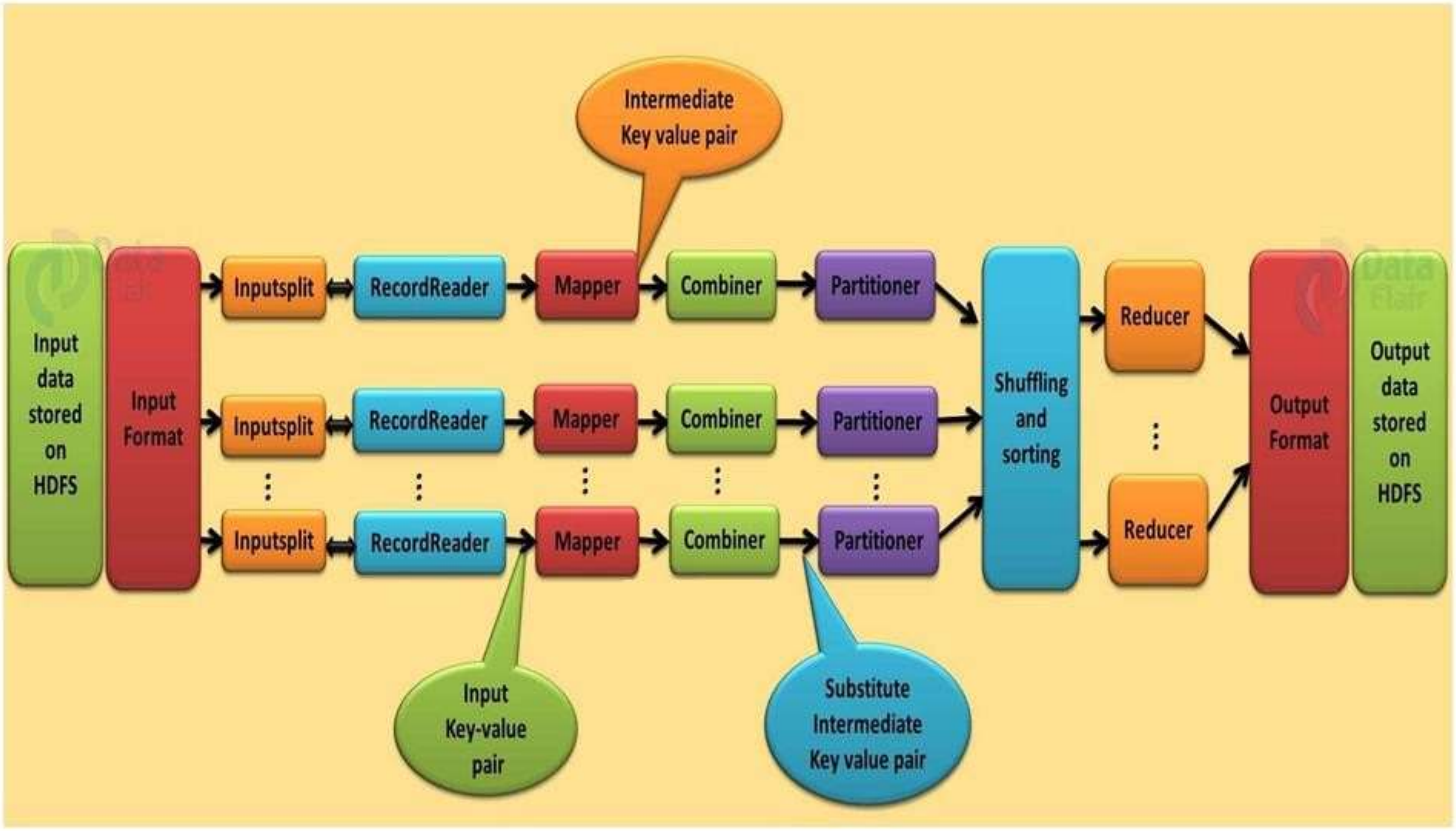
- The map or mapper's job is to process the input data.
- Generally the input data is in the form of file or directory and is stored in the Hadoop file system (HDFS). The input file is passed to the mapper function line by line. The mapper processes the data and creates several small chunks of data
- The Map task (or) Map function reads the input files as (key,value) pairs, process each input and generate zero or more output (key,value) pairs
- The Map function extends Mapper Class which is a subclass of **org.apache.Hadoop.mapreduce**
- The Map function is also good place to filter any unwanted fields/data from input files, we can get data only that are interested and remove unnecessary data

Reduce-

- 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.
- The reducer task reads the intermediate outputs (key-value pairs) generated by the different mappers, process it and generates a key to a smaller set of values.
- The reduce task extends reducer class which is a subclass of **org.apache.Hadoop.mapreduce**
- The output of the reduce task is typically written to the filesystem

- 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

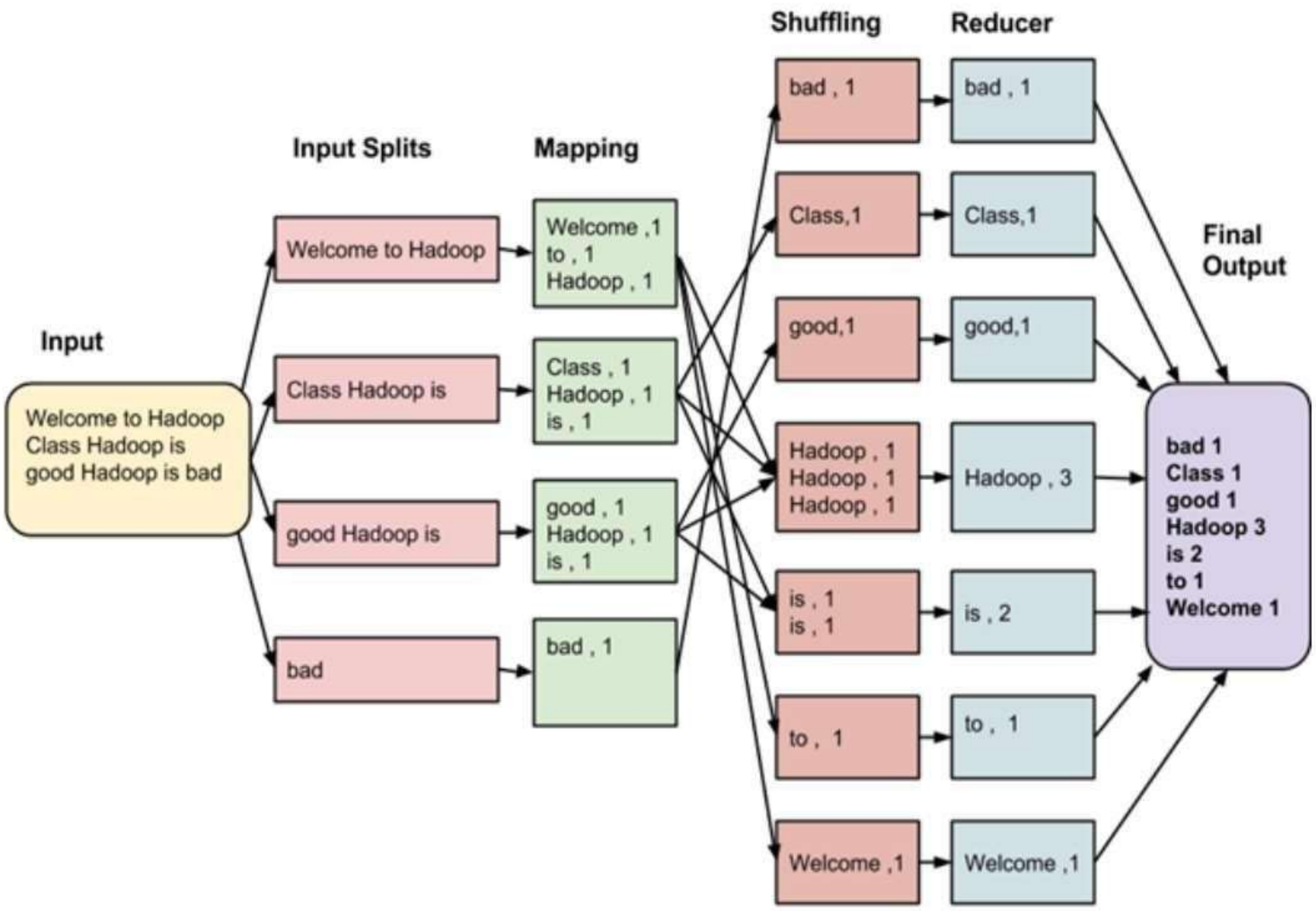
MapReduce Data(Execution) Flow



The whole process goes through a series of steps namely:

- Inputsplits
- InputFormat
- RecordReader
- Mapper
- Combiner
- Partitioner
- shuffling, and Sorting
- Reducer
- OutputFormat

Word Count Example to Understand MapReduce



Input
Welcome to Hadoop Class Hadoop is good Hadoop is bad
Input Splits
Welcome to Hadoop
Class Hadoop is
good Hadoop is
bad
Mapping
Welcome ,1 to , 1 Hadoop , 1
Class , 1 Hadoop , 1 is , 1
good , 1 Hadoop , 1 is , 1
bad , 1
Shuffling
bad , 1
Class,1
good,1
Hadoop , 1 Hadoop , 1 Hadoop , 1
is , 1 is , 1
to , 1
Welcome ,1
Reducer
bad , 1
Class,1
good,1
Hadoop , 3
is , 2
to , 1
Welcome ,1
Final Output
bad 1 Class 1 good 1 Hadoop 3 is 2 to 1 Welcome 1

- First, we divide the **input into splits** as shown in the figure. This will distribute the work among all the map nodes.
- Then, **we tokenize the words** in each of the mapper and give a hardcoded value (1) to each of the tokens or words. The rationale behind giving a hardcoded value equal to 1 is that every word, in itself, will occur once.
- Now, a list of key-value pair will be created where the key is nothing but the individual words and value is one. So, for the first line (welcome to Hadoop) we have 3 key-value pairs – **(<welcome,1>; <to, 1>;<Hadoop, 1>).** The **mapping process** remains the same on all the nodes.
- After mapper phase, **a partition process takes place where sorting and shuffling happens.** so that all the tuples with the same key are sent to the corresponding reducer.

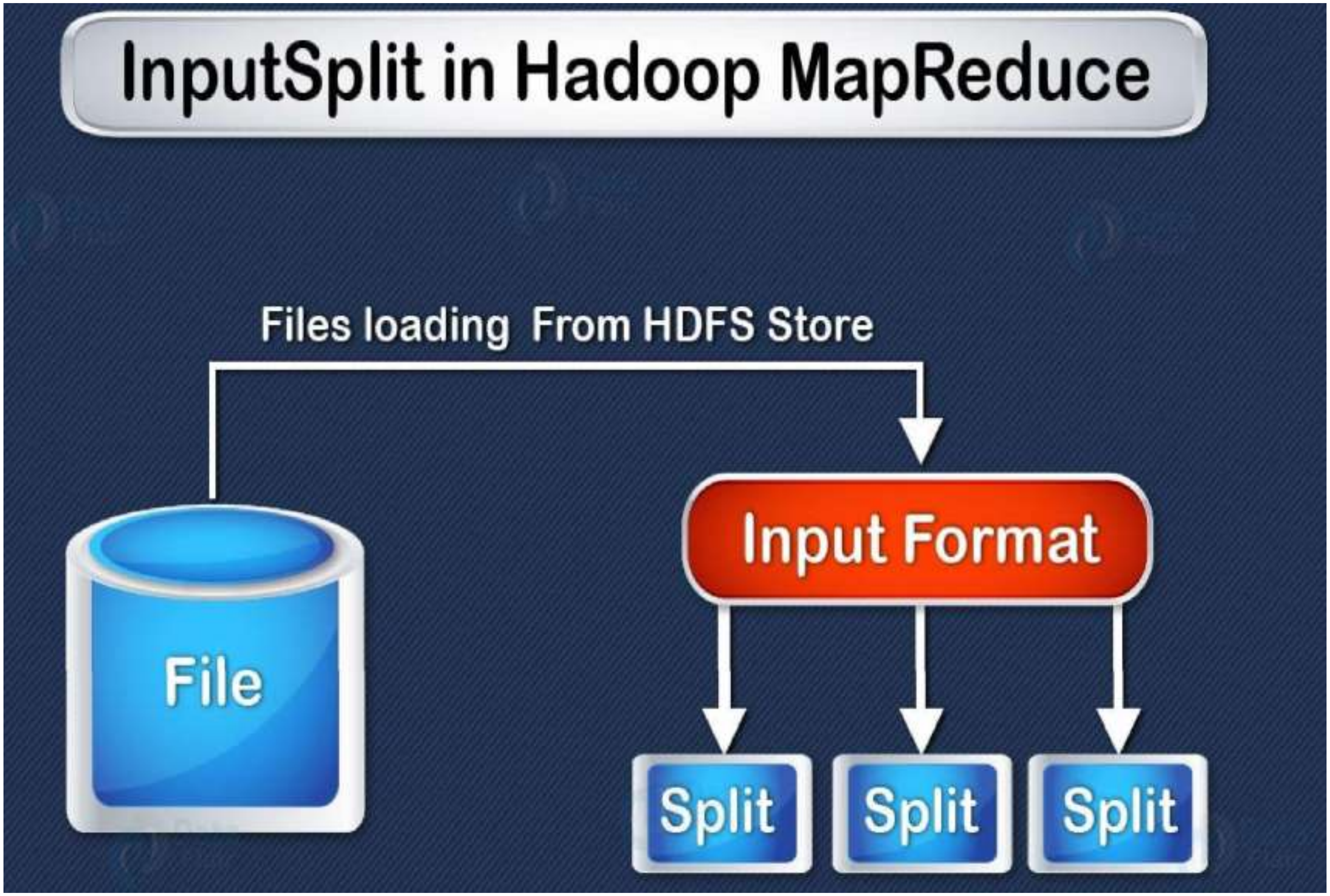
- So, after the sorting and shuffling phase, **each reducer** will have a unique key and a list of values corresponding to that very key. For example, **Hadoop [1,1,1]; is [1,1]; etc..**
- Now, each **Reducer** counts the values which are present in that list of values. As shown in the figure, reducer gets a list of values which is [1,1,1] for the key Hadoop. Then, it counts the number of ones in the very list and gives the final output as – **<Hadoop, 3>**.
- Finally, all the output key/value pairs are then collected and written in the output file

InputSplit:

- The data for a MapReduce task is stored in **input files**, and input files typically lives in **HDFS.**
- **In MapReduce framework, on job start, each input file is broken into splits and each map processes a single split**
- **An Inputsplit** is the logical representation of data. It describes a unit of work that contains a single map task in a MapReduce program.
- To get split details of an input files, Hadoop provides an InputSplit class in Org.apache.Hadoop.mapreduce package and it implementation is as follows:

```
public abstract class InputSplit
 {
     public abstract long getLength() throws IOException,
     InterruptedException
     public abstract string getLocation() throws IOException,
     InterruptedException
 }
```

- From the two methods we can get the length of a split and storage location. A good input split size is equal to the HDFS block size.


InputSplit in Hadoop MapReduce
Files loading From HDFS Store
File
Input Format
Split
Split
Split

Usually, input split is configured same as the size of block size but consider if the input split is larger than the block size. Input split represents the size of data that will go in one mapper. Consider below

Example:

Input split = 256MB

Block size = 128 MB

Then, mapper will process two blocks that can be on different machines. Which means to process the block the mapper will have to transfer the data between machines to process.

InputFormat Class:

- An Hadoop **InputFormat** is the first component in Map-reduce, it is responsible for **creating the input splits and dividing them into records of <key-value> pairs**, which are processed by map tasks one record at a time.
- FileInputFormat, by default, breaks a file into **128MB** chunks (same as blocks in HDFS)
- A MapReduce programmer doesn't care about producing the Input splits, because Hadoop provides **InputFormat Class in Org.apache.Hadoop.mapreduce package for the below two responsibilities:**

1. To provide details on how to split an InputFile into splits
2. InputSplit may contains any number of blocks(usually 1), but each mapper processes one input split
3. To create a RecordReader class that will generate the series of key/value pairs from a split

InputFormatClass (Contd.)

- 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;

}

- InputSplit in Hadoop is user defined. User can control split size according to the size of data in MapReduce program. Thus the number of map tasks is equal to the number of InputSplits

InputFormatClass (Contd.)

Data sets are specified by InputFormats

- Defines input data (e.g., a directory)
- Identifies partitions of the data that form an InputSplit
- Factory for RecordReader objects to extract (k, v) records from the input source

Example:

A very simple input format is:

- **TextInputFormat**

  Key: Line number

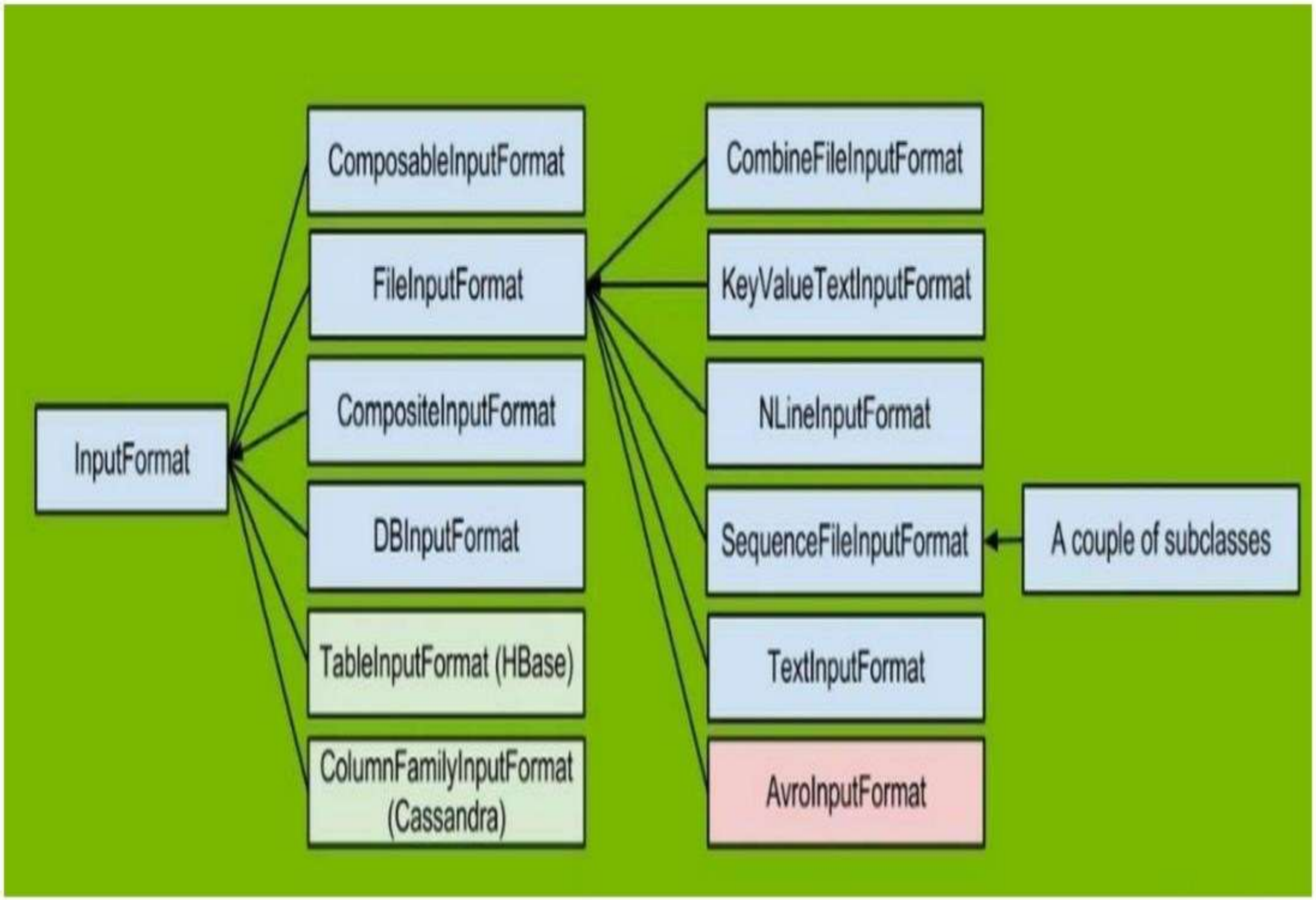
  Value: A line of text from the inputfile

- **KeyValueTextInputFormat**
  - Key and value are separated by a separator (tab by default)
- **SequenceFileInputFormat**

  of binary<key,value> pairs

- **AvroInputFormat**
  - ✓ Avro supports rich data structures (not necessarily pairs) serialized to files Or messages
  - ✓ Compact, fast, language-independent, self-describing, dynamic

InputFormat class Hierarchy



Record Reader:

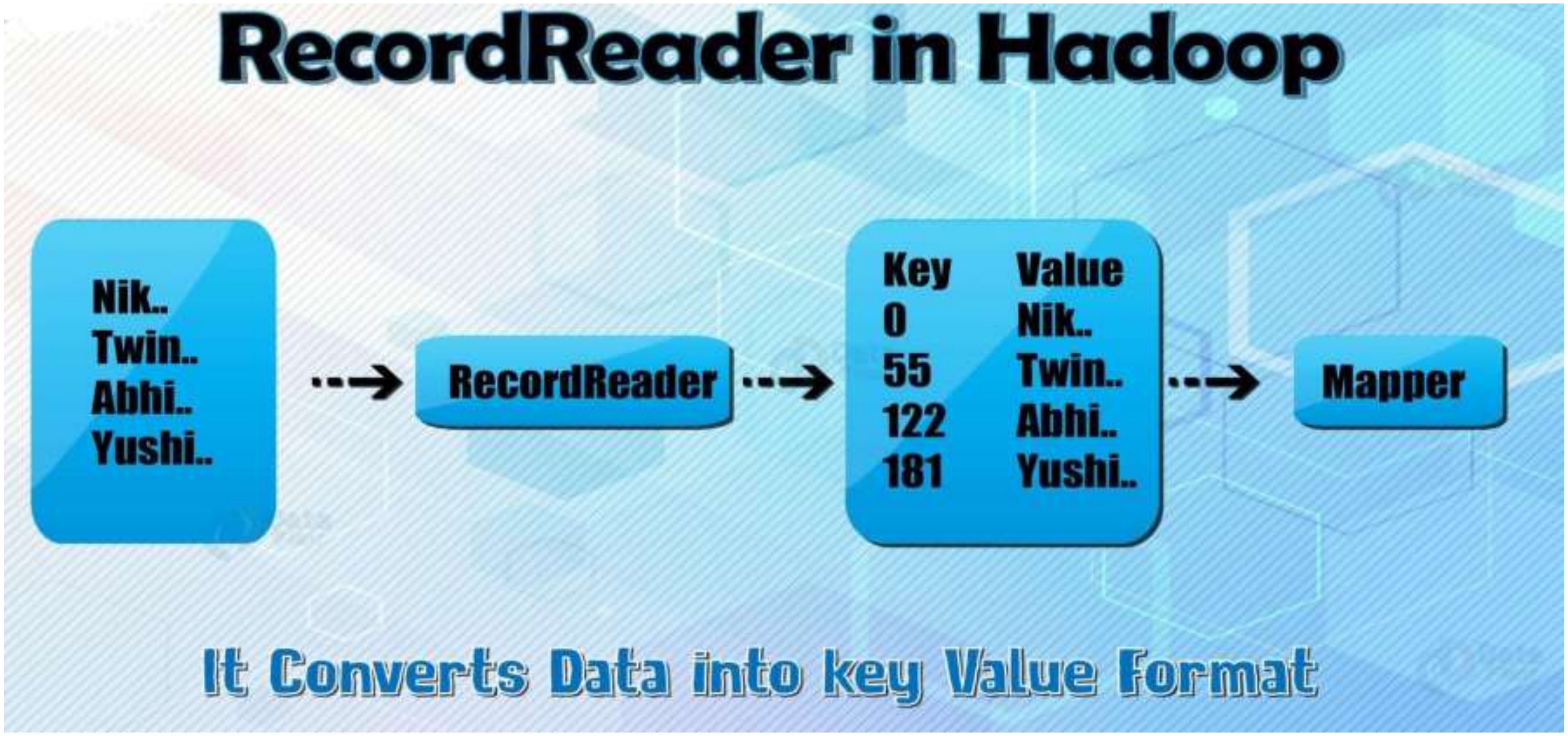
- It communicates with the **InputSplit** in Hadoop MapReduce and converts the data into key-value pairs suitable for reading by the mapper.
- By default, it uses TextInputFormat for converting data into a key-value pair. RecordReader communicates with the InputSplit until the file reading is not completed.
- It assigns byte offset (unique number) to each line present in the file. Further, these key-value pairs are sent to the mapper class for further processing
- The RecordReader class is an abstract class in the org.package.Hadoop.mapreduce package

Implementation of Record Reader Class

```
Public abstract class RecordReader<key,value> implements closeable
{
    public abstract void initialize(InputSplit split, taskAttempt Context 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;
}
```

Mapper's run() method:- Mapper class run() method retrives the (key,value) pairs from context by calling getCurrentKey() and getCurrentValue() methods and passed onto **map() method for further processing** of the records.

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


RecordReader in Hadoop
Nik..
Twin..
Abhi..
Yushi..
RecordReader
Key
Value
0
Nik..
55
Twin..
122
Abhi..
181
Yushi..
Mapper
It Converts Data into key Value Format

Mapper

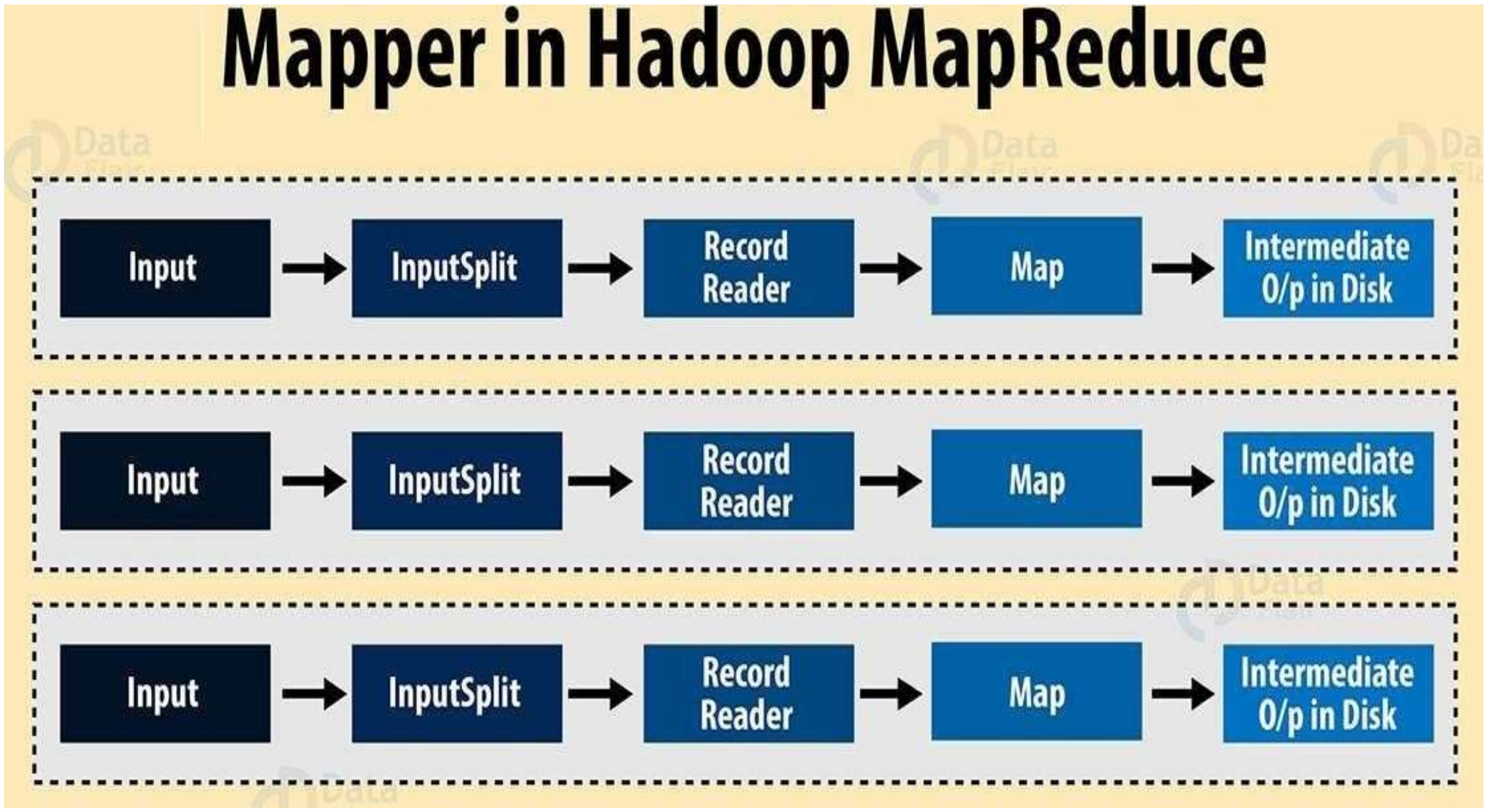
- Mapper is the first phase of processing,It processes each input record (from RecordReader) and generates new key-value pair, and this key-value pair generated by Mapper is completely different from the input pair.
- The output of Mapper is also known as **intermediate output** which is written to the local disk.
- The output of the Mapper is not stored on HDFS as this is temporary data and writing on HDFS will create unnecessary copies (also HDFS is a high latency system).
- Mappers output is passed to the combiner for further process

How Mapper Work?

- Initially InputSplits converts the physical representation of the block into logical for the Hadoop mapper.
- RecordReader's responsibility is to keep reading/converting data into key-value pairs until the end of the file. Byte offset (unique number) is assigned to each line present in the file by RecordReader. Further, this key-value pair is sent to the mapper
- The output of the mapper program is called as intermediate data (key-value pairs which are understandable to reduce).
- InputFormat determines the number of maps i.e.

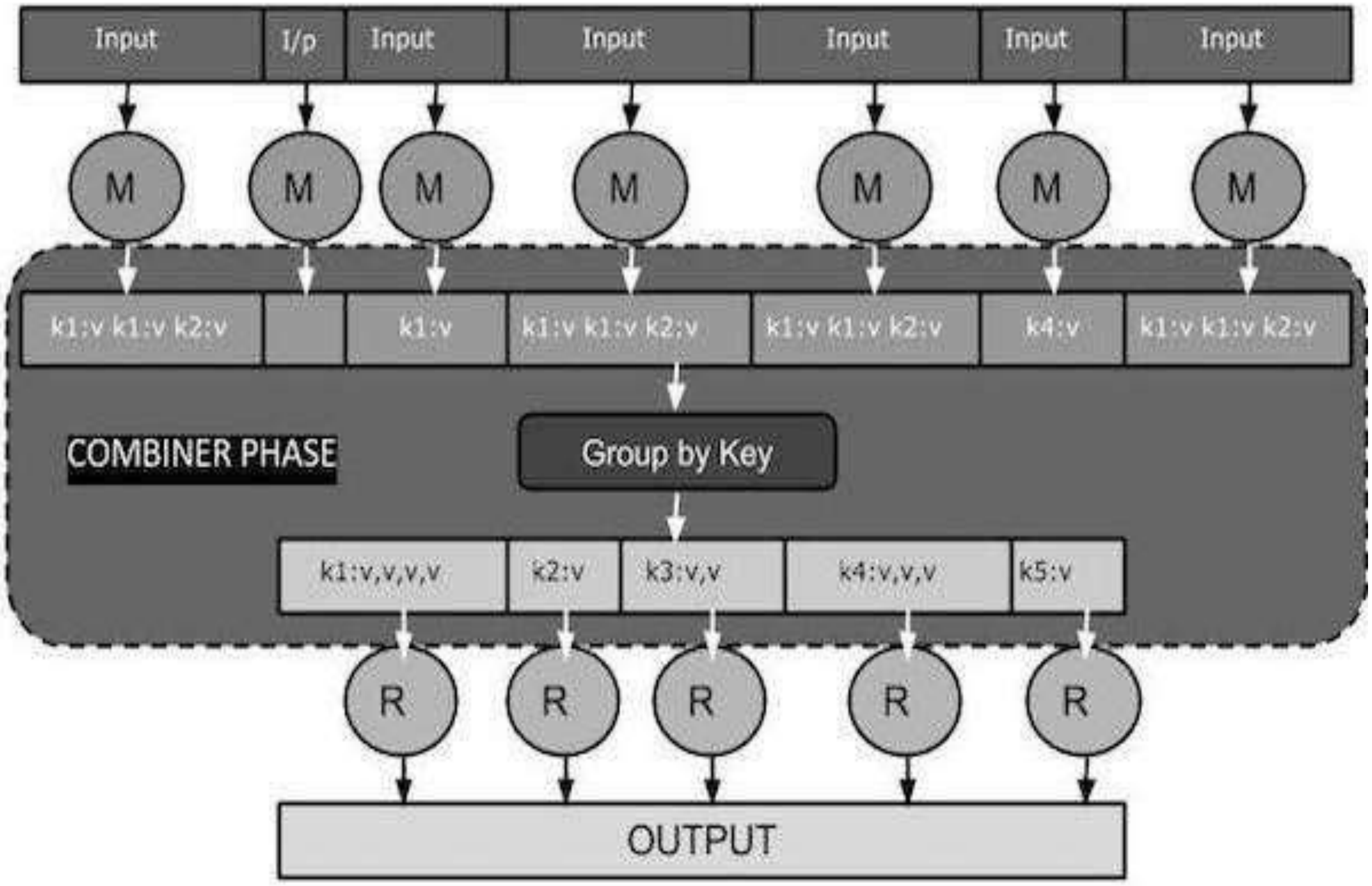
$$\textbf{No. of Mappers} = \frac{\textbf{Total Data Size}}{\textbf{InputSplit size}}$$

- For example, if data size is 1 TB and InputSplit size is 100MB then, No. of Mappers= (1000*1000)/100= 10,000


Mapper in Hadoop MapReduce
Input
InputSplit
Record Reader
Map
Intermediate O/p in Disk
Input
InputSplit
Record Reader
Map
Intermediate O/p in Disk
Input
InputSplit
Record Reader
Map
Intermediate O/p in Disk

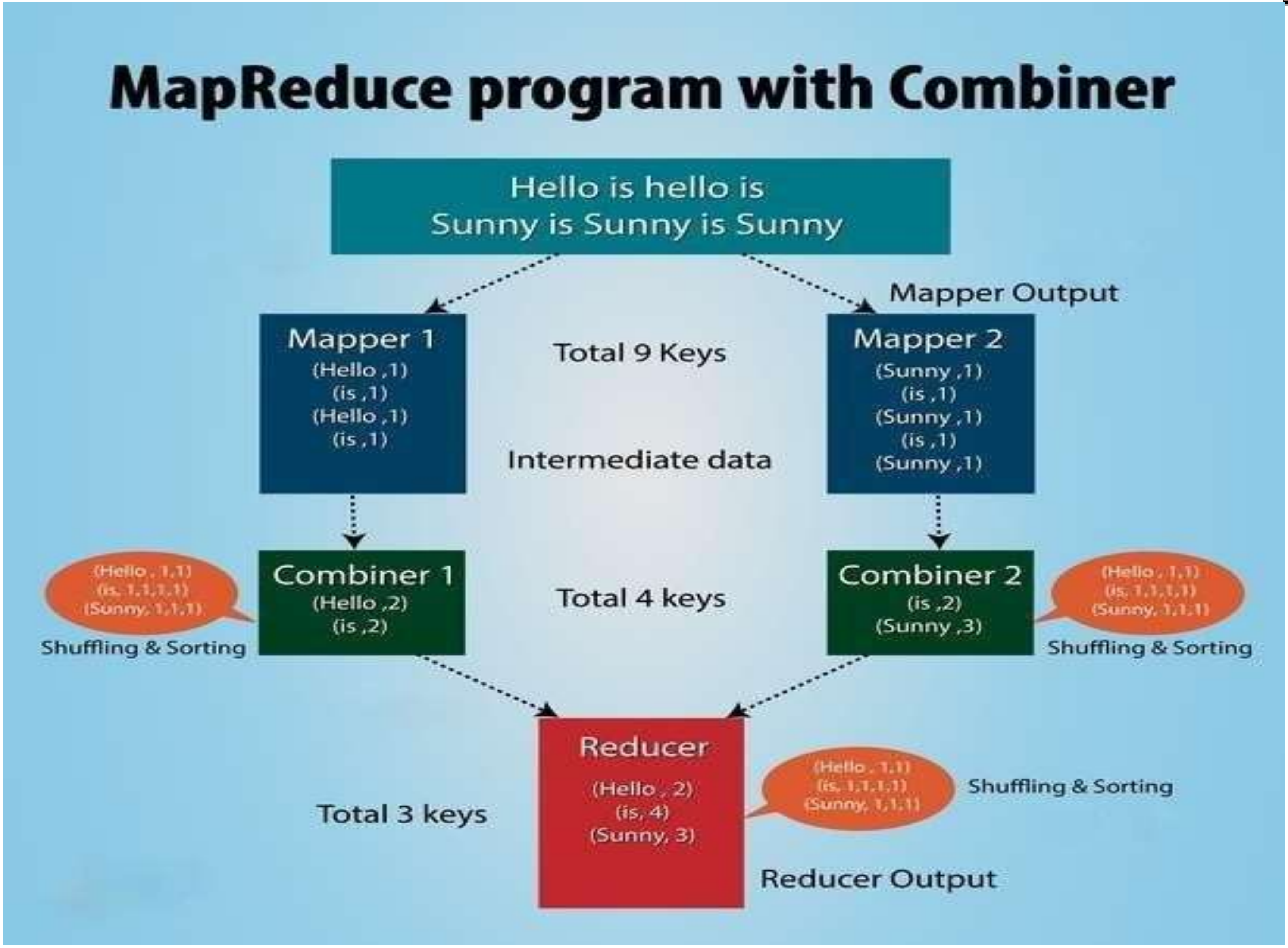
Combiner

- When the mapreduce job is to run on a large data set, Hadoop Mapper generates large chunks of intermediate data that is passed on to Hadoop reducer for further processing which leads to massive network congestion.
- **To reduce the network congestion, Mapreduce framework offers a function known as Combiner**
- The combiner is also known as '**Mini-reducer'.** The role of Combiner is to perform local aggregation on the mappers' output, which helps to minimize the data transfer between mapper and reducer
- The output of the combiner will sent over the network to the actual reducer task as input


Input
I/p
Input
Input
Input
Input
Input
M
k1:v k1:v k2:v
k1:v
k1:v k1:v k2:v
k1:v k1:v k2:v
k4:v
k1:v k1:v k2:v
COMBINER PHASE
Group by Key
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
OUTPUT

How Combiner Works:

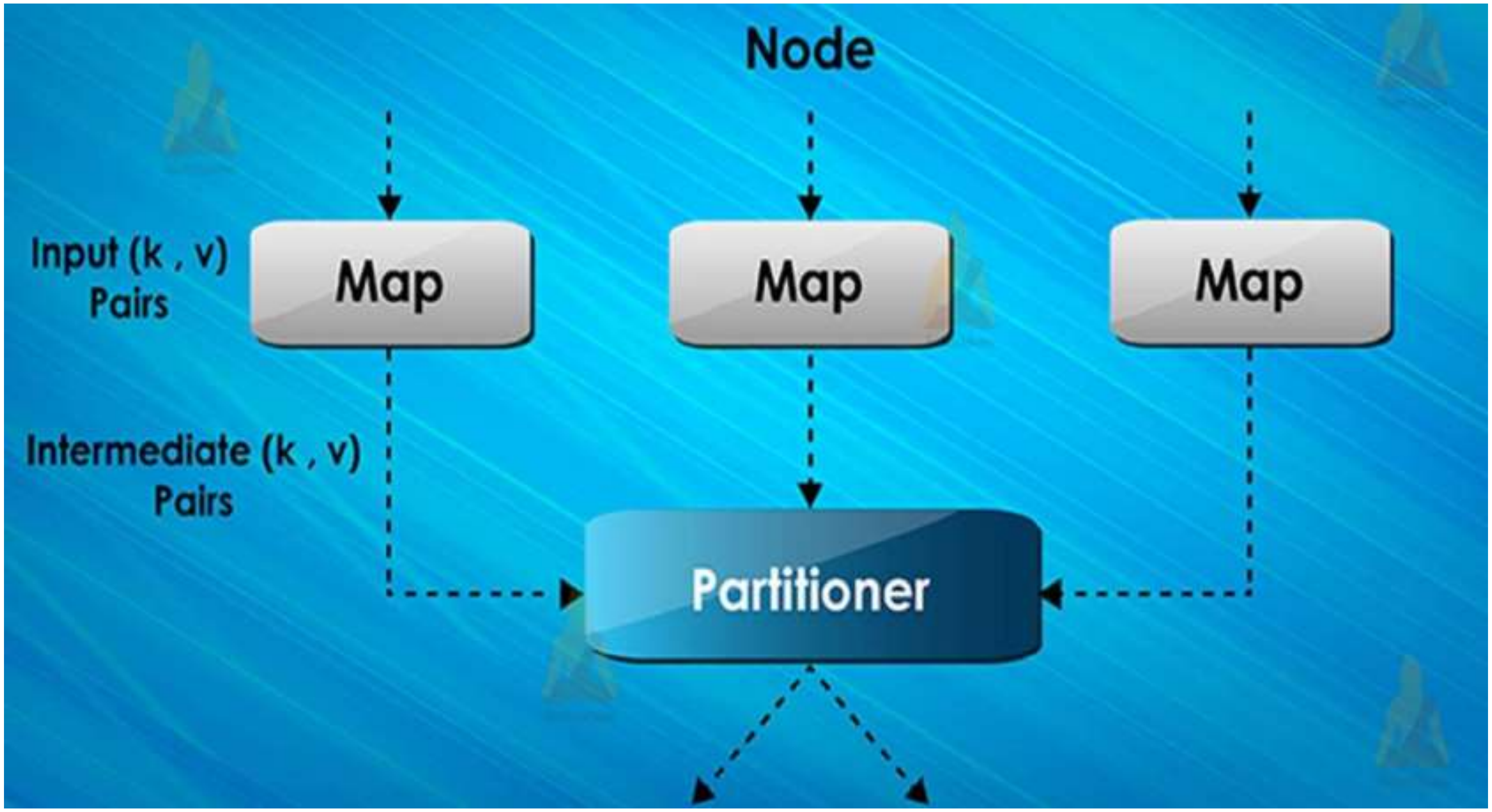
- A combiner does not have a predefined interface and it must implement the Reducer interface's reduce() method.
- A combiner operates on each map output key. It must have the same output key-value types as the Reducer class.
- A combiner can produce summary information from a large dataset because it replaces the original Map output
- **A Combiner is an optional class**, yet it helps segregating data into multiple groups for Reduce phase, which makes it easier to process.


MapReduce program with Combiner
Hello is hello is Sunny is Sunny is Sunny
Mapper Output
Mapper 1 (Hello ,1) (is ,1) (Hello ,1) (is ,1)
Total 9 Keys
Mapper 2 (Sunny ,1) (is ,1) (Sunny ,1) (is ,1) (Sunny ,1)
Intermediate data
(Hello , 1,1) (is, 1,1,1,1) (Sunny, 1,1,1)
Shuffling & Sorting
Combiner 1 (Hello ,2) (is ,2)
Total 4 keys
Combiner 2 (is ,2) (Sunny ,3)
(Hello , 1,1) (is, 1,1,1,1) (Sunny, 1,1,1)
Shuffling & Sorting
Total 3 keys
Reducer (Hello , 2) (is, 4) (Sunny, 3)
(Hello , 1,1) (is, 1,1,1,1) (Sunny, 1,1,1)
Shuffling & Sorting
Reducer Output

Partitioner

- In MapReduce framework, a map/reduce job will contain more than one reducers. When mapper emits the key/value pair, it has to be sent to one of the reducers. **The Mechanism of sending specific key/value pairs to specific reducer is called Partitioning**
- **The Partitioner phase takes place after the map phase and before the reduce phase**
- In Hadoop, the default Partitioner is HashPartitioner, which hashes a record’s key to determine which partition the record belongs to.Thus the no. of partitions is equal to no. of reducer tasks

Partitioner In MapReduce



Example:

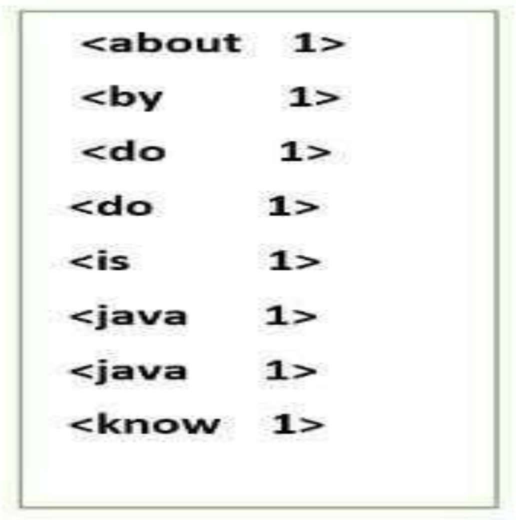
Suppose we have two reducers, consider the wordcount example, the input data is:

<1, what do you mean by object>

<2, what do you know aboutjava>

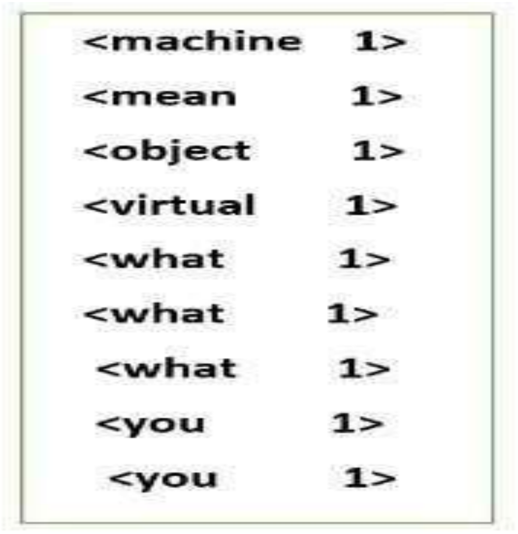
<3, What is Java Virtual Machine>

The Partitioner partitions the data and emitted by the mapper as:



<about	1>
<by	1>
<do	1>
<do	1>
<is	1>
<java	1>
<java	1>
<know	1>

Reducer-1

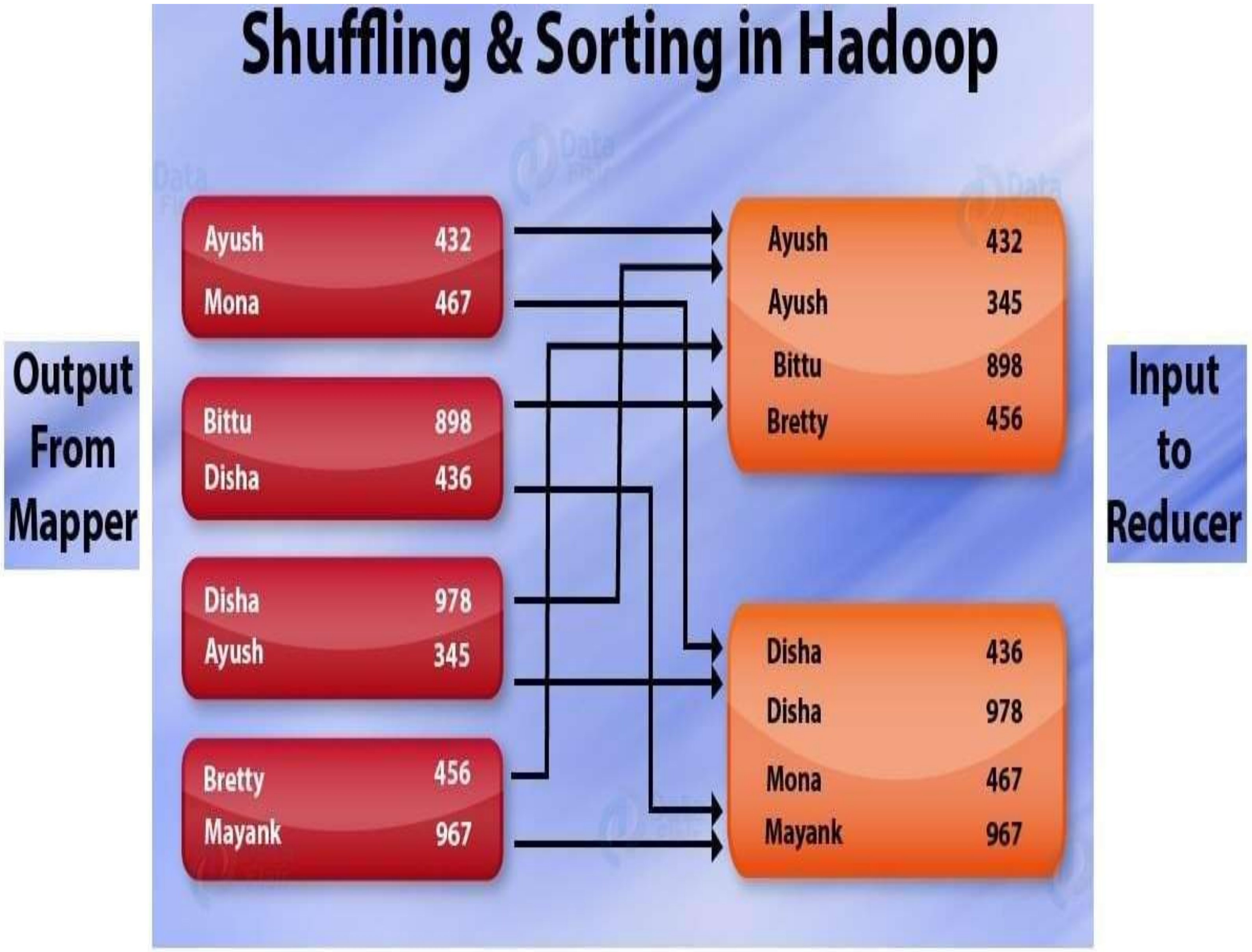


<machine	1>
<mean	1>
<object	1>
<virtual	1>
<what	1>
<what	1>
<what	1>
<you	1>
<you	1>

Reducer-2

Shuffling and Sorting

- In Hadoop, the process by which the intermediate output from **mappers** is transferred to the **reducer** is called **Shuffling.**
- **Shuffle phase** is necessary for the reducers, otherwise, they would not have any input from every mapper
- **Sort Phase** in MapReduce covers the **merging and sorting** of map outputs.
- The keys generated by the mapper are automatically **sorted** by MapReduce Framework,
- i.e. Before starting of reducer, all intermediate key-value pairs in MapReduce that are generated by mapper get sorted by key and not by value. Values passed to each reducer are not sorted; they can be in any order
- **Sorting in Hadoop** helps reducer to easily distinguish when a new reduce task should start. This saves time for the reducer
- Shuffling-Sorting occurs simultaneously to summarize the Mapper intermediate output. Shuffling and sorting in Hadoop MapReduce are not performed at all if you specify zero reducers**(setNumReduceTasks(0)).**


Shuffling & Sorting in Hadoop
Output From Mapper
Ayush 432
Mona 467
Bittu 898
Disha 436
Disha 978
Ayush 345
Bretty 456
Mayank 967
Ayush 432
Ayush 345
Bittu 898
Bretty 456
Disha 436
Disha 978
Mona 467
Mayank 967
Input to Reducer

Reducer

- In Hadoop, **Reducer** takes the output of the mapper(intermediate key-value pair) process each of them to generate the output.
- The output of the reducer is the final output, which is stored in HDFS.
- One can **aggregate, filter, and combine this data** (key, value) in a number of ways for a wide range of processing
- One-one mapping takes place between keys and reducers.
- Reducers run in parallel since they are independent of one another
- The user decides the number of reducers. By **default number of reducers** is 1.

- The Reducer has **3 main phases**:

1. **Shuffle** -Shuffle phase in Hadoop transfers the map output from Mapper to a Reducer in MapReduce
2. **Sort** - Sort phase in MapReduce covers the merging and sorting of map outputs. huffle and sort phase in Hadoop occur simultaneously and are done by the MapReduce framework.
3. **Reduce** - In this phase, after shuffling and sorting, reduce task aggregates the key-value pairs.The *OutputCollector.collect()* method, writes the output of the reduce task to the Filesystem

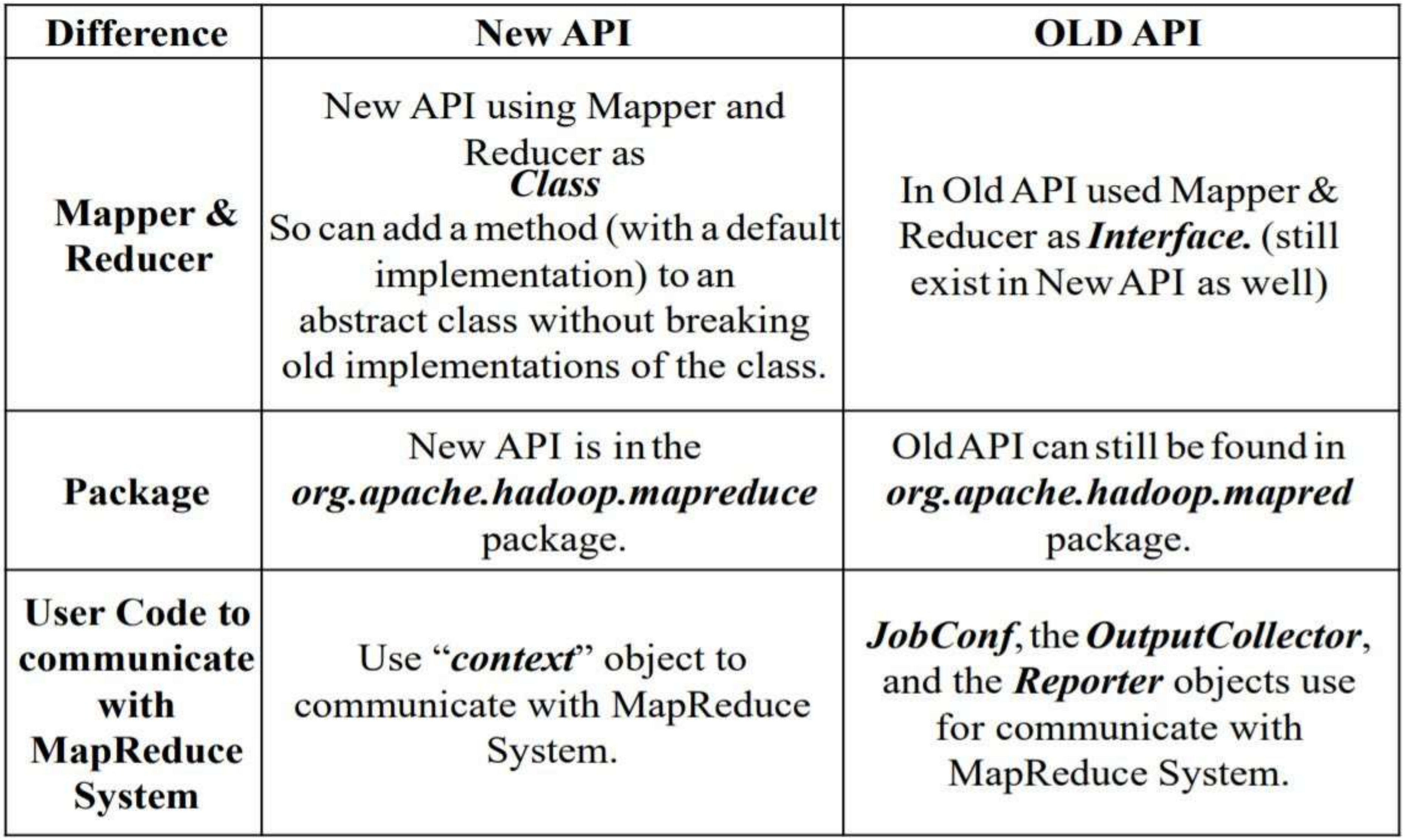
OutputFormat:

Finally Recordwriter writes these output key-value pair Reducer phase to output files that are defined by outputFormat class.

Hadoop API for MapReduce Framework (Old and New)

- Recently Hadoop new version 2.6.0 has released into Market, actually Hadoop versions are released in 3 stages 0.x.xx, 1.x.xx and 2.x.x, Up to Hadoop 0.20 all packages are In Old API (Mapred).
- From Hadoop 0.21 all packages are in New API (Mapreduce)
- Example of New Mapreduce API is **org.apache.hadoop.mapreduce**
- Example of Old Mapreduce API is **org.apache.hadoop.mapred**

Hadoop API for MapReduce Framework (Old and New) –Contd.

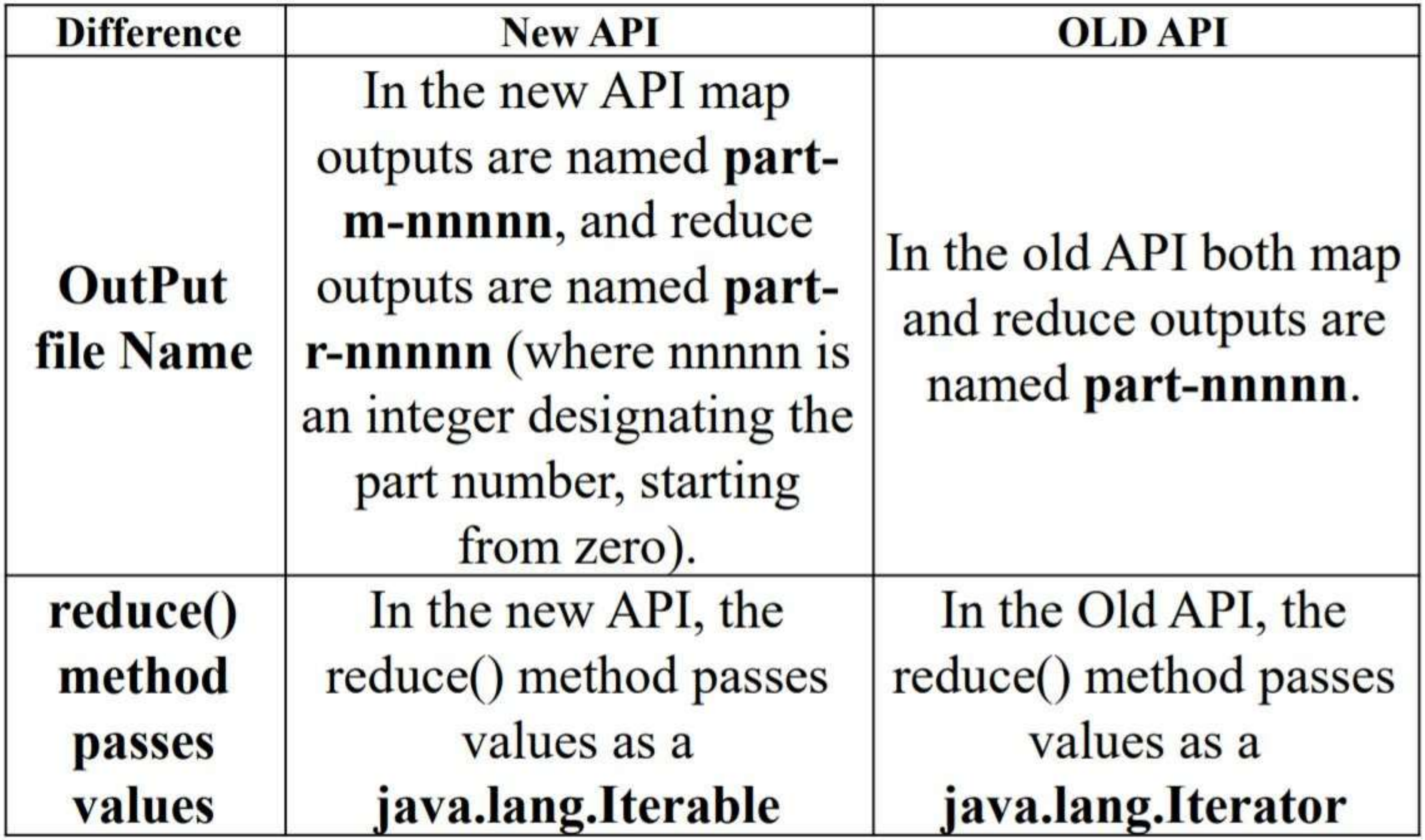


Difference	New API	OLD API
Mapper & Reducer	New API using Mapper and Reducer as ***Class*** So can add a method (with a default implementation) to an abstract class without breaking old implementations of the class.	In Old API used Mapper & Reducer as ***Interface.*** (still exist in New API as well)
Package	New API is in the ***org.apache.hadoop.mapreduce*** package.	Old API can still be found in ***org.apache.hadoop.mapred*** package.
User Code to communicate with MapReduce System	Use "***context***" object to communicate with MapReduce System.	***JobConf***, the ***OutputCollector***, and the ***Reporter*** objects use for communicate with MapReduce System.

Hadoop API for MapReduce Framework (Old and New) –Contd.

Difference	New API	OLD API
Control Mapper and Reducer execution	New API allows both mappers and reducers to control the execution flow by **overriding the run()** method.	Controlling mappers by writing a **MapRunnable**, but no equivalent exists for reducers.
Job control	Job control is done through the **JOB** class in New API.	Job Control was done through **JobClient**. (not exists in the new API)
Job Configuration	Job Configuration done through **Configuration** class via some of the helper methods on Job.	**jobconf** objet was use for Job configuration.which is extension of Configuration class. java.lang.Object extended by org.apache.hadoop.conf.Configuration extended by org.apache.hadoop.mapred.JobConf.

Hadoop API for MapReduce Framework (Old and New) –Contd.



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

Writing MapReduce Program

- **MapReduce** is the programming model to work on data within the HDFS.
- The programming language for MapReduce is Java
- The basic components of a MapReduce Program are:
  - Mapper Code
  - Reducer Code
  - Driver Code
- As the name itself states that, the code is divided into two phases mainly one is map and second is reduce. Both the phases has an input and output as (key,value) pairs

Mapper Code:

- The Mapper code reads the input files as <Key,Value> pairs and emits <Key,Value> pairs.
- The Mapper class extends MapReduceBase and implements the Mapper interface.
- The Mapper interface expects four generics, which define the types of the input and output key/value pairs.
- The first two parameters define the input key and value types, the second two define the output key and value types.
- Some of the common libraries that are included for the Mapper class:
    - import java.io.IOException;
    - import java.util.*;
    - import org.apache.hadoop.io.*;
    - import org.apache.hadoop.mapred.*;

Syntax:

Class **Mapper_className** extends Mapper<Generic Parameters for input and output key/value pairs>

```
{
    public void map(parameters) throws IOException,
    InterruptedException
    {

        // code

    }
}
```

Sample

```
public class MyMapper extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable>
{

public void map(LongWritable key, Text value,
OutputCollector<Text, IntWritable> output, Reporter reporter)
throws IOException
{
                output.collect(key,value);
}

}
```

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

Reducer Code

- The Reducer code reads the outputs generated by the different mappers as <key, value> pairs and emits <key,value> pairs.
- The Reducer class extends MapReduceBase and implements the Reducer interface.
- The Reducer interface expects four generics, which define the types of the input and output key/value pairs.
- The first two parameters define the intermediate key and value types, the second two define the final output key and value types.
- The keys are WritableComparables, the values are Writables.
- Some of the common libraries that are included for the Reducer class:
  - ✓ import java.io.IOException; import java.util.*;
  - ✓ import org.apache.hadoop.io.*;
  - ✓ import org.apache.hadoop.mapred.*

Sample

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

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

Driver Code:

- The Driver code **runs on the client machine** and is responsible for building the configuration of the job and submitting it to the Hadoop Cluster
- The Driver code **will contain the main() method** that accepts arguments from the command line.
- **The code associated with the driver class is referred to as driver code**.
- A driver class is responsible for the **execution of the MapReduce** jobs by passing various parameters using mapper and reducer classes, input, output, etc
- A **Job 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 Job’s setJarByClass() method, which Hadoop will use to locate the relevant JAR file by looking for the JAR file containing this class.
- Having constructed a Job object, now we must 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.
- The **output path** (of which there is only one) is specified by the static setOutputPath() method on FileOutputFormat.

- 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
- 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 **waitForCompletion()** submits the job and waits for it to finish
- The return value of the waitForCompletion() method is a boolean indicating success (true) or failure (false), which we translate into the program's exit code of 0 or 1

- In most cases, the command line parameters passed to the Driver program are the paths to the directory where containing the input files and the path to the output directory.
- Both these path locations are from the HDFS.
- The output location should not be present before running the program as it is created after the execution of the program.
- If the output location already exists the program will exit with an error.
- Some of the common libraries that are included for the Driver class:
  - ✓ Import org.apache.hadoop.fs.Path;
  - ✓ import org.apache.hadoop.io.*;
  - ✓ import org.apache.hadoop.mapred.*

Sample

```
public class MyDriver{
public static void main(String[] args) throws Exception {
// Create the JobConf object
JobConf conf = new JobConf(MyDriver.class);
 // Set the name of the Job
conf.setJobName("SampleJobName");
 // Set the output Key type for the Mapper
conf.setMapOutputKeyClass(Text.class);
 // Set the output Value type for the Mapper
conf.setMapOutputValueClass(IntWritable.class);
 // Set the output Key type for the Reducer
conf.setOutputKeyClass(Text.class);
 // Set the output Value type for the Reducer
conf.setOutputValueClass(IntWritable.class);
```

```
    // Set the Mapper Class
    conf.setMapperClass(MyMapper.class);
     // Set the Reducer Class
    conf.setReducerClass(Reducer.class);
     // Set the format of the input that will be provided to the program
    conf.setInputFormat(TextInputFormat.class);
     // Set the format of the output for the program
    conf.setOutputFormat(TextOutputFormat.class);
    // Set the location from where the Mapper will read the input
    FileInputFormat.setInputPaths(conf,  new Path(args[0]));
     // Set the location where the Reducer will write the output
    FileOutputFormat.setOutputPath(conf,  new Path(args[1]));
     // Run the job on the cluster
    JobClient.runJob(conf);
    }
}
```

WordCount Program using new API

```
Import java.io.IOException;
Import java. util. Iterator
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.mapreduce.Job;
Import org.apache.hadoop.io.LongWritable;
Import org.apache.hadoop.io.Text;
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;
```

// Mapper Code

```
Public class WordCount
{
     Public static class WordMap extends Mapper<LongWritable, Text, Text, Intwritable>
     {
          public final static Intwritable one=new Intwritable(1); Private
          Text word = new Text();
          Public void map(LongWritable key, Text value, Context context) Throws IOException,
          InterruptedException
          {
               StringTokenizer str=new StringTokenizer(value.toString);
               While(str.hasMoreTokens())
               {

               word.set(str.nextToken);
               Context.write(word,one);
               }

                         }
                              }
```

Computer Science & Engineering Department

// Reducer Code

```
Public  static  class WordReduce  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=sum+val.get();
            result.set(sum);
            Context.write(key,result);
       }
 }
```

// Driver Code

```
Public static void main(String  args[]) throws Exception
{
      Configuration  conf=new  Configuration();
      Job job=Job.getInstance(conf,”WordCount”);
      Job.setJarByclass(WordCount.class);
      Job.setMapperClass(WordMap.class);
      Job.setReduceClass(WordReduce.class);
      Job.setCombinerClass(WordReduce.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);
      }
}
```

WordCount program using Old API :

```
import java.io.Exception;
import java.util.*;
import org.apache.hadoop.util.*;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
```

```
public class count1 extends configured implements Tool
{
   public static class Mapclass extends MapReduce Base implements
      Mapper<longwritable,Text,Text,Intwritable>
   {
       private final Intwritable one= new Intwritable(1);
       private Text word=new Text();
       public Void map(longwritable key,Text value,
          outputcollector<Text,Intwritable,output,Reporter reporter)  throws IoExeption
      {
             StringTokenizer str=new StringTokenizer(value.toString())
             while(str.hasmoreTokens())
             {
                  word.set(str.nextToken());
               output.collect(word,one);
              }
      }
}
```

```
public static class Reduceclass extends MapReduce Base impements
Reducer<Text , Intwritable,Text,Intwritable>
{
   private IntWritable result=new IntWritable();
   public void reduce(Text key,Iterator<Intwritable>value,  outputCollector
   <Text,Intwritable> output, Reporter reporter) throws IoException
     {
     int sum=0;
     while(value.hasnext())
        sum=sum+value.next().get();
     result.set(sum);
     output.collect(key,result);
      }
}
```

```
public int run(String args[])throws Exception
{
   configuration conf=getconf();
   JobConf job =new JobConf(conf,count1.class);
   path in =new path(args[0]);
   path out =new path(args[1]);
  FileInputFormat.setInputpath(Job,in);
  FileOutputFormat.setOutputpath(Job,out);
  job.setJobName("count1");
  job.setMapperclass(Mapclass.class);
  job.setReducerclass(Reduceclass.class);
  jon.setOutputKeyClass(Text.class);
  job.setOutputValueClass(Intwritable.class);
  JobClient.runJob(job);
  return 0;
}
public static void main(String args[])throws Exception
{
    int res=ToolRunner.run(new configuration(),new count1(), args);
    System.exit(res);
}
}
```

Anatomy of a MapReduce Job Run:-
A Classic MapReduce (MapReduce1)

- A job run in classic MapReduce has **four independent entities**:

1. The **client**, which submits the MapReduce job.
2. The **jobtracker** acts like a master,which coordinates the job run.
3. The JobTracker is a Java application
4. The **tasktrackers**, which run the tasks that the job has been split into.
5. The **distributed filesystem**, which is used for sharing job files between the other entities

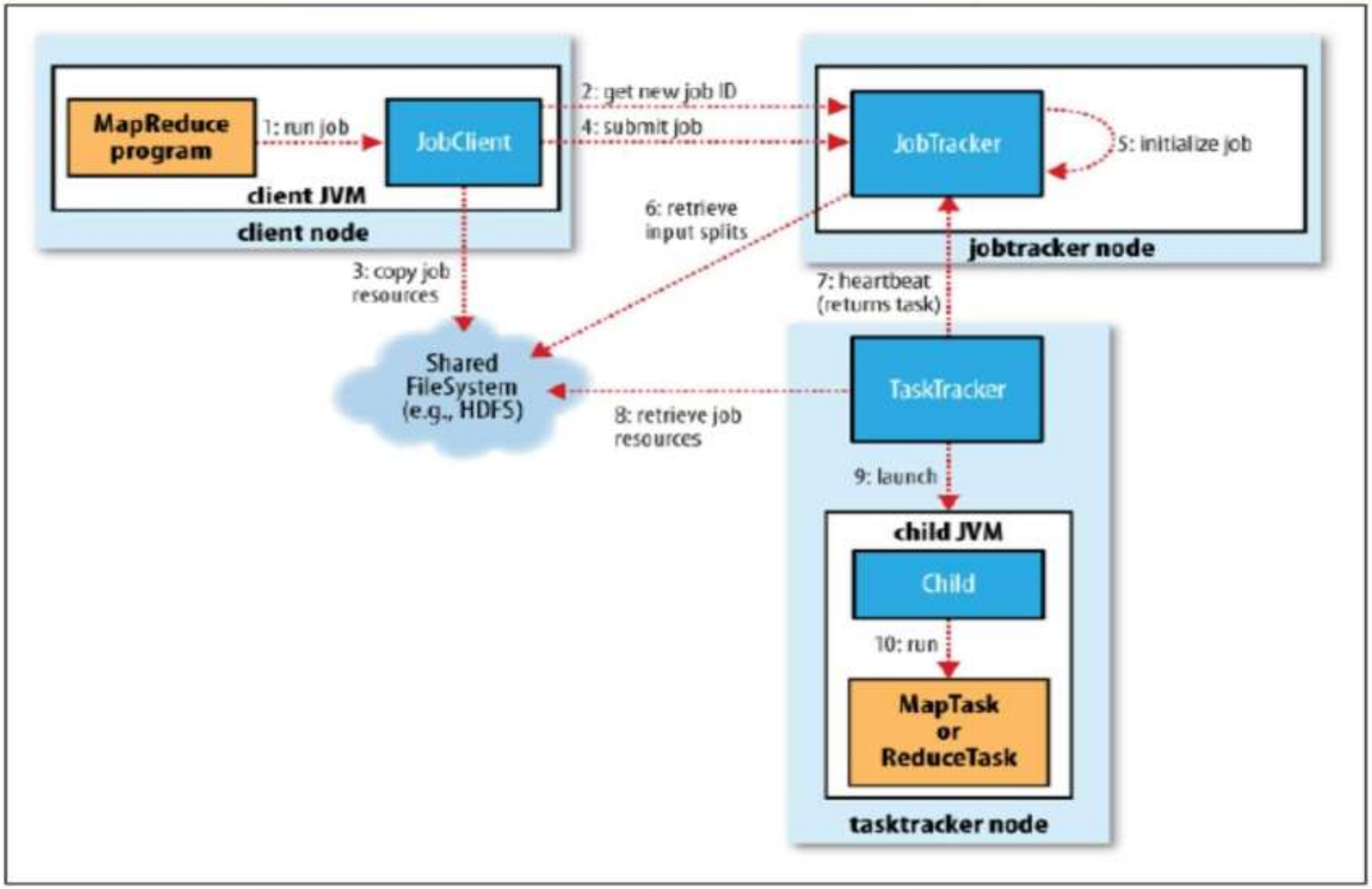


Figure 1: Classic MapReduce

Job Submission:

- **The runJob() method** on JobClient is a convenience method that creates a new JobClient instance and calls submitJob() on it .
- Having submitted the job, runJob() **polls the job's progress once a second** and reports the progress to the console if it has changed since the last report. When the job is complete, if it was successful, the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.
- The job submission process implemented by JobClient's submitJob() method does the following:
  - ☐ Asks the jobtracker for a **new job ID** (by calling getNewJobId() on JobTracker) **(step 2).**
  - ☐ **Checks the output specification** of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.

- **Computes the input splits for the job**. If the splits cannot be computed, because the input paths don’t exist, for example, then the job is not submitted and an error is thrown to the MapReduce program.
- **Copies the resources needed to run the job**, including the job JAR file, the configuration file, and the computed input splits, to the jobtracker’s filesystem in a directory named after the job ID. The job JAR is copied with a high replication factor (controlled by the mapred.submit.replication property, which defaults to 10) so that there are lots of copies across the cluster for the tasktrackers to access when they run tasks for the job**(step3)**
- Tells the jobtracker that **the job is ready for execution** (by calling submitJob() on JobTracker) **(step 4).**

Job Initialization:

- When the JobTracker receives a call to its submitJob() method, it puts it into an internal queue from where the job scheduler will pick it up and initialize it.
- Initialization involves creating an object to represent the job being run, which encapsulates its tasks, and bookkeeping information to keep track of the tasks' status and progress **(step 5).**
- To create the list of tasks to run, the job scheduler first retrieves the input splits computed by the JobClient from the shared filesystem **(step 6).**
- It then creates one map task for each split. The number of reduce tasks to create is determined by the mapred.reduce.tasks property in the JobConf, which is set by the setNumReduceTasks() method, and the scheduler simply creates this number of reduce tasks to be run. Tasks are given IDs at this point

Task Assignment:

- Tasktrackers run a simple loop that periodically sends heartbeat method calls to the jobtracker. Heartbeats tell the jobtracker that a tasktracker is alive, but they also double as a channel for messages. As a part of the heartbeat, a tasktracker will indicate whether it is ready to run a new task, and if it is, the jobtracker will allocate it a task, which it communicates to the tasktracker using the heartbeat return value **(step 7)**.
- Before it can choose a task for the tasktracker, the jobtracker must choose a job to select the task from. There are various scheduling algorithms, but the default one simply maintains a priority list of jobs. Having chosen a job, the jobtracker now chooses a task for the job
- Tasktrackers have a fixed number of slots for map tasks and for reduce tasks: for example, a tasktracker may be able to run two map tasks and two reduce tasks simultaneously

Task Execution:

- Now that the tasktracker has been assigned a task, the next step is for it to run the task.
- First, it localizes the job JAR by copying it from the shared filesystem to the tasktracker’s filesystem. It also copies any files needed from the distributed cache by the application to the local disk.
- Then, it creates a local working directory for the task, and un-jars the contents of the JAR into this directory.
- Third, it creates an instance of TaskRunner to run the task. TaskRunner launches a new Java Virtual Machine **(step 9)** to run each task in **(step 10),** so that any bugs in the user-defined map and reduce functions don’t affect the tasktracker

Progress and StatusUpdates

- MapReduce jobs are long-running batch jobs, taking anything from minutes to hours to run. Because this is a significant length of time, it's important for the user to get feedback on how the job is progressing. A job and each of its tasks have a status, which includes such things as the state of the job or task (e.g., running, successfully completed, failed), the progress of maps and reduces, the values of the job's counters, and a status message or description.

Job Completion

- When the jobtracker receives a notification that the last task for a job is complete (this will be the special job cleanup task), it changes the status for the job to "successful." Then, when the Job polls for status, it learns that the job has completed successfully, so it prints a message to tell the user and then returns from the waitForCompletion() method. Last, the jobtracker cleans up its working state for the job and instructs tasktrackers to do the same

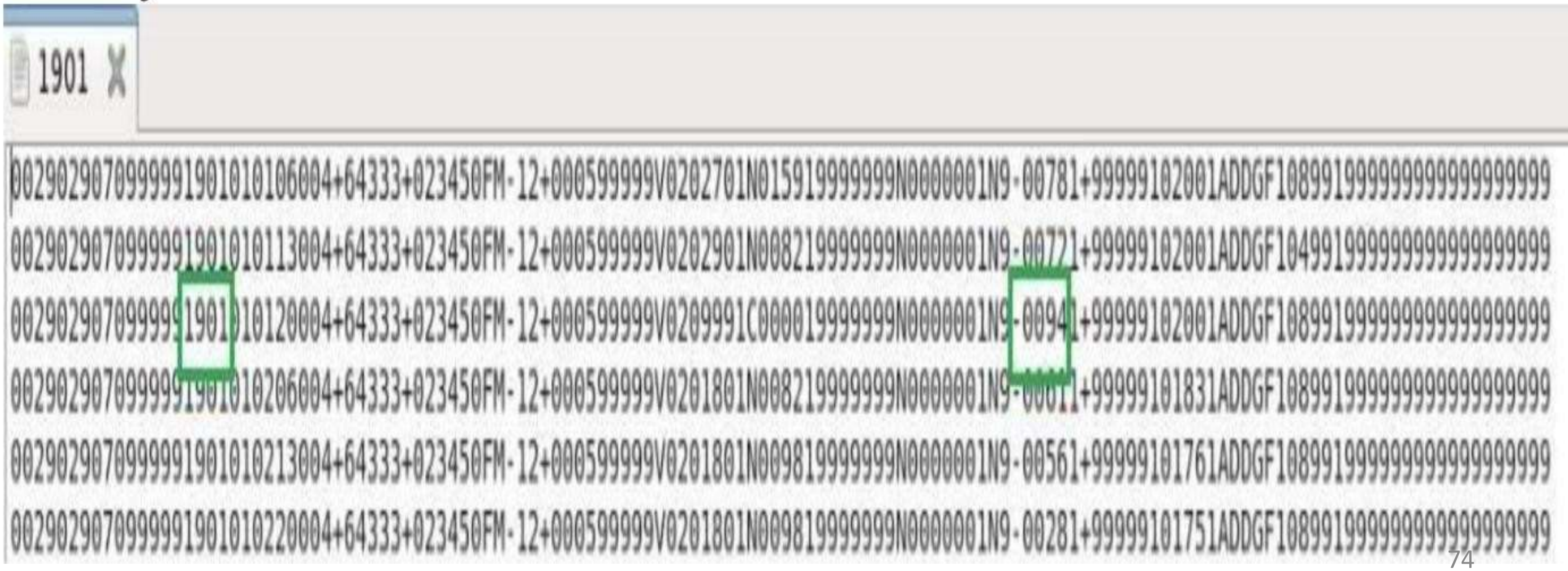
MapReduce Program on Weather Dataset

- Big data is a framework for storage and processing of data (structured/unstructured). Please check out the program below which draw out results out of semi-structured data from a weather sensor. It’s a MapReduce program written in java.
- The aim of the program is to find the average temperature in each year of NCDC data.
- This program takes a data input of multiple files where each file contains weather data of a particular year. This weather data is shared by NCDC (National Climatic Data Center) and is collected by weather sensors at many locations across the globe. NCDC input data can be downloaded from:

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

MapReduce Program on Weather Dataset-(contd.)

- There is a data file for each year. Each data file contains among other things, the year and the temperature information (which is relevant for this program).
- Below is the snapshot of the data with year and temperature field highlighted in green box. This is the snapshot of data taken from year 1901 file



1901

```
0029029070999991901010106004+64333+023450FM-12+000599999V0202701N015919999999N0000001N9-00781+99999102001ADDGF1089919999999999999999
0029029070999991901010113004+64333+023450FM-12+000599999V0202901N008219999999N0000001N9-00721+99999102001ADDGF1049919999999999999999
0029029070999991901010120004+64333+023450FM-12+000599999V0209991C000019999999N0000001N9-00941+99999102001ADDGF1089919999999999999999
0029029070999991901010206004+64333+023450FM-12+000599999V0201801N008219999999N0000001N9-00611+99999101831ADDGF1089919999999999999999
0029029070999991901010213004+64333+023450FM-12+000599999V0201801N009819999999N0000001N9-00561+99999101761ADDGF1089919999999999999999
0029029070999991901010220004+64333+023450FM-12+000599999V0201801N009819999999N0000001N9-00281+99999101751ADDGF1089919999999999999999
```

MapReduce Program on Weather Dataset- (contd.)

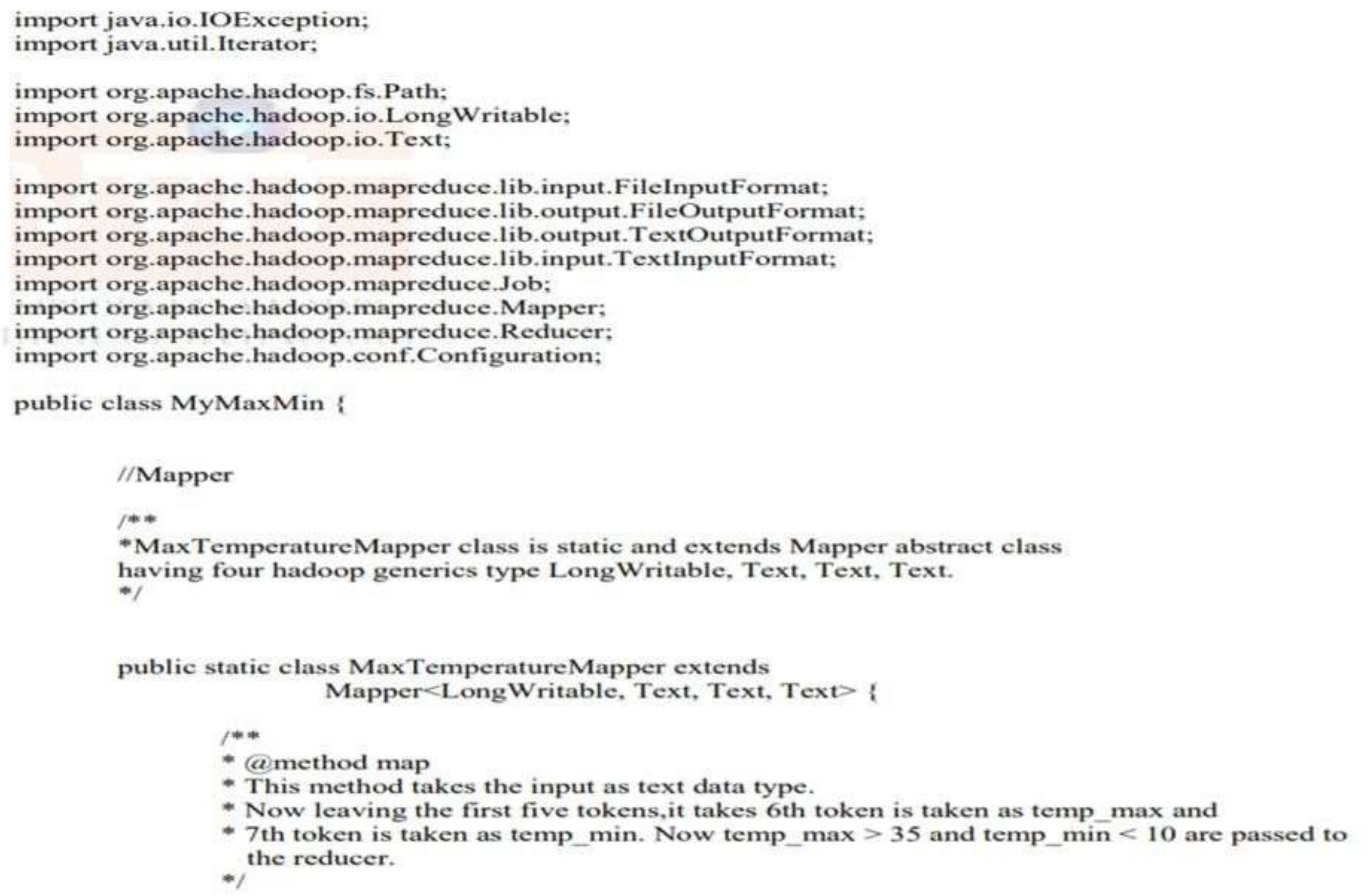
- So, in a MapReduce program there are 2 most important phases.

  Map Phase and Reduce Phase.
- You need to have an understanding of MapReduce concepts so as to understand the intricacies of MapReduce programming. It is one the major component of Hadoop along with HDFS.
- For writing any MapReduce program, **firstly,** you need to figure out the data flow, like in this example am taking just the year and temperature information in the map phase and passing it on to the reduce phase. So Map phase in my example is essentially a data preparation phase. Reduce phase on the other hand is more of a data aggregation one

MapReduce Program on Weather Dataset-(contd.)

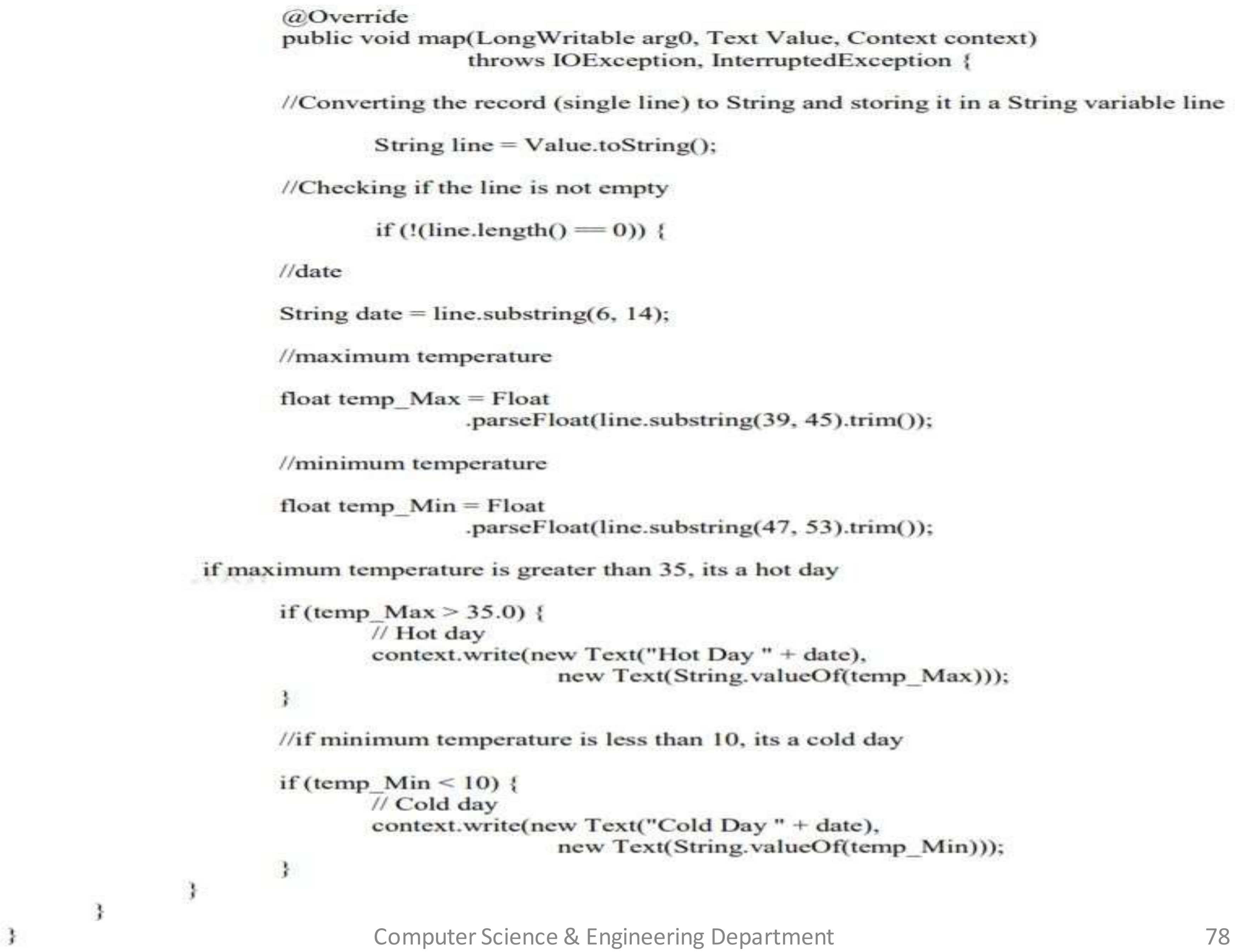
- Secondly, decide on the types for the key/value pairs—MapReduce program uses lists and (key/value) pairs as its main data primitives. So you need to decide the types for key/value pairs—K1, V1, K2, V2, K3, and V3 for the input, intermediate, and output key/value pairs.
- **In this example**, am taking LongWritable and Text as (K1,V1) for input and Text and IntWritable as both for (K2,V2) and (K3,V3)
- **Map Phase**: I will be pulling out the year and temperature data from the log data that is there in the file, as shown in the above snapshot.
- **Reduce Phase**: The data that is generated by the mapper(s) is fed to the reducer, which is another java program. This program takes all the values associated with a particular key and find the average temperature for that key. So, a key in our case is the year and value is a set of IntWritable objects which represent all the captured temperature information for that year

MapReduce Program on Weather Dataset



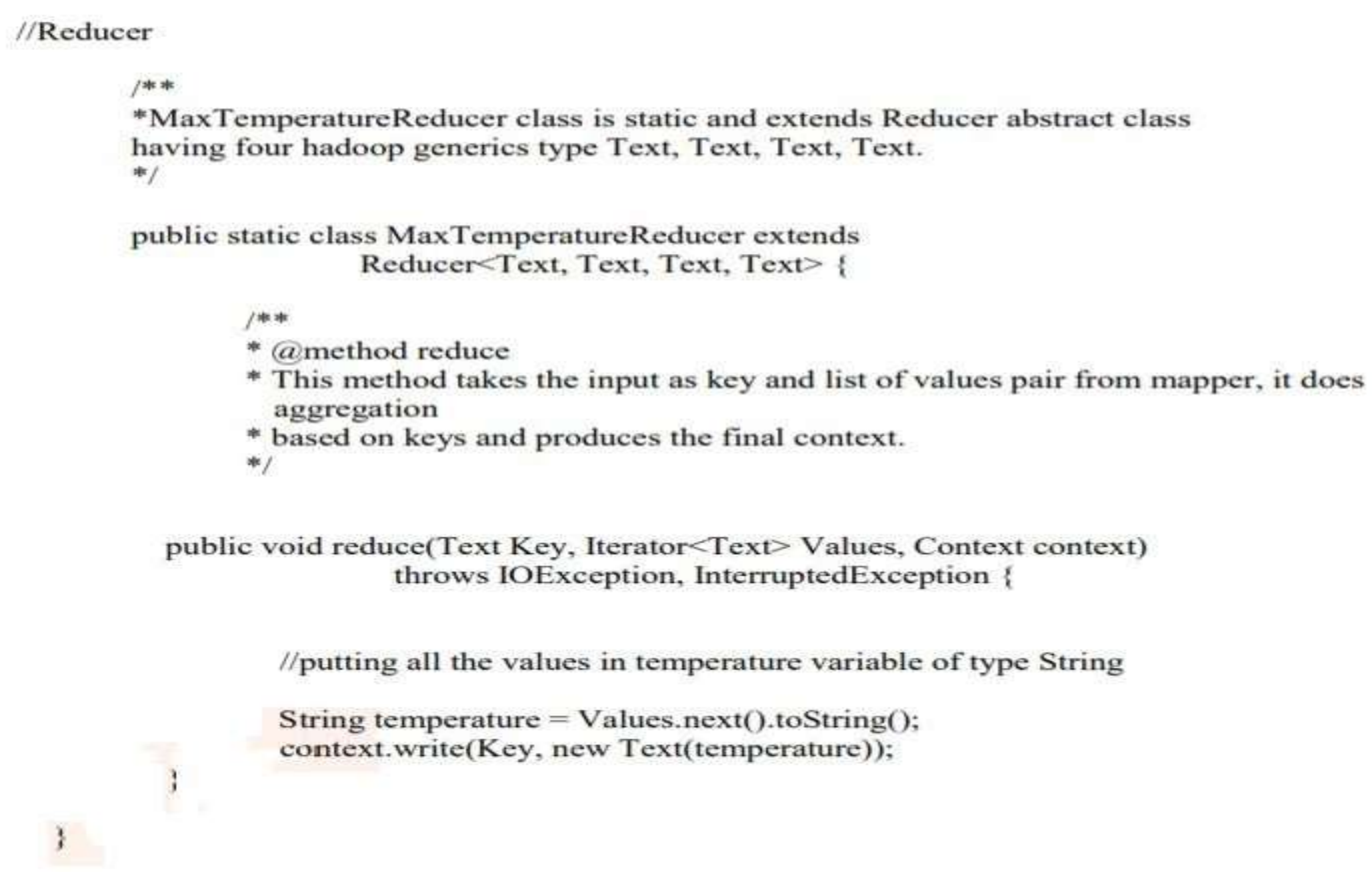
```
import java.io.IOException;
import java.util.Iterator;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.conf.Configuration;

public class MyMaxMin {


        //Mapper

        /**
        *MaxTemperatureMapper class is static and extends Mapper abstract class
        having four hadoop generics type LongWritable, Text, Text, Text.
        */


        public static class MaxTemperatureMapper extends
                        Mapper<LongWritable, Text, Text, Text> {

                /**
                * @method map
                * This method takes the input as text data type.
                * Now leaving the first five tokens,it takes 6th token is taken as temp_max and
                * 7th token is taken as temp_min. Now temp_max > 35 and temp_min < 10 are passed to
                  the reducer.
                */
```

Mapper Code



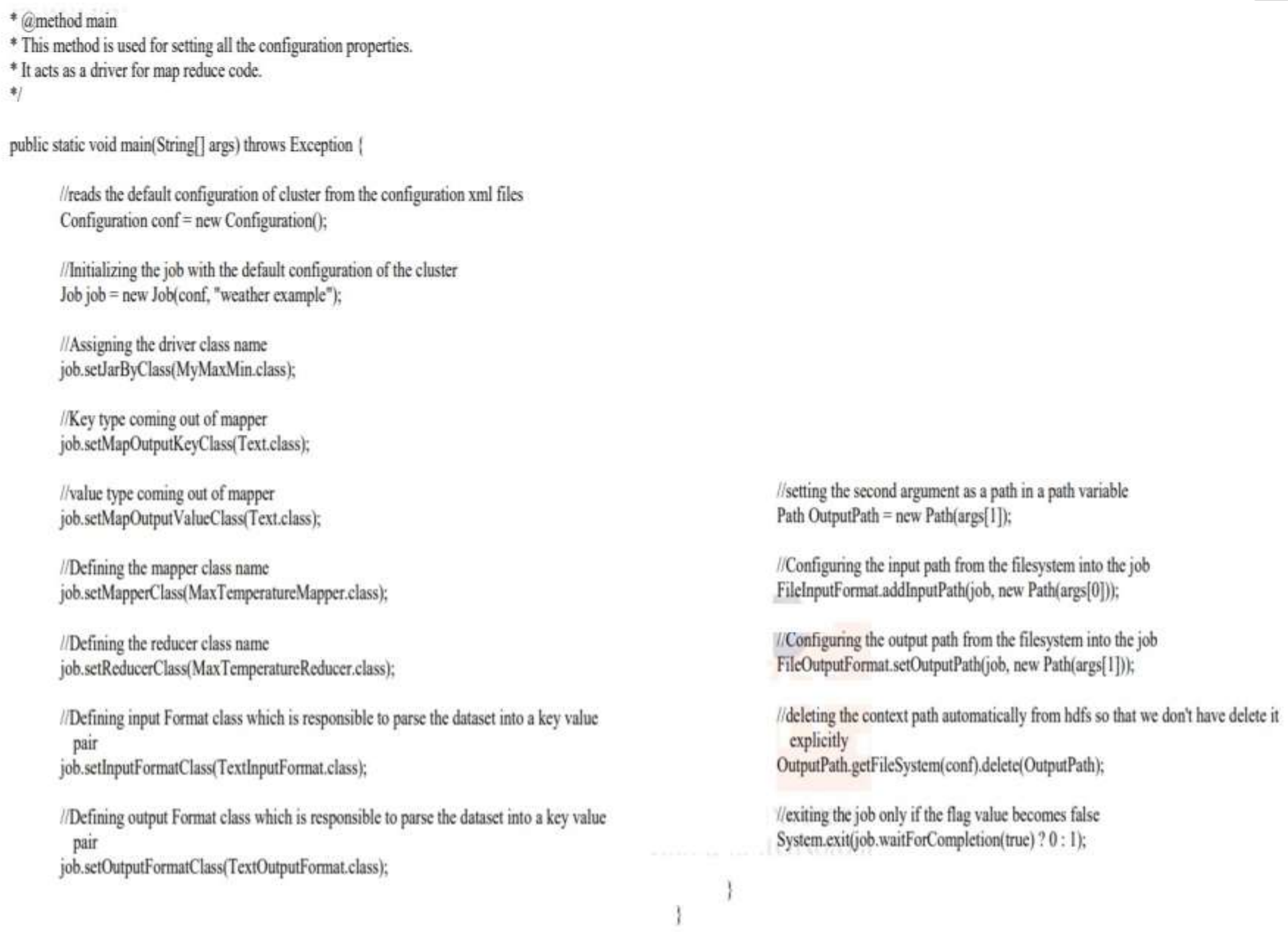
```
@Override
public void map(LongWritable arg0, Text Value, Context context)
                    throws IOException, InterruptedException {

//Converting the record (single line) to String and storing it in a String variable line

          String line = Value.toString();

//Checking if the line is not empty

          if (!(line.length() == 0)) {

//date

String date = line.substring(6, 14);

//maximum temperature

float temp_Max = Float
                    .parseFloat(line.substring(39, 45).trim());

//minimum temperature

float temp_Min = Float
                    .parseFloat(line.substring(47, 53).trim());

if maximum temperature is greater than 35, its a hot day

if (temp_Max > 35.0) {
          // Hot day
          context.write(new Text("Hot Day " + date),
                              new Text(String.valueOf(temp_Max)));
}

//if minimum temperature is less than 10, its a cold day

if (temp_Min < 10) {
          // Cold day
          context.write(new Text("Cold Day " + date),
                              new Text(String.valueOf(temp_Min)));
}
        }
    }
}
```

ReducerCode



```
//Reducer

        /**
        *MaxTemperatureReducer class is static and extends Reducer abstract class
        having four hadoop generics type Text, Text, Text, Text.
        */

        public static class MaxTemperatureReducer extends
                        Reducer<Text, Text, Text, Text> {

            /**
            * @method reduce
            * This method takes the input as key and list of values pair from mapper, it does
              aggregation
            * based on keys and produces the final context.
            */


          public void reduce(Text Key, Iterator<Text> Values, Context context)
                        throws IOException, InterruptedException {


              //putting all the values in temperature variable of type String

              String temperature = Values.next().toString();
              context.write(Key, new Text(temperature));
          }

    }
```

Driver Code



```
* @method main
* This method is used for setting all the configuration properties.
* It acts as a driver for map reduce code.
*/

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

        //reads the default configuration of cluster from the configuration xml files
        Configuration conf = new Configuration();

        //Initializing the job with the default configuration of the cluster
        Job job = new Job(conf, "weather example");

        //Assigning the driver class name
        job.setJarByClass(MyMaxMin.class);

        //Key type coming out of mapper
        job.setMapOutputKeyClass(Text.class);

        //value type coming out of mapper
        job.setMapOutputValueClass(Text.class);

        //Defining the mapper class name
        job.setMapperClass(MaxTemperatureMapper.class);

        //Defining the reducer class name
        job.setReducerClass(MaxTemperatureReducer.class);

        //Defining input Format class which is responsible to parse the dataset into a key value
          pair
        job.setInputFormatClass(TextInputFormat.class);

        //Defining output Format class which is responsible to parse the dataset into a key value
          pair
        job.setOutputFormatClass(TextOutputFormat.class);

        //setting the second argument as a path in a path variable
        Path OutputPath = new Path(args[1]);

        //Configuring the input path from the filesystem into the job
        FileInputFormat.addInputPath(job, new Path(args[0]));

        //Configuring the output path from the filesystem into the job
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        //deleting the context path automatically from hdfs so that we don't have delete it
          explicitly
        OutputPath.getFileSystem(conf).delete(OutputPath);

        //exiting the job only if the flag value becomes false
        System.exit(job.waitForCompletion(true) ? 0 : 1);

    }
}
```