**Assignment 3.1**

**List the Components of Hadoop 2.x and explain each component in detail.**

The Components of Hadoop 2.x majorly divided in to three they are,

* HDFS
* YARN
* MapReduce
* **HDFS**
* Hadoop Common Module is a Hadoop Base API (A Jar file) for all Hadoop Components. All other components works on top of this module.
* HDFS stands for Hadoop Distributed File System. It is also know as HDFS V2 as it is part of Hadoop 2.x with some enhanced features. It is used as a Distributed Storage System in Hadoop Architecture.

**Hadoop HDFS Daemons:**

**NameNode: (Single instance)**

• Runs in Master Node

• Manages the file metadata

• Client connects to NameNode for initiating all reads and writes in HDFS.

* **Data Node:** (Multiple instances/ one in each slave node)

• Runs in all slave nodes

• Data files are stored in the DataNodes in a distributed manner

• Reports to NameNode, periodically with lists of blocks they store

* **Secondary NameNode**:

• Acts as back up node to name node

• It periodically copies metadata into its own storage.

* **Block Replication**: HDFS stores multiple copies of each block in the cluster

• Block replication provides redundancy and fault tolerance

* **Replication Factor**: Number of copies of a block in cluster [Default - 3]
* **Rack Awareness:**
* Rack is a container featuring multiple servers ,stacked one above the other
* As it is vertical, it consumes less floor space as well as provides simple inter-connectivity
* HDFS is aware of the location of each server in the cluster. A cluster can comprise of multiple racks
* HDFS uses rack awareness to distribute the storage of blocks to achieve High Data Availability and Fault Tolerance
* **Metadata:**
* It refers to the location, size & storage information of each file
* It is maintained by the NameNode
* It stores the complete string: File to Blocks – Blocks to Server
* It is stored in two files in the NameNode: FSImage and EditLog

• FSImage stores all the file-names, locations and block structures.

• EditLog stores all file system modifications; for example a change in file location.

* **HDFS – Data Node Heartbeat And Block Report:**
* Data node sends heartbeats every 3 seconds via a TCP handshake
* Every 10th heartbeat is a block report
* Name node builds metadata from block report
* Communication happens through the port defined for Name node.
* **YARN**
* Hadoop 2.0 popularly known as YARN (Yet another Resource Negotiator) is the latest technology introduced in Oct 2013 that is being used widely nowadays for processing and managing distributed big data.
* Hadoop YARN is an advancement to Hadoop 1.0 released to provide performance enhancements which will benefit all the technologies connected with the Hadoop Ecosystem along with the Hive data warehouse and the Hadoop database (HBase). Hadoop YARN comes along with the Hadoop 2.x distributions that are shipped by Hadoop distributors. YARN performs job scheduling and resource management duties devoid of the users having to use Hadoop MapReduce on Hadoop Systems.
* Hadoop YARN has a modified architecture unlike the intrinsic characteristics of Hadoop 1.0 so that the systems can scale up to new levels and responsibilities can be clearly assigned to the various [components in Hadoop](https://www.dezyre.com/article/big-data-and-hadoop-training-hadoop-components-and-architecture/114) HDFS.
* YARN has taken an edge over the cluster management responsibilities from MapReduce, so that now MapReduce just takes care of the Data Processing and other responsibilities are taken care of by YARN.
* In Hadoop 2.0, the Job Tracker in YARN mainly depends on 3 important components:
  1. **Resource Manager Component:**
* This component is considered as the negotiator of all the resources in the cluster. Resource Manager is further categorized into an Application Manager that will manage all the user jobs with the cluster and a pluggable scheduler. This is a relentless YARN service that is designed for receiving and running the applications on the Hadoop Cluster. In Hadoop 2.0, a MapReduce job will be considered as an application.
  1. **Node Manager Component:**
* This is the job history server component of [YARN](https://www.dezyre.com/Big-Data-and-Hadoop/19)which will furnish the information about all the completed jobs. The NM keeps a track of all the users’ jobs and their workflow on any particular given node.

**3. Application Master Component** (aka User Job Life Cycle Manager):

* This is the component where the job actually resides and the Application Master component is responsible for managing each and every Map Reduce job and is concluded once the job completes processing.

### **Migration from Hadoop 1.0 to Hadoop 2.0**

With the advent of YARN framework as a part of the Hadoop 2.0 platform, there are several applications and tools available now for Hadoop programmers that will help them make the best out of big data which they never thought of.

YARN has been capable of providing the organizations something that is far beyond Map Reduce, by separating the cluster resource management function completely from the data processing function. With comparatively less overloaded sophisticated programming protocols and being cost effective, companies preferably would like to migrate their applications from Hadoop 1.0 to  Hadoop 2.0. An edge that YARN provides to Hadoop Users is that it is backward compatible (i.e. one can easily run an existing Map Reduce job on Hadoop 2.0 without making any modifications) thus compelling the companies to migrate from Hadoop 1.0 to Hadoop 2.0 without even giving it a second thought.

Despite the fact that most of the Hadoop applications have migrated from Hadoop 1.0 to Hadoop 2.0 there are migrations that are still in progress and companies are consistently striving hard to accomplish this long needed upgrade for their applications.

* **Map Reduce:**

Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

A MapReduce *job* usually splits the input data-set into independent chunks which are processed by the *map tasks* in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the *reduce tasks*. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

Typically the compute nodes and the storage nodes are the same, that is, the MapReduce framework and the Hadoop Distributed File System (see [HDFS Architecture Guide](https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html)) are running on the same set of nodes. This configuration allows the framework to effectively schedule tasks on the nodes where data is already present, resulting in very high aggregate bandwidth across the cluster.

The MapReduce framework consists of a single master ResourceManager, one slave NodeManager per cluster-node, and MRAppMaster per application (see [YARN Architecture Guide](https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html)).

Minimally, applications specify the input/output locations and supply *map* and *reduce* functions via implementations of appropriate interfaces and/or abstract-classes. These, and other job parameters, comprise the *job configuration*.

The Hadoop *job client* then submits the job (jar/executable etc.) and configuration to the ResourceManager which then assumes the responsibility of distributing the software/configuration to the slaves, scheduling tasks and monitoring them, providing status and diagnostic information to the job-client.

Although the Hadoop framework is implemented in Java™, MapReduce applications need not be written in Java.

* [Hadoop Streaming](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/streaming/package-summary.html) is a utility which allows users to create and run jobs with any executables (e.g. shell utilities) as the mapper and/or the reducer.
* [Hadoop Pipes](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapred/pipes/package-summary.html) is a [SWIG](http://www.swig.org/)-compatible C++ API to implement MapReduce applications (non JNI™ based).

## **MapReduce - User Interfaces**

This section provides a reasonable amount of detail on every user-facing aspect of the MapReduce framework. This should help users implement, configure and tune their jobs in a fine-grained manner. However, please note that the java doc for each class/interface remains the most comprehensive documentation available; this is only meant to be a tutorial.

Let us first take the Mapper and Reducer interfaces. Applications typically implement them to provide the map and reduce methods.

We will then discuss other core interfaces including Job, Partitioner, Input Format, Output Format, and others.

Finally, we will wrap up by discussing some useful features of the framework such as the Distributed Cache, Isolation Runner etc.

#### Mapper

[Mapper](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Mapper.html) maps input key/value pairs to a set of intermediate key/value pairs.

Maps are the individual tasks that transform input records into intermediate records. The transformed intermediate records do not need to be of the same type as the input records. A given input pair may map to zero or many output pairs.

The Hadoop MapReduce framework spawns one map task for each InputSplit generated by the InputFormat for the job.

Overall, mapper implementations are passed to the job via [Job.setMapperClass(Class)](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Job.html) method. The framework then calls [map(WritableComparable, Writable, Context)](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Mapper.html)for each key/value pair in the InputSplit for that task. Applications can then override the cleanup(Context) method to perform any required cleanup.

Output pairs do not need to be of the same types as input pairs. A given input pair may map to zero or many output pairs. Output pairs are collected with calls to context.write(WritableComparable, Writable).

Applications can use the Counter to report its statistics.

All intermediate values associated with a given output key are subsequently grouped by the framework, and passed to the Reducer(s) to determine the final output. Users can control the grouping by specifying a Comparator via [Job.setGroupingComparatorClass(Class)](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Job.html).

The Mapper outputs are sorted and then partitioned per Reducer. The total number of partitions is the same as the number of reduce tasks for the job. Users can control which keys (and hence records) go to which Reducer by implementing a custom Partitioner.

Users can optionally specify a combiner, via [Job.setCombinerClass(Class)](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Job.html), to perform local aggregation of the intermediate outputs, which helps to cut down the amount of data transferred from the Mapper to the Reducer.

The intermediate, sorted outputs are always stored in a simple (key-len, key, value-len, value) format. Applications can control if, and how, the intermediate outputs are to be compressed and the [CompressionCodec](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/io/compress/CompressionCodec.html) to be used via the Configuration.

##### How Many Maps?

The number of maps is usually driven by the total size of the inputs, that is, the total number of blocks of the input files.

The right level of parallelism for maps seems to be around 10-100 maps per-node, although it has been set up to 300 maps for very cpu-light map tasks. Task setup takes a while, so it is best if the maps take at least a minute to execute.

Thus, if you expect 10TB of input data and have a blocksize of 128MB, you’ll end up with 82,000 maps, unless Configuration.set(MRJobConfig.NUM\_MAPS, int) (which only provides a hint to the framework) is used to set it even higher.

#### Reducer

[Reducer](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Reducer.html) reduces a set of intermediate values which share a key to a smaller set of values.

The number of reduces for the job is set by the user via [Job.setNumReduceTasks(int)](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Job.html).

Overall, Reducer implementations are passed the Job for the job via the [Job.setReducerClass(Class)](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Job.html) method and can override it to initialize themselves. The framework then calls [reduce(WritableComparable, Iterable<Writable>, Context)](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Reducer.html) method for each <key, (list of values)> pair in the grouped inputs. Applications can then override the cleanup(Context) method to perform any required cleanup.

Reducer has 3 primary phases: shuffle, sort and reduce.

##### Shuffle

Input to the Reducer is the sorted output of the mappers. In this phase the framework fetches the relevant partition of the output of all the mappers, via HTTP.

##### Sort

The framework groups Reducer inputs by keys (since different mappers may have output the same key) in this stage.

The shuffle and sort phases occur simultaneously; while map-outputs are being fetched they are merged.

##### Secondary Sort

If equivalence rules for grouping the intermediate keys are required to be different from those for grouping keys before reduction, then one may specify a Comparatorvia [Job.setSortComparatorClass(Class)](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Job.html). Since [Job.setGroupingComparatorClass(Class)](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Job.html) can be used to control how intermediate keys are grouped, these can be used in conjunction to simulate *secondary sort on values*.

##### Reduce

In this phase the reduce(WritableComparable, Iterable<Writable>, Context) method is called for each <key, (list of values)> pair in the grouped inputs.

The output of the reduce task is typically written to the [FileSystem](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/fs/FileSystem.html) via Context.write(WritableComparable, Writable).

Applications can use the Counter to report its statistics.

The output of the Reducer is *not sorted*.

##### How Many Reduces?

The right number of reduces seems to be 0.95 or 1.75 multiplied by (<*no. of nodes*> \* <*no. of maximum containers per node*>).

With 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish. With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing.

Increasing the number of reduces increases the framework overhead, but increases load balancing and lowers the cost of failures.

The scaling factors above are slightly less than whole numbers to reserve a few reduce slots in the framework for speculative-tasks and failed tasks.

##### Reducer NONE

It is legal to set the number of reduce-tasks to *zero* if no reduction is desired.

In this case the outputs of the map-tasks go directly to the FileSystem, into the output path set by [FileOutputFormat.setOutputPath(Job, Path)](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/lib/output/FileOutputFormat.html). The framework does not sort the map-outputs before writing them out to the FileSystem.

#### Partitioner

[Partitioner](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Partitioner.html) partitions the key space.

Partitioner controls the partitioning of the keys of the intermediate map-outputs. The key (or a subset of the key) is used to derive the partition, typically by a *hash function*. The total number of partitions is the same as the number of reduce tasks for the job. Hence this controls which of the m reduce tasks the intermediate key (and hence the record) is sent to for reduction.

[HashPartitioner](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/lib/partition/HashPartitioner.html) is the default Partitioner.

#### Counter

[Counter](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Counter.html) is a facility for MapReduce applications to report its statistics.

Mapper and Reducer implementations can use the Counter to report statistics.

Hadoop MapReduce comes bundled with a [library](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/package-summary.html) of generally useful mappers, reducers, and partitioners.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*END\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*