Processing the Data with MapReduce

Prof. Abhaya Kumar Sahoo
School of Computer Engineering
Kalinga Institute of Industrial Technology
Deemed to be University,Bhubaneswar

Contents

- The introduction of MapReduce,
- MapReduce Architecture,
- Data flow in MapReduce Splits,
- Mapper,
- Portioning,
- Sort and shuffle,
- Combiner,
- Reducer,
- Basic Configuration of MapReduce,
- MapReduce life cycle,
- Driver Code,
- Mapper and Reducer,
- How MapReduce Works.

The introduction of MapReduce

- MapReduce is a processing technique and a program model for distributed computing based on java.
- contains two important tasks, namely Map and Reduce.
- 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).
- The processing primitive is called mapper.
- Reduce task takes the output from a map as an input and combines those data tuples into a smaller set of tuples.
- Reduce task is always performed after the map job.
- The processing primitive is called reducer.
- The major advantage of MapReduce is that it is easy to scale data processing over multiple computing nodes.

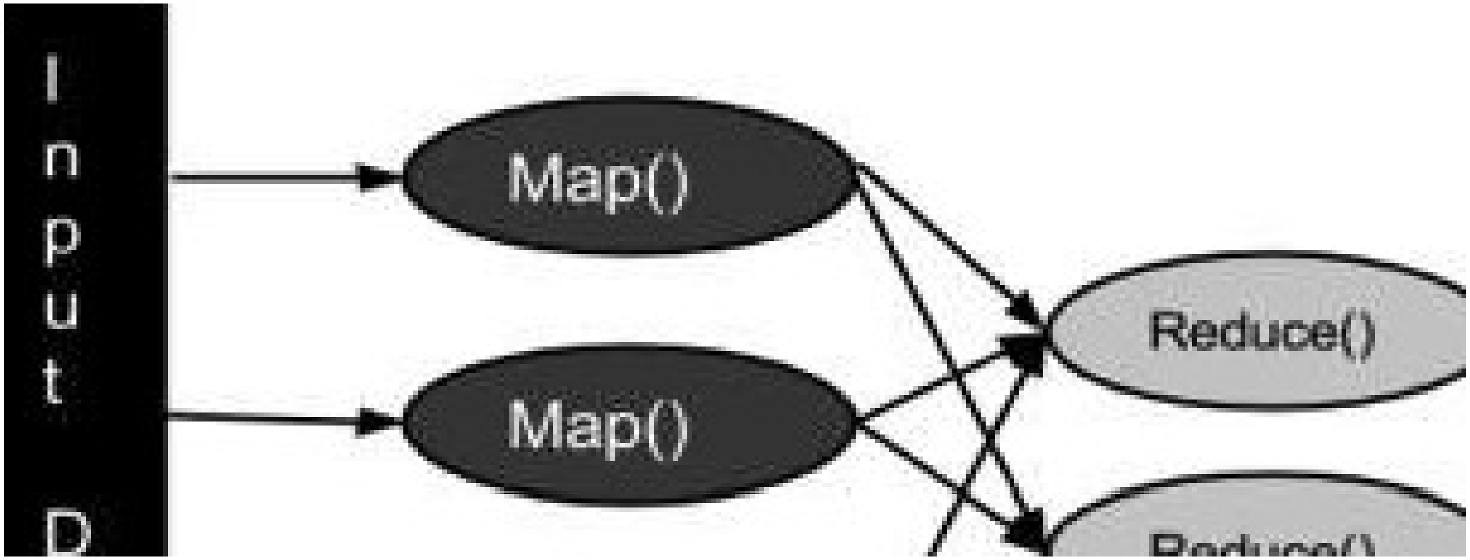
The introduction of MapReduce

- The main advantages is that 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 with a configuration change.
  - MapReduce program executes in three stages:
    - map stage,
    - shuffle stage,
    - reduce stage.
- **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.

The introduction of 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.

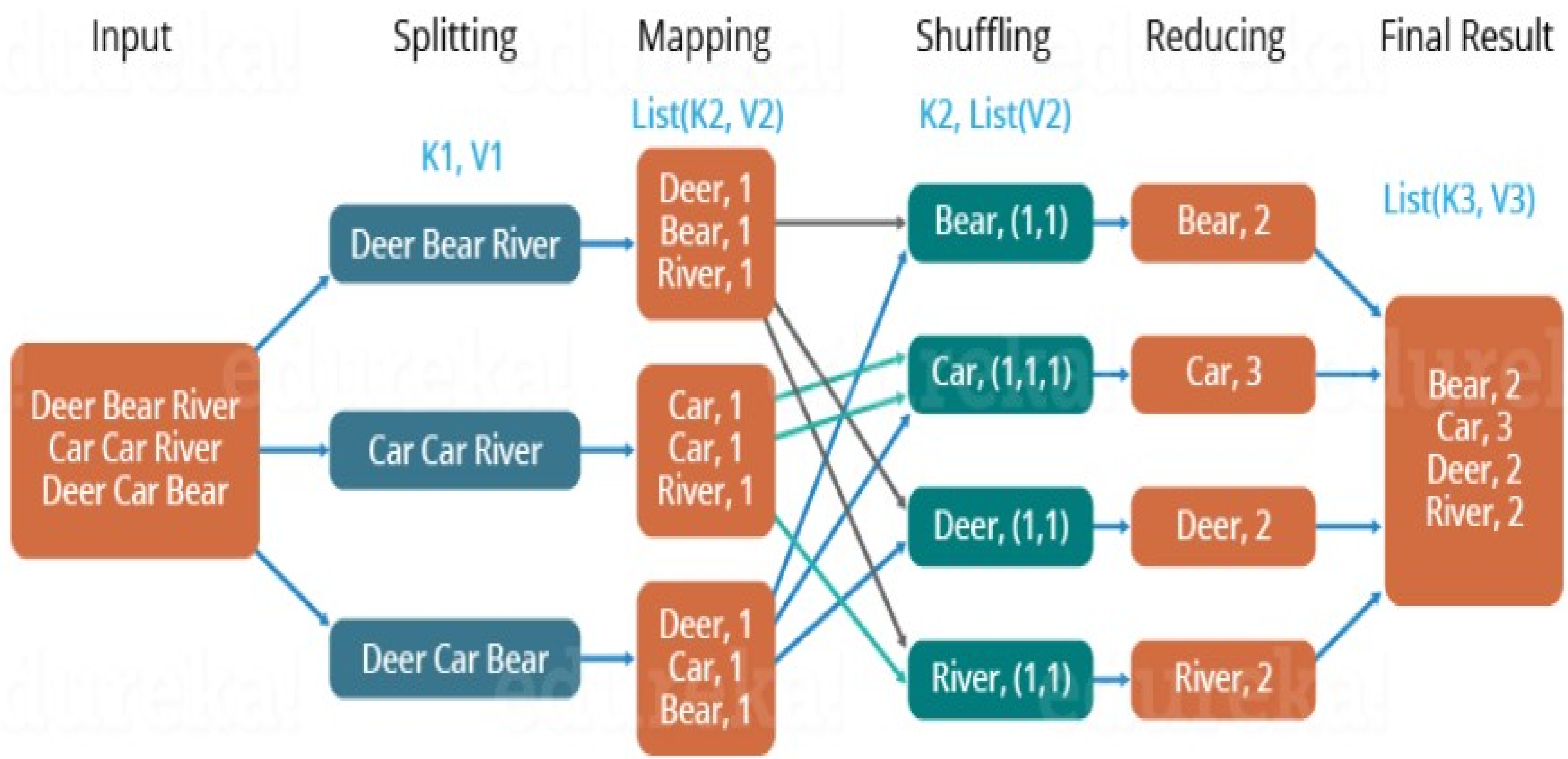
Fundamental Principle



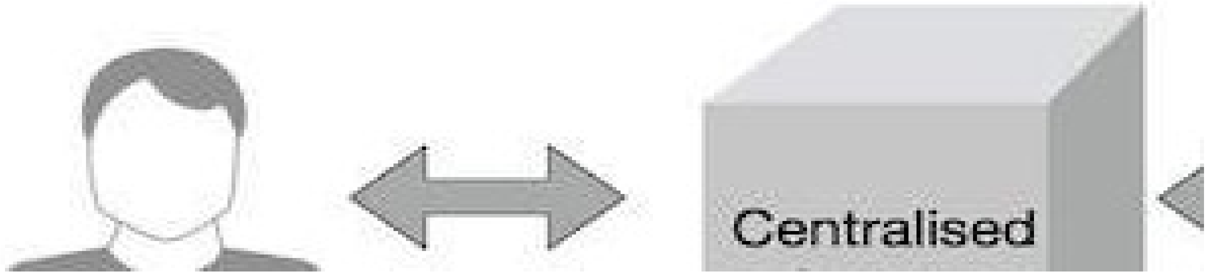
- During a MapReduce job, Hadoop sends the Map and Reduce tasks to the appropriate servers in the cluster.
- The framework manages all the details of data-passing such as issuing tasks, verifying task completion, and copying data around the cluster between the nodes.

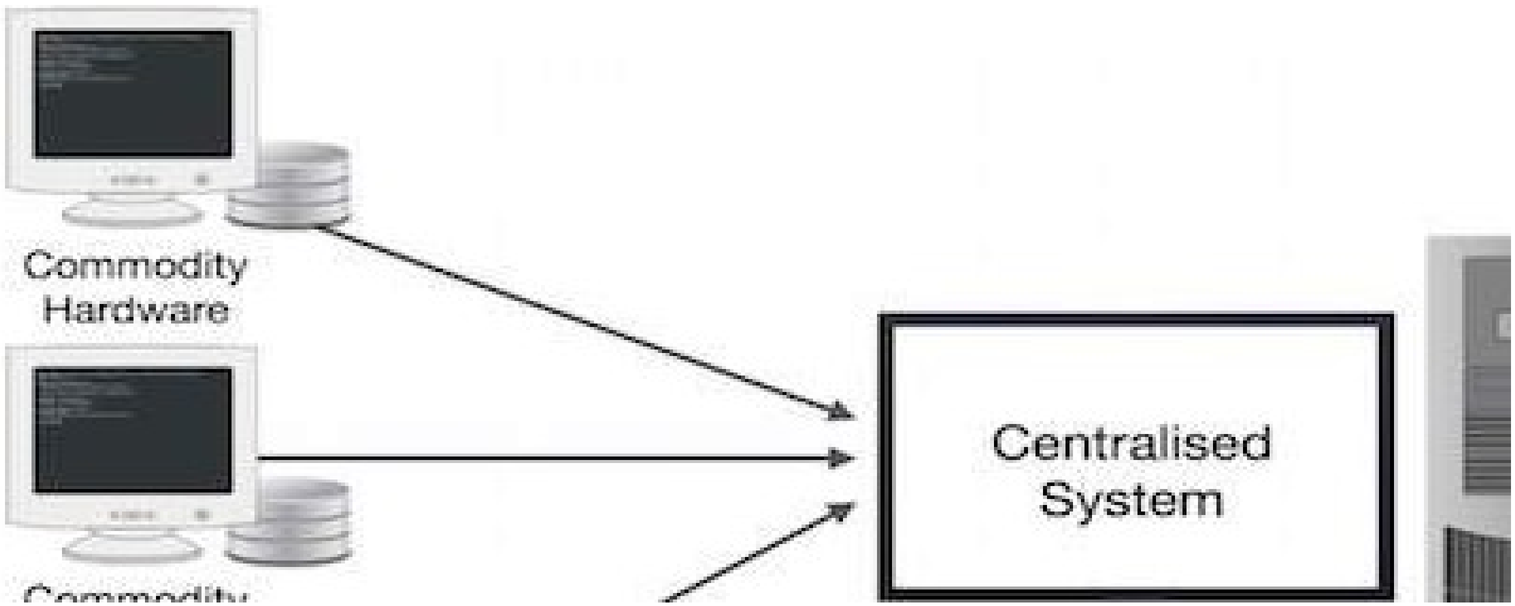
Fundamental Principle

- ❖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.


Input
Splitting
Mapping
Shuffling
Reducing
Final Result
K1, V1
List(K2, V2)
K2, List(V2)
List(K3, V3)
Deer Bear River
Car Car River
Deer Car Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, (1,1)
Car, (1,1,1)
Deer, (1,1)
River, (1,1)
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2

Why MapReduce?



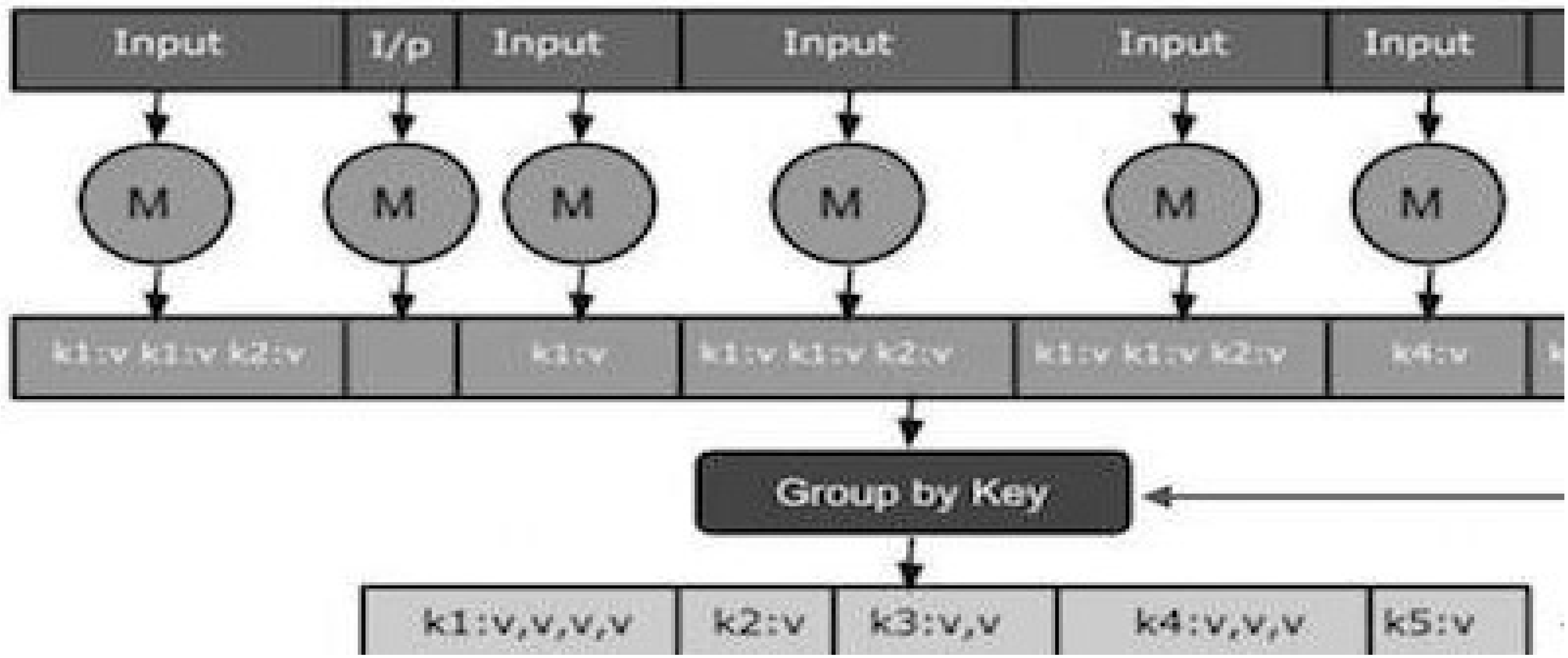


❖MapReduce divides a task into small parts and assigns them to many computers. Later, the results are collected at one place and integrated to form the result dataset.

MapReduce Works Principle

❖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.



Terms

Input Phase – Here we have a Record Reader that translates each record in an input file and sends the parsed data to the mapper in the form of key-value pairs.

Map – Map is a user-defined function, which takes a series of key-value pairs and processes each one of them to generate zero or more key-value pairs.

Intermediate Keys – They key-value pairs generated by the mapper are known as intermediate keys.

Combiner – A combiner is a type of local Reducer that groups similar data from the map phase into identifiable sets.

- It takes the intermediate keys from the mapper as input and applies a user-defined code to aggregate the values in a small scope of one mapper.
- It is not a part of the main MapReduce algorithm , it is optional.

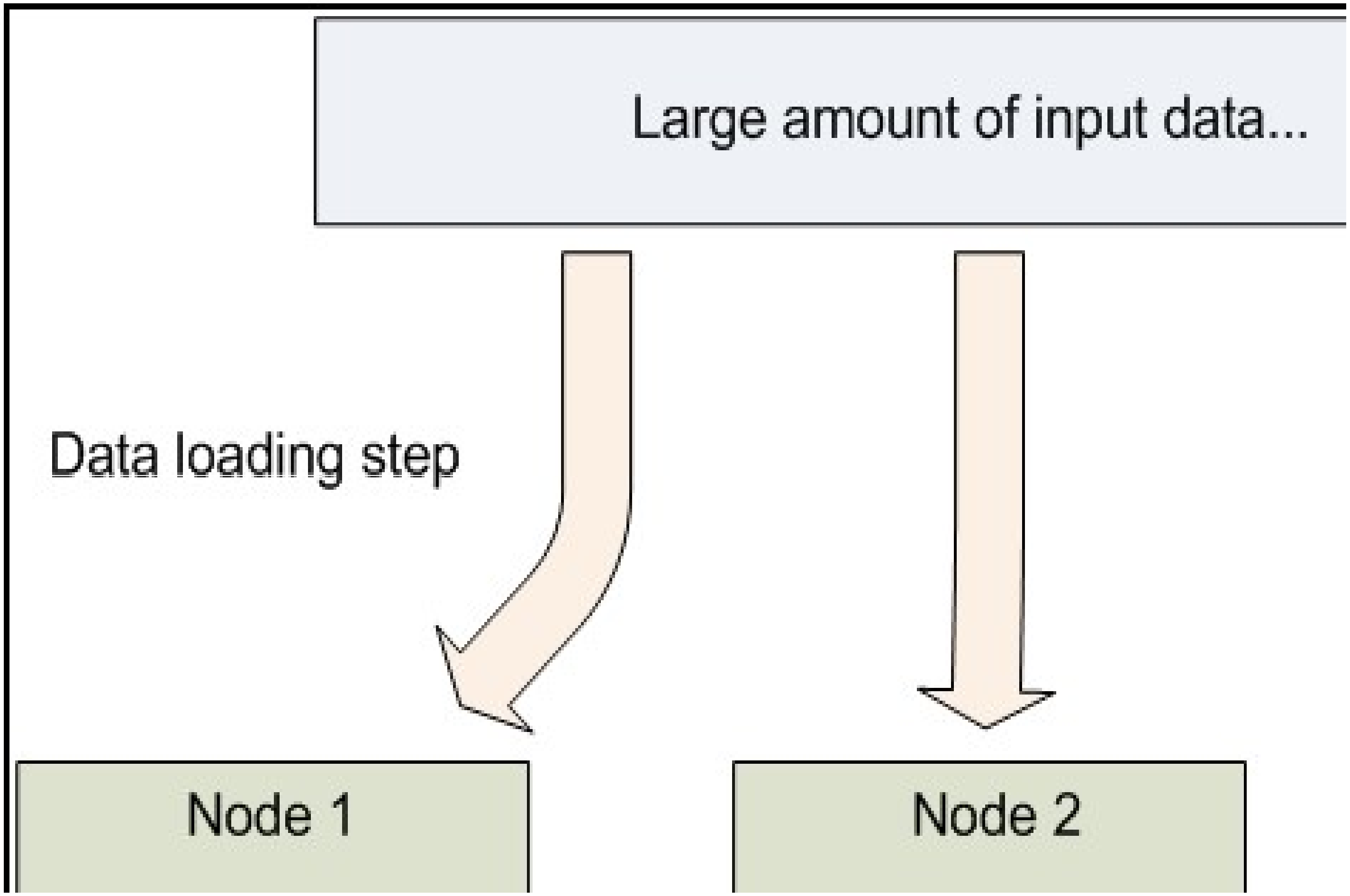
Terms

Shuffle and Sort – The Reducer task starts with the Shuffle and Sort step. It downloads the grouped key-value pairs onto the local machine, where the Reducer is running. The individual key-value pairs are sorted by key into a larger data list. The data list groups the equivalent keys together so that their values can be iterated easily in the Reducer task.

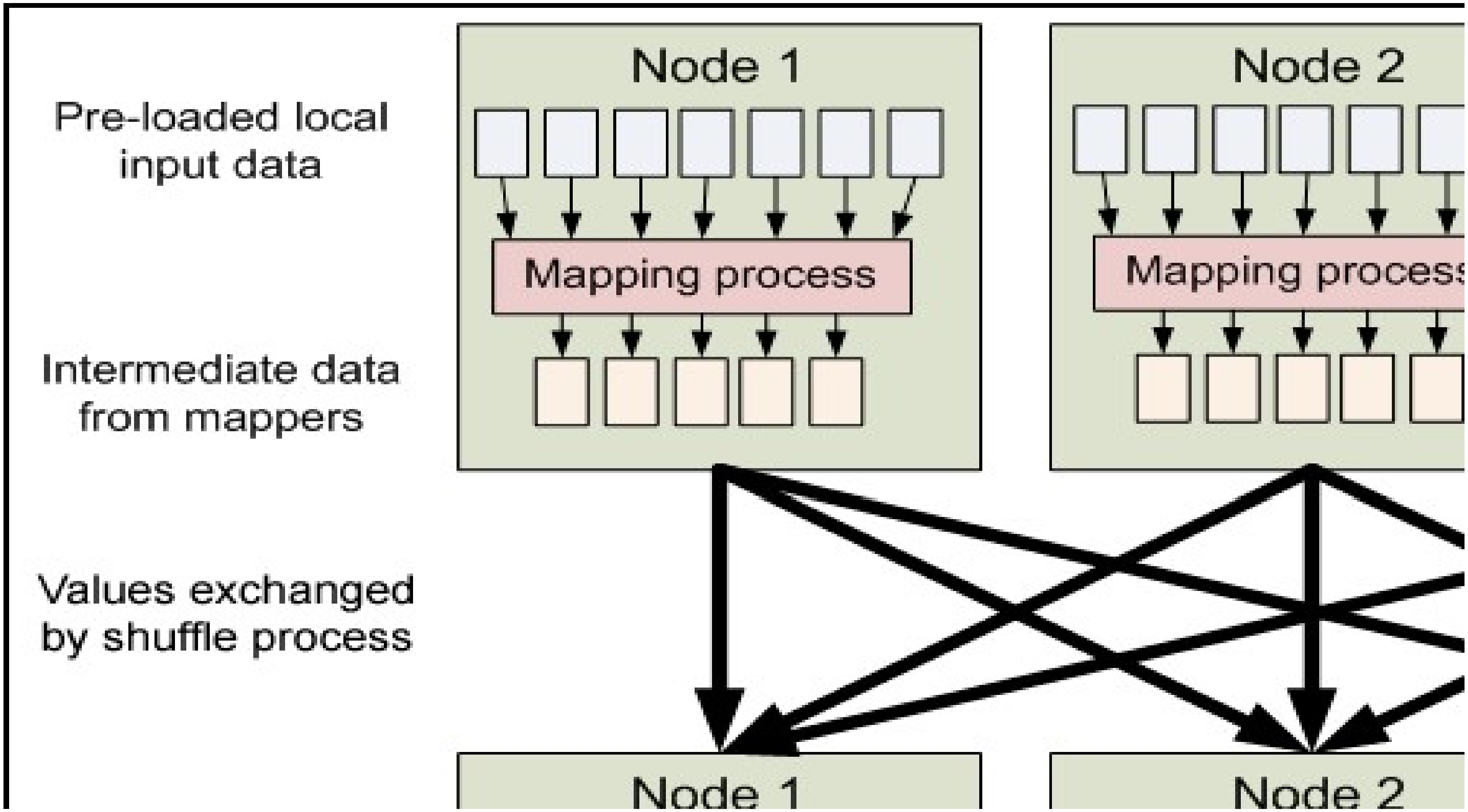
Reducer – The Reducer takes the grouped key-value paired data as input and runs a Reducer function on each one of them. Here, the data can be aggregated, filtered, and combined in a number of ways, and it requires a wide range of processing. Once the execution is over, it gives zero or more key-value pairs to the final step.

Output Phase – In the output phase, we have an output formatter that translates the final key-value pairs from the Reducer function and writes them onto a file using a record writer.

Data Distribution

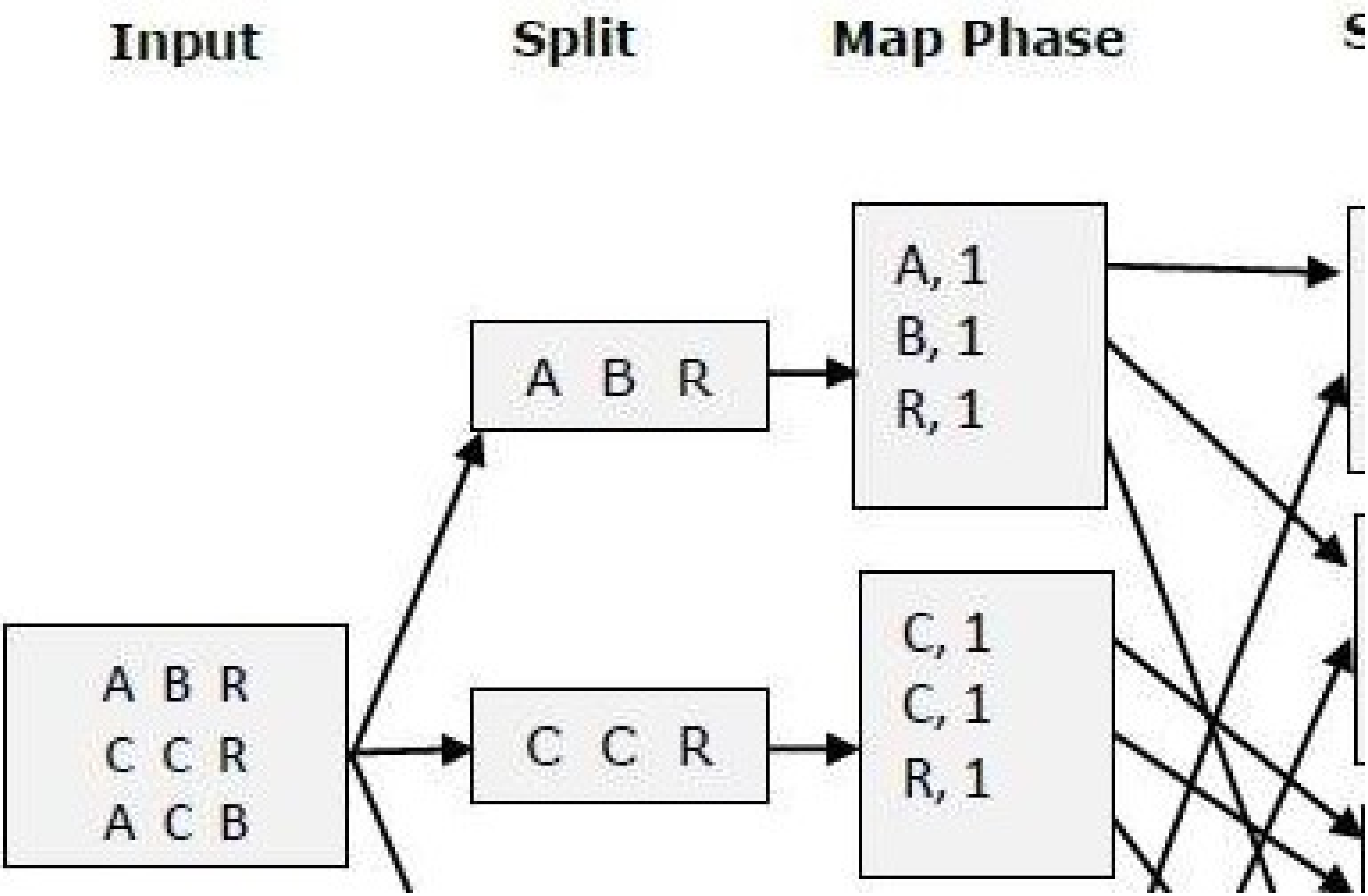


MapReduce: Isolated Processes



In MapReduce, records are processed in isolation by tasks called *Mappers*. The output from the Mappers is then brought together into a second set of tasks called *Reducers*, where results from different mappers can be merged together.

Simple Example



Terms

Shuffle and Sort – The Reducer task starts with the Shuffle and Sort step. It downloads the grouped key-value pairs onto the local machine, where the Reducer is running. The individual key-value pairs are sorted by key into a larger data list. The data list groups the equivalent keys together so that their values can be iterated easily in the Reducer task.

Reducer – The Reducer takes the grouped key-value paired data as input and runs a Reducer function on each one of them. Here, the data can be aggregated, filtered, and combined in a number of ways, and it requires a wide range of processing. Once the execution is over, it gives zero or more key-value pairs to the final step.

Output Phase – In the output phase, we have an output formatter that translates the final key-value pairs from the Reducer function and writes them onto a file using a record writer.

Programming 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.

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.

Programming Terminology

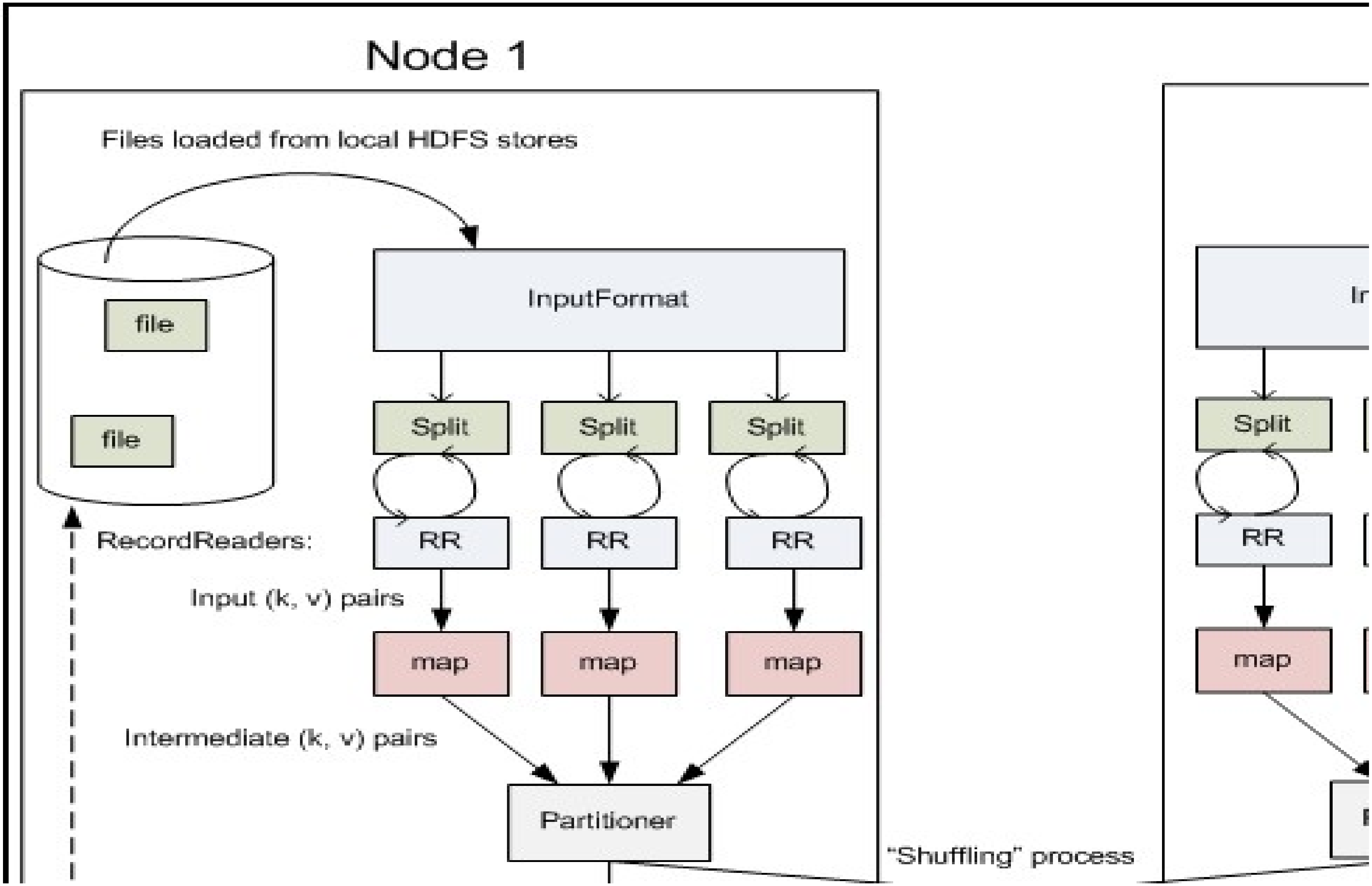
Task Tracker - Tracks the task and reports status to JobTracker.

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.

Data Flow



Node 1
Files loaded from local HDFS stores
file
file
InputFormat
Split
Split
Split
RecordReaders:
RR
RR
RR
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Partitioner
"Shuffling" process
Split
RR
map

Data Flow

Input files: This is where the data for a MapReduce task is initially stored.

- The format of these files is arbitrary.
- It is typical to be very large -- tens of gigabytes or more.
- **InputFormat:** It defines how these input files are split up and read.
- InputFormat is a class that provides the following functionality:
  - Selects the files or other objects that should be used for input
  - Defines the *InputSplits* that break a file into tasks
  - Provides a factory for *RecordReader* objects that read the file
- Several InputFormats are provided with Hadoop.
- An abstract type is called *FileInputFormat*.
- all InputFormats that operate on files inherit functionality and properties from this class.
- When starting a Hadoop job, FileInputFormat is provided with a path containing files to read.

Data Flow

- The FileInputFormat will read all files in directory.
- Then it divides these files into one or more InputSplits each.
- We can choose which InputFormat to apply to our input files for a job by calling the setInputFormat() method of the *JobConf* object that defines the job.

The different InputFormat are:

- *TextInputFormat:* It is default format.
    - It treats each line of each input file as a separate record.
    - This is useful for unformatted data or line-based records like log files.
- *KeyValueInputFormat:* This format treats each line of input as a separate record.
- While the TextInputFormat treats the entire line as the value, the KeyValueInputFormat breaks the line itself into the key and value by searching for a tab character.

Data Flow

- This is particularly useful for reading the output of one MapReduce job as the input to another.
- *SequenceFileInputFormat:* Reads special binary files that are specific to Hadoop.
- These files include many features designed to allow data to be rapidly read into Hadoop mappers.
- Sequence files are block-compressed and provide direct serialization and deserialization of several arbitrary data types (not just text).
- Sequence files can be generated as the output of other MapReduce tasks and are an efficient intermediate representation for data that is passing from one MapReduce job to anther.

Data Flow

- **RecordReader:** The InputSplit has defined a slice of work, but does not describe how to access it.
- The *RecordReader* class actually loads the data from its source and converts it into (key, value) pairs suitable for reading by the Mapper.
- The RecordReader instance is defined by the InputFormat.
- The default InputFormat, *TextInputFormat*, provides a *LineRecordReader*, which treats each line of the input file as a new value.
- The key associated with each line is its byte offset in the file.
- The RecordReader is invoke repeatedly on the input until the entire InputSplit has been consumed.
- Each invocation of the RecordReader leads to another call to the map() method of the Mapper.

BASIC OPERATIONS

- Splits
- Mapper
- Portioning
- Sort
- shuffle
- Combiner
- Reducer

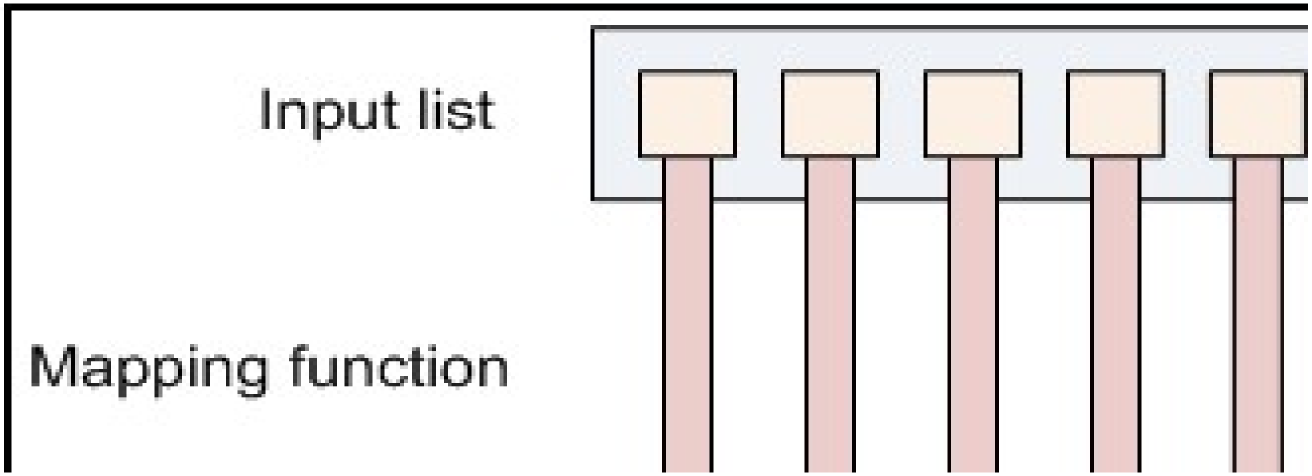
- **Splits:** Hadoop divides the input to a MapReduce job into fixed-size pieces called input splits, or just splits.
- Hadoop creates one map task for each split, which runs the userdefined map function for each record in the split.
- So if we are processing the splits in parallel, the processing is better load-balanced if the splits are small.

BASIC OPERATIONS

- An InputSplit describes a unit of work that comprises a single map task in a MapReduce program.
- A MapReduce program applied to a data set, collectively referred to as a Job, is made up of several (possibly several hundred) tasks.
- Map tasks may involve reading a whole file, they involve reading only part of a file.

BASIC OPERATIONS

- **Mapper:** The first phase of a MapReduce program is called *mapping*.
- A list of data elements are provided, one at a time, to a function called the *Mapper*, which transforms each element individually to an output data element.



BASIC OPERATIONS

- The Mapper performs the user-defined work of the first phase of the MapReduce program.
- Given a key and a value, the map() method emits (key, value) pair(s) which are forwarded to the Reducers.
- A new instance of Mapper is instantiated in a separate Java process for each map task (InputSplit) that makes up part of the total job input.
- The individual mappers are intentionally not provided with a mechanism to communicate with one another in any way.
- OutputCollector class
    - collect()
- Reporter class
    - getInputSplit()
    - setStatus()
    - incrCounter()

BASIC OPERATIONS

- **Partitioner:** A partitioner works like a condition in processing an input dataset.
- The partition phase takes place after the Map phase and before the Reduce phase.
- The number of partitioners is equal to the number of reducers.
- That means a partitioner will divide the data according to the number of reducers.
- Therefore, the data passed from a single partitioner is processed by a single Reducer.
- A partitioner partitions the key-value pairs of intermediate Map-outputs.
- It partitions the data using a user-defined condition, which works like a hash function.
- The total number of partitions is same as the number of Reducer tasks for the job.

BASIC OPERATIONS

- *Partitioner* interface
- Syntax:

```
public interface Partitioner<K, V> extends JobConfigurable
{
int getPartition(K key, V value, int numPartitions);
}
```

- *HashPartitioner* (default partitioner)
  - hashCode() modulo the number of partitions total to determine which partition to send a given (key, value) pair to.

BASIC OPERATIONS

•**Shuffle:**

• **Sort:** **Context** class (user-defined class) collects the matching valued keys as a collection.

RawComparator class to sort the key-value pairs for similar key-value pair.

Example: (K2, {V2, V2, …})

Combiner: The Combiner is **a "mini-reduce" process which operates only on data generated by one machine.**

•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.

•A combiner does not have a predefined interface and it must implement the Reducer interface's reduce() method.

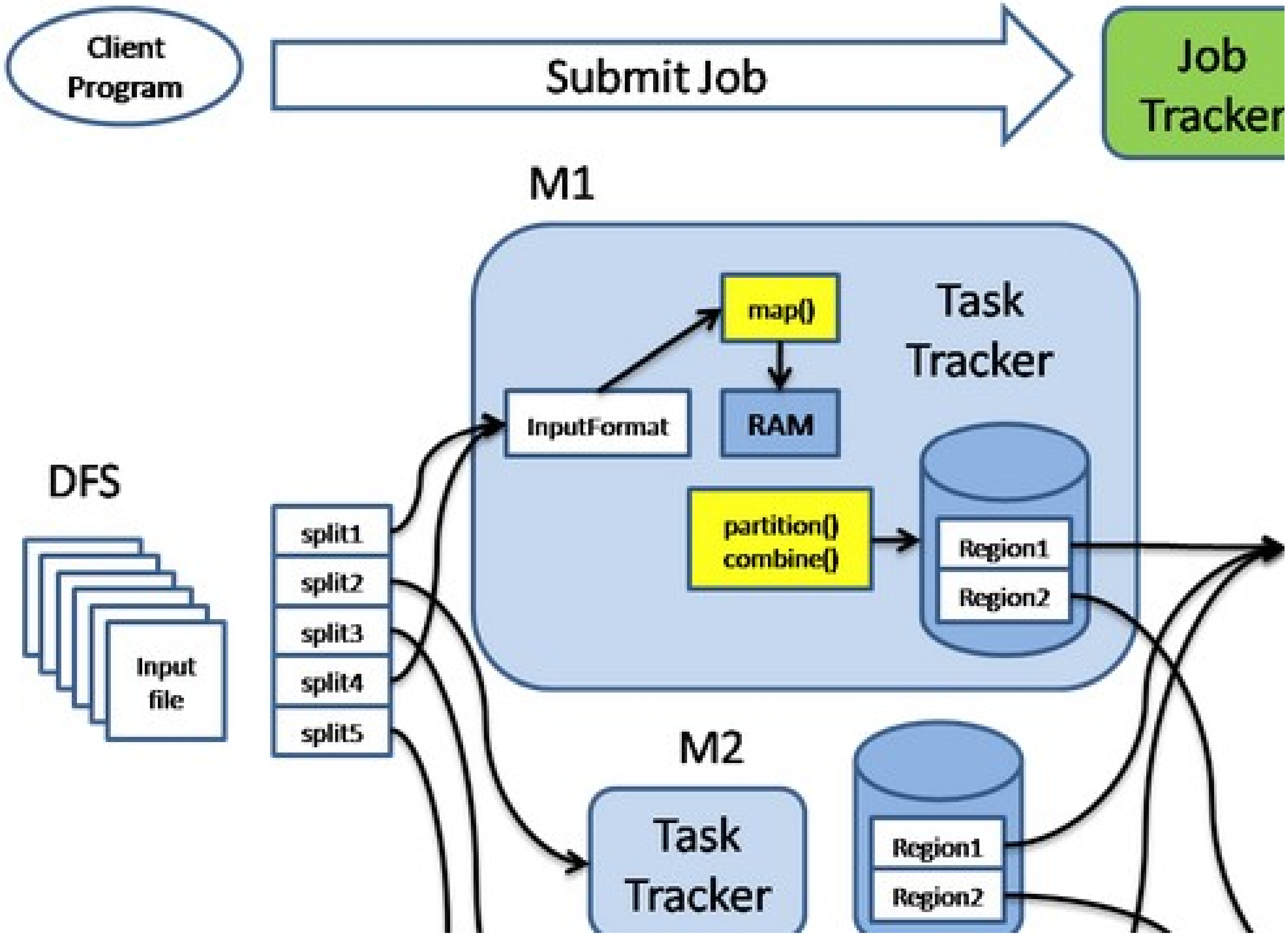
conf.setCombinerClass(Reduce.class);

BASIC OPERATIONS

- **Reduce:** For each key in the partition assigned to a Reducer.
    - *Reducer* class
    - *JobContext.getConfiguration()* is used to configure
    - *reduce()*

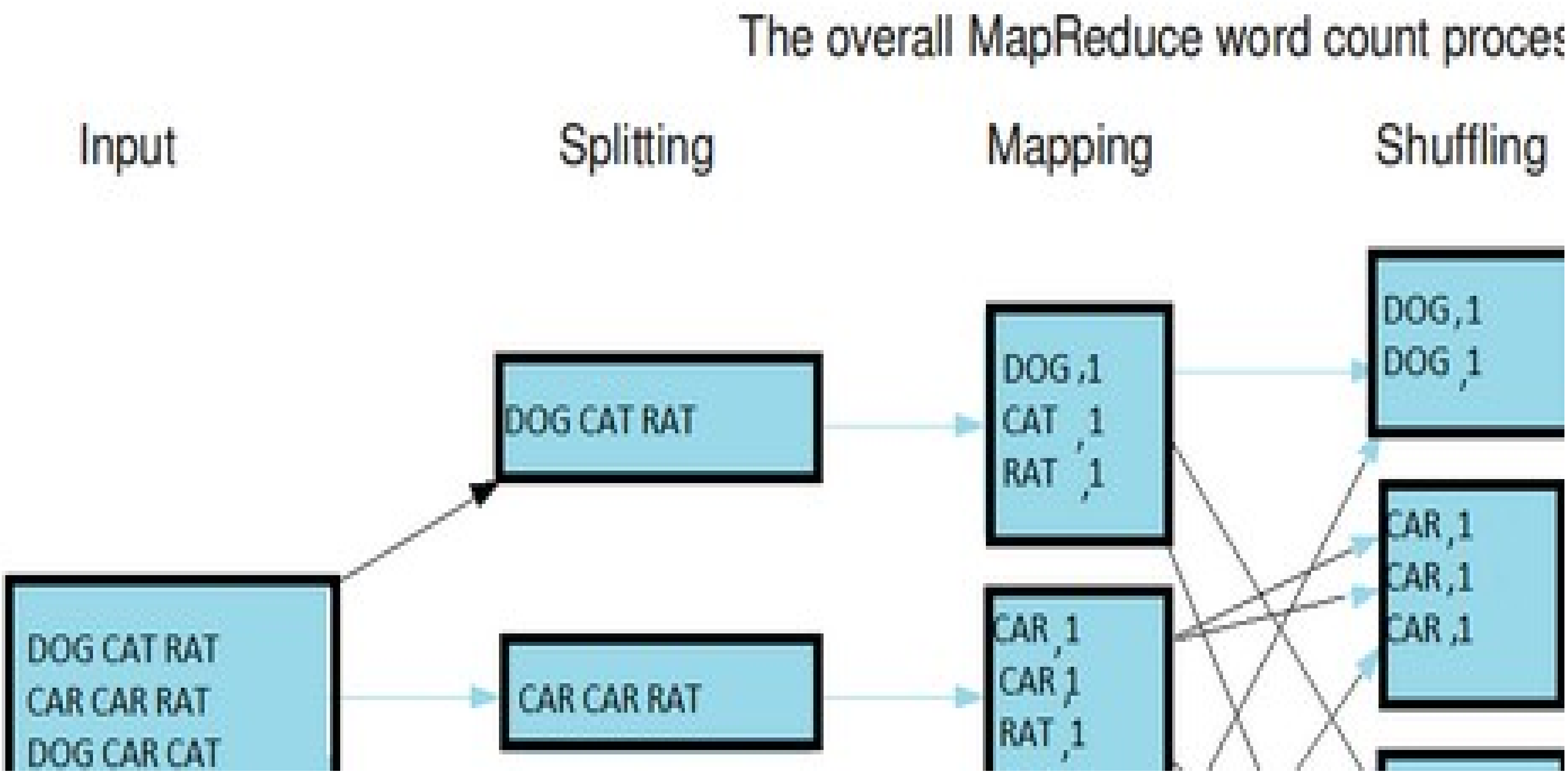
    reduce(KEYIN key, Iterable<VALUEIN> values, org.apache.hadoop.mapreduce.Reducer.Context context)

How MapReduce works



Word count Example

Count the number of words in the file.



Mapreduce life Cycle

Job Submission

The submit() method on job creates an internal instance of JobSubmitter and calls submitJobInternal() method on it.

•waitForCompletion() polls the job's progress once a second after submitting the job.

On calling this method following happens.

•It goes to JobTracker and gets a jobId for the job

•Perform checks if the the output directory has been specified or not.

• If specified , whether the directory already exists or is new.

•And throws error if any such thing fails.

•Computes input split and throws error if it fails to do so, because the input paths don't exist.

•Copies the resources to JobTracker file system in a directory named after Job Id.

•These resources include configuration files, job jar file,and computed input splits.

Finally it calls submitJob() method on JobTracker.

Mapreduce life Cycle

Job Initialization

Job tracker performs following steps

•Creates book keeping object to track tasks and their progress

For each input split creates a map tasks.

•The number of reduce tasks is defined by the configuration mapred.reduce.tasks set by setNumReduceTasks().

•Tasks are assigned taskId's at this point.

•In addition to this 2 other tasks are created: Job initialization task and Job clean up task, and are run by tasktrackers

•FileOutputCommitter creates the output directory to store the tasks output as well as temporary output

•Job clean up tasks which delete the temporary directory after the job is complete.

Mapreduce life Cycle

Task Assignment

TaskTracker sends a heartbeat to jobtracker every five seconds.

It will indicate whether it is ready to run a new task. They also send the available slots on them.

Here is how job allocation takes place.

•JobTracker first selects a job to select the task from, based on job scheduling algorithms.

•The default scheduler fills empty map task before reduce task slots.

•For a map task then it chooses a tasktracker which is in following order of priority: data-local, rack-local and then network.

•For a reduce task, it simply chooses a task tracker which has empty slots.

Mapreduce life Cycle

Task Execution:

•TaskTracker copies the job jar file from the shared filesystem (HDFS).

• Tasktracker creates a local working directory, and un-jars the jar file into the local file system.

• It then creates an instance of TaskRunner

Mapreduce life Cycle

Action:

Tasktracker starts TaskRunner in a new JVM to run the map or reduce task.

•Seperate process is needed so that the TaskTracker does not crash in case of bug in user code or JVM.

•The child process communicates it progress to parent process .

•Each task can perform setup and cleanup actions, which are run in the same JVM as the task itself, based on OutputComitter.

Mapreduce life Cycle

Job/Task Progress

Here is how the progress is monitored of a job/task

- JobClient keeps polling the JobTracker for progress.
- Each child process reports its progress to parent task tracker.
- If a task reports progress, it sets a flag to indicate that the status change should be sent to the tasktracker.
- The flag is checked in a separate thread every 3 seconds, and if set it notifies the tasktracker of the current task status.
- Task tracker sends its progress to JobTracker over the heartbeat for every five seconds.
- JobTracker then assembles task progress from all task trackers and keeps a view of job.
- The Job receives the latest status by polling the jobtracker every second.

Mapreduce life Cycle

Job Completion

On Job Completion the clean up task is run.

- Task sends the task tracker job completion. Which is sent to job tracker.
- Job Tracker then send the job completion message to client.
- Jobtracker cleans up its working state for the job and instructs tasktrackers to do the same, It cleans up all the temporary directories.
- This causes jobclient's waitForJobToComplete() method to return.

Mapreduce Program

```
package hadoop;
 import java.util.*;
import java.io.IOException;
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
 import org.apache.hadoop.io.*;
 import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;
```

Mapreduce Program

```
public class ProcessUnits
{
public static class E_EMapper extends MapReduceBase implements
Mapper<LongWritable,                                        /*Input key Type */
Text,                                                               /*Input value Type*/
Text,                                                               /*Output key Type*/
IntWritable>                                                        /*Output value Type*/
{
//Map function
public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
Reporter reporter) throws IOException
{
 String line = value.toString();
String lasttoken = null;
StringTokenizer s = new StringTokenizer(line,"\t");
String year = s.nextToken();
while(s.hasMoreTokens())
{
lasttoken=s.nextToken();
}
```

Mapreduce Program

```
int avgprice = Integer.parseInt(lasttoken);
output.collect(new Text(year), new IntWritable(avgprice));
}
}
```

Mapreduce Program

Reducer class

```
public static class E_EReduce extends MapReduceBase implements
 Reducer< Text, IntWritable, Text, IntWritable >
{
//Reduce function
 public void reduce(Text key, Iterator <IntWritable> values, OutputCollector>Text,
IntWritable> output, Reporter reporter) throws IOException
{
 int maxavg=30;
int val=Integer.MIN_VALUE;
while (values.hasNext())
{
 if((val=values.next().get())>maxavg)
 {
 output.collect(key, new IntWritable(val));
 }
 }
 }
 }
```

Mapreduce Program

```
//Main function
public static void main(String args[])throws Exception
{
JobConf conf = new JobConf(Eleunits.class);
conf.setJobName("max_eletricityunits");
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);
conf.setMapperClass(E_EMapper.class);
 conf.setCombinerClass(E_EReduce.class);
conf.setReducerClass(E_EReduce.class);
conf.setInputFormat(TextInputFormat.class);
conf.setOutputFormat(TextOutputFormat.class);
FileInputFormat.setInputPaths(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));
 JobClient.runJob(conf);
}
 }
```

Mapreduce Program

1979	23	23	2	43	24	25	26	26
1980	26	27	28	28	28	30	31	31
1981	31	32	32	32	33	34	35	36

The above data is saved as **sample.txt**

Mapreduce Program

Step-1: create a directory to store the compiled java classes.

```
$ mkdir unit
```

Step-2: Download Hadoop-core-1.2.1.jar, which is used to compile and execute the MapReduce program.

Step 3: compile the **ProcessUnits.java** program and to create a jar for the program.

```
$ javac -classpath hadoop-core-1.2.1.jar -d units ProcessUnits.java
$ jar -cvf units.jar -C units/
```

Step 4: create an input directory in HDFS.

```
$HADOOP_HOME/bin/hadoop fs -mkdir input_dir
```

Step 5: copy the input file named **sample.txt** in the input directory of HDFS.

```
$HADOOP_HOME/bin/hadoop fs -put /home/hadoop/sample.txt input_dir
```

Step 6: verify the files in the input directory

```
$HADOOP_HOME/bin/hadoop fs -ls input_dir/
```

Step 7: run the application

```
$HADOOP_HOME/bin/hadoop jar units.jar hadoop.ProcessUnits input_dir output_dir
```

Mapreduce Program

```
INFO mapreduce.Job: Job job_1414748220717_0002
completed successfully
14/10/31 06:02:52
INFO mapreduce.Job: Counters: 49

File System Counters

    FILE: Number of bytes read=61
    FILE: Number of bytes written=279400
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0

    HDFS: Number of bytes read=546
    HDFS: Number of bytes written=40
    HDFS: Number of read operations=9
    HDFS: Number of large read operations=0
    HDFS: Number of write operations=2 Job Counters
```

Mapreduce Program

Step-8: verify the resultant files in the output folder.

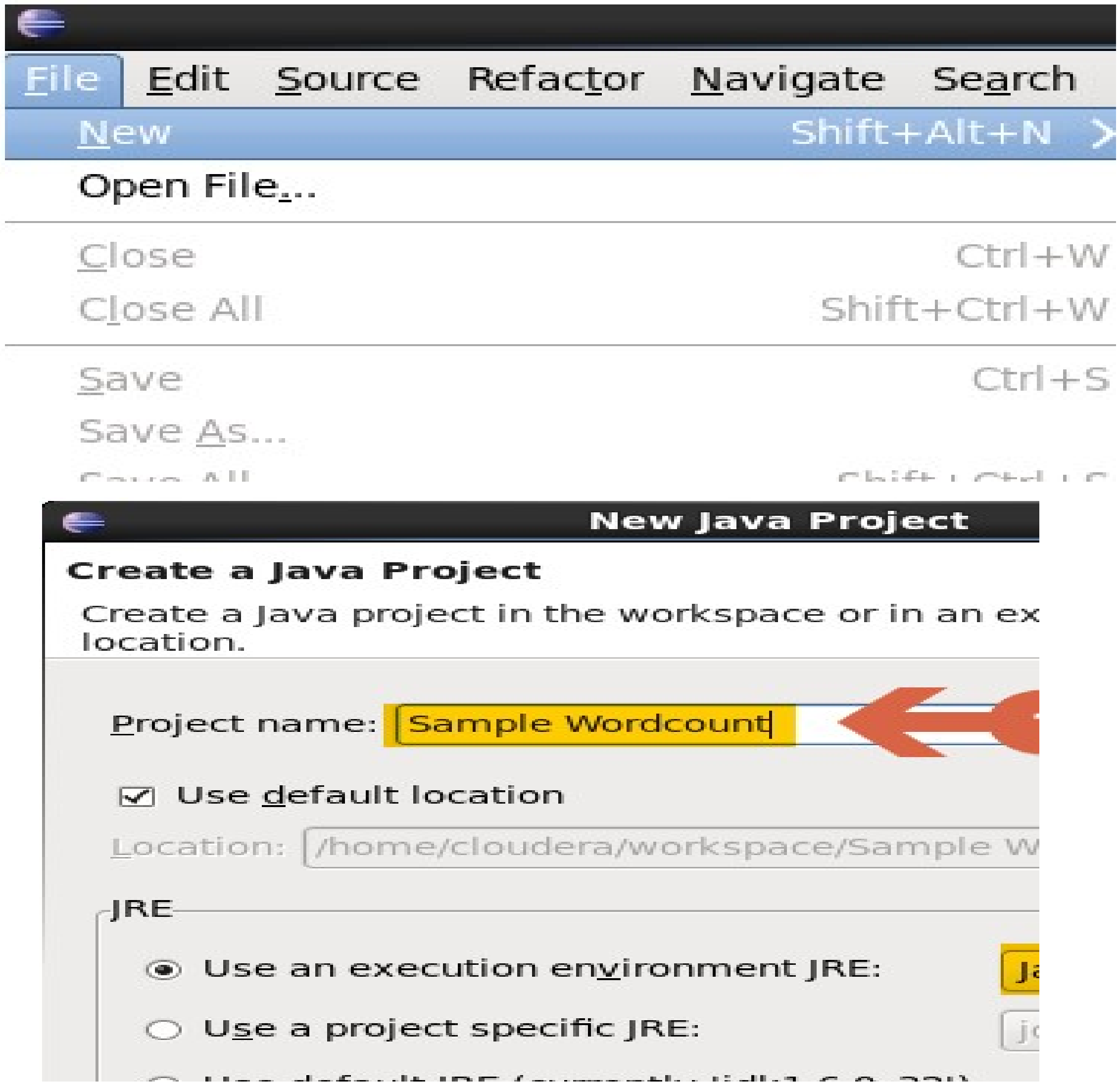
```
$HADOOP_HOME/bin/hadoop fs -ls output_dir/
```

Step-9: see the output

```
$HADOOP_HOME/bin/hadoop fs -cat output_dir
```

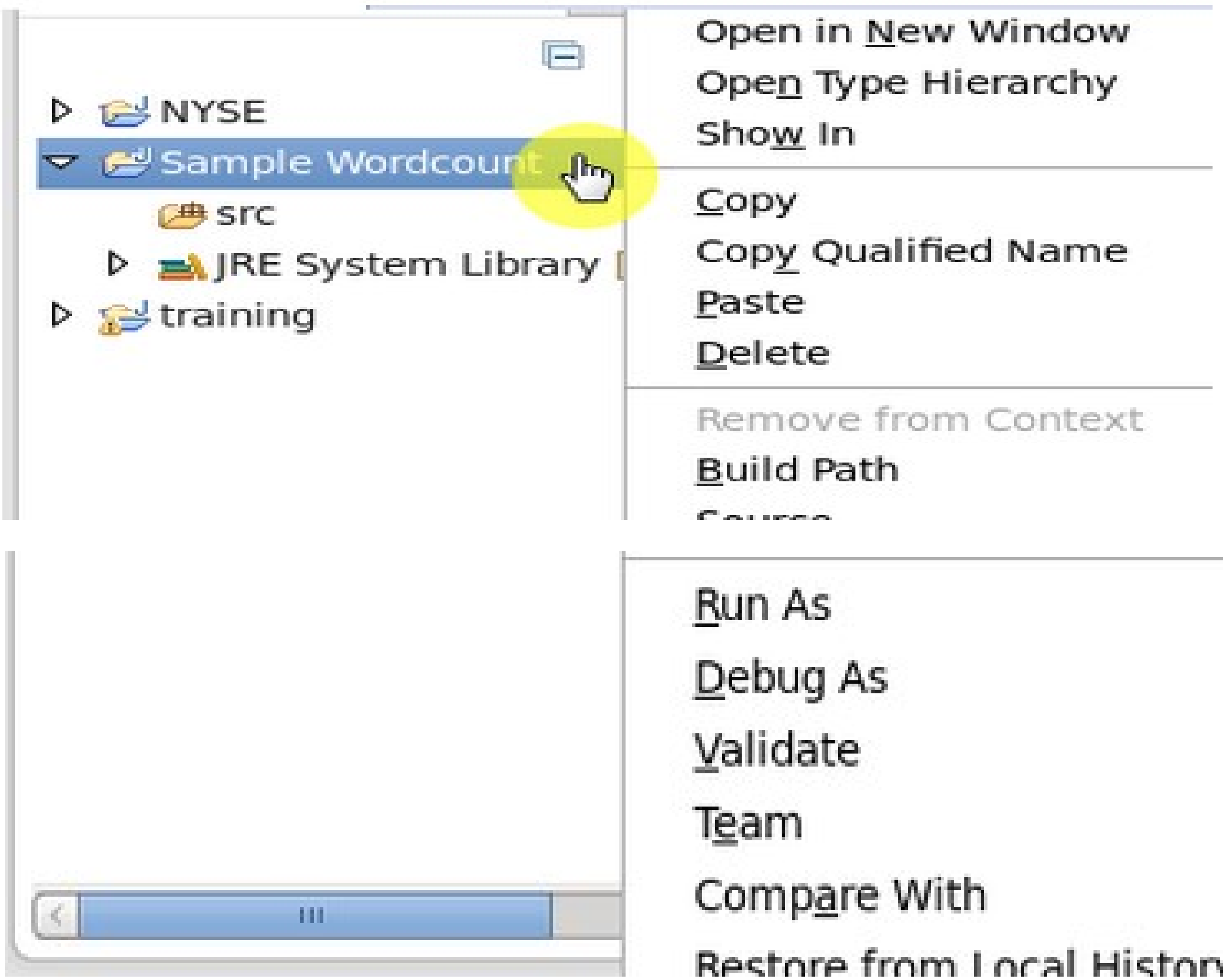
Wordcount Program

Step-1: Open Eclipse> File > New > Java Project >(Name it – Sample Wordcount) > Finish

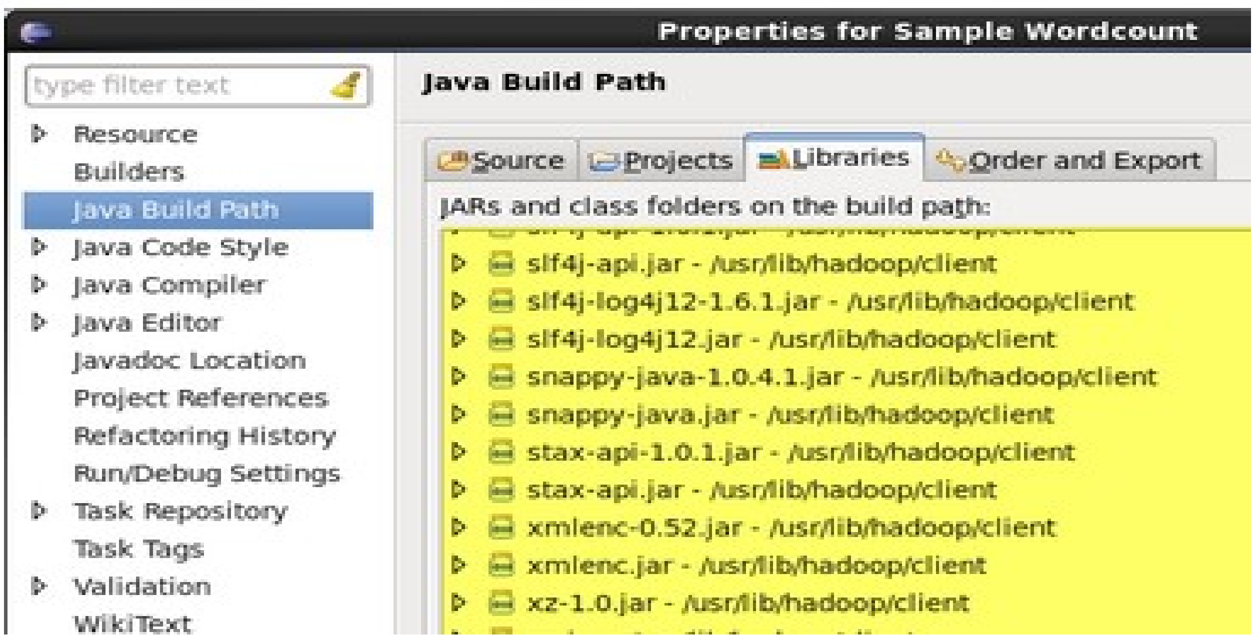


Wordcount Program

Step-2: Get references to hadoop libraries by clicking on Add JARS

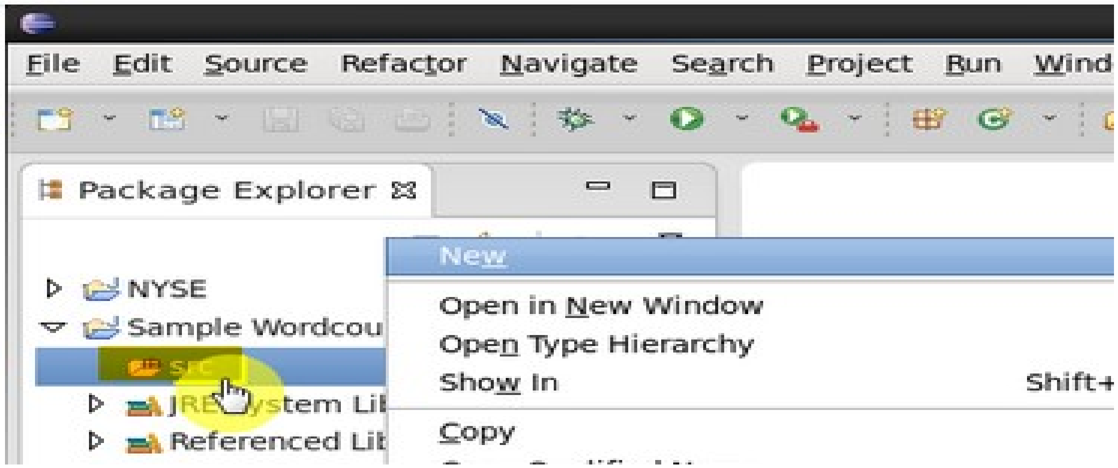


Wordcount Program



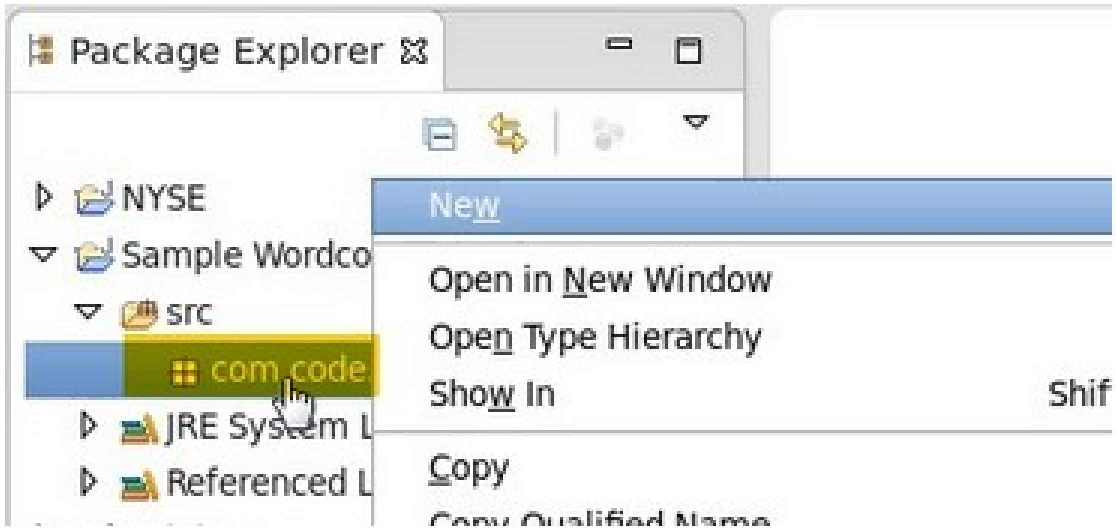
Wordcount Program

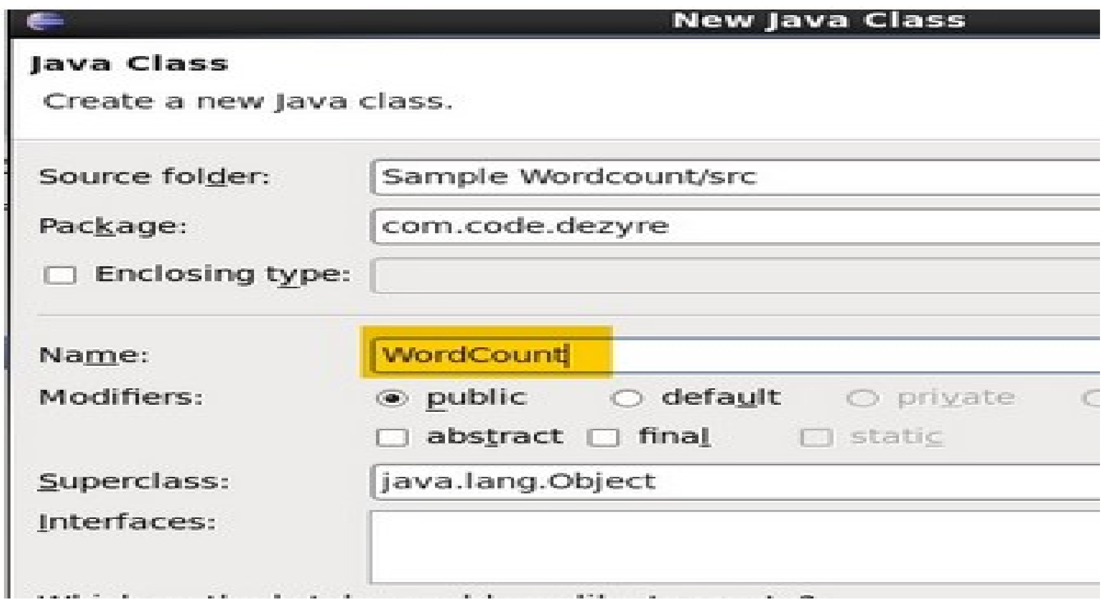
Step-3: Create a new package within the project Right Click > New > Package (Name it) > Finish



Wordcount Program

Step-4: Create a a WordCount class under the project



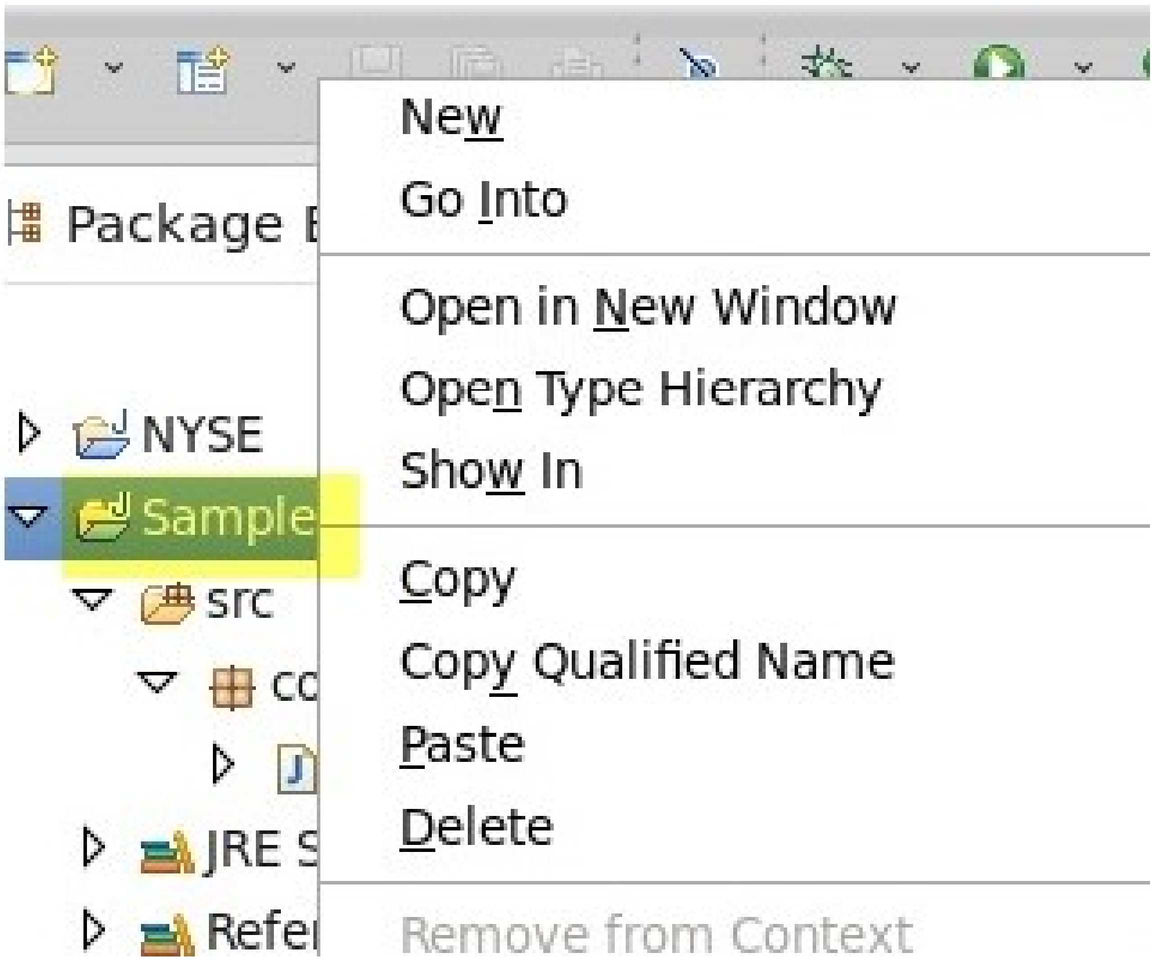


Wordcount Program

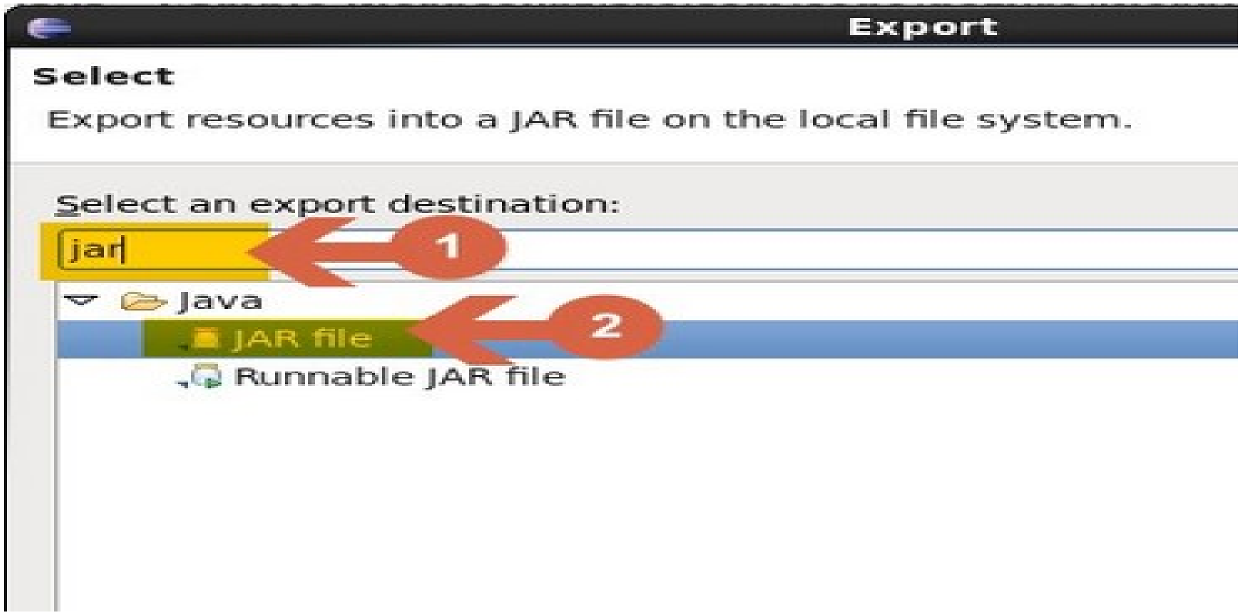
Step-5: Write mapper code, reducer code and configuration code

Wordcount Program

Step-6: Create the JAR file for the wordcount class



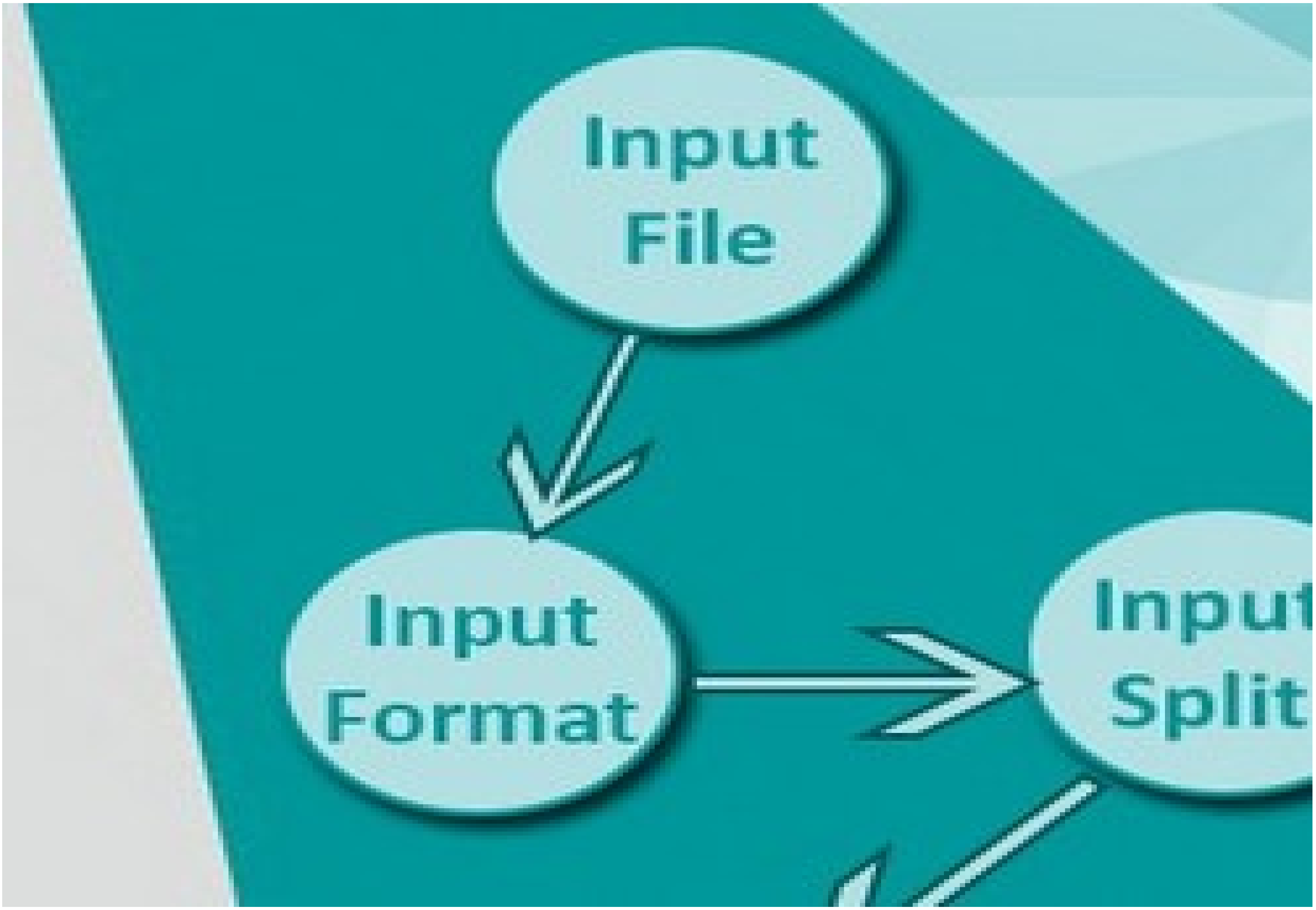
Wordcount Program



Wordcount Program

Hadoop InputOutput Format

Input Format



Hadoop InputOutput Format

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.

- **FileInputFormat in Hadoop**
- **TextInputFormat**
- **KeyValueTextInputFormat**
- **SequenceFileInputFormat**
- **SequenceFileAsTextInputFormat**
- **SequenceFileAsBinaryInputFormat**
- **NLineInputFormat**
- **DBInputFormat.**

How we get the data to mapper?

2 methods used to get the data to mapper in MapReduce:

- getsplits()
- createRecordReader()

```
public abstract class InputFormat<K, V>
{
public abstract List<InputSplit> getSplits(JobContext context)
throws IOException, InterruptedException;
public abstract RecordReader<K, V>
createRecordReader(InputSplit split,
TaskAttemptContext context) throws IOException,
InterruptedException;
}
```

Hadoop Input Format

FileInputFormat in Hadoop: FileInputFormat in Hadoop is the base class for all file-based InputFormats.

•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.

•**TextInputFormat:** TextInputFormat ion Hadoop is the default InputFormat of MapReduce.

•TextInputFormat treats each line of each input file as a separate record and performs no parsing.

KeyValueTextInputFormat: KeyValueTextInputFormat in Hadoop is similar to TextInputFormat as it also treats each line of input as a separate record.

•While TextInputFormat treats entire line as the value, but the KeyValueTextInputFormat breaks the line itself into key and value by a tab character (‘/t’).

SequenceFileInputFormat: SequenceFileInputFormat in Hadoop is an InputFormat which reads sequence files.

•Sequence files are binary files that stores sequences of binary key-value pairs.

•Sequence files are block-compressed and provide direct serialization and deserialization of several arbitrary data types (not just text).

•Key & Value both are user-defined.

Hadoop Input Format

SequenceFileAsTextInputFormat: SequenceFileAsTextInputFormat in Hadoop is another form of SequenceFileInputFormat which converts the sequence file key values to Text objects.

•By calling 'tostring()' conversion is performed on the keys and values.

•This InputFormat makes sequence files suitable input for streaming.

SequenceFileAsBinaryInputFormat: SequenceFileAsBinaryInputFormat in Hadoop is a SequenceFileInputFormat using which we can extract the sequence file's keys and values as a binary object.

NLineInputFormat: It is another form of TextInputFormat where the keys are byte offset of the line and values are contents of the line.

•Each mapper receives a variable number of lines of input with TextInputFormat and KeyValueTextInputFormat.

•The number depends on the size of the split and the length of the lines.

•And if we want our mapper to receive a fixed number of lines of input, then we use NLineInputFormat.

Example: N is the number of lines of input that each mapper receives. By default (N=1), each mapper receives exactly one line of input.

• If N=2, then each split contains two lines. One mapper will receive the first two Key-Value pairs and another mapper will receive the second two key-value pairs.

Hadoop Input Format

DBInputFormat: DBInputFormat in Hadoop is an InputFormat that reads data from a relational database, using JDBC.

•it doesn’t have portioning capabilities.

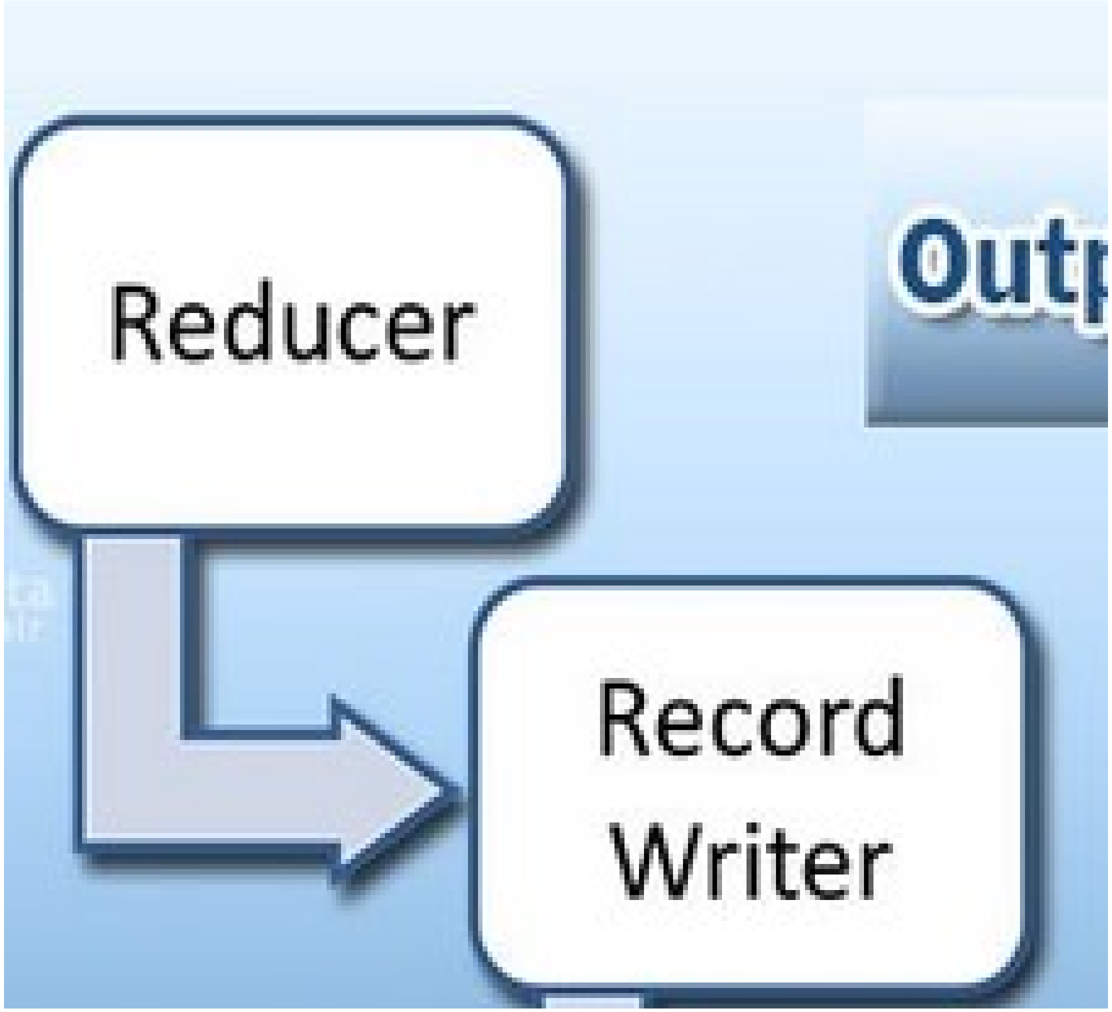
• so we need to careful to read to many mappers.

•joining is used to make datasets from HDFS using MultipleInputs.

•Here Key is LongWritables while Value is DBWritables.

Hadoop InputOutput Format

Output Format



Hadoop Output Format

OutputFormat is used to write to files on the local disk or in HDFS.

•OutputFormat describes the output-specification for a Map-Reduce job.

•Based on output specification following things happened:

•MapReduce job checks that the output directory.

•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.

Types of Hadoop Output Format

- textOutputFormat,
- sequenceFileOutputFormat,
- mapFileOutputFormat,
- sequenceFileAsBinaryOutputFormat,
- DBOutputFormat,
- LazyOutputForma, and
- MultipleOutputs

TextOutputFormat:

- MapReduce default OutputFormat is TextOutputFormat, it writes (key, value) pairs on individual lines of text files
- The keys and values can be of any type.
- TextOutputFormat turns it to string by calling toString().
- Each key-value pair is separated by a tab character.
- It can be changed using MapReduce.output.textoutputformat.separator property.
- KeyValueTextOutputFormat is used for reading output text files since it breaks lines into key-value pairs based on a configurable separator.

MapReduce Join

•It is used to achieve larger dataset.

•Joining of two datasets begin by comparing size of each dataset.

•If one dataset is smaller as compared to the other dataset then smaller dataset is distributed to every datanode in the cluster.

•Once it is distributed, either Mapper or Reducer uses smaller dataset to perform lookup for matching records from large dataset and then combine those records to form output records.

Map-side join - When the join is performed by the mapper, it is called as map-side join.

• In this type, the join is performed before data is actually consumed by the map function.

• It is mandatory that the input to each map is in the form of a partition and is in sorted order.

•Also, there must be an equal number of partitions and it must be sorted by the join key.

Reduce-side join - When the join is performed by the reducer, it is called as reduce-side join.

•There is no necessity in this join to have dataset in a structured form (or partitioned).

• Here, map side processing emits join key and corresponding tuples of both the tables.

• As an effect of this processing, all the tuples with same join key fall into the same reducer which then joins the records with same join key.

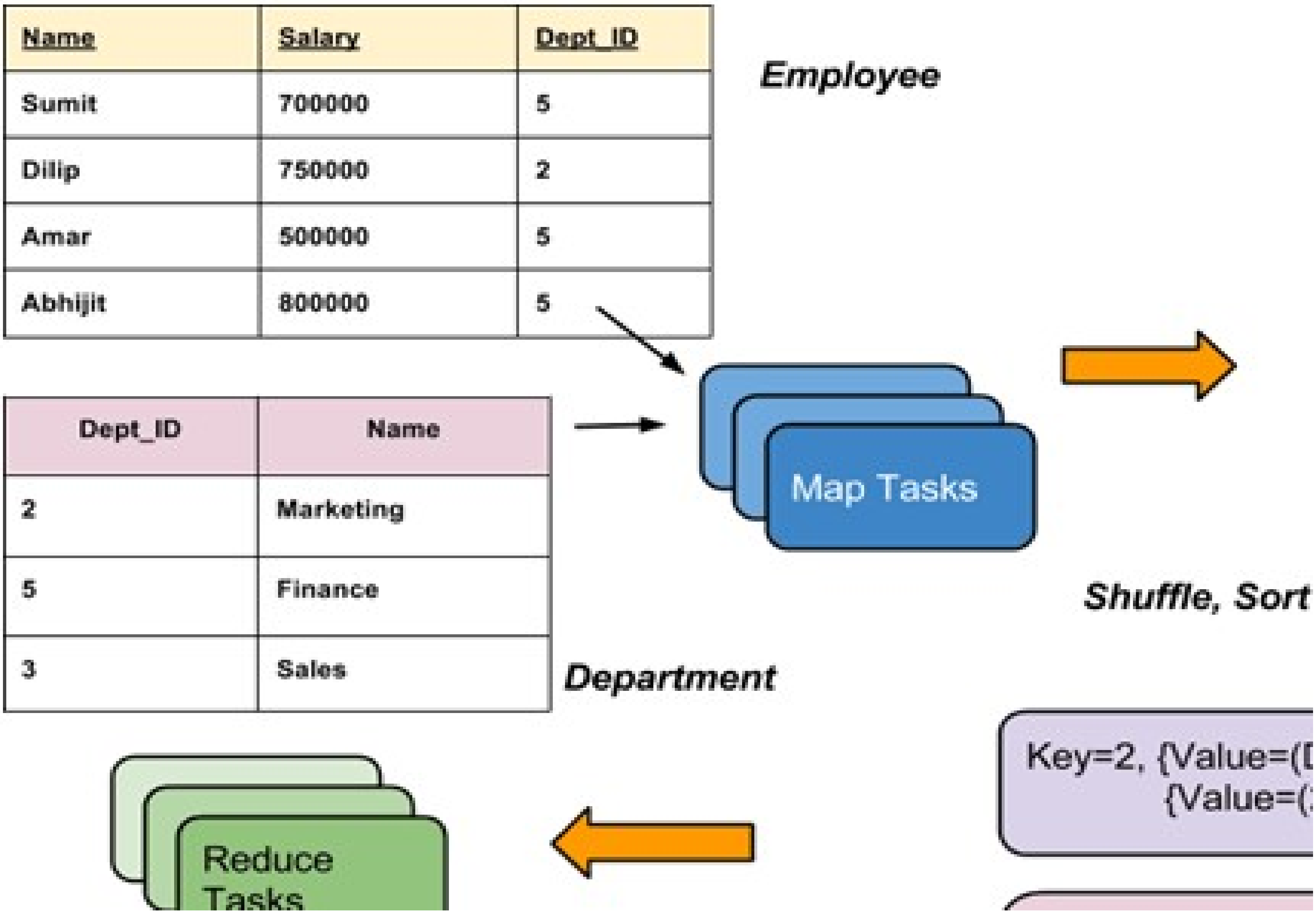
MapReduce Join Example

Name	Salary	Dept_I
Sumit	700000	5
Dilip	750000	2

Dept_ID	
2	Marketi
5	Financ

Department

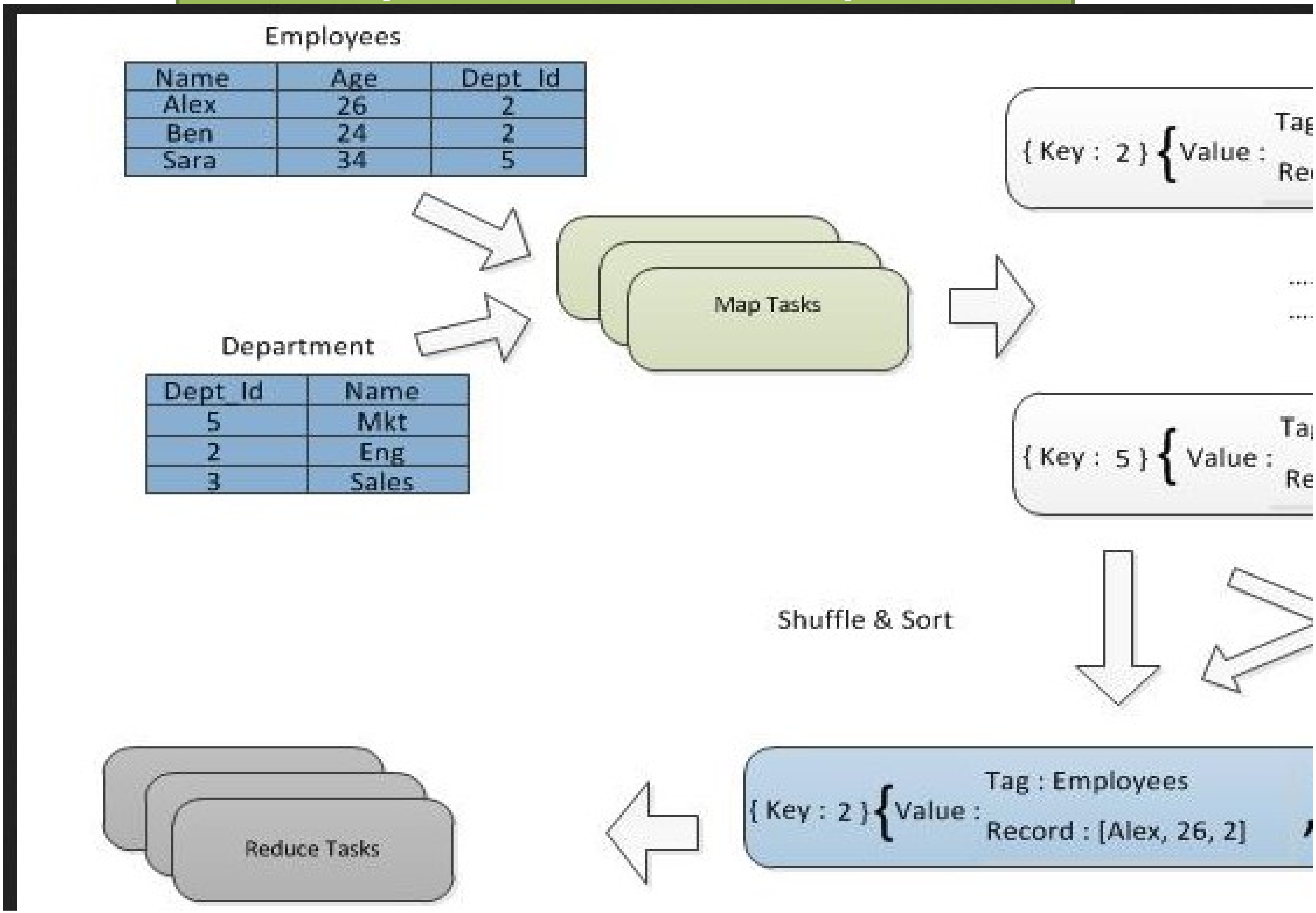
MapReduce Join Example-1



Name	Salary	Dept_ID
Sumit	700000	5
Dilip	750000	2
Amar	500000	5
Abhijit	800000	5

Dept_ID	Name
2	Marketing
5	Finance
3	Sales

MapReduce Join Example-2



pseudo code

```
map (K table, V rec)
 {

   dept_id = rec.Dept_Id

   tagged_rec.tag = table

   tagged_rec.rec = rec

   emit(dept_id, tagged_rec)

}
```

```
reduce (K dept_id, list<tagged_rec> tagged_recs)
{

 for (tagged_rec : tagged_recs)
{

 for (tagged_rec1 : taagged_recs)
 {

 if (tagged_rec.tag != tagged_rec1.tag)
{

 joined_rec = join(tagged_rec, tagged_rec1)

 }
 emit (tagged_rec.rec.Dept_Id, joined_rec)

 }
}
```

NOW TIME FOR U !!!!!

1: How Hadoop is different from Data-mining and Data-ware Housing?

2:Explain with diagram, How Mapreduce works?

3:Explain Streaming and pipe.

4:What is counter? Explain different types of counter.

5:Briefly explain different types of sorting used in mapreduce.

THANKS