Welcome to Big Data Analytics Module

Instructor : Tushar Kakaiya

MapReduce Framework

Chapter - 05

Introducing Map Reduce Framework

- MapReduce is a programming model for distributed data processing based on Java.
- Hadoop can run MapReduce programs written in various languages.
- Most importantly, MapReduce programs are inherently parallel, thus putting very large-scale data analysis into the hands of anyone with enough machines at their disposal.
- The MapReduce algorithm 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). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job.

Introducing Map Reduce Framework

- The major advantage of MapReduce is that it is easy to scale data processing over multiple computing nodes.
- Under the MapReduce model, the data processing primitives are called **mappers** and **reducers**. Decomposing a data processing application into mappers and reducers is sometimes nontrivial. But, once 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 is merely a configuration change. This simple scalability is what has attracted many programmers to use the MapReduce model.

Brief History

- MapReduce was first popularized as a programming model in 2004 by Jeffery Dean and Sanjay Ghemawat of Google (Dean & Ghemawat, 2004). In their paper, “MAPREDUCE: SIMPLIFIED DATA PROCESSING ON LARGE CLUSTERS,” they discussed Google’s approach to collecting and analyzing website data for search optimizations. Google’s proprietary MapReduce system ran on the Google File System (GFS).

- Apache, the open source organization, began using MapReduce in the “Nutch” project, which is an open source web search engine that still is active today. Hadoop began as a subproject in the Apache Lucene project, which provides text search capabilities across large databases.

- In 2006, Doug Cutting, an employee of Yahoo!, designed Hadoop, naming it after his son’s toy elephant. While it was originally a subproject, Cutting released Hadoop as an open source Apache project in 2007. (Hadoop, 2011). In 2008, Hadoop became a top level project at Apache. On July 2008, an experimental 4000 node cluster was created using Hadoop, and in 2009 during a performance test, Hadoop was able to sort a terabyte of data in 17 hours.

Usage

- In the eCommerce industry today, several of the major players use Hadoop for high volume data processing (Borthakur, 2009). Amazon, Yahoo and Zvents use Hadoop for search processing. Using Hadoop helps them to determine search intent from text entered, and optimize future searches using statistical analysis.

- Facebook, Yahoo, ContextWeb, Joost, and Last.fm use Hadoop to process logs and mine for click stream data. Click stream data records an end users activity and profitability on a web site.

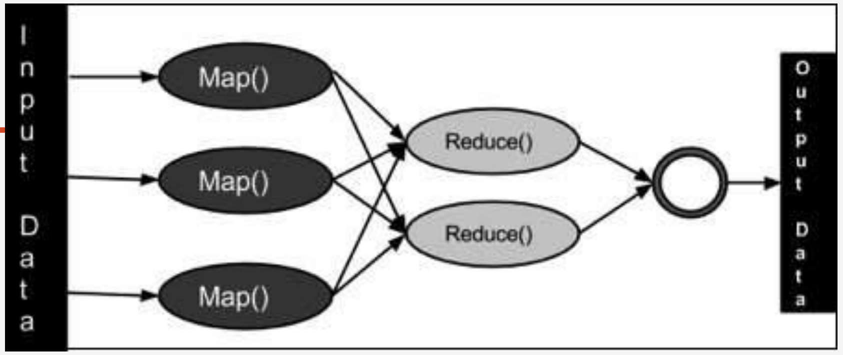
- **Facebook** and AOL are using Hadoop in their data warehouse as a way to effectively store and mine the large amounts of data they collect. The New York Times and Eyalike are using Hadoop to store and analyze videos and images.

- In order to use MapReduce, you don't need to have a large investment in infrastructure. Hadoop is also available on many public cloud provider offerings in a pay per use model. **Amazon** offers an elastic map reduce service. **Google and IBM** are offering MapReduce in IBM Blue Cloud.

Advantages

- **Simple Coding Model:** The programmer does not have to implement parallelism, distributed data passing or any of the complexities that they would otherwise be faced with
- **Scalable :** can scale across thousands of nodes with same code reusability and without any major code rewrite (with some config changes only!)
- **Supports Unstructured Data :** Unstructured data is data that does not follow a specified format for Big data. The MapReduce model processes large unstructured data sets with a distributed algorithm on a Hadoop cluster.
- **Fault Tolerance :** Typically the distributed file systems that MapReduce support, along with the Master Process, enable MapReduce jobs to survive hardware failures

MapReduce Algorithm



Generally MapReduce paradigm is based on sending the computer / resources to where the data resides!

MapReduce program executes in three stages, namely map stage, shuffle stage, and 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.

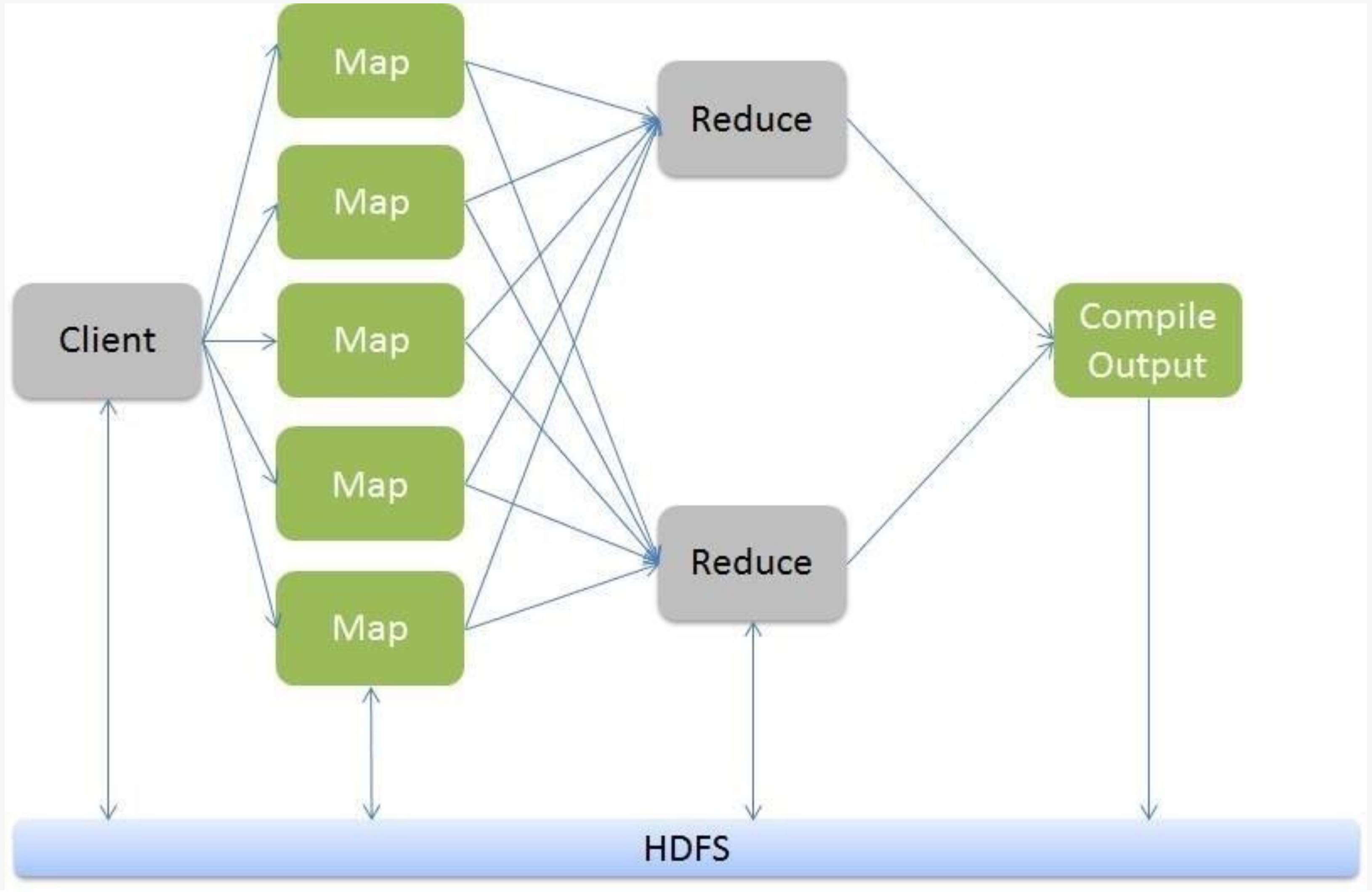
- **Reduce stage** − This stage is the combination of the **Shuffle** stage and the **Reduce** stage. The Reducer's job is to process the data that comes from the mapper. After processing, it produces a new set of output, which will be stored in the HDFS.

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.

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.

MapReduce Algorithm



MapReduce Algorithm

The whole process goes through four phases of execution namely, splitting, mapping, shuffling, and reducing.

Let's understand this with an example –

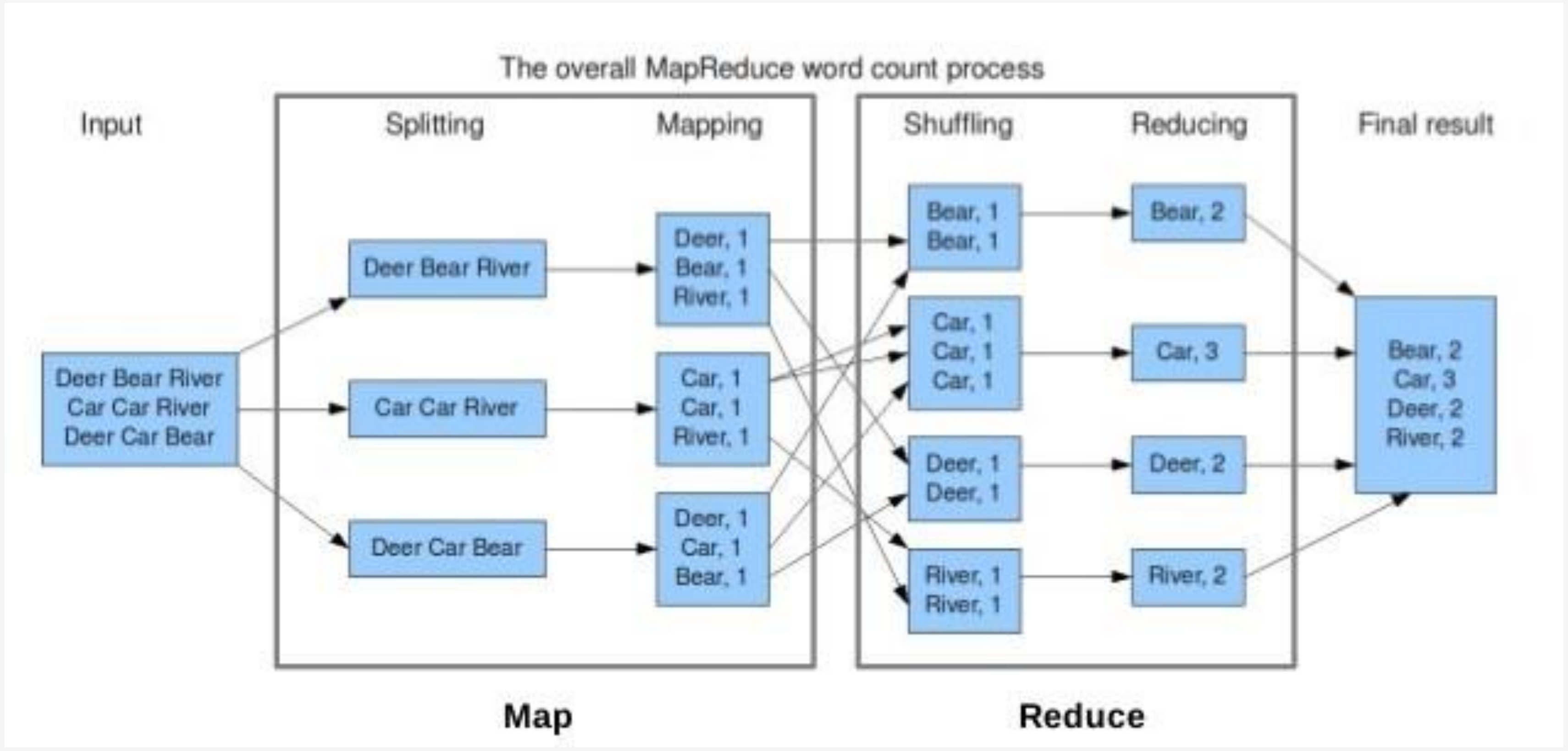
Consider you have following input data for your Map Reduce Program

```
Deer Bear River

Car Car River

Deer Car Bear
```

The final output of the MapReduce task is

Deer	2
Bear	2
Car	3
River	2

MapReduce Algorithm



MapReduce Algorithm

Input Splits:

An input to a MapReduce job is divided into fixed-size pieces called **input splits** Input split is a chunk of the input that is consumed by a single map.

Mapping

This is the very first phase in the execution of map-reduce program. In this phase data in each split is passed to a mapping function to produce output values. In our example, a job of mapping phase is to count a number of occurrences of each word from input splits and prepare a list in the form of <word, frequency>

This phase generally contains business logic and it should be written mandatorily.

Shuffling

This phase consumes the output of Mapping phase. Its task is to consolidate the relevant records from Mapping phase output. In our example, the same words are clubbed together along with their respective frequency. Framework does Sort and Shuffle on its own, we generally do not write the Shufflers.

Reducing

In this phase, output values from the Shuffling phase are aggregated. This phase combines values from Shuffling phase and returns a single output value. In short, this phase summarizes the complete dataset. Generally contains aggregation logic and this phase is optional. Sometimes, we may have a requirement where we do not need Reducer and only Mappers should suffice.

MapReduce Architecture in detail

- One map task is created for each split which then executes map function for each record in the split.
- It is always beneficial to have multiple splits because the time taken to process a split is small as compared to the time taken for processing of the whole input. When the splits are smaller, the processing is better to load balanced since we are processing the splits in parallel. However, it is also not desirable to have splits too small in size. When splits are too small, the overload of managing the splits and map task creation begins to dominate the total job execution time.
- For most jobs, it is better to make a split size equal to the size of an HDFS block (which is 64 MB in Rel.1 and 128 MB in Rel.2 , by default).
- Execution of map tasks results into writing output to a local disk on the respective node and not to HDFS. Reason for choosing local disk over HDFS is, to avoid replication which takes place in case of HDFS store operation. Map output is intermediate output which is processed by reduce tasks to produce the final output. Once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication becomes overkill. In the event of node failure, before the map output is consumed by the reduce task, Hadoop reruns the map task on another node and re-creates the map output.
- Reduce task doesn't work on the concept of data locality. An output of every map task is fed to the reduce task. Map output is transferred to the machine where reduce task is running.
- On this machine, the output is merged and then passed to the user-defined reduce function.
- Unlike the map output, reduce output is stored in HDFS (the first replica is stored on the local node and other replicas are stored on off-rack nodes).

How MapReduce Organizes Work?

Hadoop divides the job into tasks. There are two types of tasks:

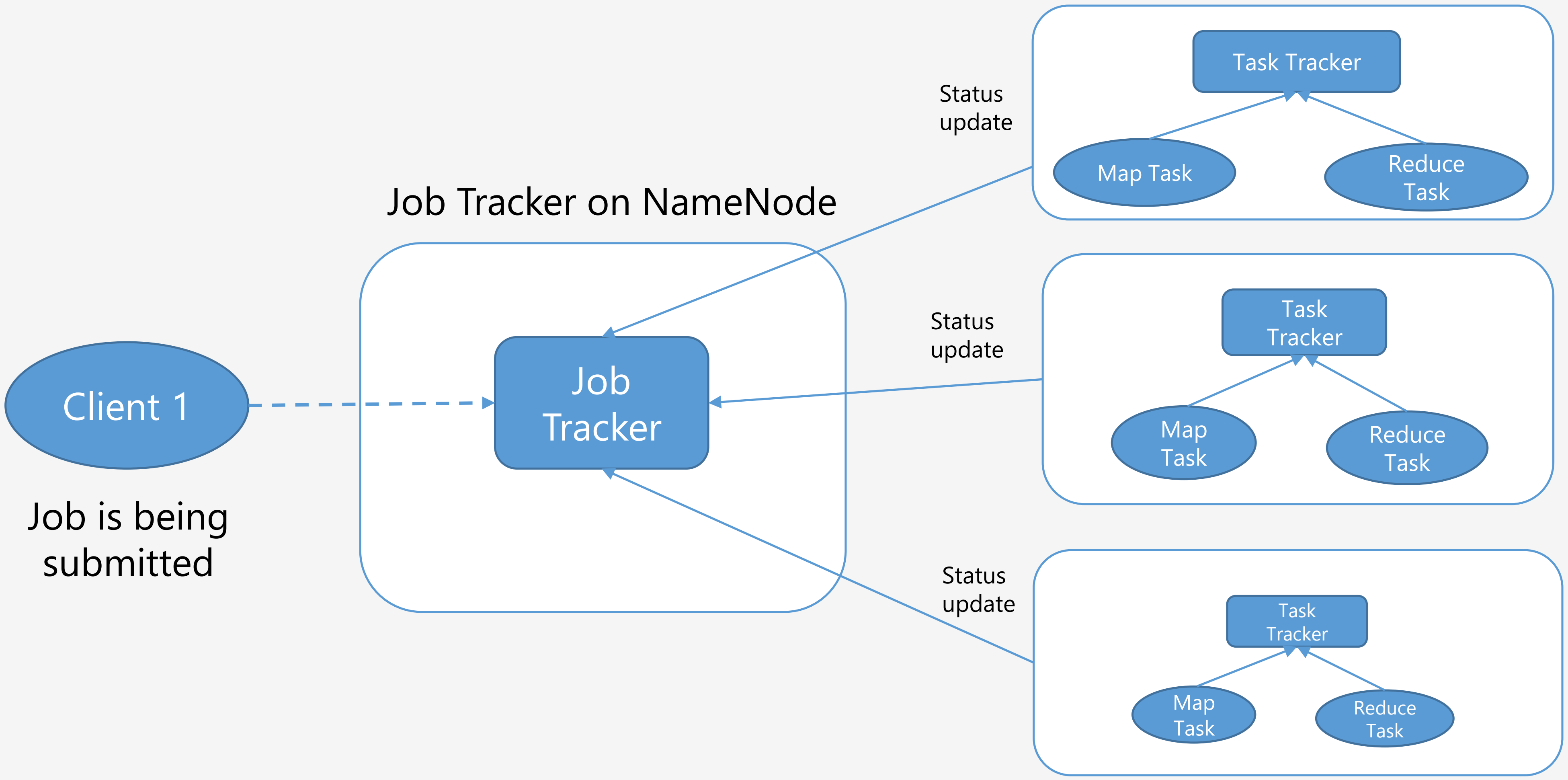
Map tasks (Splits & Mapping)

Reduce tasks (Shuffling, Reducing)

The complete execution process (execution of Map and Reduce tasks, both) is controlled by two types of entities (In Hadoop Rel. 1) called as:

- **Jobtracker**: Acts like a **master** (responsible for complete execution of submitted job)
- **Multiple Task Trackers**: Acts like **slaves,** each of them performing the job

How MapReduce Organizes Work?



How MapReduce Organizes Work?

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 Locality Advantages

Data Locality / Moving Processing Unit to Data (MapReduce Approach)

Instead of moving data to the processing unit, we are moving processing unit to the data in the MapReduce Framework. In the traditional system, we used to bring data to the processing unit and process it. But, as the data grew and became very huge, bringing this huge amount of data to the processing unit posed following issues:

- Moving huge data to processing is costly and deteriorates the network performance.
- Processing takes time as the data is processed by a single unit which becomes the bottleneck.
- Master node can get over-burdened and may fail.

WordCount Example – Practical Demo

Let’s say we have a file of words and we want to count no. of occurrences for each word.

The entire MapReduce program can be fundamentally divided into three parts:

- Mapper Phase Code
- Reducer Phase Code
- Driver Code

WordCount Example – Practical Demo

```
WordCount.Java File

package com.hadoop.mr

import java.io.IOException;

import java.util.StringTokenizer;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.fs.Path;
```

WordCount Example – Practical Demo

Mapper code:

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

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

String line = value.toString();

StringTokenizer tokenizer = new StringTokenizer(line);

while (tokenizer.hasMoreTokens()) {

value.set(tokenizer.nextToken());

context.write(value, new IntWritable(1));

}
```

WordCount Example – Practical Demo

We have created a class Map that extends the class Mapper which is already defined in the MapReduce Framework.

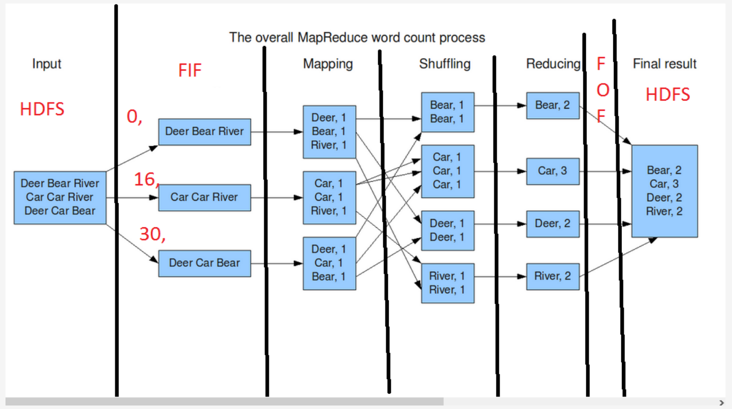
We define the data types of input and output key/value pair after the class declaration using angle brackets.

Both the input and output of the Mapper is a key/value pair.

•Input:

•The *key* is nothing but the offset of each line in the text file: *LongWritable*

•The *value* is each individual line (as shown in the figure at the right): *Text*



•Output:

The *key* is the tokenized words: *Text*

We have the hardcoded *value* in our case which is 1: *IntWritable*

Example – Dear 1, Bear 1, etc.

We have written a java code where we have tokenized each word and assigned them a hardcoded value equal to *1*.

WordCount Example – Practical Demo

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

public void reduce(Text key, Iterable<IntWritable> values,Context context)

throws IOException,InterruptedException {

int sum=0;

for(IntWritable x: values)

{

sum+=x.get();

}

context.write(key, new IntWritable(sum));

}

}
```

WordCount Example – Practical Demo

We have created a class Reduce which extends class Reducer like that of Mapper.

We define the data types of input and output key/value pair after the class declaration using angle brackets as done for Mapper.

Both the input and the output of the Reducer is a key-value pair.

Input:

- The *key* nothing but those unique words which have been generated after the sorting and shuffling phase: *Text*
- The *value* is a list of integers corresponding to each key: *IntWritable*
- Example – Bear, [1, 1], etc.

Output:

- The *key* is all the unique words present in the input text file: *Text*
- The *value* is the number of occurrences of each of the unique words: *IntWritable*
- Example – Bear, 2; Car, 3, etc.

We have aggregated the values present in each of the list corresponding to each key and produced the final answer.

In general, a single reducer is created for each of the unique words, but, you can specify the number of reducer in mapred-site.xml.

WordCount Example – Practical Demo

Driver Code:

```
JobConf conf = new JobConf(WordCount.class);

conf.setJobName("wordcount");

conf.setOutputKeyClass(Text.class);

conf.setOutputValueClass(IntWritable.class);

conf.setMapperClass(Map.class);

conf.setReducerClass(Reduce.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);
```

WordCount Example – Practical Demo

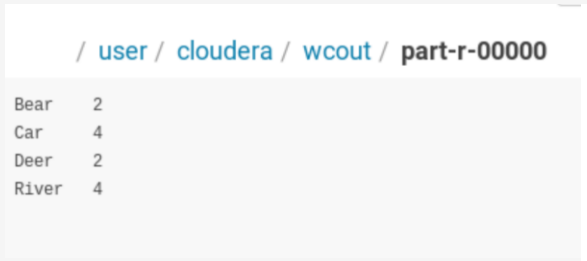
- In the driver class, we set the configuration of our MapReduce job to run in Hadoop.
- We specify the name of the job , the data type of input/output of the mapper and reducer.
- We also specify the names of the mapper and reducer classes.
- The path of the input and output folder is also specified.
- The method setInputFormatClass () is used for specifying that how a Mapper will read the input data or what will be the unit of work. Here, we have chosen TextInputFormat so that single line is read by the mapper at a time from the input text file.
- The main () method is the entry point for the driver. In this method, we instantiate a new Configuration object for the job.

WordCount Example – Practical Demo

Run the MapReduce code:

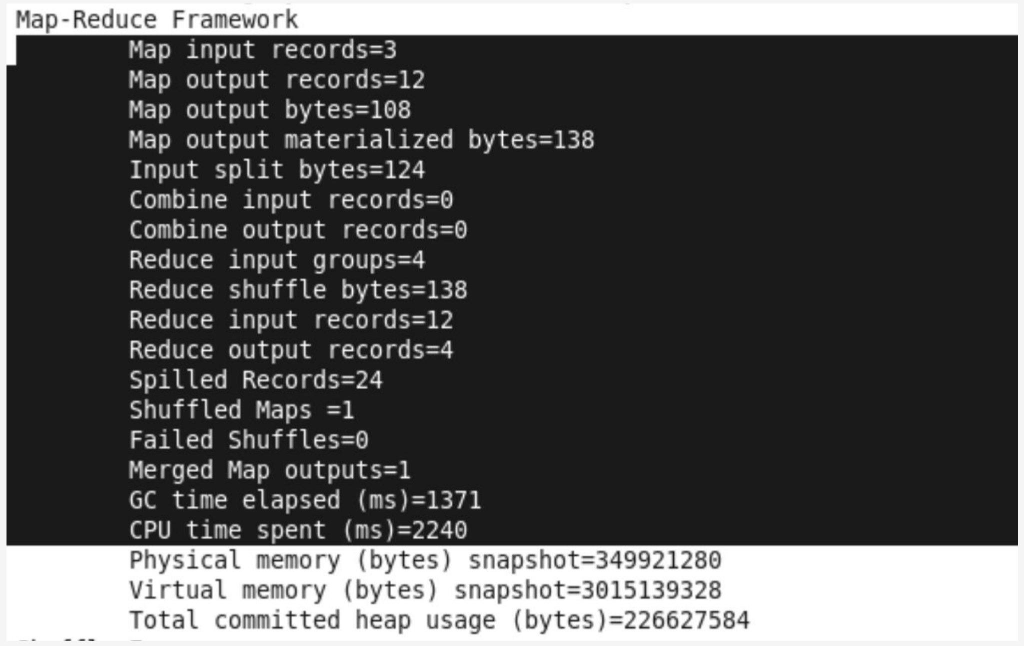
The command for running a MapReduce code is:

hadoop jar hadoop-mr-wc.jar Input/wcinput wcout

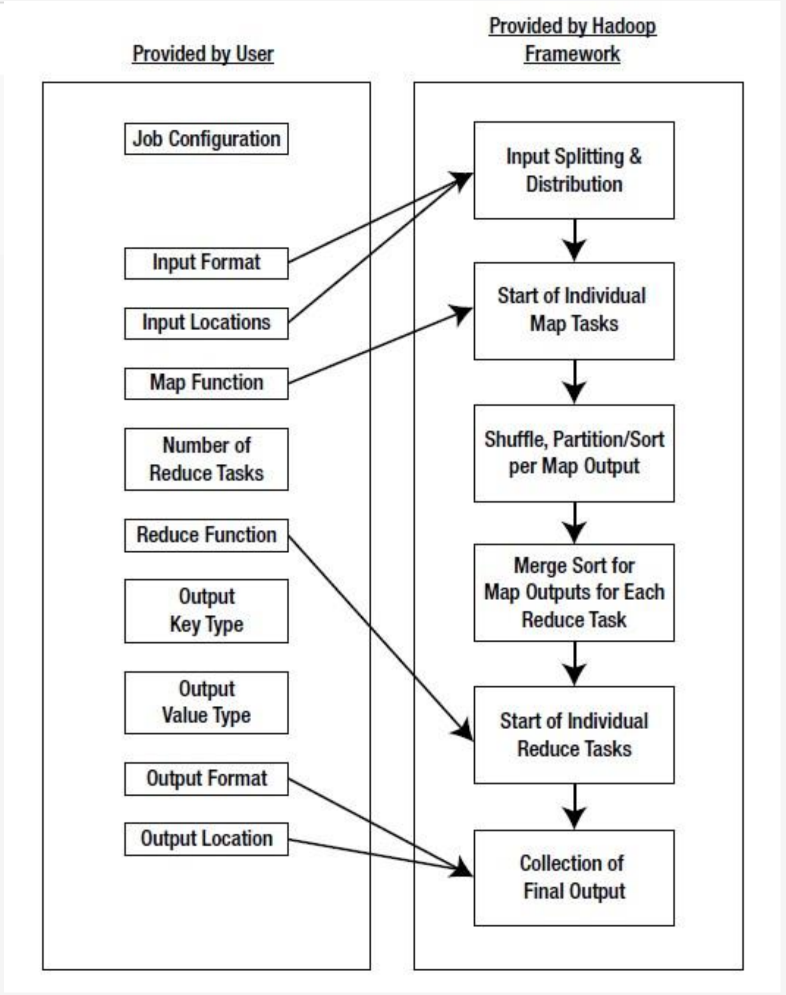


/ user / cloudera / wcout / part-r-00000

```
Bear    2
Car     4
Deer    2
River   4
```



```
Map-Reduce Framework
        Map input records=3
        Map output records=12
        Map output bytes=108
        Map output materialized bytes=138
        Input split bytes=124
        Combine input records=0
        Combine output records=0
        Reduce input groups=4
        Reduce shuffle bytes=138
        Reduce input records=12
        Reduce output records=4
        Spilled Records=24
        Shuffled Maps =1
        Failed Shuffles=0
        Merged Map outputs=1
        GC time elapsed (ms)=1371
        CPU time spent (ms)=2240
        Physical memory (bytes) snapshot=349921280
        Virtual memory (bytes) snapshot=3015139328
        Total committed heap usage (bytes)=226627584
```


Provided by User
Provided by Hadoop Framework
Job Configuration
Input Format
Input Locations
Map Function
Number of Reduce Tasks
Reduce Function
Output Key Type
Output Value Type
Output Format
Output Location
Input Splitting & Distribution
Start of Individual Map Tasks
Shuffle, Partition/Sort per Map Output
Merge Sort for Map Outputs for Each Reduce Task
Start of Individual Reduce Tasks
Collection of Final Output

Sales Per Country Example – Practical Demo

The input data used is **SalesJan2009.csv**. It contains Sales related information like Product name, price, payment mode, city, country of client etc.

The goal is to ***Find out Number of Products Sold in Each Country.***

MapReduce – 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. Let us take an example to understand how the partitioner works.

Shuffler is built in feature of Framework and work on hash key functions.

To customize shuffler – there is replacement of shuffler named as Partitioner.

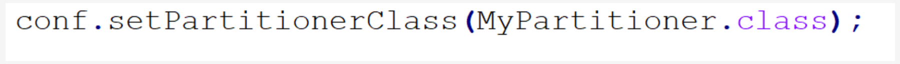
If in your code, partitioner is not specified at all then by default it calls built in shuffler.

But if u code Partitioner, then no shuffler is called and your code would be executed which u have written under partitioner class.

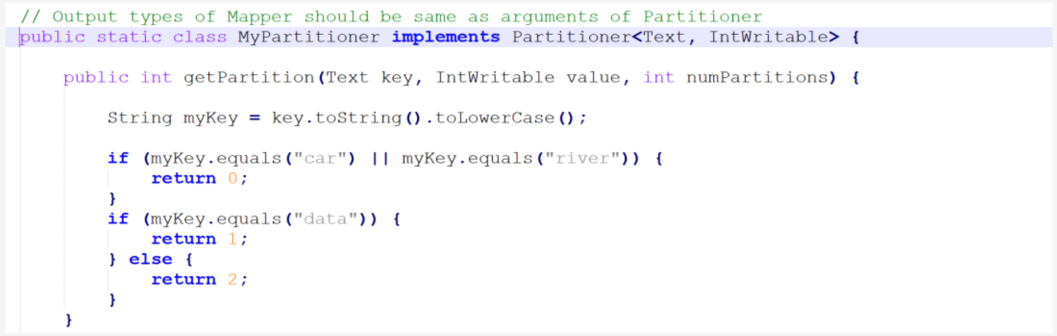
MapReduce – Partitioner

We can run the Example of WordCount with Partitioner using WithPartitioner.Java file.

- In the Driver class, we can specify which class to make Partitioner class where the Partitioning logic is implemented, by setPartitionerClass method,

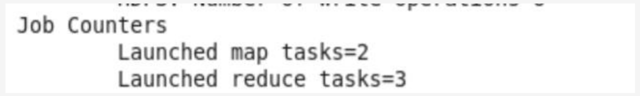


```
conf.setPartitionerClass(MyPartitioner.class);
```



```
// Output types of Mapper should be same as arguments of Partitioner
public static class MyPartitioner implements Partitioner<Text, IntWritable> {

    public int getPartition(Text key, IntWritable value, int numPartitions) {

        String myKey = key.toString().toLowerCase();

        if (myKey.equals("car") || myKey.equals("river")) {
            return 0;
        }
        if (myKey.equals("data")) {
            return 1;
        } else {
            return 2;
        }
    }
```

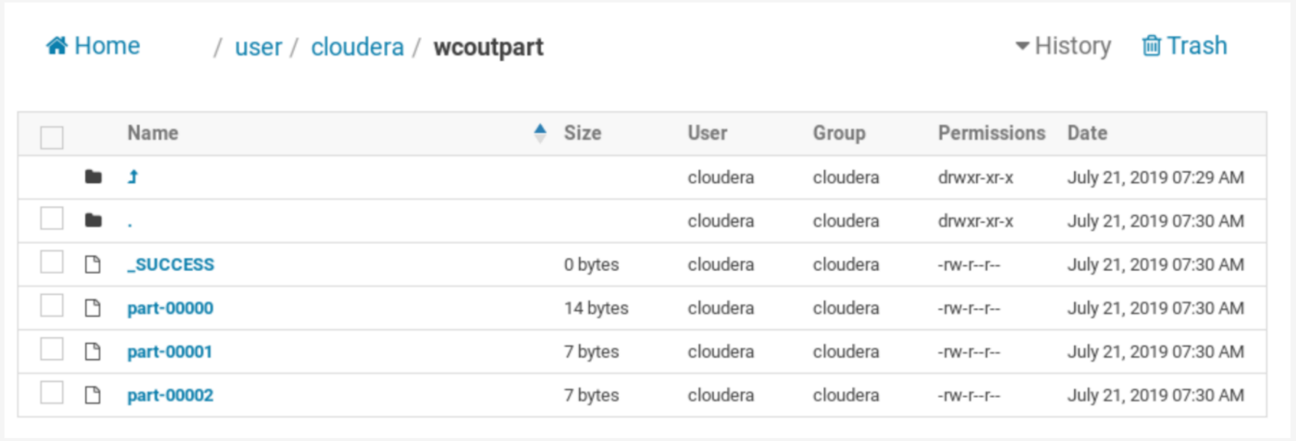
When we try to run the Job, we can see the stats are clearly showing to invoke 3 reducers rather than 1 default in previous basic MR program.



```
Job Counters
        Launched map tasks=2
        Launched reduce tasks=3
```

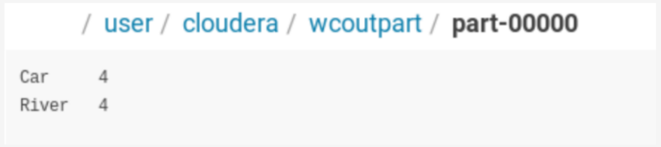
MapReduce – Partitioner

We can see 3 output part files are there in the Output HDFS Directory



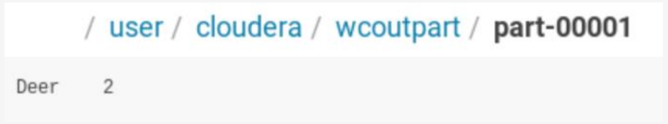
Home / user / cloudera / wcoutpart History Trash

	Name	Size	User	Group	Permissions	Date
	↥		cloudera	cloudera	drwxr-xr-x	July 21, 2019 07:29 AM
	.		cloudera	cloudera	drwxr-xr-x	July 21, 2019 07:30 AM
	_SUCCESS	0 bytes	cloudera	cloudera	-rw-r--r--	July 21, 2019 07:30 AM
	part-00000	14 bytes	cloudera	cloudera	-rw-r--r--	July 21, 2019 07:30 AM
	part-00001	7 bytes	cloudera	cloudera	-rw-r--r--	July 21, 2019 07:30 AM
	part-00002	7 bytes	cloudera	cloudera	-rw-r--r--	July 21, 2019 07:30 AM



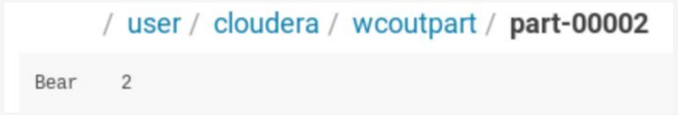
/ user / cloudera / wcoutpart / part-00000

```
Car     4
River   4
```



/ user / cloudera / wcoutpart / part-00001

```
Deer    2
```



/ user / cloudera / wcoutpart / part-00002

```
Bear    2
```

MapReduce – Combiner

On a large dataset when we run **MapReduce job**, large chunks of intermediate data is generated by the Mapper and this intermediate data is passed on the Reducer for further processing, which leads to enormous network congestion. MapReduce framework provides a function known as **Hadoop Combiner** that plays a key role in reducing network congestion.

A Combiner, also known as **a semi-reducer**, is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class.

The main function of a Combiner is to summarize the map output records with the same key. The output (key-value collection) of the combiner will be sent over the network to the actual Reducer task as input.

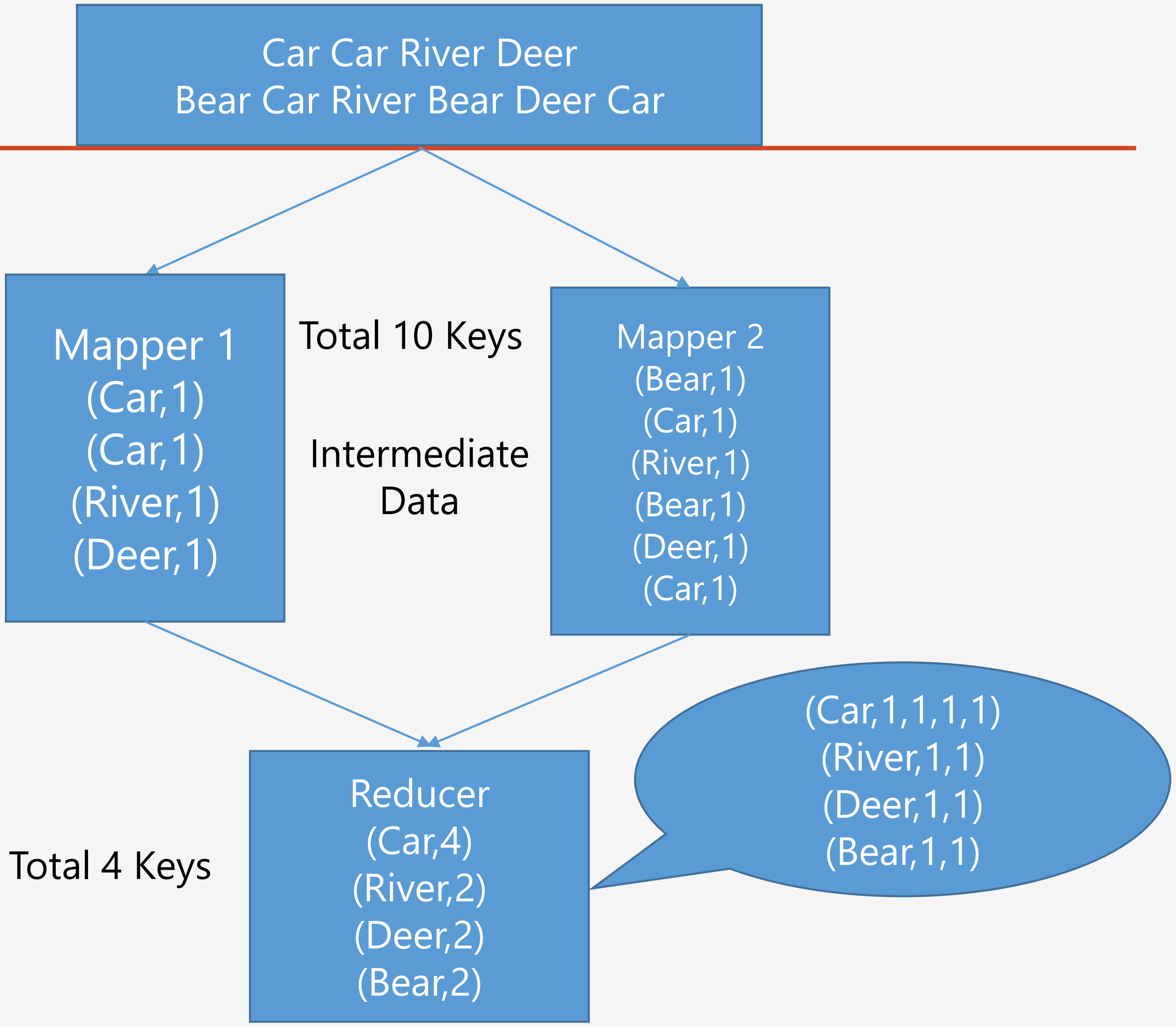
The Combiner class is used in between the Map class and the Reduce class to reduce the volume of data transfer between Map and Reduce. Usually, the output of the map task is large and the data transferred to the reduce task is high.

MapReduce – Combiner

MapReduce Program without Combiner

In the above diagram, no combiner is used. Input is split into two mappers and 10 keys are generated from the mappers. Now we have (10 **key/value**) intermediate data, the further mapper will send directly this data to reducer and while sending data to the reducer, it consumes some network bandwidth (bandwidth means time taken to transfer data between 2 machines). It will take more time to transfer data to reducer if the size of data is big.

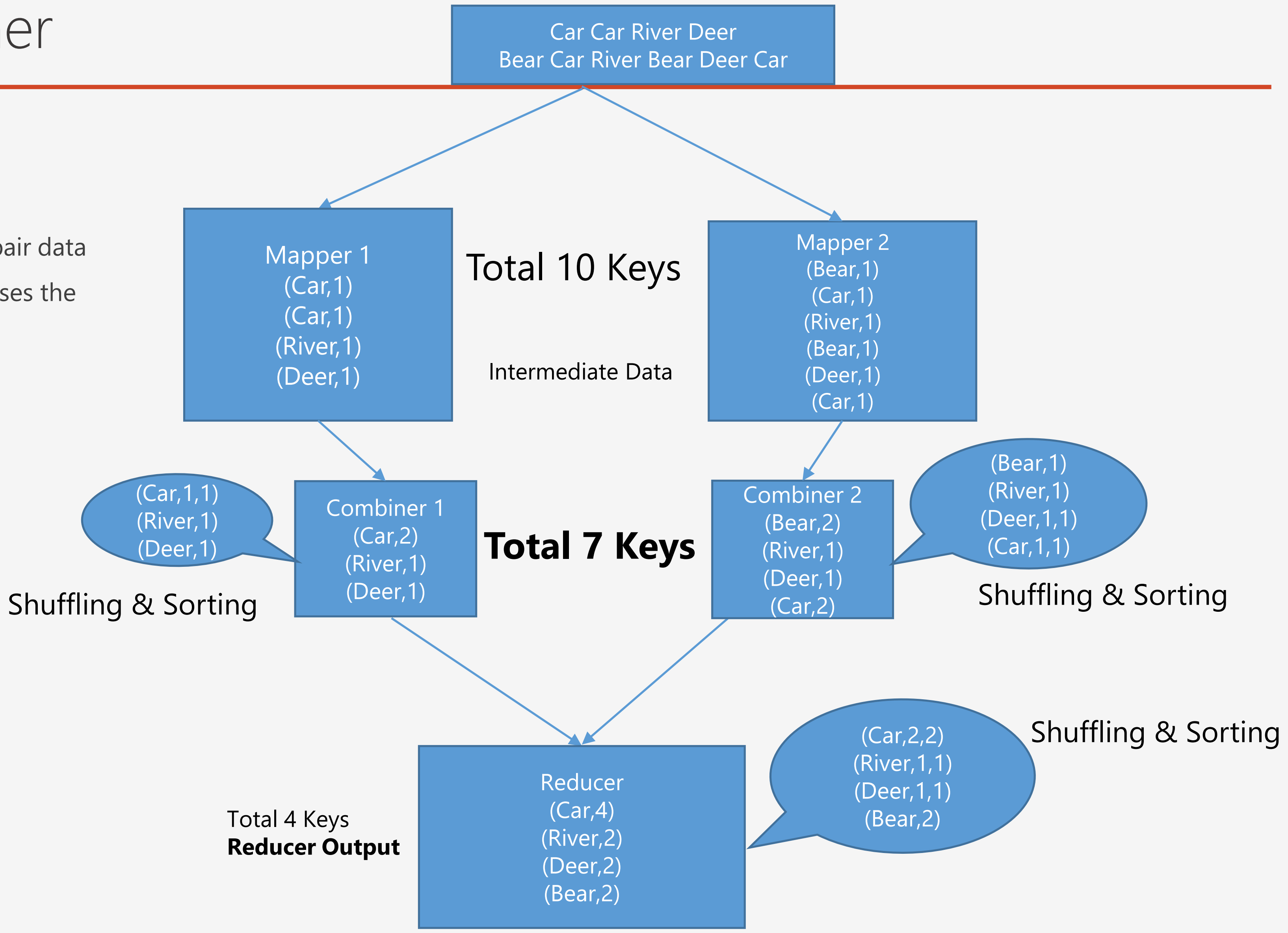
Now in between mapper and reducer if we use a hadoop combiner, then combiner shuffles intermediate data (10 key/value) before sending it to the reducer and generates 7 key/value pair as an output.



MapReduce – Combiner

MapReduce Program with Combiner

Reducer now needs to process only 7 key/value pair data which is generated from 2 combiners. This increases the overall performance.


Car Car River Deer
Bear Car River Bear Deer Car
Mapper 1 (Car,1) (Car,1) (River,1) (Deer,1)
Total 10 Keys
Intermediate Data
Mapper 2 (Bear,1) (Car,1) (River,1) (Bear,1) (Deer,1) (Car,1)
(Car,1,1) (River,1) (Deer,1)
Combiner 1 (Car,2) (River,1) (Deer,1)
Total 7 Keys
Combiner 2 (Bear,2) (River,1) (Deer,1) (Car,2)
(Bear,1) (River,1) (Deer,1,1) (Car,1,1)
Shuffling & Sorting
Shuffling & Sorting
Shuffling & Sorting
(Car,2,2) (River,1,1) (Deer,1,1) (Bear,2)
Total 4 Keys
Reducer Output
Reducer (Car,4) (River,2) (Deer,2) (Bear,2)

MapReduce – Combiner

Advantages of MapReduce Combiner

As we have discussed what is Hadoop MapReduce Combiner in detail, now we will discuss some advantages of Mapreduce Combiner.

- Hadoop Combiner reduces the time taken for data transfer between mapper and reducer.
- It decreases the amount of data that needed to be processed by the reducer.
- The Combiner improves the overall performance of the reducer.

Disadvantages of Hadoop combiner in MapReduce

There are also some disadvantages of hadoop Combiner. Let’s discuss them one by one-

- MapReduce jobs cannot depend on the Hadoop combiner execution because there is no guarantee in its execution.
- In the local filesystem, the key-value pairs are stored in the Hadoop and run the combiner later which will cause expensive disk IO.

Shuffling and Sorting in Hadoop MapReduce

In **Hadoop**, the process by which the intermediate output from **mappers** is transferred to the **reducer** is called Shuffling. Reducer gets 1 or more keys and associated values on the basis of reducers. Intermediated **key-value** generated by mapper is sorted automatically by key.

Shuffle phase in Hadoop transfers the map output from Mapper to a Reducer in MapReduce. **Sort phase** in MapReduce covers the merging and sorting of map outputs. Data from the mapper are grouped by the key, split among reducers and sorted by the key. Every reducer obtains all values associated with the same key. Shuffle and sort phase in Hadoop occur simultaneously and are done by the MapReduce framework.

Shuffling and Sorting in Hadoop MapReduce

Shuffling in MapReduce

The process of transferring data from the mappers to reducers is known as shuffling i.e. the process by which the system performs the sort and transfers the map output to the reducer as input. So, MapReduce shuffle phase is necessary for the reducers, otherwise, they would not have any input (or input from every mapper). As shuffling can start even before the map phase has finished so this saves some time and completes the tasks in lesser time.

Sorting in MapReduce

The keys generated by the mapper are automatically sorted by MapReduce Framework, i.e. Before starting of reducer, all intermediate **key-value pairs** in MapReduce that are generated by mapper get sorted by key and not by value. Values passed to each reducer are not sorted; they can be in any order.

Sorting in Hadoop helps reducer to easily distinguish when a new reduce task should start. This saves time for the reducer. Reducer starts a new reduce task when the next key in the sorted input data is different than the previous. Each reduce task takes key-value pairs as input and generates key-value pair as output.

Note that shuffling and sorting in Hadoop MapReduce is not performed at all if you specify zero reducers (setNumReduceTasks(0)). Then, the MapReduce job stops at the map phase, and the map phase does not include any kind of sorting (so even the map phase is faster).