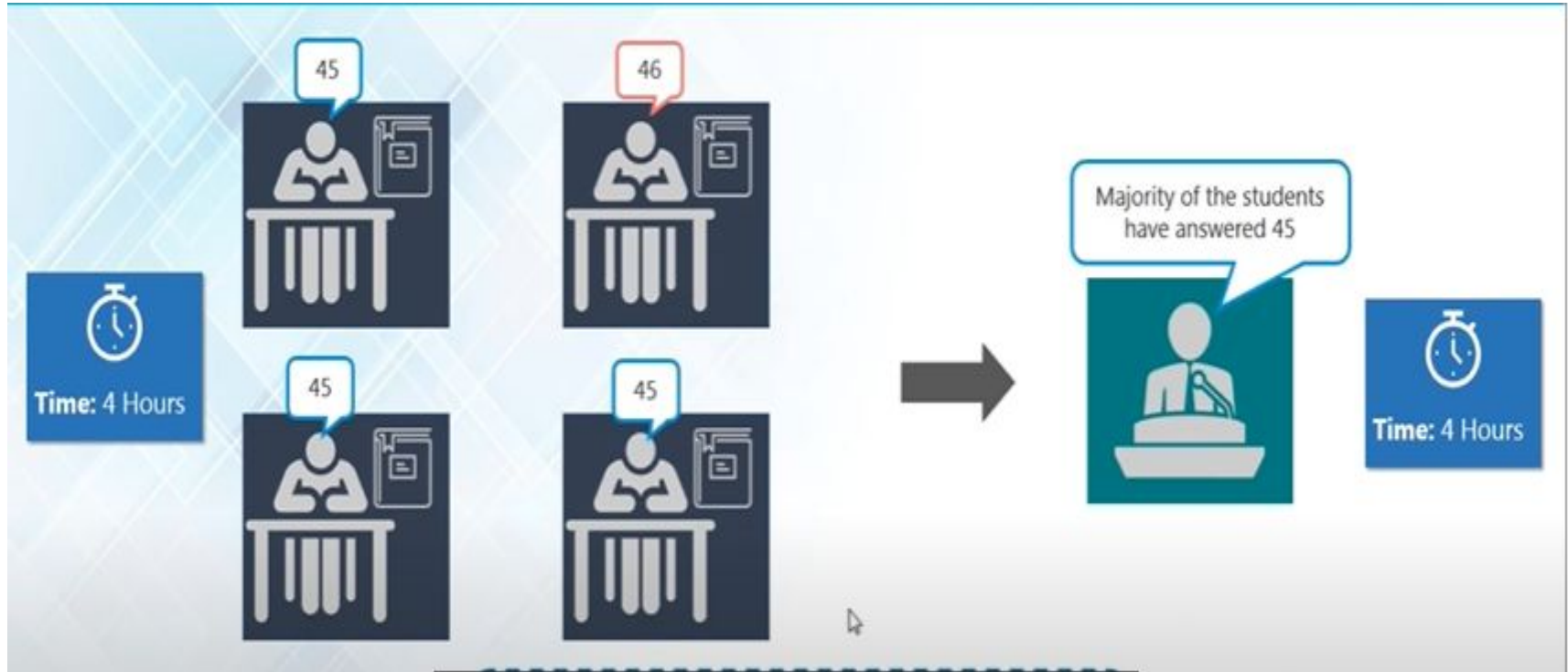


MapReduce

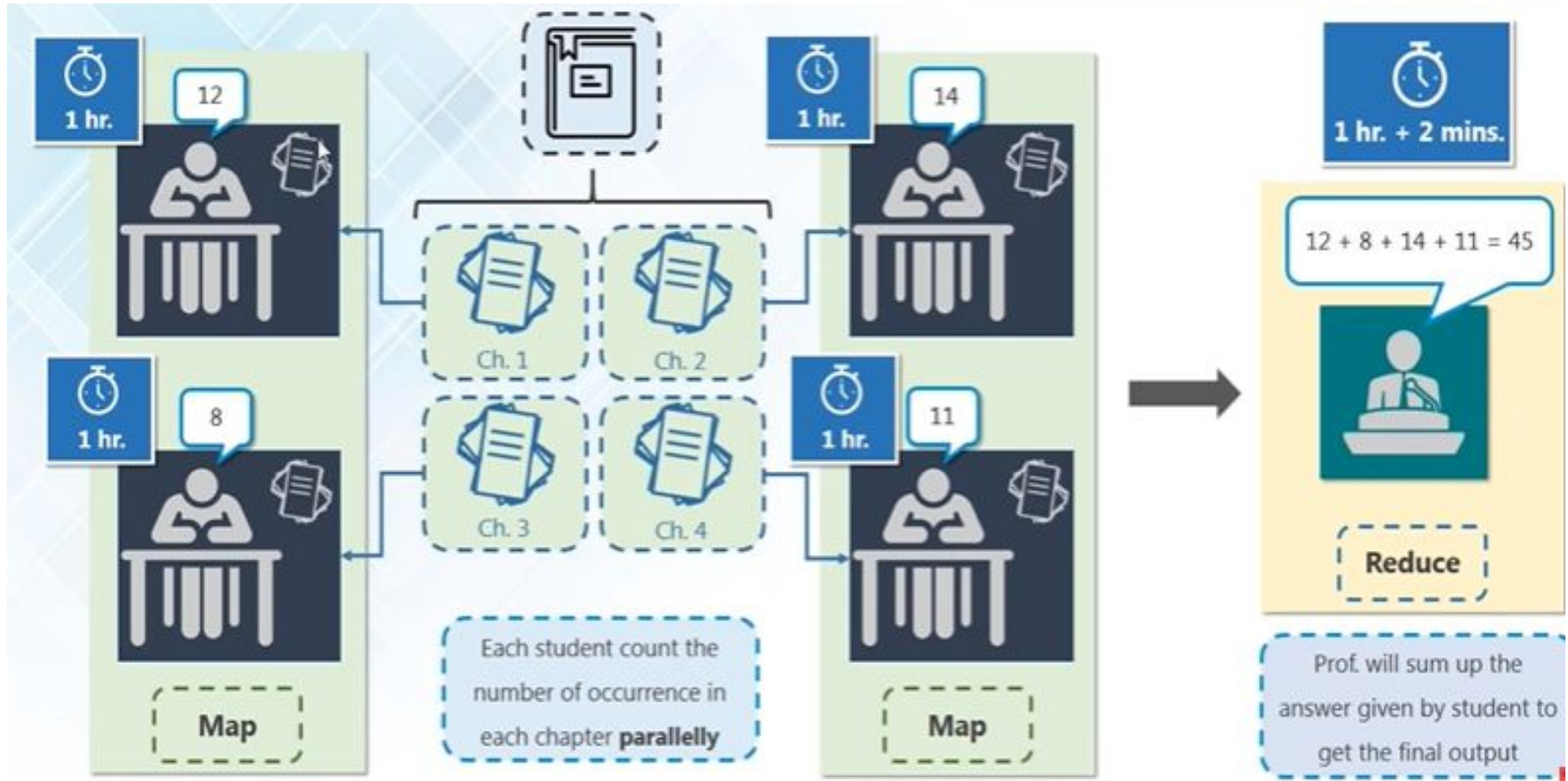
Prepared By : Prof.Shital Pathar

Story of MapReduce



Each student has to count the occurrence of word **data** in the book.

Story of MapReduce



Each student count the no of occurrence in each chapter **parallelly**.

Prof. will sum up the answer given by students to get final **output**.

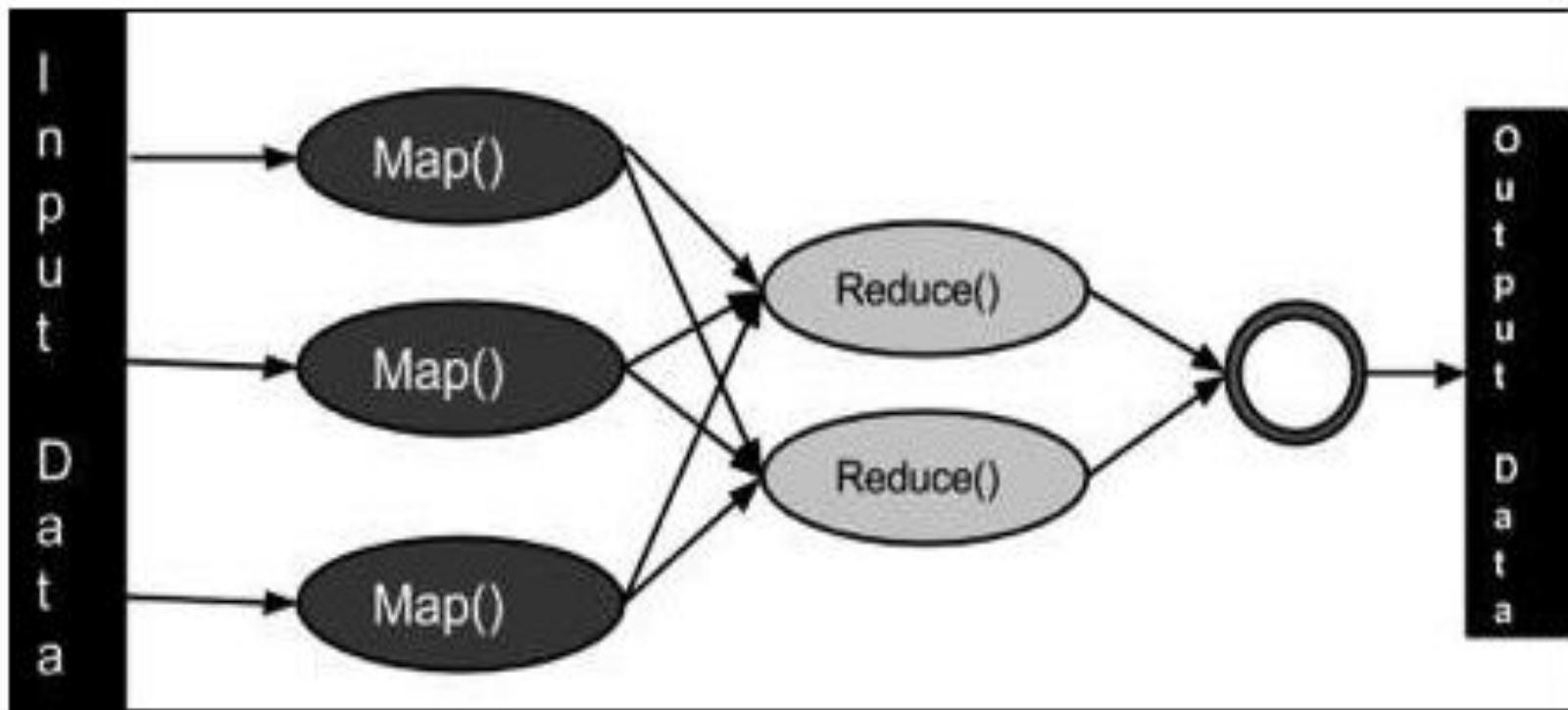
Map Reduce : Data processing using Programming

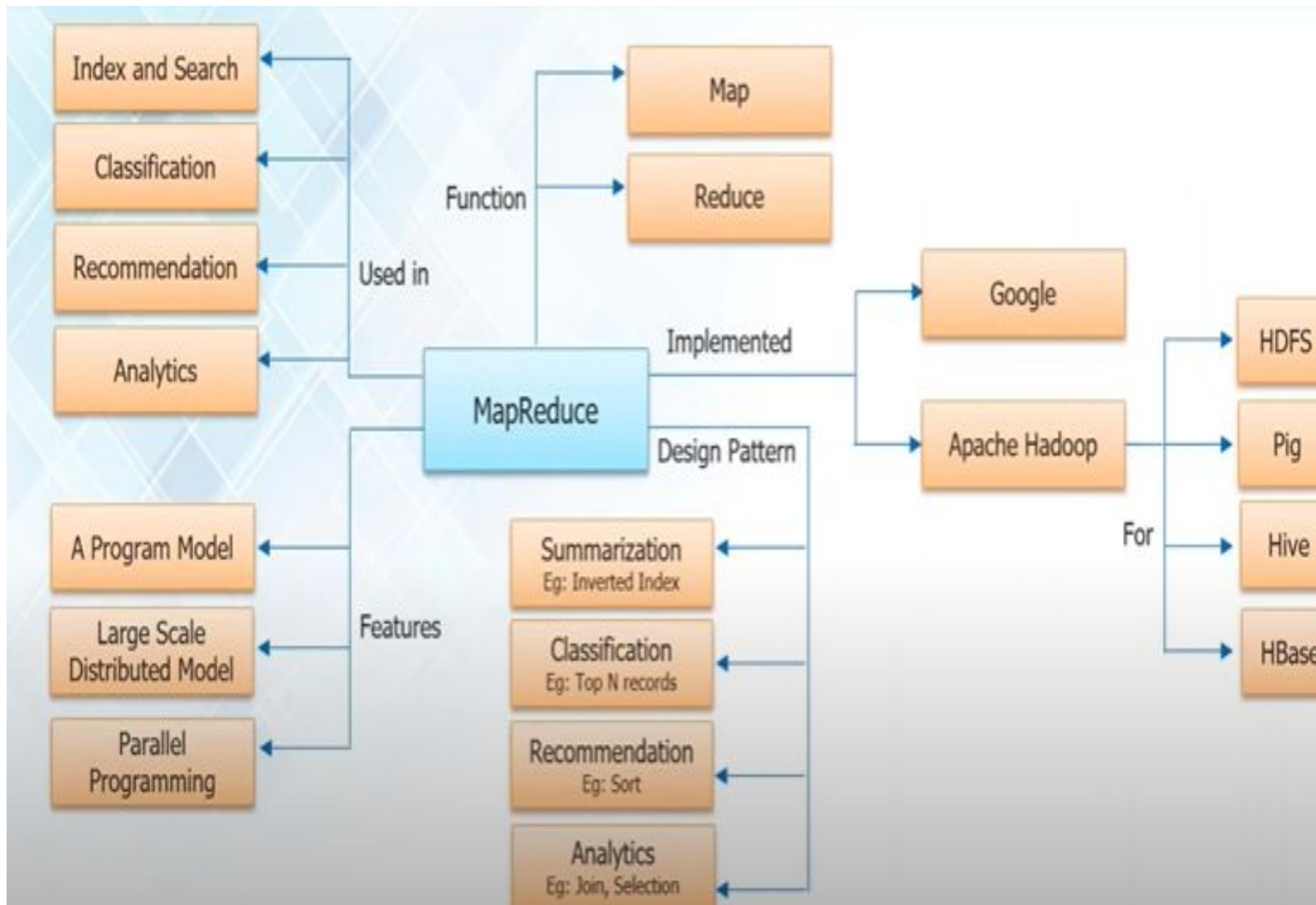
- Hadoop Mapreduce is the processing component of apache hadoop.
- it processes data parallely in distributed environment.



What is MapReduce?

- Hadoop MapReduce is a programming framework for easily writing applications which **process** vast amounts of data (multi-terabyte data-sets) in-**parallel** on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.
- Map reduce allows us to perform, **distributed and parallel processing on large data sets.**





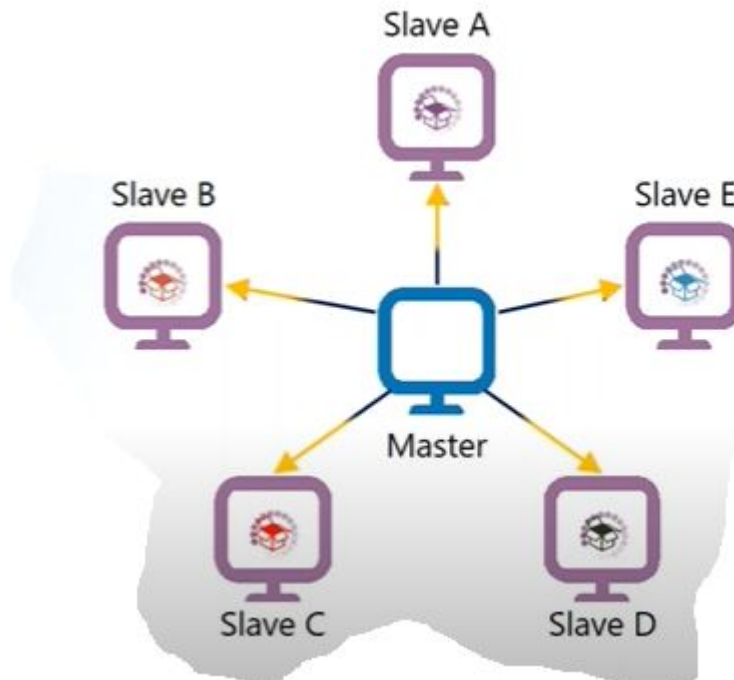
Advantages of Mapreduce

→ Parallel processing of data

- ◆ Processing becomes faster

→ Data locality

- ◆ Moving **data to processing** is very costly
- ◆ In map reduce we move **processing to data**



What is MapReduce?

- A MapReduce job usually splits the input data-set into independent chunks which are processed by the map tasks in a completely parallel manner.
- The framework sorts the outputs of the maps, which are then input to the reduce tasks. Typically both the input and the output of the job are stored in a file-system.
- The MapReduce algorithm contains two important tasks, namely Map and Reduce.
- The **Map task** takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key-value pairs).
- The **Reduce task** takes the output from the Map as an input and combines those data tuples (key-value pairs) into a smaller set of tuples. The reduce task is always performed after the map job.
- In between map and reduce there is a small phase called as **shuffle and sort**.
- The framework takes care of **scheduling tasks, monitoring them and re-executes** the failed tasks.

Inputs and Outputs

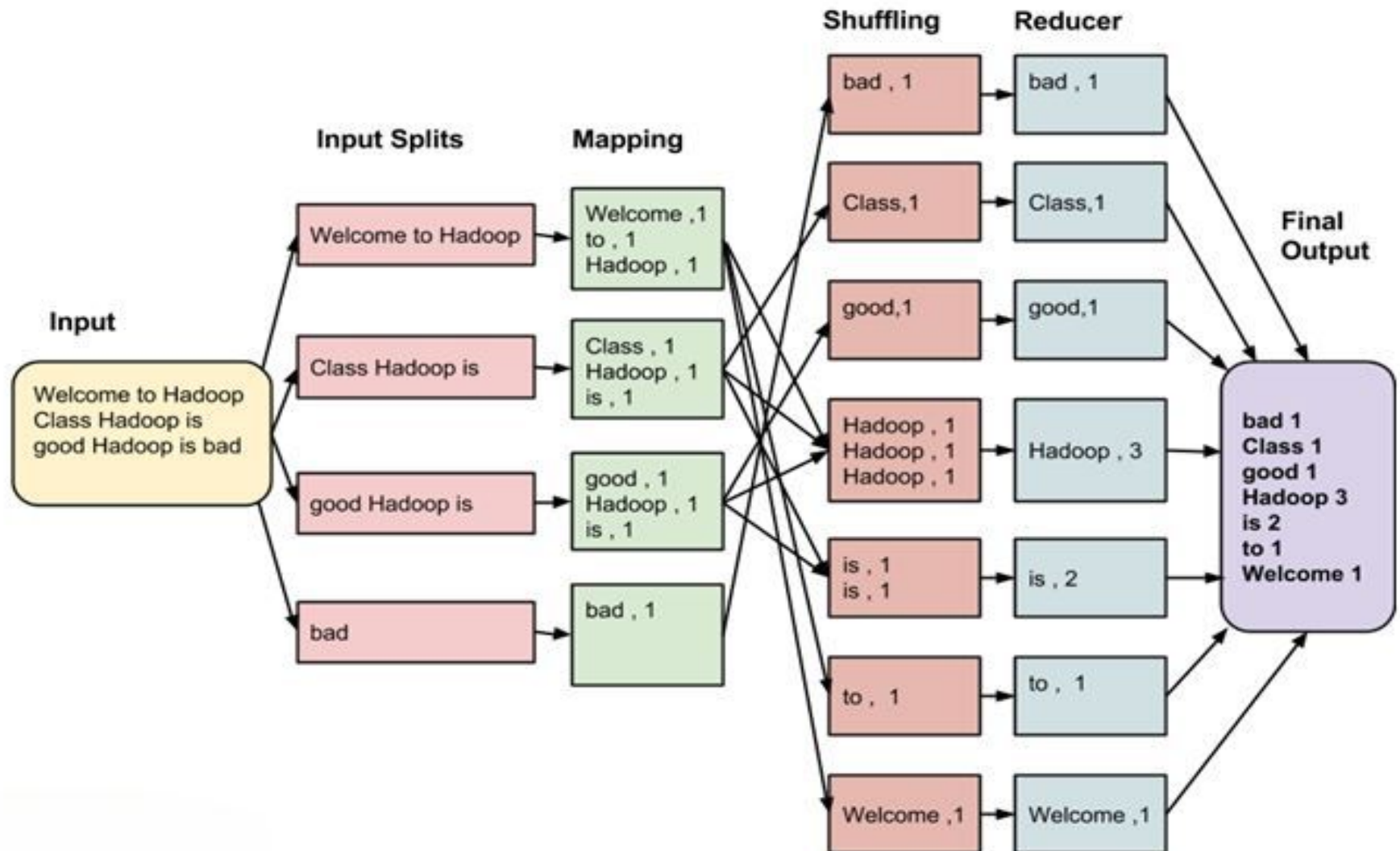
The MapReduce framework operates exclusively on $\langle \text{key}, \text{value} \rangle$ pairs, that is, the framework views the input to the job as a set of $\langle \text{key}, \text{value} \rangle$ pairs and produces a set of $\langle \text{key}, \text{value} \rangle$ pairs as the output of the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

- Input : $\langle \text{key}, \text{value} \rangle$ pairs
- Output : $\langle \text{key}, \text{value} \rangle$ pairs

Input and Output types of a MapReduce job:

(input) $\langle k1, v1 \rangle \rightarrow$ **map** $\rightarrow \langle k2, v2 \rangle \rightarrow$ **combine** $\rightarrow \langle k2, v2 \rangle \rightarrow$
reduce $\rightarrow \langle k3, v3 \rangle$ (output)

MapReduce Word Count Example



MapReduce Wordcount Example

There are 3 major parts in Map reduce code.

1. **Mapper code**

This code defines of Mapper logic how the map tasks will process the data to produce the key value pair to be aggregated.

2. **Reducer Code**

This code defines the reducer logic,the reducer combines the intermediate results in order to produce the output.

3. **Driver Code**

this code defines all configuration required to perform the Map reduce job.

Packages and classes

```
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;
```

hadoop-common.jar(package)

```
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;
```

**hadoop-mapreduce-client-core
.jar(package)**

Mapper Class

Name of the Mapper class
which inherits the Mapper class

public static class TokenizerMapper extends Mapper
<Object, Text, Text, IntWritable>

Mapper class takes 4 arguments
<KEYIN,VALUEIN,KEYOUT,VALUEOUT>

Reducer Class

Name of the Reducer class which inherits the Reducer class

```
public static class IntSumReducer extends Reducer  
<Text,IntWritable,Text,IntWritable>
```

Reducer class takes 4 arguments

<KEYIN,VALUEIN,KEYOUT,VALUEOUT>

Step-1 Create Input File(input.txt)


what is hadoop
class hadoop is
good hadoop is bad

Step-2

Split the Input into lines

Mapper Code

Mapper key Byte Offset, Mapper value Input Type, Mapper key Output type, Mapper Value Output



```
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);
        }
    }
}
```

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

Step-3 Map Job

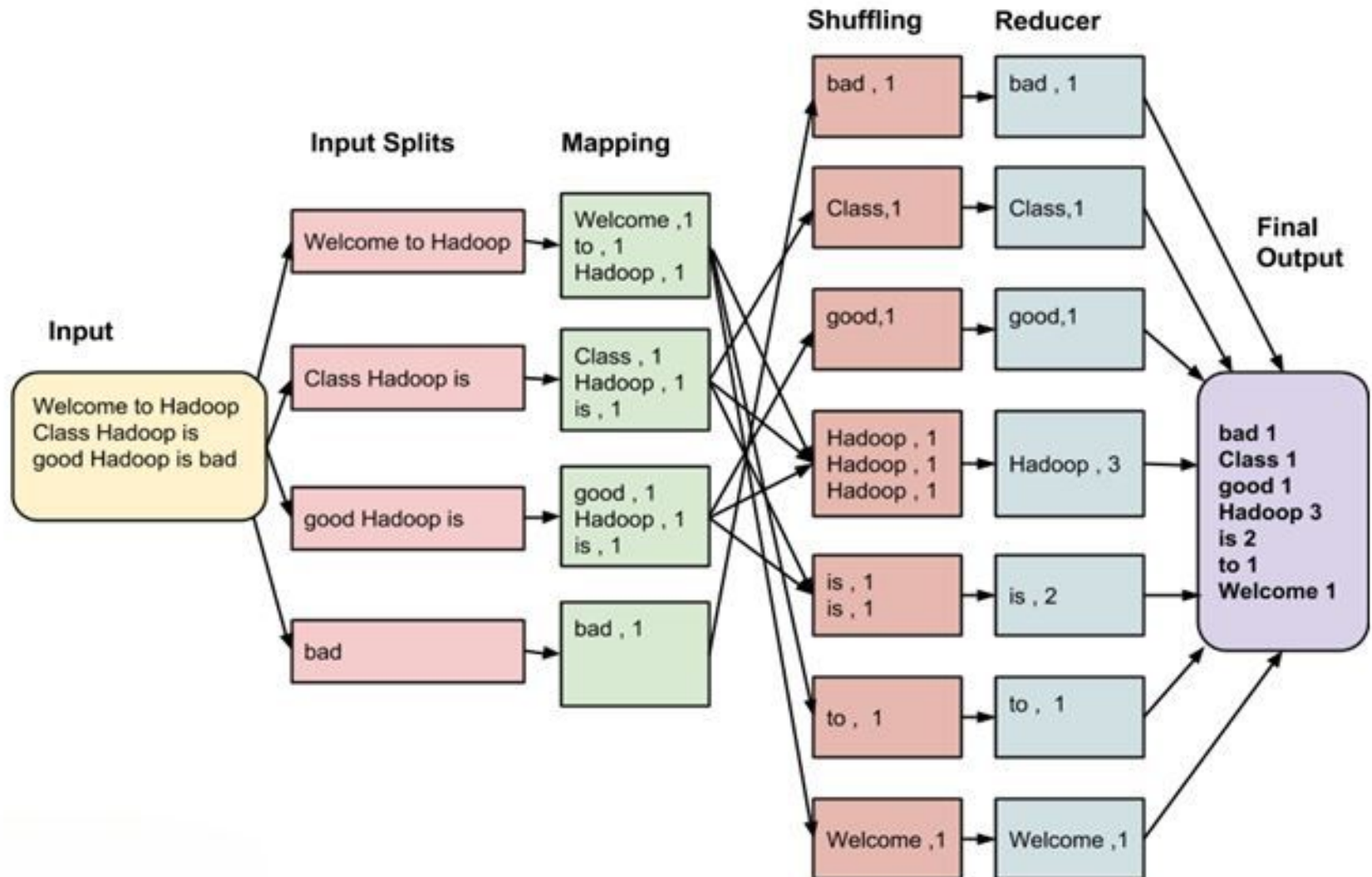
Input(Byte Offset, Text)=
<14,What is hadoop>

Output(Text, IntWritable)
<what,1>
<is,1>
<hadoop,1>

Context object:

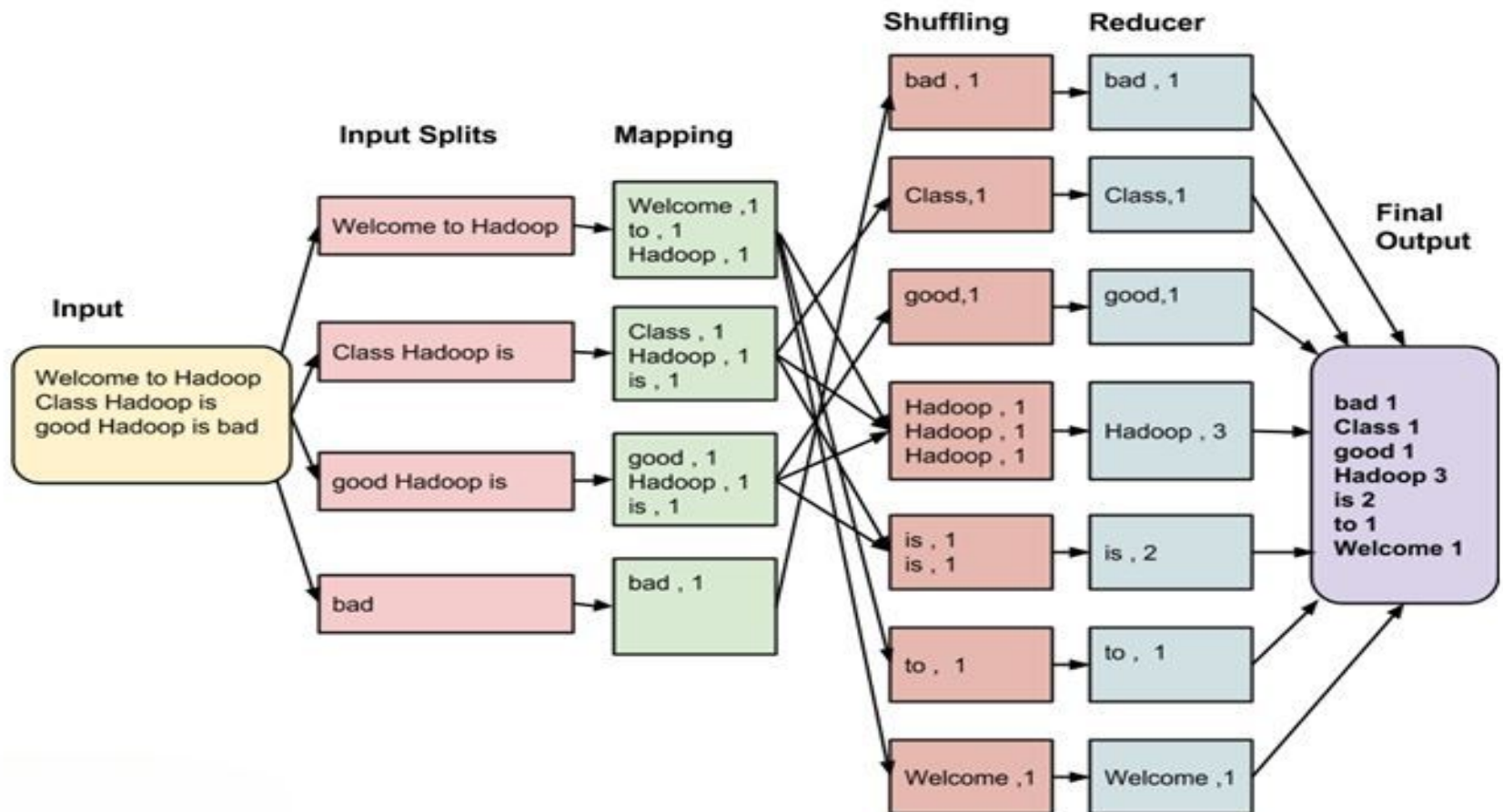
allows the Mapper/Reducer to interact with the rest of the Hadoop system. It includes configuration data for the job as well as interfaces which allow it to emit output.

Step-4 Sorting and shuffling



Step-5 Reduce Job

Input(Text key, Iterable<IntWritable>)
output (text,IntWritable)

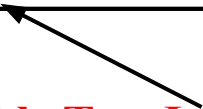


Reducer Code

Reducer key Input type, reducer value Input Type, Reducer key Output type, Reducer Value Output Type

```
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);
    }
}
```



Driver Code

In the driver class, we set the configuration for our Mapreduce Job to execute in hadoop.

```
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);  
}
```

- Specifies the name of the **job**, the name of **Input/output** of the **mapper** and **reducer**.
- Specify the **names of the mapper and reducer** classes.
- **Path** of the **Input** and **Output** Folder.
- main method is the entry point for the driver

Step for Map reduce Task

Step -1: Start hadoop entities.

start-dfs.sh

start-yarn.sh

Step-2 :Run jps and verify as below:

```
[hadoop@hadoop-clone ~]$ jps
15826 Jps
10514 DataNode
11159 NodeManager
11015 ResourceManager
10764 SecondaryNameNode
10380 NameNode
```

Step 3 : Compile WordCount.java and create a jar:(go to the directory where you have wordcount.java file)

Assuming environment variables are set appropriately:

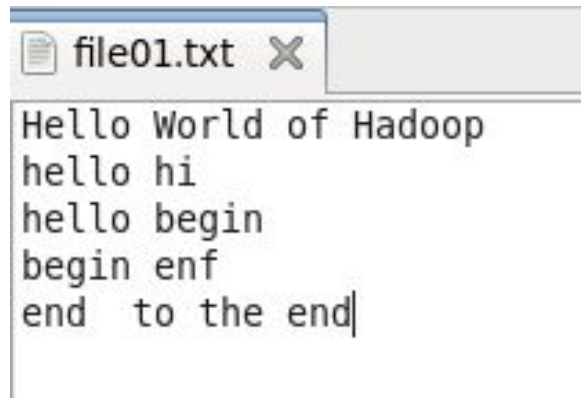
```
$ hadoop com.sun.tools.javac.Main WordCount.java
```

```
$ jar cf wc.jar WordCount*.class
```


Step 4: Store the input data file in HDFS

Step 4.1 : create a text file in local machine (file01.txt)

Path of the file on local machine - /home/hadoop/Desktop/file01.txt

A screenshot of a text editor window titled 'file01.txt'. The window contains the following text:

```
Hello World of Hadoop  
hello hi  
hello begin  
begin enf  
end to the end|
```

Step 4.2 - create a directory to store our input and data file

```
hdfs dfs -mkdir /wordcountdemo  
hdfs dfs -mkdir /wordcountdemo/input  
hdfs dfs -mkdir /wordcountdemo/output
```

Step 4.3 - Put the file file01.txt from local machine to HDFS directory

```
hdfs dfs -put /home/hadoop/Desktop/file01.txt /wordcountdemo/input
```

Step 5 : Run the application

**Syntax: `hadoop jar <Path of the jar file>
<classname>
<Path of the input file>
<path of output file>`**

```
hadoop jar /home/hadoop/Desktop/wc.jar  
WordCount  
/wordcountdemo/input/file01.txt  
/wordcountdemo/output
```

```
[hadoop@hadoop-clone ~]$ hdfs dfs -cat /wordcountdemo/output/part-r-0000
WARNING: HADOOP_PREFIX has been replaced by HADOOP_HOME. Using value of HADOOP_P
PREFIX.
2023-04-15 11:48:12,393 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
cat: `/wordcountdemo/output/part-r-0000': No such file or directory
[hadoop@hadoop-clone ~]$ hdfs dfs -cat /wordcountdemo/output/part-r-00000
WARNING: HADOOP_PREFIX has been replaced by HADOOP_HOME. Using value of HADOOP_P
PREFIX.
2023-04-15 11:48:28,920 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
Hadoop 1
Hello 1
World 1
begin 2
end 2
enf 1
hello 2
hi 1
of 1
the 1
to 1
```

Exercise

Write a MapReduce program to categorize the book in Big Book or Small book.

(if no of pages > 300 = Big Book
else Small book)

input data(inputbook.txt)

350

250

150

450

120

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.

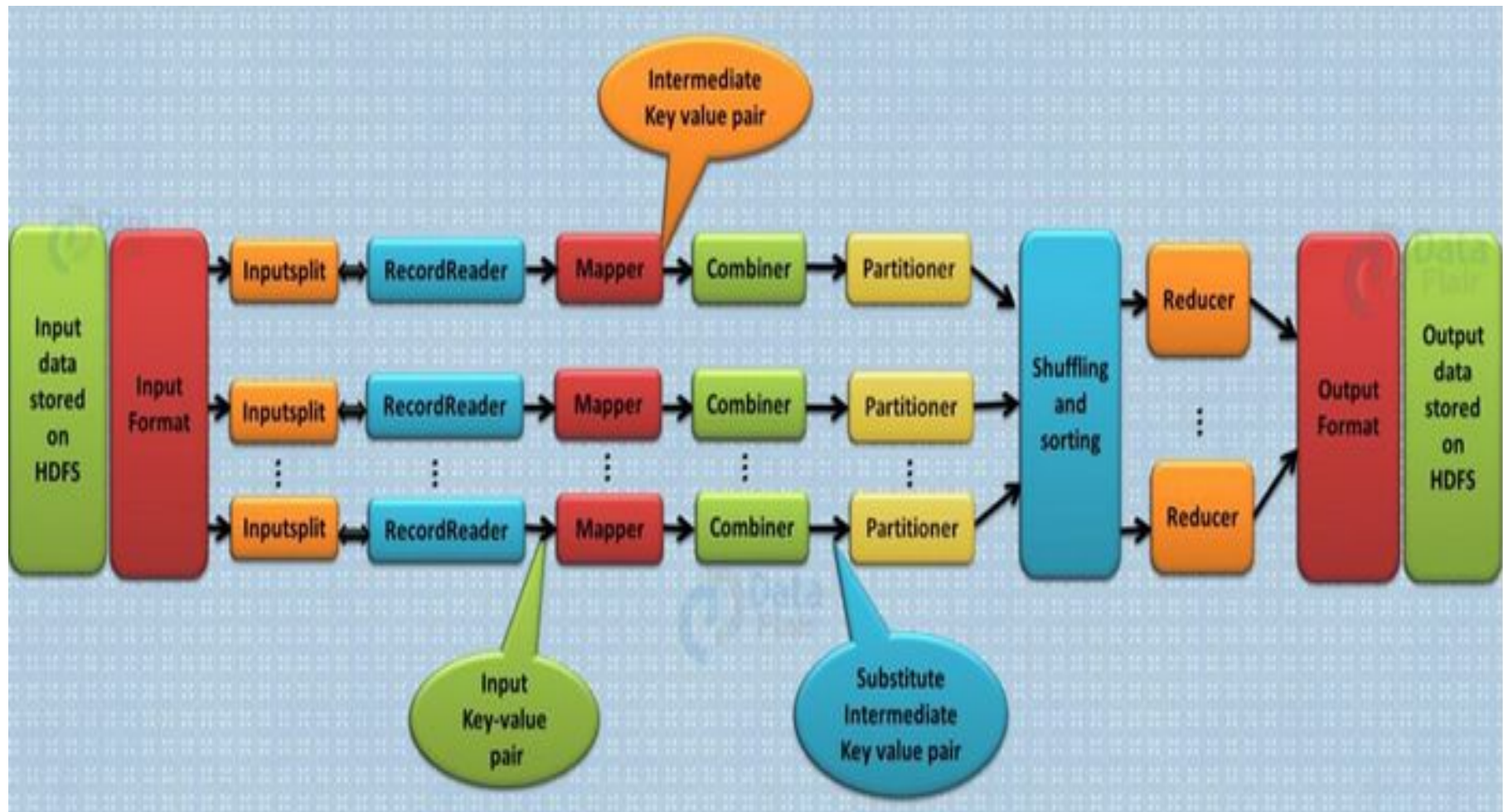
Failed Task v/s Killed Task

- 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

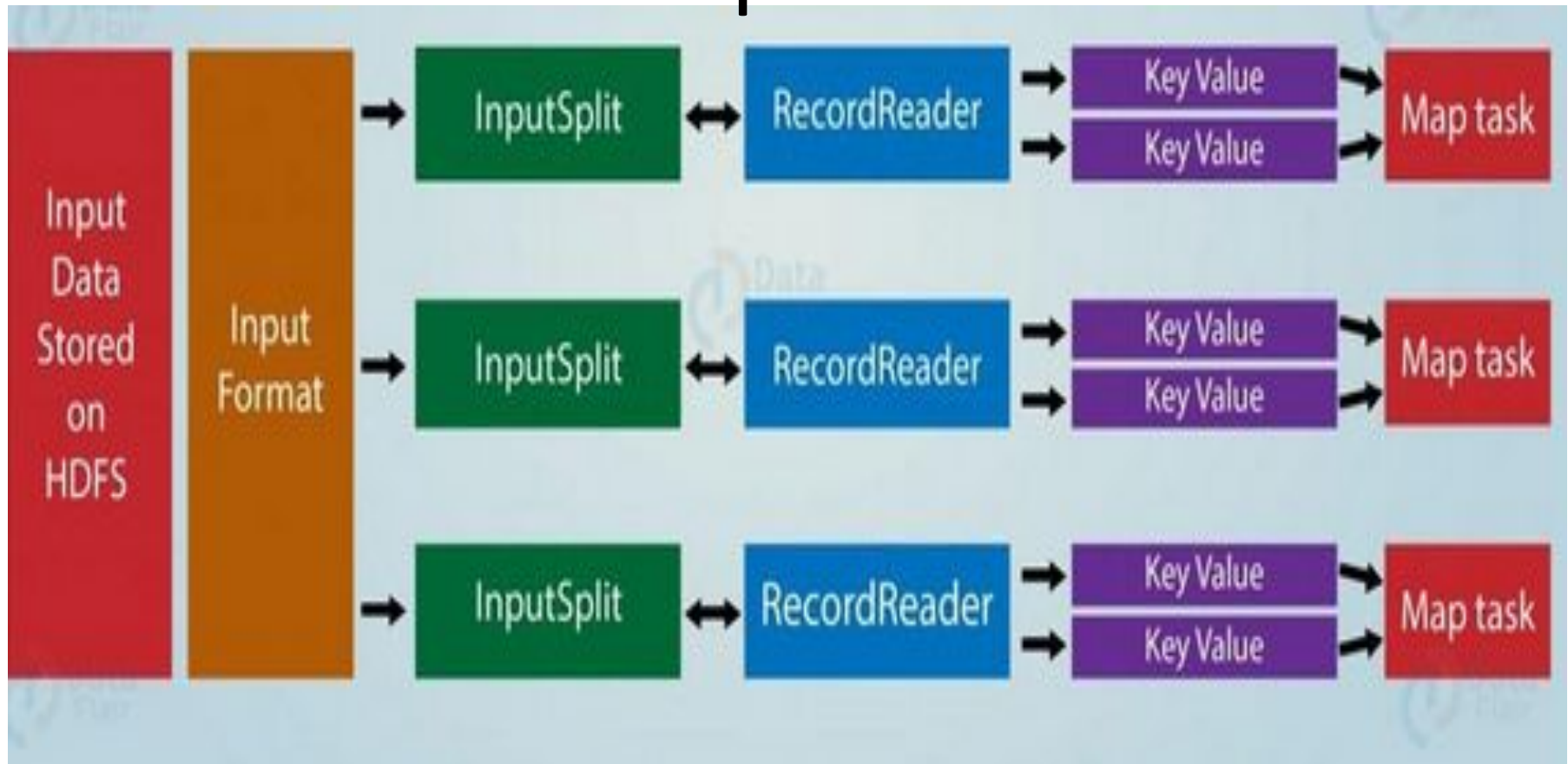
Map Reduce Job Execution Workflow



Key Value pairing in Hadoop Mapreduce

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

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.

1.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.

2.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.

Types of InputFormat in MapReduce

- FileInputFormat
- TextInputFormat(default)
- KeyValueTextInputFormat
- SequenceFileInputFormat
- SequenceFileAsTextInputFormat
- SequenceFileAsBinaryInputFormat
- NLineInputFormat
- DBInputFormat

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.

3.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 Input Splits.**
- The split is divided into records and each record will be processed by the mapper.
- Length of Input Split 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

Example.txt

130 MB

File split into
2 blocks

Block 1

128 MB

Block
2

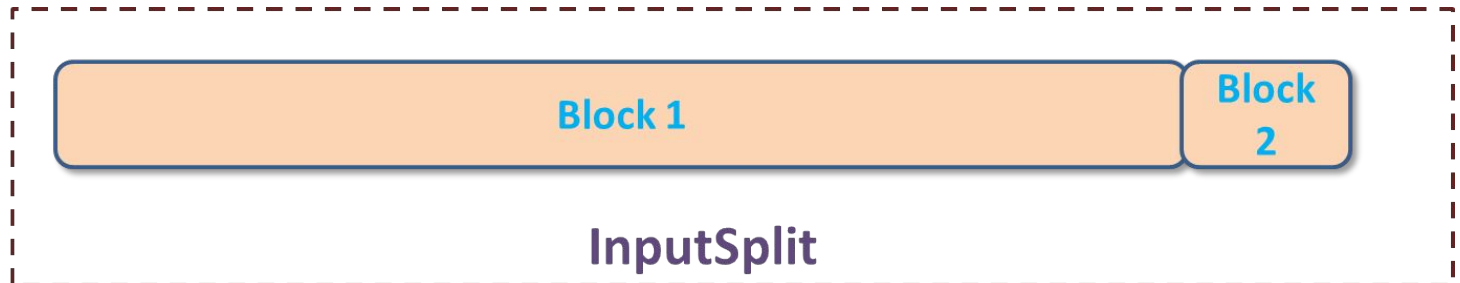
2 MB

Logical grouping
of blocks

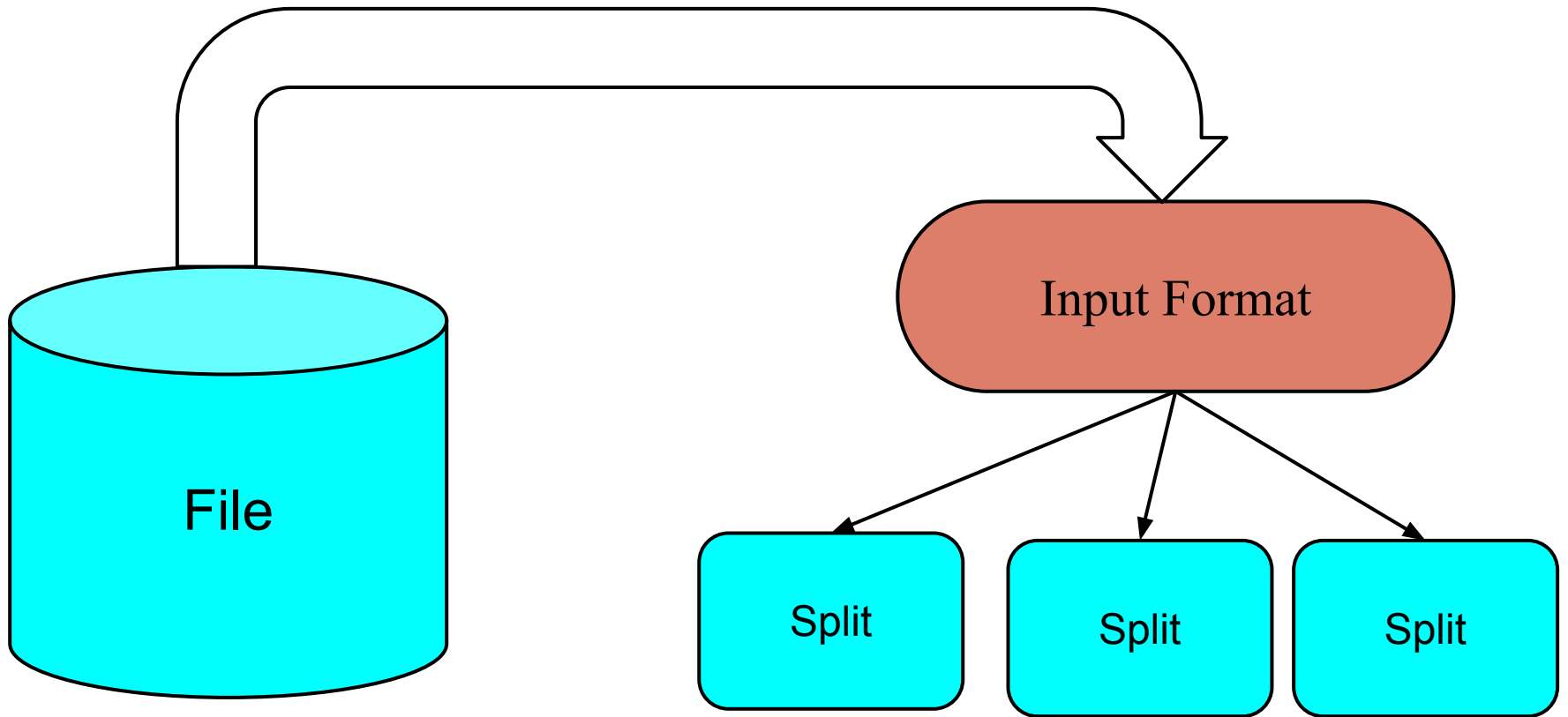
Block 1

Block
2

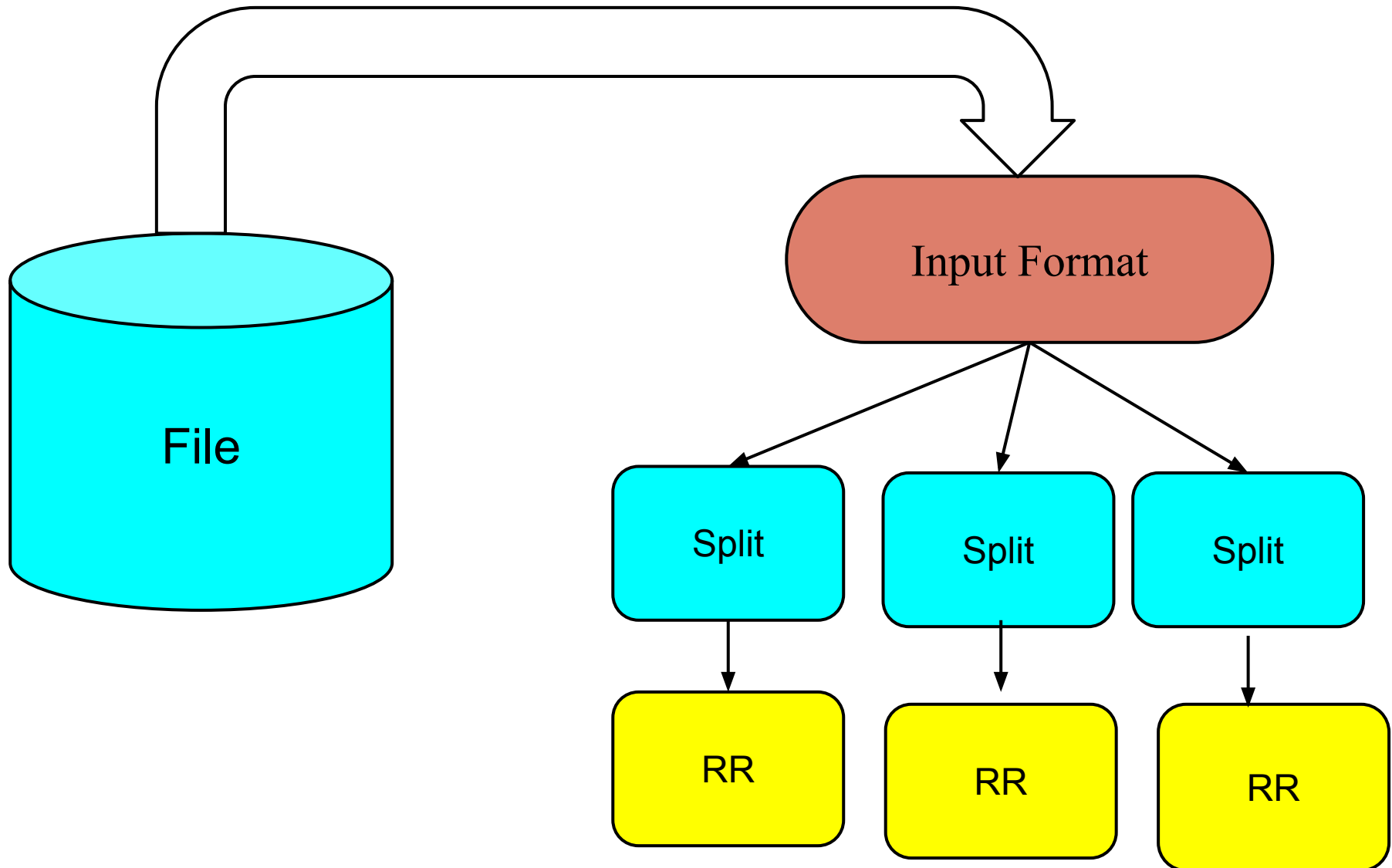
InputSplit



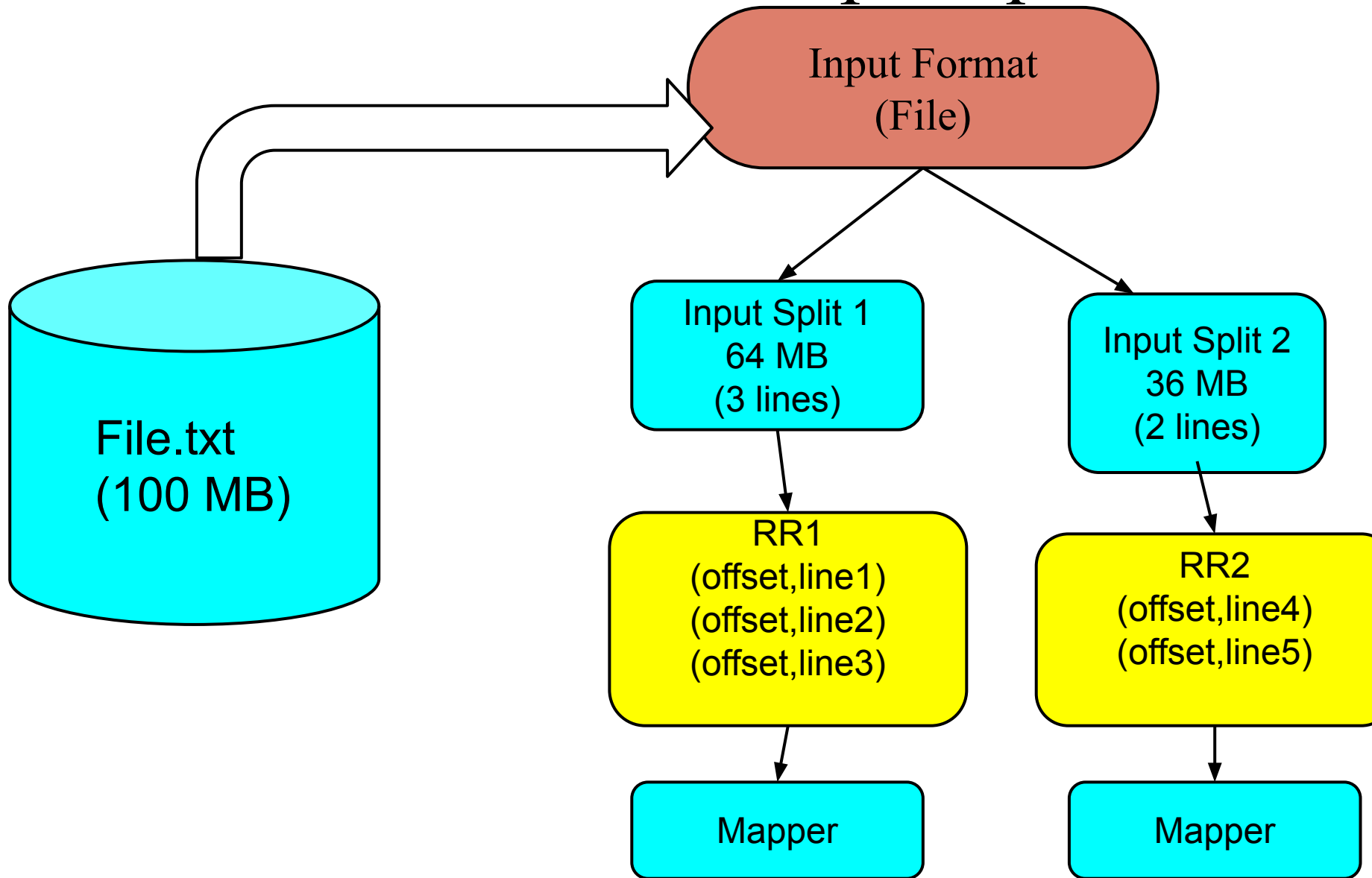
InputSplit in Hadoop Mapreduce



4. Record Reader in Hadoop Mapreduce



Record Reader in Hadoop Mapreduce



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 sends them 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.

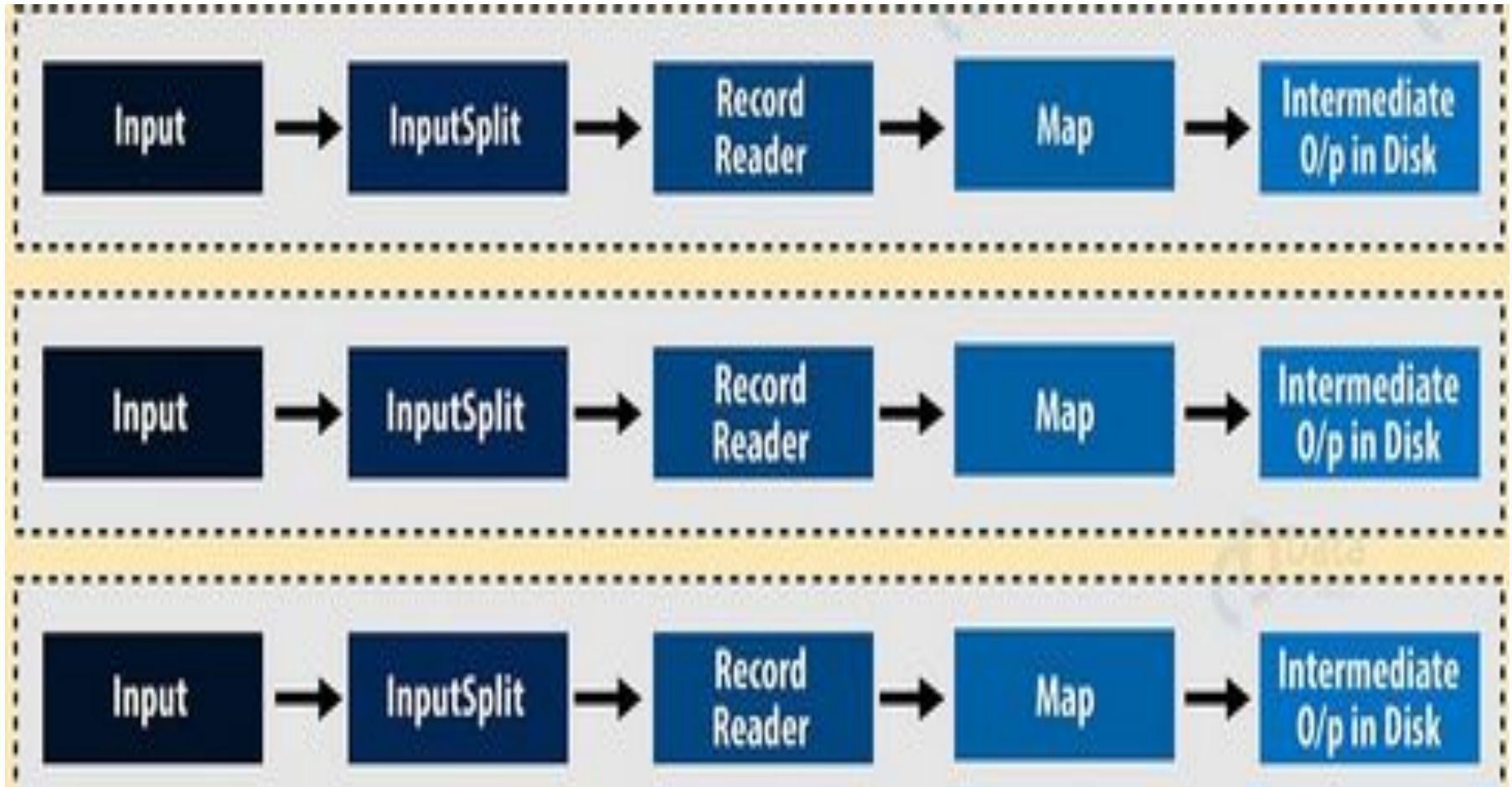
5.Mapper

- 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.
- after the completion or failure of the job the temporary location used on the local filesystem gets cleared automatically. No manual clean up process is required, it's automatically handled by the framework.

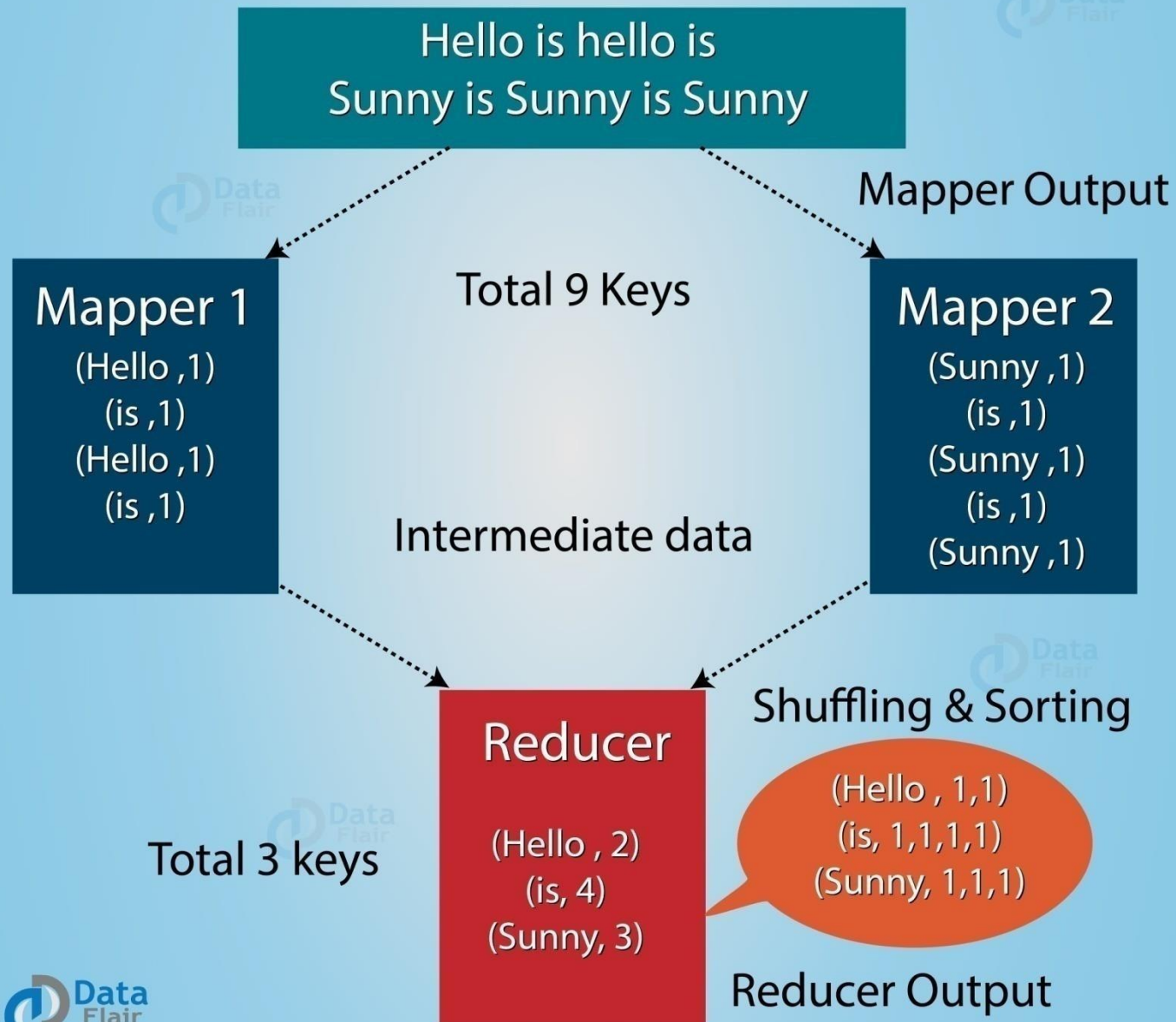
Mapper in Hadoop mapreduce



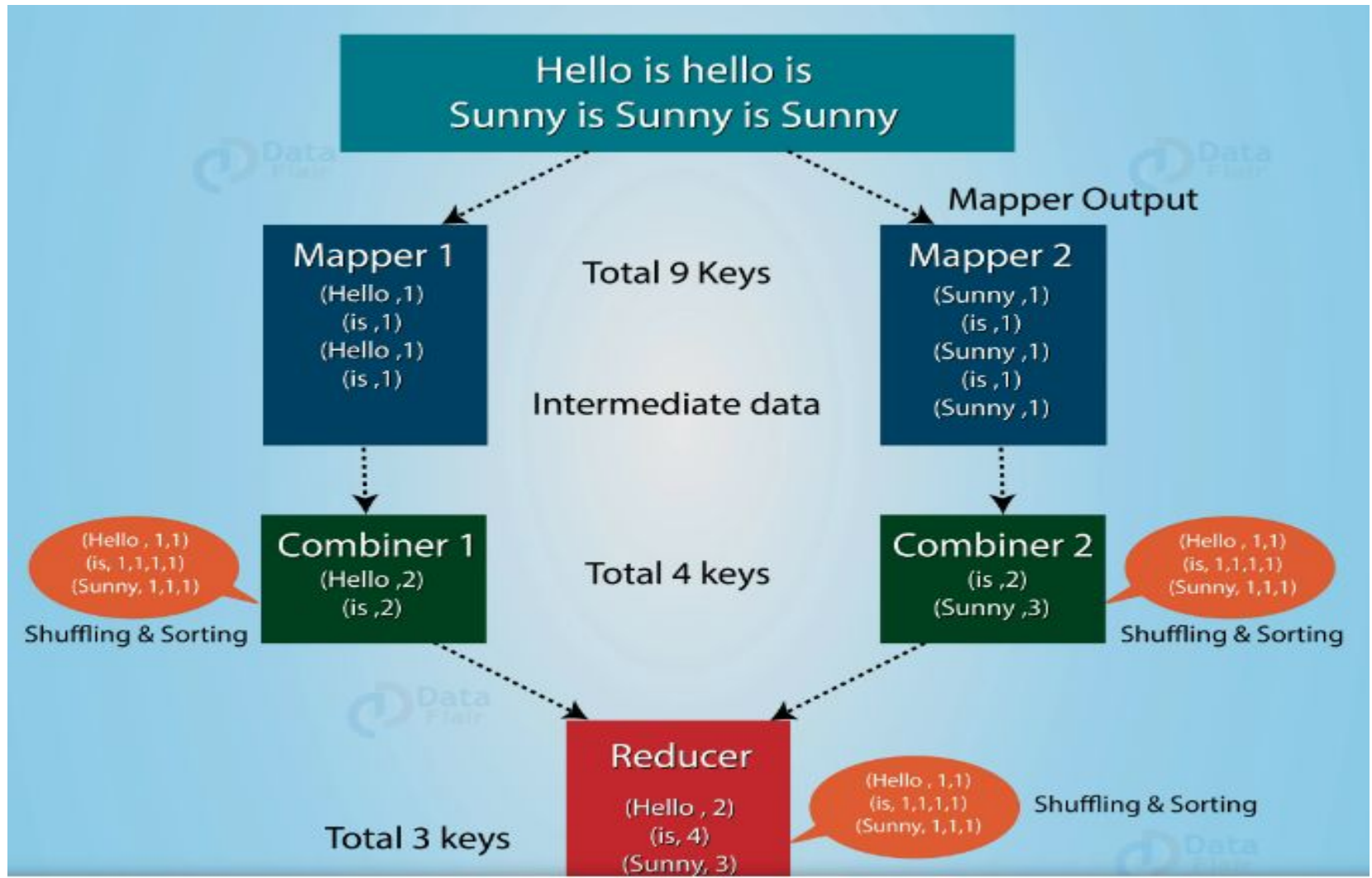
How many map tasks in Hadoop?

- 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.
- **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



Map Reduce Program with combiner



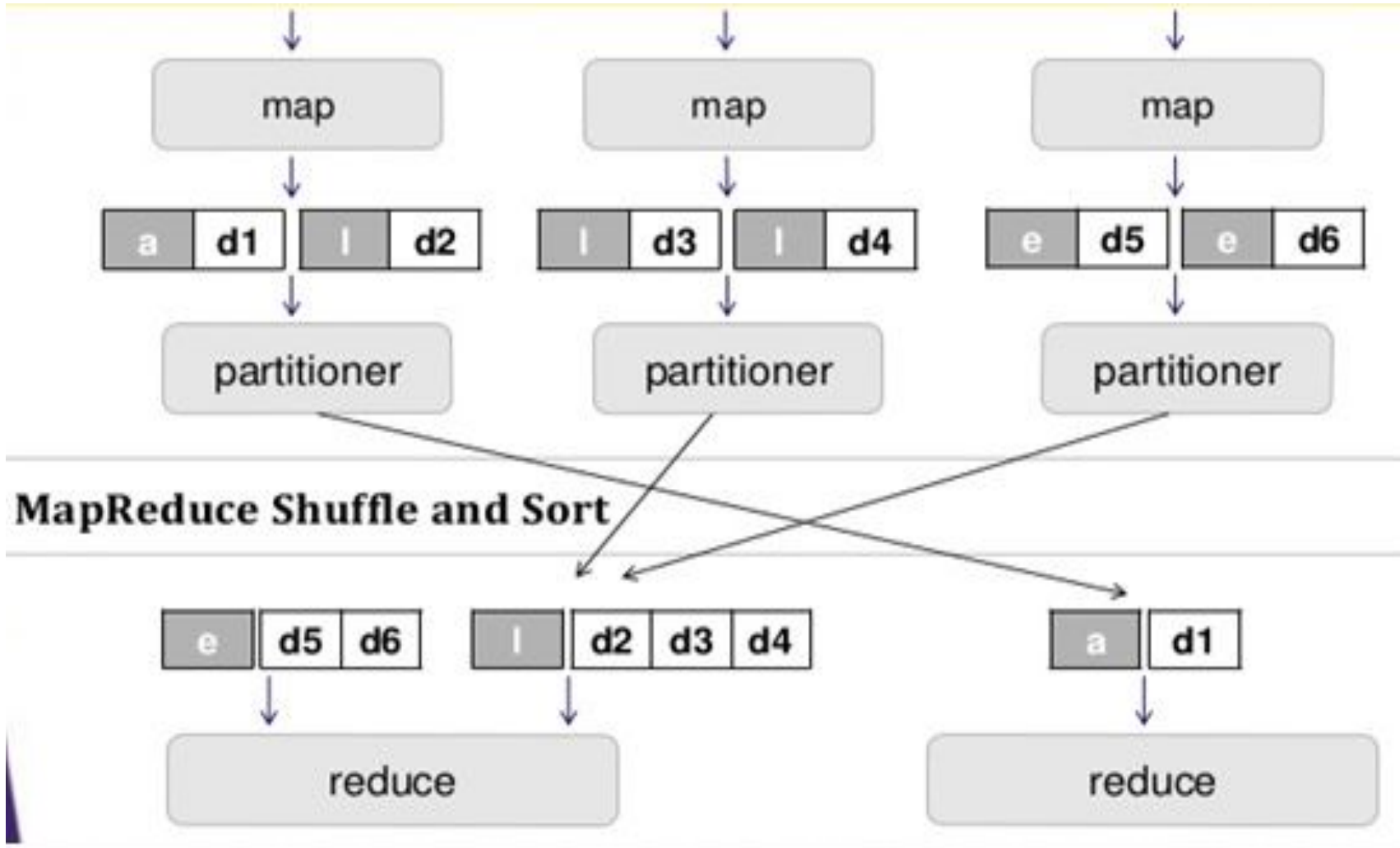
6. Combiner

- 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.

7.Partitioner



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

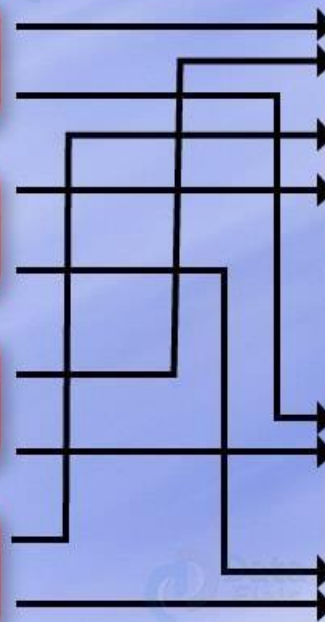
- Partitioning allows even distribution of the map output over the reducer.
- e.g. $h(k) = k \% 2$ divides data into two partitions
- Sometimes default partitioning may result in poor partitioning of data, if in data input in MapReduce job one key appears more than any other key.
- In such case, to send 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

**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**

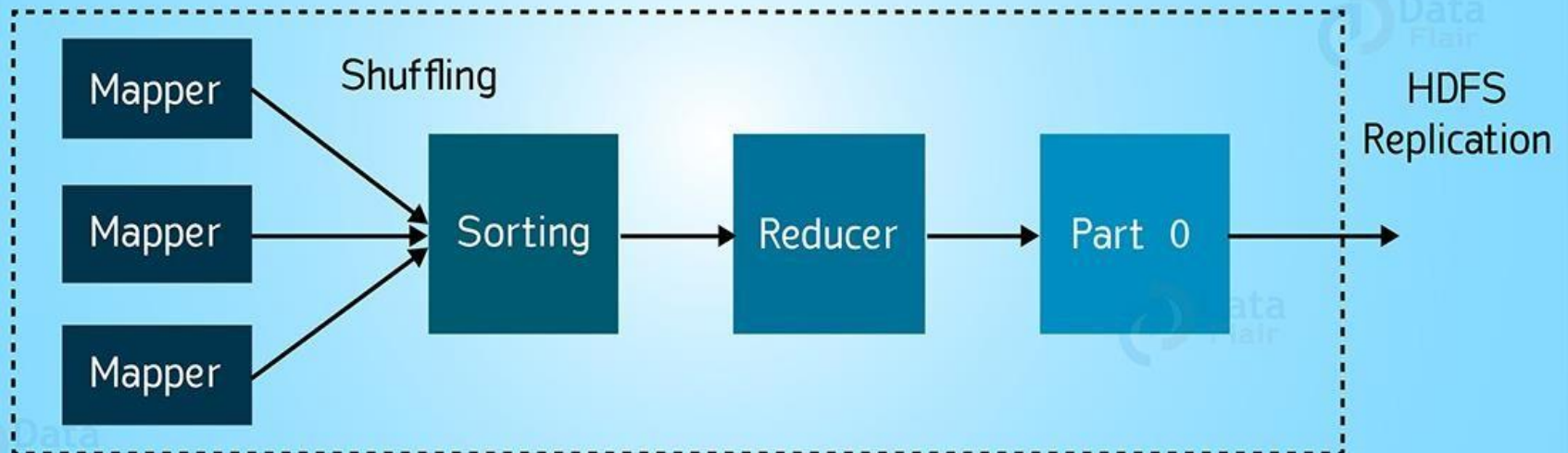
8.Shuffling and Sorting

- 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.

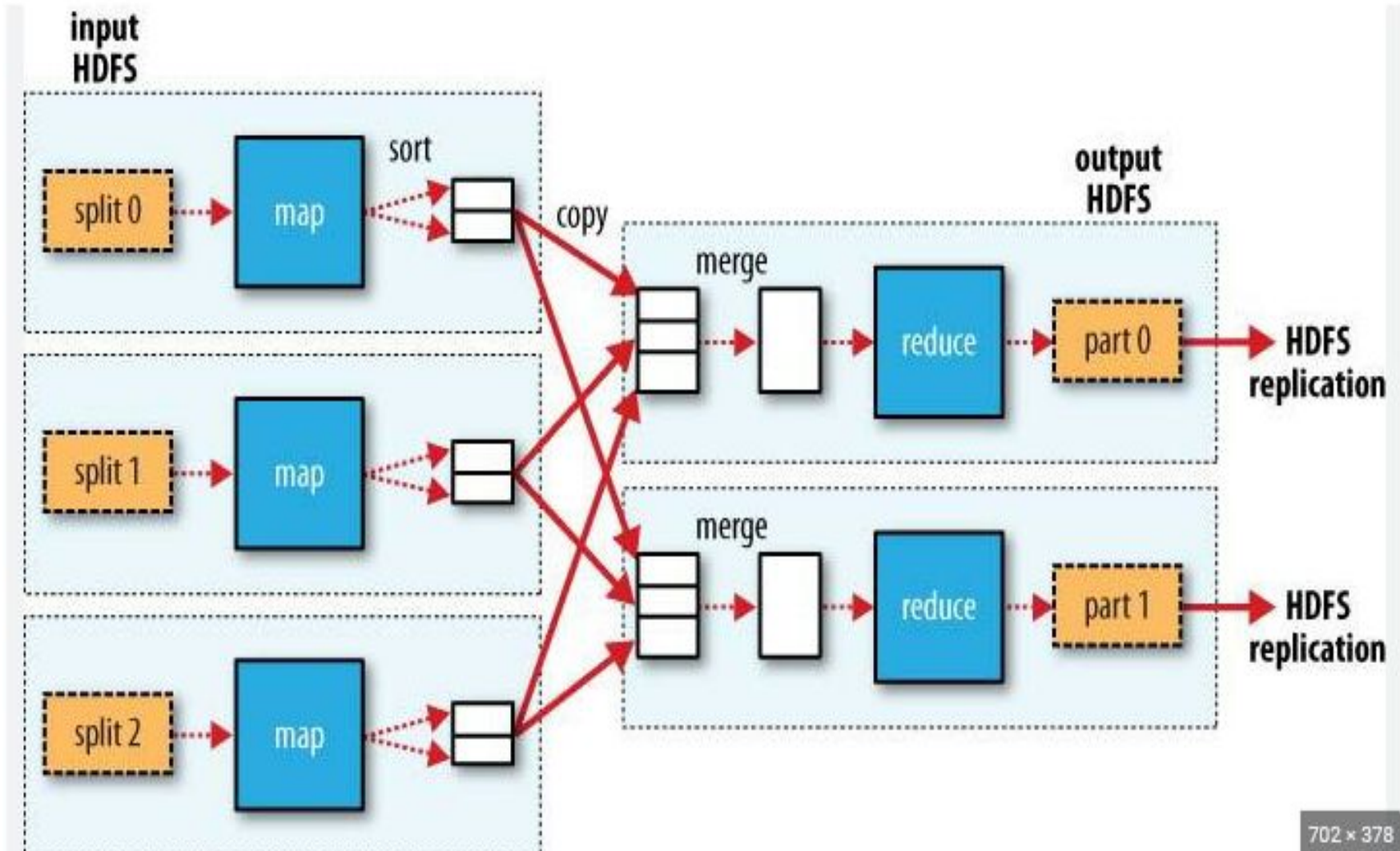
9.Reducer



Hadoop Reducer



Many Reducer



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 ($\text{no. of nodes} * \text{no. of the maximum container per node}$).
- 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.

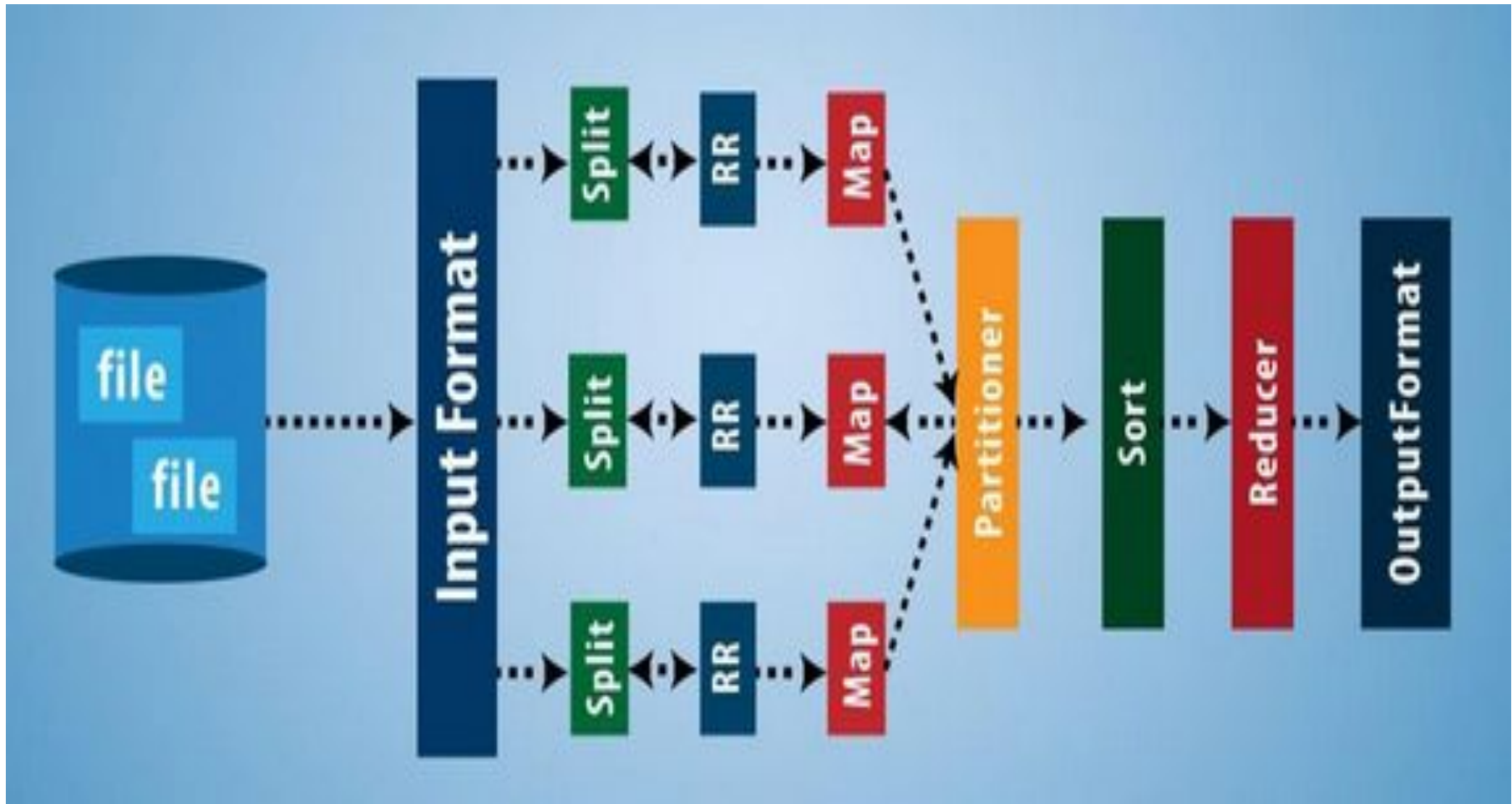
10.RecordWriter

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

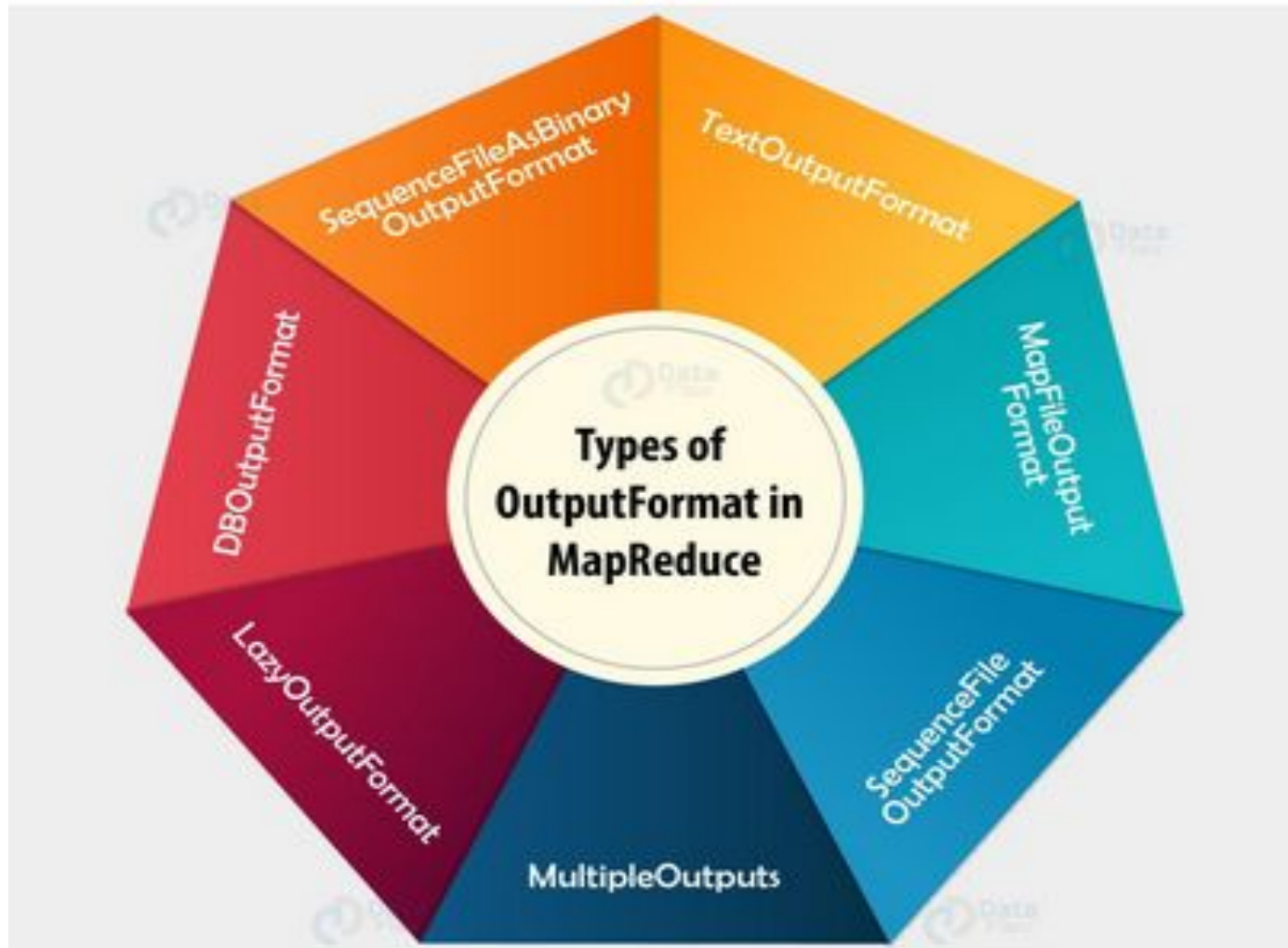
11.OutputFormat

- The 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.

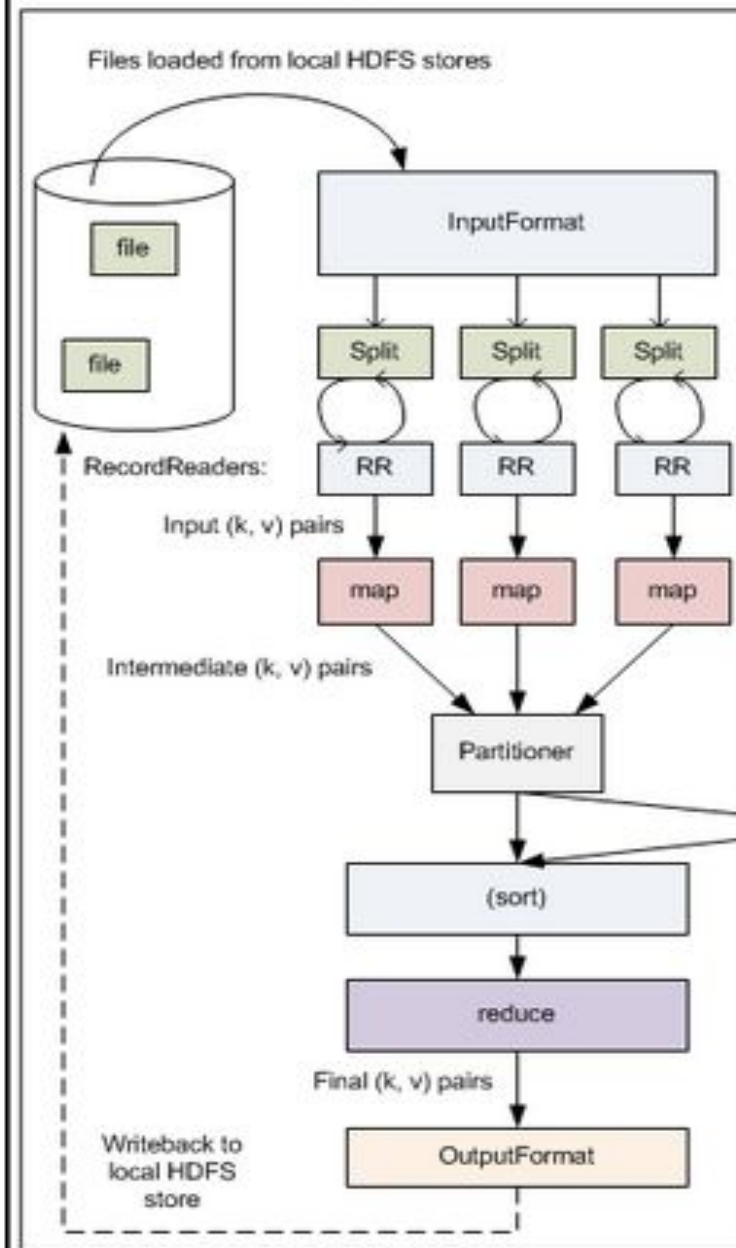
OutputFormat in Mapreduce



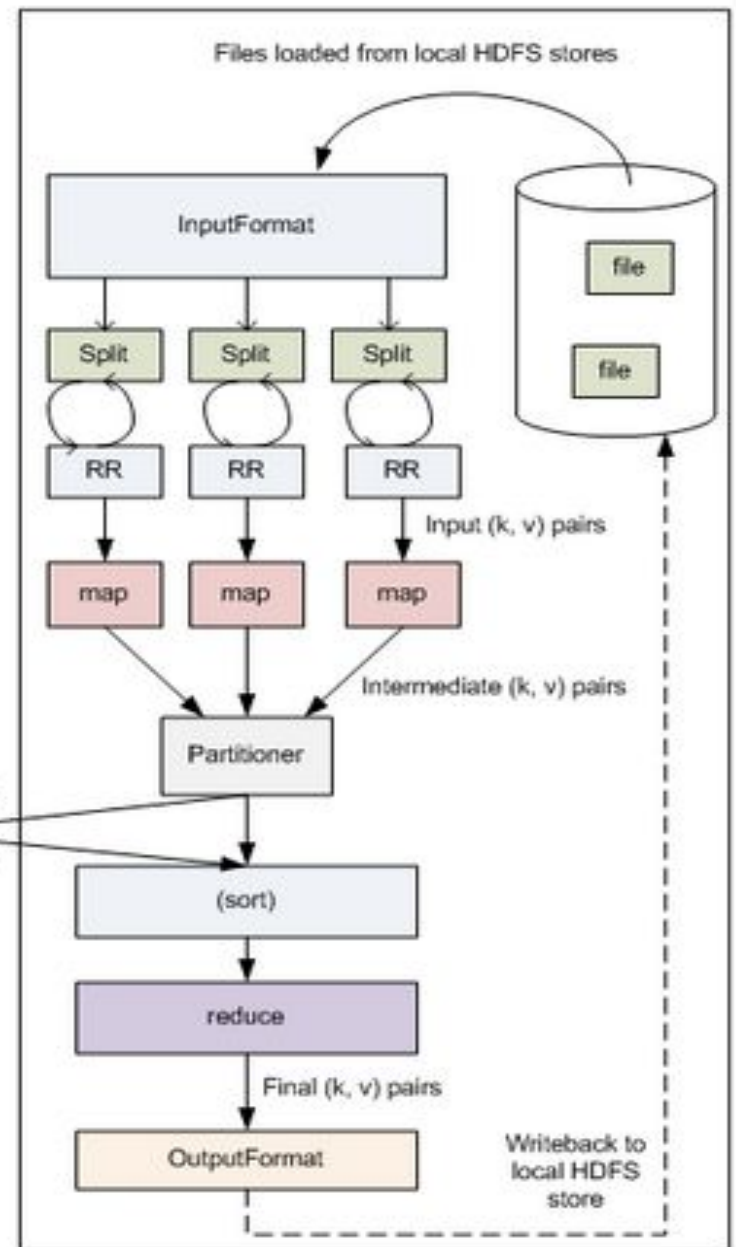
Types of Hadoop Output Formats



Node 1

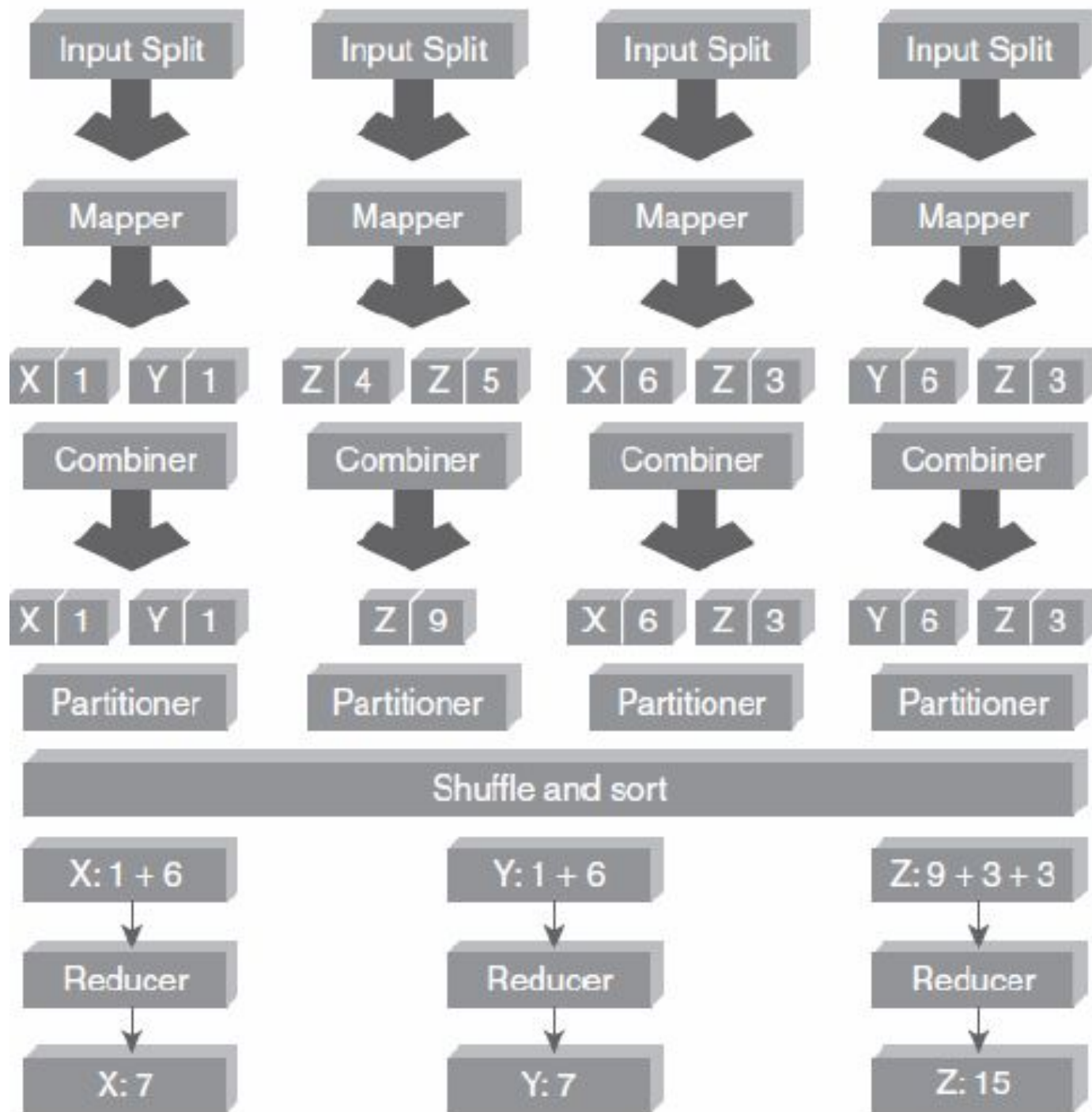


Node 2



"Shuffling" process

Intermediate (k, v) pairs exchanged by all nodes



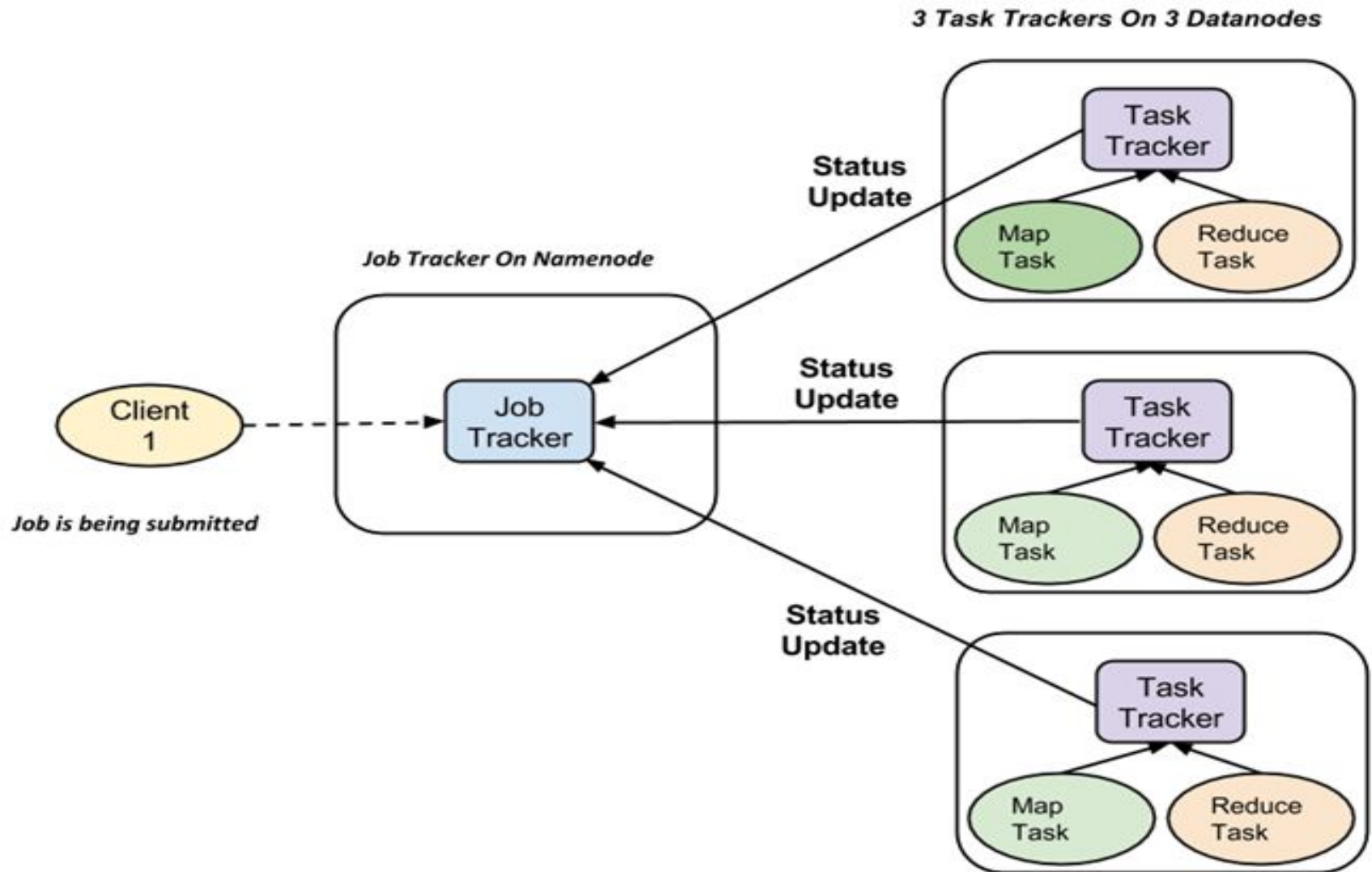
MapReduce Daemons

The complete (execution process of map and reduce tasks ,both) is controlled by two types of daemons.

1. **Job Tracker** : acts like a **master** ,responsible for complete execution of jobs.
2. **Task Tracker** : acts like a **slave**, each of them performs the job.

for every job submitted for the execution in the system ,there is **one job tracker** that resides on Namenode and there are **multiple task tracker** which reside on data node.

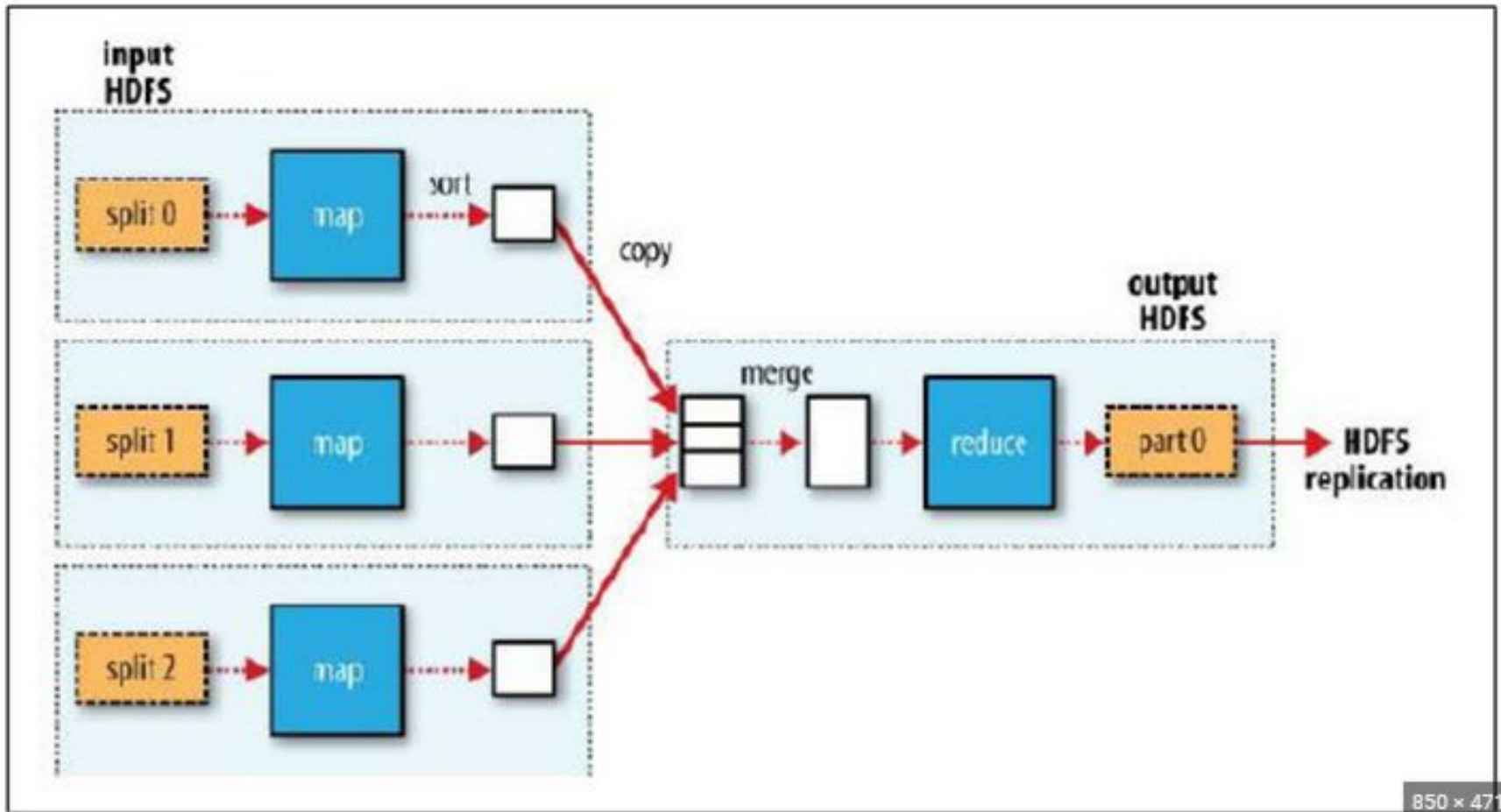
Map Reduce -Job Tracker Task Tracker



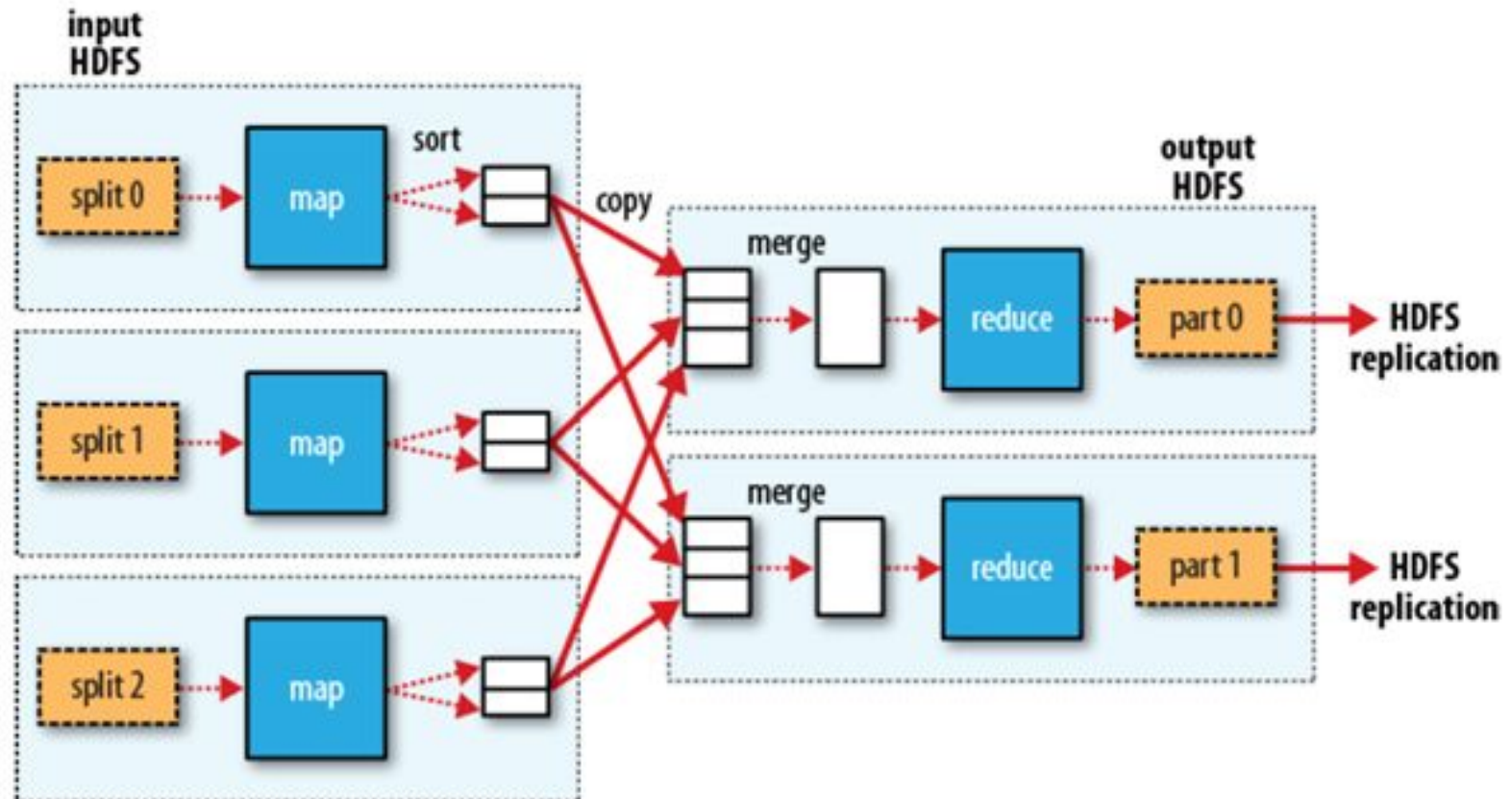
Map reduce Execution steps

- A job is divided into multiple tasks which are then run onto multiple data nodes in a cluster.
- It is the responsibility of job tracker to coordinate the activity by scheduling tasks to run on different data nodes.
- Execution of individual task is then to look after by task tracker, which resides on every data node executing part of the job.
- Task tracker's responsibility is to send the progress report to the job tracker.
- In addition, task tracker periodically sends '**heartbeat**' signal to the Jobtracker so as to notify him of the current state of the system.
- Thus job tracker keeps track of the overall progress of each job. In the event of task failure, the job tracker can reschedule it on a different task tracker.

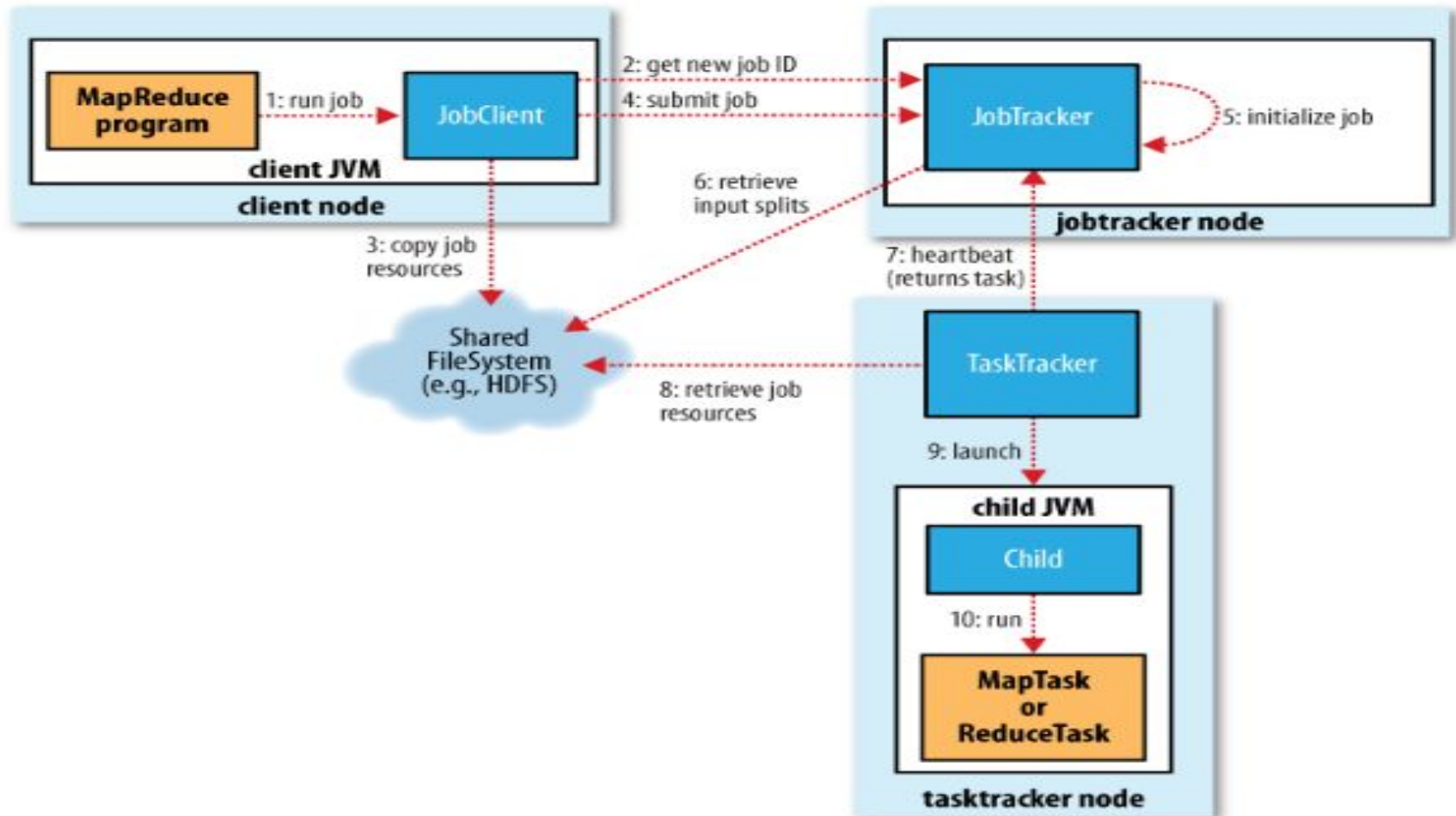
Data Flow in Mapreduce



Data Flow in Map reduce



Map Reduce Architecture (Version1)(Working of Map reduce)



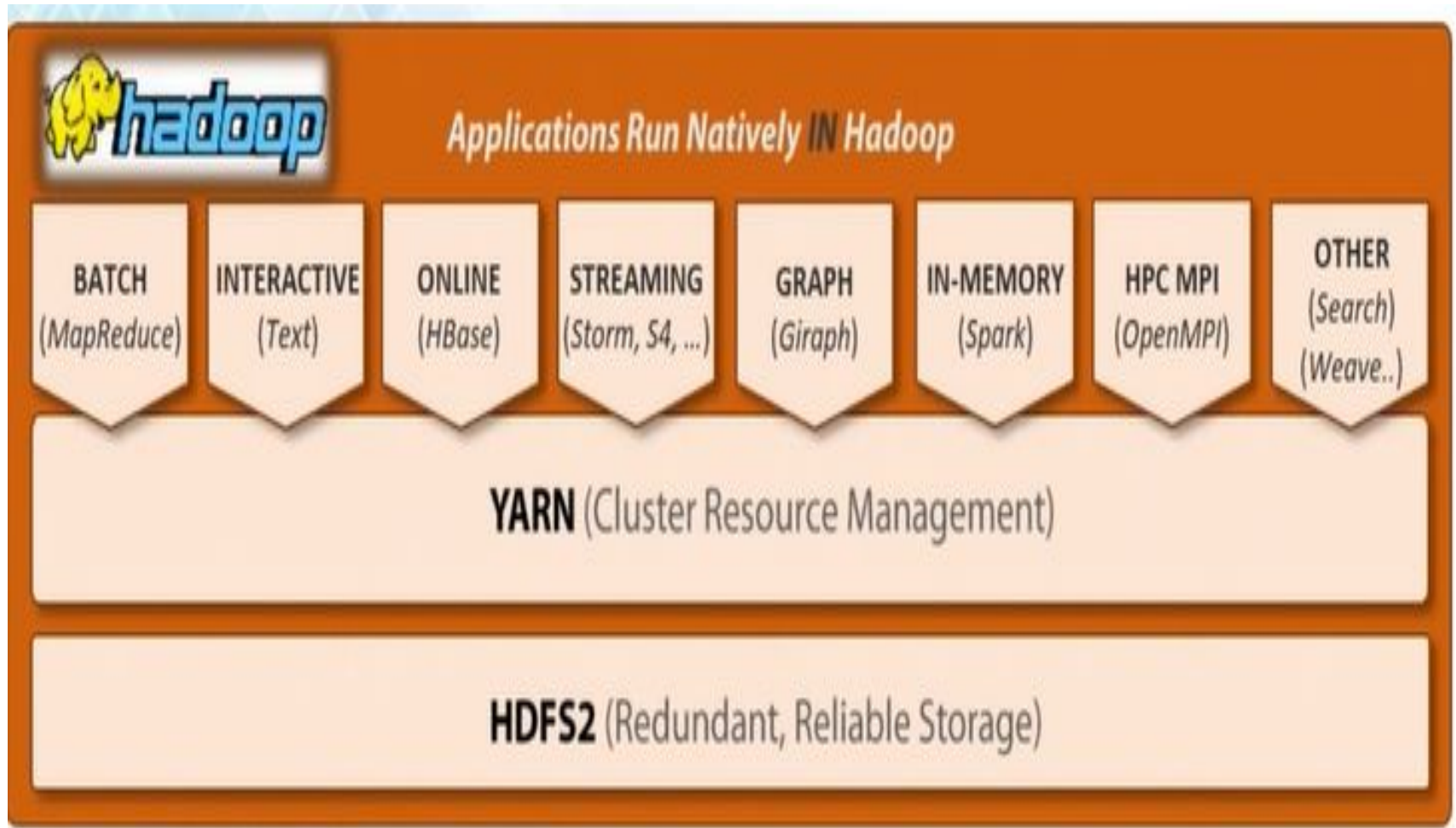
MapReduce architecture consists of various components. A brief description of these sections can enhance our understanding of how it works.

- **Job:** This is real work that needs to be done or processed
- **Task:** This is a piece of real work that needs to be done or processed. The MapReduce task covers many small tasks that need to be done.
- **Job Tracker:** This tracker plays a role in organizing tasks and tracking all tasks assigned to a task tracker.
- **Task Tracker:** This tracker plays the role of tracking activity and reporting activity status to the task tracker.
- **Input data:** This is used for processing in the mapping phase.
- **Exit data:** This is the result of mapping and mitigation.
- **Client:** This is a program or Application Programming Interface (API) that sends tasks to MapReduce. It can accept services from multiple clients.
- **Hadoop MapReduce Master:** This plays the role of dividing tasks into sections.
- **Job Parts:** These are small tasks that result in the division of the primary function

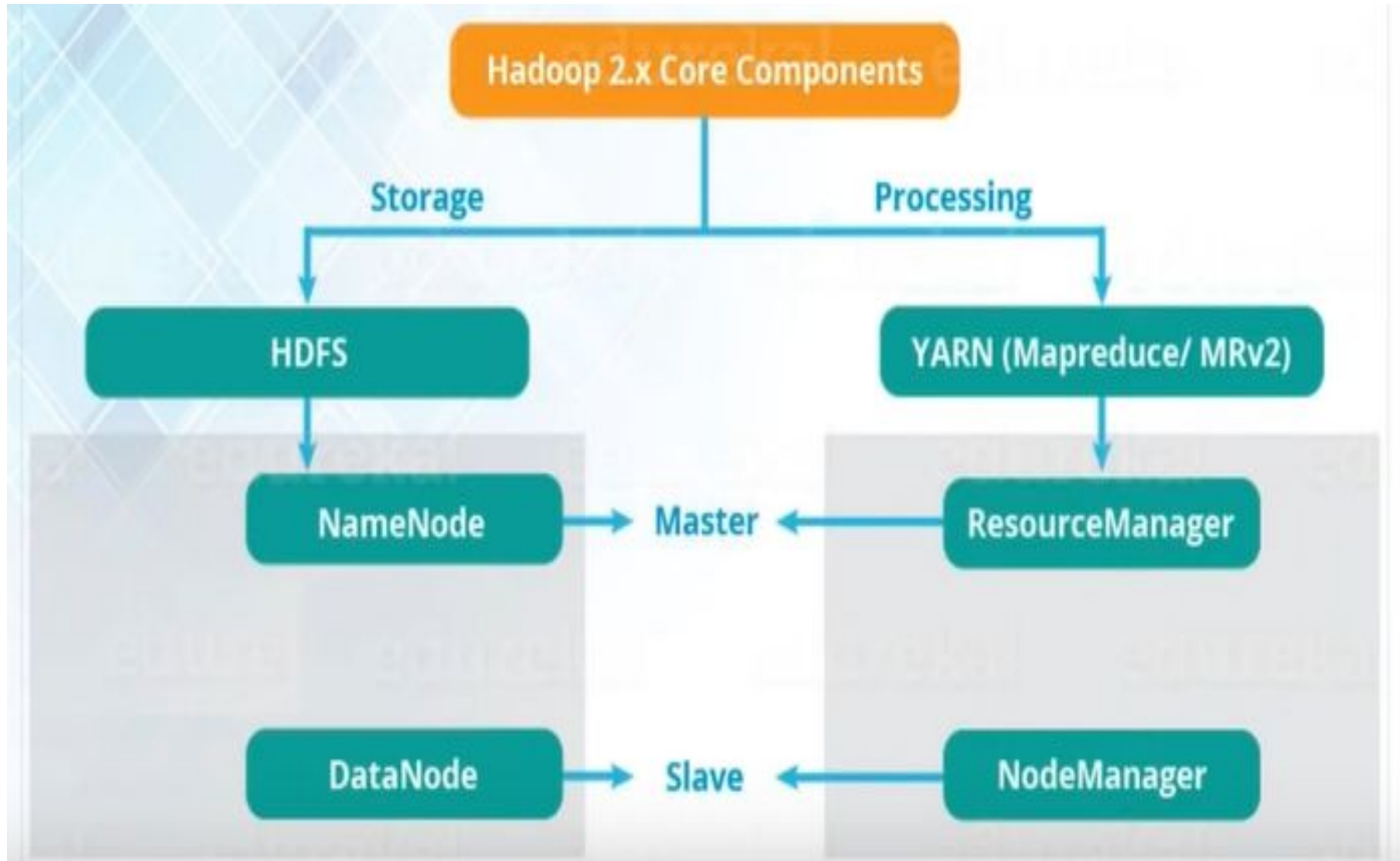
YARN

Yet Another Resource Negotiator

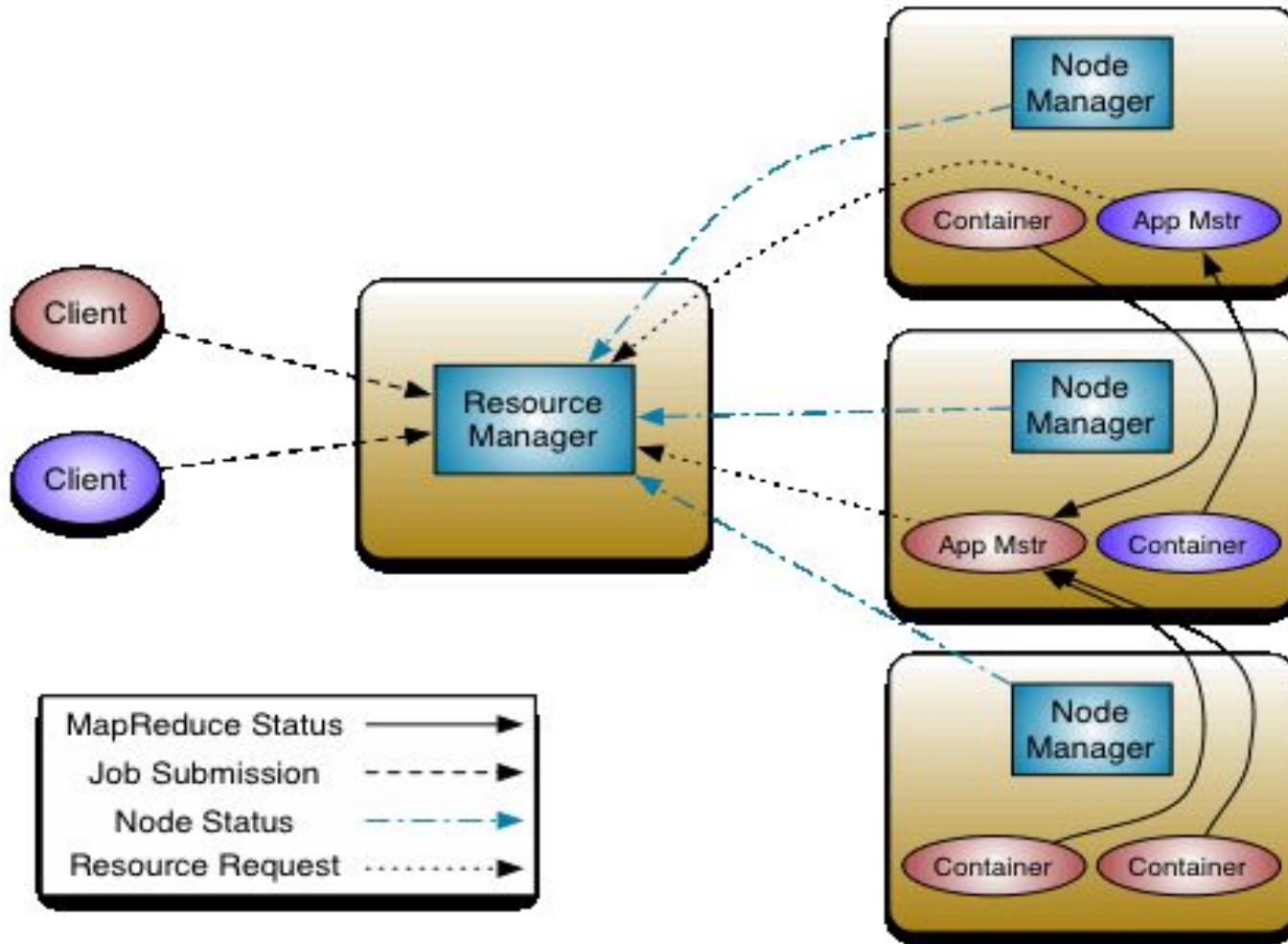
YARN- Moving beyond Mapreduce



HDFS 2.x Daemons



YARN Architecture



ResourceManager:

- Master daemon that manages all other daemons & accepts job submission
- Allocates first container for the AppMaster

Resource
Manager

NodeManager:

- Responsible for containers, monitoring their resource usage i.e. (cpu, memory, disk, network) & reports the same to RM

Node
Manager

AppMaster:

- One per application
- Coordinates and manages MR Jobs
- Negotiates resources from RM

App
Master

container

Container:

- Allocates certain amount of resources (memory, CPU etc.) on a slave node (NM)



Hadoop 2.x YARN components

→ Client

- » Submits a MapReduce Job

→ Resource Manager

- » Cluster Level resource manager
- » Long Life, High Quality Hardware

→ Node Manager

- » One per Data Node
- » Monitors resources on Data Node

→ Job History Server

- » Maintains information about submitted MapReduce jobs after their ApplicationMaster terminates

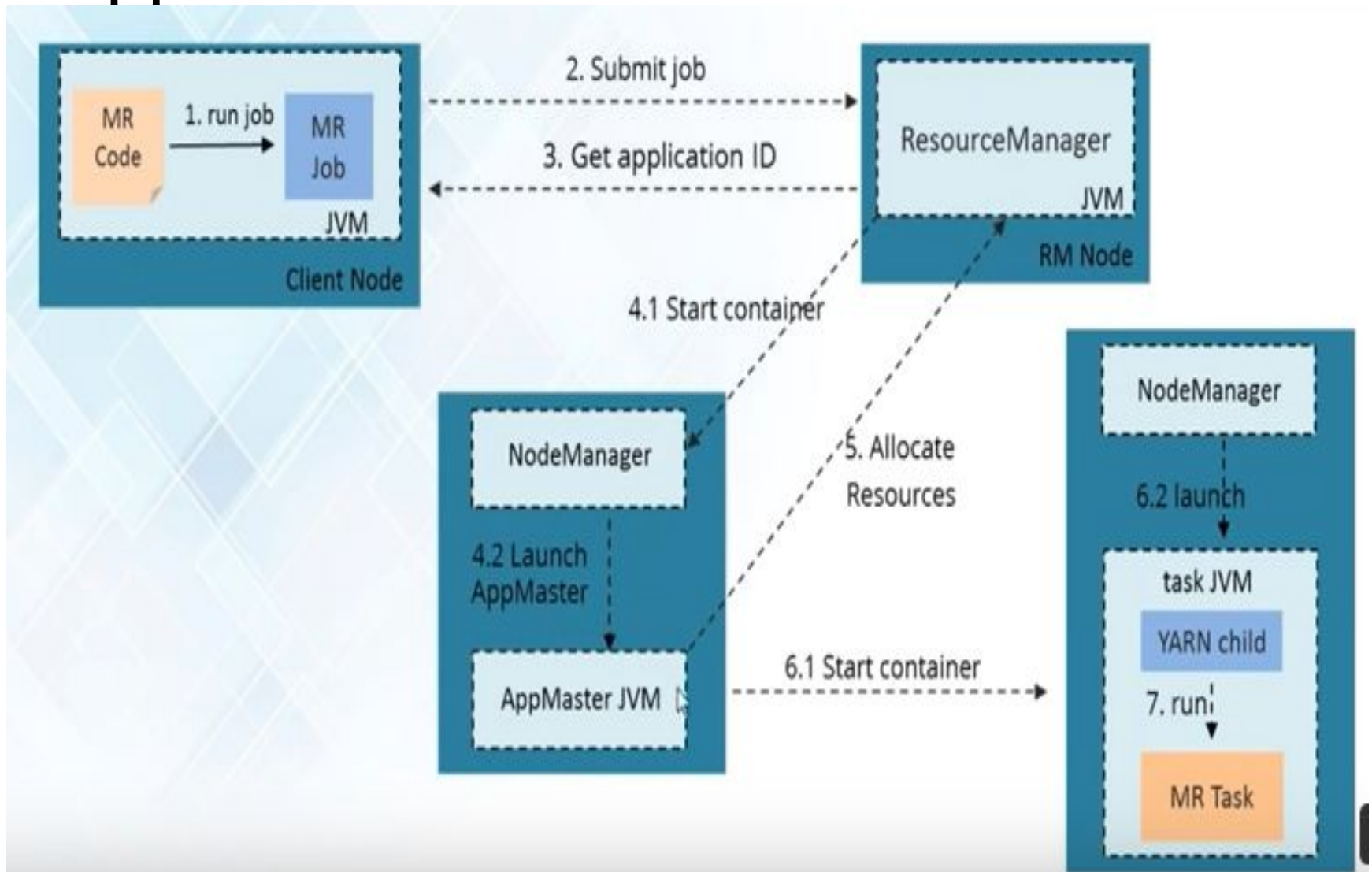
→ ApplicationMaster

- » One per application
- » Short life
- » Coordinates and Manages MapReduce Jobs
- » Negotiates with Resource Manager to schedule tasks
- » The tasks are started by NodeManager(s)

→ Container

- » Created by NM when requested
- » Allocates certain amount of resources (memory, CPU etc.) on a slave node

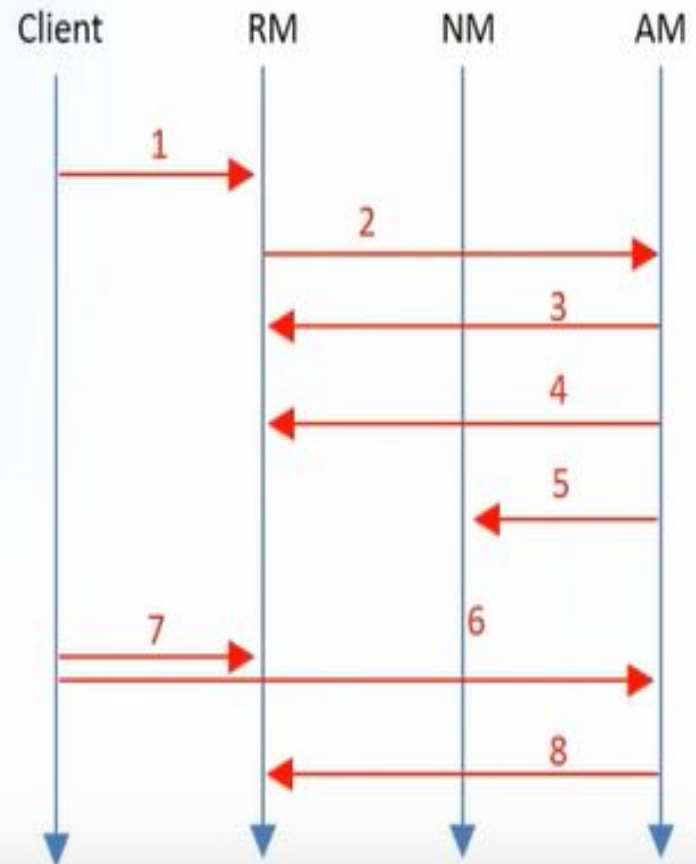
Application Workflow in YARN



Application Workflow in YARN

→ Execution Sequence :

1. Client submits an application
2. RM allocates a container to start AM
3. AM registers with RM
4. AM asks containers from RM
5. AM notifies NM to launch containers
6. Application code is executed in container
7. Client contacts RM/AM to monitor application's status
8. AM unregisters with RM



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!