

# VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY (Autonomous)

Department of Computer Science and  
Engineering



IV B.Tech – I Semester (Sections – C & D)

**Big Data Analytics** (Elective – I)

**UNIT – III : Writing MapReduce Programs**

# Syllabus

## UNIT-III

### Writing MapReduce Programs:

A Weather Dataset, Understanding Hadoop API for MapReduce Framework (Old and New),

Basic programs of Hadoop MapReduce: Driver code, Mapper code, Reducer code, RecordReader, Combiner, Partitioner

# Outline of Unit-III

- Introduction to MapReduce
- Understanding Hadoop API for MapReduce Framework (Old and New)
- Basic programs of Hadoop MapReduce
- Word Count
- A Weather Dataset
- Driver code
- Mapper code
- Reducer code
- RecordReader
- Combiner
- Partitioner
- Experiments on MapReduce Programs
- Demo on Practice of MR Programs

# 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.

# HADOOP -- MAPREDUCE

## 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.

## HADOOP -- MAPREDUCE

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.

## HADOOP -- 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).

## HADOOP -- MAPREDUCE

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.



## Architecture to show about MapReduce Jobs

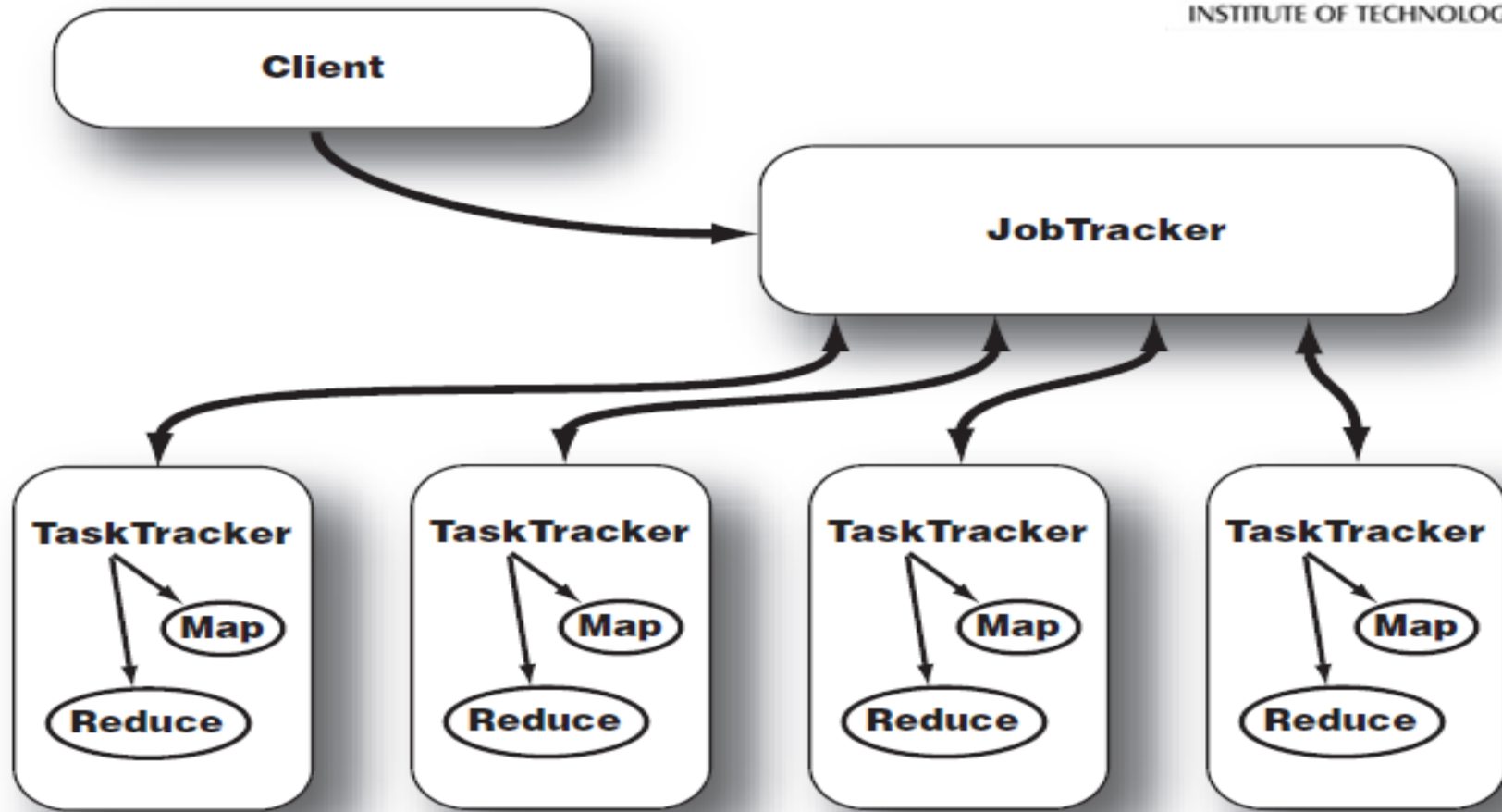


Fig. JobTracker and TaskTracker interaction. After a client calls the JobTracker to begin a data processing job, the JobTracker partitions the work and assigns different map and reduce tasks to each TaskTracker in the cluster.

## HADOOP -- MAPREDUCE

### **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.

## HADOOP -- MAPREDUCE

### ◦ **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.

## HADOOP -- MAPREDUCE

### 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.

## HADOOP -- MAPREDUCE

During a MapReduce job, Hadoop sends the Map and Reduce tasks to the appropriate servers in the cluster.

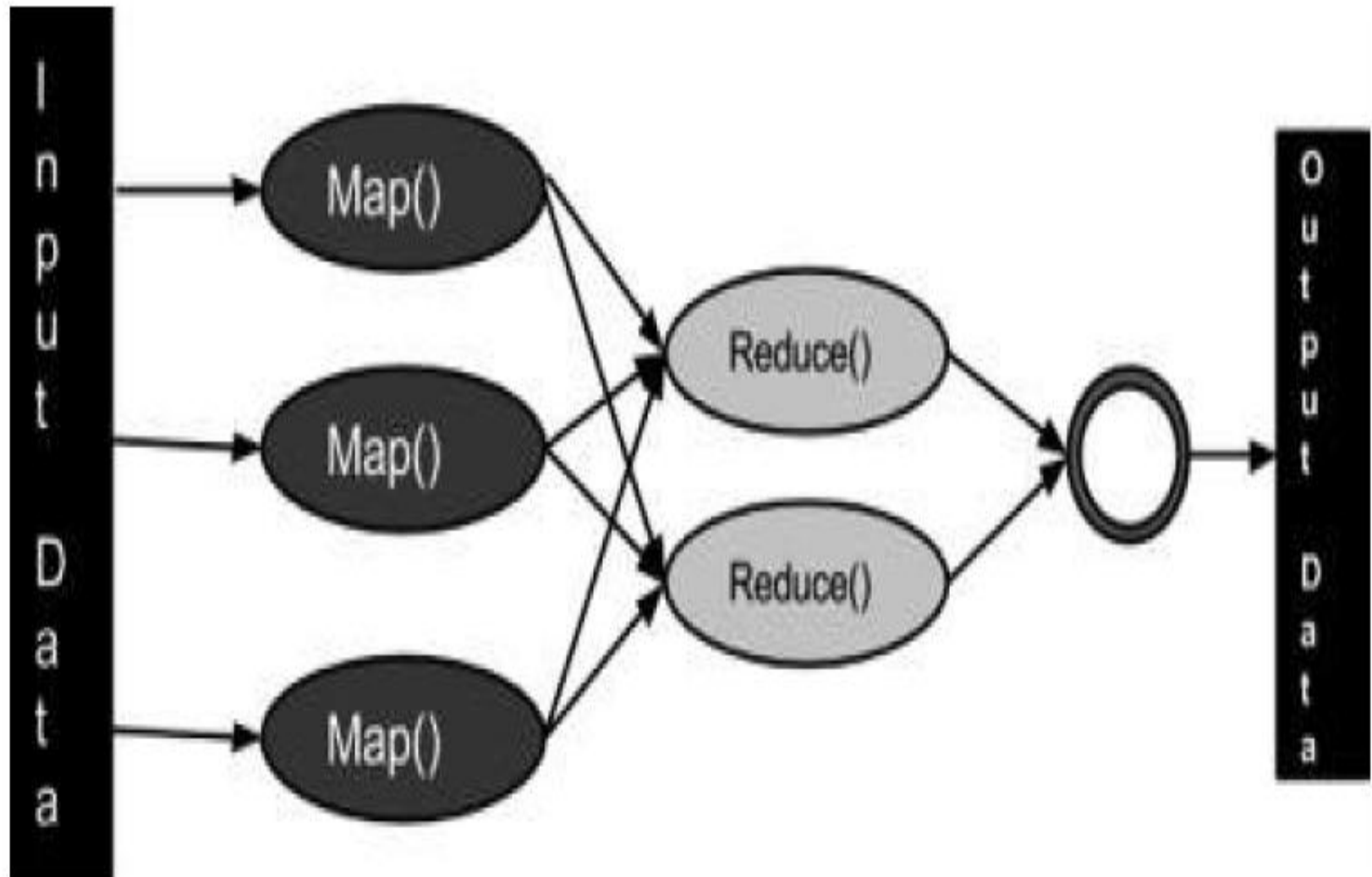
The framework manages all the details of data-passing such as issuing tasks, verifying task completion, and copying data around the cluster between the nodes.

## HADOOP -- MAPREDUCE

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.

# HADOOP -- MAPREDUCE



# HADOOP -- MAPREDUCE

## Inputs and Outputs (*Java Perspective*)

The MapReduce framework operates on  $\langle \text{key}, \text{value} \rangle$  pairs, that is, the framework views the input to the job as a set of  $\langle \text{key}, \text{value} \rangle$  pairs and produces a set of  $\langle \text{key}, \text{value} \rangle$  pairs as the output of the job, conceivably of different types.



## HADOOP -- MAPREDUCE

• 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.**

# HADOOP -- MAPREDUCE

	Input	Output
Map	$\langle k1, v1 \rangle$	list $\langle k2, v2 \rangle$
Reduce	$\langle k2, listv2 \rangle$	list $\langle k3, v3 \rangle$

## HADOOP -- MAPREDUCE

### 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.

## HADOOP -- MAPREDUCE

### Terminology – Contd.

**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.

# HADOOP -- MAPREDUCE

## Terminology – Contd.

**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.

## 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
<b>Mapper &amp; Reducer</b>	New API using Mapper and Reducer as <b>Class</b> 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 <b>Interface</b> . (still exist in New API as well)
<b>Package</b>	New API is in the <b><i>org.apache.hadoop.mapreduce</i></b> package.	Old API can still be found in <b><i>org.apache.hadoop.mapred</i></b> package.
<b>User Code to communicate with MapReduce System</b>	Use “ <b><i>context</i></b> ” object to communicate with MapReduce System.	<b><i>JobConf</i></b> , the <b><i>OutputCollector</i></b> , and the <b><i>Reporter</i></b> objects use for communicate with MapReduce System.

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

Difference	New API	OLD API
<b>Control Mapper and Reducer execution</b>	New API allows both mappers and reducers to control the execution flow by <b>overriding the run()</b> method.	Controlling mappers by writing a <b>MapRunnable</b> , but no equivalent exists for reducers.
<b>Job control</b>	Job control is done through the <b>JOB</b> class in New API.	Job Control was done through <b>JobClient</b> . (not exists in the new API)
<b>Job Configuration</b>	Job Configuration done through <b>Configuration</b> class via some of the helper methods on Job.	<b>jobconf</b> 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
<b>OutPut file Name</b>	In the new API map outputs are named <b>part-m-nnnnn</b> , and reduce outputs are named <b>part-r-nnnnn</b> (where nnnnn is an integer designating the part number, starting from zero).	In the old API both map and reduce outputs are named <b>part-nnnnn</b> .
<b>reduce() method passes values</b>	In the new API, the reduce() method passes values as a <b>java.lang.Iterable</b>	In the Old API, the reduce() method passes values as a <b>java.lang.Iterator</b>

So these are the Main Differences between Old and New MR APIs.

# MapReduce Concepts

- What is MapReduce
- MapReduce programming paradigm
  - ☐ Introduction
  - ☐ The Model
  - ☐ Visualization
- Hadoop Daemons
- Job Tracker and Task Tracker
- Hadoop MapReduce framework
- Apache Hadoop MapReduce API
  - ☐ Hadoop Data types
  - ☐ API
  - ☐ Phases of Reducer
  - ☐ Visualization
- MapReduce Implementation in Hadoop
  - ☐ The big picture behind word count MapReduce
  - ☐ Writing a Driver class
  - ☐ Writing a Mapper class
  - ☐ Writing a Reducer class
- Combiner & Partitioner

# What is MapReduce?

- MapReduce is a framework for performing distributed data processing using the MapReduce programming paradigm.
- In the MapReduce paradigm, each job has a user-defined map phase, which is a parallel, share-nothing processing of input; followed by a user-defined reduce phase where the output of the map phase is aggregated.
- Typically, HDFS is the storage system for both input and output of the MapReduce jobs.

# MapReduce Programming Paradigm - Introduction

- **MapReduce** is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.
- Seen from a high level, Map-Reduce is really two things:
  1. A *strategy* or *model* for writing programs that can easily be made to process data in parallel.
  2. A *framework* that runs these programs in parallel, automatically handling the details of division of labor, distribution, synchronization, and fault- tolerance.
- In the map-reduce programming model, work is divided into two phases: a *map* phase and a *reduce* phase. Both of these phases work on key-value pairs.

# Typical problem solved by MapReduce

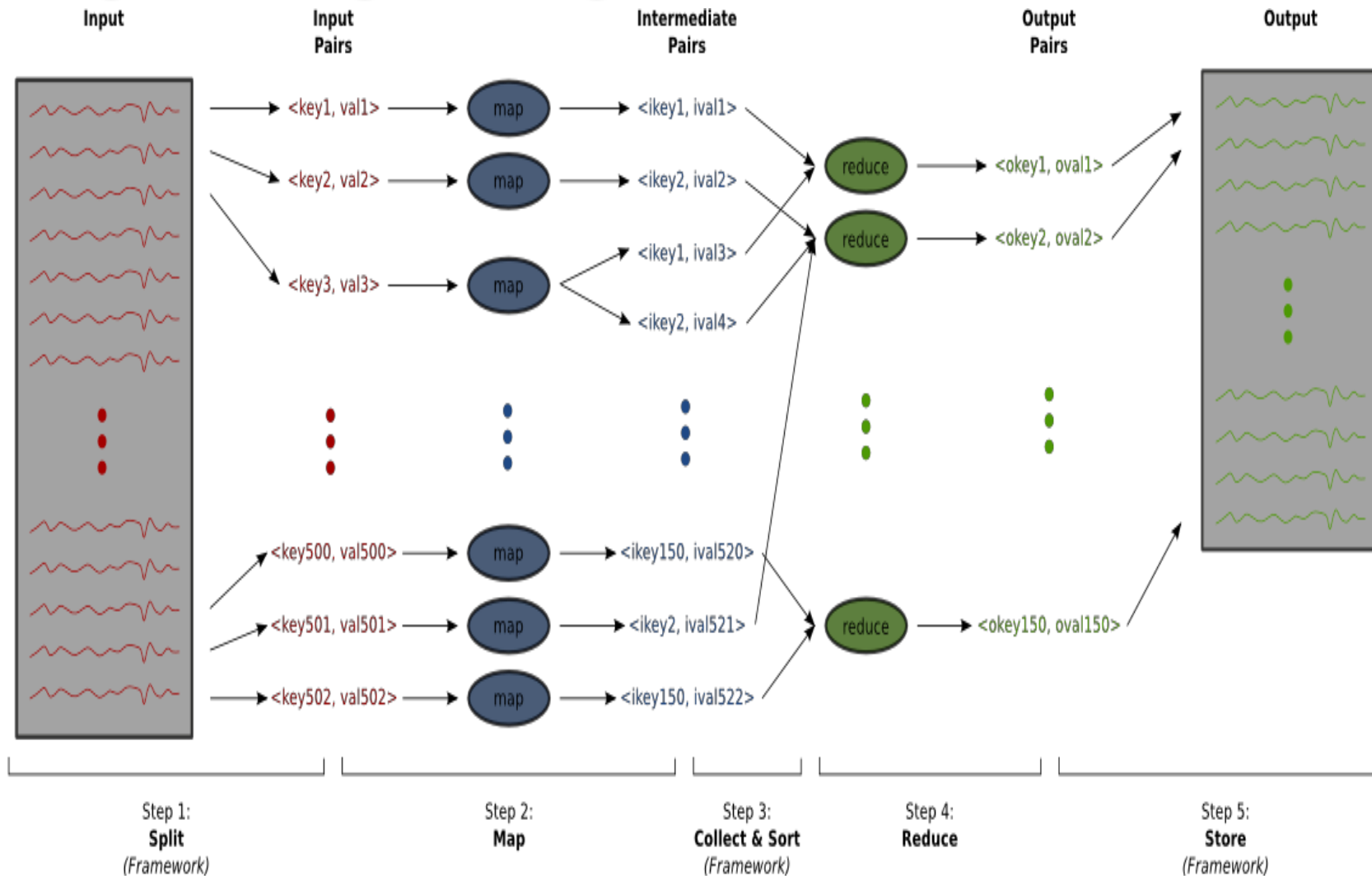
- Read a lot of data
- **Map:** extract something you care about for each record and analyze
- Shuffle and Sort
- **Reduce:** Aggregate, summarize, filter or transform
- Write the results

Framework stays the same Map and Reduce  
change to fit the problem

# More specifically...

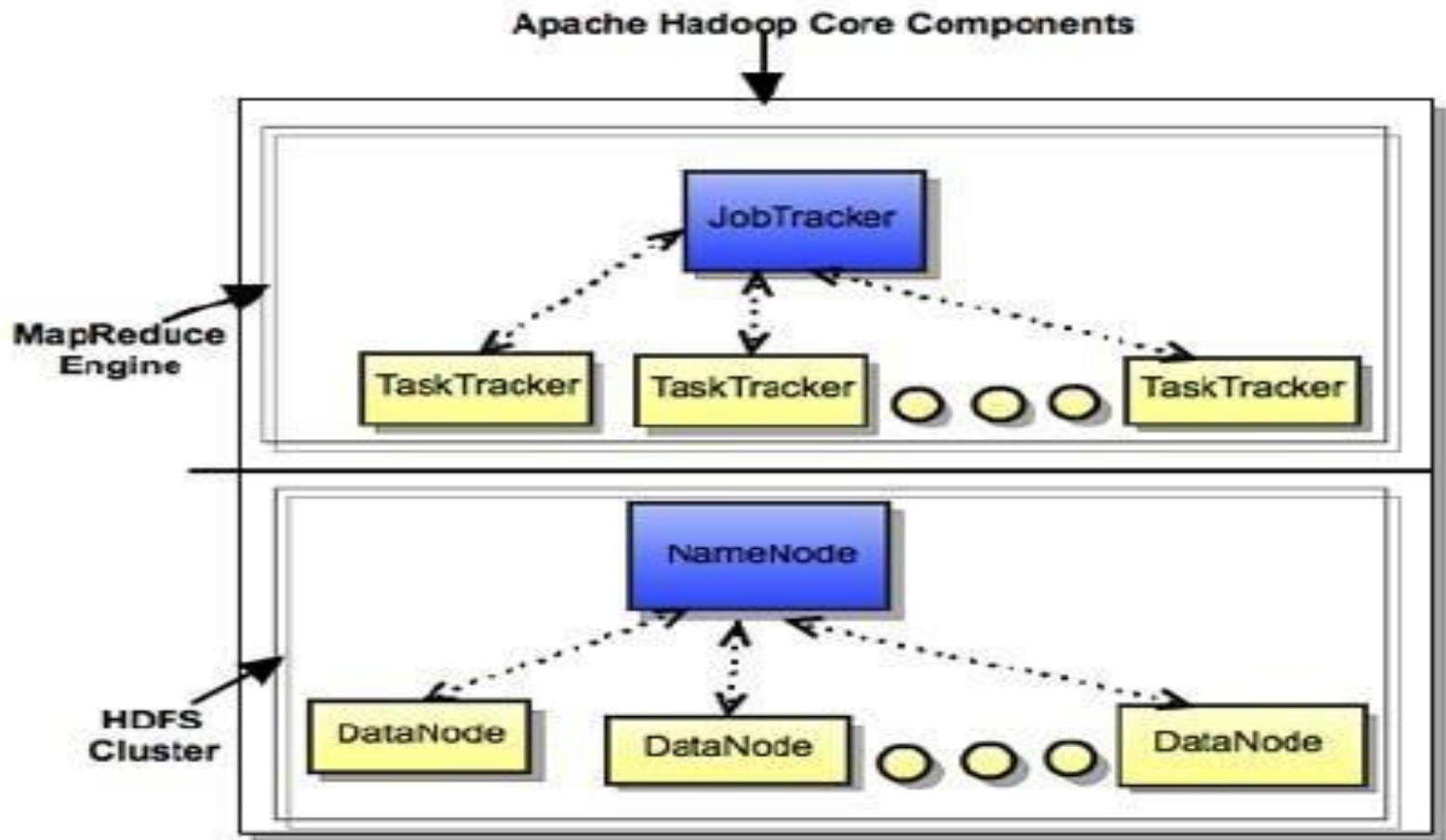
- Programmer specifies two primary methods:
  - $\text{Map}(k, v) \rightarrow \langle K', V' \rangle^*$
  - $\text{Reduce}(\langle K', |V'| \rangle^*) \rightarrow \langle k', v' \rangle^{**}$
- All  $v'$  with same  $k'$  are reduced together, in order.
- Usually also specify
  - $\text{Partition}(k', \text{total partitions}) \rightarrow \text{partition for } k'$ 
    - Often a simple hash of the key
    - Allows reduce operations for different  $k'$  to be parallelized

# Visualization Of MapReduce Programming Paradigm





# Hadoop Daemons (HDFS and MapReduce Daemons)

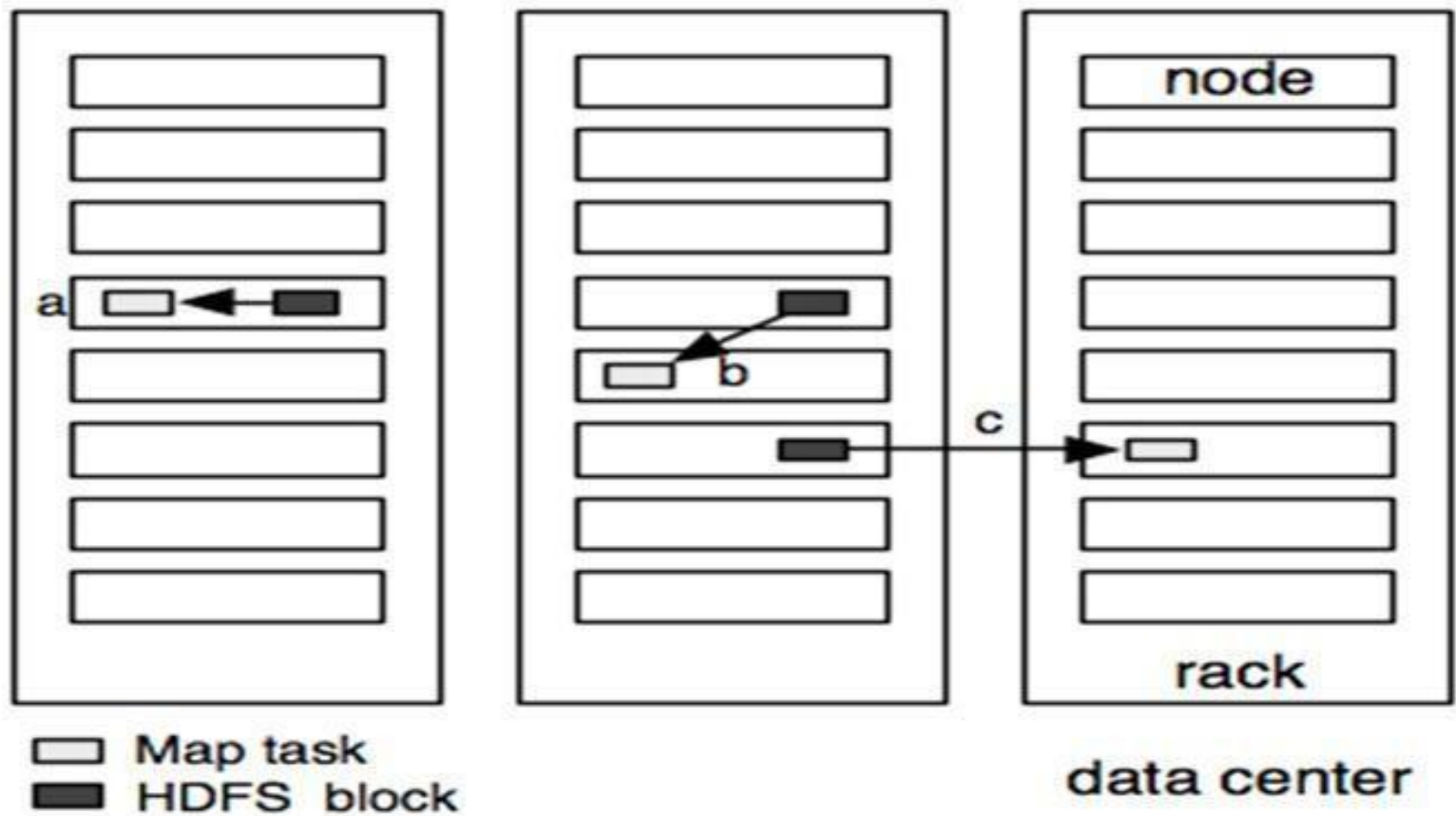




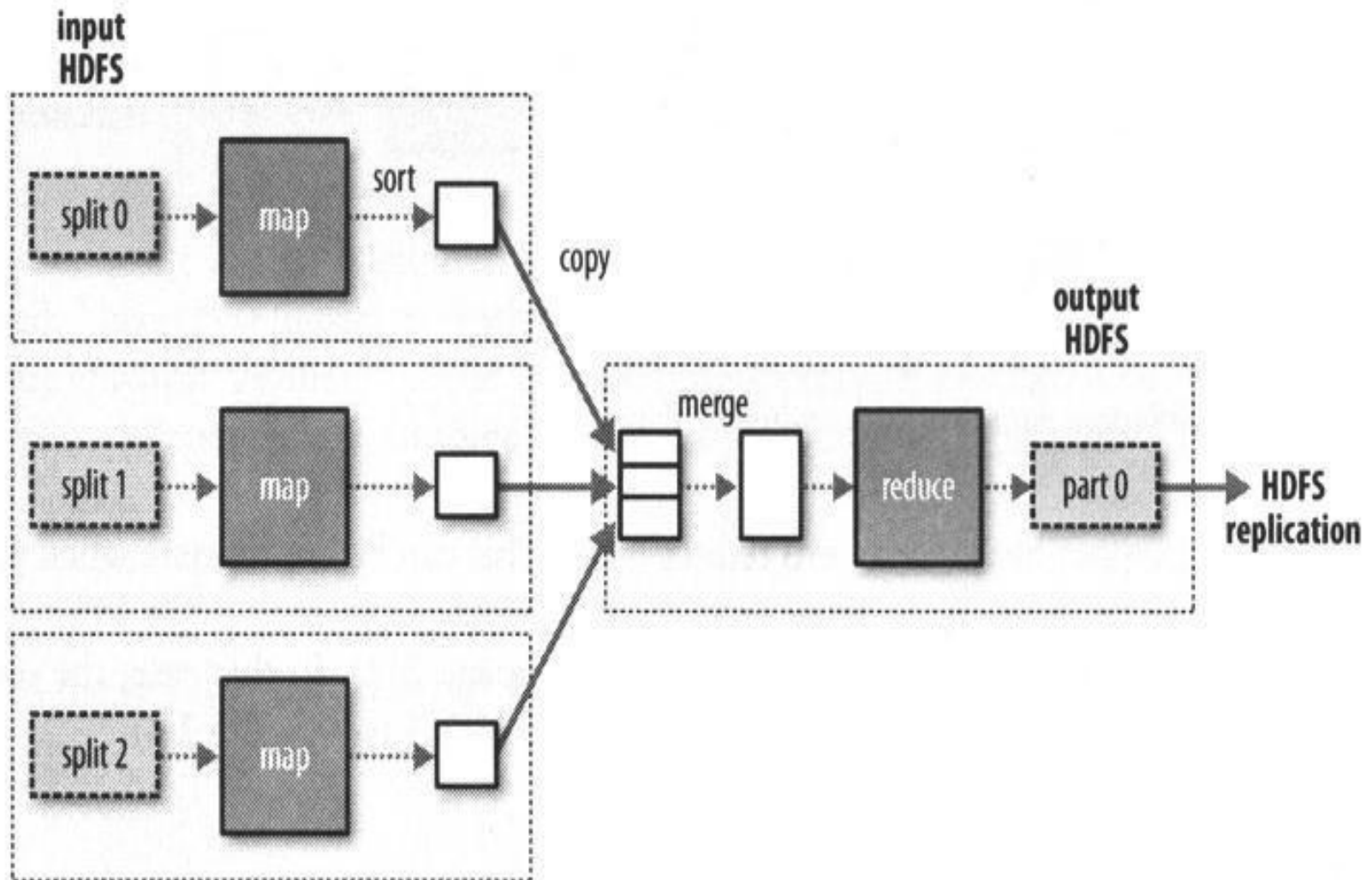
# JobTracker & TaskTrackers

- **JobTracker** is the master of the system which manages the jobs and resources in the cluster (TaskTrackers). The JobTracker tries to schedule each map as close to the actual data being processed i.e. on the TaskTracker which is running on the same DataNode as the underlying block.
- **TaskTrackers** are the slaves which are deployed on each machine. They are responsible for running the map and reduce tasks as instructed by the JobTracker.

# Data Center

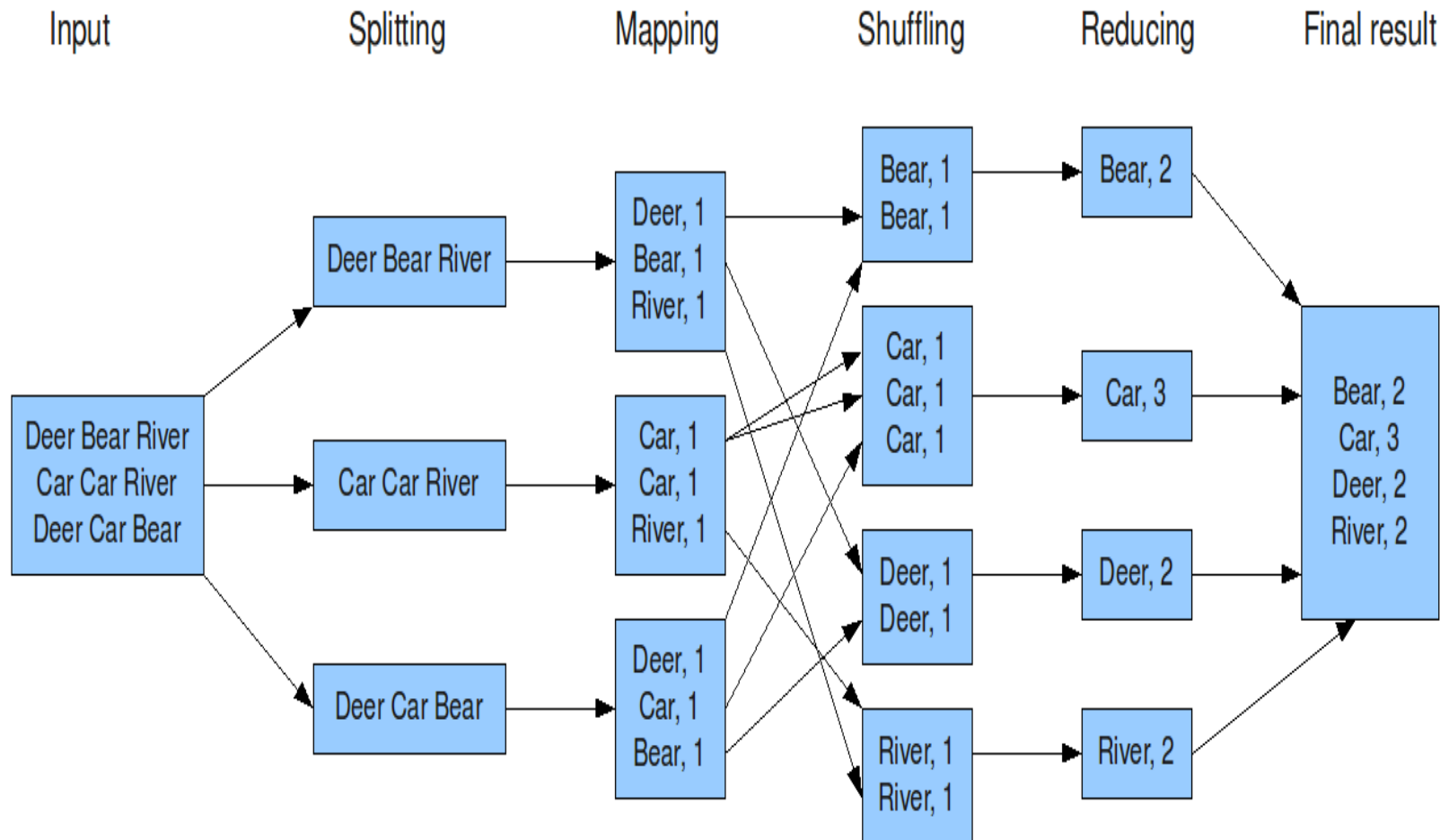


# Hadoop MapReduce Framework



# The Big Picture Behind Word Count Mapreduce

The overall MapReduce word count process



## Basic Programs of Hadoop MapReduce – Contd.

The basic components of a MapReduce program are:

Driver code,  
Mapper code,  
Reducer code,  
RecordReader,  
Combiner,  
Partitioner

## Basic Programs of Hadoop

### MapReduce – Contd.

**MapReduce** is the programming model to work on data within the HDFS.

The programming language for MapReduce is Java.

Hadoop also provides streaming where in other languages could also be used to write MapReduce programs.

All data emitted in the flow of a MapReduce program is in the form of <Key, Value> pairs.

A MapReduce program consists of the following 3 parts :

Driver

Mapper

Reducer

# Basic Programs of Hadoop

## MapReduce – Contd.

### Driver

The Driver code runs on the client machine and is responsible for building the configuration of the job and submitting it to the Hadoop Cluster.

The Driver code will contain the main() method that accepts arguments from the command line.

Some of the common libraries that are included for the Driver class :

```
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.io.*;  
import org.apache.hadoop.mapred.*;
```



## Basic Programs of Hadoop MapReduce – Contd.

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.



## Basic Programs of Hadoop

### MapReduce – Contd.

The next step the Driver program should do is to configure the Job that needs to be submitted to the cluster.

To do this we create an object of type **JobConf** and pass the name of the Driver class.

The JobConf class allows you to configure the different properties for the Mapper, Combiner, Partitioner, Reducer, InputFormat and OutputFormat.

## Basic Programs of Hadoop MapReduce – Contd.

### 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);  
    }  
}
```

## Basic Programs of Hadoop MapReduce – Contd.

// 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);
```

```
}
```

```
}
```

# Basic Programs of Hadoop

## MapReduce – Contd.

### Mapper

The Mapper code reads the input files as <Key,Value> pairs and emits <Key,Value> pairs.

The Mapper class extends MapReduceBase and implements the Mapper interface.

The Mapper interface expects four generics, which define the types of the input and output key/value pairs.

The first two parameters define the input key and value types, the second two define the output key and value types.

Some of the common libraries that are included for the Mapper class :

```
import java.io.IOException;  
import java.util.*;  
import org.apache.hadoop.io.*;  
import org.apache.hadoop.mapred.*;
```

## Basic Programs of Hadoop MapReduce – Contd.

### 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 responsible for writing the intermediate data generated by the Mapper.

# Basic Programs of Hadoop

## MapReduce – Contd.

### Reducer

The Reducer code reads the outputs generated by the different mappers as  $\langle \text{Key}, \text{Value} \rangle$  pairs and emits  $\langle \text{Key}, \text{Value} \rangle$  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`.

## Basic Programs of Hadoop MapReduce – Contd.

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.*;
```

## Basic Programs of Hadoop MapReduce – Contd.

### Sample

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

The reduce() function accepts the key, an iterator, OutputCollector and an Reporter object.

The OutputCollector is responsible for writing the final output result.



# Basic Programs of Hadoop

## MapReduce – Contd.

### 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.

## Basic Programs of Hadoop MapReduce – Contd.

### Combiner – Contd.

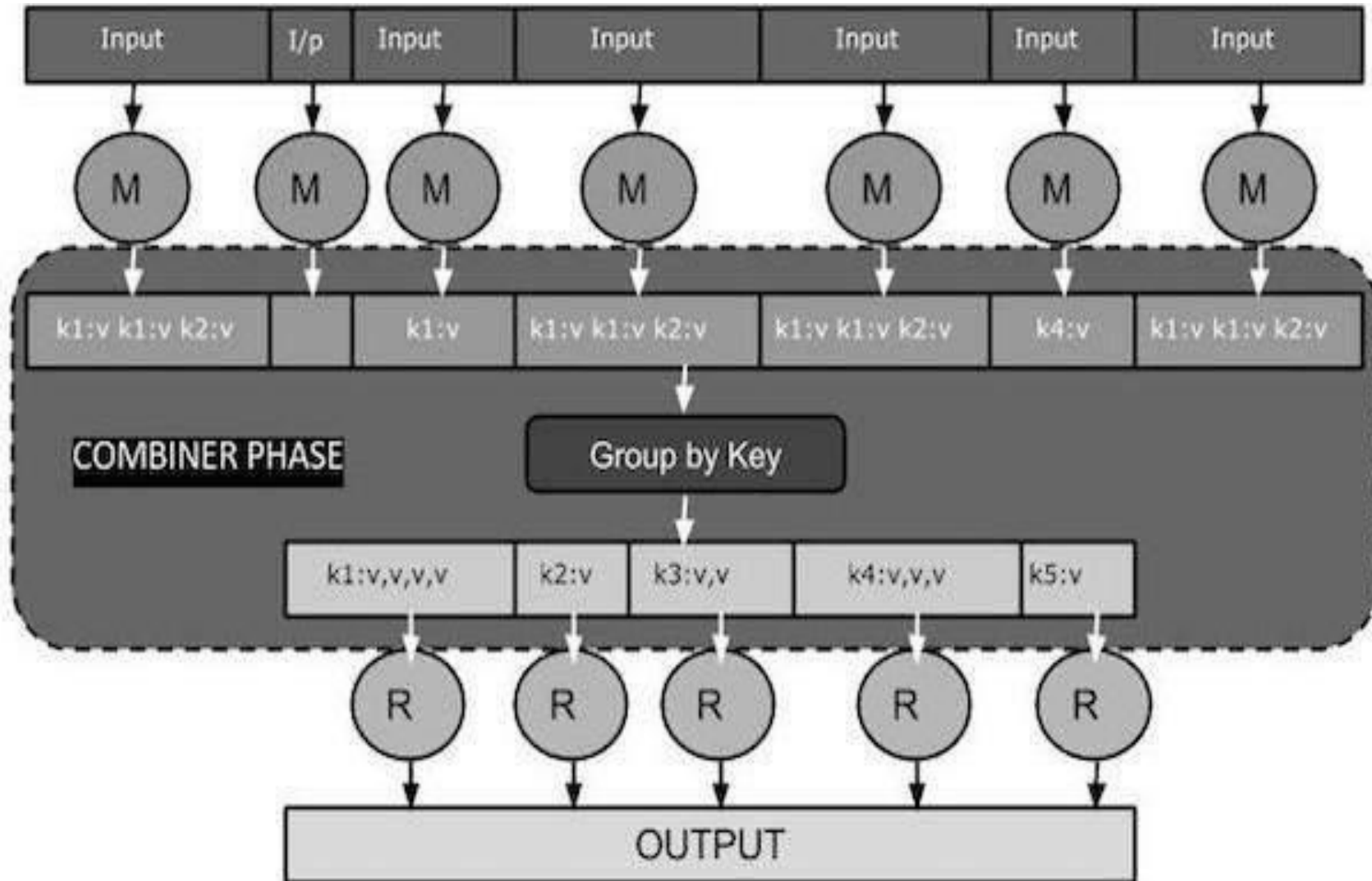
The Combiner class is used in between the Map class and the Reduce class to reduce the volume of data transfer between Map and Reduce.

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.

# Basic Programs of Hadoop

## MapReduce – Contd.



## Basic Programs of Hadoop MapReduce – Contd.

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.

## Basic Programs of Hadoop MapReduce – Contd.

### 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.

# Basic Programs of Hadoop

## MapReduce – Contd.

### MapReduce Combiner Implementation

We will discuss about Mapper, Reducer, Combiners and other phases which are required for MR Job Processing through this example.

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.

# Basic Programs of Hadoop

## MapReduce – Contd.

### 1) 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>

## Basic Programs of Hadoop MapReduce – Contd.

### 2) 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>



## Basic Programs of Hadoop MapReduce – Contd.

The Map phase reads each key-value pair, divides each word from the value using StringTokenizer, treats each word as key and the count of that word as value.

The following code snippet shows the Mapper class and the map function.

## Basic Programs of Hadoop

### MapReduce – Contd.

```
public static class TokenizerMapper extends Mapper<Object, Text,  
Text, IntWritable>  
{  
    private final static IntWritable one = new IntWritable(1);  
    private Text word = new Text();  
  
    public void map(Object key, Text value, Context  
        context) throws IOException, InterruptedException  
    {  
        StringTokenizer itr = new StringTokenizer(value.toString());  
        while (itr.hasMoreTokens())  
        {  
            word.set(itr.nextToken());  
            context.write(word, one);  
        }  
    }  
}
```

## Basic Programs of Hadoop MapReduce – Contd.

**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>

## Basic Programs of Hadoop MapReduce – Contd.

### 3) 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>

## Basic Programs of Hadoop MapReduce – Contd.

The Combiner phase reads each key-value pair, combines the common words as key and values as collection.

Usually, the code and operation for a Combiner is similar to that of a Reducer.

Following is the code snippet for Mapper, Combiner and Reducer class declaration.

```
job.setMapperClass(TokenizerMapper.class);  
job.setCombinerClass(IntSumReducer.class);  
job.setReducerClass(IntSumReducer.class);
```

## Basic Programs of Hadoop MapReduce – Contd.

**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>

## Basic Programs of Hadoop MapReduce – Contd.

### 4) 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>

## Basic Programs of Hadoop

### MapReduce – Contd.

The Reducer phase reads each key-value pair. Following is the code snippet for the Combiner.

```
public static class IntSumReducer extends
    Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable>
        values,Context context) throws IOException,
        InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
    sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
    }
}
```



## Basic Programs of Hadoop MapReduce – Contd.

**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>

## Basic Programs of Hadoop MapReduce – Contd.

### 5) 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.

## Basic Programs of Hadoop

### MapReduce – Contd.

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

In this example, we haven't considered the sorting phase.

So, the outputs are generated with respect to the lines.

# What is writable?

- Hadoop defines its own “box” classes for strings
  - (*Text*), integers (*IntWritable*), etc.

All values are instances of *Writable*

All keys are instances of *WritableComparable*

- 

*WritableComparator*

- Compares *WritableComparable* data

- Will call *WritableComparable.compare()*

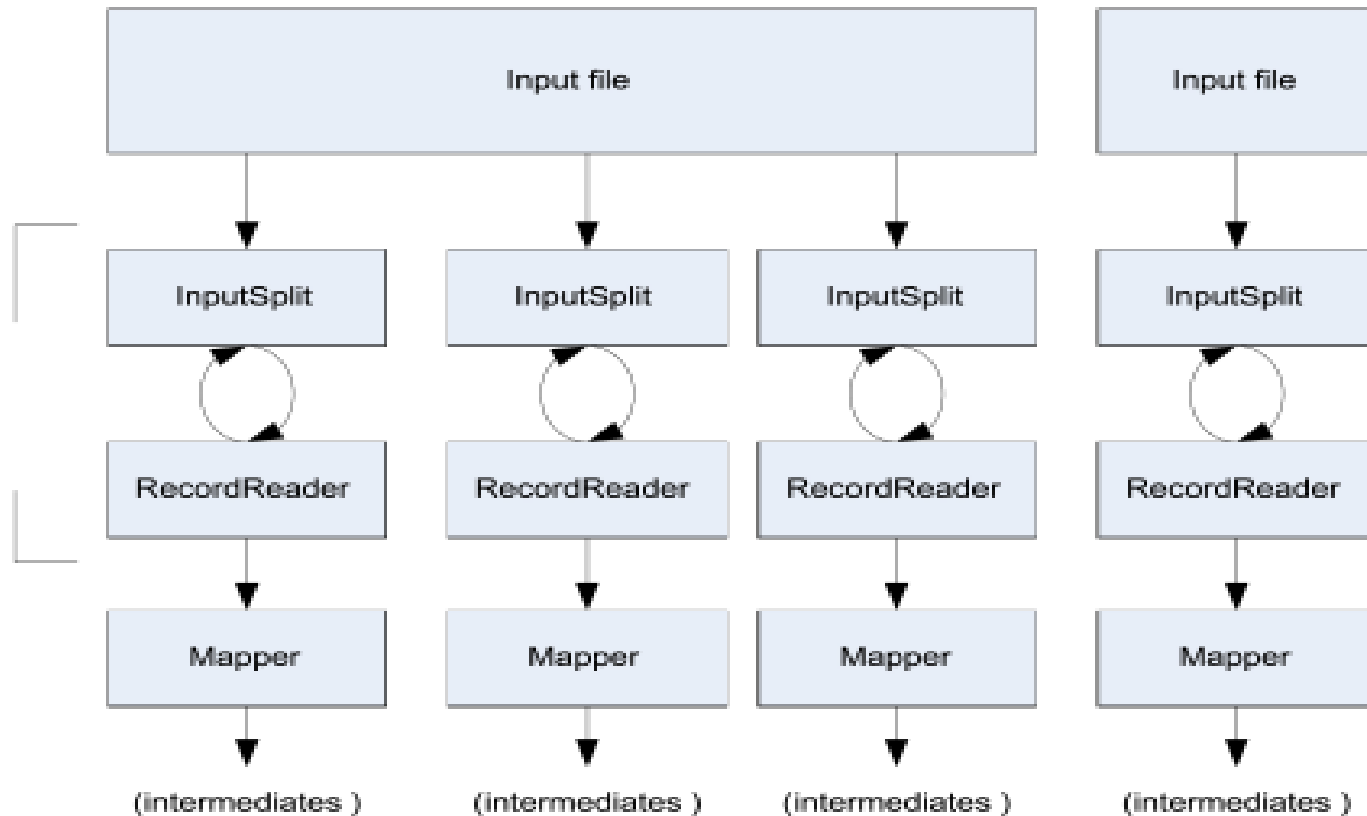
Can provide fast path for serialized data

*JobConf.setOutputValueGroupingComparator()*

# Why Writable Interface?

- To achieve serialization format for MapReduce programs
- *Writables* don't store their type in serialized representation, Since at the point of deserialization it is known which type is expected, unlike *Java Serialization*
- *WritableComparable* is also an interface extends both writable interface and `java.lang.Comparable`

# Getting Data To The Mapper



# Reading Data

- 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

# Input Format : Responsibilities

Divide input data into logical input splits

- Data in HDFS is divided into block, but processed as input splits
- *InputSplit* may contains any number of blocks (usually 1)
- Each Mapper processes one input split

Creates *RecordReaders* to extract <key, value> pairs



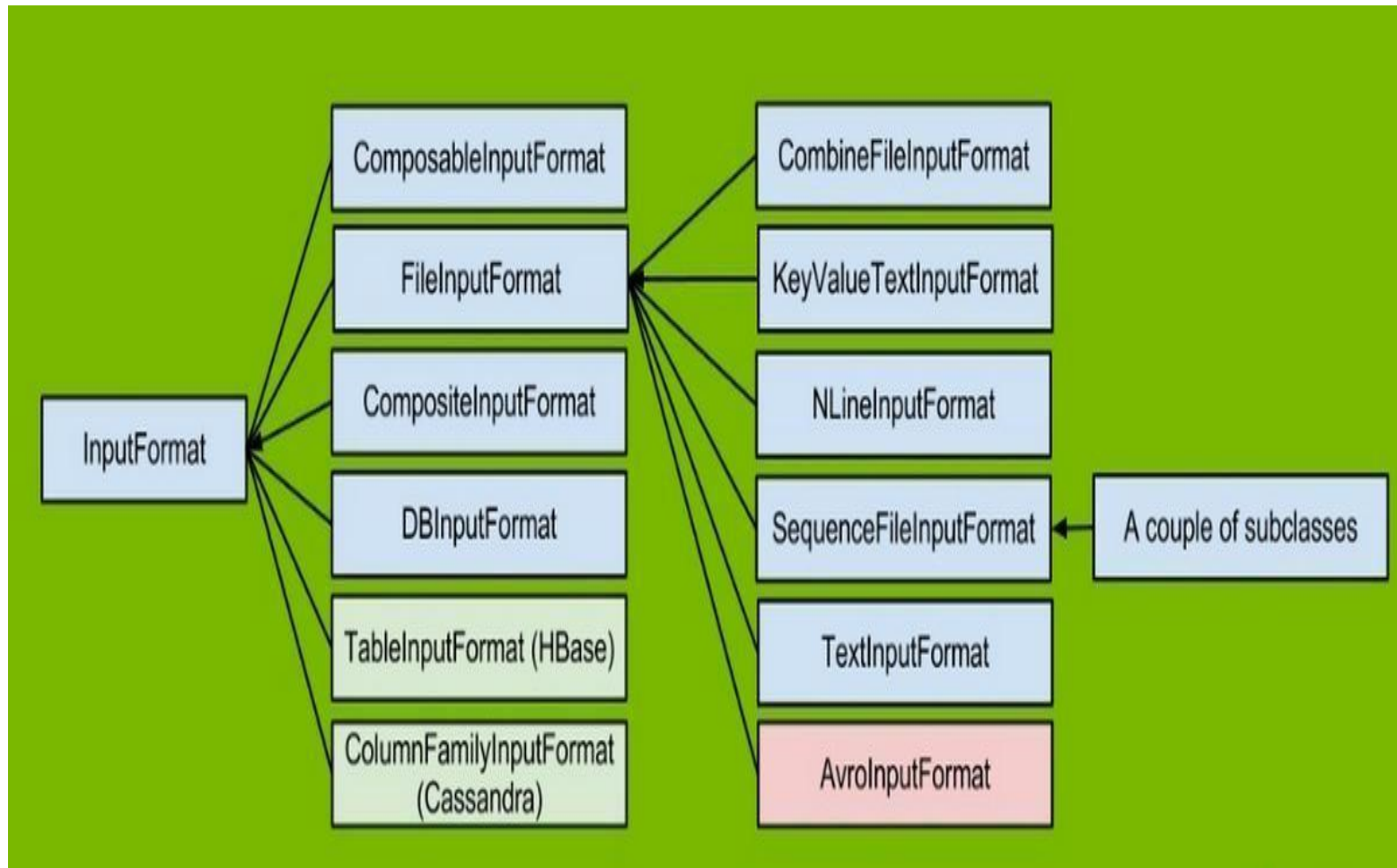
# Input Formats

- Most Hadoop programs read their input from a file.
- The data from a Hadoop input file must be parsed as map (key, value) pairs, *even before the **map()** function.*
- The key and value types from the input file are separate from the key and value types using for **map** and **reduce**.
- A very simple input format: **TextInputFormat**
  - Key:** Line number
  - Value:** A line of text from the input file

# Input Formats

- **TextInputFormat**
  - Each \n-terminated line is a value
  - SequenceFiles are flat files consisting The byte offset of that line is a key
- **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 <key,value> pairs) serialized to files Or messages
  - Compact, fast, language-independent, self-describing, dynamic

# InputFormat class Hierarchy



## MapReduce Program for the Implementation of Word Count

**The following code block counts the number of words in a program.**

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

## MapReduce Program for the Implementation of Word Count – Contd.

```
public class WordCount
{
    public static class TokenizerMapper extends Mapper<Object, Text, Text,
    IntWritable>
    {
        private final static IntWritable one = new IntWritable(1); private Text
        word = new Text();

        public void map(Object key, Text value, Context
        context) throws IOException, InterruptedException
        {
            StringTokenizer itr = new StringTokenizer(value.toString()); while
            (itr.hasMoreTokens())
            {
                word.set(itr.nextToken()); context.write(word, one);
            }
        }
    }
}
```

## MapReduce Program for the Implementation of Word Count – Contd.

```
public static class IntSumReducer extends  
    Reducer<Text,IntWritable,Text,IntWritable>  
{  
    private IntWritable result = new IntWritable();  
    public void reduce(Text key, Iterable<IntWritable> values, Context  
        context) throws IOException, InterruptedException  
    {  
        int sum = 0;  
        for (IntWritable val : values)  
        {  
            sum += val.get();  
        }  
        result.set(sum); context.write(key, result);  
  
    }  
}
```

## MapReduce Program for the Implementation of Word Count – Contd.

```
public static void main(String[] args) throws Exception
{
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

## MapReduce Program 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

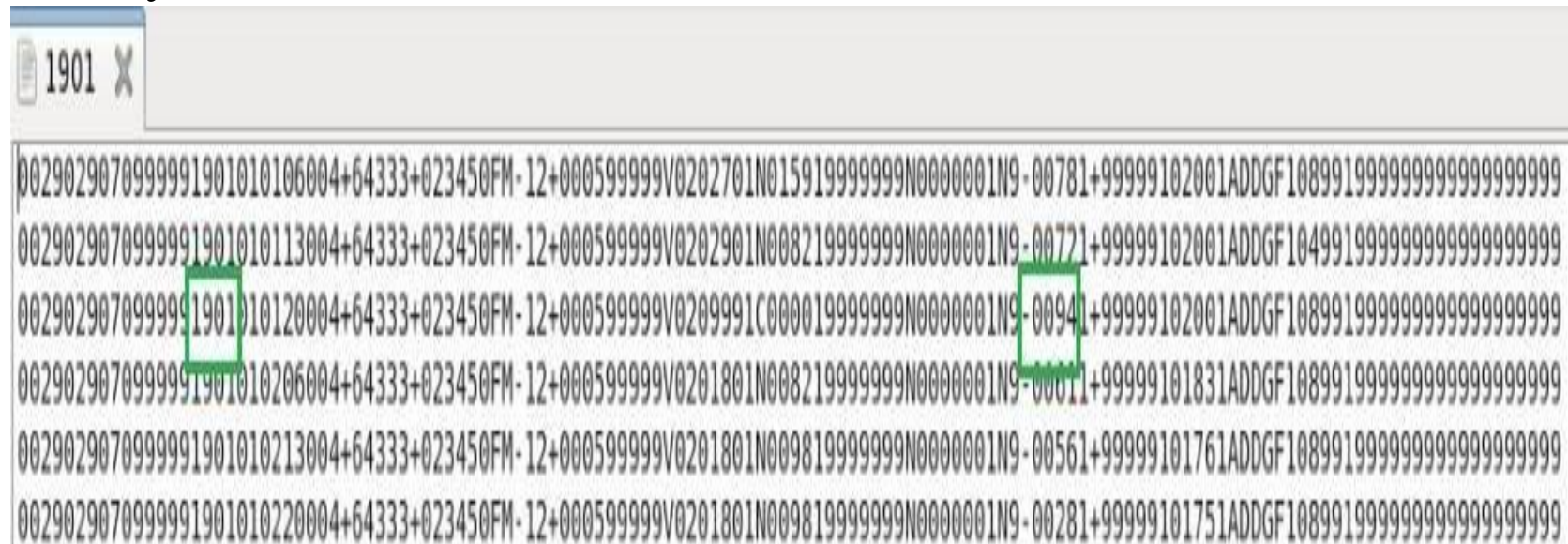
<https://github.com/tomwhite/hadoop-book/tree/master/input/ncdc/all>



## 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:



```

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+000599999V0201801N008219999999N0000001N9-00011+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 – Contd.

[https://github.com/  
msamarawickrama/  
Hadoop-  
MapReduce](https://github.com/msamarawickrama/Hadoop-MapReduce)

I  
m  
p  
l  
e  
m  
e  
n  
t  
a  
t  
i  
o  
n  
M  
a  
p  
R  
e  
d  
u  
c  
e  
P  
r  
o  
g  
r  
a  
m

## MapReduce Program on Weather Dataset – Contd.

[https://blog.eduonix.com  
/bigdata-and-  
hadoop/hadoop-project-  
on-ncdc-national-  
climate-data-center-  
noaa-dataset/](https://blog.eduonix.com/bigdata-and-hadoop/hadoop-project-on-ncdc-national-climate-data-center-noaa-dataset/)

I  
m  
p  
l  
e  
m  
e  
n  
t  
a  
t  
i  
o  
n  
M  
a  
p  
R  
e  
d  
u  
c  
e  
P  
r  
o  
g  
r  
a  
m

# Thank You All...