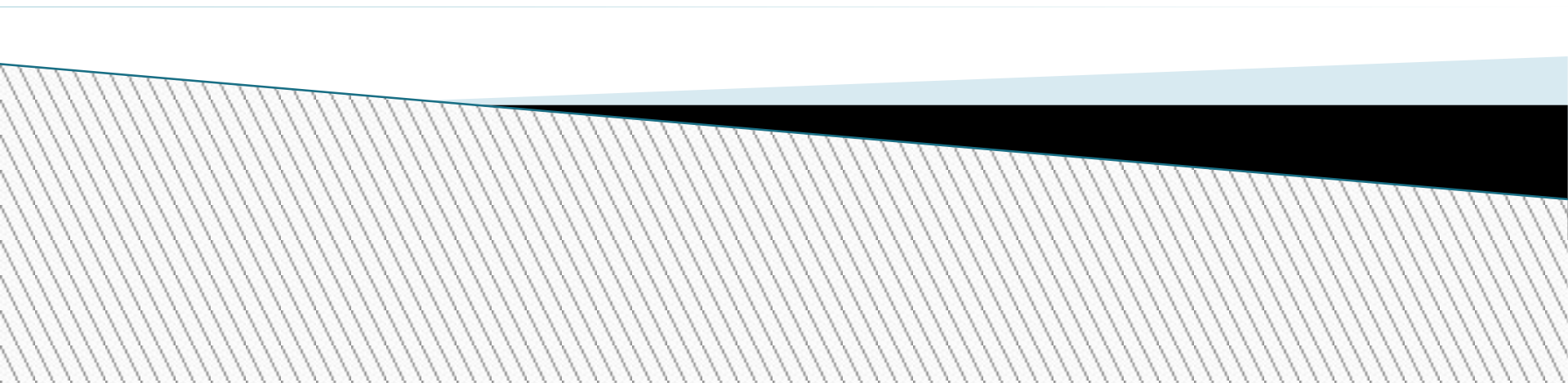


Apache Hadoop Training

–Sasidhar Sharma
Cloudera Certified Hadoop Professional
sasis937@gmail.com

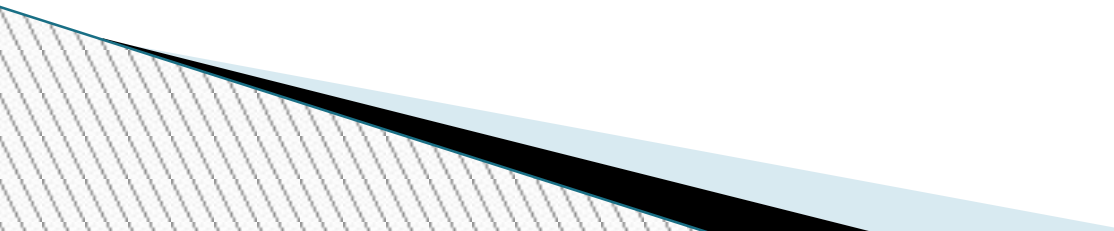
Module 1 – Motivation & Basics



MapReduce Framework

» » ▶ Module 1

In this module you will learn

- ▶ What is MapReduce job?
 - ▶ What is input split?
 - ▶ What is mapper?
 - ▶ What is reducer?
- 

What is MapReduce job?

- ▶ It's a framework for processing the data residing on HDFS
 - Distributes the task (map/reduce) across several machines
- ▶ Consist of typically 5 phases:
 - Map
 - Partitioning
 - Sorting
 - Shuffling
 - Reduce
- ▶ Mapper phase works typically on one block of data (`dfs.block.size`)

MapReduce Terminology

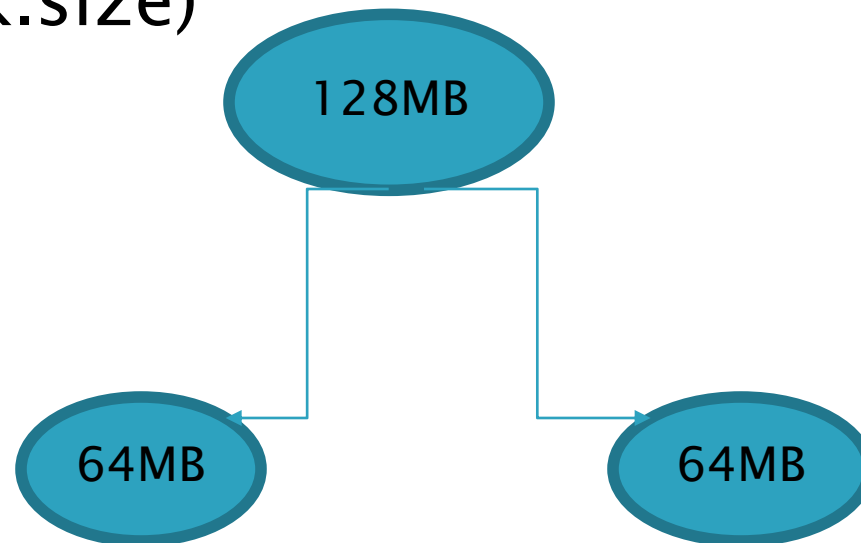
- ▶ What is job?
 - Complete execution of mapper and reducers over the entire data set
- ▶ What is task?
 - Single unit of execution (map or reduce task)
 - Execution of map or reduce over a portion of data typically a block of data (dfs.block.size)
- ▶ What is “*task attempt*”?
 - Instance of an attempt to execute a task (map or reduce task)
 - If task is failed working on particular portion of data, another task will run on that portion of data on that machine itself
 - If a task fails 4 times, then the task is marked as failed and entire job fails

Terminology continued

- ▶ How many tasks can run on portion of data?
 - Maximum 4
 - If “*speculative execution*” is ON, more task will run
- ▶ What is “*failed task*”?
 - Task can be failed due to exception, machine failure etc.
 - A failed task will be re-attempted again (4 times)
- ▶ What is “*killed task*”?
 - If task fails 4 times, then task is killed and entire job fails.
 - Task which runs as part of speculative execution will also be marked as killed

Input Split

- ▶ Portion or chunk of data on which mapper operates
- ▶ Typically is equal to one block of data (dfs.block.size)



Here there are 2 blocks, so there will be two input splits

Input Split cont'd

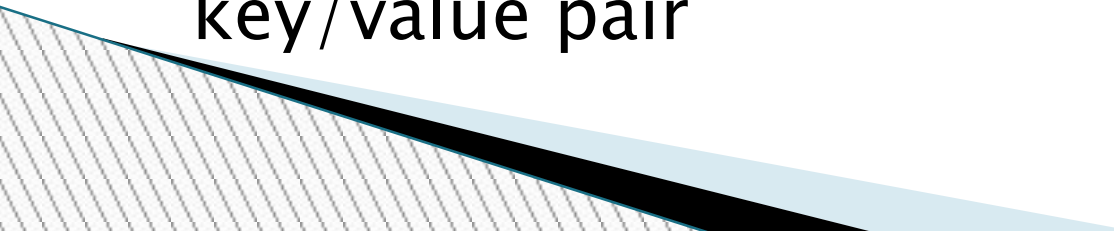
- ▶ Each mapper works only on one input split
- ▶ Input split size can be controlled.
 - Useful for performance improvement
 - Generally input split is equal to block size (64MB)
 - What if you want mapper to work only on 32 MB of a block data?
- ▶ Controlled by 3 properties:
 - `Mapred.min.split.size` (default 1)
 - `Mapred.max.split.size` (default `LONG.MAX_VALUE`)
 - `Dfs.block.size` (default 64 MB)

Input Split cont'd

$\text{Max}(\text{minSplitSize}, \text{min}(\text{maxSplitSize}, \text{blockSize}))$

Min Split Size	Max Split Size	block size	Split size taken
1	LONG.MAX_VALUE	64	64
1	– – “ – –	128	128
128	– – “ – –	128	128
1	32	64	32

What is Mapper?

- ▶ Mapper is the first phase of MapReduce job
 - ▶ Works typically on one block of data (dfs.block.size)
 - ▶ MapReduce framework ensures that map task is run closer to the data to avoid network traffic
 - Several map tasks runs parallel on different machines and each working on different portion (block) of data
 - ▶ Mapper reads key/value pairs and emits key/value pair
- 

Mapper cont'd

Map (in_key,in_val) -----> out_key,out_val

- ▶ Mapper can use or can ignore the input key (in_key)
- ▶ Mapper can emit
 - Zero key value pair
 - 1 key value pair
 - “n” key value pair

Mapper cont'd

- ▶ Map function is called for one record at a time
 - Input Split consist of records
 - For each record in the input split, map function will be called
 - Each record will be sent as key -value pair to map function
 - So when writing map function keep ONLY one record in mind

What is reducer?

- ▶ Reducer runs when all the mapper tasks are completed
- ▶ After mapper phase , all the intermediate values for a given intermediate keys is grouped

KEY ,(Val1,Val2,Val3.....Valn)

- ▶ This list is given to the reducer
 - Reducer operates on Key, and List of Values
 - When writing reducer keep ONLY one key and its list of value in mind

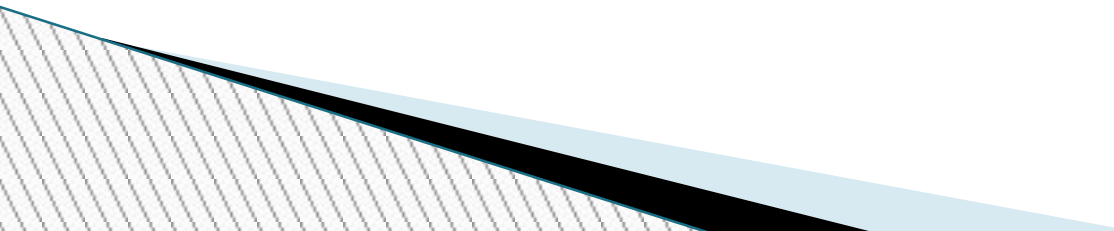
Reducer cont'd

- ▶ NOTE all the values for a particular intermediate keys goes to one reducer
- ▶ There can be Zero, one or “n” reducer.
 - For better load balancing you should have more than one reducer

MapReduce Framework

» ▶ Module 2

In this module you will learn

- ▶ Hadoop primitive data types
 - ▶ What are the various input formats, and what kind of key values they provide to the mapper function
 - ▶ Seeing TextInputFormat in detail
 - ▶ How input split is processed by the mapper?
 - ▶ Understanding the flow of word count problem
- 

Hadoop Primitive Data types

- ▶ Hadoop has its own set of primitive data types for representing its variables
 - For efficient serialization over the network
 - While implementing mapper and reducer functionality you need to emit/write ONLY Hadoop Primitive data types OR your custom class extending from Writable or WritableComparable interface(More on this later)

Hadoop Primitive Data types

cont'd

Java Primitive Data types	Hadoop Primitive Data Types
Integer	IntWritable
Long	LongWritable
Float	FloatWritable
Byte	ByteWritable
String	Text

Input Format

- ▶ Before running the job on the data residing on HDFS, you need to tell “*what kind of data it is?*”
 - Is data is textual data?
 - Is data is binary data?
- ▶ Specify “*InputFormat*” of the data
 - While writing mapper function, you should keep “*input format*” of data in mind, since this will be input key value pair to map function

Input Format cont'd

- ▶ Base class is FileInputFormat

Input Format	Key	Value
Text Input Format	Offset of the line within a file	Entire line till “\n” as value
Key Value Text Input Format	Part of the record till the first delimiter	Remaining record after the first delimiter
Sequence File Input Format	Key needs to be determined from the header	Value needs to be determined from the header

Input Format cont'd

Input Format	Key Data Type	Value Data Type
Text Input Format	LongWritable	Text
Key Value Text Input Format	Text	Text
Sequence File Input Format	ByteWritable	ByteWritable

Text Input Format

- ▶ Efficient for processing text data
- ▶ Example:

```
hello, how are you?  
Hey I am fine?How about you?  
This is plain text  
I will be using Text Input Format
```

Text Input Format cont'd

- ▶ Internally every line is associated with an offset
- ▶ This offset is treated as key. The first column is offset

```
0 Hello, how are you?
```

```
1 Hey I am fine?How about you?
```

```
2 This is plain text
```

```
3 |I will be using Text Input Format
```


Text Input Format cont'd

0 Hello, how are you?

1 Hey I am fine?How about you?

2 This is plain text

3 I will be using Text Input Format

Key	Value
0	Hello, how are you?
1	Hey I am fine?How about you?
2	This is plain text
3	I will be using Text Input Format

How Input Split is processed by mapper?

- ▶ Input split by default is the block size (`dfs.block.size`)
- ▶ Each input split / block comprises of records
 - A record is one line in the input split terminated by “\n” (new line character)
- ▶ Every input format has “*RecordReader*”
 - *RecordReader* reads the records from the input split
 - *RecordReader* reads ONE record at a time and call the map function.
 - *If the input split has 4 records, 4 times map function will be called, one for each record*

Word Count Problem

- ▶ Counting the number of times a word has appeared in a file
- ▶ Example:

Hello, how are you?

Hello, I am fine? How about you?

I am solving word count problem

Word Count Problem cont'd

Hello, how are you?

Hello, I am fine? How about you?

I am solving word count problem

Output of the Word Count problem

Key	Value
Hello	2
How	2
I	2

Word Count Mapper

- ▶ Assuming one input split

```
1 Hello, how are you?
```

```
2 Hello, I am fine? How about you?
```

```
3 |I am solving word count problem
```

- ▶ Input format is Text Input Format
 - Key = Offset which is of type LongWritable
 - Value = Entire Line as Text
- ▶ Remember map function will be called 3times
 - Since there are only 3 records in this input split

Word Count Mapper cont'd

- ▶ Map function for this record

```
1 Hello, how are you?
```

- ▶ Map(1,Hello, how are you?) :

Input to the map function

```
(Hello,1)  
(how,1)  
(are,1)  
(you,1)
```

Intermediate key value pairs
from the mapper

Word Count Mapper cont'd

- ▶ Map function for this record

```
hello, I am fine? How about you?
```

- ▶ Map(2, Hello, I am fine? How about you?)=

Input to the mapper

```
(Hello,1)  
(I,1)  
(am,1)  
:  
:  
:
```

Intermediate key value pair from the mapper

Word Count Mapper cont'd

Pseudo Code

```
Map (inputKey, InputValue)  
{  
    Break the inputValue into individual words;  
    For each word in the individual words  
    {  
        write (word , 1)  
    }  
}
```


Word Count Reducer

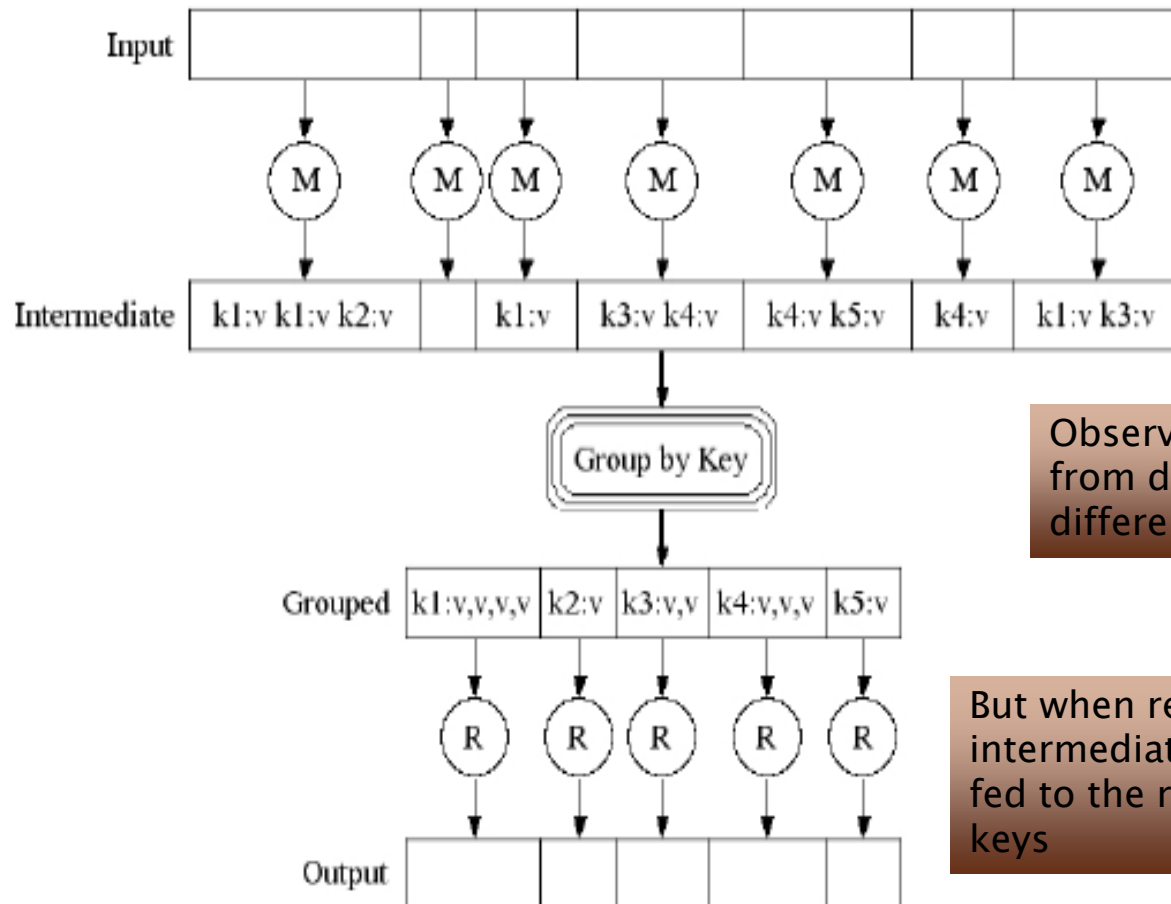
- ▶ Reducer will receive the intermediate key and its list of values
- ▶ If a word “*Hello*” has been emitted 4 times in the map phase, then input to the reducer *will be*
 - (*Hello*, {1, 1, 1, 1})
- ▶ To count the number of times the word “*Hello*” has appeared in the file, just add the number of 1’s

Word Count Reducer cont'd

Pseudo Code

```
Reduce (inputKey, listOfValues)
{
    sum = 0;
    for each value in the listOfValues
    {
        sum = sum + value;
    }
    write(inputKey,sum)
}
```

MapReduce Flow-1

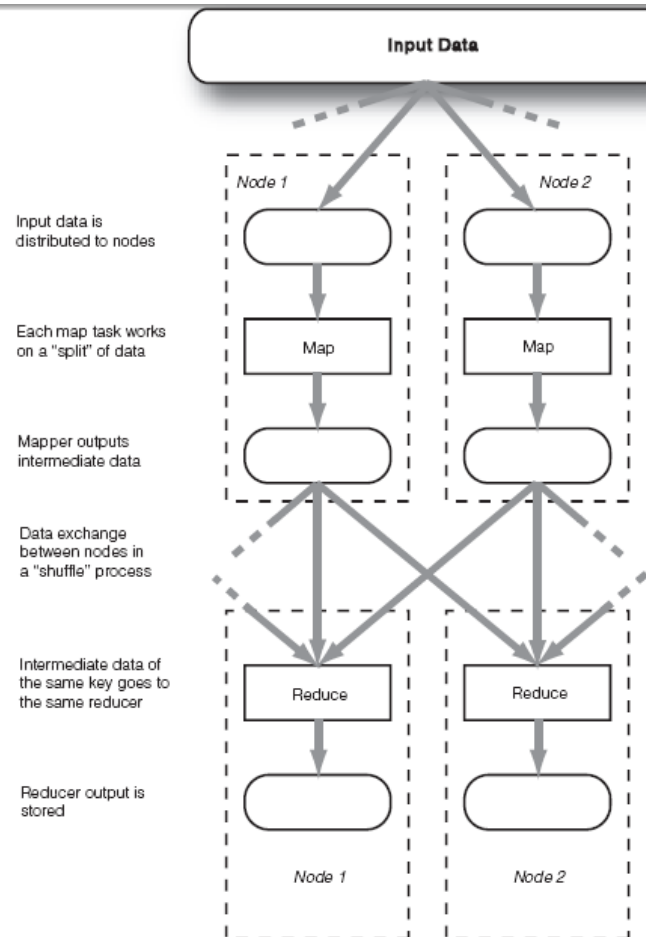


Number of Input Splits = No . of map

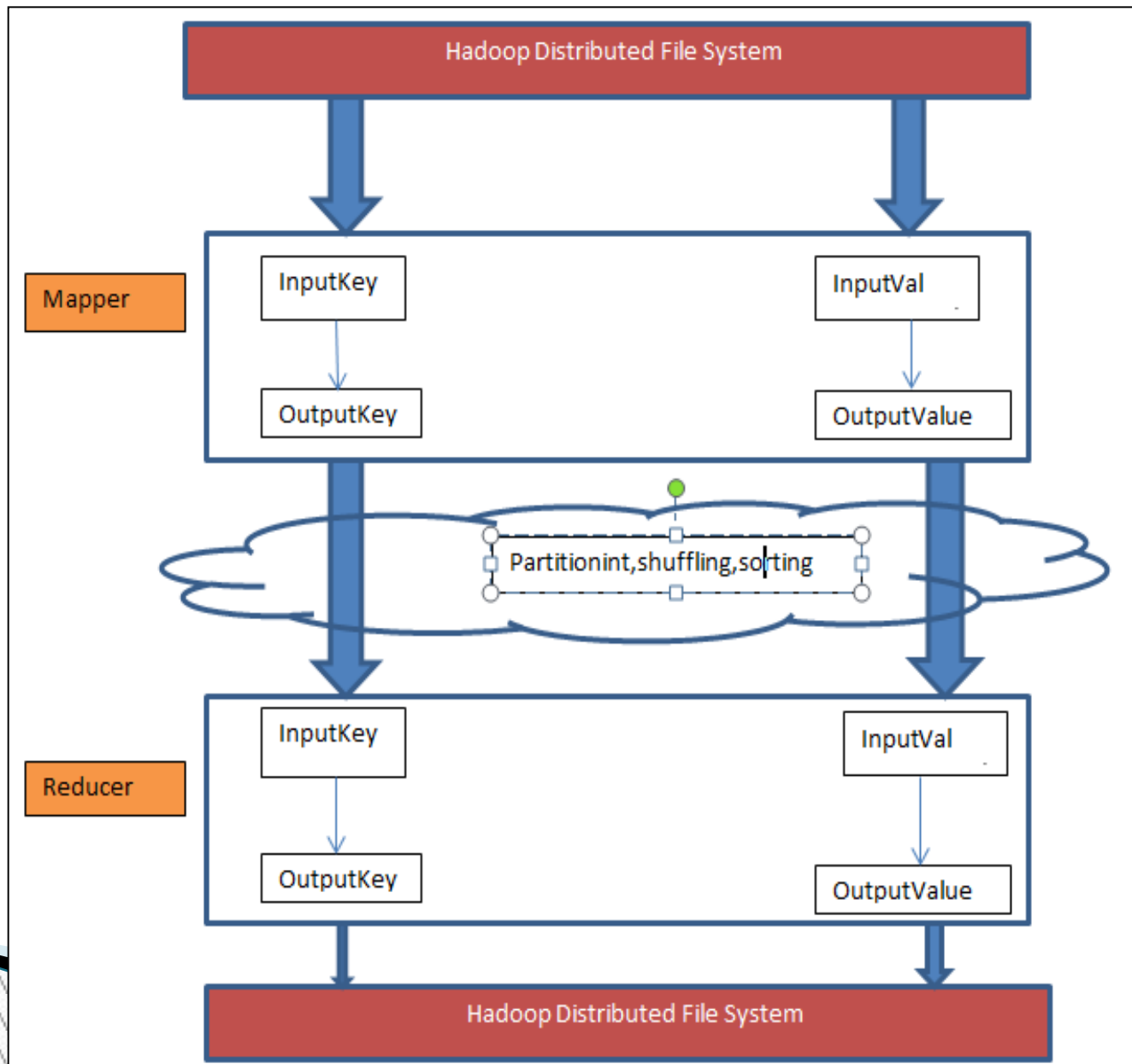
Observe same key can be generated from different mapper running on different machine

But when reducers runs, the keys and its intermediate values are grouped and fed to the reducer in sorted order of keys

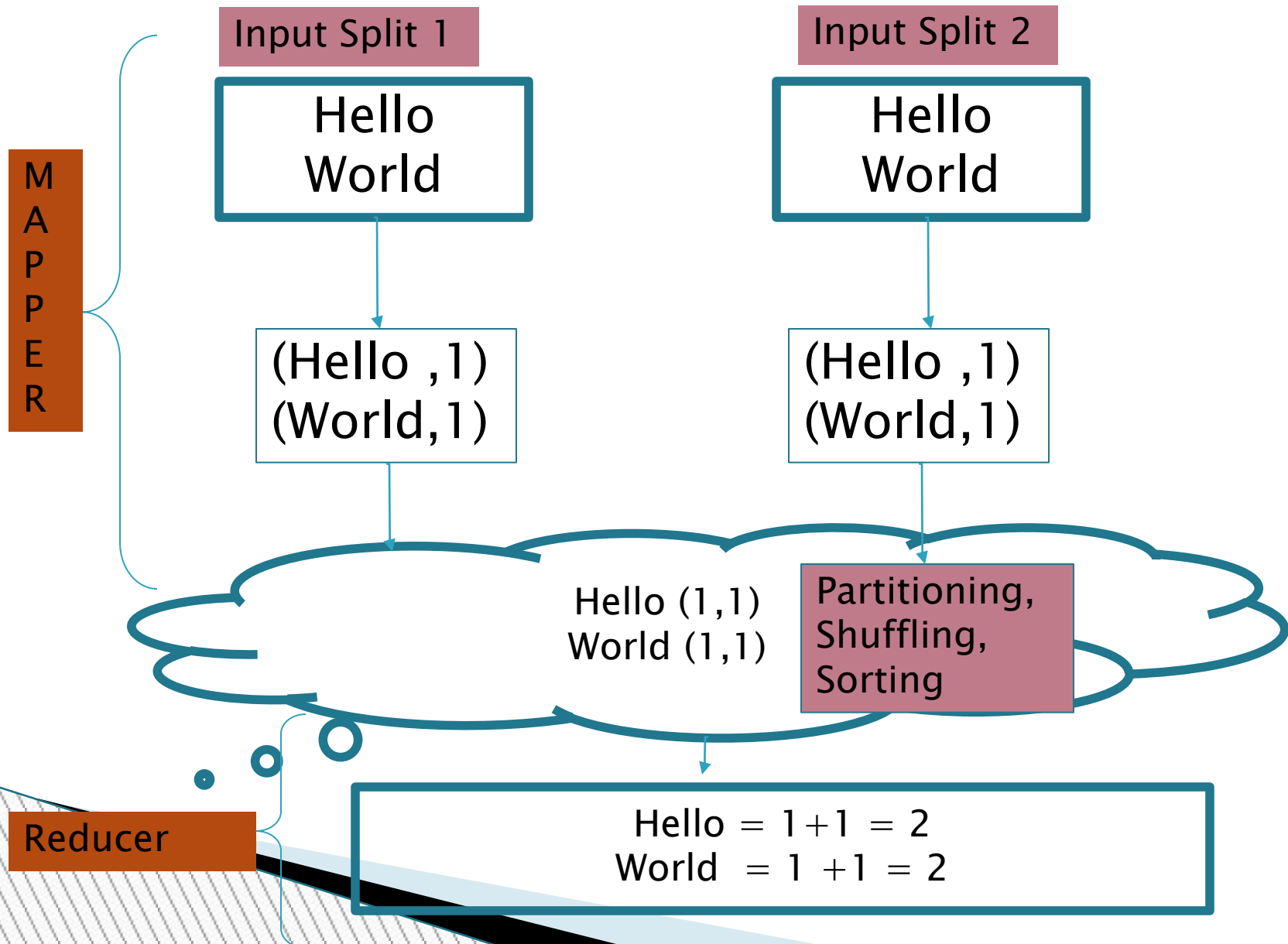
MapReduce Flow-2



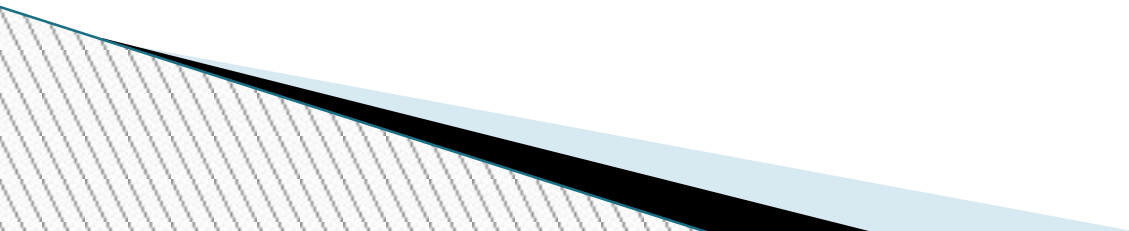
MapReduce Flow – 3



MapReduce Flow– Word Count



MapReduce – Data Flow




Important feature of map reduce job

- ▶ Intermediate output key and value generated by the map phase is stored on local file system of the machine where it is running
 - Intermediate key-value is stored as sequence file (Binary key -value pairs)
- ▶ Map tasks always runs on the machine where the data is present
 - For data localization
 - For reducing data transfer over the network

Features cont'd

- ▶ After the map phase is completed, the intermediate key-value pairs are copied to the local file system of the machine where reducers will run
 - If only ONE reducer is running, then all the intermediate key-value pairs will be copied to the machine
 - Reducer can be very slow if there are large number of output key value pairs are generated
 - So its better to run more than one reducer for load balancing
 - If more than one reducers are running, PARTITIONER decides which intermediate key value pair should go to which reducer

Features cont'd

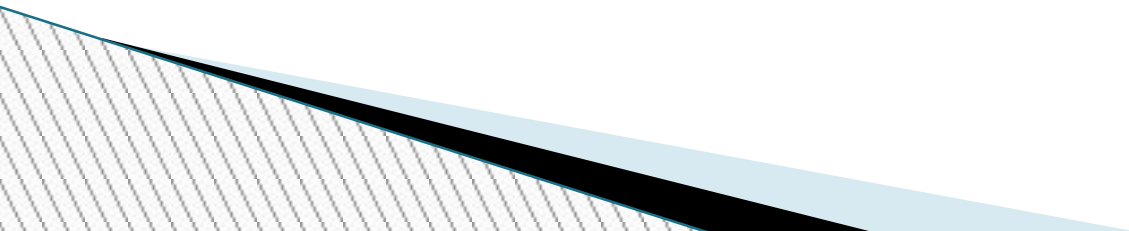
- ▶ Data localization is not applicable for reducer
 - Intermediate key-value pairs are copied
 - ▶ Keys and its list of values are always given to the reducer in SORTED order with respect to key
 - ▶ SORTING on keys happens both after the mapper phase and before the reducer phase
 - Sorting at mapper phase is just an optimization
- 

MapReduce Framework

» » ▶ Module 3

In this module you will learn

- ▶ How will you write mapper class?
- ▶ How will you write reducer class?
- ▶ How will you write driver class?



Writing Mapper Class

```
public class WordCountMapper extends Mapper<LongWritable,  
Text, Text, IntWritable>{  
  
    @Override  
    public void map(LongWritable inputKey, Text  
inputVal, Context context)  
    {  
        String line = value.toString();  
        String[] splits = line.split("/ /W+");  
        for(String outputKey:splits)  
        {  
            context.write(new Text(outputKey), new IntWritable(1));  
        }  
    }  
}
```

Writing Mapper Class

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

- Your Mapper class should extend from Mapper class
- `Mapper<LongWritable,Text,Text,IntWritable>`
 - First argument : Input key given by input format you use
 - Second argument: Input value given by input format
 - Third argument: Output key which you emit from mapper
 - Fourth argument: Output value which you emit from mapper

Writing Mapper Class

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

```
    @Override
```

```
    public void map(LongWritable inputKey, Text inputVal, Context context)
```

```
{
```

- Override the map function
- First argument: Input key to your mapper
- Second argument: Input value to your mapper
- Third argument: Using this context object you will emit output key value pair

```
}
```

Writing Mapper Class

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

    @Override
    public void map(LongWritable inputKey, Text value, Context context)
    {
        String line = value.toString();
        String[] splits = line.split("/ /W+");
        for(String outputKey : splits)
        {
            context.write(new Text(outputKey), new IntWritable(1));
        }
    }
}
```

Step 1: Take the String object from the input value

Step 2: Splitting the string object obtained in step 1, split them into individual words and take them in array

Step 3: Iterate through each words in the array and emit individual word as key and emit value as 1, which is of type IntWritable

Writing Reducer Class

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

Writing Reducer Class

```
public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
```

- Your Reducer class should extend from Reducer class
- `Reducer<Text,IntWritable,Text,IntWritable>`
 - First argument : Input key given by the output key of map output
 - Second argument: Input value given by output value of map output
 - Third argument: Output key which you emit from reducer
 - Fourth argument: Output value which you emit from reducer

Writing Reducer Class

```
public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
```

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

- Reducer will get key and list of values
 - Example: Hello {1,1,1,1,1,1,1,1,1,1}

Writing Reducer Class

```
public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
```

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

```
        int sum = 0;
```

```
        for (IntWritable val : values) {  
            sum += val.get();  
        }
```

```
        output.write(key, new IntWritable(sum));
```

```
    }  
}
```

- Iterate through list of values and add the values

Writing Driver Class

- ▶ Step1 :Get the configuration object, which tells you where the namenode and job tracker are running
- ▶ Step2 :Create the job object
- ▶ Step3: Specify the input format. by default it takes the TextInputFormat
- ▶ Step4: Set Mapper and Reducer class

Driver Class cont'd

```
Configuration conf = new Configuration();  
Path path = new Path("/home/dev/GsData/hadoop-  
localhost.xml");  
conf.addResource(path);
```

```
Job job = new Job(conf,"Basic Word Count Job");  
job.setJarByClass(WordCountDriver.class);
```

```
job.setMapperClass(WordCountMapper.class);  
job.setReducerClass(WordCountReducer.class);
```

```
job.setInputFormatClass(TextInputFormat.class);
```

```
job.setMapOutputKeyClass(Text.class);  
job.setMapOutputValueClass(IntWritable.class);
```

Driver Class cont'd

```
job.setOutputKeyClass(Text.class);  
job.setOutputValueClass(IntWritable.class);  
  
FileInputFormat.addInputPath(job, new Path(args[0]));  
FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
job.waitForCompletion(true);
```

Launching the MapReduce Job

- ▶ Create the jar file
 - Either from eclipse or from command line

```
hadoop jar <jarName.jar> <DriverClassName> <input>  
<output>
```

- ▶ <input> could be a file or directory consisting of files on HDFS
- ▶ <output> Name should be different for every run.
- ▶ If the <output> directory with the same name is present, exception will be thrown

MapReduce

» ▶ Module 4

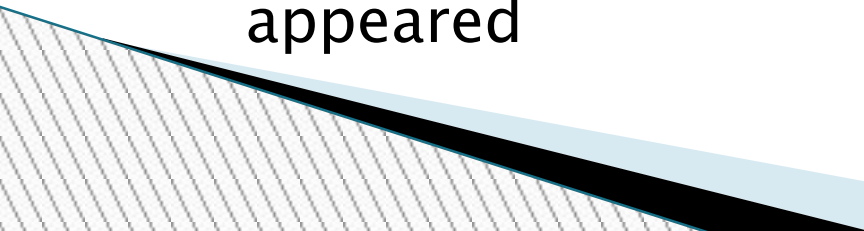
In this module you will learn

- ▶ Word Co-Occurrence problem
- ▶ Average Word Length Problem
- ▶ Inverted Index problem
- ▶ Hands on

Word Co-Occurrence Problem

- ▶ Measuring the frequency with which two words appearing in a set of documents
- ▶ Used for recommendation like
 - *“You might like this also “*
 - *“People who choose this also choose that”*
 - *Examples:*
 - Shopping recommendations
 - Identifying people of interest
- ▶ Similar to word count but two words at a time

Inverted Index Problem

- ▶ Used for faster look up
 - Example: Indexing done for keywords in a book
 - ▶ Problem statement:
 - From a list of files or documents map the words to the list of files in which it has appeared
 - ▶ Output
 - *word* = List of documents in which this *word* has appeared
- 

Indexing Problem cont'd

File A

This is cat
Big fat hen

Output from
the mapper

This: File A
is: File A
cat: File A
Big: File A
fat: File A
hen: File A

Final Output

This: File A, File B
is: File A, File B
cat: File A
fat: File A, File B

File B

This is dog
My dog is fat

This: File B
is: File B
dog: File B
My: File B
dog: File B
is: File B
fat: File B

Indexing problem cont'd

▶ Mapper

For each word in the line, *emit(word, file_name)*

▶ Reducer

- Remember for *word*, all the file names list will be coming to the reducer
- *Emit(word, file_name_list)*

Average Word Length Problem

- ▶ Consider the record in a file

Hey How is Henry these days?

- ▶ Problem Statement
 - Calculate the average word length for each character
- ▶ Output:

Character	Average word length
H	$(3+3+5) / 3 = 3.66$
I	$2 / 1 = 2$
T	$5 / 1 = 5$
D	$4 / 1 = 4$

Average Word Length cont'd

▶ Mapper

- For each word in a line,
emit(firstCharacterOfWord,lengthOfWord)

▶ Reducer

- You will get a character as key and list of values corresponding to length
- For each value in listOfValue
 - Calculate the sum and also count the number of values
 - *Emit(character,sum/count)*

MapReduce Framework

» » ▶ Module 4

In this module you will learn

- ▶ What is combiner?
- ▶ What is Partitioner?
- ▶ Hands On

Combiner

- ▶ Large number of mapper running will produce large amounts of intermediate data
 - This data needs to be passed to the reducer over the network
 - Lot of network traffic
 - Shuffling/Copying the mapper output to the machine where reducer will run will take lot of time

Combiner cont'd

▶ Similar to reducer

- Runs on the same machine as the mapper task
- Runs the reducer code on the intermediate output of the mapper
 - Thus minimizing the intermediate key-value pairs
- Combiner runs on intermediate output of each mapper

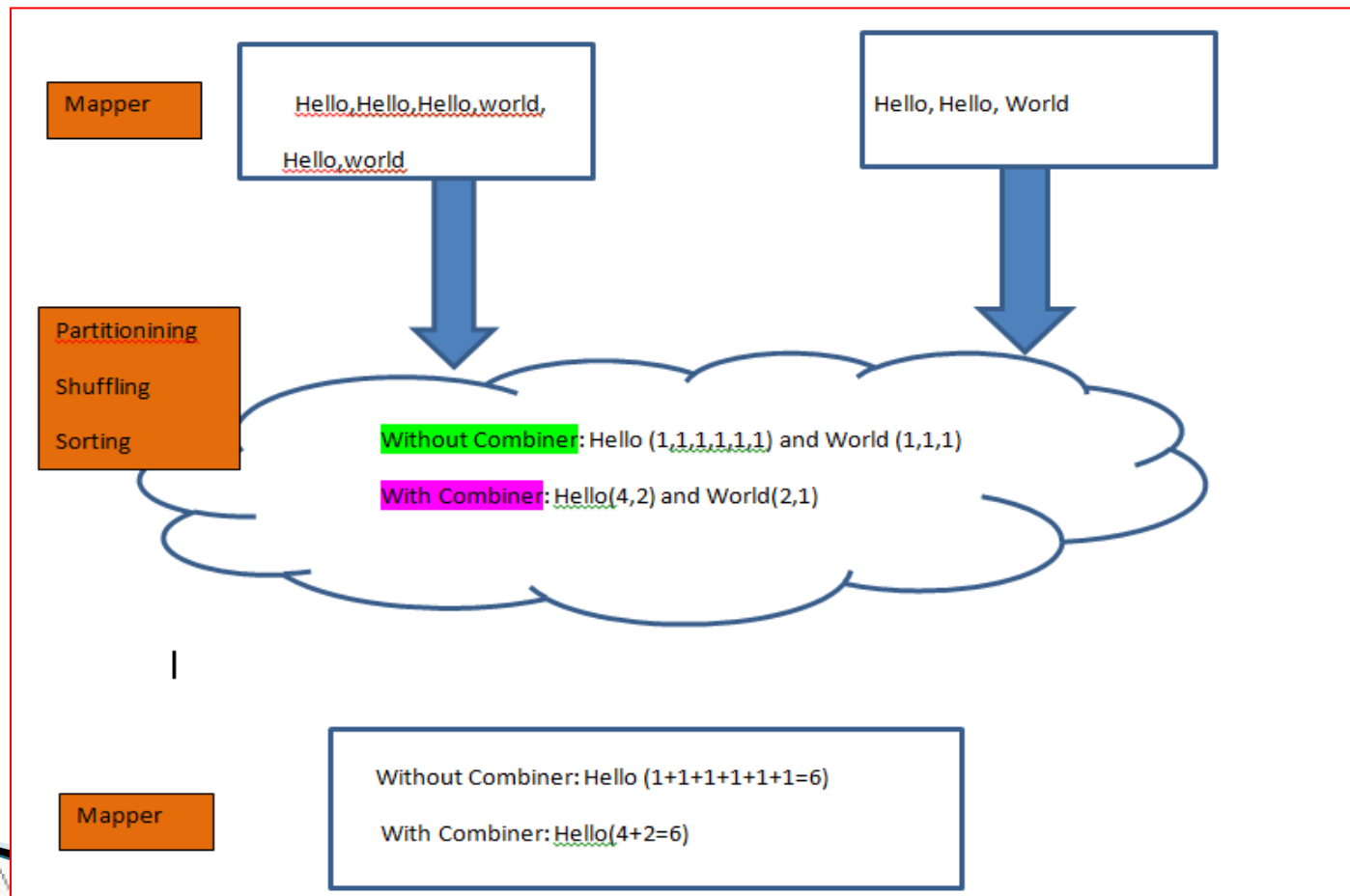
▶ Advantages

- Minimize the data transfer across the network
- Speed up the execution
- Reduces the burden on reducer

Combiner cont'd

- ▶ Combiner has the same signature as reducer class
- ▶ Can make the existing reducer to run as combiner, if
 - The operation is associative or commutative in nature
 - Example: Sum, Multiplication
 - Average operation cannot be used
- ▶ Combiner may or may not run. Depends on the framework

Combiner cont'd



Combiner cont'd

- ▶ In the driver class specify
 - *`Job.setCombinerClass(MyReducer.class);`*

Important things about Combiner

- ▶ Framework decides whether to run the combiner or not
- ▶ Combiner may run more than once on the same mapper
 - Depends on two properties `io.sort.factor` and `io.sort.mb`

Partitioner

- ▶ It is called after you emit your key value pairs from mapper
 - *context.write(key,value)*
- ▶ Large number of mappers running will generate large amount of data
 - And If only one reducer is specified, then all the intermediate key and its list of values goes to a single reducer
 - Copying will take lot of time
 - Sorting will also be time consuming
 - Whether single machine can handle that much amount of intermediate data or not?

Partitioner cont'd

- ▶ Partitioner divides the keys among the reducers
 - If more than one reducer running, then partitioner decides which key value pair should go to which reducer
- ▶ Default is “*Hash Partitioner*”
 - Calculates the “*hashcode*” and do the modulo operator with total number of reducers which are running

Hash code of key % numOfReducer

The above operation returns between ZERO and (numOfReducer - 1)

Partitioner cont'd

▶ Example:

- Number of reducers = 3
 - Key = *"Hello"*
 - Hash code = 30 (let's say)
 - The key *"Hello"* and its list of values will go to $30 \% 3 = 0$ th reducer
- Key = *"World"*
- Hash Code = 31 (let's say)
- The key *"world"* and its list of values will go to $31 \% 3 = 1$ st reducer

Thank You

By
Mr. Sasidhar

sasis937@gmail.com

