**UNIT-3**

**MapReduce Technique**

**TOPICS:**

How MapReduce works?, Anatomy of a Map Reduce Job Run, Failures, A Word Count Example of MapReduce.

**What is MapReduce?**

Apache MapReduce is the processing engine of Hadoop that processes and computes vast volumes of data. [MapReduce](https://www.simplilearn.com/tutorials/hadoop-tutorial/mapreduce) programming paradigm allows you to scale unstructured data across hundreds or thousands of commodity servers in an Apache Hadoop cluster.

* **MapReduce** is the processing layer of **Hadoop**. MapReduce programming model is designed for processing large volumes of data in parallel by dividing the work into a set of independent tasks.
* You need to put business logic in the way MapReduce works and rest things will be taken care by the framework.
* Work (complete job) which is submitted by the user to master is divided into small works (tasks) and assigned to slaves.  
  MapReduce programs are written in a particular style influenced by functional programming constructs, specifical idioms for processing lists of data.
* Here in MapReduce, we get inputs from a list and it converts it into output which is again a list. It is the heart of Hadoop.
* Hadoop is so much powerful and efficient due to MapRreduce as here parallel processing is done.
* This is what MapReduce is in Big Data. In the next step of Mapreduce Tutorial we have MapReduce Process, MapReduce dataflow how MapReduce divides the work into sub-work, why MapReduce is one of the best paradigms to process data.

1. **The Map Phase**
2. **The Reduce Phase**

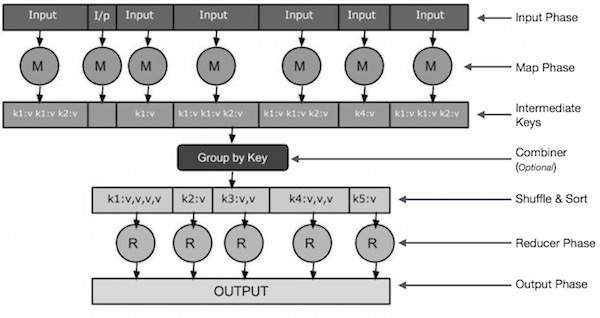
**How MapReduce Works?**

The MapReduce algorithm contains two important tasks, namely Map and Reduce.

* The Map task takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key-value pairs).
* The Reduce task takes the output from the Map as an input and combines those data tuples (key-value pairs) into a smaller set of tuples.

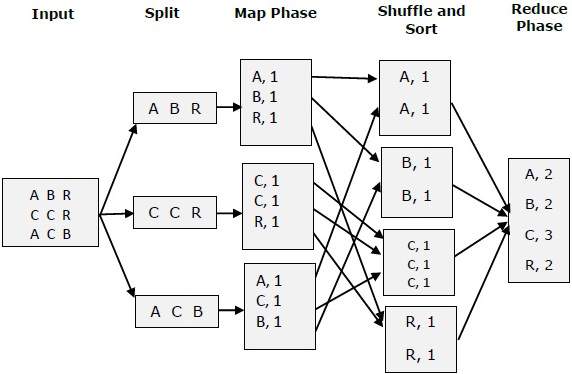
The reduce task is always performed after the map job.

Let us now take a close look at each of the phases and try to understand their significance.

****

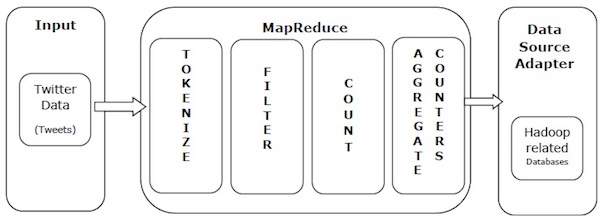
* Input Phase − Here we have a Record Reader that translates each record in an input file and sends the parsed data to the mapper in the form of key-value pairs.
* Map − Map is a user-defined function, which takes a series of key-value pairs and processes each one of them to generate zero or more key-value pairs.
* Intermediate Keys − They key-value pairs generated by the mapper are known as intermediate keys.
* Combiner − A combiner is a type of local Reducer that groups similar data from the map phase into identifiable sets. It takes the intermediate keys from the mapper as input and applies a user-defined code to aggregate the values in a small scope of one mapper. It is not a part of the main MapReduce algorithm; it is optional.
* Shuffle and Sort − The Reducer task starts with the Shuffle and Sort step. It downloads the grouped key-value pairs onto the local machine, where the Reducer is running. The individual key-value pairs are sorted by key into a larger data list. The data list groups the equivalent keys together so that their values can be iterated easily in the Reducer task.
* Reducer − The Reducer takes the grouped key-value paired data as input and runs a Reducer function on each one of them. Here, the data can be aggregated, filtered, and combined in a number of ways, and it requires a wide range of processing. Once the execution is over, it gives zero or more key-value pairs to the final step.
* Output Phase − In the output phase, we have an output formatter that translates the final key-value pairs from the Reducer function and writes them onto a file using a record writer.

Let us try to understand the two tasks Map & Reduce with the help of a small diagram −



**MapReduce-Example**

Let us take a real-world example to comprehend the power of MapReduce. Twitter receives around 500 million tweets per day, which is nearly 3000 tweets per second. The following illustration shows how Tweeter manages its tweets with the help of MapReduce.



As shown in the illustration, the MapReduce algorithm performs the following actions −

* Tokenize − Tokenizes the tweets into maps of tokens and writes them as key-value pairs.
* Filter − Filters unwanted words from the maps of tokens and writes the filtered maps as key-value pairs.
* Count − Generates a token counter per word.
* Aggregate Counters − Prepares an aggregate of similar counter values into small manageable units.

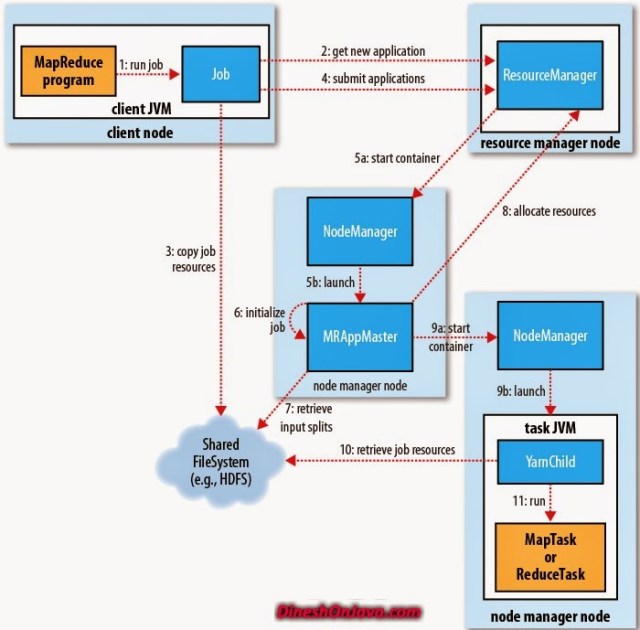
**Anatomy of a Map Reduce Job Run**

To run a MapReduce job with a single method call: submit() on a Job object (you

can also call waitForCompletion(), which submits the job if it hasn’t been submitted already, then waits for it to finish.

Five Independent Entities:

* The client, which submits the MapReduce job.
* The YARN resource manager, which coordinates the allocation of compute resources on the cluster.
* The YARN node managers, which launch and monitor the compute containers on machines in the cluster.
* The MapReduce application master, which coordinates the tasks running the MapReduce job.
* The application master and the MapReduce tasks run in containers that are scheduled by the resource manager and managed by the node managers. The distributed filesystem,which is used for sharing job files between the other entities.





**Job Submission:**

* The submit() method on Job creates an internal JobSubmitter instance and calls submitJobInternal() on it .
* Having submitted the job, waitForCompletion() polls the job’s progress once per second and reports the progress to the console if it has changed since the last report.
* When the job completes successfully, the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.

The job submission process implemented by JobSubmitter does the following:

* Asks the resource manager for a new application ID, used for the MapReduce job ID (step 2).
* Checks the output specification of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.
* Computes the input splits for the job. If the splits cannot be computed (because the input paths don’t exist, for example), the job is not submitted and an error is thrown to the MapReduce program.
* Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the shared filesystem in a directory named after the job ID (step 3).
* The job JAR is copied with a high replication factor (controlled by the mapreduce.client.submit.file.replication property, which defaults to 10) so that there are lots of copies across the cluster for the node managers to access when they run tasks for the job.
* Submits the job by calling submitApplication() on the resource manager (step 4).

Job Initialization:

* When the resource manager receives a call to its submitApplication() method, it hands off the request to the YARN scheduler.
* The scheduler allocates a container, and the resource manager then launches the application master’s process there, under the node manager’s management (steps 5a and 5b).
* The application master for MapReduce jobs is a Java application whose main class is MRAppMaster.
* It initializes the job by creating a number of bookkeeping objects to keep track of the job’s progress, as it will receive progress and completion reports from the tasks (step 6).
* it retrieves the input splits computed in the client from the shared filesystem (step 7).

Task Assignment:

* If the job does not qualify for running as an uber task, then the application master requests containers for all the map and reduce tasks in the job from the resource manager (step 8).
* Requests for map tasks are made first and with a higher priority than those for reduce tasks, since all the map tasks must complete before the sort phase of the reduce can start (see Shuffle and Sort).
* Requests for reduce tasks are not made until 5% of map tasks have completed (see Reduce slow start).

Task Execution:

* Once a task has been assigned resources for a container on a particular node by the resource manager’s scheduler, the application master starts the container by contacting the node manager (steps 9a and 9b).
* The task is executed by a Java application whose main class is YarnChild. Before it can run the task, it localizes the resources that the task needs, including the job configuration and JAR file, and any files from the distributed cache (step 10; see Distributed Cache). Finally, it runs the map or reduce task (step 11).

Streaming:

* Streaming runs special map and reduce tasks for the purpose of launching the user-supplied executable and communicating with it.
* The Streaming task communicates with the process (which may be written in any language) using standard input and output streams.

Job Completion:

* When the application master receives a notification that the last task for a job is complete, it changes the status for the job to “successful.”
* Then, when the Job polls for status, it learns that the job has completed successfully, so it prints a message to tell the user and then returns from the waitForCompletion() method.
* Job statistics and counters are printed to the console at this point.

**Failures:**

**Hadoop Processing Generation1 Failures:**

1. The mapper May
2. The Reducer May
3. The Task Tracker May
4. The Job tracker May

**Hadoop Processing Generation2 Failures:**

1. The Mapper May
2. The Reducer May
3. The Node Manager May
4. The Application Master May
5. The resource Manager May

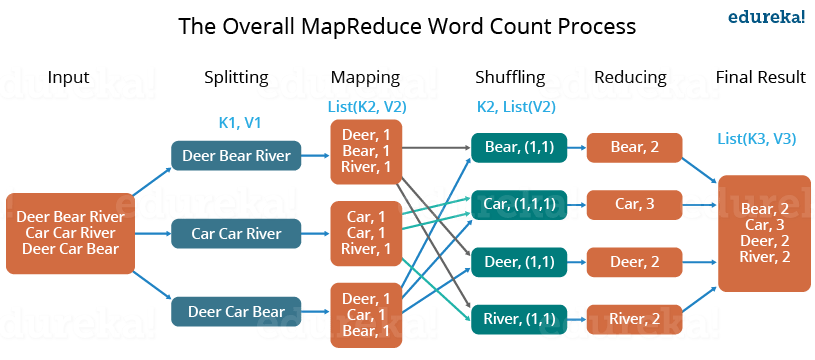
**MapReduce Example:**

**A Word Count Example of MapReduce:**

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 into 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 mappers 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 the mapper phase, a partition process takes place where sorting and shuffling happen 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.

**Advantages of using MapReduce**

The advantages of using MapReduce are as follows:

* MapReduce can define mapper and reducer in several different languages using Hadoop streaming.
* MapReduce facilitates automatic parallelization and distribution, reducing the time required to run programs.
* MapReduce provides fault tolerance by re-executing, writing map output to a distributed file system, and restarting failed map or reducer tasks.
* Processing of data using MapReduce is a cost-effective solution.
* MapReduce processes large volumes of unstructured data very quickly.
* Using HDFS and HBase security, MapReduce ensures data security by allowing only approved users to access data stored in the system.
* MapReduce programming utilizes a simple programming model to handle tasks more efficiently and quickly and is easy to learn.
* MapReduce is flexible and works with several Hadoop languages to handle and store data.