Unit 2: Hadoop Ecosystem [6 Hrs.]

Hadoop – Daemons and Their Features

Daemons mean Process. Hadoop Daemons are a set of processes that run on Hadoop. Hadoop is a framework written in Java, so all these processes are Java Processes.
Apache Hadoop 2 consists of the following Daemons:

- NameNode
- DataNode
- Secondary Name Node
- Resource Manager
- Node Manager

Namenode, Secondary NameNode, and Resource Manager work on a Master System while the Node Manager and DataNode work on the Slave machine.

1. NameNode

NameNode works on the Master System. The primary purpose of Namenode is to manage all the MetaData. Metadata is the list of files stored in HDFS(Hadoop Distributed File System). As we know the data is stored in the form of blocks in a Hadoop cluster. So the DataNode on which or the location at which that block of the file is stored is mentioned in MetaData. All information regarding the logs of the transactions happening in a Hadoop cluster (when or who read/wrote the data) will be stored in MetaData. MetaData is stored in the memory.

2. DataNode

DataNode works on the Slave system. The NameNode always instructs DataNode for storing the Data. DataNode is a program that runs on the slave system that serves the read/write request from the client. As the data is stored in this DataNode, they should possess high memory to store more Data.

3. Secondary NameNode

Secondary NameNode is used for taking the hourly backup of the data. In case the Hadoop cluster fails, or crashes, the secondary Namenode will take the hourly backup or checkpoints of that data and store this data into a file name fsimage. This file then gets transferred to a new system. A new MetaData is assigned to that new system and a new Master is created with this MetaData, and the cluster is made to run again correctly.
This is the benefit of Secondary Name Node. Now in Hadoop2, we have High-Availability and Federation features that minimize the importance of this Secondary Name Node in Hadoop2.
Major Function Of Secondary NameNode:
It groups the Edit logs and Fsimage from NameNode together.
It continuously reads the MetaData from the RAM of NameNode and writes into the Hard Disk.
As secondary NameNode keeps track of checkpoints in a Hadoop Distributed File System, it is also known as the checkpoint Node.

4. Resource Manager

Resource Manager is also known as the Global Master Daemon that works on the Master System. The Resource Manager Manages the resources for the applications that are running in a Hadoop Cluster. The Resource Manager Mainly consists of 2 things.

A. ApplicationsManager
B. Scheduler

An Application Manager is responsible for accepting the request for a client and also makes a memory resource on the Slaves in a Hadoop cluster to host the Application Master. The scheduler is utilized for providing resources for applications in a Hadoop cluster and for monitoring this application.

5. Node Manager

The Node Manager works on the Slaves System that manages the memory resource within the Node and Memory Disk. Each Slave Node in a Hadoop cluster has a single NodeManager Daemon running in it. It also sends this monitoring information to the Resource Manager.

Hadoop Configuration modes

Hadoop offers three configuration modes:

1. Standalone Mode: This is the default mode in Hadoop. In this mode, Hadoop runs as a single Java process. It does not use HDFS; instead, it reads data directly from the local file system. It is primarily used for debugging and as a basic platform for development.

2. Pseudo-Distributed Mode: In this mode, each Hadoop daemon runs in a separate Java process. It simulates a cluster on a single machine. HDFS runs on the local file system, and each daemon operates as if it were running on a separate node. This mode is suitable for development and testing because it closely resembles a real Hadoop cluster without the need for multiple machines.

3. Fully-Distributed Mode: In this mode, Hadoop operates in a real distributed environment with multiple nodes forming a cluster. Each node runs one or more Hadoop daemons. HDFS distributes data across the cluster's nodes, and MapReduce jobs run across the cluster, utilizing its computational resources. This mode is used in production environments for large-scale data processing.

During development, the preferred configuration mode often depends on the scale of the project, available resources, and the specific requirements of the development process. However, for most development tasks, the Pseudo-Distributed Mode is often preferred, and here's why:

Pseudo-Distributed Mode for Development:

1. Resource Efficiency: Pseudo-Distributed Mode allows developers to simulate a multi-node Hadoop cluster on a single machine. This eliminates the need for provisioning multiple servers or VMs, making it more resource-efficient for development environments where resources might be limited.

2. Realistic Environment: It closely mirrors the behavior of a fully-distributed Hadoop cluster, allowing developers to test their applications in an environment that closely resembles the production setup. This helps catch issues early in the development cycle.

3. Ease of Setup: Setting up a Pseudo-Distributed Mode cluster is relatively straightforward compared to a fully-distributed cluster. It requires configuring a few files to simulate the

cluster environment on a single machine, making it convenient for developers to get started quickly.

4. Debugging: Since all components run on the same machine, it's easier to debug issues in the code or configuration. Developers can directly inspect logs and monitor resource usage without the complexity of distributed environments.

5. Learning: For beginners or those new to Hadoop, Pseudo-Distributed Mode provides a gentle introduction to the Hadoop ecosystem without the overhead of managing a full cluster. It allows developers to grasp the fundamentals of Hadoop development in a controlled environment.

In conclusion, while all three configuration modes have their merits, Pseudo-Distributed Mode is often preferred during development due to its resource efficiency, realistic environment simulation, ease of setup, debugging capabilities, and suitability for learning purposes.

Hadoop Cluster Setup:

Setting up a Hadoop cluster involves several key steps to ensure its proper functioning and efficiency. Initially, the hardware requirements must be addressed, with nodes designated as master and worker nodes equipped with adequate RAM, CPU, and storage. Subsequently, the software installation process entails downloading and configuring the Hadoop distribution across all nodes, ensuring consistency in versions to prevent compatibility issues. Configuration files such as core-site.xml and yarn-site.xml are then edited to define parameters like file system settings and memory allocation. Networking setup is crucial for enabling seamless communication between nodes, often involving static IP assignment or DNS configuration. Security measures like Kerberos authentication and encryption safeguard data integrity and access control. Testing and monitoring the cluster's performance are integral to verifying its functionality, with tools like Apache Ambari facilitating this process. As the data volume grows, scaling the cluster by adding more nodes ensures continued efficiency, requiring adjustments to configuration and data rebalancing. Finally, backup and disaster recovery strategies are implemented to mitigate risks and ensure uninterrupted operation in the face of failures or disasters.

Hadoop Streaming:

Hadoop Streaming is a versatile utility within the Hadoop ecosystem, enabling the execution of MapReduce jobs using executables or scripts written in languages beyond Java, such as Python or Perl. This functionality offers developers flexibility and ease of use, allowing them to leverage existing scripts or preferred languages for data processing tasks. The process involves specifying mapper and reducer executables or scripts when submitting MapReduce jobs, with Hadoop Streaming managing the execution by launching separate processes for each task. While this flexibility streamlines development, careful consideration must be given to performance considerations, as Hadoop Streaming may introduce overhead compared to native Java MapReduce programs due to process launching and data serialization. Nonetheless, its ability to accommodate a diverse range of programming languages and workflows makes it a valuable tool for developers seeking efficient and adaptable solutions within the Hadoop framework.

Unit 3: Distributed Storage and Processing of Big Data [12 Hrs.]

When a dataset outgrows the storage capacity of a single physical machine, it becomes necessary to partition it across a number of separate machines. Filesystems that manage the storage across a network of machines are called distributed file systems. Hadoop comes with a distributed file system called HDFS, which stands for Hadoop Distributed File System.

The Design of HDFS :

HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.

Very large files:
"Very large" in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data.

Streaming data access :
HDFS is built around the idea that the most efficient data processing pattern is a write-once, read many-times pattern. A dataset is typically generated or copied from source, then various analyses are performed on that dataset over time.

Commodity hardware :
Hadoop doesn't require expensive, highly reliable hardware to run on. It's designed to run on clusters of commodity hardware (commonly available hardware available from multiple vendors3) for which the chance of node failure across the cluster is high, at least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

These are areas where HDFS is not a good fit today:

Low-latency data access :
Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS.

Lots of small files : Since the namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode.

Multiple writers, arbitrary file modifications: Files in HDFS may be written to by a single writer. Writes are always made at the end of the file. There is no support for multiple writers, or for modifications at arbitrary offsets in the file.

HDFS Concepts

Blocks:

HDFS has the concept of a block, but it is a much larger unit—64 MB by default. Files in HDFS are broken into block-sized chunks, which are stored as independent units.

Having a block abstraction for a distributed file system brings several benefits.:

The first benefit : A file can be larger than any single disk in the network. There's nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster.

Second: Making the unit of abstraction a block rather than a file simplifies the storage subsystem. The storage subsystem deals with blocks, simplifying storage management (since blocks are a fixed size, it is easy to calculate how many can be stored on a given disk) and eliminating metadata concerns.

Third: Blocks fit well with replication for providing fault tolerance and availability. To insure against corrupted blocks and disk and machine failure, each block is replicated to a small number of physically separate machines (typically three).

Why Is a Block in HDFS So Large?

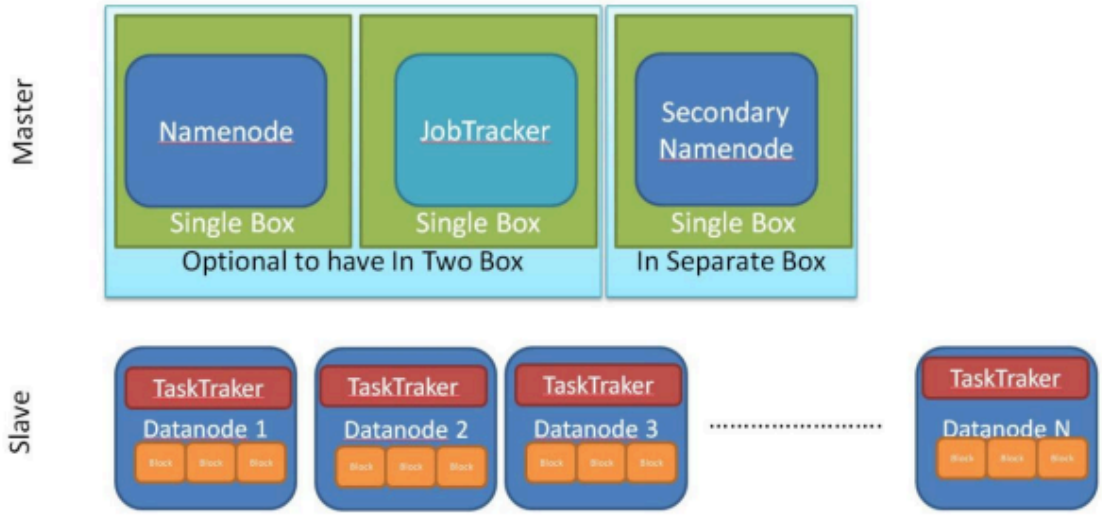
HDFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks. By making a block large enough, the time to transfer the data from the disk can be made to be significantly larger than the time to seek to the start of the block. Thus the time to transfer a large file made of multiple blocks operates at the disk transfer rate. A quick calculation shows that if the seek time is around 10 ms, and the transfer rate is 100 MB/s, then to make the seek time 1% of the transfer time, we need to make the block size around 100 MB. The default is actually 64 MB, although many HDFS installations use 128 MB blocks. This figure will continue to be revised upward as transfer speeds grow with new generations of disk drives.

Namenodes and Datanodes:

An HDFS cluster has two types of nodes operating in a master-worker pattern: a namenode (the master) and a number of datanodes (workers). The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located.

Apache Hadoop is designed to have Master Slave architecture:

Master: Namenode, JobTracker
Slave: {DataNode, TaskTracker}, ... {DataNode, TaskTracker}



HDFS is one primary component of Hadoop cluster and HDFS is designed to have Master-slave architecture.

- The Master (NameNode)
manages the file system namespace operations like opening, closing, and renaming files and directories and determines the mapping of blocks to DataNodes along with regulating access to files by clients -

-Slaves (DataNodes) are responsible for serving read and write requests from the file system's clients along with performing block creation, deletion, and replication upon instruction from the Master (NameNode).

Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing.

NameNode failure: if the machine running the namenode failed, all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes.

What precautions HDFS is taking to recover file system in case of namenode failure

The first way is to back up the files that make up the persistent state of the filesystem metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic. The usual configuration choice is to write to a local disk as well as a remote NFS mount.

Second way: It is also possible to run a secondary namenode, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. But this can be shaped to act as a primary namenode.

HDFS Federation :

The namenode keeps a reference to every file and block in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling . HDFS Federation, introduced in the 0.23 release series, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace. For example, one namenode might manage all the files rooted under /user, say, and a second namenode might handle files under /share. Each Namenode Namespace volumes are independent of each other, which means namenodes do not communicate with one another, and furthermore the failure of one namenode does not affect the availability of the namespaces managed by other namenodes. Block pool storage is not partitioned, however, so datanodes register with each namenode in the cluster and store blocks from multiple block pools.

HDFS High-Availability:

The namenode is still a single ***point of failure (SPOF),*** since if it did fail, all clients—including MapReduce jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping. In such an event the whole Hadoop system would effectively be out of service until a new namenode could be brought online.

To recover from a failed namenode in this situation, ***an administrator starts a new primary namenode with one of the filesystem metadata replicas, and configures datanodes and clients to use this new namenode.***

The new namenode is not able to serve requests until it has

i) loaded its namespace image into memory,
ii) replayed it's edit log, and
iii) received enough block reports from the datanodes to leave safe mode. On large clusters with many files and blocks, the time it takes for a namenode to start from cold can be 30 minutes or more.

The 0.23 release series of Hadoop remedies this situation by adding support for HDFS high availability (HA). In this implementation there is a pair of namenodes in an active standby configuration. In the event of the failure of the active namenode, the standby takes over its duties to continue servicing client requests without a significant interruption.

A few architectural changes are needed to allow this to happen:

- The namenodes must use highly-available shared storage to share the edit log.
- Datanodes must send block reports to both namenodes since the block mappings are stored in a namenode's memory, and not on disk.
- Clients must be configured to handle namenode failover, which uses a mechanism that is transparent to users.

Failover and fencing:

The transition from the active namenode to the standby is managed by a new entity in the system called the failover controller. Failover controllers are pluggable, but the first implementation uses ZooKeeper to ensure that only one namenode is active.

Failover may also be initiated manually by an administrator, in the case of routine maintenance, for example. This is known as a graceful failover, since the failover controller arranges an orderly transition for both namenodes to switch roles.

In the case of an ungraceful failover, The HA implementation goes to great lengths to ensure that the previously active namenode is prevented from doing any damage and causing corruption—a method known as fencing

HDFS basic commands

Here are some basic HDFS (Hadoop Distributed File System) commands along with their descriptions:

1. hadoop fs -ls /path:
- Description: Lists the contents of the specified directory in HDFS.
- Example: `hadoop fs -ls /user/myfolder`

2. hadoop fs -mkdir /path:
- Description: Creates a new directory in HDFS at the specified path.
- Example: `hadoop fs -mkdir /user/newfolder`

3. hadoop fs -put localfile /path:
- Description: Copies a file or directory from the local file system to HDFS.
- Example: `hadoop fs -put myfile.txt /user/myfolder`

4. hadoop fs -get /path localfile:
- Description: Copies a file or directory from HDFS to the local file system.
- Example: `hadoop fs -get /user/myfolder/myfile.txt ./`

5. hadoop fs -cat /path :
- Description: Displays the contents of a file in HDFS.
- Example: `hadoop fs -cat /user/myfolder/myfile.txt`

6. **hadoop fs -rm /path :**
   - Description: Deletes a file or directory from HDFS.
   - Example: `hadoop fs -rm /user/myfolder/myfile.txt`

7. **hadoop fs -mv /src /dest :**
   - Description: Moves a file or directory within HDFS from the source path to the destination path.
   - Example: `hadoop fs -mv /user/myfolder/myfile.txt /user/newfolder/`

8. **hadoop fs -cp /src /dest :**
   - Description: Copies a file or directory within HDFS from the source path to the destination path.
   - Example: `hadoop fs -cp /user/myfolder/myfile.txt /user/backup/`

9. **hadoop fs -du -s -h /path :**
   - Description: Displays the summary of disk usage for files and directories in HDFS at the specified path in a human-readable format.
   - Example: `hadoop fs -du -s -h /user/myfolder`

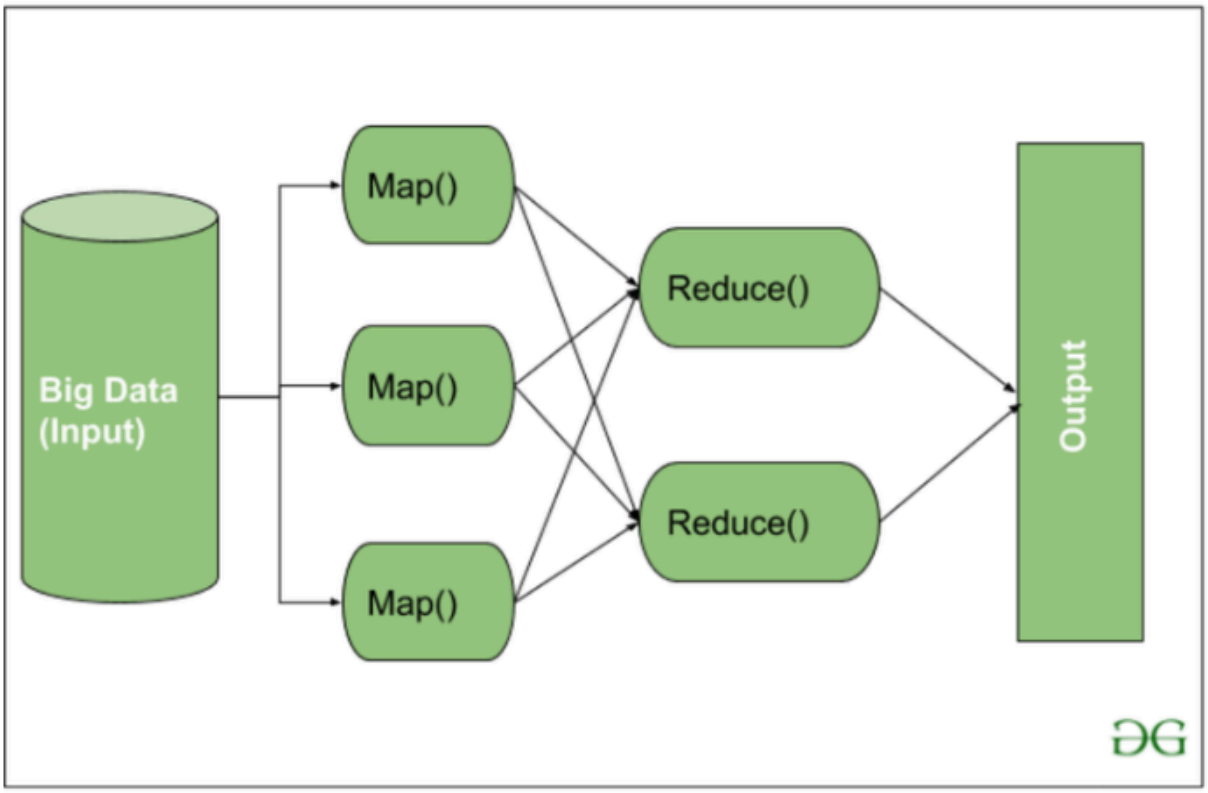
10. **hadoop fs -chmod permission /path :**
    - Description: Changes the permissions of a file or directory in HDFS.
    - Example: `hadoop fs -chmod 755 /user/myfolder/myfile.txt`

These commands provide basic functionalities for interacting with files and directories in HDFS, facilitating data management tasks within a Hadoop cluster.

Map Reduce

One of the three components of Hadoop is Map Reduce. The first component of Hadoop, that is, Hadoop Distributed File System (HDFS) , is responsible for storing the file. The second component that is, Map Reduce is responsible for processing the file.

MapReduce has mainly 2 tasks which are divided phase-wise.In first phase, Map is utilized and in next phase Reduce is utilized.



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.**

It used to be the case that the only way to access data stored in the Hadoop Distributed File System (HDFS) was using MapReduce. Other query-based methods are now utilized to obtain data from the HDFS using structured query language (SQL)-like commands, such as Hive and Pig. These, however, typically run alongside tasks created using the MapReduce approach.

This is so because MapReduce has unique benefits. To speed up processing, MapReduce executes logic (illustrated above) on the server where the data already sits, rather than transferring the data to the location of the application or logic.

MapReduce first appeared as a tool for Google to analyze its search results. However, it quickly grew in popularity thanks to its capacity to split and process terabytes of data in parallel, producing quicker results.

MapReduce is essential to the operation of the Hadoop framework and a core component. While “reduce tasks” shuffle and reduce the data, “map tasks” deal with separating and mapping the data. MapReduce makes concurrent processing easier by dividing petabytes of data into smaller chunks and processing them in parallel on Hadoop commodity servers. In the end, it collects all the information from several servers and gives the application a consolidated output.

For example, let us consider a Hadoop cluster consisting of 20,000 affordable commodity servers containing 256MB data blocks in each. It will be able to process around five terabytes worth of data simultaneously. Compared to the sequential processing of such a big data set, the usage of MapReduce cuts down the amount of time needed for processing.

To speed up the processing, MapReduce eliminates the need to transport data to the location where the application or logic is housed. Instead, it executes the logic directly on the server home to the data itself. Both the accessing of data and its storing are done using server disks. Further, the input data is typically saved in files that may include organized, semi-structured, or unstructured information. Finally, the output data is similarly saved in the form of files.

The main benefit of MapReduce is that users can scale data processing easily over several computing nodes. The data processing primitives used in the MapReduce model are mappers and reducers. Sometimes it is difficult to divide a data processing application into mappers and reducers. However, scaling an application to run over hundreds, thousands, or tens of thousands of servers in a cluster is just a configuration modification after it has been written in the MapReduce manner.

Functional vs Imperative programming language

Functional Programming	*Imperative Programming*
It is generally a process of developing software simply by composing pure functions, avoiding or minimizing side effects, shared data, and mutable data.	It is generally a process of describing steps that simply change the state of the computer.

It mainly focuses on what programs should be executed or operate i.e., results.	It mainly focuses on describing how the program executes or operates i.e., process.
It uses functions to perform everything.	It uses statements that change a program's state.
Its advantages include bugs-free code, increase performance, better encapsulation, increase reusability, increase testability, etc.	Its advantages include easy to learn, easy to read, a conceptual model is easy to learn, etc.
Its characteristics include low importance of the order of execution, stateless programming model, primary manipulations units, primary flow control, etc.	Its characteristics include a sequence of statements, contain state and can change state, might have some side effects, stateful programming model, etc.
These programs are generally structured as successive nested function calls.	These programs are structured as successive assignments of values to variable names.
In this, the command execution order is not fixed.	In this, the command execution order is fixed.

It involves writing programs in terms of functions and mathematical structures.	It involves writing programs as a series of instructions or statements that can actively modify memory.
It is more expressive and safer as compared to imperative programming.	It is less expressive and safer than functional programming.
It requires the explicit representation of data structures that are used.	It requires data structure to be represented as changes to state.
In this, new values are usually associated with the same name through command repetition.	In this, new values are usually associated with different names through recursive function call nesting.

Types of Input File Format

FileInputFormat: It is the base class for all file-based Input Formats. It specifies the input directory where data files are located. It will read all files and divide these files into one or more Input Splits.

TextInputFormat: Each line in the text file is a record.Key:Byte offset of line Value: Content of the line.

KeyValueTextInputFormat: Everything before the separator is the key, and everything after is value.

SequenceFileInputFormat: To read any sequence files. Key and values are user defined.

SequenceFileasTextInputFormat: Similar to SequenceFileInputFormat. It converts sequence file key values to text objects.

SequenceFileasBinaryInputFormat: To read any sequence files. It is used to extract sequence files keys and values as an opaque binary object.

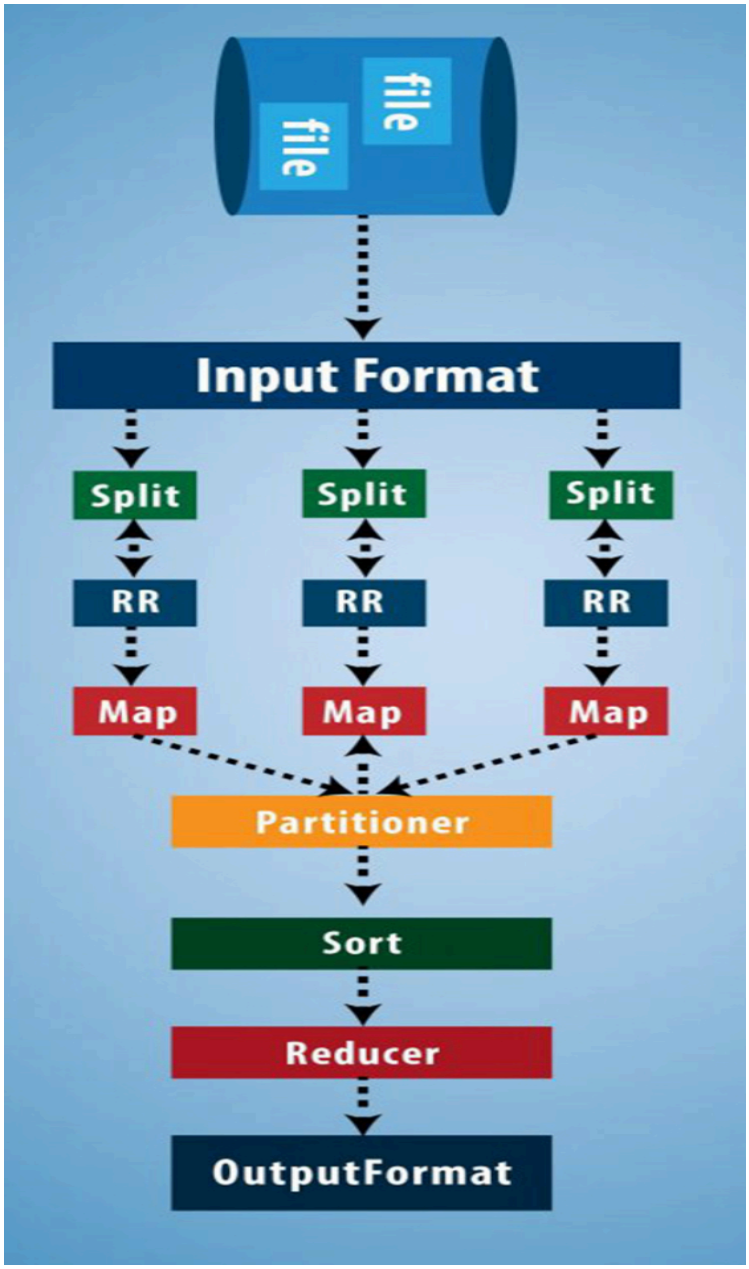
NLineInputFormat: Similar to TextInputFormat, But each split is guaranteed to have exactly N lines.

DBInputFormat: To read data from RDS. Key is LongWrittables and values are DB Writable.

Output Formats

- The OutputFormat checks the Output-Specification for execution of the Map- Reduce job. For e.g check that the output directory doesn't already exist.
- It determines how RecordWriter Implementation is used to write output to output files. Output Files are stored in a File System.

- The OutputFormat decides the way the output key-value pairs are written in the output files by RecordWriter.



TextOutputFormat : It is the MapReduce default Hadoop reducer Output Format which writes key,value pairs on individual lines of text files.

SequenceFileOutputFormat : It writes sequence files for its output and it is the intermediate format used between MapReduce jobs.

MapFileOutputFormat: It writes output as map files.The key in the MapFile must be added in order to ensure that the reducer emits keys in sorted order.

MultipleOutputs : It allows writing data to files whose names are derived from the output keys and values.

LazyOutputFormat : It is a wrapper OutputFormat which ensures that the output file will be created only when the record is emitted for a given partition

DBOutputFormat : It writes to the relational database and HBase and sends the reduced output to a SQL Table.