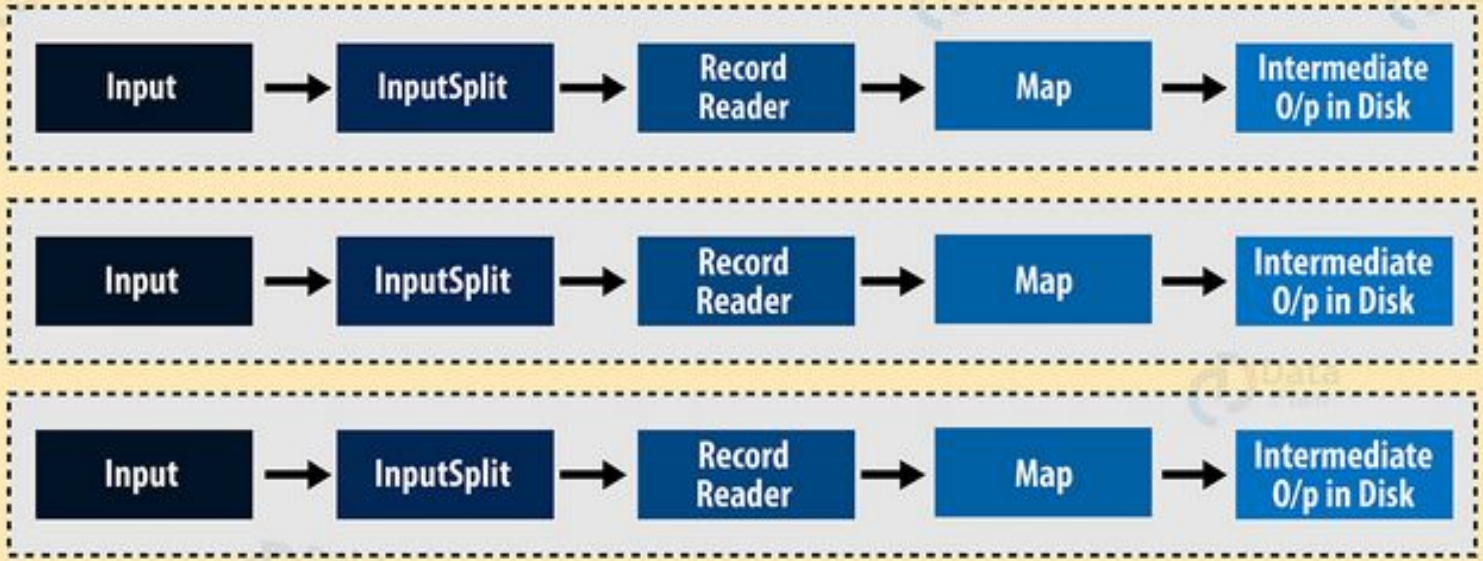

hadoop

MAPPER CLASS

- First phase of processing
- Processes each input record and generates an intermediate key-value pair.
- Mapper store intermediate-output on the local disk.

Mapper in Hadoop MapReduce



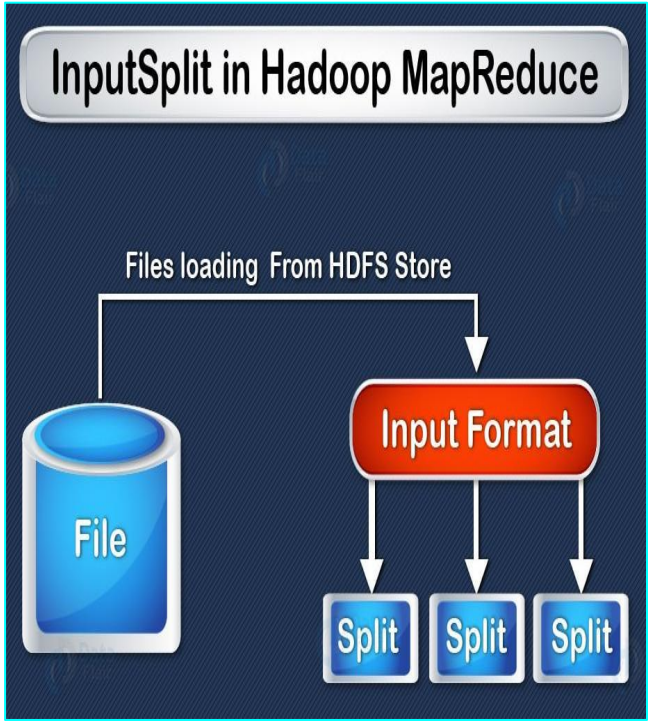
InputSplit (getSplits())

- An input to a MapReduce job is divided into fixed-size pieces called **input splits.**
- InputSplit is a chunk of the input that is consumed by a single map.
- It is the logical representation of data present in the block.
- InputSplit doesn't contain actual data, but reference to data.
- By default, split size is approx equal to block size (128 MB).
- InputSplit is user defined and the user can control split size based on the size of data in MapReduce program.
- Split size can be changed by changing - **mapred.max.split.size**

No. of map tasks/Input Splits= **total data size**

----

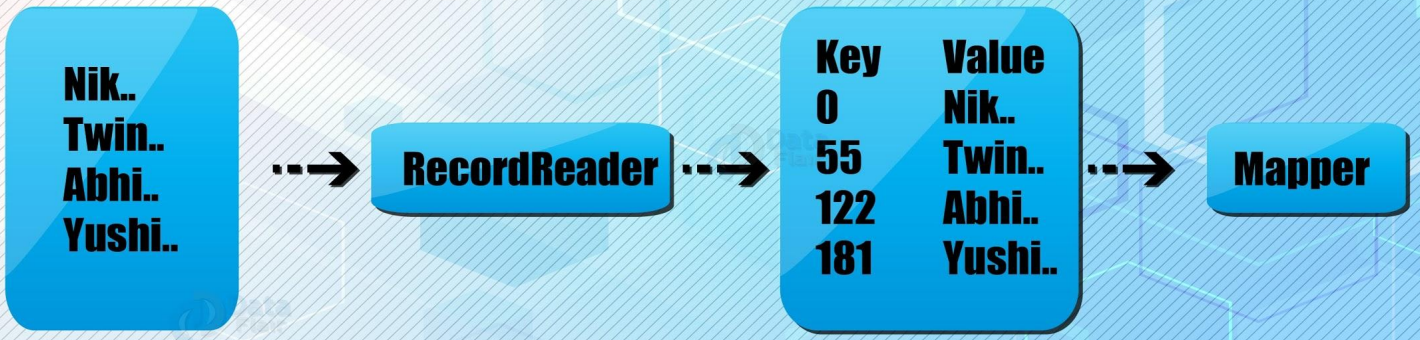
input split size



Record Reader (createRecordReader())

- The MapReduce RecordReader in Hadoop takes the byte-oriented view of input, provided by the InputSplit and presents as a record-oriented view for Mapper.
- It uses the data within the boundaries that were created by the InputSplit and creates Key-value pair.
- InputFormat class is responsible for input split and record reader.

RecordReader in Hadoop

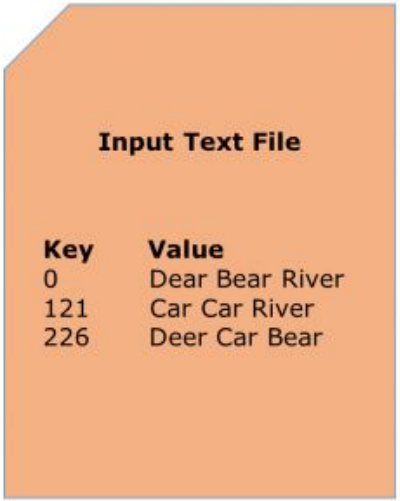


It Converts Data into key Value Format

Map Phase

```
public static class MapForWordCount extends Mapper<LongWritable, Text, Text, IntWritable>{

    public void map(LongWritable key, Text value, Context con) throws IOException, InterruptedException{

        String line = value.toString();
        String[] words=line.split(",");

        for(String word: words ){

              Text outputKey = new Text(word.toUpperCase().trim());
              IntWritable outputValue = new IntWritable(1);
              con.write(outputKey, outputValue);
        }
    }
}
```

- ➢ We have created a class Map that extends the class Mapper which is already defined in the MapReduce Framework.
- ➢ We define the data types of input and output key/value pair after the class declaration using angle brackets.
- ➢ Both the input and output of the Mapper is a key/value pair.
  - Input:
    - The *key* is nothing but the offset of each line in the text file: *LongWritable*
    - The *value* is each individual line (as shown in the figure at the right): *Text*
  - Output:
    - The *key* is the tokenized words: *Text*
    - We have the hardcoded *value* in our case which is 1: *IntWritable*
    - Example – Dear 1, Bear 1, etc.
- ➢ We have written a java code where we have tokenized each word and assigned them a hardcoded value equal to *1*.



Intermediate I/O in Disk

- In Hadoop,the output of Mapper is stored on local disk,as it is intermediate output.
- There is no need to store intermediate data on HDFS because :
  - Data write is costly and involves replication which further increases cost head and time.
  - Intermediate data is required only unless it is sent to the reducer for further processing to get the final output,so not needed to store permanently,thus stored on local disk only.

Mapper for Log

```
public static class MapForLog extends Mapper<LongWritable, Text, Text, IntWritable>{

    public void map(LongWritable key, Text value, Context con) throws IOException, InterruptedException{

        String line = value.toString();
        String[] words=line.split(",");
        Text outputKey = null;
        long diff;
        IntWritable outputValue;
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        LocalDateTime time1, time2 = null;

        for(int i=0;i<words.length;i=i+8)
        {
            outputKey = new Text(words[i+1].trim());
            time1 = LocalDateTime.parse(words[i+5],formatter);
            time2 = LocalDateTime.parse(words[i+7],formatter);
            diff = java.time.Duration.between(time1,time2).toMinutes();
            outputValue = new IntWritable((int)diff);

            con.write(outputKey, outputValue);
        }
    }
}
```

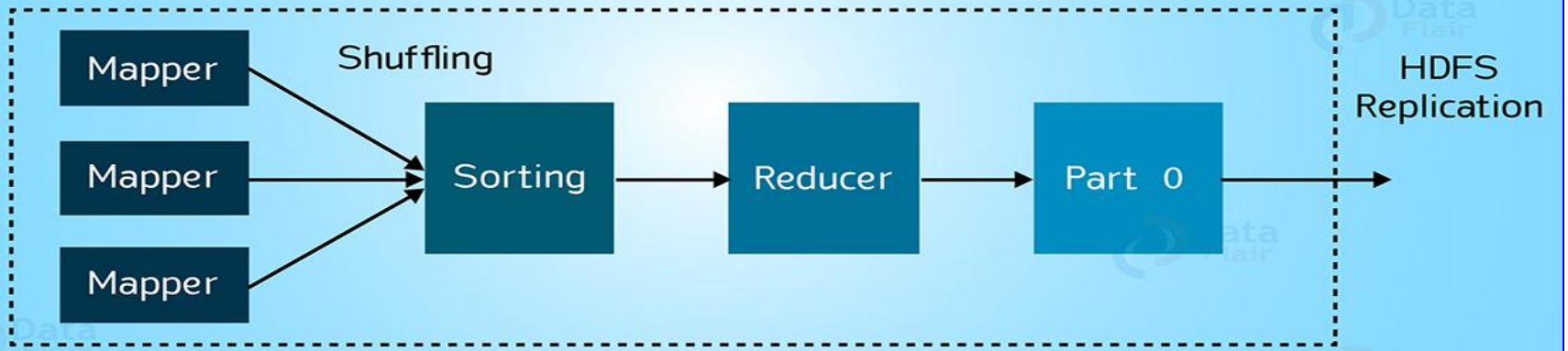
Mapper for Weather

```
public static class MapForWeather extends Mapper<LongWritable, Text, Text, DoubleWritable>{

    public void map(LongWritable key, Text value, Context con) throws IOException, InterruptedException{

        String line = value.toString();
        String[] words=line.split(",");
        Text outputKey = null;

        for(int i=0;i<words.length;i=i+9)
        {
        outputKey = new Text(words[i].trim());
        double min = Double.parseDouble(words[i+1]) + Double.parseDouble(words[i+3]) + Double.parseDouble(words[i+5]) + Double.parseDouble(words[i+7]);
        double max = Double.parseDouble(words[i+2]) + Double.parseDouble(words[i+4]) + Double.parseDouble(words[i+6]) + Double.parseDouble(words[i+8]);
        double mean = (min+max)/8;
        con.write(outputKey, new DoubleWritable(mean));
        }
    }
}
```

REDUCER CLASS

- The Reducer process the output of the mapper.
- Hadoop Reducer takes a set of an intermediate key-value pair produced by the mapper as the input and runs a Reducer function on each of them.
- 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.
- With the help of *Job.setNumreduceTasks(int)* the user set the number of reducers for the job.

Hadoop Reducer



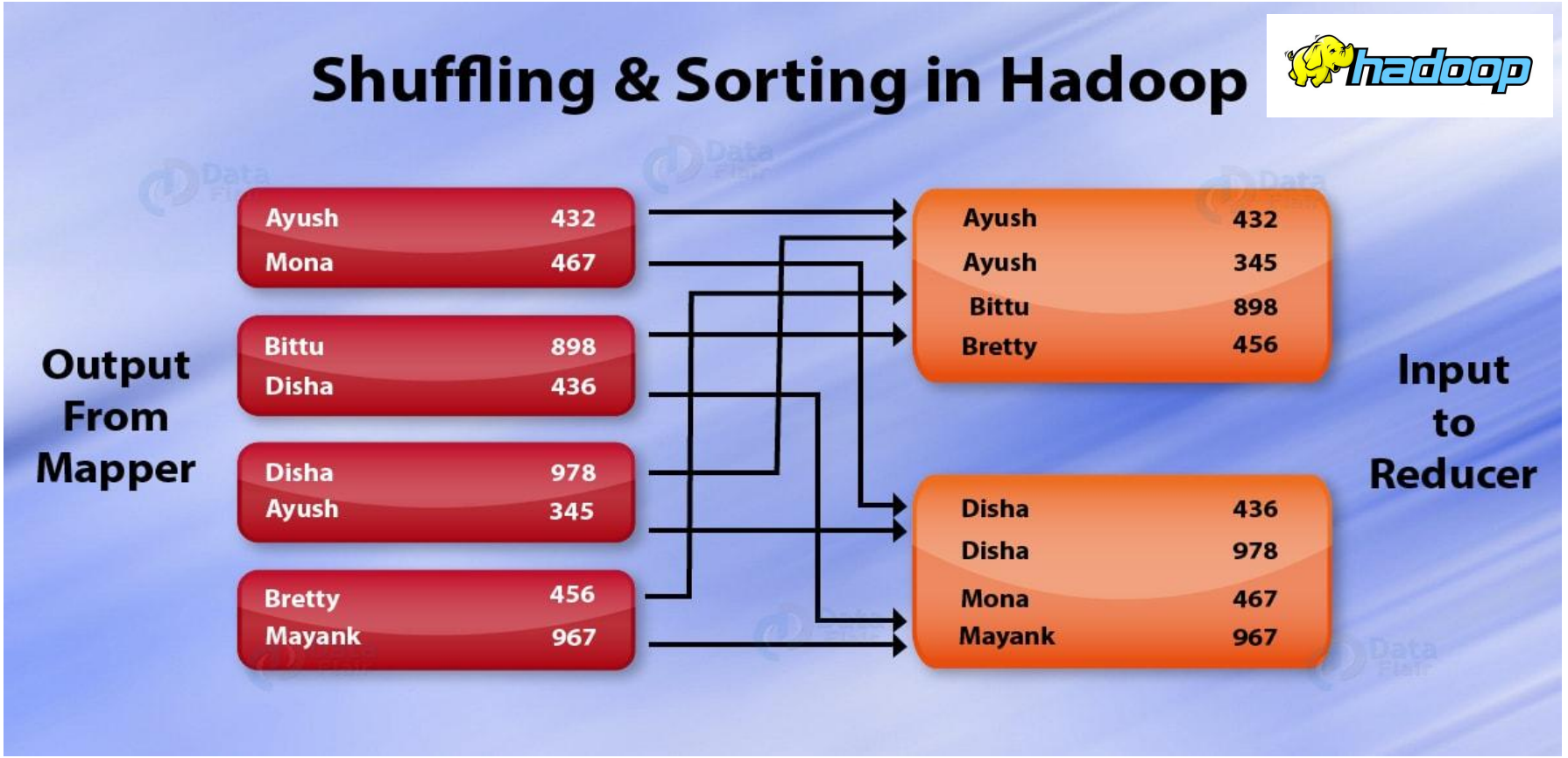
Shuffling / Partitioning

- Partitioning of the keys of the intermediate map output is controlled by the Partitioner.
- By hash function, key (or a subset of the key) is used to derive the partition.
- According to the key-value each mapper output is partitioned and records having the same key value go into the same partition (within each mapper), and then each partition is sent to a reducer.
- Partition class determines which partition a given (key, value) pair will go.

Sorting

- In this phase, the input from different mappers is again sorted based on the similar keys in different Mappers.
- The shuffle and sort phases occur concurrently.

Shuffling & Sorting in Hadoop



Reduce Phase

```
public static class ReduceForWordCount extends Reducer<Text, IntWritable, Text, IntWritable>{

    public void reduce(Text word, Iterable<IntWritable> values, Context con) throws IOException, InterruptedException{

        int sum = 0;

       for(IntWritable value : values){

           sum += value.get();
       }
       con.write(word, new IntWritable(sum));
    }
}
```

- We have created a class Reduce which extends class Reducer like that of Mapper.
- We define the data types of input and output key/value pair after the class declaration using angle brackets as done for Mapper.
- Both the input and the output of the Reducer is a key-value pair.
- Input:
  - The *key* nothing but those unique words which have been generated after the sorting and shuffling phase: *Text*
  - The *value* is a list of integers corresponding to each key: *IntWritable*
  - Example – Bear, [1, 1], etc.
- Output:
  - The *key* is all the unique words present in the input text file: *Text*
  - The *value* is the number of occurrences of each of the unique words: *IntWritable*
  - Example – Bear, 2; Car, 3, etc.
- We have aggregated the values present in each of the list corresponding to each key and produced the final answer.
- In general, a single reducer is created for each of the unique words, but, you can specify the number of reducer in mapred-site.xml.

Part 0

- The output files are by default named part-x-yyyyy where:
  - x is either 'm' or 'r', depending on whether the job was a map only job, or reduce
  - yyyyy is the mapper or reducer task number (zero based)
- So a job which has 32 reducers will have files named part-r-00000 to part-r-00031, one for each reducer task.

Reducer for Log

```
public static class ReduceForLog extends Reducer<Text, IntWritable, Text, IntWritable>{
    int max_sum = 0;
    Text max_logtime = new Text();
    public void reduce(Text word, Iterable<IntWritable> values, Context con) throws IOException, InterruptedException{

        int sum = 0;

       for(IntWritable value : values)
       {
            sum += value.get();
       }
       if(sum > max_sum)
       {
           max_sum = sum;
           max_logtime.set(word);
       }
    }
    protected void cleanup(Context con) throws IOException, InterruptedException{
        con.write(max_logtime, new IntWritable(max_sum));
    }
}
```

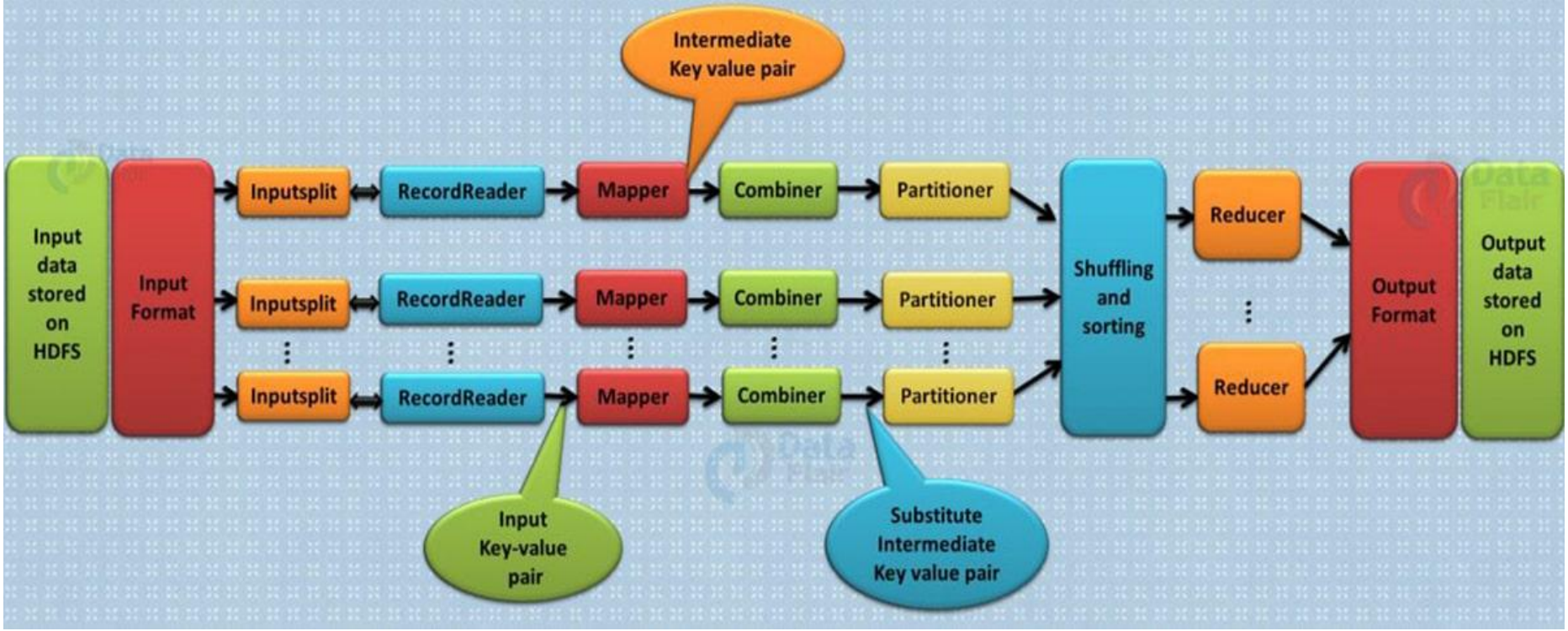
Reducer for Weather

```
public static class ReduceForWeather extends Reducer<Text, DoubleWritable, Text, DoubleWritable>{
    double max_temp=0.0;
    double min_temp=999.0;
    Text hottest=new Text();
    Text coolest=new Text();
    public void reduce(Text word, Iterable<DoubleWritable> values, Context con) throws IOException, InterruptedException{

        double sum = 0;

       for(DoubleWritable value : values)
       {
           sum += value.get();
       }
       if(sum > max_temp) {
            max_temp = sum;
            hottest.set(word);
        }
        if(sum < min_temp) {
            min_temp = sum;
            coolest.set(word);
        }
    }
    protected void cleanup(Context con) throws IOException, InterruptedException {
        con.write(hottest, new DoubleWritable(max_temp));
        con.write(coolest, new DoubleWritable(min_temp));
    }
}
```

Reducer methods

- Protected void **setup** - called once at the start of the task
- Protected void **reduce** - called once for each key/partition
- Protected void **run** - advanced option to control how reducer works
- Protected void **cleanup** - called once at the end of the task

MapReduce Job Execution Flow



Combiner

- There are number of key-value pairs generated together by the mapper.
- So, it becomes necessary to control congestion.
- This task is done by Combiner.
- However, MapReduce may or may not have Combiner.

How is data brought to mapper?

Input Format

- Initially, the data for a MapReduce task is stored in input files, and input files typically reside in HDFS.
- Although these files format is arbitrary, line-based log files and binary format can be used.
- Using InputFormat we define how these input files are split and read.
- The InputFormat class is one of the fundamental classes in the Hadoop MapReduce framework which provides the following functionality:
  - The files or other objects that should be used for input is selected by the InputFormat.
  - InputFormat defines the Data splits, which defines both the size of individual Map tasks and its potential execution server.
  - InputFormat defines the RecordReader, which is responsible for reading actual records from the input files.

Types of InputFormat in MapReduce

- FileInputFormat
- TextInputFormat
- KeyValueTextInputFormat
- SequenceFileInputFormat
- SequenceFileAsTextInputFormat
- NLineInputFormat
- SequenceFileAsBinaryInputFormat
- DBInputFormat

FileInputFormat

- It is the base class for all file-based InputFormats.
- Hadoop FileInputFormat specifies input directory where data files are located.
- When we start a Hadoop job, FileInputFormat is provided with a path containing files to read.
- FileInputFormat will read all files and divides these files into one or more InputSplits.

Output Format

- The Output Format and InputFormat functions are alike.
- OutputFormat instances provided by Hadoop are used to write to files on the HDFS or local disk.
- OutputFormat describes the output-specification for a Map-Reduce job.
- On the basis of output specification;
  - MapReduce job checks that the output directory does not already exist.
  - OutputFormat provides the RecordWriter implementation to be used to write the output files of the job. Output files are stored in a FileSystem.
- FileOutputFormat.setOutputPath() method is used to set the output directory. Every Reducer writes a separate file in a common output directory.

DRIVER CLASS

We should provide the following configuration details in a driver class -

- Input class for jar
- Mapper className
- Reducer ClassName
- Output key class
- Input key class
- File input path
- File out path

Driver Class

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

    Configuration c=new Configuration();
    String[] files=new GenericOptionsParser(c,args).getRemainingArgs();
    Path input=new Path(files[0]);
    Path output=new Path(files[1]);
    Job j=new Job(c,"wordcount");
    j.setJarByClass(WordCount.class);
    j.setMapperClass(MapForWordCount.class);
    j.setReducerClass(ReduceForWordCount.class);
    j.setOutputKeyClass(Text.class);
    j.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(j, input);
    FileOutputFormat.setOutputPath(j, output);
    System.exit(j.waitForCompletion(true)?0:1);
}
```

Configuration c = new Configuration();

Package - org.apache.hadoop.conf.Configuration

Role - Use of configuration class in Hadoop is for setting all required configurations like.. defining which type of input/output class, type of input/output file format, mapper and reducer classes etc.. we are using for our mapreduce program.

String[] files=new GenericOptionsParser(c,args).getRemainingArgs();

Package - org.apache.hadoop.util.GenericOptionsParser

Role - GenericOptionsParser is a utility to parse command line arguments generic to the Hadoop framework. GenericOptionsParser recognizes several standard command line arguments

Path input=new Path(files[0]); / Path output=new Path(files[1]);

Package - org.apache.hadoop.fs.Path

Role - Names a file or directory in a FileSystem. Path strings use slash as the directory separator. A path string is absolute if it begins with a slash.

Job j=new Job(c,"weather");

Package - org.apache.hadoop.mapreduce.Job

Role - The job submitter's view of the Job.

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 various facets of the job via Job and then submits the job and monitor its progress.

j.setJarByClass(Weather.class);

Package - org.apache.hadoop.mapreduce.Job

Role - Here we help Hadoop to find out that which jar it should send to nodes to perform Map and Reduce tasks.

Hence, using this setJarByClass method we tell Hadoop to find out the relevant jar by finding out that the class specified as it's parameter to be present as part of that jar.

j.setMapperClass(MapForWeather.class); / j.setReducerClass(ReduceForWeather.class);

Package - org.apache.hadoop.mapreduce.Job

Role - This method tells the job, which class is the mapper/reducer for that particular job.

j.setOutputKeyClass(Text.class); / j.setOutputValueClass(IntWritable.class);

Package - org.apache.hadoop.io.IntWritable; / org.apache.hadoop.io.LongWritable; / org.apache.hadoop.io.Text;

Role - This method tells the format of output of the job.

FileInputFormat.addInputPath(j, input);

Package - org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

Role - Set InputFormat

FileOutputFormat.setOutputPath(j, output);

Package - org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

Role - Set OutputFormat

Packages

```
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.LongWritable;
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;
import org.apache.hadoop.util.GenericOptionsParser;
```