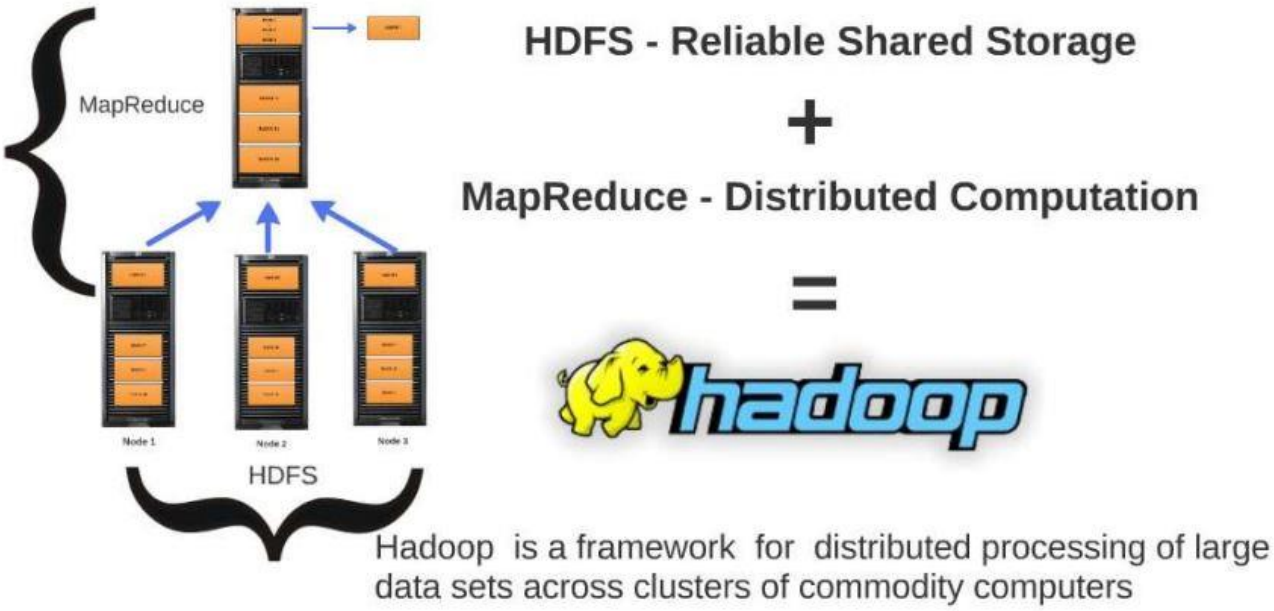
Working with Hadoop

Interacting with HDFS:

HDFS is a distributed file system for storing very large data files, running on clusters of commodity hardware. Unlike other distributed systems, HDFS is highly fault-tolerant and designed using low-cost hardware.

HDFS holds very large amount of data and provides easier access. To store such huge data, the files are stored across multiple machines. These files are stored in redundant fashion to rescue the system from possible data losses in case of failure. HDFS also makes applications available to parallel processing



Features of HDFS:

- It is suitable for the distributed storage and processing.
- Hadoop provides a command interface to interact with HDFS.
- The built-in servers of namenode and datanode help users to easily check the status of cluster.
- Streaming access to file system data.
- HDFS provides file permissions and authentication.

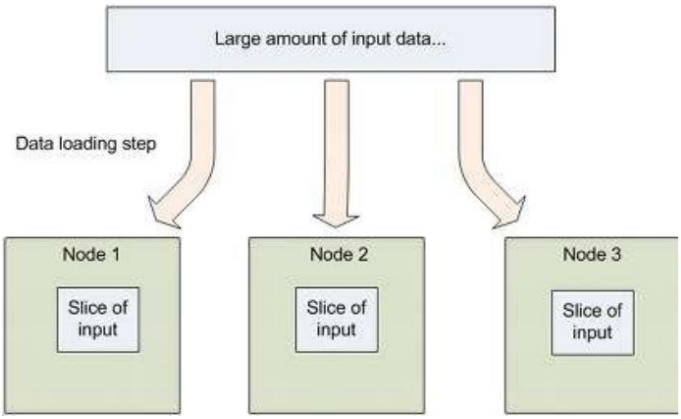
The following are some of the **salient features** that could be of interest to many users:

- Hadoop is written in Java and is supported on all major platforms.

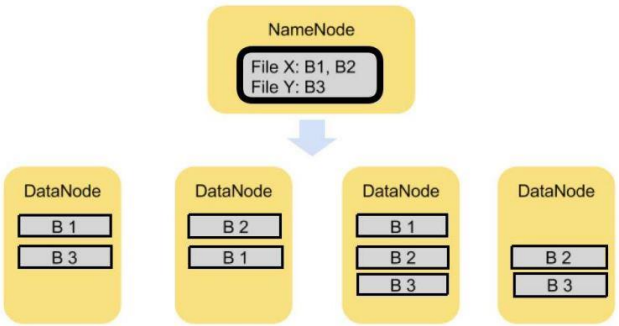
- Hadoop supports shell-like commands to interact with HDFS directly.
- The NameNode and Datanodes have built in web servers that makes it easy to check current status of the cluster.

Key points:

- Runs on Commodity hardware: Doesn't require expensive machines
- Large Files; Write-once, Read-many (WORM)
- Files are split into blocks



- Actual blocks go to DataNodes
- The metadata is stored at NameNode
- Replicate blocks to different node



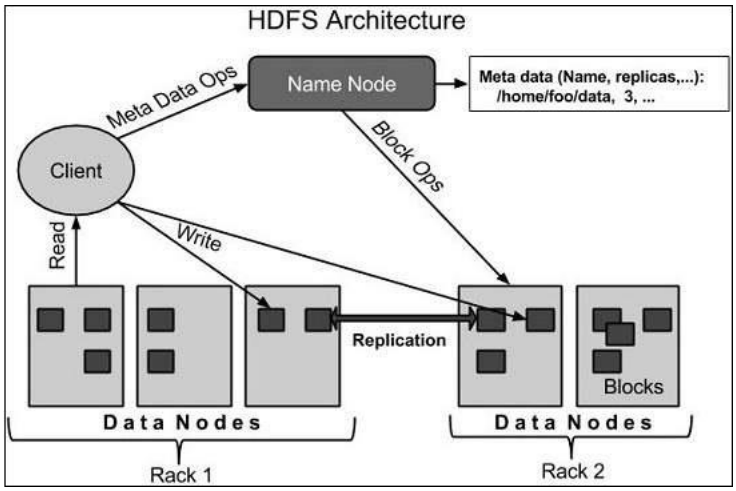
- Default configuration:

  Block size = 128MB

  Replication Factor = 3

HDFS Architecture

Given below is the architecture of a Hadoop File System.



HDFS follows the master-slave architecture and it has the following elements.

Namenode

The namenode is the commodity hardware that contains the GNU/Linux operating system and the namenode software. It is a software that can be run on commodity hardware. The system having the namenode acts as the master server and it does the following tasks −

- Manages the file system namespace.
- Regulates client's access to files.
- It also executes file system operations such as renaming, closing, and opening files and directories.

Datanode

The datanode is a commodity hardware having the GNU/Linux operating system and datanode software. For every node (Commodity hardware/System) in a cluster, there will be a datanode. These nodes manage the data storage of their system.

- Datanodes perform read-write operations on the file systems, as per client request.
- They also perform operations such as block creation, deletion, and replication according to the instructions of the namenode.

Block

Generally the user data is stored in the files of HDFS. The file in a file system will be divided into one or more segments and/or stored in individual data nodes. These file segments are called as blocks. In other words, the minimum amount of data that HDFS can read or write is called a Block. The default block size is 64MB, but it can be increased as per the need to change in HDFS configuration.

Goals of HDFS

Fault detection and recovery − Since HDFS includes a large number of commodity hardware, failure of components is frequent. Therefore HDFS should have mechanisms for quick and automatic fault detection and recovery.

Huge datasets − HDFS should have hundreds of nodes per cluster to manage the applications having huge datasets.

Hardware at data − A requested task can be done efficiently, when the computation takes place near the data. Especially where huge datasets are involved, it reduces the network traffic and increases the throughput.

Interacting With HDFS (On Command Prompt):

Command: hdfs

Usage: hdfs [--config confdir]

COMMAND Example:

- hdfs dfs
- hdfs dfsadmin
- hdfs fsck
- hdfs namenode

1. List the file/directory

Syntax: hdfs dfs -ls *[-d] [-h] [-R]* <hdfs-dir-path>

2. Creating a directory

Syntax: hdfs dfs -mkdir *[-p]* <hdfs-dir-path>

3. Create a file on local & put it on HDFS

Syntax: vi filename.txt

hdfs dfs -put *[options]* <local-file-path> <hdfs-dir-path>

4. Get a file from HDFS to local

Syntax: hdfs dfs -get <hdfs-file-path> [local-dir-path]

5. Copy From LOCAL To HDFS

Syntax: hdfs dfs –copyFromLocal <local-file-path> <hdfs-file-path>

6. Copy To LOCAL From HDFS

Syntax: hdfs dfs –copyToLocal <hdfs-file-path> <local-file-path>

7. Move a file from local to HDFS

Syntax: hdfs dfs –moveFromLocal <hdfs-file-path> <local-file-path>

8. Copy a file within HDFS

Syntax: hdfs dfs –cp <hdfs-source-file-path> <hdfs-dest-file-path>

9. Move a file within HDFS

Syntax: hdfs dfs –mv <hdfs-source-file-path> <hdfs-dest-file-path>

10. Merge files on HDFS

Syntax: hdfs dfs -getmerge *[-nl]* <hdfs-dir-path> <local-file-path>

11. View file contents

Syntax: hdfs dfs –cat <hdfs-file-path>

hdfs dfs -tail <hdfs-file-path>

hdfs dfs –text <hdfs-file-path>

12. Set replication factor

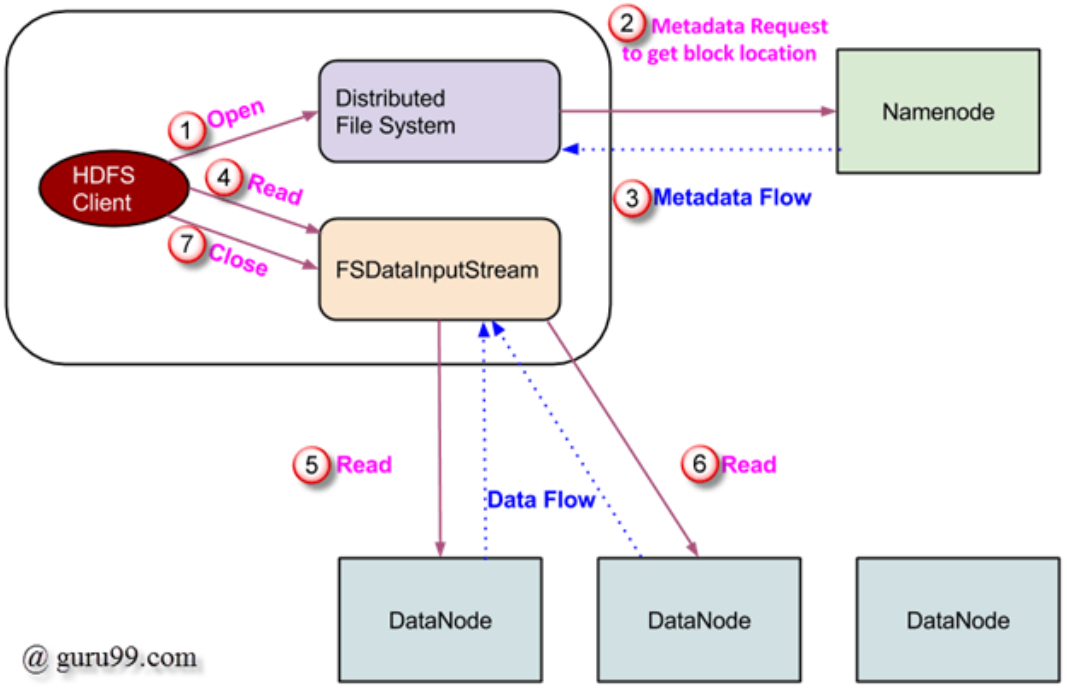
Syntax: hdfs dfs -setrep -w -R n <hdfs-file-path>

For more commads and examples, refer to this link https://www.slideshare.net/ApacheApex/hadoop-interacting-with-hdfs

Steps to read and write into HDFS

Read Operation In HDFS

Data read request is served by HDFS, NameNode, and DataNode. Let's call the reader as a 'client'. Below diagram depicts file read operation in Hadoop.



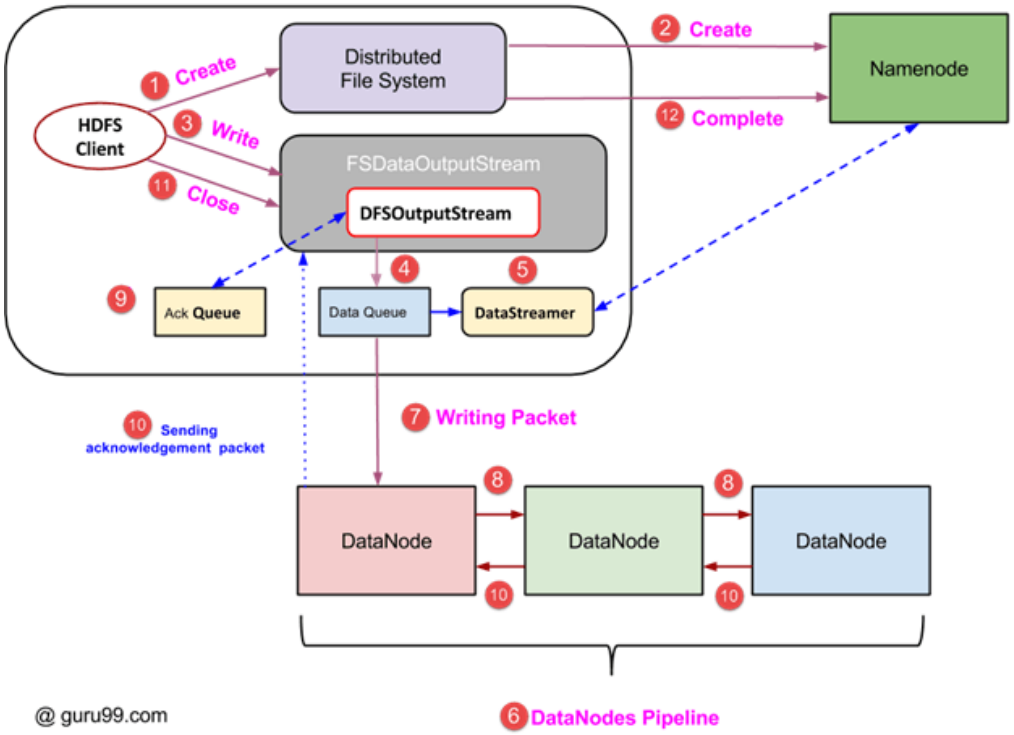
1. A client initiates read request by calling **'open()'** method of FileSystem object; it is an object of type **DistributedFileSystem**.
2. This object connects to namenode using RPC and gets metadata information such as the locations of the blocks of the file. Please note that these addresses are of first few blocks of a file.
3. In response to this metadata request, addresses of the DataNodes having a copy of that block is returned back.
4. Once addresses of DataNodes are received, an object of type **FSDataInputStream** is returned to the client. **FSDataInputStream** contains **DFSInputStream** which takes care of interactions with DataNode and NameNode. In step 4 shown in the above diagram, a client invokes **'read()'** method which causes **DFSInputStream** to establish a connection with the first DataNode with the first block of a file.
5. Data is read in the form of streams wherein client invokes **'read()'** method repeatedly. This process of **read()** operation continues till it reaches the end of block.

6. Once the end of a block is reached, DFSInputStream closes the connection and moves on to locate the next DataNode for the next block
7. Once a client has done with the reading, it calls **a close()** method.

For more detail Read operation, refer to this link : https://youtu.be/2E-rjuKVEh8

Write Operation In HDFS

In this section, we will understand how data is written into HDFS through files.



1. A client initiates write operation by calling 'create()' method of DistributedFileSystem object which creates a new file - Step no. 1 in the above diagram.
2. DistributedFileSystem object connects to the NameNode using RPC call and initiates new file creation. However, this file creates operation does not associate any blocks with the file. It is the responsibility of NameNode to verify that the file (which is being created) does not exist already and a client has correct permissions to create a new file. Once a new record in NameNode is created, an object of type

FSDataOutputStream is returned to the client. A client uses it to write data into the HDFS. Data write method is invoked (step 3 in the diagram).

3. FSDataOutputStream contains DFSOutputStream object which looks after communication with DataNodes and NameNode. While the client continues writing data, **DFSOutputStream** continues creating packets with this data. These packets are enqueued into a queue which is called as **DataQueue**.
4. There is one more component called **DataStreamer** which consumes this **DataQueue**. DataStreamer also asks NameNode for allocation of new blocks thereby picking desirable DataNodes to be used for replication.
5. Now, the process of replication starts by creating a pipeline using DataNodes. In our case, we have chosen a replication level of 3 and hence there are 3 DataNodes in the pipeline.
6. The DataStreamer pours packets into the first DataNode in the pipeline.
7. Every DataNode in a pipeline stores packet received by it and forwards the same to the second DataNode in a pipeline.
8. Another queue, 'Ack Queue' is maintained by DFSOutputStream to store packets which are waiting for acknowledgment from DataNodes.
9. Once acknowledgment for a packet in the queue is received from all DataNodes in the pipeline, it is removed from the 'Ack Queue'. In the event of any DataNode failure, packets from this queue are used to reinitiate the operation.
10. After a client is done with the writing data, it calls a close() method (Step 9 in the diagram) Call to close(), results into flushing remaining data packets to the pipeline followed by waiting for acknowledgment.
11. Once a final acknowledgment is received, NameNode is contacted to tell it that the file write operation is complete.

For detailed Write operation, refer this link : https://youtu.be/hwCh4OMjLK0

Anatomy of MapReduce Program

How MapReduce Works? Complete Process.

The whole process goes through four phases of execution namely, splitting, mapping, shuffling, and reducing.

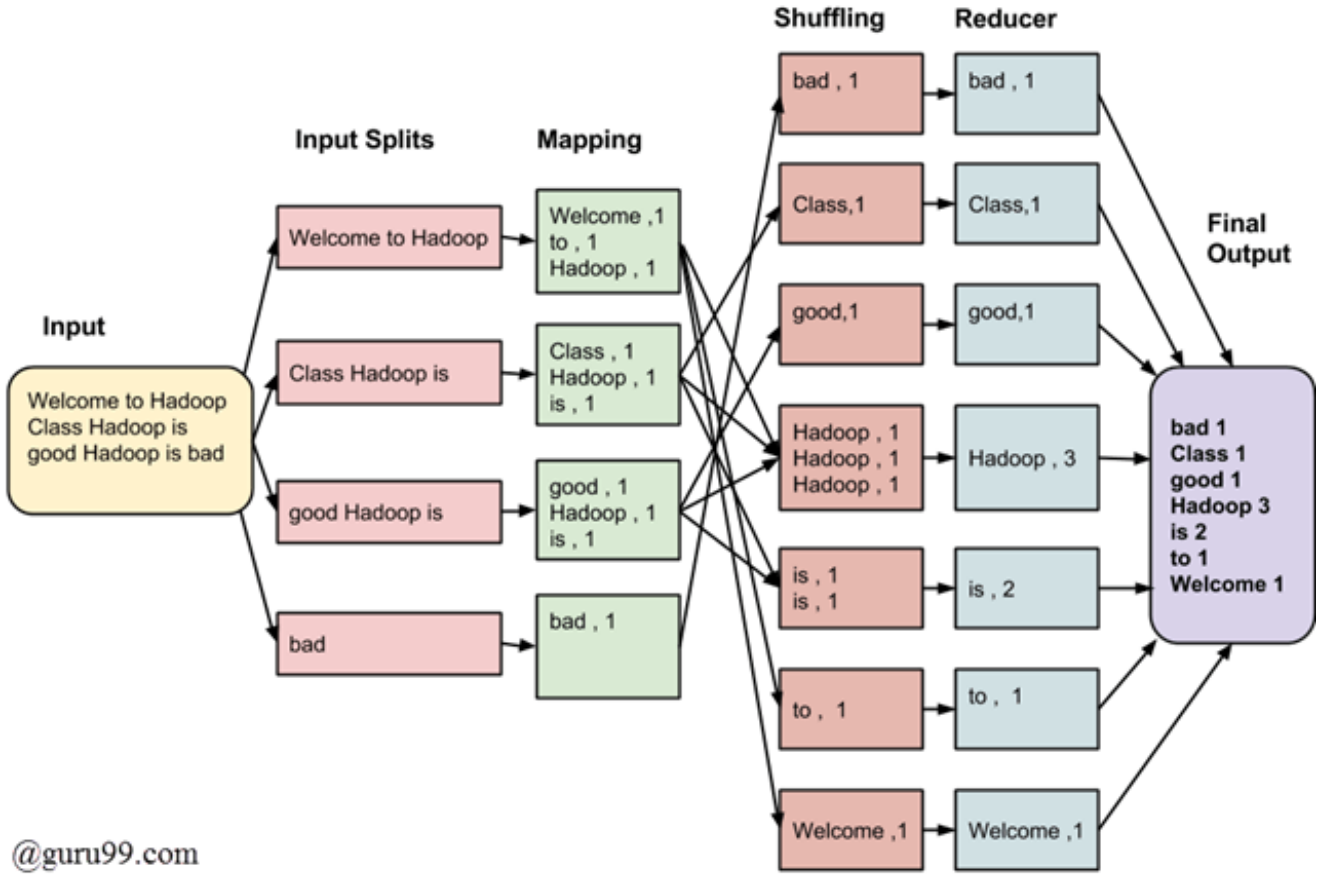
Let's understand this with an example –

Consider you have following input data for your Map Reduce Program:

```
Welcome to Hadoop Class

Hadoop is good

Hadoop is bad
```



MapReduce Architecture

The final output of the MapReduce task is

Bad	1
Class	1
good	1
Hadoop	3
is	2
to	1
Welcome	1

The data goes through the following phases

Input Splits:

An input to a MapReduce job is divided into fixed-size pieces called **input splits** Input split is a chunk of the input that is consumed by a single map

Mapping

This is the very first phase in the execution of map-reduce program. In this phase data in each split is passed to a mapping function to produce output values. In our example, a job of mapping phase is to count a number of occurrences of each word from input splits (more details about input-split is given below) and prepare a list in the form of <word, frequency>

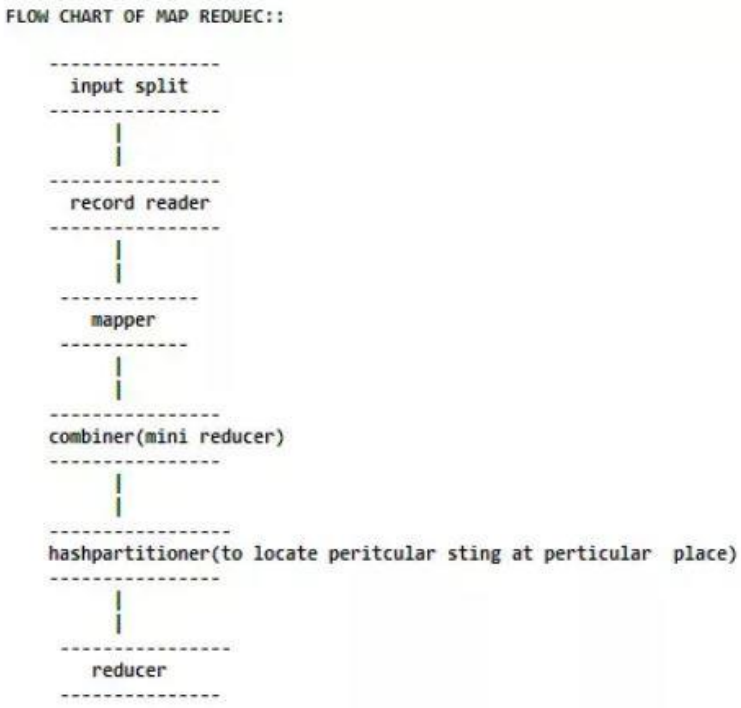
Shuffling

This phase consumes the output of Mapping phase. Its task is to consolidate the relevant records from Mapping phase output. In our example, the same words are clubed together along with their respective frequency.

Reducing

In this phase, output values from the Shuffling phase are aggregated. This phase combines values from Shuffling phase and returns a single output value. In short, this phase summarizes the complete dataset.

In our example, this phase aggregates the values from Shuffling phase i.e., calculates total occurrences of each word.



How MapReduce Organizes Work?

Hadoop divides the job into tasks. There are two types of tasks:

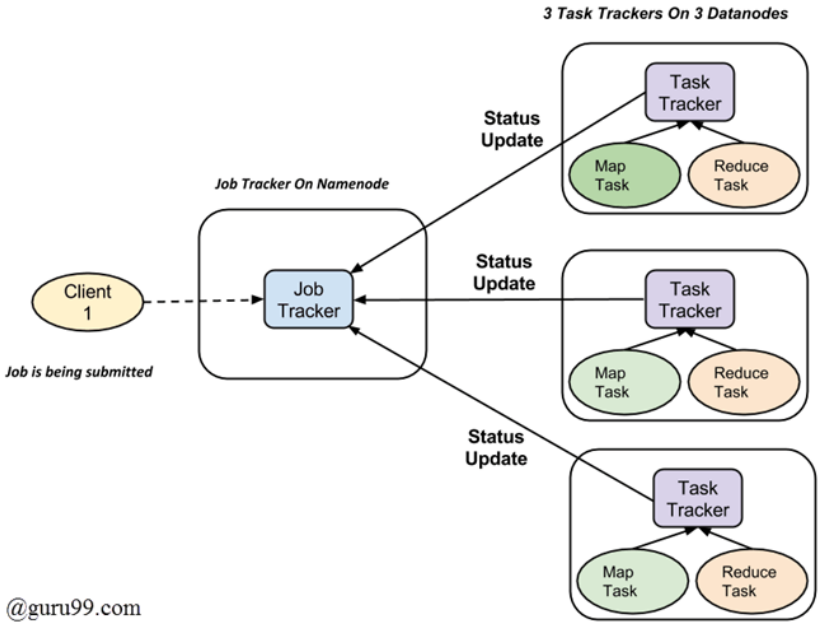
1. **Map tasks** (Splits & Mapping)
2. **Reduce tasks** (Shuffling, Reducing)

as mentioned above.

The complete execution process (execution of Map and Reduce tasks, both) is controlled by two types of entities called a

1. **Jobtracker**: Acts like a **master** (responsible for complete execution of submitted job)
2. **Multiple Task Trackers**: Acts like **slaves,** each of them performing the job

For every job submitted for execution in the system, there is one **Jobtracker** that resides on **Namenode** and there are **multiple tasktrackers** which reside on **Datanode**.



- A job is divided into multiple tasks which are then run onto multiple data nodes in a cluster.
- It is the responsibility of job tracker to coordinate the activity by scheduling tasks to run on different data nodes.
- Execution of individual task is then to look after by task tracker, which resides on every data node executing part of the job.
- Task tracker's responsibility is to send the progress report to the job tracker.
- In addition, task tracker periodically sends '**heartbeat**' signal to the Jobtracker so as to notify him of the current state of the system.
- Thus job tracker keeps track of the overall progress of each job. In the event of task failure, the job tracker can reschedule it on a different task tracker.

Hadoop data types:

Below is the list of few data types in Java along with the equivalent Hadoop variant:

1. **Integer** –> **IntWritable**: It is the Hadoop variant of *Integer*. It is used to pass integer numbers as key or value.
2. **Float** –> **FloatWritable**: Hadoop variant of *Float* used to pass floating point numbers as key or value.
3. **Long** –> **LongWritable**: Hadoop variant of *Long* data type to store long values.
4. **Short** –> **ShortWritable**: Hadoop variant of *Short* data type to store short values.
5. **Double** –> **DoubleWritable**: Hadoop variant of *Double* to store double values.
6. **String** –> **Text**: Hadoop variant of *String to pass string characters as key or value.*
7. **Byte** –> **ByteWritable**: Hadoop variant of *byte* to store sequence of bytes.
8. **null** –> **NullWritable**: Hadoop variant of *null* to pass null as a key or value.

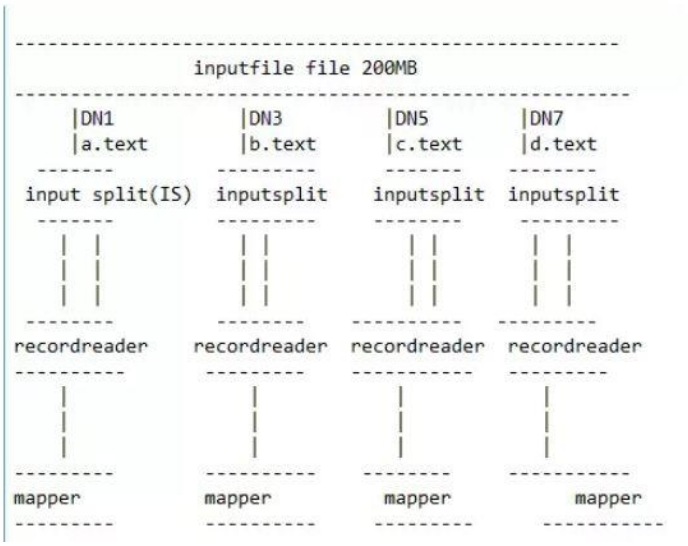
   Usually *NullWritable* is used as data type for output key of the reducer, when the output key is not important in the final result

Mapper:

Mapper is a first phase to solve your problem. Most of the time programmer writes 60% to 70% of the logic on Mapper phase only. In this phase all the computation, processing and distribution of data takes place. So this is very very important on the point of designing strategy to deal with the problem. In this phase programmer writes its logic that will deal with the data. Mapper phase works on concept of parallel processing (Hadoop uses divide and conquer approach to solve the problem) for fast execution.

If I talk more logically then, Mapper is not a first step to solve your problem, Input Split is first step but programmers don't write logic/code for this it is already defined on Hadoop framework. Then

Record Reader comes, that takes input from Input Split and break your problem to more narrow and pass it to Mapper (so ideally Mapper is the third phase of Hadoop framework)



Key points

- Mapper 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 Map/Reduce framework spawns one map task for each InputSplit generated by the InputFormat for the job.

Reducer

There are two sub parts that internally works before your code give its final result, that are shuffle and sort. Shuffle just to collect similar type of works into single unit and Sort for sorting data into some order (generally these two phase are already defined you just need to write logic for Reducer phase only)

In Hadoop**,** Reducer takes the output of the Mapper **(**intermediate key-value pair) process each of them to generate the output. The output of the reducer is the final output, which is stored in HDFS. Usually, in the Hadoop Reducer, we do aggregation or summation sort of computation.

Key Points:

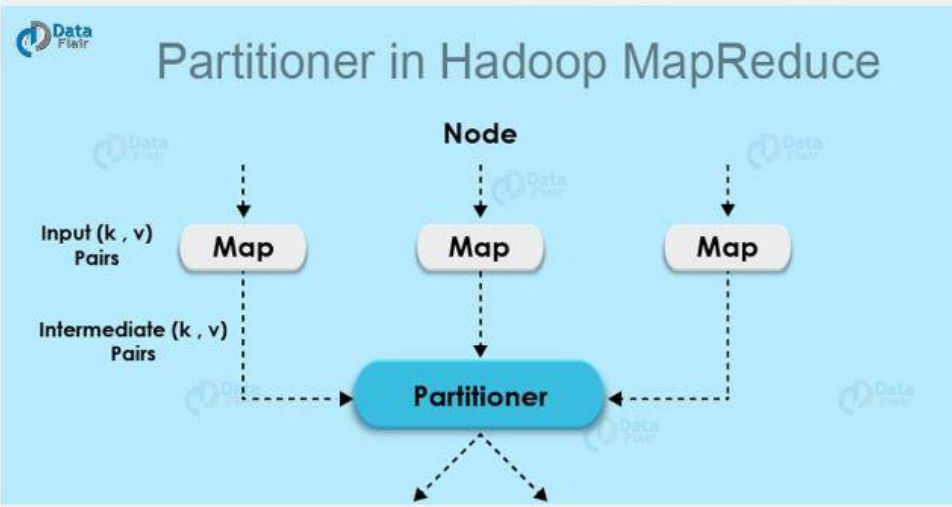
- Reducer reduces a set of intermediate values which share a key to a smaller set of values.
- The number of reduces for the job is set by the user via JobConf.setNumReduceTasks(int).
- Overall, Reducer implementations are passed the JobConf for the job via the JobConfigurable.configure(JobConf) method and can override it to initialize themselves. The

framework then calls reduce(WritableComparable, Iterator, OutputCollector, Reporter) method for each <key, (list of values)> pair in the grouped inputs. Applications can then override the Closeable.close() method to perform any required cleanup.

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

Partitioner

The Partitioner in MapReduce controls the partitioning of the key of the intermediate mapper output. By hash function, key (or a subset of the key) is used to derive the partition. A total number of partitions depends on the number of reduce task.



Hadoop Partitioner – Internals of MapReduce Partitioner

Partitioning of the keys of the intermediate map output is controlled by the Partitioner. By hash function, key (or a subset of the key) is used to derive the partition. According to the **key-value** each mapper output is partitioned and records having the same key value go into the same partition (within each mapper), and then each partition is sent to a reducer. Partition class determines which partition a given (key, value) pair will go. Partition phase takes place after map phase and before reduce phase.

In conclusion, Hadoop Partitioner allows even distribution of the map output over the reducer. In Partitioner, partitioning of map output take place on the basis of the key and sorted.

Combiner

Hadoop Combiner **is also known as "**Mini-Reducer**" that summarizes the Mapper output record with the same Key before passing to the Reducer.**

Hadoop Combiner – Best Explanation to MapReduce Combiner

On a large dataset when we run MapReduce job, large chunks of intermediate data is generated by the Mapper and this intermediate data is passed on the Reducer for further processing, which leads to enormous network congestion. MapReduce framework provides a function known as Hadoop Combiner that plays a key role in reducing network congestion.

We will discuss some **advantages** of Mapreduce Combiner.

- Hadoop Combiner reduces the time taken for data transfer between mapper and reducer.
- It decreases the amount of data that needed to be processed by the reducer.
- The Combiner improves the overall performance of the reducer.

There are also some **disadvantages** of hadoop Combiner. Let's discuss them one by one-

- MapReduce jobs cannot depend on the Hadoop combiner execution because there is no guarantee in its execution.
- In the local filesystem, the key-value pairs are stored in the Hadoop and run the combiner later which will cause expensive disk IO.

In conclusion, we can say that MapReduce Combiner plays a key role in reducing network congestion. MapReduce combiner improves the overall performance of the reducer by summarizing the output of Mapper.

Reading and Writing Custom-Formatted HDFS Data

Use MapReduce and the CREATE EXTERNAL TABLE command to read and write data with custom formats on HDFS.

To **read** custom-formatted data:

1. Author and run a MapReduce job that creates a copy of the data in a format accessible to Greenplum Database.
2. Use CREATE EXTERNAL TABLE to read the data into Greenplum Database.

To **write** custom-formatted data:

1. Write the data.
2. Author and run a MapReduce program to convert the data to the custom format and place it on the Hadoop Distributed File System.

Word count with predefined mapper and reducer

In Hadoop, MapReduce is a computation that decomposes large manipulation jobs into individual tasks that can be executed in parallel across a cluster of servers. The results of tasks can be joined together to compute final results.

MapReduce consists of 2 steps:

- **Map Function –** It takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (Key-Value pair).

  Example – (Map function in Word Count)

Input	Set of data	Bus, Car, bus, car, train, car, bus, car, train, bus, TRAIN,BUS, buS, caR, CAR, car, BUS, TRAIN
Output	Convert into another set of data	(Bus,1), (Car,1), (bus,1), (car,1), (train,1),

	(Key,Value)	(car,1), (bus,1), (car,1), (train,1), (bus,1), (TRAIN,1),(BUS,1), (buS,1), (caR,1), (CAR,1), (car,1), (BUS,1), (TRAIN,1)

- **Reduce Function –** Takes the output from Map as an input and combines those data tuples into a smaller set of tuples.

  Example – (Reduce function in Word Count)

Input **(output of Map function)**	Set of Tuples	(Bus,1), (Car,1), (bus,1), (car,1), (train,1), (car,1), (bus,1), (car,1), (train,1), (bus,1), (TRAIN,1),(BUS,1), (buS,1), (caR,1), (CAR,1), (car,1), (BUS,1), (TRAIN,1)
Output	Converts into smaller set of tuples	(BUS,7), (CAR,7), (TRAIN,4)

Work Flow of the Program

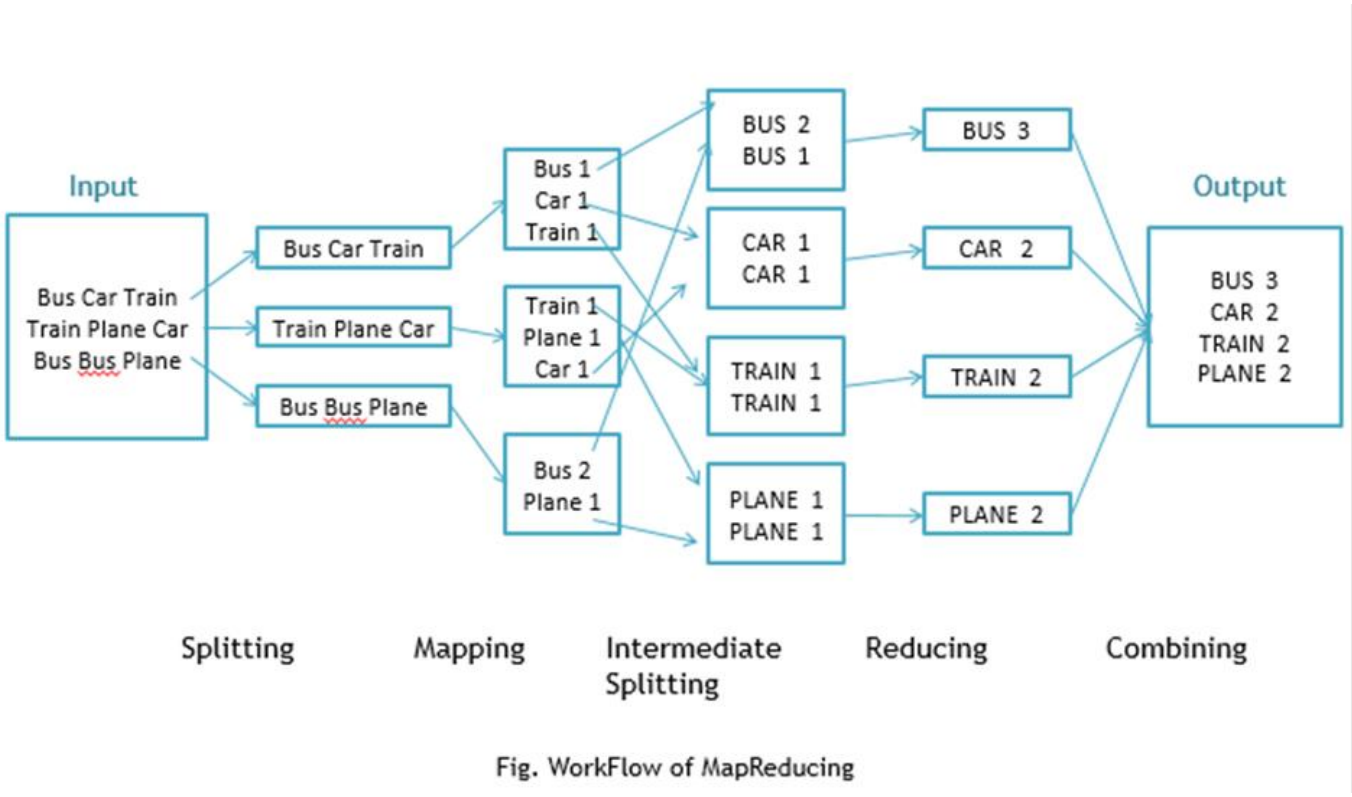


Fig. WorkFlow of MapReducing

Workflow of MapReduce consists of 5 steps:

1. **Splitting** – The splitting parameter can be anything, e.g. splitting by space, comma, semicolon, or even by a new line ('\n').
2. **Mapping** – as explained above.
3. **Intermediate splitting** – the entire process in parallel on different clusters. In order to group them in "Reduce Phase" the similar KEY data should be on the same cluster.
4. **Reduce** – it is nothing but mostly group by phase.
5. **Combining** – The last phase where all the data (individual result set from each cluster) is combined together to form a result.

Introduction to Hive

The Hadoop ecosystem contains different sub-projects (tools) such as Sqoop, Pig, and Hive that are used to help Hadoop modules.

- **Sqoop:** It is used to import and export data to and from between HDFS and RDBMS.
- **Pig:** It is a procedural language platform used to develop a script for MapReduce operations.
- **Hive:** It is a platform used to develop SQL type scripts to do MapReduce operations.

What is Hive

Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.

Initially Hive was developed by Facebook, later the Apache Software Foundation took it up and developed it further as an open source under the name Apache Hive. It is used by different companies. For example, Amazon uses it in Amazon Elastic MapReduce.

Hive is not

- A relational database
- A design for OnLine Transaction Processing (OLTP)
- A language for real-time queries and row-level updates

Features of Hive

- It stores schema in a database and processed data into HDFS.
- It is designed for OLAP.
- It provides SQL type language for querying called HiveQL or HQL.
- It is familiar, fast, scalable, and extensible.

Some key differences between Hive and relational databases are the following;

- Relational databases are of "**Schema on READ and Schema on Write**". First creating a table then inserting data into the particular table. On relational database tables, functions like Insertions, Updates, and Modifications can be performed.
- Hive is "**Schema on READ only**". So, functions like the update, modifications, etc. don't work with this. Because the Hive query in a typical cluster runs on multiple Data Nodes. So it is not possible to update and modify data across multiple nodes.(Hive versions below 0.13)
- Also, Hive supports "**READ Many WRITE Once**" pattern. Which means that after inserting table we can update the table in the latest Hive versions.

Introduction to Spark

Industries are using Hadoop extensively to analyze their data sets. The reason is that Hadoop framework is based on a simple programming model (MapReduce) and it enables a computing solution that is scalable, flexible, fault-tolerant and cost effective. Here, the main concern is to maintain speed in processing large datasets in terms of waiting time between queries and waiting time to run the program.

Spark was introduced by Apache Software Foundation for speeding up the Hadoop computational computing software process.

As against a common belief, **Spark is not a modified version of Hadoop** and is not, really, dependent on Hadoop because it has its own cluster management. Hadoop is just one of the ways to implement Spark.

Spark uses Hadoop in two ways – one is **storage** and second is **processing**. Since Spark has its own cluster management computation, it uses Hadoop for storage purpose only.

What is Apache Spark?

Apache Spark is a lightning-fast cluster computing technology, designed for fast computation. It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing. The main feature of Spark is its **in-memory cluster computing** that increases the processing speed of an application.

Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming. Apart from supporting all these workload in a respective system, it reduces the management burden of maintaining separate tools.

Evolution of Apache Spark

Spark is one of Hadoop's sub project developed in 2009 in UC Berkeley's AMPLab by Matei Zaharia. It was Open Sourced in 2010 under a BSD license. It was donated to Apache software foundation in 2013, and now Apache Spark has become a top level Apache project from Feb-2014.

Features of Apache Spark

- **Speed** − Spark helps to run an application in Hadoop cluster, up to 100 times faster in memory, and 10 times faster when running on disk. This is possible by reducing number of read/write operations to disk. It stores the intermediate processing data in memory.
- **Supports multiple languages** − Spark provides built-in APIs in Java, Scala, or Python. Therefore, you can write applications in different languages. Spark comes up with 80 high-level operators for interactive querying.
- **Advanced Analytics** − Spark not only supports 'Map' and 'reduce'. It also supports SQL queries, Streaming data, Machine learning (ML), and Graph algorithms.