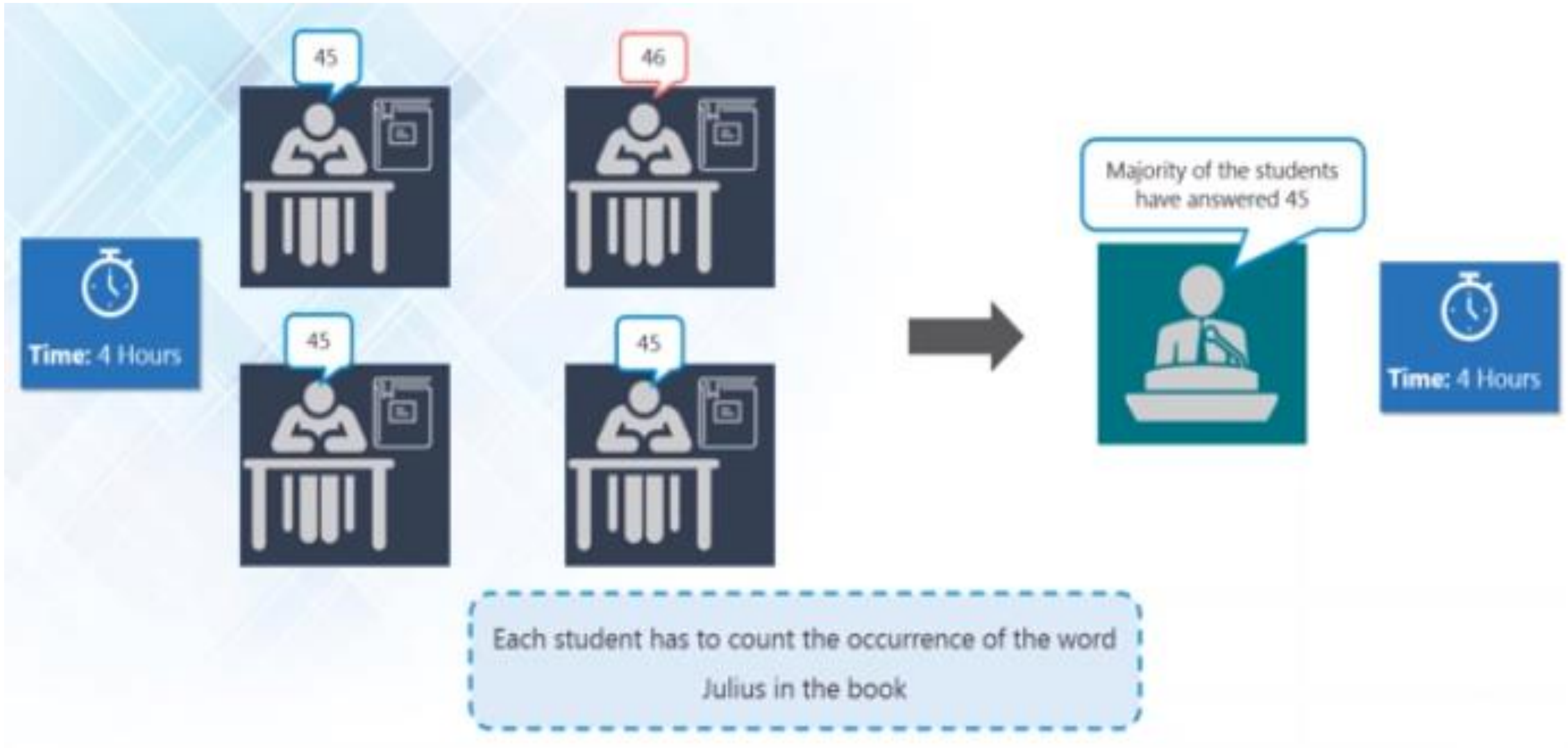
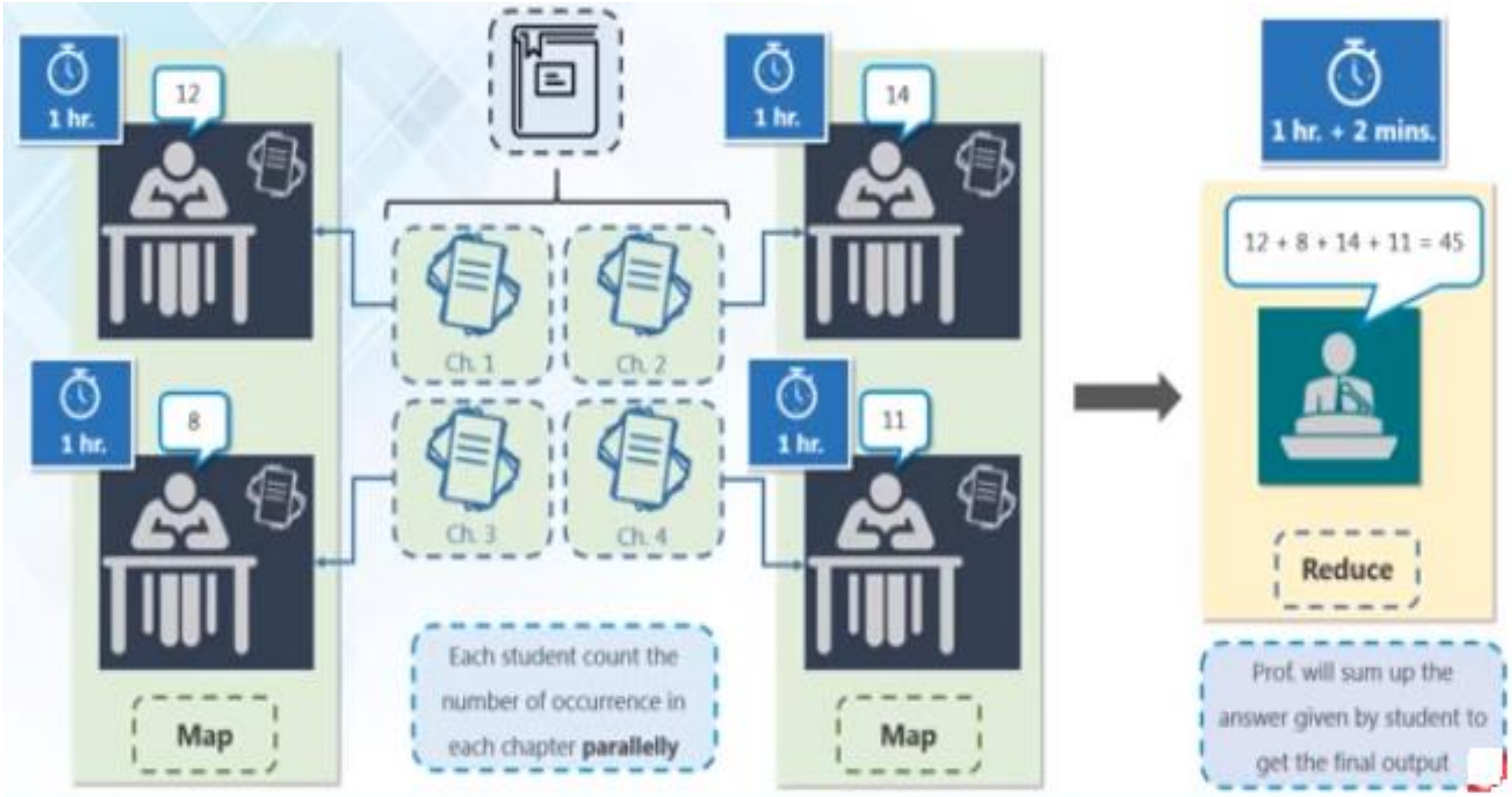
MapReduce

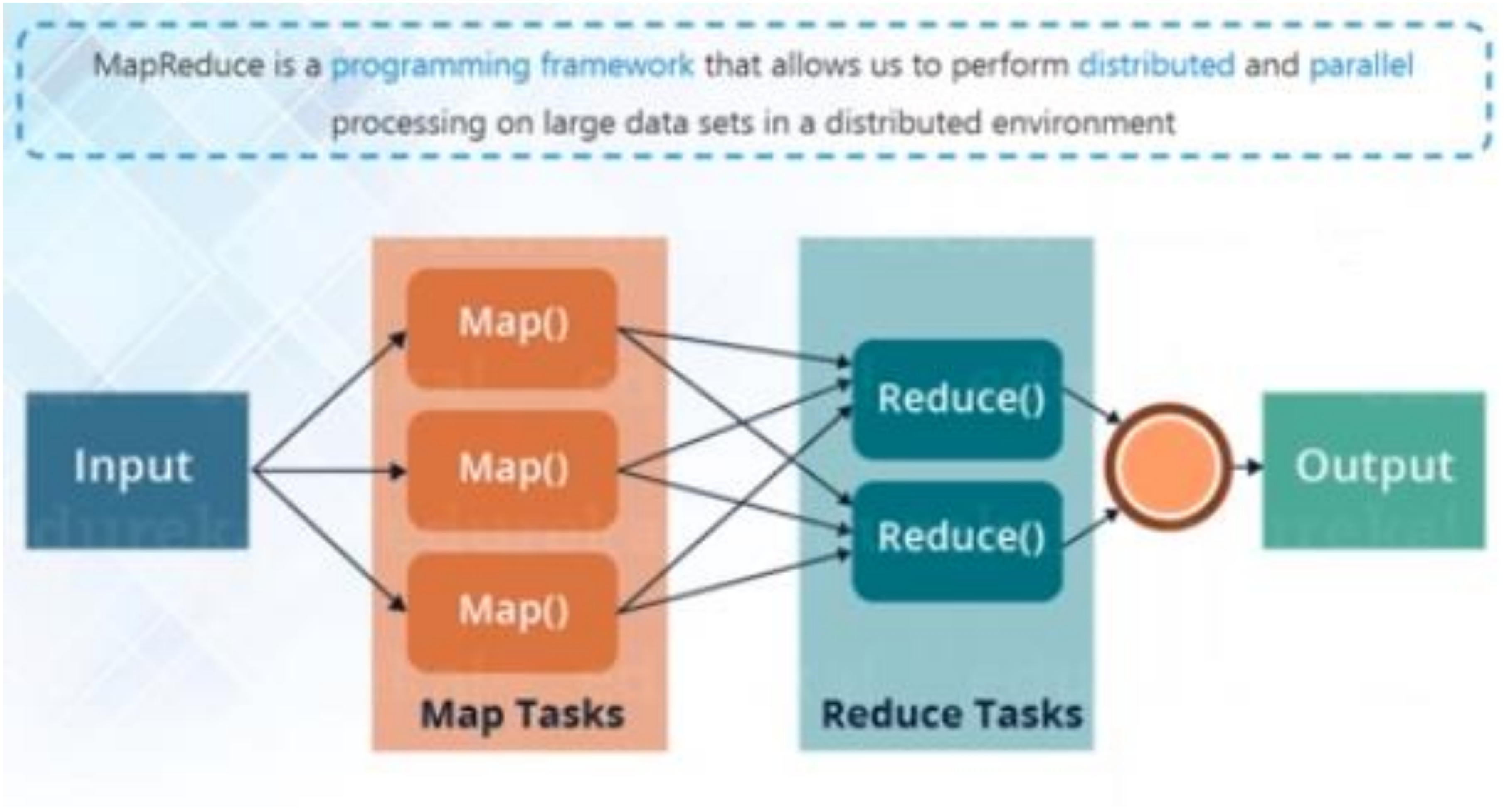
Story of MapReduce


Time: 4 Hours
45
46
45
45
Majority of the students have answered 45
Time: 4 Hours
Each student has to count the occurrence of the word Julius in the book

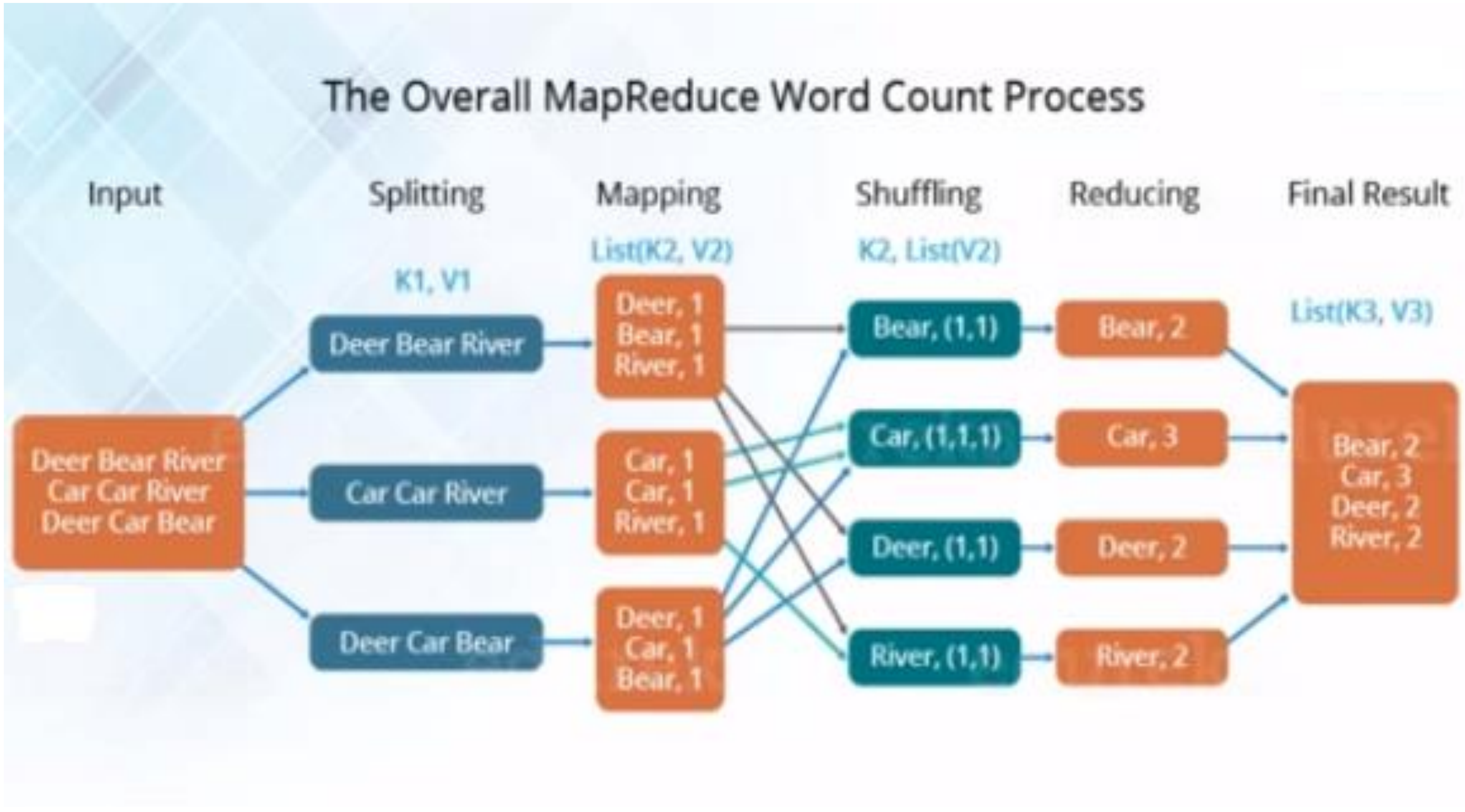
Story of MapReduce


1 hr.
12
1 hr.
8
Map
Ch. 1
Ch. 2
Ch. 3
Ch. 4
Each student count the number of occurrence in each chapter **parallelly**
1 hr.
14
1 hr.
11
Map
1 hr. + 2 mins.
12 + 8 + 14 + 11 = 45
Reduce
Prof. will sum up the answer given by student to get the final output

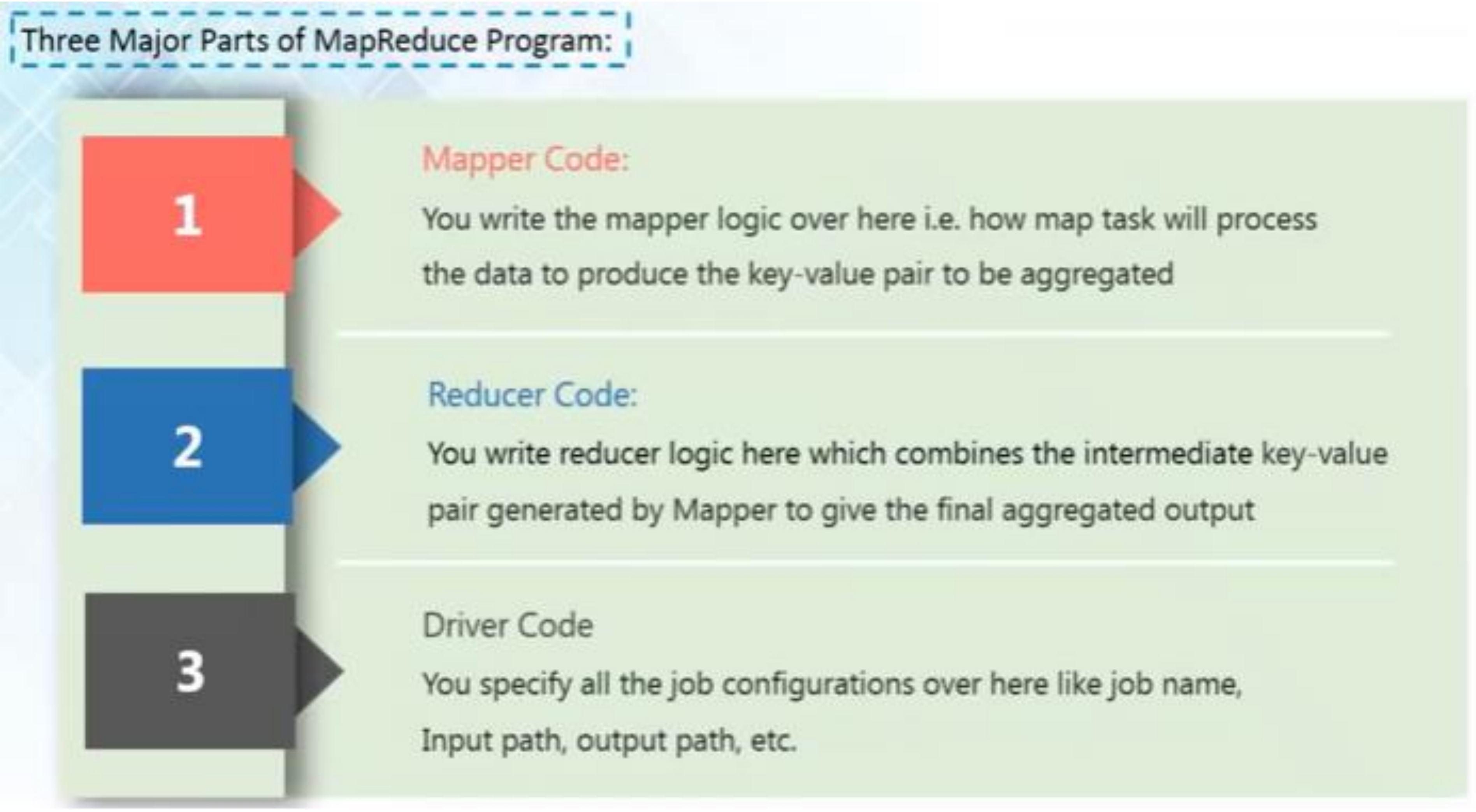
What is MapReduce?


MapReduce is a programming framework that allows us to perform distributed and parallel processing on large data sets in a distributed environment

MapReduce Word Count Example



MapReduce Wordcount Example



Three Major Parts of MapReduce Program:

1

Mapper Code:

You write the mapper logic over here i.e. how map task will process the data to produce the key-value pair to be aggregated

2

Reducer Code:

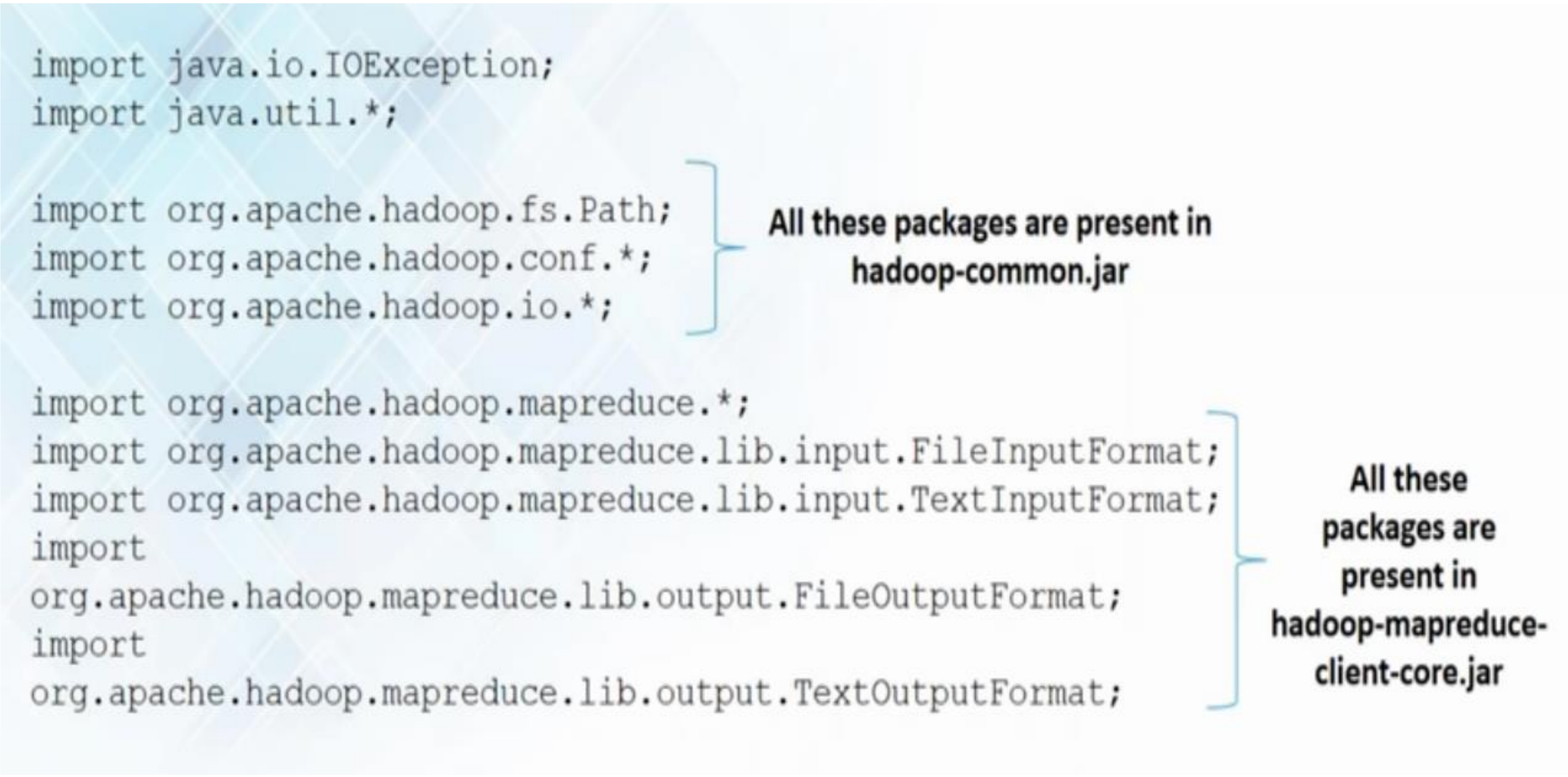
You write reducer logic here which combines the intermediate key-value pair generated by Mapper to give the final aggregated output

3

Driver Code

You specify all the job configurations over here like job name, Input path, output path, etc.

Packages and classes

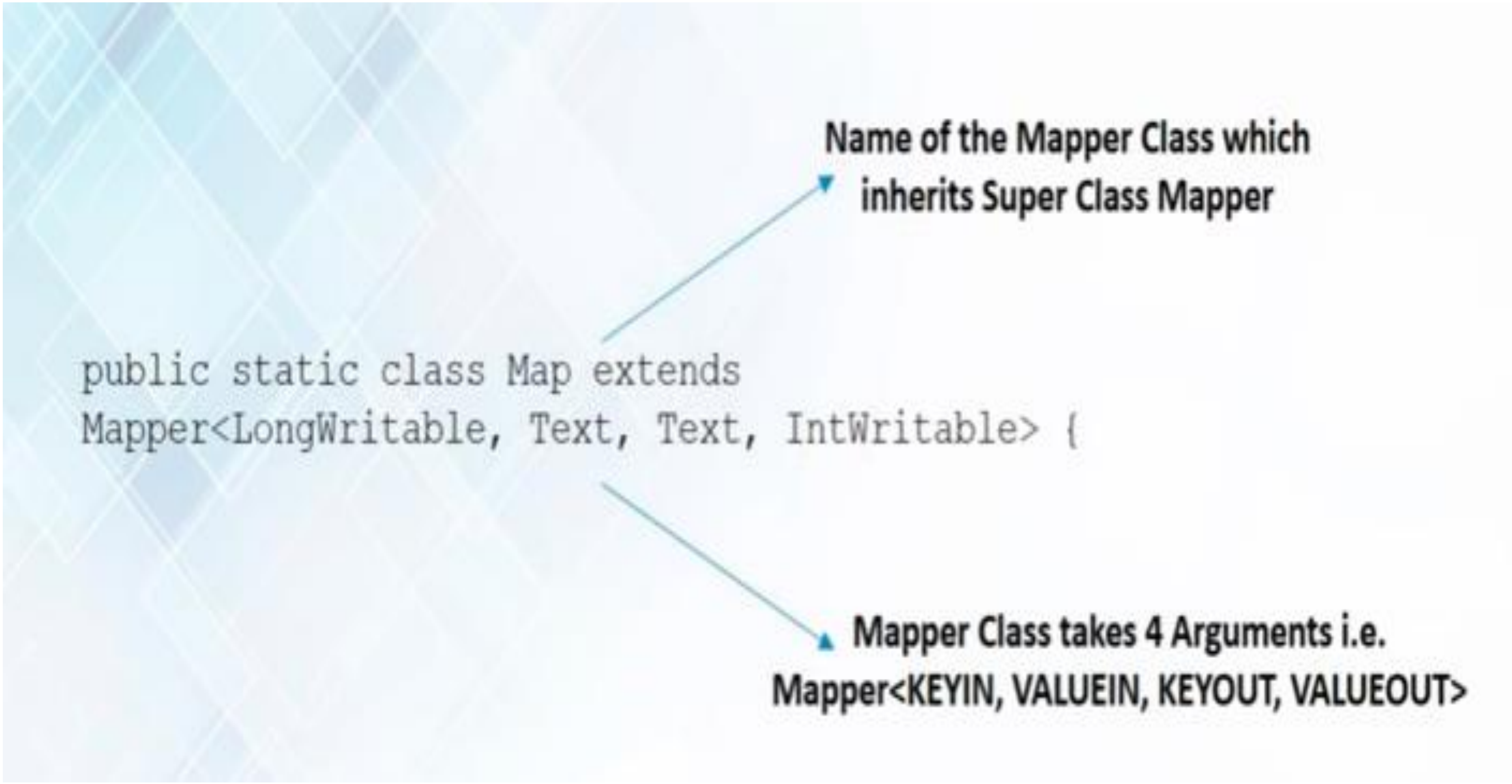


```
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;

import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import
org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import
org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
```

All these packages are present in hadoop-common.jar

All these packages are present in hadoop-mapreduce-client-core.jar

Mapper Class

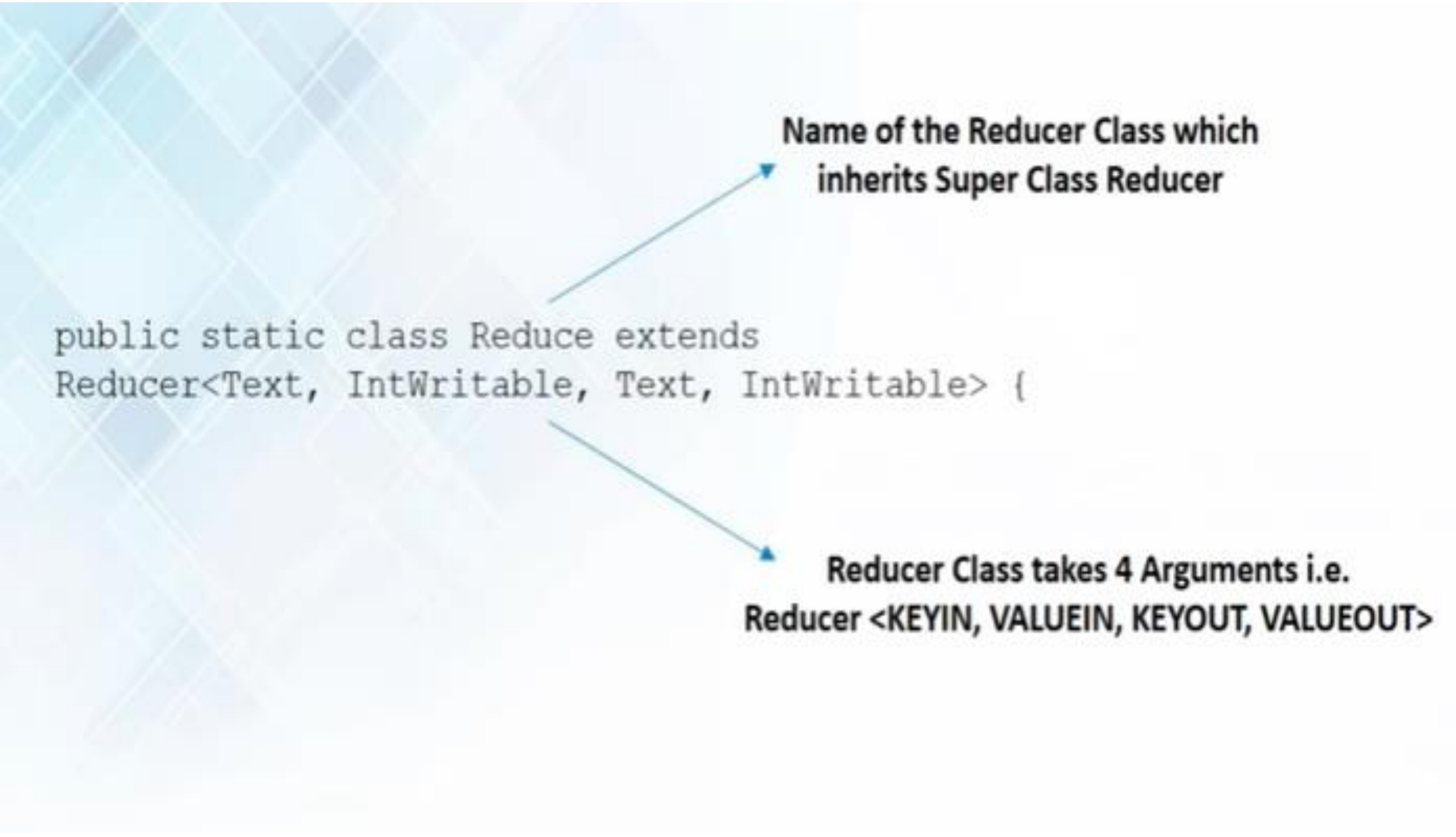


Name of the Mapper Class which inherits Super Class Mapper

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

Mapper Class takes 4 Arguments i.e.
Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>

Reducer Class



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

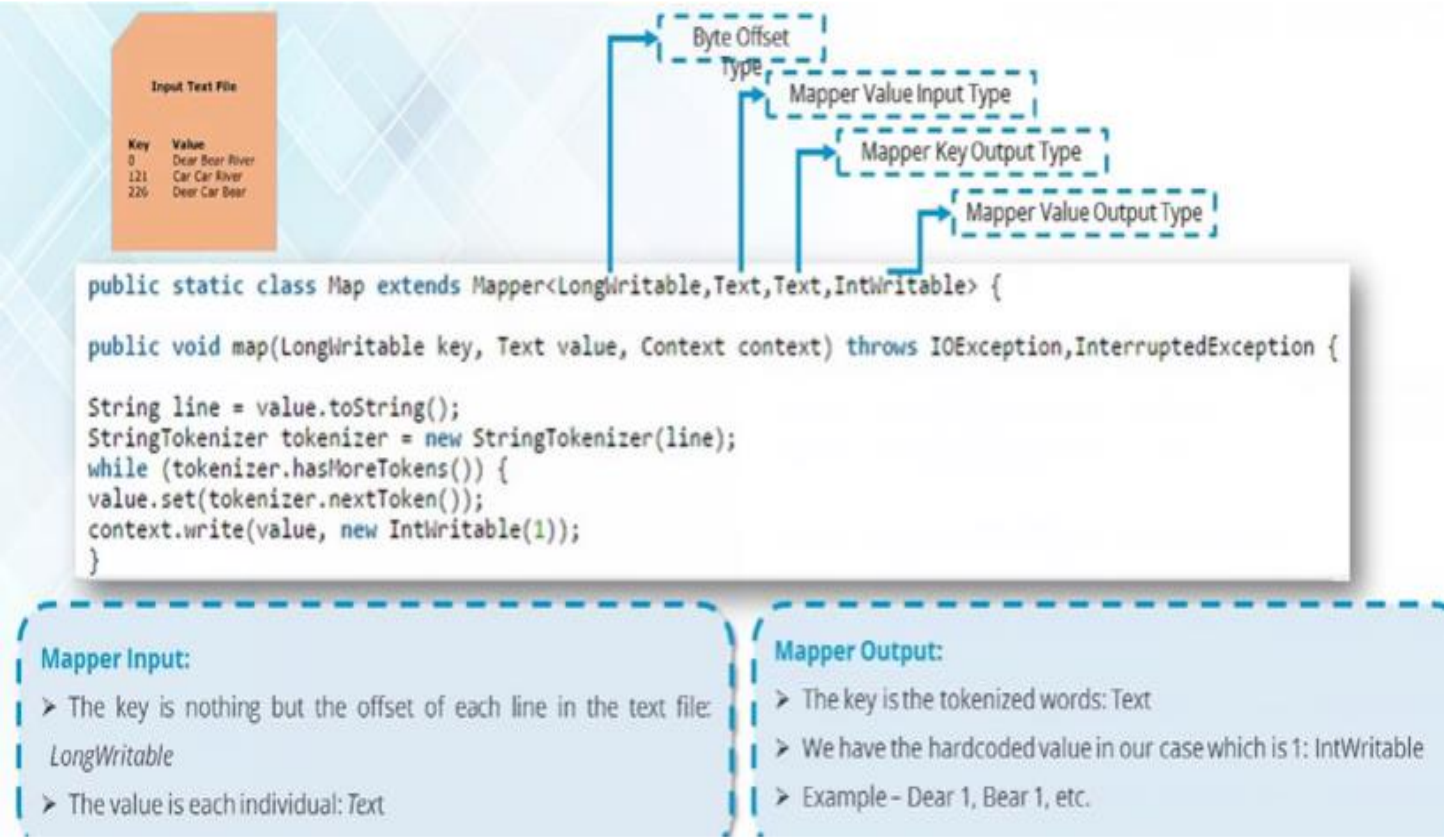
Name of the Reducer Class which inherits Super Class Reducer

Reducer Class takes 4 Arguments i.e. Reducer <KEYIN, VALUEIN, KEYOUT, VALUEOUT>

Byte Offset

- Byte offset is the number of character that exists counting from the beginning of a line.
- Byte offset is represented hexadecimal.
- E.g. this line “what is byte offset” will have a byte offset of 19. This is used as key value in hadoop
- Key = byte offset, value = line

Mapper Code



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

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

String line = value.toString();
StringTokenizer tokenizer = new StringTokenizer(line);
while (tokenizer.hasMoreTokens()) {
value.set(tokenizer.nextToken());
context.write(value, new IntWritable(1));
}
```

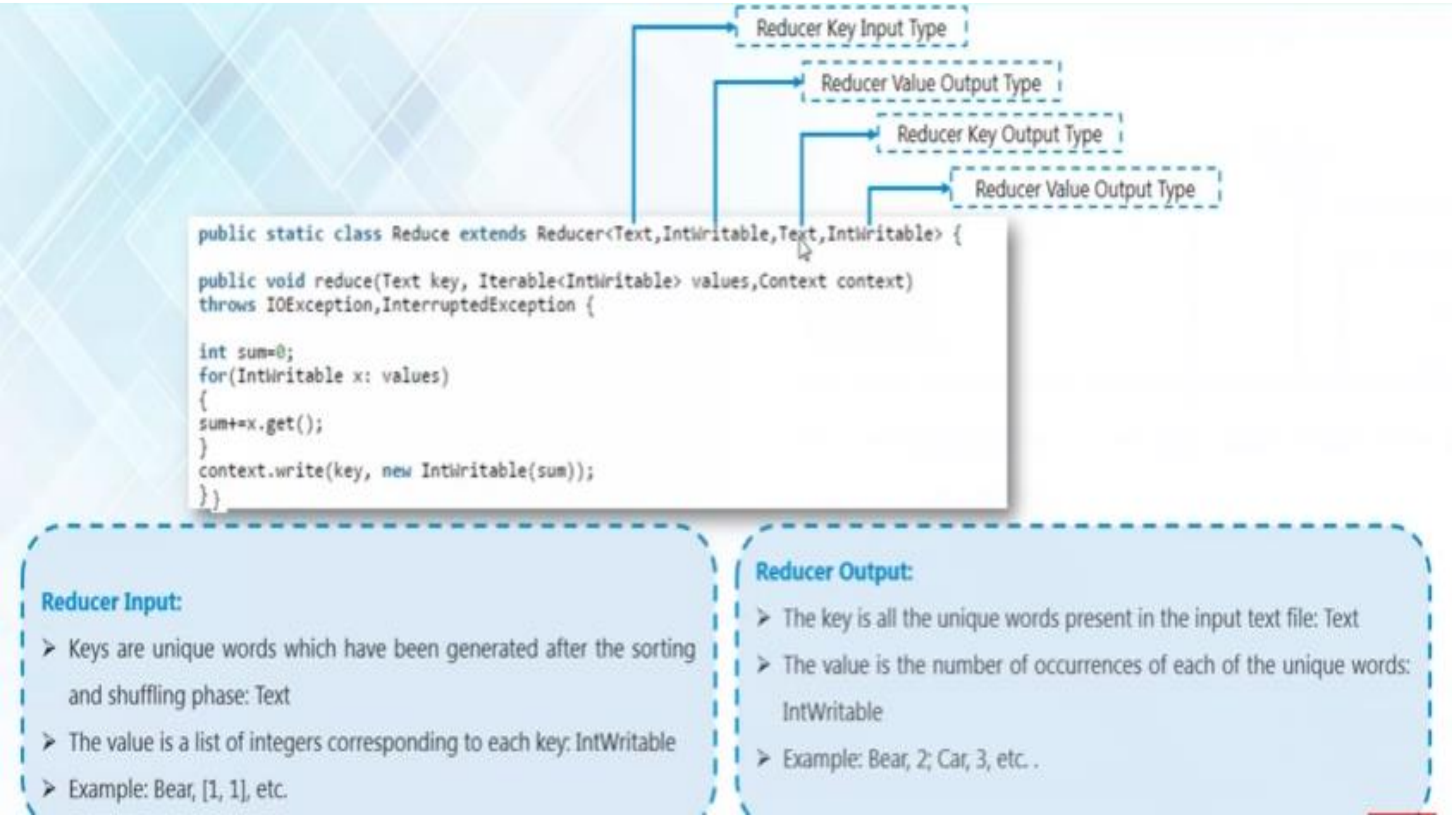
Mapper Input:

- The key is nothing but the offset of each line in the text file: *LongWritable*
- The value is each individual: *Text*

Mapper 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.

Reducer Code



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

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

int sum=0;
for(IntWritable x: values)
{
sum+=x.get();
}
context.write(key, new IntWritable(sum));
}}
```

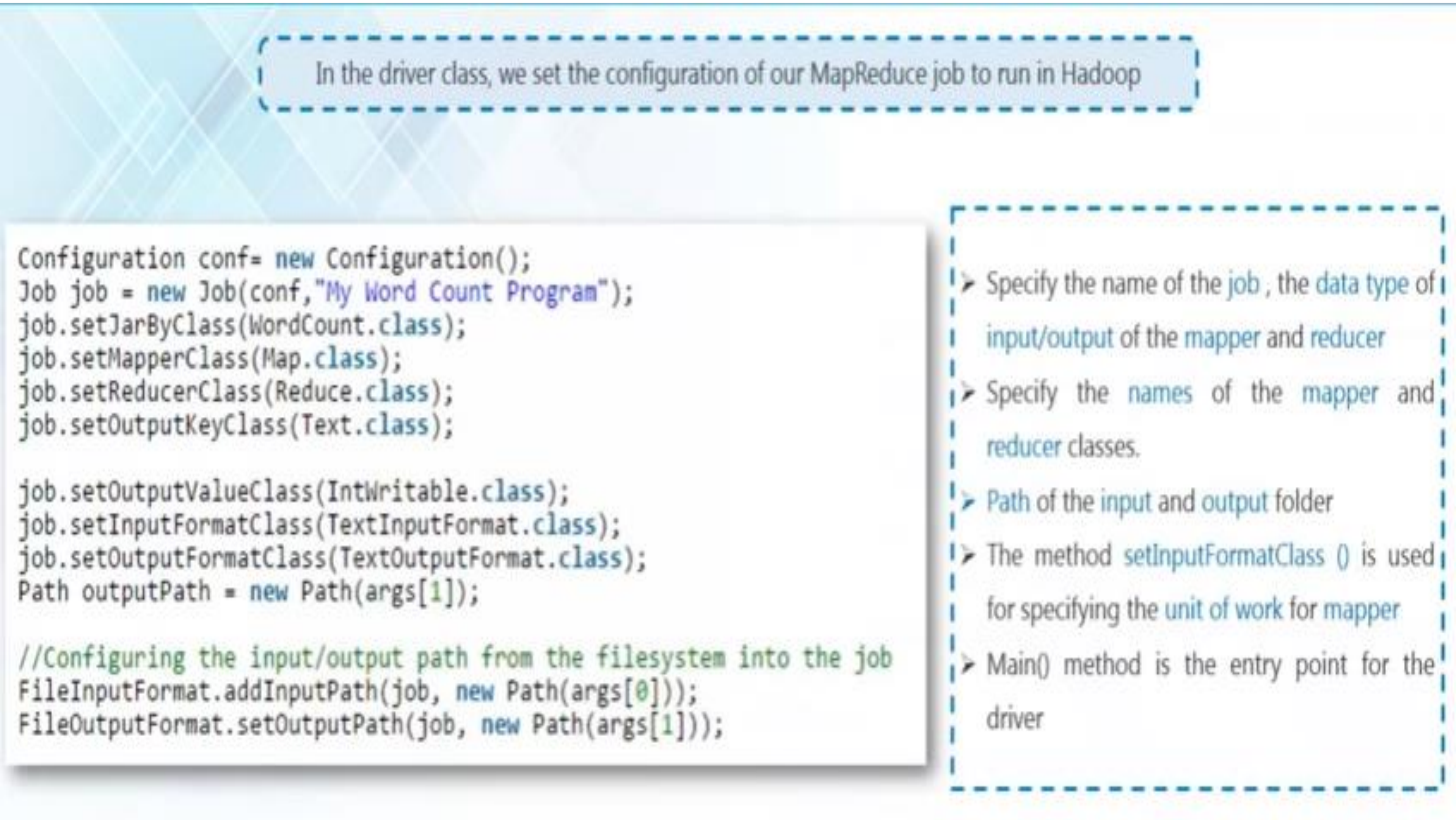
Reducer Input:

- Keys are 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.

Reducer 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. .

Driver Code



In the driver class, we set the configuration of our MapReduce job to run in Hadoop

```
Configuration conf= new Configuration();
Job job = new Job(conf,"My Word Count Program");
job.setJarByClass(WordCount.class);
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);
job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
Path outputPath = new Path(args[1]);

//Configuring the input/output path from the filesystem into the job
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

- Specify the name of the job , the data type of input/output of the mapper and reducer
- Specify the names of the mapper and reducer classes.
- Path of the input and output folder
- The method setInputFormatClass () is used for specifying the unit of work for mapper
- Main() method is the entry point for the driver

MapReduce Word Count Example

What is Task in Map Reduce?

- A task in MapReduce is an execution of a Mapper or a Reducer on a slice of data.
- It is also called Task-In-Progress (TIP).
- It means processing of data is in progress either on mapper or reducer.

What is Task Attempt?

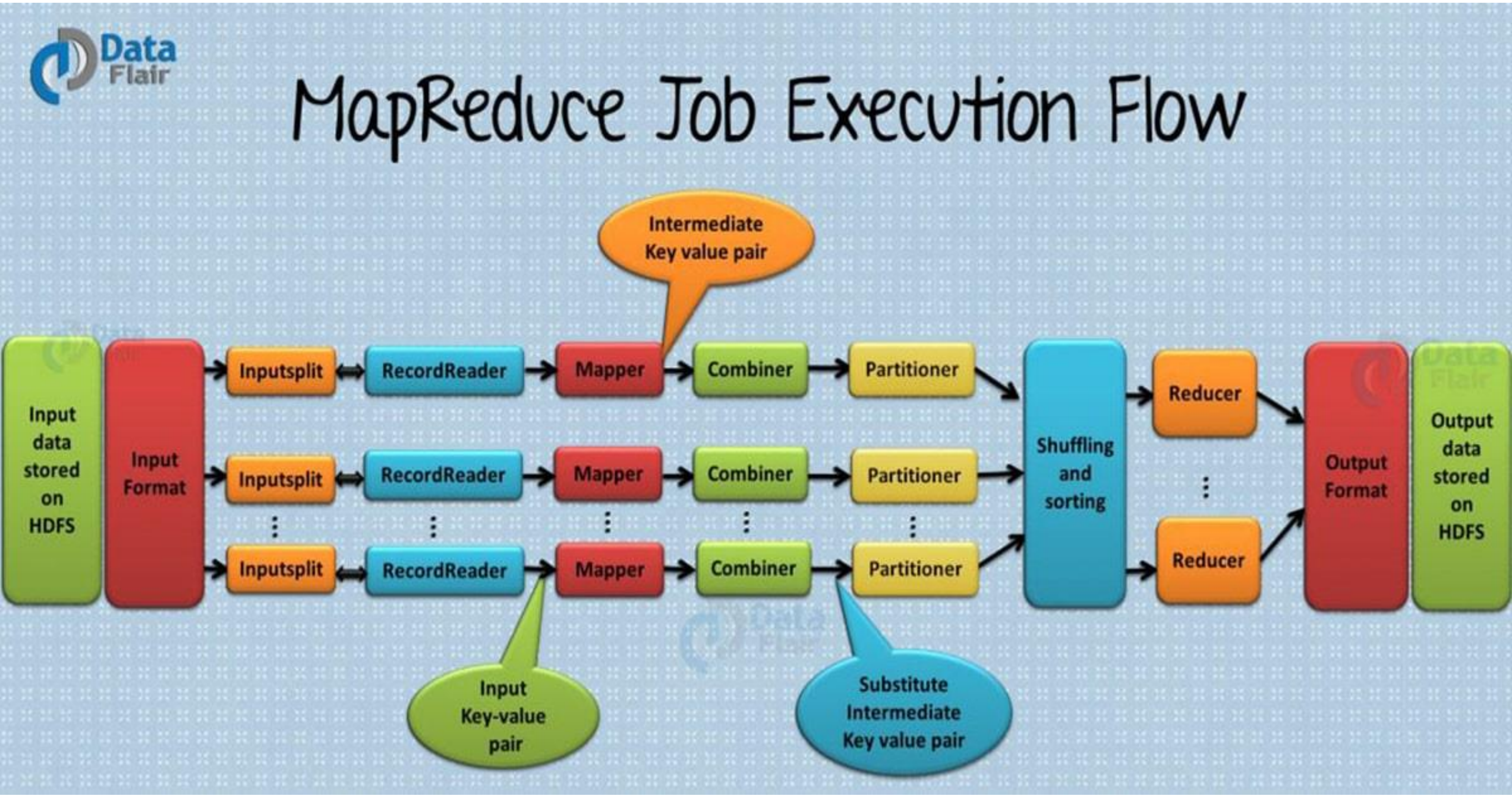
- Task Attempt is a particular instance of an attempt to execute a task on a node.
- There is a possibility that anytime any machine can go down.
- For example, while processing data if any node goes down, framework reschedules the task to some other node. This rescheduling of the task cannot be infinite.
- There is an upper limit for that as well. The default value of task attempt is 4. If a task (Mapper or reducer) fails 4 times, then the job is considered as a failed job.
- For high priority job or huge job, the value of this task attempt can also be increased.

Failed Task v/s Killed Task

- A failed task attempt is a task attempt that completed, but with an unexpected status value.
- A killed task attempt is a duplicate copy of a task attempt that was started as part of speculative execution.
- Hadoop uses "speculative execution." The same task may be started on multiple boxes. The first one to finish wins, and the other copies are killed.
- Failed tasks are tasks that error out.
- There are a few reasons Hadoop can kill tasks by its own decisions:
  - Task does not report progress during timeout (default is 10 minutes)
  - FairScheduler or CapacityScheduler needs the slot for some other pool (FairScheduler) or queue (CapacityScheduler).
  - Speculative execution causes results of task not to be needed since it has completed on other place.

MapReduce Phases

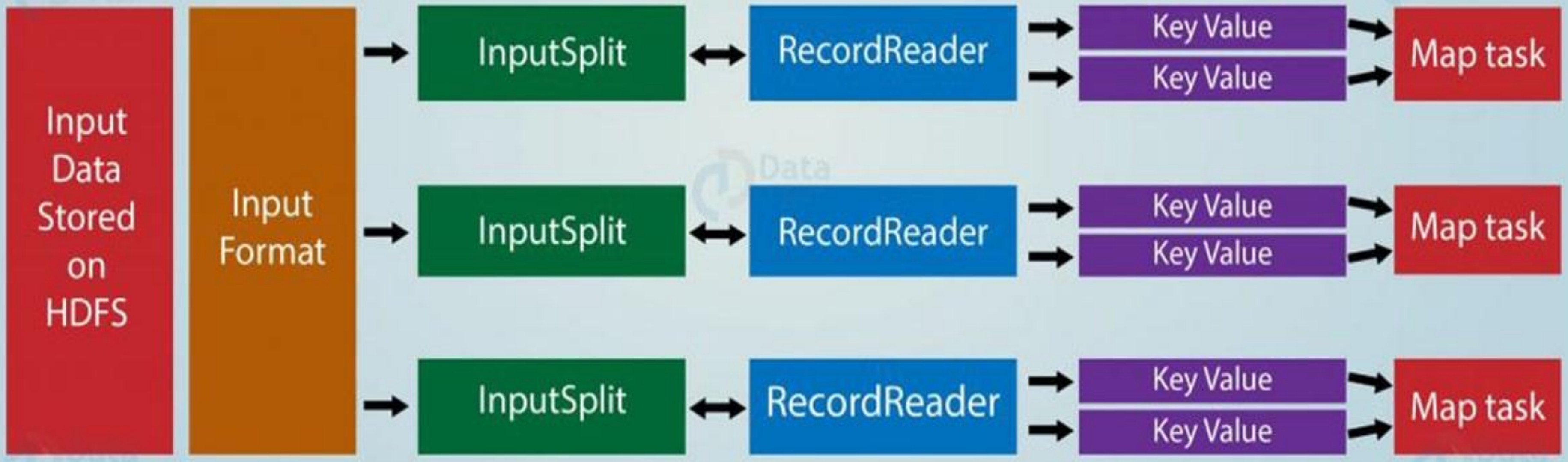
- Mapper
  - RecordReader
  - Map
  - Combiner
  - Partitioner
- Reducer
  - Shuffle
  - Sort
  - Reduce
  - Output Format


Data Flair
MapReduce Job Execution Flow
Intermediate Key value pair
Input data stored on HDFS
Input Format
Inputsplit
RecordReader
Mapper
Combiner
Partitioner
Shuffling and sorting
Reducer
Output Format
Output data stored on HDFS
Input Key-value pair
Substitute Intermediate Key value pair

Key-Value Pairing

- The output types of the Map should match the input types of the Reduce as shown below:
  - **Map:** (K1, V1) -> list (K2, V2)
  - **Reduce:** {(K2, list (V2 }) -> list (K3, V3)

Key Value Pairing in Hadoop MapReduce



Key – Value Pairing

- Generation of a key-value pair in Hadoop depends on the data set and the required output.
- In general, the key-value pair is specified in 4 places:
  - Map input,
  - Map output,
  - Reduce input,
  - Reduce output

Key – Value Pairing

- **a. Map Input**
  - Map-input by default will take the line offset as the key and the content of the line will be the value as Text. By using custom InputFormat we can modify them.
- **b. Map Output**
  - Map basic responsibility is to filter the data and provide the environment for grouping of data based on the key.
  - **Key** – It will be the field/ text/ object on which the data has to be grouped and aggregated on the reducer side.
  - **Value** – It will be the field/ text/ object which is to be handled by each individual reduce method.
- **c. Reduce Input**
  - The output of Map is the input for reduce, so it is same as Map-Output.
- **d. Reduce Output**
  - It depends on the required output.

Input Files

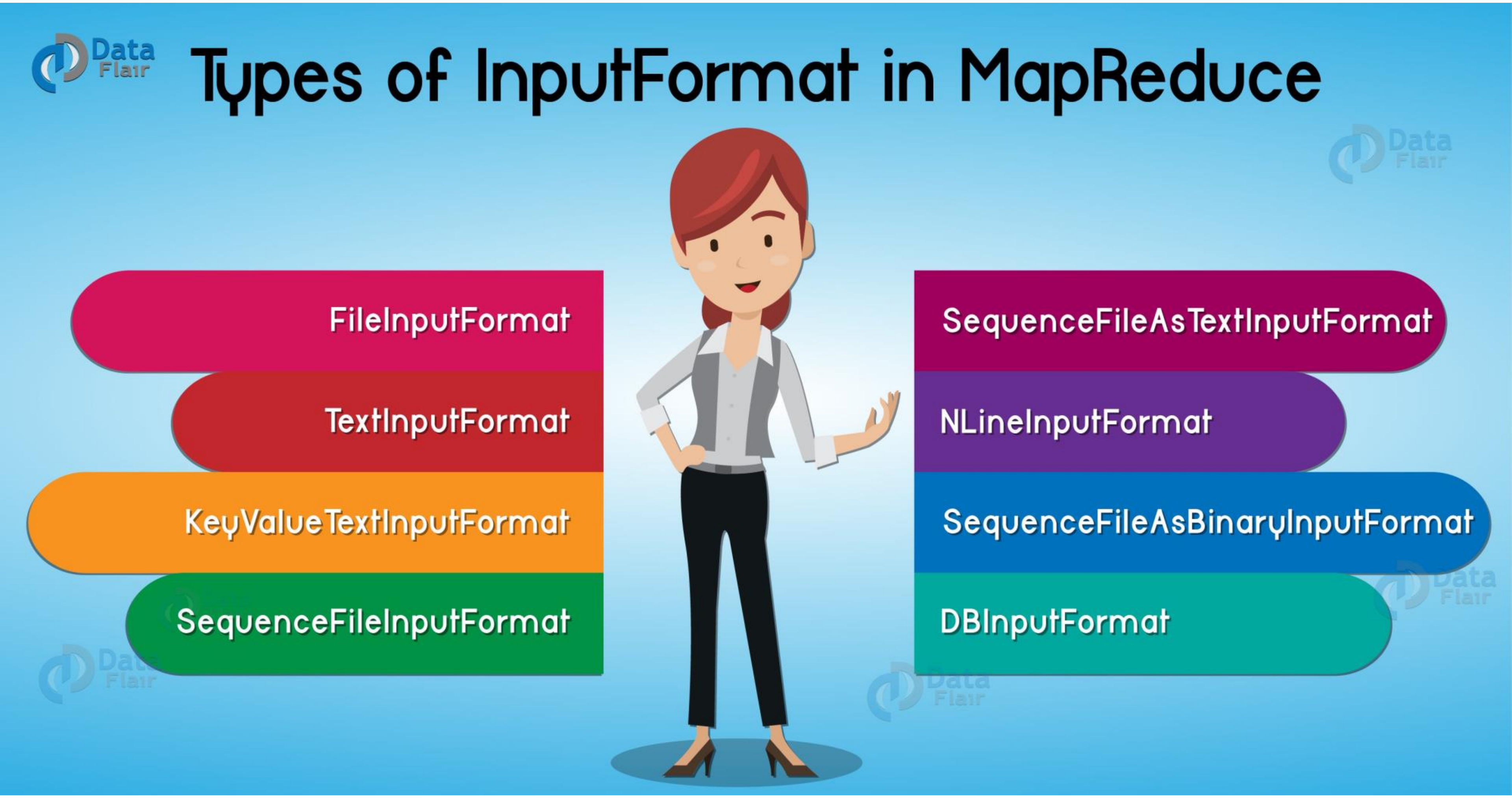
- The data for a MapReduce task is stored in **input files**, and input files typically lives in **HDFS**.
- The format of these files is arbitrary, while line-based log files and binary format can also be used.

InputFormat

- **InputFormat** defines how input files are split and read.
  - e.g. Line by line
- It selects the files or other objects that are used for input.
- InputFormat creates InputSplit.

Functionality of InputFormat

- 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

InputSplits

- Created by InputFormat
- Logically represents the data which will be processed by an individual **Mapper.**
- One map task is created for each split
- So, no. of map tasks = no. of InputSplits.
- The split is divided into records and each record will be processed by the mapper.
- Length of InputSplit is measured in bytes

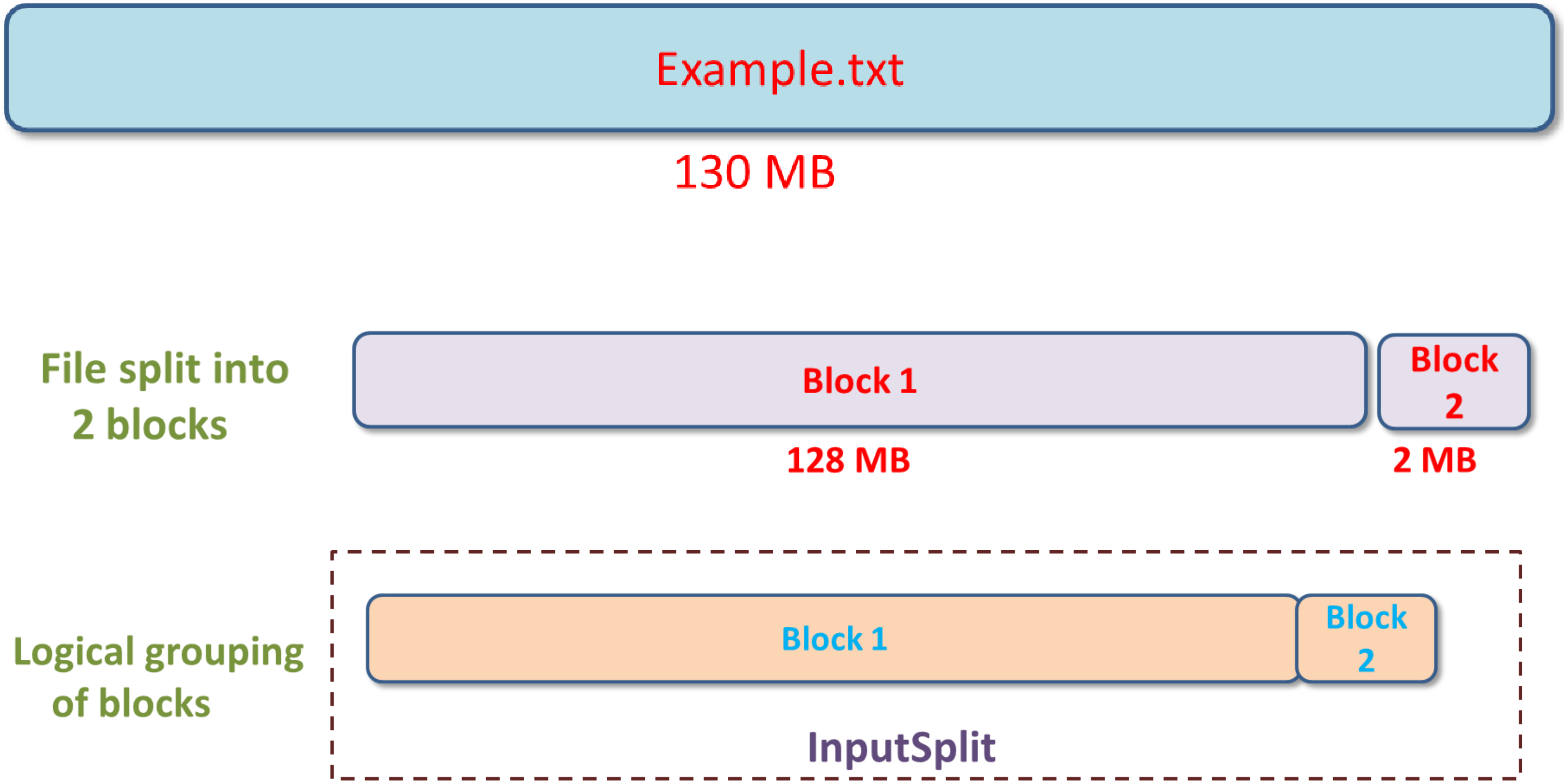
InputSplit vs Block

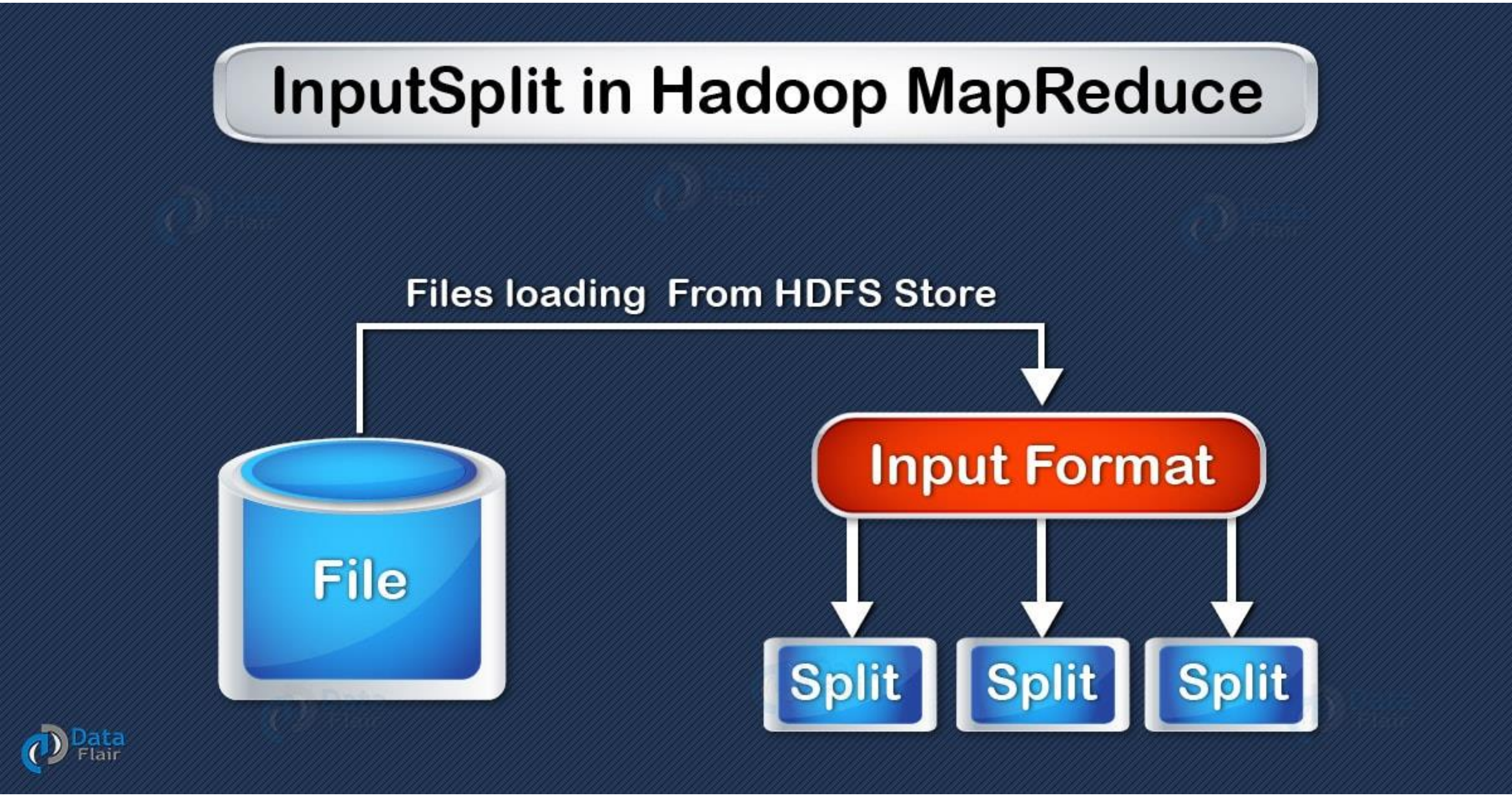
- **Block –** The default size of the HDFS block is 128 MB which we can configure as per our requirement. All blocks of the file are of the same size except the last block, which can be of same size or smaller. The files are split into 128 MB blocks and then stored into Hadoop FileSystem.
- **InputSplit –** By default, split size is approximately equal to block size. InputSplit is user defined and the user can control split size based on the size of data in MapReduce program.

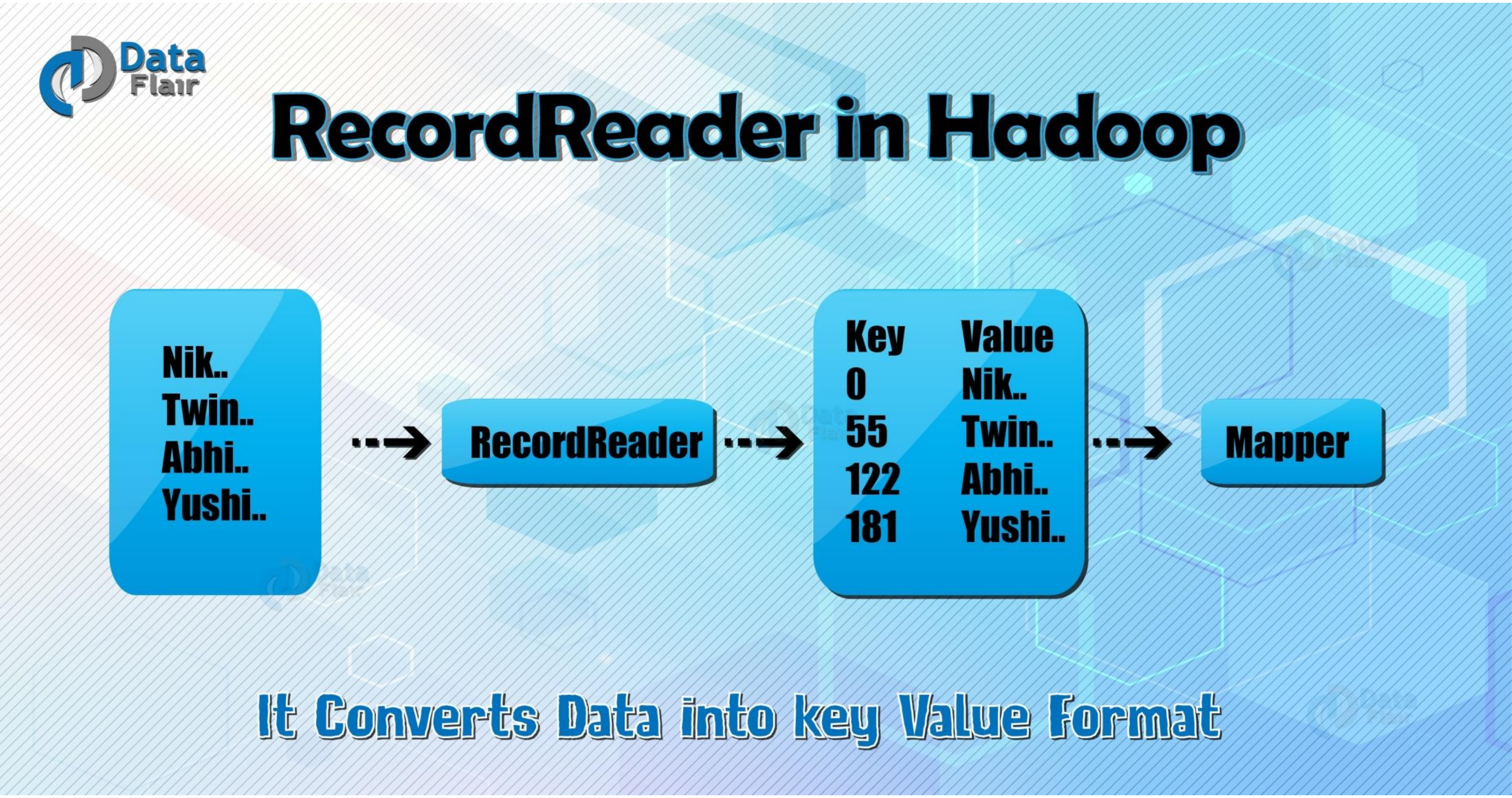
Data Representation

- **Block** – It is the physical representation of data. It contains a minimum amount of data that can be read or write.
- **InputSplit** – It is the logical representation of data present in the block. It is used during data processing in MapReduce program or other processing techniques. InputSplit doesn’t contain actual data, but a reference to the data.

InputSplit vs Block




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


Data Flair
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

RecordReader

- 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.
- 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 for further processing.

Process of conversion

- By calling **'getSplit ()'** the client calculates the splits for the job. Then it sent to the application master. It uses their storage locations to schedule map tasks that will process them on the cluster.
- After that map task passes the split to the **createRecordReader**() method. From that, it obtains RecordReader for the split.
- RecordReader generates record (key-value pair). Then it passes to the map function.

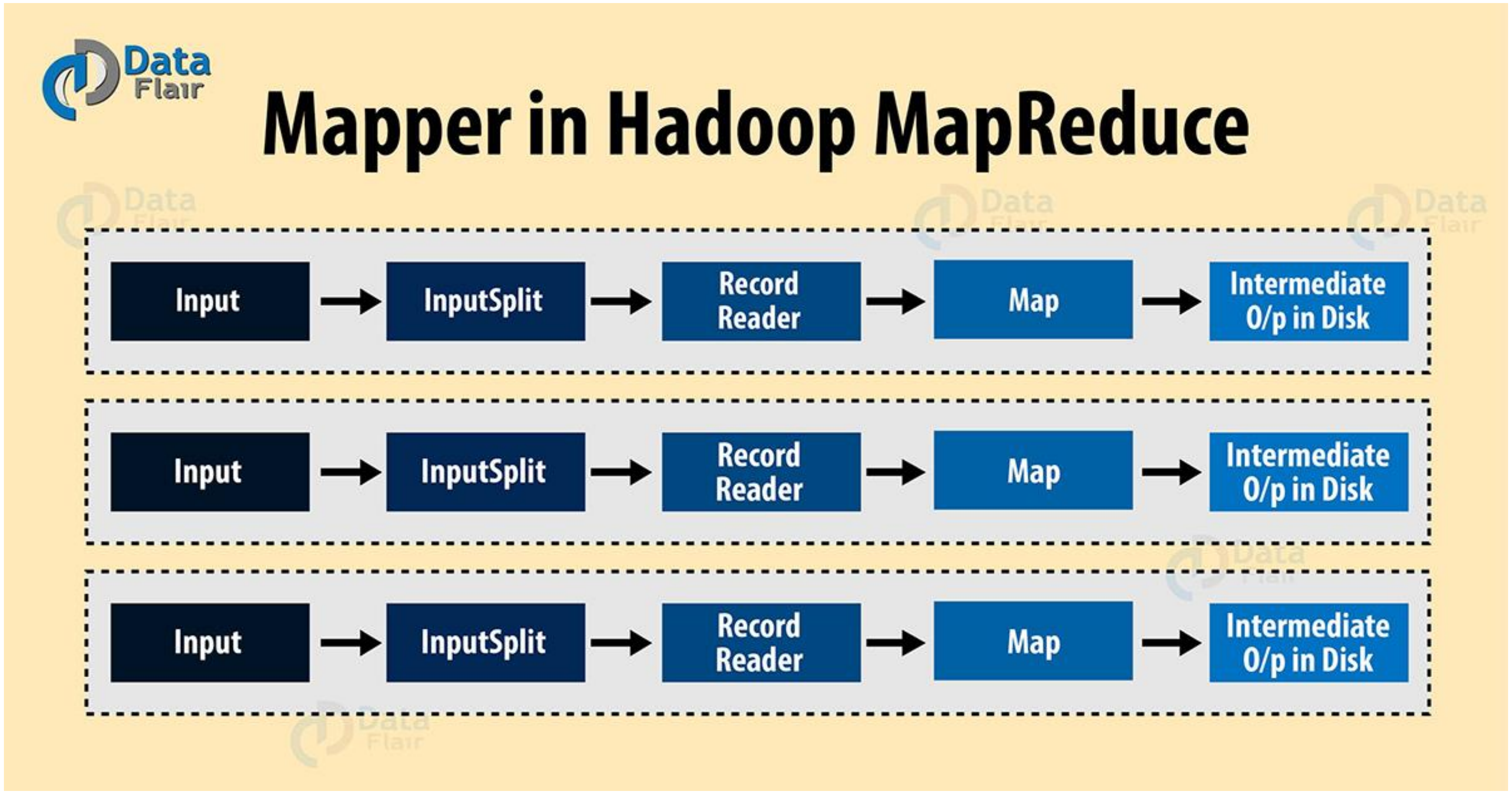
Mapper

- 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. Mappers output is passed to the combiner for further process.

Disk Spill in Mapreduce

- Every map task is allocated some memory.
- Map output is stored in the buffer (default size 100MB but can be modified by io.sort.mb property).
- Disk Spill : When a certain threshold determined by io.sort.spill.percent (which is 80% by default) is reached the data is written on the local disk.
- And after the completion or failure of the job the temporary location used on the local filesystem get's cleared automatically. No manual clean up process is required, it's automatically handled by the framework.

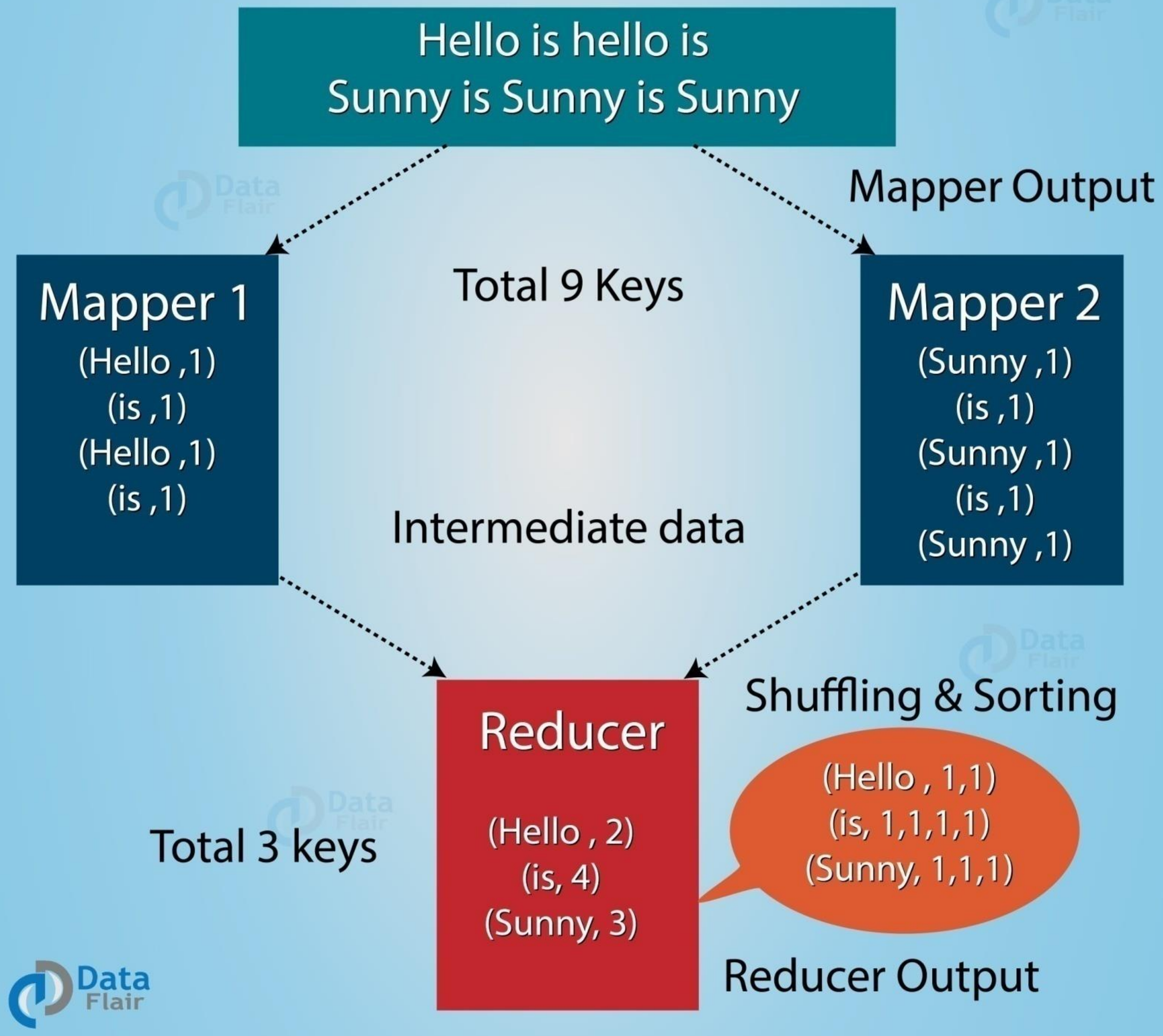
Mapper


Data Flair
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

How many map tasks in Hadoop?

- The total number of blocks of the input files handles the number of map tasks in a program.
- The right level of parallelism is around 10-100 maps/node, although for CPU-light map tasks it has been set up to 300 maps. Since task setup takes some time, it's better if the maps take at least a minute to execute.
- For example, if we have a block size of 128 MB and we expect 10TB of input data, we will have 82,000 maps. Thus, the InputFormat determines the number of maps.
- Hence, **No. of Mapper= {(total data size)/ (input split size)}**
- For example, if data size is 1 TB and InputSplit size is 100 MB then, No. of Mapper= (1000*1000)/100= 10,000

MapReduce program without Combiner



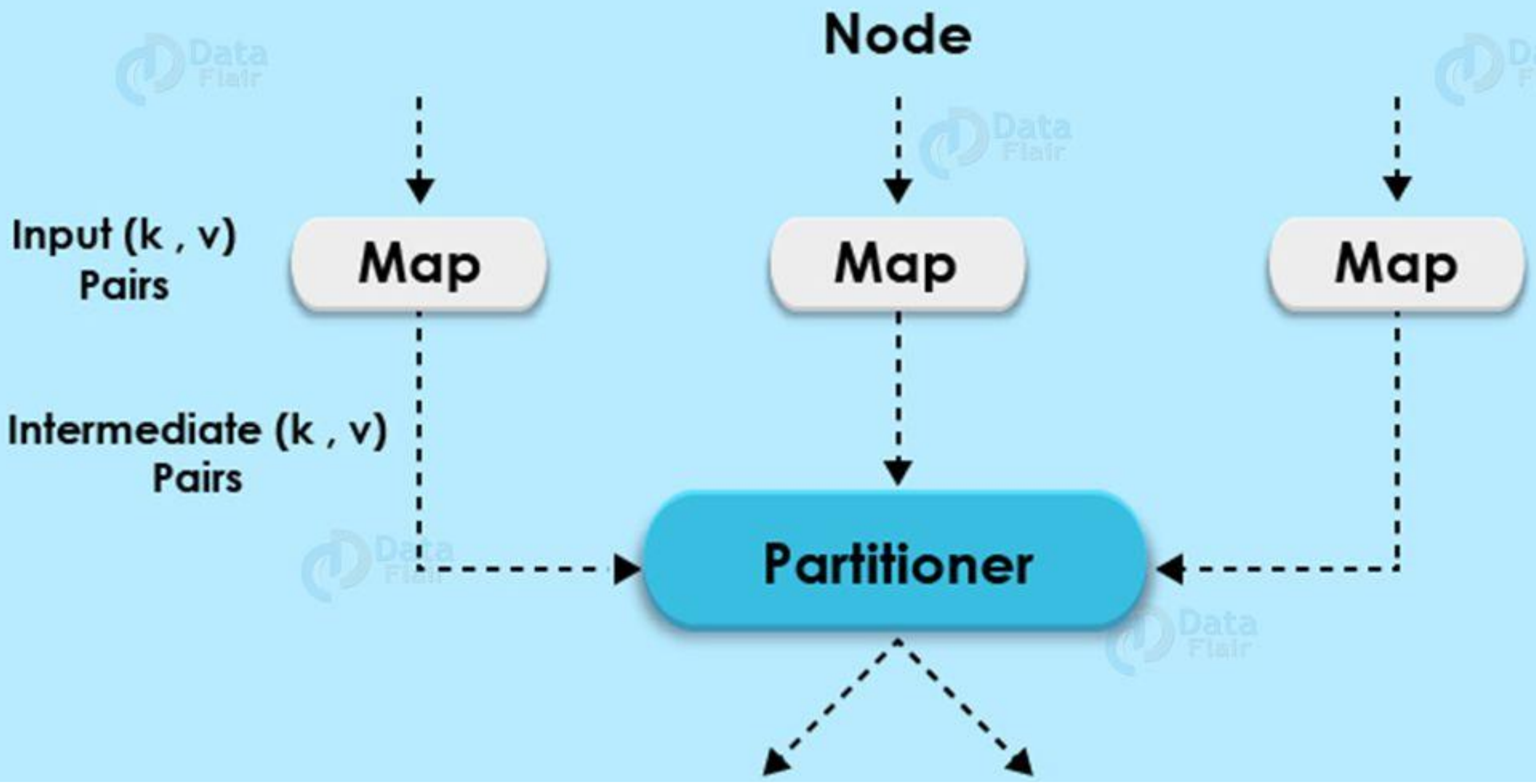
Combiner

- The combiner is also known as ‘Mini-reducer’.
- Hadoop MapReduce Combiner performs local aggregation on the mappers’ output, which helps to minimize the data transfer between mapper and **reducer**.
- Its usage is optional.
- Once the combiner functionality is executed, the output is then passed to the partitioner for further work.

Advantages of Combiner

- Use of combiner reduces the time taken for data transfer between mapper and reducer.
- Combiner improves the overall performance of the reducer.
- It decreases the amount of data that reducer has to process.

Partitioner in Hadoop MapReduce



Partitioner

- Comes into the picture if we are working on more than one reducer (for one reducer partitioner is not used).
- Takes the output from combiners and performs partitioning.
- Partitioning of output takes place on the basis of the key and then sorted. By hash function, key (or a subset of the key) is used to derive the partition.
- According to the key value in MapReduce, each combiner output is partitioned, and a record having the same key value goes into the same partition, and then each partition is sent to a reducer.

How many Partitioner in Hadoop?

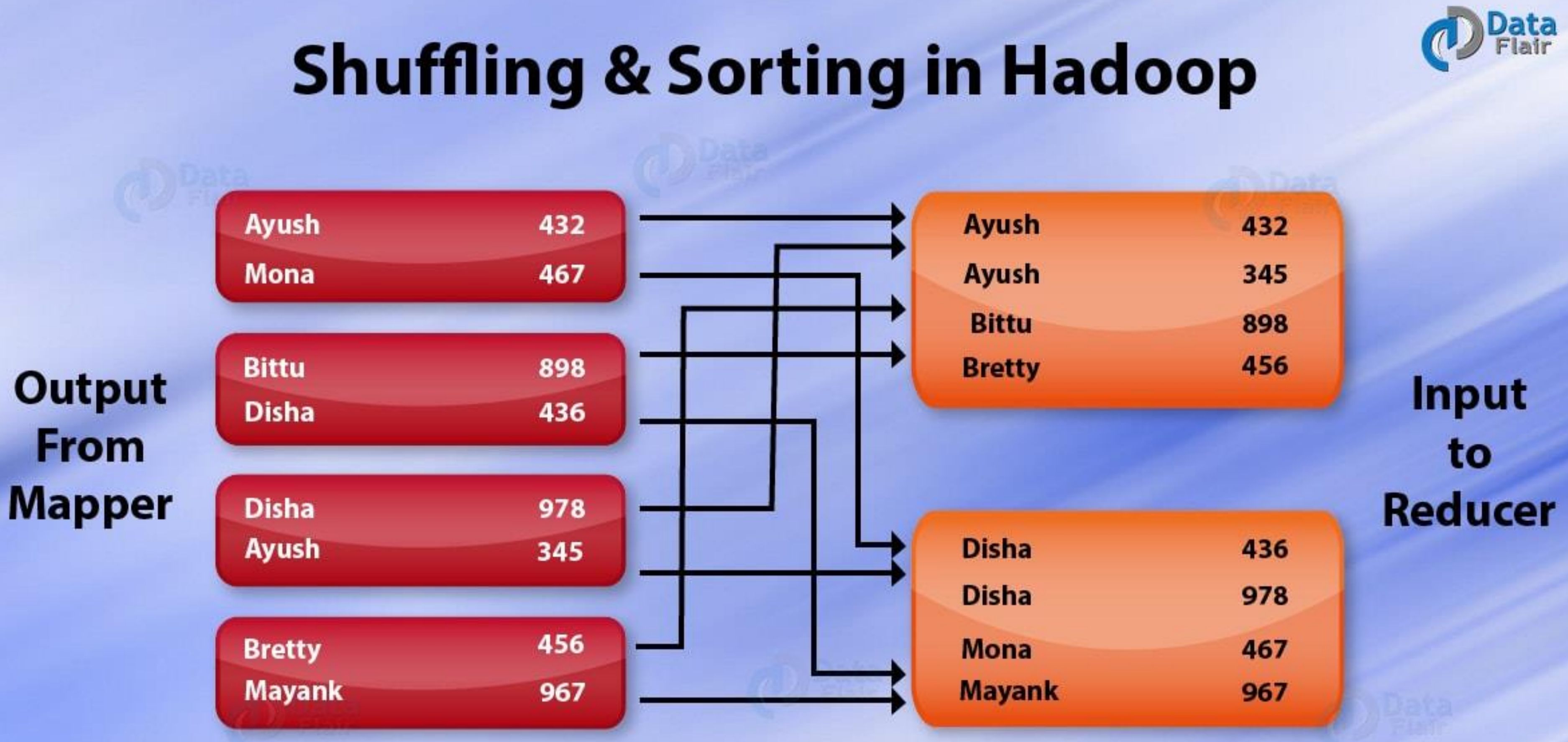
- The total number of Partitioner depends on the number of reducers.
- Hadoop Partitioner divides the data according to the number of reducers.
- It is set by JobConf.setNumReduceTasks() method.
- The important thing to notice is that the framework creates partitioner only when there are many reducers.

Partitioning

- Partitioning allows even distribution of the map output over the reducer.
- e.g. h(k) = k%2 divides data into two partitions
- Sometimes default partitioning may result in poor partitioning of data, if in data input in MapReduce job one key appears more than any other key.
- In such case, to send data to the partition we use two mechanisms which are as follows:
  - The key appearing more number of times will be sent to one partition.
  - All the other key will be sent to partitions on the basis of their hashCode().
- If there is poor partitioning , the solution is to create Custom Partitioning.

Custom Partitioner Demonstration

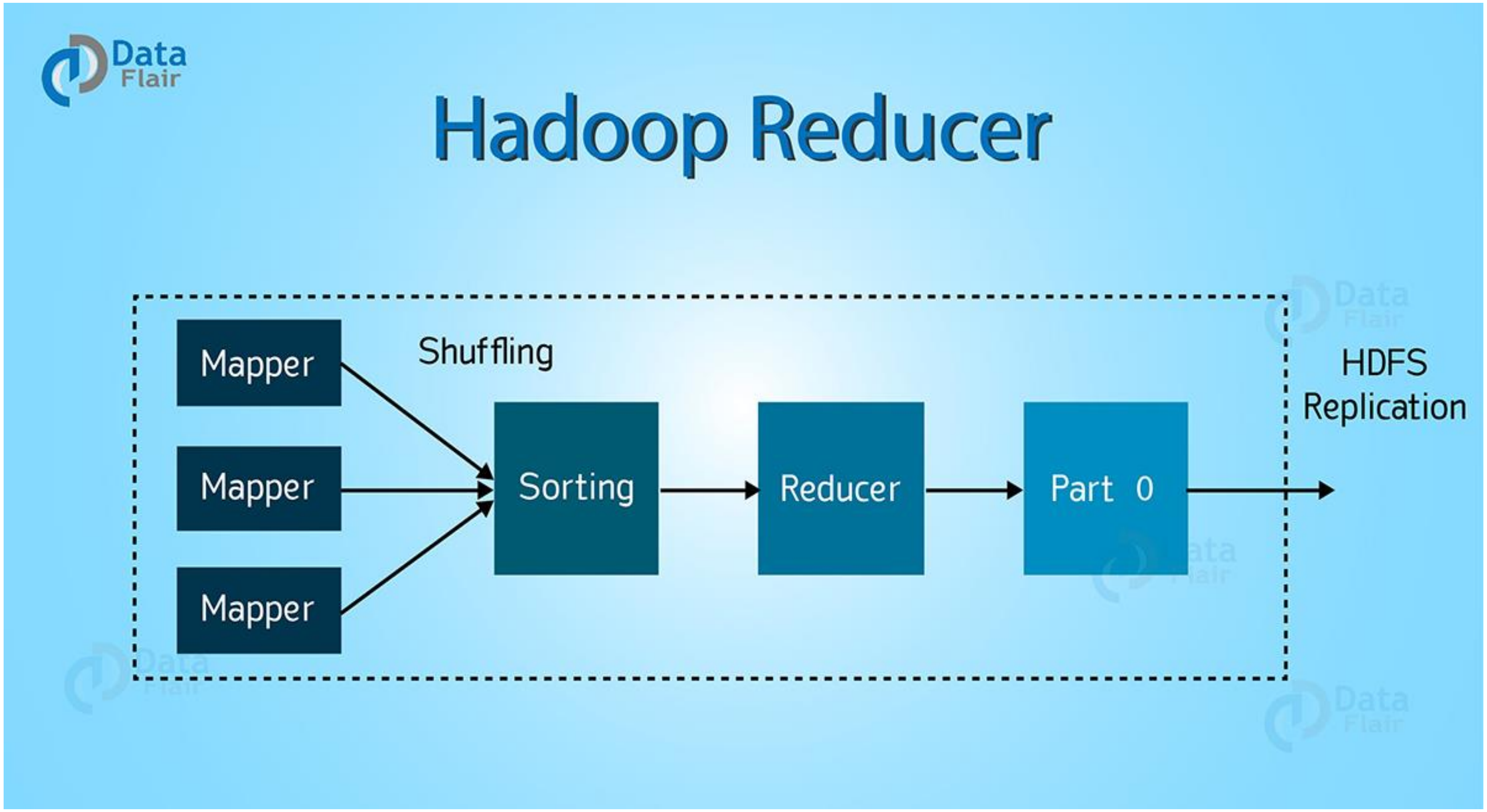
Shuffling & Sorting in Hadoop



Shuffling and Sorting

- Shuffling and Sorting in Hadoop occurs simultaneously.
- The process of transferring data from the mappers to reducers is shuffling. It is also the process by which the system performs the sort. Then it transfers the map output to the reducer as input.
- MapReduce Framework automatically sort the keys generated by the mapper. Thus, before starting of reducer, all intermediate key-value pairs get sorted by key and not by value. Input from different mappers is again sorted based on the similar keys in different Mappers.
- If we want to sort reducer values, then we use a secondary sorting technique.

Reducer


Data Flair
Hadoop Reducer
Mapper
Mapper
Mapper
Shuffling
Sorting
Reducer
Part 0
HDFS Replication

Reducer

- It takes the set of intermediate key-value pairs produced by the mappers as the input and then runs a reducer function on each of them to generate the output.
- The output of the reducer is the final output, which is stored in HDFS.

Number of Reducers

- With the help of *Job.setNumreduceTasks(int)* the user set the number of reducers for the job. The right number of reducers are set by the formula, which is 0.95 or 1.75 multiplied by (<no. of nodes> * <no. of the maximum container per node>).
- With 0.95, all reducers immediately launch and start transferring map outputs as the maps finish. With 1.75, the first round of reducers is finished by the faster nodes and second wave of reducers is launched doing a much better job of load balancing.

Number of Reducers

- Assume, 100 reduce slots available in your cluster.
- With a load factor of 0.95 all the 95 reduce tasks will start at the same time, since there are enough reduce slots available for all the tasks.
  - This means that no tasks will be waiting in the queue, until one of the rest finishes.
  - Recommended when the reduce tasks are "small", i.e., finish relatively fast, or they all require the same time, more or less.
- With a load factor of 1.75, 100 reduce tasks will start at the same time, as many as the reduce slots available, and the 75 rest will be waiting in the queue, until a reduce slot becomes available.
  - Offers better load balancing, since if some tasks are "heavier" than others, i.e., require more time, then they will not be the bottleneck of the job, since the other reduce slots, instead of finishing their tasks and waiting, will now be executing the tasks in the queue.
  - Also lightens the load of each reduce task, since the data of the map output is spread to more tasks.

With the increase of the number of reducers

- Framework overhead increases.
- Load balancing increases.
- Cost of failures decreases.

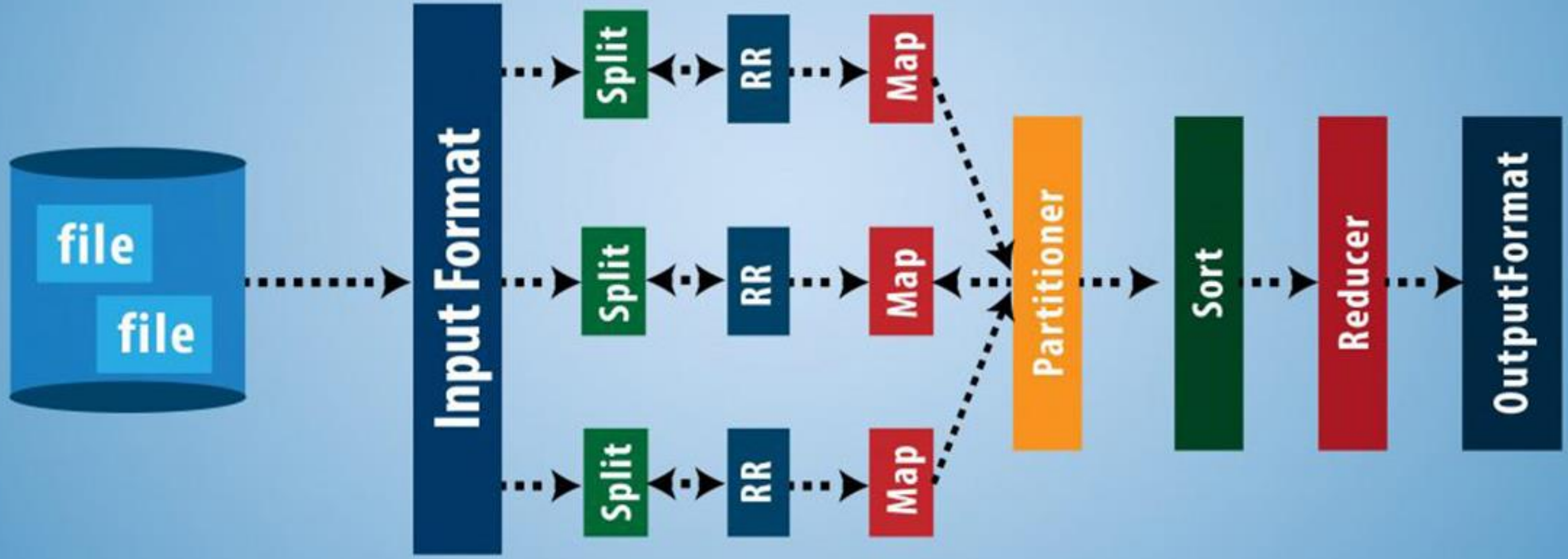
RecordWriter

- It writes these output key-value pairs from the Reducer phase to the output files.

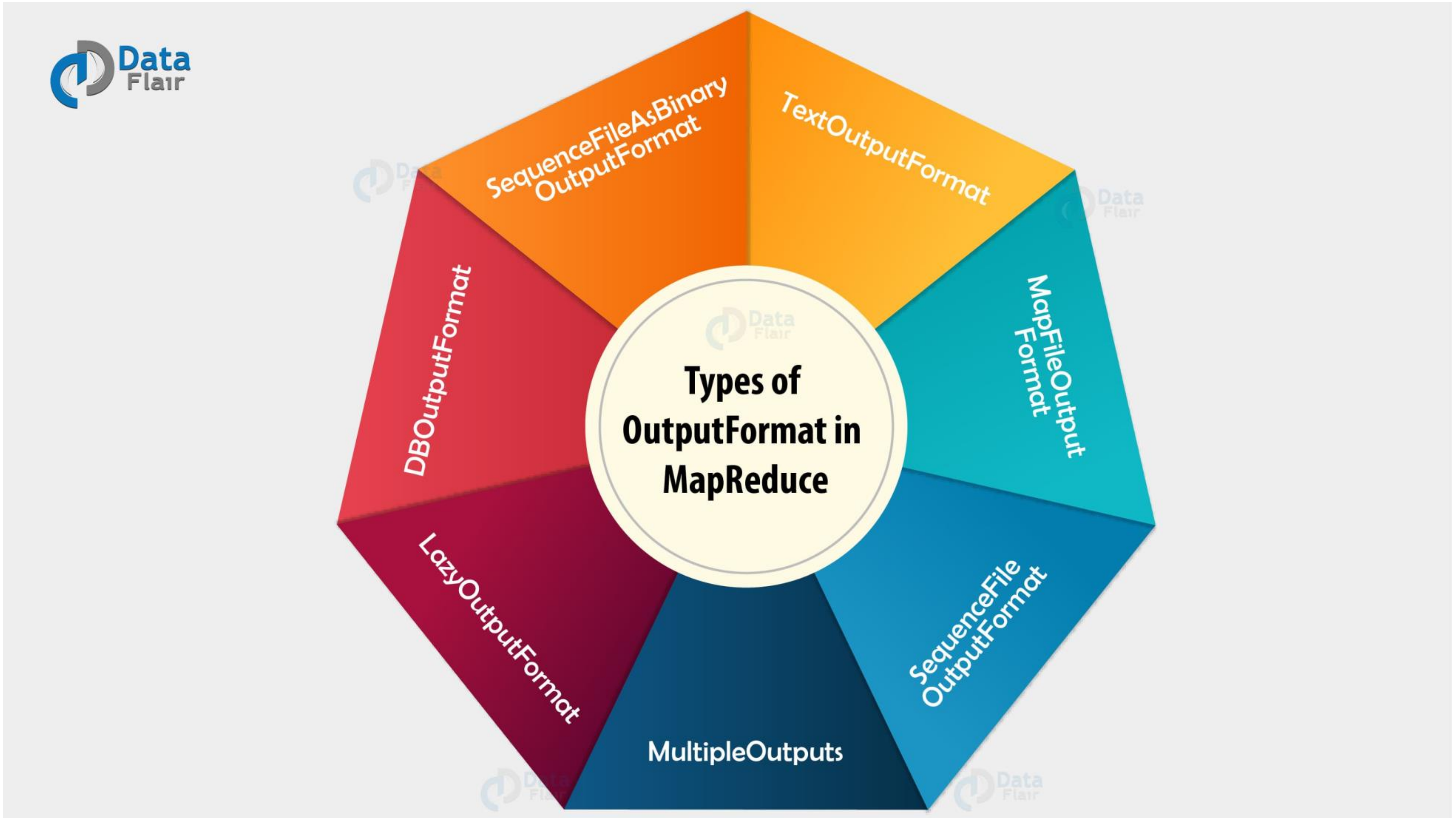
OutputFormat

- The way these output key-value pairs are written in output files by RecordWriter is determined by the OutputFormat.
- OutputFormat instances provided by the Hadoop are used to write files in HDFS or on the local disk. Thus the final output of reducer is written on HDFS by OutputFormat instances.

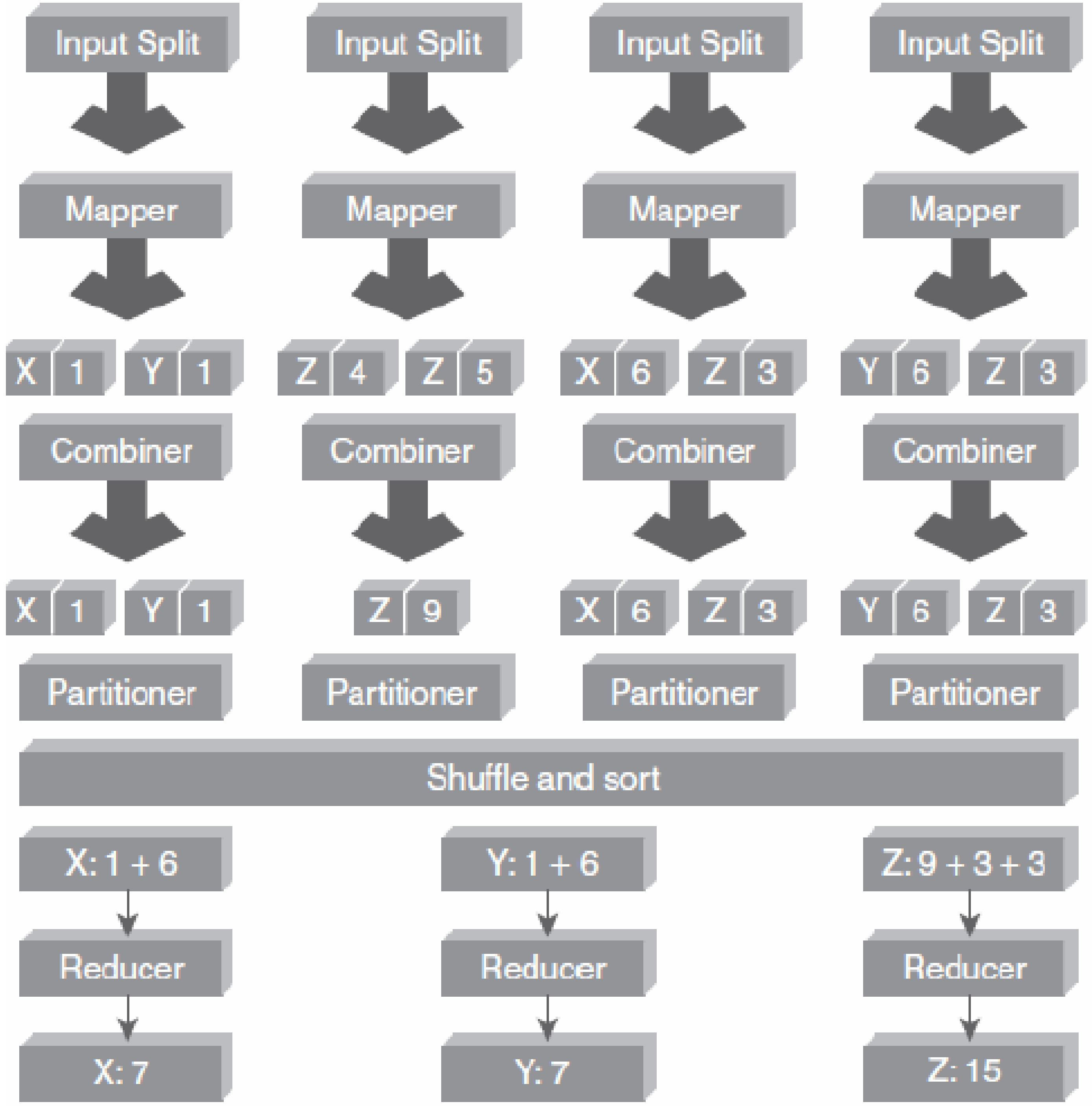
OutputFormat in Hadoop MapReduce



Types of Hadoop Output Formats



Types of OutputFormat in MapReduce
TextOutputFormat
MapFileOutput Format
SequenceFile OutputFormat
MultipleOutputs
LazyOutputFormat
DBOutputFormat
SequenceFileAsBinary OutputFormat


Input Split
Input Split
Input Split
Input Split
Mapper
Mapper
Mapper
Mapper
X 1
Y 1
Z 4
Z 5
X 6
Z 3
Y 6
Z 3
Combiner
Combiner
Combiner
Combiner
X 1
Y 1
Z 9
X 6
Z 3
Y 6
Z 3
Partitioner
Partitioner
Partitioner
Partitioner
Shuffle and sort
X: 1 + 6
Y: 1 + 6
Z: 9 + 3 + 3
Reducer
Reducer
Reducer
X: 7
Y: 7
Z: 15

References

- www.data-flair.training
- https://techvidvan.com/tutorials/mapreduce-job-execution-flow/
- https://data-flair.training/forums/topic/in-map-reduce-why-map-write-output-to-local-disk-instead-of-hdfs/
- https://stackoverflow.com/questions/21980110/what-is-ideal-number-of-reducers-on-hadoop

Thank you!