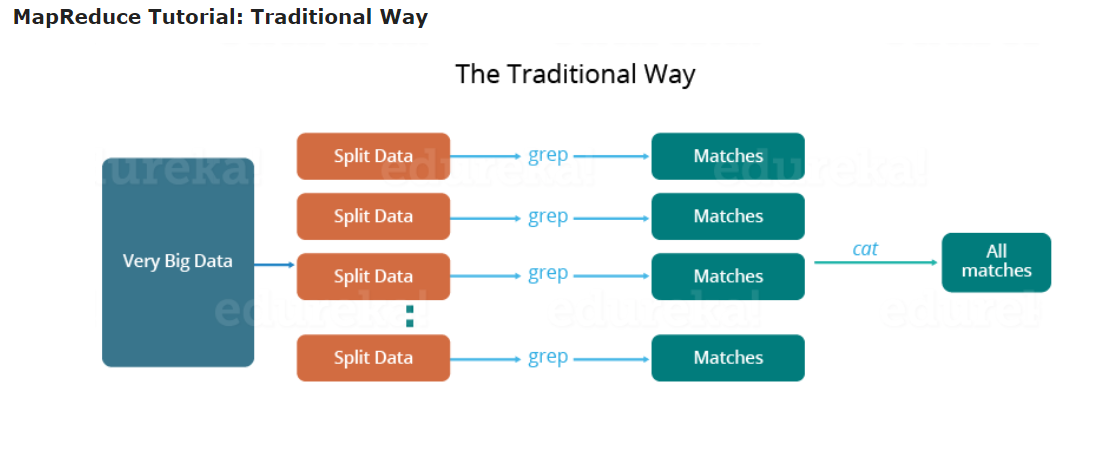
## Understanding the Hadoop MapReduce fundamentals

* Understand MapReduce objects
* Learn how to decide the number of Maps in MapReduce
* Learn how to decide the number of Reduces in MapReduce
* Understand MapReduce dataflow
* Take a closer look at Hadoop MapReduce terminologies

Understanding MapReduce objects

As we know, MapReduce operations in Hadoop are carried out mainly by three objects: Mapper, Reducer, and Driver.

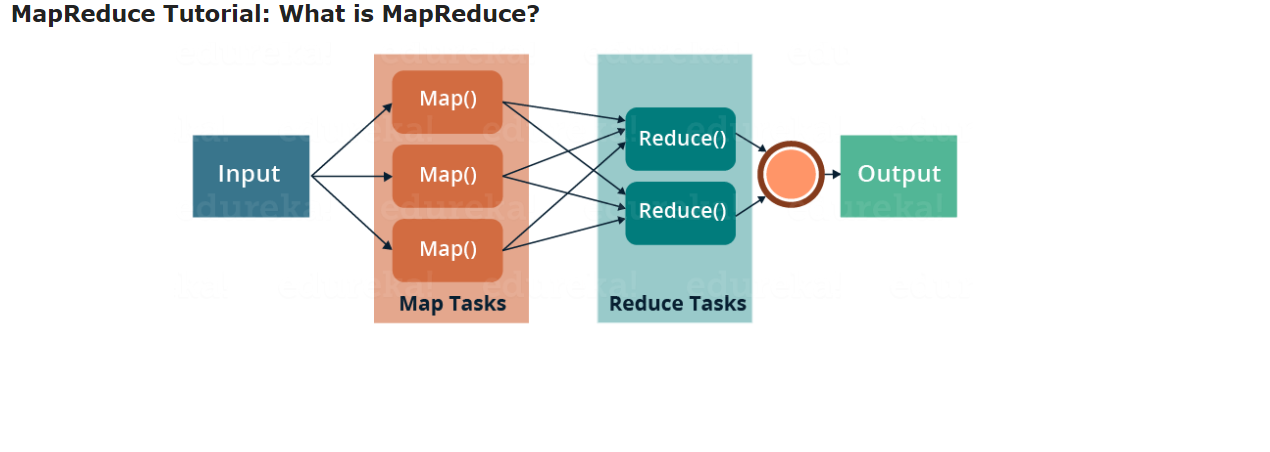
* **Mapper**: This is designed for the Map phase of MapReduce, which starts MapReduce operations by carrying input files and splitting them into several pieces. For each piece, it will emit a key-value data pair as the output value.
* **Reducer**: This is designed for the Reduce phase of a MapReduce job; it accepts key-based grouped data from the Mapper output, reduces it by aggregation logic, and emits the (key, value) pair for the group of values.
* **Driver**: This is the main file that drives the MapReduce process. It starts the execution of MapReduce tasks after getting a request from the client application with parameters. The Driver file is responsible for building the configuration of...



So, just like in the traditional way, I will split the data into smaller parts or blocks and store them in different machines. Then, I will find the highest temperature in each part stored in the corresponding machine. At last, I will combine the results received from each of the machines to have the final output. Let us look at the challenges associated with this traditional approach:

1. **Critical path problem:** It is the amount of time taken to finish the job without delaying the next milestone or actual completion date. So, if, any of the machines delays the job, the whole work gets delayed.
2. **Reliability problem:** What if, any of the machines which is working with a part of data fails? The management of this failover becomes a challenge.
3. **Equal split issue:** How will I divide the data into smaller chunks so that each machine gets even part of data to work with. In other words, how to equally divide the data such that no individual machine is overloaded or under utilized.
4. **Single split may fail:** If any of the machine fails to provide the output, I will not be able to calculate the result. So, there should be a mechanism to ensure this fault tolerance capability of the system.
5. **Aggregation of result:** There should be a mechanism to aggregate the result generated by each of the machines to produce the final output.

These are the issues which I will have to take care individually while performing parallel processing of huge data sets when using traditional approaches.



MapReduce is a programming framework that allows us to perform distributed and parallel processing on large data sets in a distributed environment.

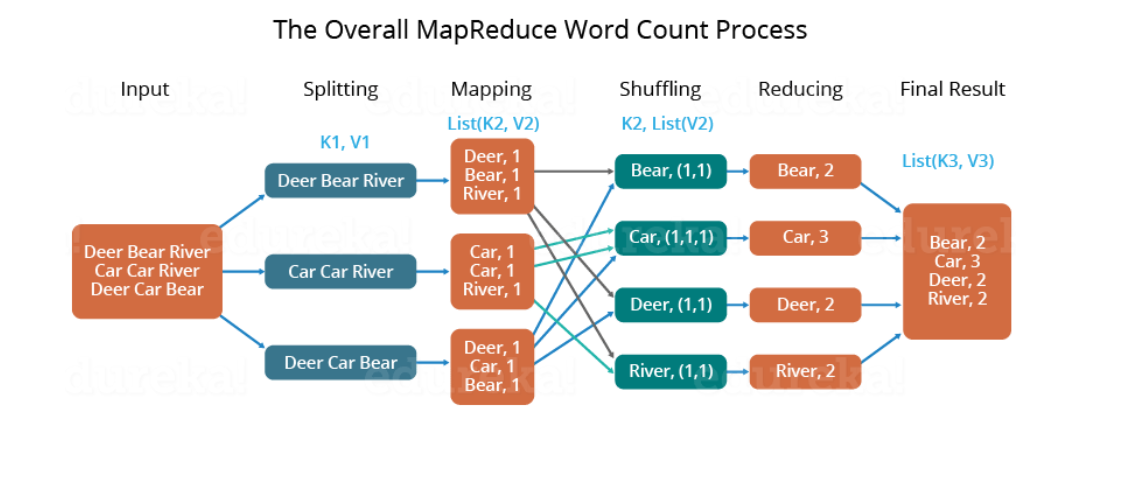
* MapReduce consists of two distinct tasks – Map and Reduce.
* As the name MapReduce suggests, reducer phase takes place after mapper phase has been completed.
* So, the first is the map job, where a block of data is read and processed to produce key-value pairs as intermediate outputs.
* The output of a Mapper or map job (key-value pairs) is input to the Reducer.
* The reducer receives the key-value pair from multiple map jobs.
* Then, the reducer aggregates those intermediate data tuples (intermediate key-value pair) into a smaller set of tuples or key-value pairs which is the final output.

## ****MapReduce Tutorial: A Word Count Example of MapReduce****

Let us understand, how a MapReduce works by taking an example where I have a text file called example.txt whose contents are as follows:

**Dear, Bear, River, Car, Car, River, Deer, Car and Bear**

Now, suppose, we have to perform a word count on the sample.txt using MapReduce. So, we will be finding the unique words and the number of occurrences of those unique words.



* First, we divide the input in three splits as shown in the figure. This will distribute the work among all the map nodes.
* Then, we tokenize the words in each of the mapper and give a hardcoded value (1) to each of the tokens or words. The rationale behind giving a hardcoded value equal to 1 is that every word, in itself, will occur once.
* Now, a list of key-value pair will be created where the key is nothing but the individual words and value is one. So, for the first line (Dear Bear River) we have 3 key-value pairs – Dear, 1; Bear, 1; River, 1. The mapping process remains the same on all the nodes.
* After mapper phase, a partition process takes place where sorting and shuffling happens so that all the tuples with the same key are sent to the corresponding reducer.
* So, after the sorting and shuffling phase, each reducer will have a unique key and a list of values corresponding to that very key. For example, Bear, [1,1]; Car, [1,1,1].., etc.
* Now, each Reducer counts the values which are present in that list of values. As shown in the figure, reducer gets a list of values which is [1,1] for the key Bear. Then, it counts the number of ones in the very list and gives the final output as – Bear, 2.
* Finally, all the output key/value pairs are then collected and written in the output file.

## ****MapReduce Tutorial: MapReduce Example Program****

Before jumping into the details, let us have a glance at a MapReduce example program to have a basic idea about how things work in a MapReduce environment practically. I have taken the same word count example where I have to find out the number of occurrences of each word. And Don’t worry guys, if you don’t understand the code when you look at it for the first time, just bear with me while I walk you through each part of the MapReduce code.

### ****Source code:****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61 | package co.edureka.mapreduce;  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;    public class WordCount  {  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));  }  }  }    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));  }  }    public static void main(String[] args) throws Exception {    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]));  //deleting the output path automatically from hdfs so that we don't have to delete it explicitly  outputPath.getFileSystem(conf).delete(outputPath);  //exiting the job only if the flag value becomes false  System.exit(job.waitForCompletion(true) ? 0 : 1);  }  } |

## ****MapReduce Tutorial: Explanation of MapReduce Program****

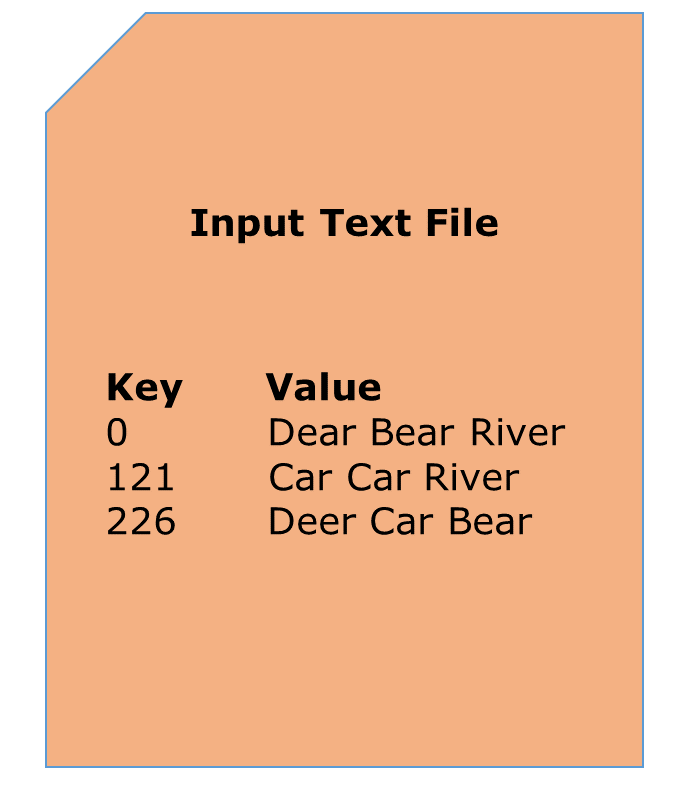
The entire MapReduce program can be fundamentally divided into three parts:

* Mapper Phase Code
* Reducer Phase Code
* Driver Code

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

## ****Mapper code:****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 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:
  + 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.

## ****Reducer Code:****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | 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:
  + 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.

## ****Driver Code:****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | 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.

## ****Run the MapReduce code:****

The command for running a MapReduce code is:

hadoop jar hadoop-mapreduce-example.jar WordCount /sample/input /sample/output

**THE BIG DATA PROBLEM**

We live in the age of big data. It is not easy to measure the total volume of data stored electronically, but a recent analyst estimate puts the size of the “The Digital Universe” at 4.4 zettabytes in 2013. The same firm is forecasting a tenfold growth by 2020 to 44 zettabytes - which in more familiar terms is 44 billion terabytes.  This flood of data is coming from a myriad of sources. For example:

* The New York Stock Exchange generates about four terabytes of new trade data per day.
* Facebook hosts approximately 240 billion pictures, growing seven petabytes per month.
* Ancestry.com, the genealogy site, stores around 10 petabytes of data.
* The Internet Archive stores around 18.5 petabytes of data.
* The Large Hadron Collider processes one petabyte of data every day - the equivalent of around 210,000 DVDs.

**ACCESS TIME IS A KEY ISSUE**

The problem is that although the storage capacities of hard drives have increased massively over the years, access speeds - the rates at which data can be read from drives - have not kept up. A typical drive from 1990 could store 1.4 MB of data, and had a transfer speed of 4.4 MB/s, so you could read all the data from a full drive in around five minutes. Today, more 20 years later, one terabyte drives are commonplace, but with typical transfer speeds of around 100 MB/s, it takes more than two and a half hours to read all of the data.

The obvious way to reduce access times is to read from multiple disks at once. Imagine if we had 100 drives, each holding one hundredth of the data. Working in parallel, we could then read that terabyte of data in less than two minutes.

Using only one hundredth of a disk seems wasteful. However, we can of course store many datasets on the disk array. We can imagine that the users of such a system would be happy to share access in return for shorter read-times, and statistically, that read requests will likely be spread over time, so they won't interfere with each other too much. This is what big data storage systems such as HDFS (the Hadoop Distributed File System) provide.

But there is more to Big Data than simply being able to access it conveniently.

**DISTRIBUTED PROCESSING**

Hadoop MapReduce is a software framework for creating applications that process large datasets in parallel, on clusters of commodity hardware in a reliable, fault-tolerant manner.

A MapReduce job splits ("maps") the input dataset into independent chunks, which are processed on different nodes in the cluster, in a parallel manner. The framework then sorts the outputs of each chunk, which are merged ("reduced") back together. Usually, both the input and the output of the job are stored in a distributed filesystem.

Typically the compute nodes doing the processing and the distributed storage nodes are the same commodity servers. In other words, the MapReduce framework and the Hadoop Distributed File System (HDFS) are running on the same cluster of servers. This configuration enables the framework to efficiently schedule tasks on the nodes where data is already present, resulting in high aggregate transfer bandwidth across the cluster.

That, in a nutshell, is what Hadoop provides; a reliable, scalable platform for the storage and analysis of large datasets. Further, rather than relying on hardware robustness to deliver high-availability, the system is designed to detect and handle hardware failures, thus delivering a highly-available service even when using cheap, commodity servers.

**A FRAMEWORK FOR MANY APPLICATIONS**

The use of Hadoop is most strongly associated with data analysis, and applications that provide insight to support decisions. However, any application that requires the storage and processing of large data sets can benefit from Hadoop. For example, at Search Technologies we are finding Hadoop to be a valueable tool for creating highly accurate search applications.

**ENGAGING WITH HADOOP**

The Hadoop framework is implemented in Java, and MapReduce applications can be developed in Java or any JVM-based language. It is designed to scale up from a single server to thousands of machines, each offering local computation and storage. What’s more, Hadoop is affordable since it runs on commodity hardware, and is an open source framework. Commercial packaging and support for production systems is available from companies such as Cloudera.

If you have a Hadoop application in mind and need technical expertise, feel free to contact us for an informal discussion. Search Technologies provides expert implementation services for Hadoop projects.