**UNIT – 4 BIG DATA**

What is MapReduce?

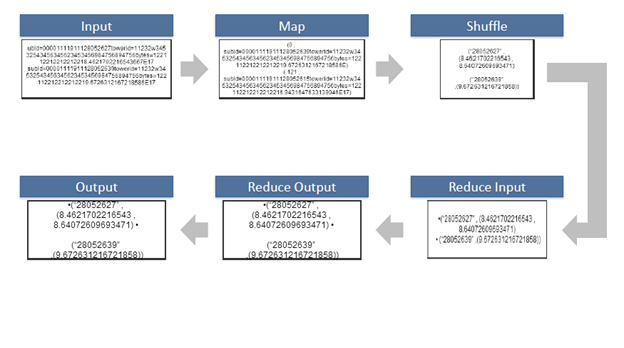
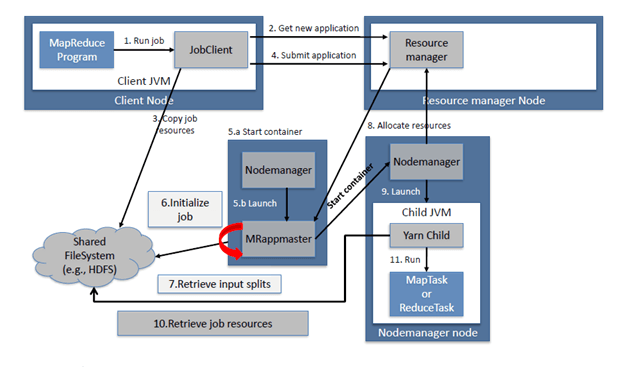
A MapReduce is a data processing tool which is used to process the data parallelly in a distributed form. It was developed in 2004, on the basis of paper titled as "MapReduce: Simplified Data Processing on Large Clusters," published by Google.

The MapReduce is a paradigm which has two phases, the mapper phase, and the reducer phase. In the Mapper, the input is given in the form of a key-value pair. The output of the Mapper is fed to the reducer as input. The reducer runs only after the Mapper is over. The reducer too takes input in key-value format, and the output of reducer is the final output.

Backward Skip 10sPlay VideoForward Skip 10s

Steps in Map Reduce

* The map takes data in the form of pairs and returns a list of <key, value> pairs. The keys will not be unique in this case.
* Using the output of Map, sort and shuffle are applied by the Hadoop architecture. This sort and shuffle acts on these list of <key, value> pairs and sends out unique keys and a list of values associated with this unique key <key, list(values)>.
* An output of sort and shuffle sent to the reducer phase. The reducer performs a defined function on a list of values for unique keys, and Final output <key, value> will be stored/displayed.

**Explanation of MapReduce Program**

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

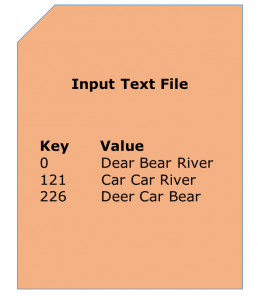
* Mapper Phase Code
* Reducer Phase Code
* Driver Code

We will understand the code for each of these three parts sequentially.

**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));  
}

* 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:

1. The *key* is nothing but the offset of each line in the text file: *LongWritable*
2. The *value* is each individual line (as shown in the figure at the right): *Text*

* Output:

1. The *key* is the tokenized words: *Text*
2. We have the hardcoded *value* in our case which is 1: *IntWritable*
3. 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*.

**Reducer Code:**

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

* 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:

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

* Output:

1. The *key* is all the unique words present in the input text file: *Text*
2. The *value* is the number of occurrences of each of the unique words: *IntWritable*
3. 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.

**Driver Code:**

Configuration conf= new Configuration();  
Job job = new Job(conf,"My Word Count Program");  
job.setJarByClass(WordCount.class);  
job.setMapperClass(Map.class);  
job.setReducerClass(Reduce.class);  
job.setOutputKeyClass(Text.class);  
   
job.setOutputValueClass(IntWritable.class);  
job.setInputFormatClass(TextInputFormat.class);  
job.setOutputFormatClass(TextOutputFormat.class);  
Path outputPath = new Path(args[1]);  
   
//Configuring the input/output path from the filesystem into the job  
FileInputFormat.addInputPath(job, new Path(args[0]));  
FileOutputFormat.setOutputPath(job, new Path(args[1]));

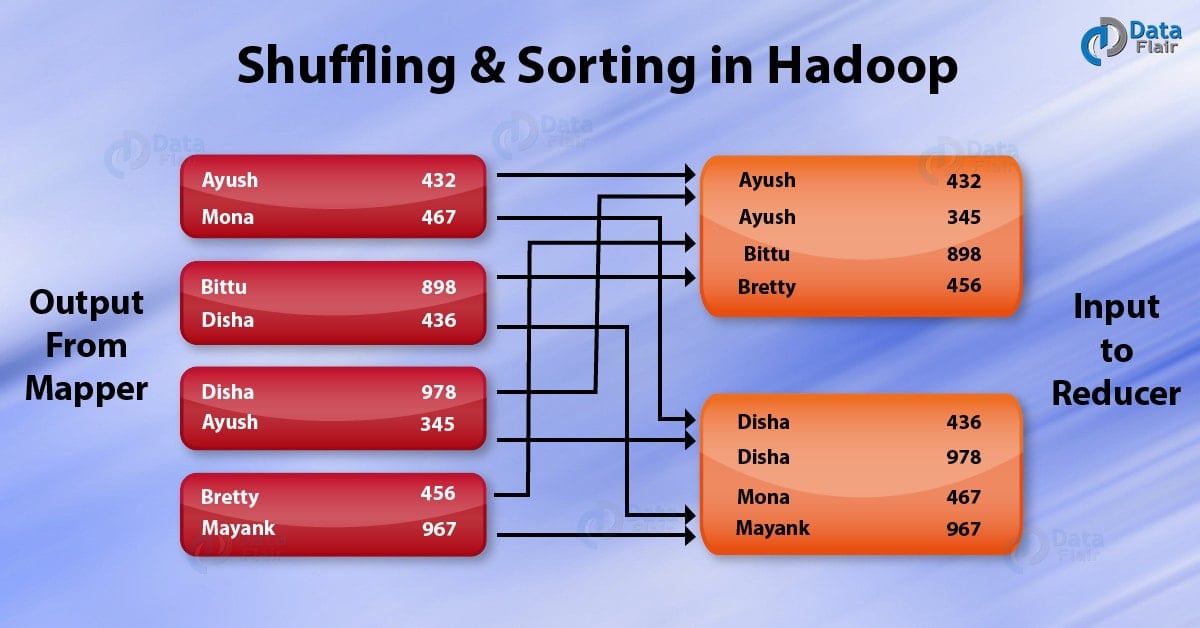
* In the driver class, we set the configuration of our MapReduce job to run in Hadoop.
* We specify the name of the job, the data type of input/output of the mapper and reducer.
* We also specify the names of the mapper and reducer classes.
* The path of the input and output folder is also specified.
* 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.

# Shuffling and Sorting in Hadoop

## **1. Objective**

In [**Hadoop**](https://data-flair.training/blogs/hadoop-tutorial-for-beginners/), 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. In this blog, we will discuss in detail about shuffling and Sorting in **Hadoop** **MapReduce**.

Here we will learn what is sorting in Hadoop, what is shuffling in Hadoop, what is the purpose of Shuffling and sorting phase in[**MapReduce**](https://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/), how MapReduce shuffle works and how MapReduce sort works. We will also learn what is secondary sorting in MapReduce?

[](https://data-flair.training/blogs/wp-content/uploads/sites/2/2017/04/Shuffling-Sorting-in-hadoop-01-1.jpg)

*Shuffling and Sorting in Hadoop MapReduce*

## **2. What is Shuffling and Sorting in Hadoop MapReduce?**

Before we start with Shuffle and Sort in MapReduce, let us revise the other phases of MapReduce like[**Mapper**](https://data-flair.training/blogs/mapper-in-hadoop-mapreduce/), [**reducer**](https://data-flair.training/blogs/reducer-in-hadoop-mapreduce/) in MapReduce, [**Combiner**](https://data-flair.training/blogs/combiner-in-hadoop-mapreduce-advantages-disadvantages/), [**partitioner in MapReduce**](https://data-flair.training/blogs/partitioner-in-hadoop-mapreduce-hadoop-internals/)and[**inputFormat in MapReduce.**](https://data-flair.training/blogs/hadoop-inputformat-types/)

**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.

Let us now understand both these processes in details below:

## **3. 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.

## **4. Sorting in MapReduce**

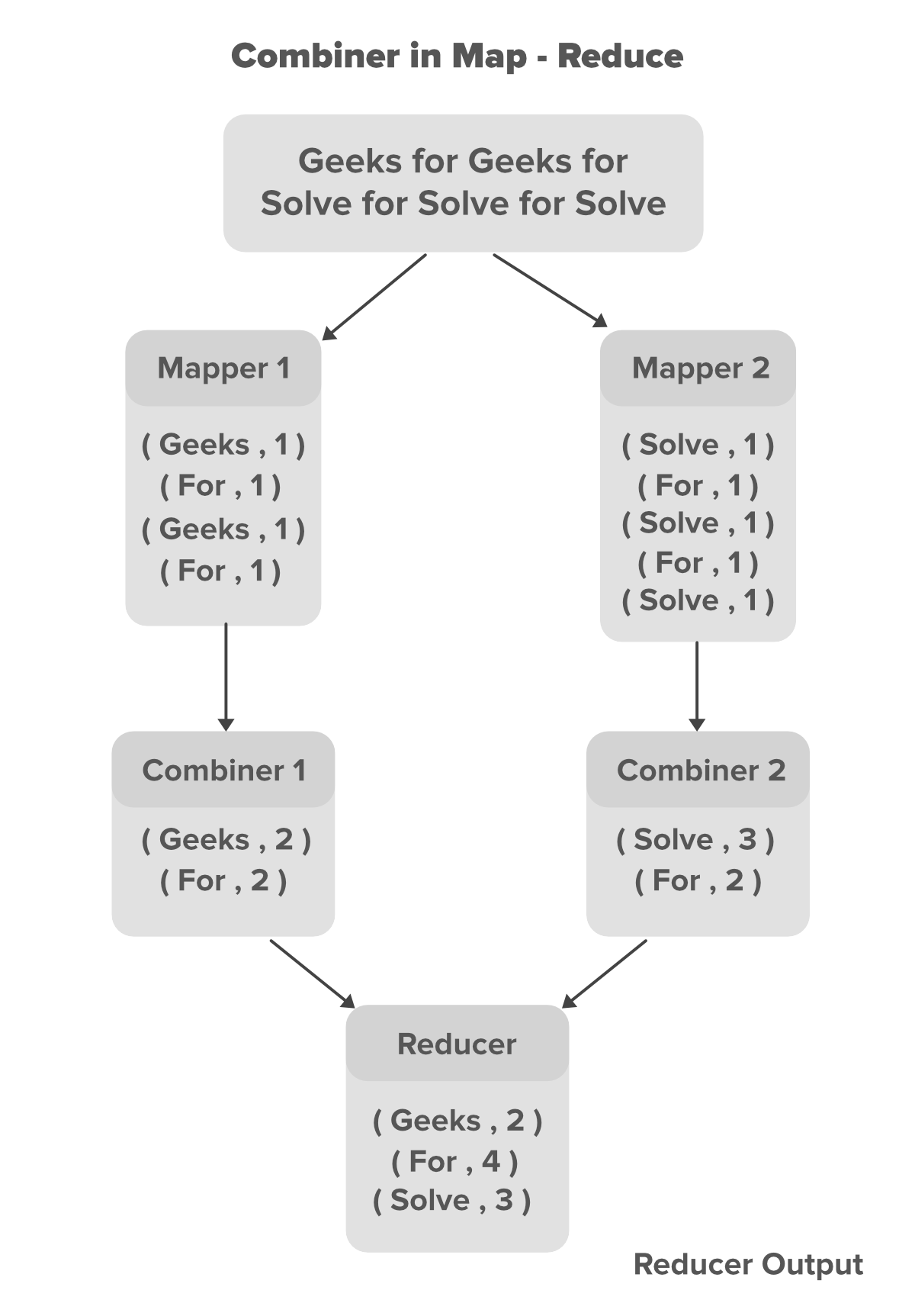
The keys generated by the mapper are automatically sorted by MapReduce Framework, i.e. Before starting of reducer, all intermediate [**key-value pairs**](https://data-flair.training/blogs/key-value-pairs-hadoop-mapreduce/)in MapReduce that are generated by mapper get sorted by key and not by value. Values 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.

**What is a combiner?**

**Combiner** always works in between Mapper and Reducer. The output produced by the Mapper is the intermediate output in terms of key-value pairs which is massive in size. If we directly feed this huge output to the Reducer, then that will result in increasing the **Network Congestion**. So to minimize this Network congestion we have to put combiner in between Mapper and Reducer. These combiners are also known as **semi-reducer**. It is not necessary to add a combiner to your Map-Reduce program, it is optional. Combiner is also a class in our java program like **Map** and **Reduce** class that is used in between this **Map** and **Reduce** classes. Combiner helps us to produce abstract details or a summary of very large datasets. When we process or deal with very large datasets using Hadoop Combiner is very much necessary, resulting in the enhancement of overall performance.

**How does combiner work?**



In the above example, we can see that two Mappers are containing different data. the main text file is divided into two different Mappers. Each mapper is assigned to process a different line of our data. in our above example, we have two lines of data so we have two Mappers to handle each line. Mappers are producing the intermediate key-value pairs, where the name of the particular word is **key** and its count is its **value**. For example for the data **Geeks For Geeks For** the key-value pairs are shown below.

// Key Value pairs generated for data Geeks For Geeks For

(Geeks,1)

(For,1)

(Geeks,1)

(For,1)

The key-value pairs generated by the Mapper are known as the intermediate key-value pairs or intermediate output of the Mapper. Now we can minimize the number of these key-value pairs by introducing a **combiner** for each Mapper in our program. In our case, we have 4 key-value pairs generated by each of the Mapper. since these intermediate key-value pairs are not ready to directly feed to Reducer because that can increase Network congestion so **Combiner** will combine these intermediate key-value pairs before sending them to Reducer. The combiner combines these intermediate key-value pairs as per their **key**. For the above example for data **Geeks For Geeks For** the combiner will partially reduce them by merging the same pairs according to their **key** value and generate new key-value pairs as shown below.

// Partially reduced key-value pairs with combiner

(Geeks,2)

(For,2)

With the help of Combiner, the Mapper output got partially reduced in terms of size(key-value pairs) which now can be made available to the Reducer for better performance. Now the Reducer will again Reduce the output obtained from combiners and produces the final output that is stored on [HDFS(Hadoop Distributed File System)](https://www.geeksforgeeks.org/introduction-to-hadoop-distributed-file-systemhdfs/).

**Advantage of combiners**

* Reduces the time taken for transferring the data from  Mapper to  Reducer.
* Reduces the size of the intermediate output generated by the Mapper.
* Improves performance by minimizing Network congestion.
* Reduces the workload on the Reducer: Combiners can help reduce the amount of data that needs to be processed by the Reducer. By performing some aggregation or reduction on the data in the Mapper phase itself, combiners can reduce the number of records that are passed on to the Reducer, which can help improve overall performance.

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).

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.

**Partitioner**

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.

**MapReduce Partitioner Implementation**

For the sake of convenience, let us assume we have a small table called Employee with the following data. We will use this sample data as our input dataset to demonstrate how the partitioner works.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Id** | **Name** | **Age** | **Gender** | **Salary** |
| 1201 | gopal | 45 | Male | 50,000 |
| 1202 | manisha | 40 | Female | 50,000 |
| 1203 | khalil | 34 | Male | 30,000 |
| 1204 | prasanth | 30 | Male | 30,000 |
| 1205 | kiran | 20 | Male | 40,000 |
| 1206 | laxmi | 25 | Female | 35,000 |
| 1207 | bhavya | 20 | Female | 15,000 |
| 1208 | reshma | 19 | Female | 15,000 |
| 1209 | kranthi | 22 | Male | 22,000 |
| 1210 | Satish | 24 | Male | 25,000 |
| 1211 | Krishna | 25 | Male | 25,000 |
| 1212 | Arshad | 28 | Male | 20,000 |
| 1213 | lavanya | 18 | Female | 8,000 |

We have to write an application to process the input dataset to find the highest salaried employee by gender in different age groups (for example, below 20, between 21 to 30, above 30).

Input Data

The above data is saved as **input.txt** in the “/home/hadoop/hadoopPartitioner” directory and given as input.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1201 | gopal | 45 | Male | 50000 |
| 1202 | manisha | 40 | Female | 51000 |
| 1203 | khaleel | 34 | Male | 30000 |
| 1204 | prasanth | 30 | Male | 31000 |
| 1205 | kiran | 20 | Male | 40000 |
| 1206 | laxmi | 25 | Female | 35000 |
| 1207 | bhavya | 20 | Female | 15000 |
| 1208 | reshma | 19 | Female | 14000 |
| 1209 | kranthi | 22 | Male | 22000 |
| 1210 | Satish | 24 | Male | 25000 |
| 1211 | Krishna | 25 | Male | 26000 |
| 1212 | Arshad | 28 | Male | 20000 |
| 1213 | lavanya | 18 | Female | 8000 |

Based on the given input, following is the algorithmic explanation of the program.

Map Tasks

The map task accepts the key-value pairs as input while we have the text data in a text file. The input for this map task is as follows −

**Input** − The key would be a pattern such as “any special key + filename + line number” (example: key = @input1) and the value would be the data in that line (example: value = 1201 \t gopal \t 45 \t Male \t 50000).

**Method** − The operation of this map task is as follows −

* Read the **value** (record data), which comes as input value from the argument list in a string.
* Using the split function, separate the gender and store in a string variable.

String[] str = value.toString().split("\t", -3);

String gender=str[3];

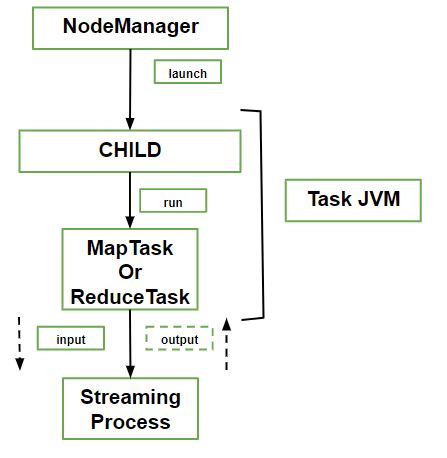
* Send the gender information and the record data **value** as output key-value pair from the map task to the **partition task**.

context.write(new Text(gender), new Text(value));

* Repeat all the above steps for all the records in the text file.

**Output** − You will get the gender data and the record data value as key-value pairs.

**MapReduce Job Execution**

Once the resource manager’s scheduler assign a resources to the task for a container on a particular node, the container is started up by the application master by contacting the node manager. The task whose main class is **YarnChild** is executed by a Java application .  
It localizes the resources that the task needed before it can run the task. It includes the job configuration, any files from the distributed cache and JAR file. It finally runs the map or the reduce task. Any kind of bugs in the user-defined map and reduce functions (or even in YarnChild) don’t affect the node manager as YarnChild runs in a dedicated JVM. So it can’t be affected by a crash or hang.  
All actions running in the same JVM as the task itself are performed by each task setup. These are determined by the **OutputCommitter** for the job. The commit action moves the task output to its final location from its initial position for a file-based jobs. When speculative execution is enabled, the commit protocol ensures that only one of the duplicate tasks is committed and the other one is aborted.  
  
 **What does Streaming means?**  
Streaming reduce tasks and runs special map for the purpose of launching the user supplied executable and communicating with it. Using standard input and output streams, it communicates with the process. The Java process passes input key-value pairs to the external process during execution of the task. It runs the process through the user-defined map or reduce function and passes the output key-value pairs back to the Java process.  
It is as if the child process ran the map or reduce code itself from the manager’s point of view. MapReduce jobs can take anytime from tens of second to hours to run, that’s why are long-running batches. It’s important for the user to get feedback on how the job is progressing because this can be a significant length of time. Each job including the task has a status including the state of the job or task, values of the job’s counters, progress of maps and reduces and the description or status message. These statuses change over the course of the job.  
The task keeps track of its progress when a task is running like a part of the task is completed. This is the proportion of the input that has been processed for map tasks. It is a little more complex for the reduce task but the system can still estimate the proportion of the reduce input processed. When a task is running, it keeps track of its progress (i.e., the proportion of the task completed). For map tasks, this is the proportion of the input that has been processed. For reduce tasks, it’s a little more complex, but the system can still estimate the proportion of the reduce input processed.

**Process involved –**

* Read an input record in a mapper or reducer.
* Write an output record in a mapper or reducer.
* Set the status description.
* Increment a counter using Reporter’s incrCounter() method or Counter’s increment() method.
* Call Reporter’s or TaskAttemptContext’s progress() method.

## **Introduction to Hive**

Here, let’s have a look at the birth of Hive and what exactly Hive is.

### **Birth of Hive**

Facebook played an active role in the birth of Hive as Facebook uses Hadoop to handle Big Data. Hadoop uses MapReduce to process data. Previously, users needed to write lengthy, complex codes to process and analyze data. Not everyone was well-versed in Java and other complex programming languages. On the other hand, many individuals were comfortable with writing queries in SQL. For this reason, there was a need to develop a language similar to SQL, which was well-known to all users. This is how the Hive Query Language, also known as HiveQL, came to be.

### **What is Hive in Hadoop?**

Hive is a data warehouse system used to query and analyze large datasets stored in HDFS. Hive uses a query language called HiveQL, which is similar to SQL.



               Fig: Hive operation

The image above demonstrates a user writing queries in the HiveQL language, which is then converted into MapReduce tasks. Next, the data is processed and analyzed. HiveQL works on structured data, such as numbers, addresses, dates, names, and so on. HiveQL allows multiple users to query data simultaneously.

So, what do we do with semi-structured and unstructured data like emails, images, videos? Enter Apache Pig.

## **Introduction to Pig**

Pig also came into existence to solve issues with MapReduce. Let’s take a close look at Apache Pig.

### **Birth of Pig**

Although MapReduce helped process and analyze Big Data faster, it had its flaws. Individuals who were unfamiliar with programming often found it challenging to write lengthy Java codes. Eventually, it became a difficult task to maintain and optimize the code, and as a result, the processing time increased.

This was the reason Yahoo faced problems when it came to processing and analyzing large datasets. Apache Pig was developed to analyze large datasets without using time-consuming and complex Java codes. Pig was explicitly developed for non-programmers.

### **What is Pig in Hadoop?**

Pig is a scripting platform that runs on Hadoop clusters, designed to process and analyze large datasets. Pig uses a language called Pig Latin, which is similar to SQL. This language does not require as much code in order to analyze data. Although it is similar to SQL, it does have significant differences. In Pig Latin, 10 lines of code is equivalent to  200 lines in Java. This, in turn, results in shorter development times.

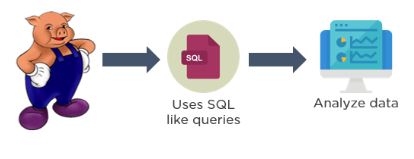


                                                   Fig: Pig operation

What stands out about Pig is that it operates on various types of data, including structured, semi-structured, and unstructured data. Whether you’re working with structured, semi-structured, or unstructured data, Pig takes care of it all.

Many people wonder what makes Pig better than Hive. Hive does have its advantages over Pig in a few ways—and we’ll compare these different features—to help you make a more informed decision when it comes to choosing which platform best suits your requirements.