9,176 Tweets per second
1,023 Instagram images uploaded per second
5,036 Skype calls per second
86,497 Google searches per second
86,302 YouTube videos watched per second
2,957,983 Emails sent per second
and much more…

Big Data

extremely large data sets that may be analyzed computationally to reveal patterns, trends, and associations, especially relating to human behavior and interactions.

Or

Big data is a term that describes the large volume of data – both structured and unstructured – that inundates a business on a day-to-day basis. But it's not the amount of data that's important. It's what organizations do with the data that matters. Big data can be analyzed for insights that lead to better decisions and strategic business moves.

Or

The term "big data" refers to data that is so large, fast or complex that it's difficult or impossible to process using traditional methods. The act of accessing and storing large amounts of information for analytics has been around a long time. But the concept of big data gained momentum in the early 2000s when industry analyst Doug Laney articulated the now-mainstream definition of big data as the three V's:

Volume: Organizations collect data from a variety of sources, including business transactions, smart (IoT) devices, industrial equipment, videos, social media and more. In the past, storing it would have been a problem – but cheaper storage on platforms like data lakes and Hadoop have eased the burden.

Velocity: With the growth in the Internet of Things, data streams in to businesses at an unprecedented speed and must be handled in a timely manner. RFID tags, sensors and smart meters are driving the need to deal with these torrents of data in near-real time.

Variety: Data comes in all types of formats – from structured, numeric data in traditional databases to unstructured text documents, emails, videos, audios, stock ticker data and financial transactions.

What Is Structured Data?

Structured data is information that has been formatted and transformed into a well-defined data model. The raw data is mapped into predesigned fields that can then later be extracted and read through SQL easily. SQL relational databases, consisting of tables with rows and columns, are the perfect example of structured data.

The relational model of structured data utilizes memory since it minimizes data redundancy. However, this also means that structured data is more inter-dependent and less flexible.

Examples of Structured Data

Structured data is generated by both humans and machines. There are numerous examples of structured data that is generated by machines, such as POS data like quantity, barcodes, and weblog statistics. Similarly, anyone who works on data would have used spreadsheets once in their lifetime, which is a classic case of structured data generated by humans. Due to the organization of structured data, it is easier to analyze than both semi-structured and unstructured data.

Semi-Structured Data

Your data may not always be structured or unstructured – semi-structured data is what lies another category between these two that is partially structured. Such data is defined as semi-structured. Semi-structured data is a type of data that has some consistent and definite characteristics, it does not confine into a rigid structure such as that needed for relational databases. Organizational properties like metadata or semantics tags are used with semi-structured data to make it more manageable, however, it still contains some variability and inconsistency.

Example

An example of semi-structured data is delimited files. It contains elements that can break down the data into separate hierarchies. Similarly, in digital photographs, the image does not have a pre-defined structure itself. Still, if it is taken from a smartphone, it would have structured attributes

like geotag, device ID, and DateTime stamp. After being stored, images can also be assigned tags such as ‘pet’ or ‘dog’ to provide a structure.

On some occasions, unstructured data is classified as semi-structured because it has one or more classifying attributes.

What is Unstructured Data?

Data present in absolute raw form is termed as unstructured. This data is difficult to process due to its complex arrangement and formatting. Unstructured data management may take data from many forms, including social media posts, chats, satellite imagery, IoT sensor data, emails, and presentations in order to organize it in a logical, predefined manner. In contrast, the meaning of structured data is data that follows pre-defined data models and is easy to analyze. Structured data examples would include alphabetically arranged names of customers and properly organized credit card numbers.

Unstructured data can be anything that’s not in any specific format. This can be a paragraph from a book with relevant information. An example of unstructured data could be Log files that are not easy to separate. Social media comments and posts that need to be analyzed.

Characteristics of Big Data

1**. Volume**

Volume refers to the huge amounts of data that is collected and generated every second in large organizations. This data is generated from different sources such as IoT devices, social media, videos, financial transactions, and customer logs.

Storing and processing this huge amount of data was a problem earlier. But now distributed systems such as Hadoop are used for organizing data collected from all these sources. The size of the data is crucial for understanding its value. Also, the volume is useful in determining whether a collection of data is Big Data or not.

Data volume can vary. For example, a text file is a few kilobytes whereas a video file is a few megabytes.

2. Variety

Another one of the most important Big Data characteristics is its variety. It refers to the different sources of data and their nature. The sources of data have changed over the years. Earlier, it was only available in spreadsheets and databases. Nowadays, data is present in photos, audio files, videos, text files, and PDFs.

The variety of data is crucial for its storage and analysis.

3. Velocity

This term refers to the speed at which the data is created or generated. This speed of data producing is also related to how fast this data is going to be processed. This is because only after analysis and processing, the data can meet the demands of the clients/users.

Massive amounts of data are produced from sensors, social media sites, and application logs – and all of it is continuous. If the data flow is not continuous, there is no point in investing time or effort on it.

4. Value

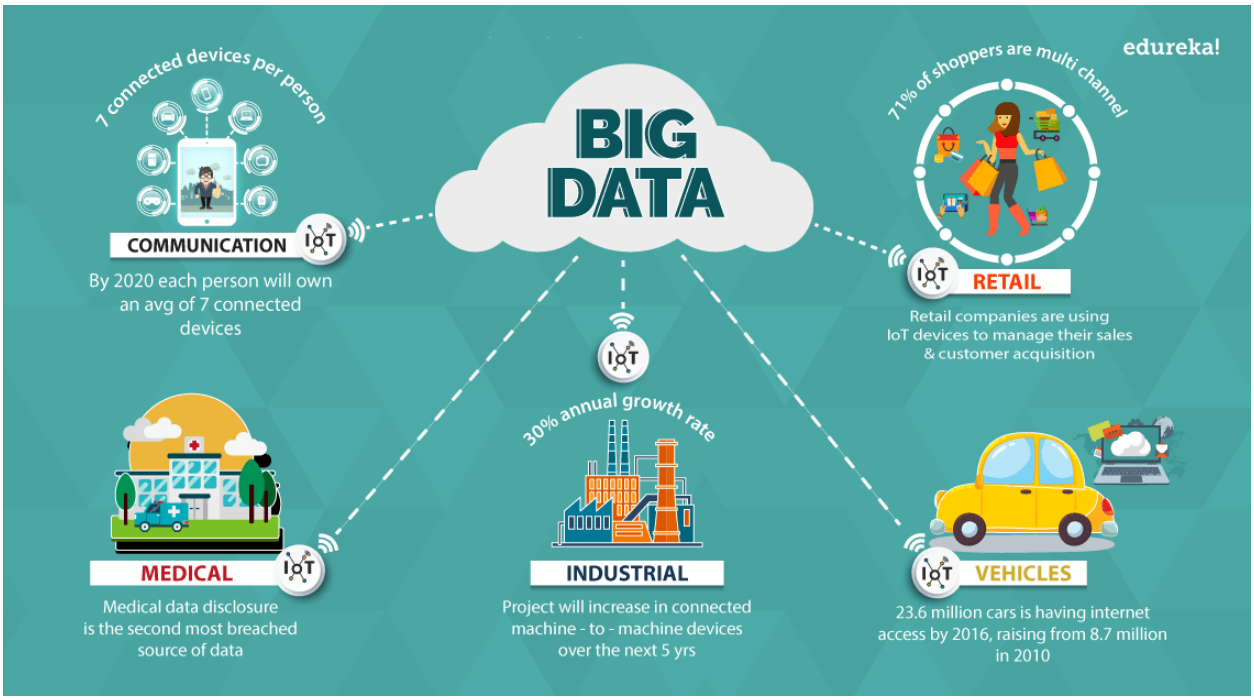
Among the characteristics of Big Data, value is perhaps the most important. No matter how fast the data is produced or its amount, it has to be reliable and useful. Otherwise, the data is not good enough for processing or analysis. Research says that poor quality data can lead to almost a 20% loss in a company’s revenue.

Data scientists first convert raw data into information. Then this data set is cleaned to retrieve the most useful data. Analysis and pattern identification is done on this data set. If the process is a success, the data can be considered to be valuable.

5. Veracity

This feature of Big Data is connected to the previous one. It defines the degree of trustworthiness of the data. As most of the data you encounter is unstructured, it is important to filter out the unnecessary information and use the rest for processing.

Applications of Big Data



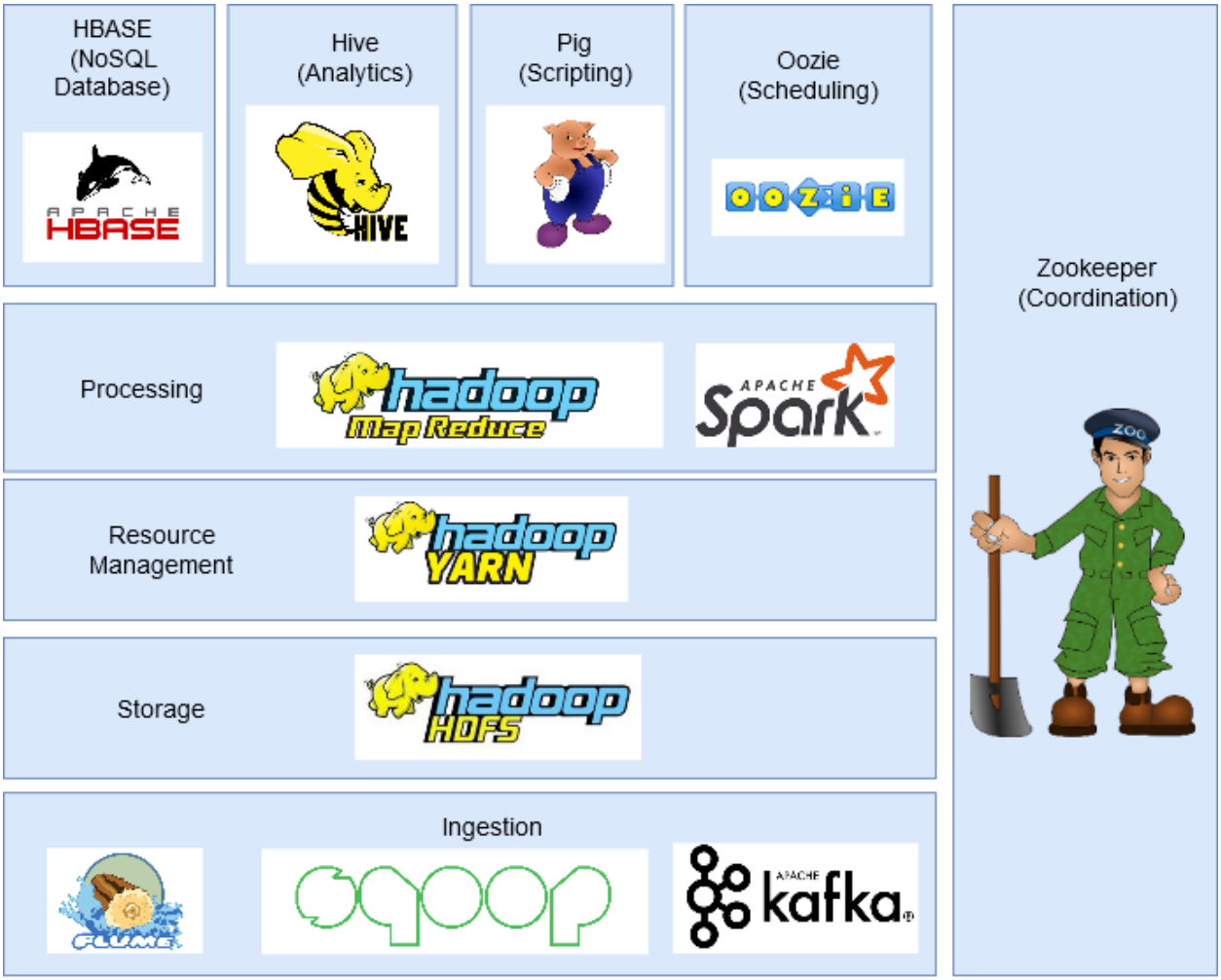
Hadoop

Apache Hadoop is an open-source framework based on Google's file system that can deal with big data in a distributed environment. This distributed environment is built up of a cluster of machines that work closely together to give an impression of a single working machine.

Here are some of the important properties of Hadoop you should know:

- Hadoop is highly scalable because it handles data in a distributed manner.
- Compared to vertical scaling in RDBMS, Hadoop offers horizontal scaling
- It creates and saves replicas of data making it fault-tolerant.
- It is economical as all the nodes in the cluster are commodity hardware which is nothing but inexpensive machines.
- Hadoop utilizes the data locality concept to process the data on the nodes on which they are stored rather than moving the data over the network thereby reducing traffic.
- It can handle any type of data: structured, semi-structured, and unstructured. This is extremely important in today's time because most of our data (emails, Instagram, Twitter, IoT devices, etc.) has no defined format.

Components of Hadoop Ecosystem



HDFS (Hadoop Distributed File System)

It is the storage component of Hadoop that stores data in the form of files.

Each file is divided into blocks of 128MB (configurable) and stores them on different machines in the cluster.

It has a master-slave architecture with two main components: Name Node and Data Node.

Name node is the master node and there is only one per cluster. Its task is to know where each block belonging to a file is lying in the cluster.

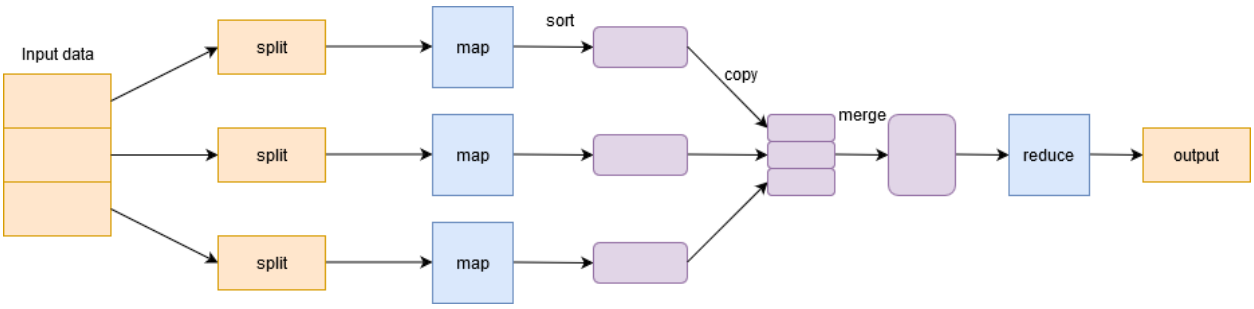
Data node is the slave node that stores the blocks of data and there are more than one per cluster. Its task is to retrieve the data as and when required. It keeps in constant touch with the Name node through heartbeats.

MapReduce

To handle Big Data, Hadoop relies on the MapReduce algorithm introduced by Google and makes it easy to distribute a job and run it in parallel in a cluster. It essentially divides a single task into multiple tasks and processes them on different machines.

In layman terms, it works in a divide-and-conquer manner and runs the processes on the machines to reduce traffic on the network.

It has two important phases: Map and Reduce.



Map phase filters, groups, and sorts the data. Input data is divided into multiple splits. Each map task works on a split of data in parallel on different machines and outputs a key-value pair. The output of this phase is acted upon by the reduce task and is known as the Reduce phase. It aggregates the data, summarizes the result, and stores it on HDFS.

YARN

YARN or Yet Another Resource Negotiator manages resources in the cluster and manages the applications over Hadoop. It allows data stored in HDFS to be processed and run by various data processing engines such as batch processing, stream processing, interactive processing, graph processing, and many more. This increases efficiency with the use of YARN.

HBase

HBase is a Column-based NoSQL database. It runs on top of HDFS and can handle any type of data. It allows for real-time processing and random read/write operations to be performed in the data.

Pig

Apache Pig

Pig was developed for analyzing large datasets and overcomes the difficulty to write map and reduce functions. It consists of two components: Pig Latin and Pig Engine.

Pig Latin is the Scripting Language that is similar to SQL. Pig Engine is the execution engine on which Pig Latin runs. Internally, the code written in Pig is converted to MapReduce functions and makes it very easy for programmers who aren't proficient in Java.

Hive

Hive is a distributed data warehouse system developed by Facebook. It allows for easy reading, writing, and managing files on HDFS. It has its own querying language for the purpose known as Hive Querying Language (HQL) which is very similar to SQL. This makes it very easy for programmers to write MapReduce functions using simple HQL queries.

Sqoop

A lot of applications still store data in relational databases, thus making them a very important source of data. Therefore, Sqoop plays an important part in bringing data from Relational Databases into HDFS.

The commands written in Sqoop internally converts into MapReduce tasks that are executed over HDFS. It works with almost all relational databases like MySQL, Postgres, SQLite, etc. It can also be used to export data from HDFS to RDBMS.

Flume

Flume is an open-source, reliable, and available service used to efficiently collect, aggregate, and move large amounts of data from multiple data sources into HDFS. It can collect data in real-time as well as in batch mode. It has a flexible architecture and is fault-tolerant with multiple recovery mechanisms.

Kafka

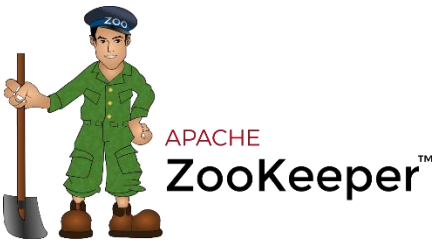
There are a lot of applications generating data and a commensurate number of applications consuming that data. But connecting them individually is a tough task. That's where Kafka comes in. It sits between the applications generating data (Producers) and the applications consuming data (Consumers).

Kafka is distributed and has in-built partitioning, replication, and fault-tolerance. It can handle streaming data and also allows businesses to analyze data in real-time.

Oozie

Oozie is a workflow scheduler system that allows users to link jobs written on various platforms like MapReduce, Hive, Pig, etc. Using Oozie you can schedule a job in advance and can create a pipeline of individual jobs to be executed sequentially or in parallel to achieve a bigger task. For example, you can use Oozie to perform ETL operations on data and then save the output in HDFS.

Zookeeper



In a Hadoop cluster, coordinating and synchronizing nodes can be a challenging task. Therefore, Zookeeper is the perfect tool for the problem.

It is an open source, distributed, and centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services across the cluster.

Spark

Spark is an alternative framework to Hadoop built on Scala but supports varied applications written in Java, Python, etc. Compared to MapReduce it provides in-memory processing which accounts for faster processing. In addition to batch processing offered by Hadoop, it can also handle real-time processing.

Hadoop Distributed File System

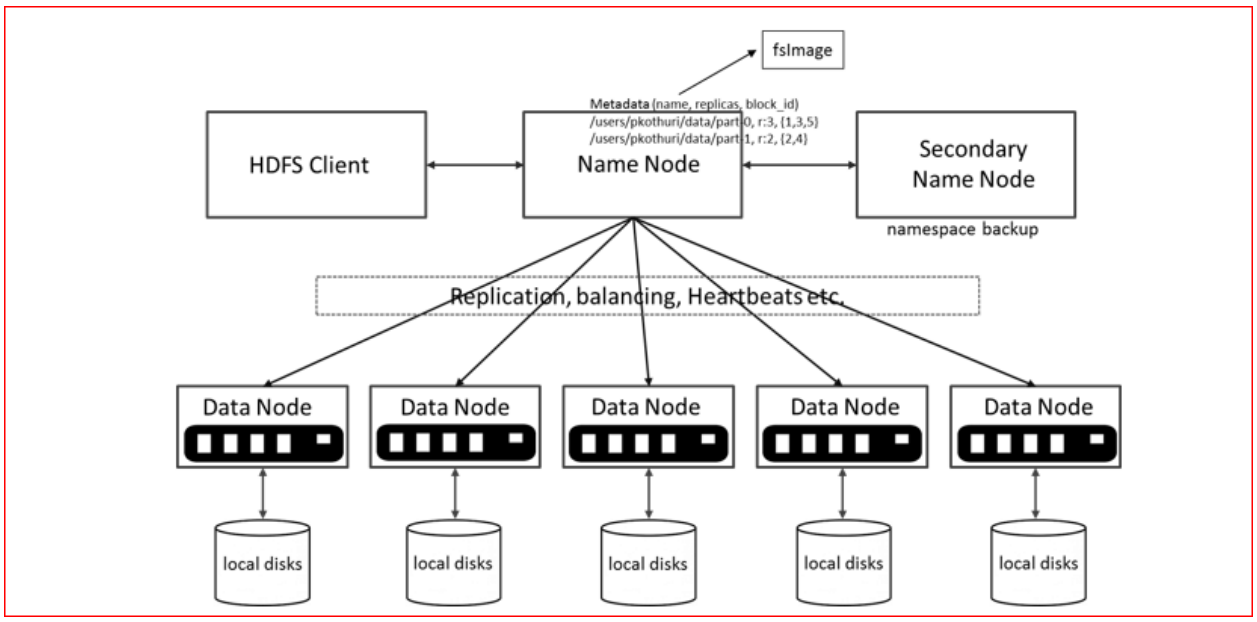
HDFS is a distributed file system that handles large data sets running on commodity hardware. It is used to scale a single Apache Hadoop cluster to hundreds (and even thousands) of nodes. HDFS is one of the major components of Apache Hadoop, the others being MapReduce and YARN. HDFS should not be confused with or replaced by Apache HBase, which is a column-oriented non-relational database management system that sits on top of HDFS and can better support real-time data needs with its in-memory processing engine.

or

Hadoop File System was developed using distributed file system design. It is run on 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.

Architecture of HDFS



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.

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.

Hadoop HDFS split large files into small chunks known as Blocks. Block is the physical representation of data. It contains a minimum amount of data that can be read or write. HDFS stores each file as blocks. HDFS client doesn't have any control on the block like block location, Namenode decides all such things.

By default, HDFS block size is 128MB which you can change as per your requirement. All HDFS blocks are the same size except the last block, which can be either the same size or smaller. Hadoop framework break files into 128 MB blocks and then stores into the Hadoop file system. Apache Hadoop application is responsible for distributing the data block across multiple nodes.

The default size of each block is 128MB in Apache Hadoop 2.x (64MB is Apache Hadoop 1.x)

Example-

Suppose file size is 513MB, and we are using the default configuration of block size 128MB. Then, the Hadoop framework will create 5 blocks, first four blocks 128MB, but the last block will be of 1MB only.

Hence from the example it clear that it is not necessary that in HDFS each file stored should be an exact multiple of the configured block size 128mb, 256mb etc. Therefore, final block for file uses only as much space as is needed.

Block in HDFS simplifies the storage of the **Datanodes**. **Namenode** maintains metadata of all the blocks. HDFS Datanode does not need to concern about the block metadata like file permissions etc.

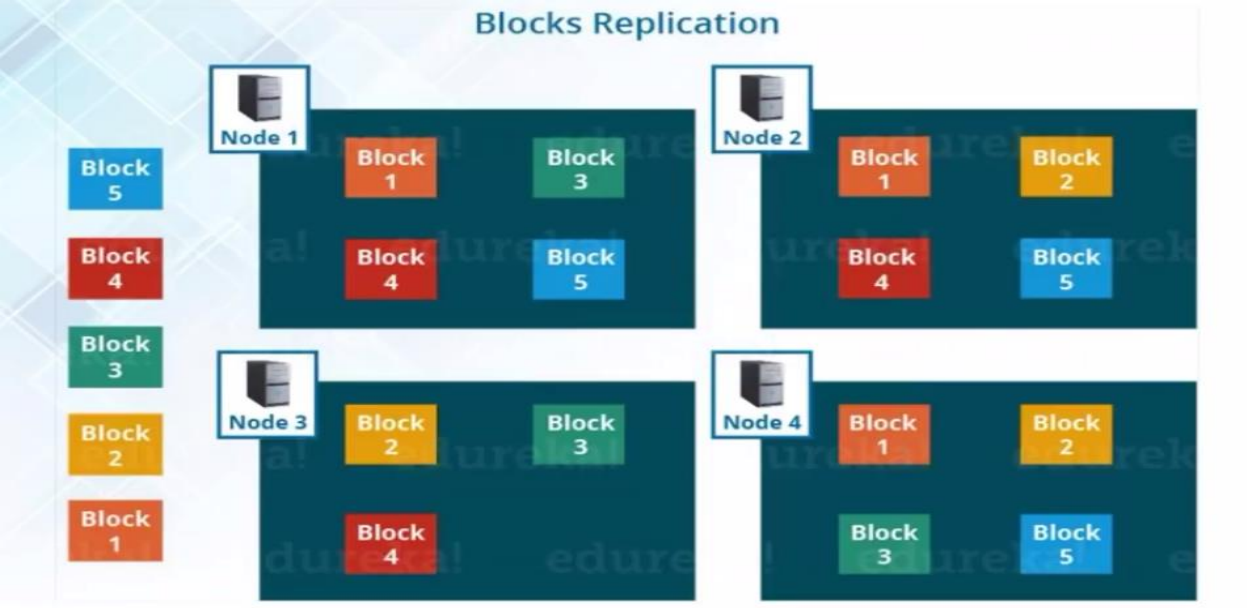
Why is HDFS Block size 128 MB?

HDFS store terabytes and petabytes of data. If HDFS Block size is 4kb like Linux file system, then we will have too many data blocks in Hadoop HDFS, hence too much of metadata. So, maintaining and managing this huge number of blocks and metadata will create huge overhead and traffic which is something which we don't want.

Block size can't be so large that the system is waiting a very long time for one last unit of data processing to finish its work.

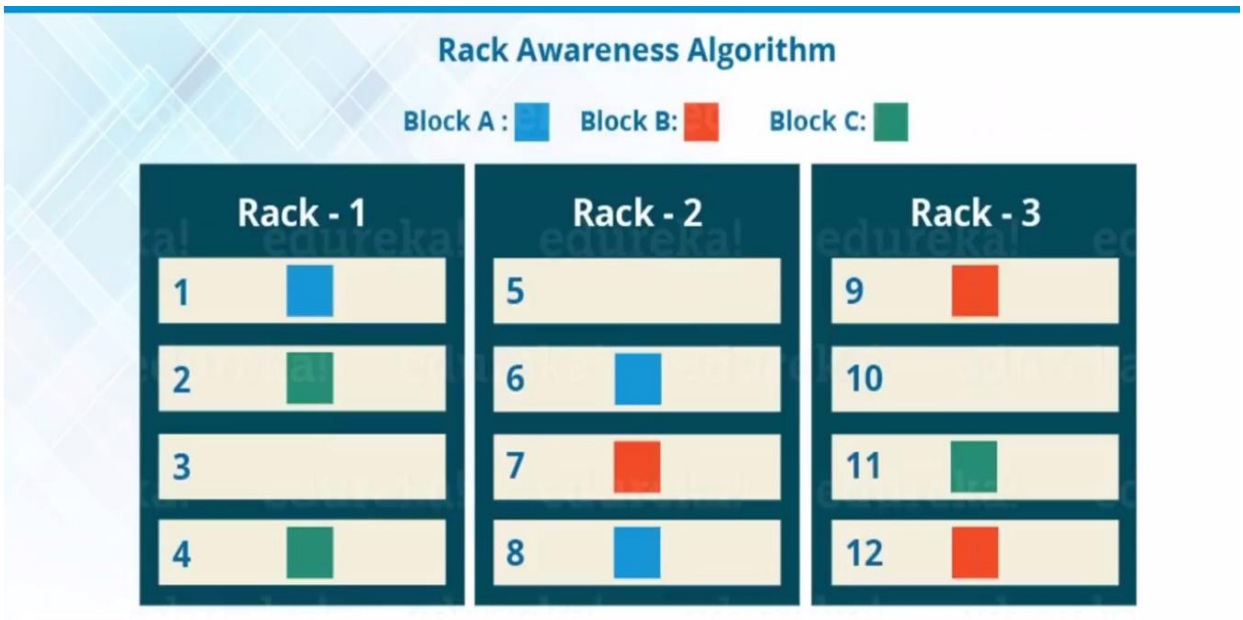
Hadoop follows default replication factor "3"

Replication ensures the availability of the data. Replication is nothing but making a copy of something and the number of times you make a copy of that particular thing can be expressed as it's Replication Factor. In File blocks that the HDFS stores the data in the form of various blocks at the same time Hadoop is also configured to make a copy of those file blocks. By default the Replication Factor for Hadoop is set to 3 which can be configured means you can change it Manually as per your requirements, replication blocks are made for the backup purpose.



How does Hadoop decide where to store the replicas of the blocks created?

The **Rack** is the collection of around 40-50 DataNodes connected using the same network switch. If the network goes down, the whole rack will be unavailable. A large Hadoop cluster is deployed in multiple racks.

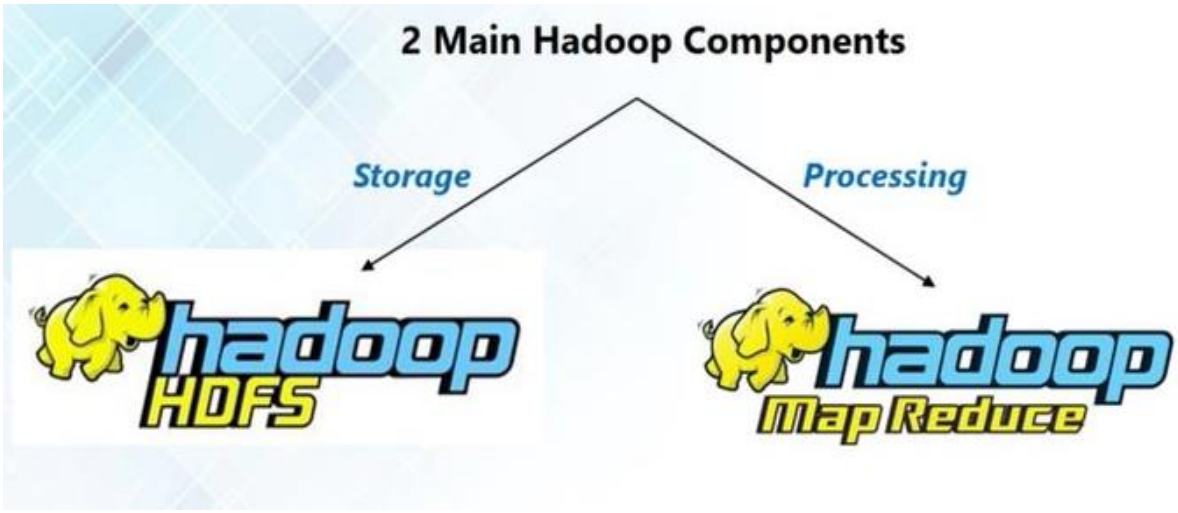


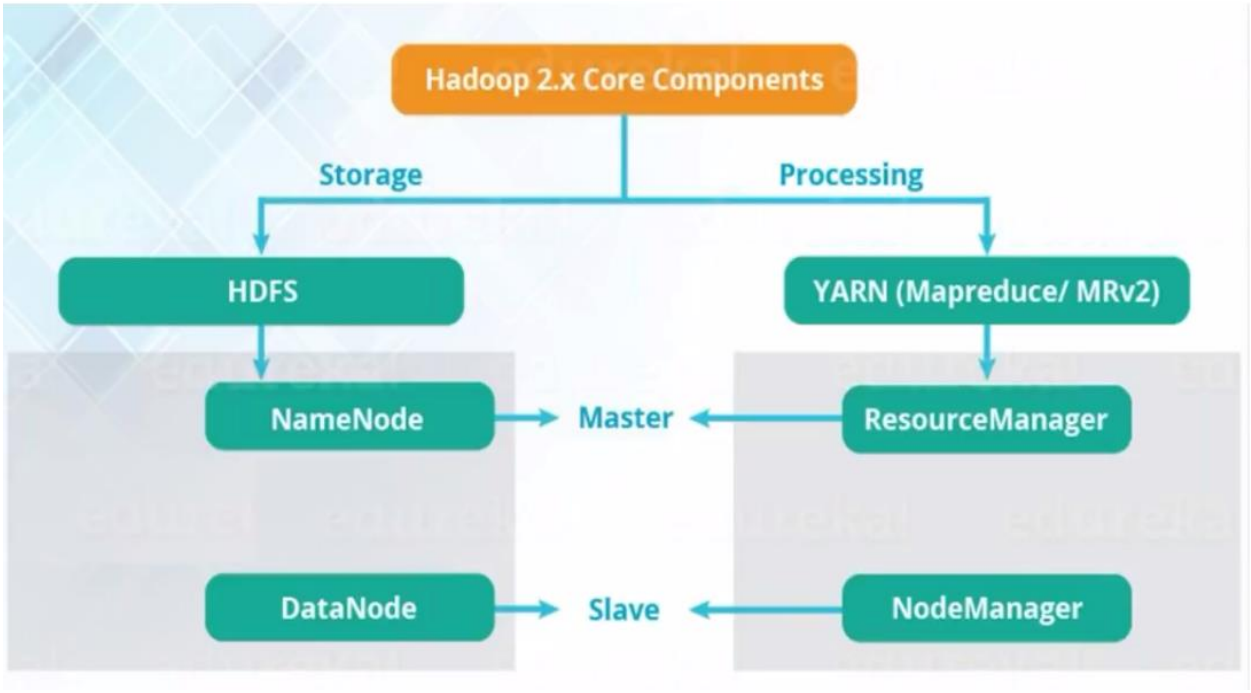
In a large Hadoop cluster, there are multiple racks. Each rack consists of DataNodes. Communication between the DataNodes on the same rack is more efficient as compared to the communication between DataNodes residing on different racks.

To reduce the network traffic during file read/write, NameNode chooses the closest DataNode for serving the client read/write request. NameNode maintains rack ids of each DataNode to achieve this rack information. This concept of choosing the closest DataNode based on the rack information is known as Rack Awareness.

The reasons for the Rack Awareness in Hadoop are:

- To reduce the network traffic while file read/write, which improves the cluster performance.
- To achieve fault tolerance, even when the rack goes down (discussed later in this article).
- Achieve high availability of data so that data is available even in unfavorable conditions.
- To reduce the latency, that is, to make the file read/write operations done with lower delay.





HDFS Commands Type:

User Commands:
hdfs dfs – runs filesystem commands on the HDFS
hdfs fsck – runs a filesystem checking command.

Administration Command:
hdfs dfsadmin – runs HDFS administration commands.

A) HDFS User Commands:

1) hdfs dfs -ls /
List all the files/directories for the given hdfs destination path

```
hduser@Hadoop:~$ hdfs dfs -ls /
Found 2 items
drwxr-xr-x   - hduser supergroup          0 2018-03-31 18:08 /dir_1_underHDFS
drwxr-xr-x   - hduser supergroup          0 2018-03-31 18:08 /dir_2_underHDFS
hduser@Hadoop:~$
```

2) hdfs dfs -ls –R /"directoryname"

Recursively list all files in hadoop directory and all subdirectories in the directory.

```
hduser@Hadoop:~$ hdfs dfs -ls -R /dir_1_underHDFS
-rw-r--r--   1 hduser supergroup          0 2018-03-31 18:18 /dir_1_underHDFS/file1_underdir.txt
hduser@Hadoop:~$
```

3) hdfs dfs -cat /hadoop/test

This command will display the content of the HDFS file test on your stdout.

```
hduser@Hadoop:~$ hdfs dfs -cat /hadoop/test
Hi, This is a test file to show the sample command !!
^Chduser@Hadoop:~$ ^C
hduser@Hadoop:~$
```

4) hdfs dfs -put /home/hduser/samplefile /hadoop

Copies the file from local file system to HDFS.

{File samplefile was already present in local file system}

```
hduser@Hadoop:~$ hdfs dfs -put /home/hduser/samplefile /hadoop
hduser@Hadoop:~$ hdfs dfs -ls /hadoop
Found 2 items
-rw-r--r--   1 hduser supergroup          0 2018-03-31 18:32 /hadoop/samplefile
-rw-r--r--   1 hduser supergroup         54 2018-03-31 18:27 /hadoop/test
hduser@Hadoop:~$
```

5) hdfs dfs -put –f /home/hduser/samplefile /hadoop

Copies the file from local file system to HDFS, and in case the local already exist in the given destination path, using -f option with put command will overwrite it.

6) hdfs dfs -cp /hadoop/test /hadoop1

Copies file from source to destination on HDFS. In this case, copying file 'test' from hadoop directory to hadoop1 directory.

```
hduser@Hadoop:~$ hdfs dfs -cp /hadoop/test /hadoop1
hduser@Hadoop:~$ hdfs dfs -ls /hadoop1
Found 1 items
-rw-r--r--   1 hduser supergroup         54 2018-03-31 18:41 /hadoop1/test
hduser@Hadoop:~$
```

7) hdfs dfs -rm /hadoop1/test

Deletes the file (sends it to the trash).

```
hduser@Hadoop:~$ hdfs dfs -rm /hadoop1/test
18/03/31 18:46:20 INFO fs.TrashPolicyDefault: Namenode trash configuration: Deletion interval = 0 minutes, Emptier interval = 0 minutes.
Deleted /hadoop1/test
hduser@Hadoop:~$ hdfs dfs -ls /hadoop1
hduser@Hadoop:~$
hduser@Hadoop:~$
hduser@Hadoop:~$
hduser@Hadoop:~$
```

8) hdfs dfs -df /hadoop

Shows the capacity, free and used space of the filesystem.

```
hduser@Hadoop:~$ hdfs dfs -df /hadoop
Filesystem                     Size   Used    Available  Use%
hdfs://localhost:54310  36095803392  65471  25803112448    0%
hduser@Hadoop:~$ ^C
```

9) hdfs dfs -du -h /hadoop/test

Show the amount of space, in bytes, used by the files that match the specified file pattern. Formats the sizes of files in a human-readable fashion.

```
hduser@Hadoop:~$ hdfs dfs -du -h /hadoop/test
54  /hadoop/test
hduser@Hadoop:~$
```

10) hdfs dfs -df -h /hadoop

Shows the capacity, free and used space of the filesystem. -h parameter Formats the sizes of files in a human-readable fashion.

```
hduser@Hadoop:~$ hdfs dfs -df -h /hadoop
Filesystem                 Size  Used  Available  Use%
hdfs://localhost:54310  33.6 G  56 K     24.0 G    0%
hduser@Hadoop:~$
```

B) HDFS Administration Command:

1) hadoop version

To check the version of Hadoop

```
hduser@Hadoop:~$ hadoop version
Hadoop 2.7.4
Subversion https://shv@git-wip-us.apache.org/repos/asf/hadoop.git -r cd915e1e8d9d0131462a0b7301586c175728a282
Compiled by kshvachk on 2017-08-01T00:29Z
Compiled with protoc 2.5.0
From source with checksum 50b0468318b4ce9bd24dc467b7ce1148
This command was run using /usr/local/hadoop/share/hadoop/common/hadoop-common-2.7.4.jar
hduser@Hadoop:~$
```

2) hdfs fsck /

It checks the health of the Hadoop file system.

```
hduser@Hadoop:~$ hdfs fsck /
Connecting to namenode via http://localhost:50070/fsck?ugi=hduser&path=%2F
FSCK started by hduser (auth:SIMPLE) from /127.0.0.1 for path / at Sat Mar 31 19:05:48 IST 2018
...Status: HEALTHY
 Total size:    54 B
 Total dirs:    5
 Total files:   3
 Total symlinks:                0
 Total blocks (validated):      1 (avg. block size 54 B)
 Minimally replicated blocks:   1 (100.0 %)
 Over-replicated blocks:        0 (0.0 %)
 Under-replicated blocks:       0 (0.0 %)
 Mis-replicated blocks:         0 (0.0 %)
 Default replication factor:    1
 Average block replication:     1.0
 Corrupt blocks:                0
 Missing replicas:              0 (0.0 %)
 Number of data-nodes:          1
 Number of racks:               1
FSCK ended at Sat Mar 31 19:05:48 IST 2018 in 14 milliseconds


The filesystem under path '/' is HEALTHY
```

3) hdfs dfsadmin -refreshNodes

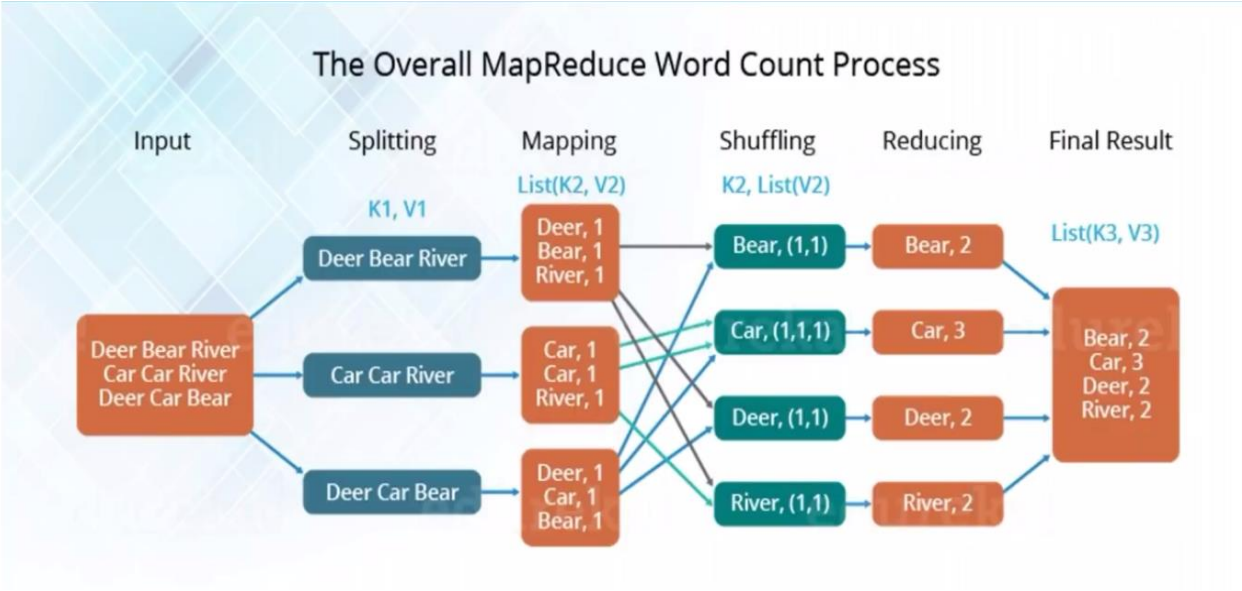
Re-read the hosts and exclude files to update the set of Datanodes that are allowed to connect to the Namenode and those that should be decommissioned or recommissioned.

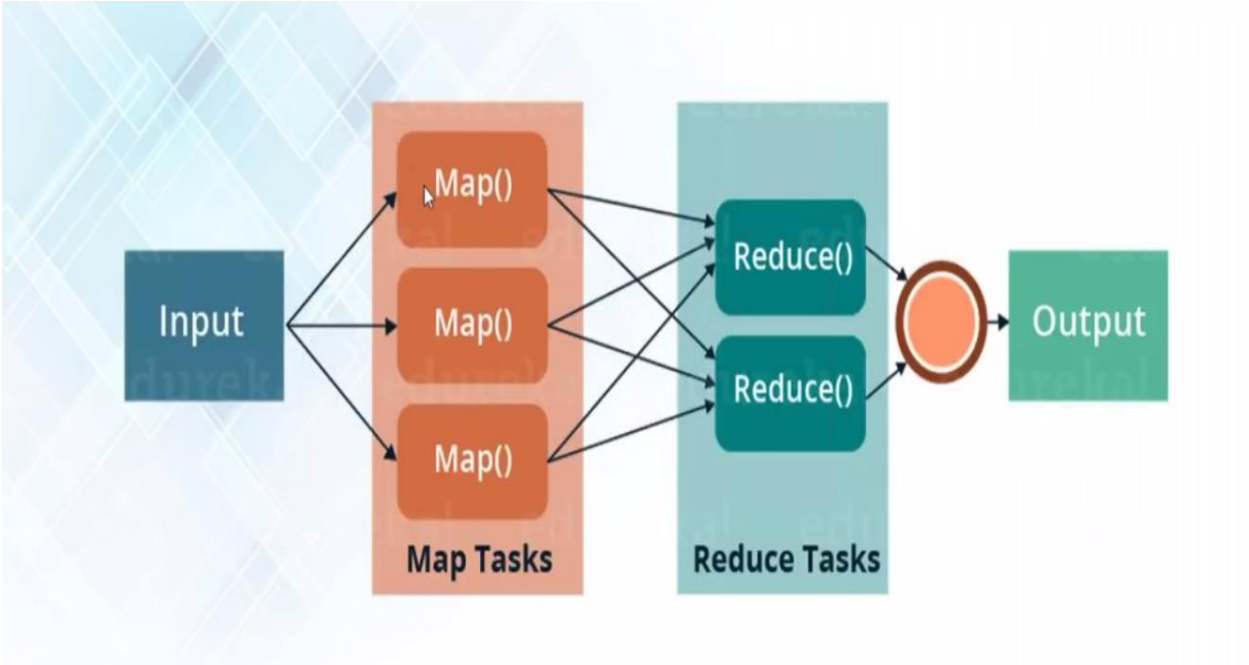
```
hduser@Hadoop:~$ hdfs dfsadmin -refreshNodes
Refresh nodes successful
hduser@Hadoop:~$
```

4) hdfs namenode -format

This command runs only once while starting the Hadoop environment. It Formats the NameNode so never use this command as this will delete all the data in the Hadoop cluster, all data will be lost with no recovery option.

Phases of MapReduce “WordCount” program

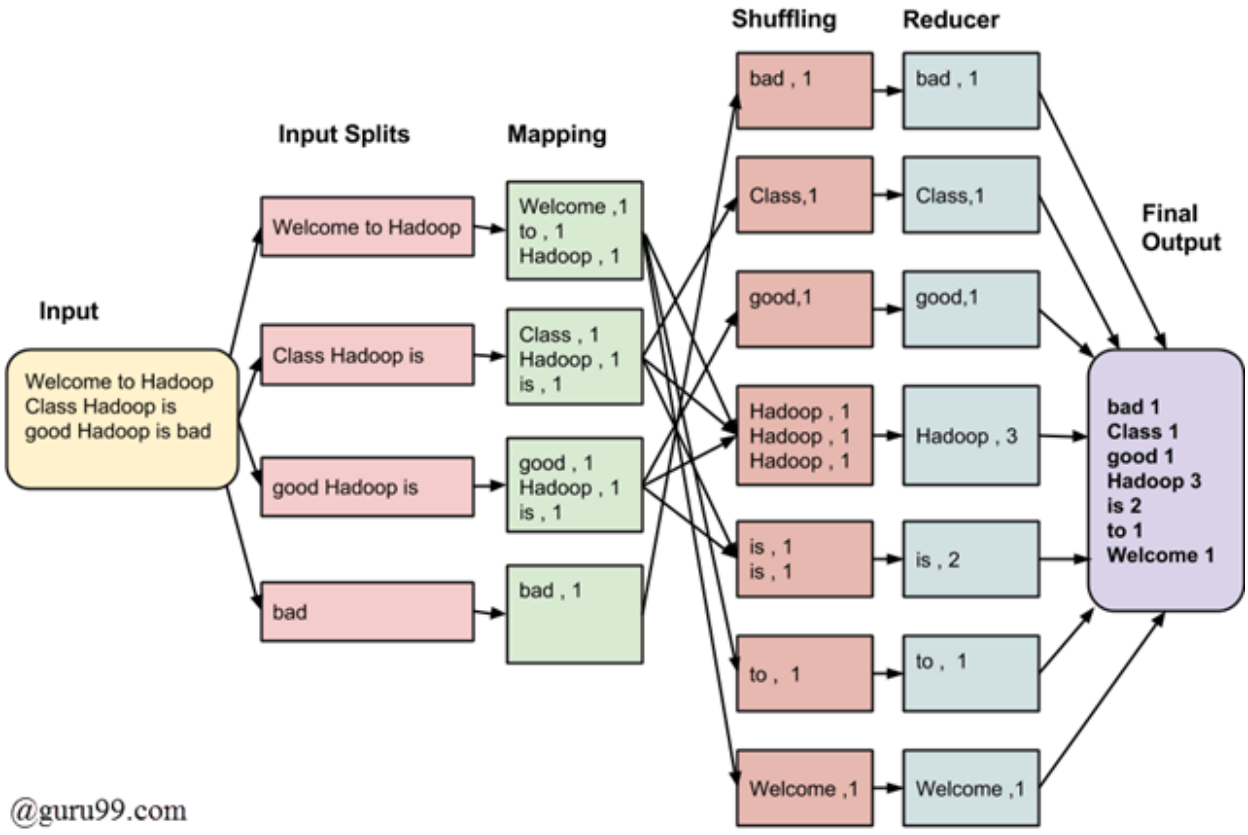




MapReduce is a software framework and programming model used for processing huge amounts of data. MapReduce program work in two phases, namely, Map and Reduce. Map tasks deal with splitting and mapping of data while Reduce tasks shuffle and reduce the data.

Hadoop is capable of running MapReduce programs written in various languages: Java, Ruby, Python, and C++. The programs of Map Reduce in cloud computing are parallel in nature, thus are very useful for performing large-scale data analysis using multiple machines in the cluster.

The input to each phase is key-value pairs. In addition, every programmer needs to specify two functions: map function and reduce function.



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 of MapReduce in Big Data

Input Splits:

An input to a MapReduce in Big Data 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

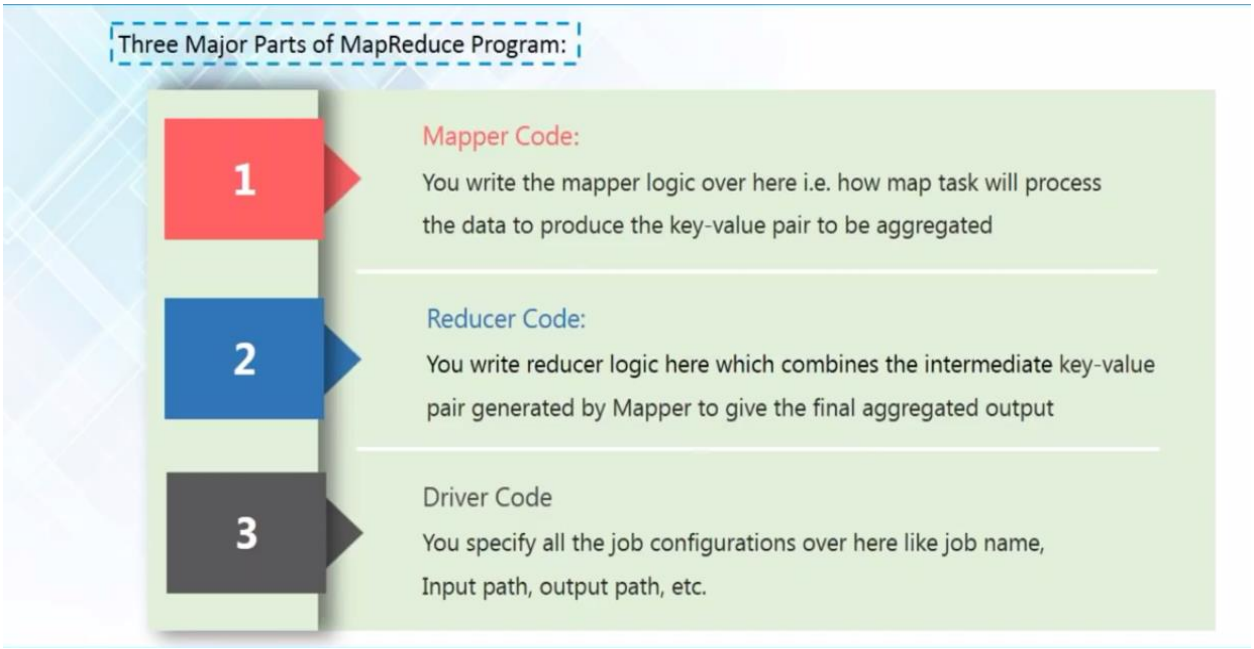
In this phase, output values from the Shuffling phase are aggregated. This phase combines values from Shuffling phase and returns a single output value. In short, this phase summarizes the complete dataset.

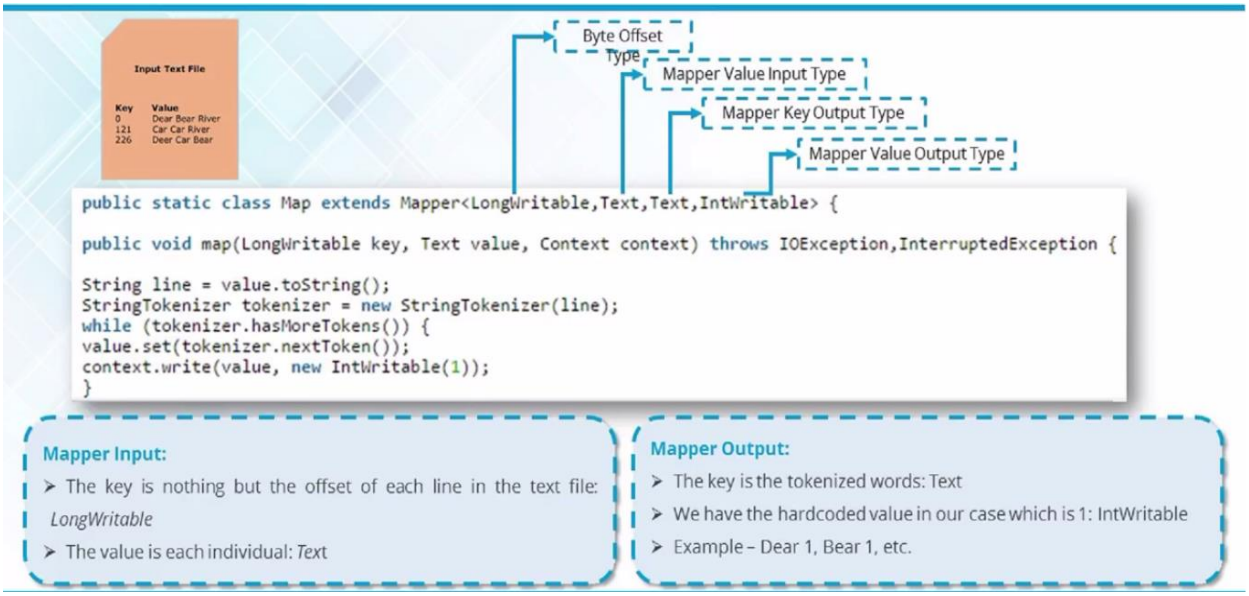
MapReduce Architecture explained in detail:

- One map task is created for each split which then executes map function for each record in the split.
- It is always beneficial to have multiple splits because the time taken to process a split is small as compared to the time taken for processing of the whole input. When the splits are smaller, the processing is better to load balanced since we are processing the splits in parallel.
- However, it is also not desirable to have splits too small in size. When splits are too small, the overload of managing the splits and map task creation begins to dominate the total job execution time.
- For most jobs, it is better to make a split size equal to the size of an HDFS block (which is 64 MB, by default).
- Execution of map tasks results into writing output to a local disk on the respective node and not to HDFS.
- Reason for choosing local disk over HDFS is, to avoid replication which takes place in case of HDFS store operation.
- Map output is intermediate output which is processed by reduce tasks to produce the final output.
- Once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication becomes overkill.
- In the event of node failure, before the map output is consumed by the reduce task, Hadoop reruns the map task on another node and re-creates the map output.

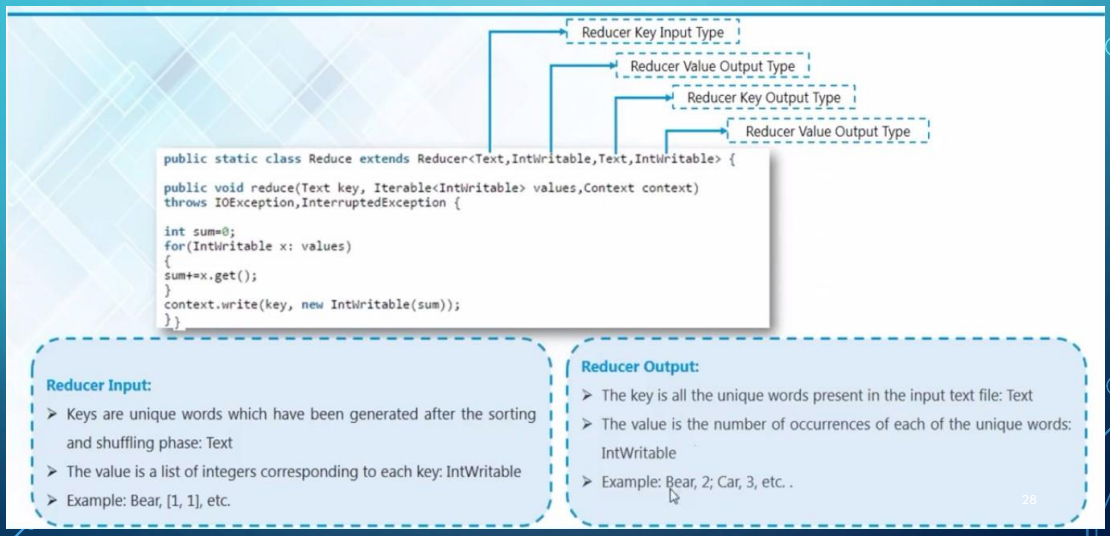
- Reduce task doesn't work on the concept of data locality. An output of every map task is fed to the reduce task. Map output is transferred to the machine where reduce task is running.
- On this machine, the output is merged and then passed to the user-defined reduce function.
- Unlike the map output, reduce output is stored in HDFS (the first replica is stored on the local node and other replicas are stored on off-rack nodes). So, writing the reduce output.

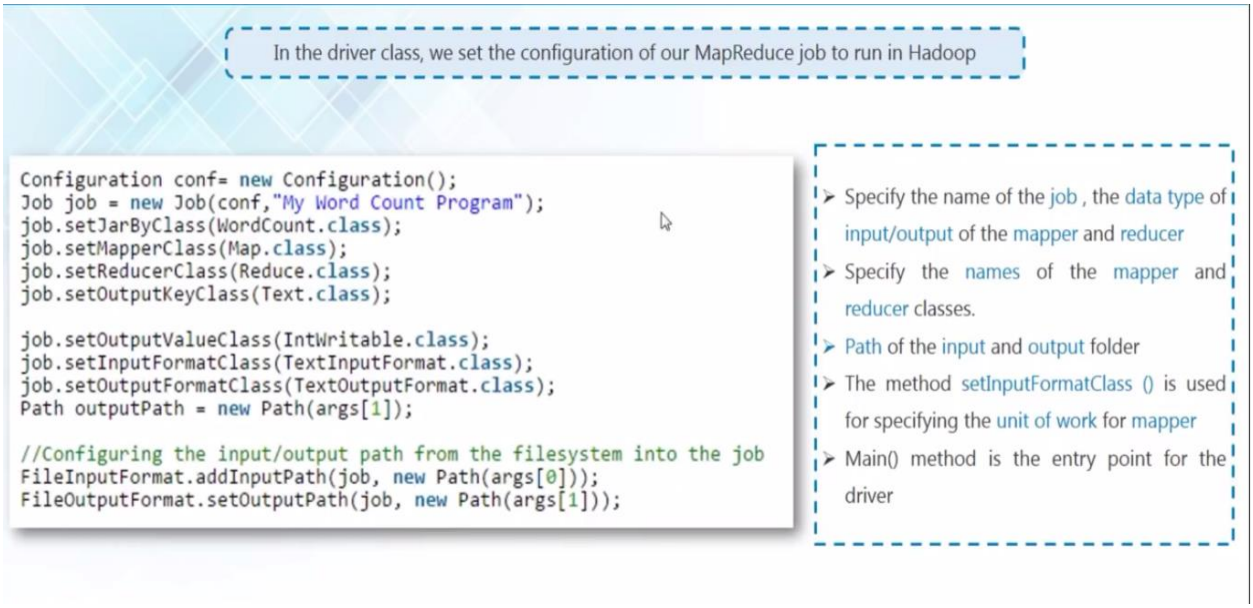
Part of map reduce program.





REDUCER CODE





MapReduce "WordCount" program?

 # WordCountDriver

```
import java.io.IOException;

import java.net.URI;
import java.net.URISyntaxException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
//import org.apache.hadoop.mapred.FileInputFormat;
//import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;


public class WordCountDriver {
        public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException, URISyntaxException
```

```
        {
                Configuration conf = new Configuration();
                //FileSystem fs = FileSystem.get(conf);
                //System.out.println(fs.getUri());
                Job j = new Job();// getConf(), "Max temperature");
                j.setJobName("My First Job");
                j.setJarByClass(WordCountDriver.class );
                j.setMapperClass(WordCountMapper.class );
                //j.setCombinerClass(WordCountReducer.class);
                //j.setPartitionerClass(WordCountPartitioner.class);
                j.setReducerClass(WordCountReducer.class);
                //j.setNumReduceTasks(0);
                //j.setCombinerClass(WordCountReducer.class);
                //j.setCombinerClass(WordCountReducer.class);
                //j.setNumReduceTasks(0);
                j.setOutputKeyClass(Text.class);
                j.setOutputValueClass(IntWritable.class);
                FileInputFormat.addInputPath(j, new Path(args[0]));
                FileOutputFormat.setOutputPath(j, new Path(args[1]));
                URI uri = new URI(args[1].toString());
                FileSystem fs =  FileSystem.get(uri, conf);
                boolean x = fs.delete(new Path(uri),true);
                int abc =  j.waitForCompletion(true) ? 0 : 1;
                }
}
```

 # WordCountMapper

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.DoubleWritable;
```

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class WordCountMapper extends Mapper<LongWritable,  Text, Text, IntWritable>

{

	@Override
	protected void map(LongWritable key, Text value,
				org.apache.hadoop.mapreduce.Mapper.Context context)
throws IOException, InterruptedException {
String inputstring = value.toString();
//StringTokenizer stk = new StringTokenizer(inputstring);
//while(stk.hasMoreTokens())
//{
//	context.write(new Text(stk.nextToken()),new IntWritable(1));
//}
//for(int i =0; i<inputstring.split(" ").length;i++)
//{
//	context.write(new Text(inputstring.split(" ")[i]),new IntWritable(1));

for (String x : inputstring.split(" "))
	//how are you
	//[0] how
	//[1] are
	//[2] you
{
	context.write(new Text(x),new IntWritable(1));
	// how 1
```

```
        //are 1
        //you 1
}
        }

}
```

 # WordCountMapper

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.Reducer.Context;

public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable>{

        @Override
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
                int y = 0;
                for(IntWritable x : values)
                {
                        y ++;

                }

                context.write(key, new IntWritable(y) );
                }
}
```

Combiner

A Combiner, also known as a semi-reducer, is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class.

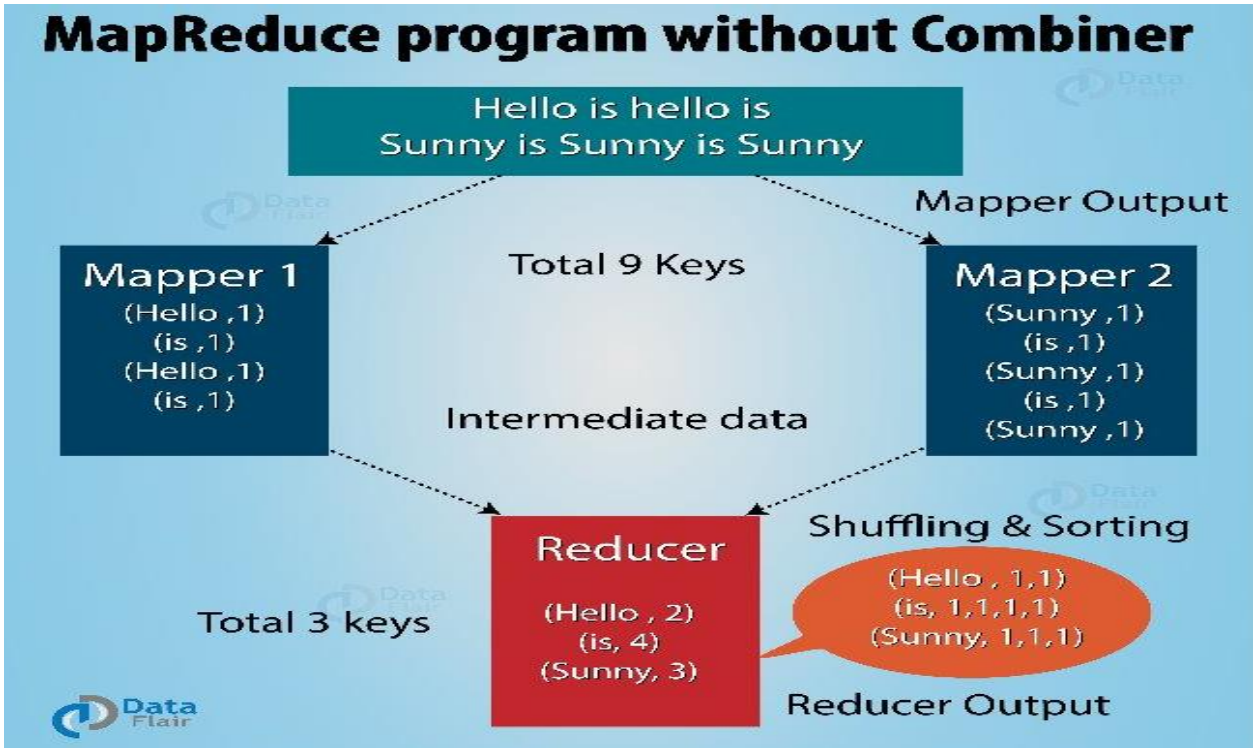
The main function of a Combiner is to summarize the map output records with the same key. The output (key-value collection) of the combiner will be sent over the network to the actual Reducer task as input.

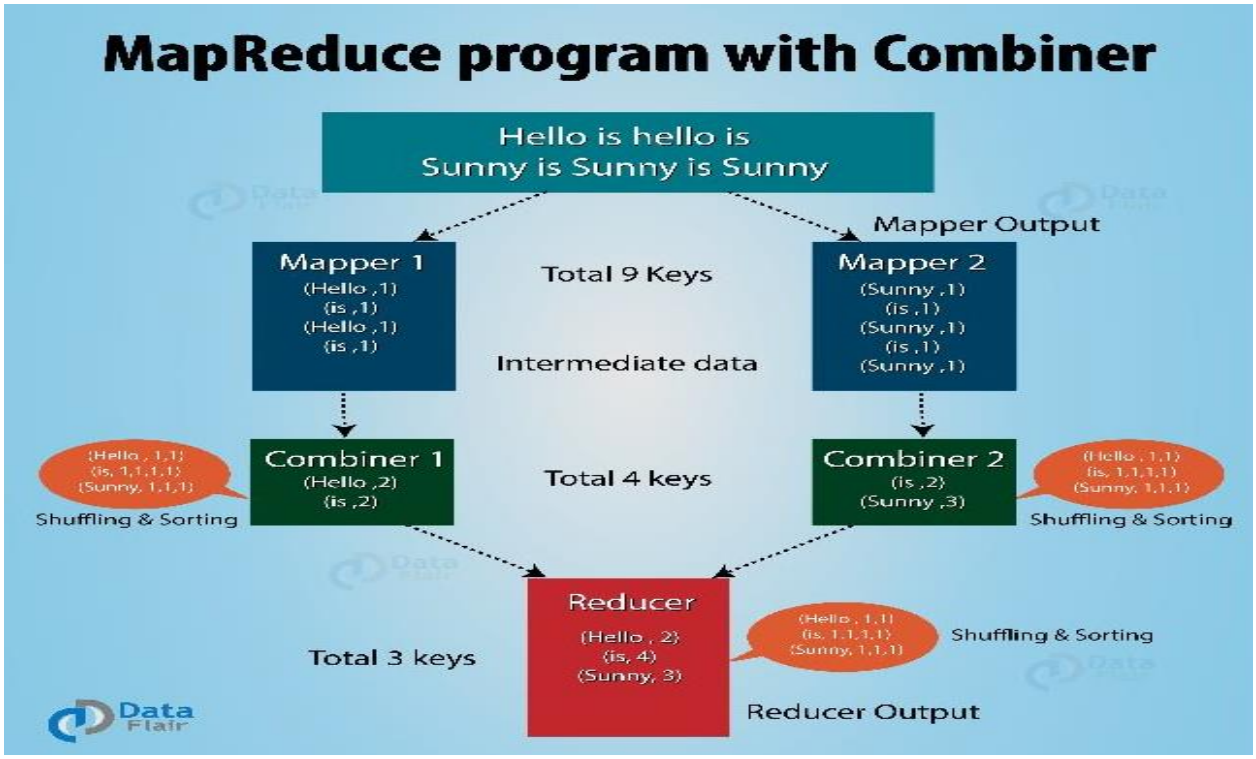
How Combiner Works?

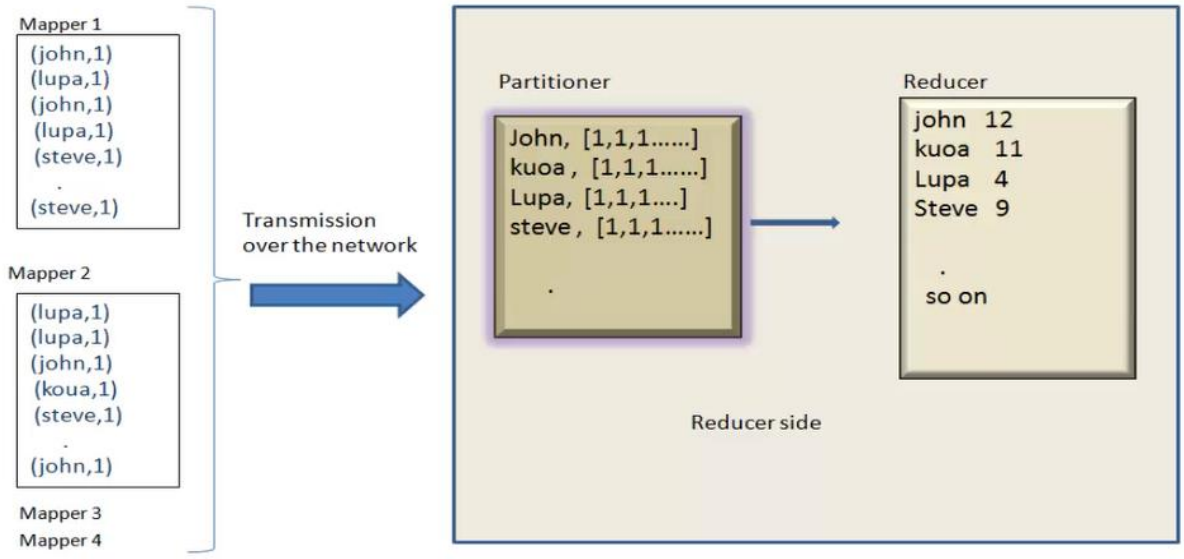
Here is a brief summary on how MapReduce Combiner works −

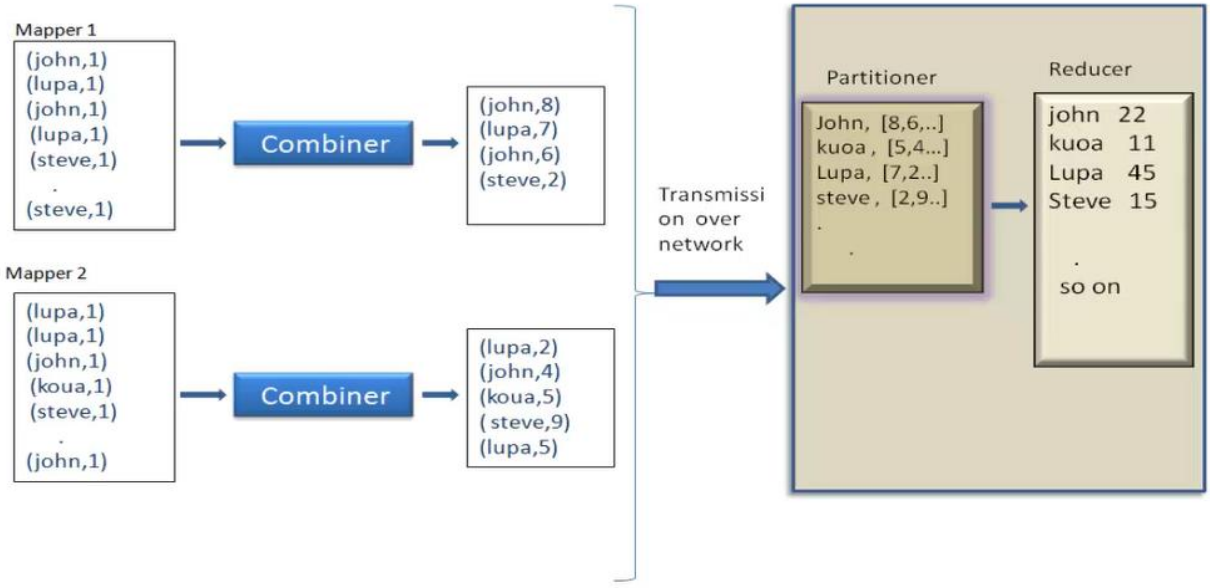
- A combiner does not have a predefined interface and it must implement the Reducer interface's reduce() method.
- A combiner operates on each map output key. It must have the same output key-value types as the Reducer class.
- A combiner can produce summary information from a large dataset because it replaces the original Map output.

Although, Combiner is optional, yet it helps segregating data into multiple groups for Reduce phase, which makes it easier to process.


MapReduce program without Combiner
Hello is hello is
Sunny is Sunny is Sunny
Mapper Output
Mapper 1
(Hello ,1)
(is ,1)
(Hello ,1)
(is ,1)
Total 9 Keys
Intermediate data
Mapper 2
(Sunny ,1)
(is ,1)
(Sunny ,1)
(is ,1)
(Sunny ,1)
Shuffling & Sorting
Reducer
(Hello , 2)
(is, 4)
(Sunny, 3)
(Hello , 1,1)
(is, 1,1,1,1)
(Sunny, 1,1,1)
Total 3 keys
Reducer Output
Data Flair


MapReduce program with Combiner
Hello is hello is
Sunny is Sunny is Sunny
Mapper Output
Mapper 1
(Hello ,1)
(is ,1)
(Hello ,1)
(is ,1)
Total 9 Keys
Intermediate data
Mapper 2
(Sunny ,1)
(is ,1)
(Sunny ,1)
(is ,1)
(Sunny ,1)
Combiner 1
(Hello ,2)
(is ,2)
Total 4 keys
Combiner 2
(is ,2)
(Sunny ,3)
Shuffling & Sorting
Shuffling & Sorting
Reducer
(Hello , 2)
(is, 4)
(Sunny, 3)
Shuffling & Sorting
Total 3 keys
Reducer Output
Data Flair



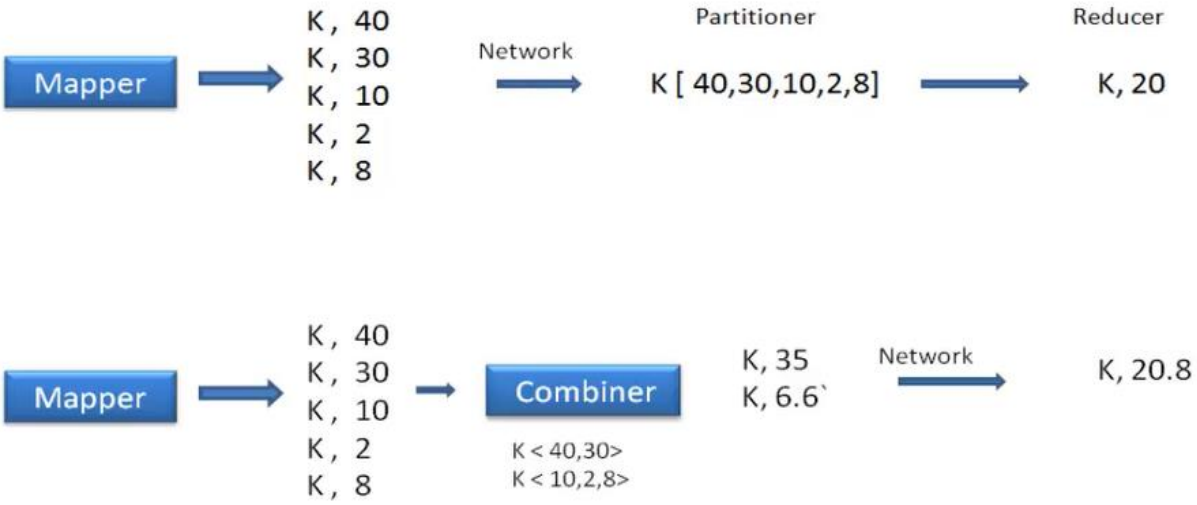


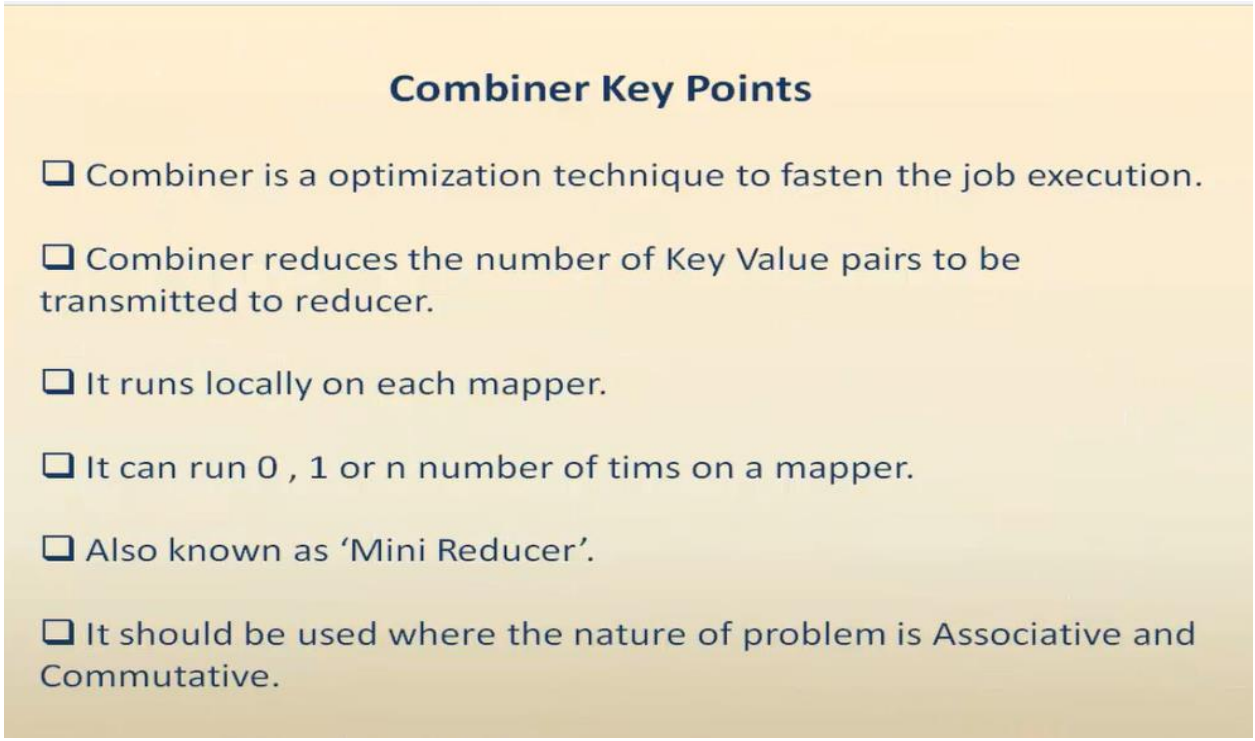
Condition for combiner

- The nature of the problem should be:

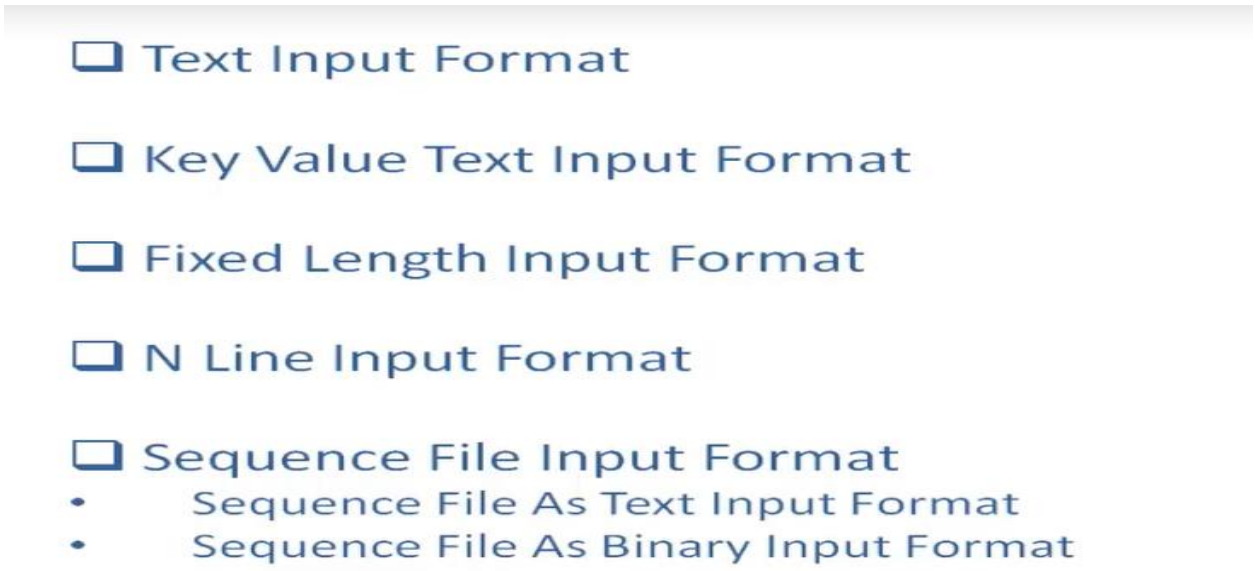
Commutative A+B = B+A

Associative (A+B)+C = A+(B+C)

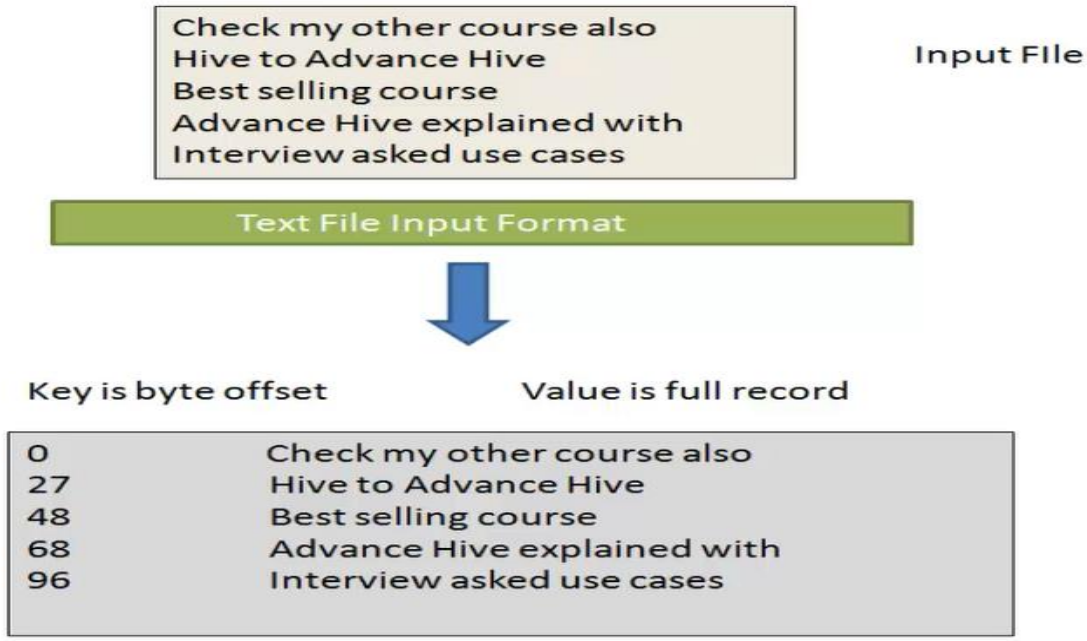




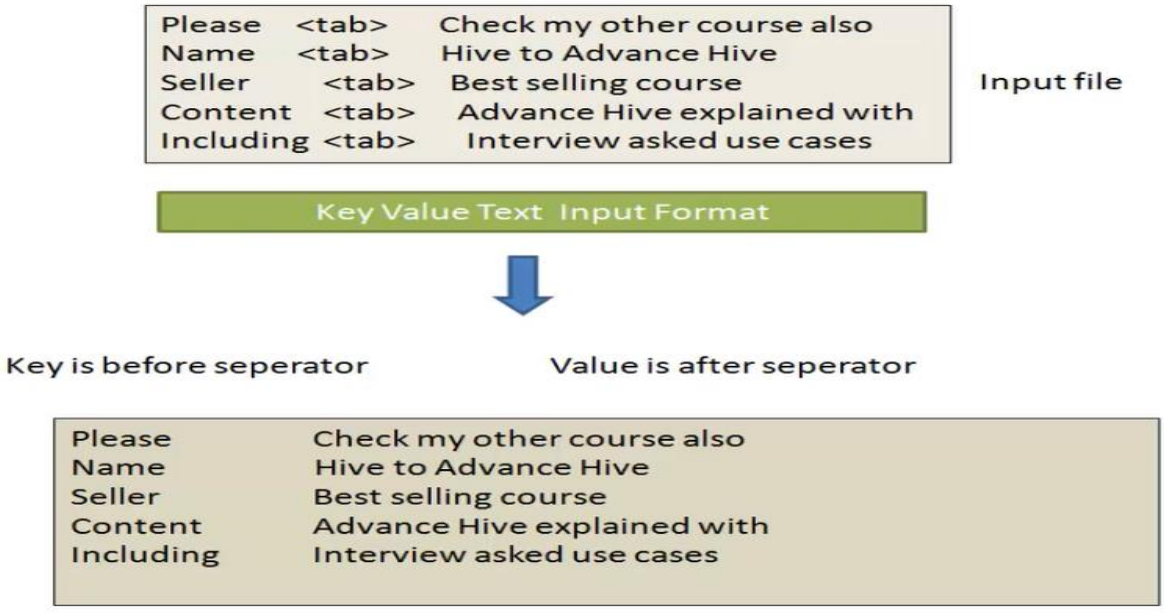
File input formats



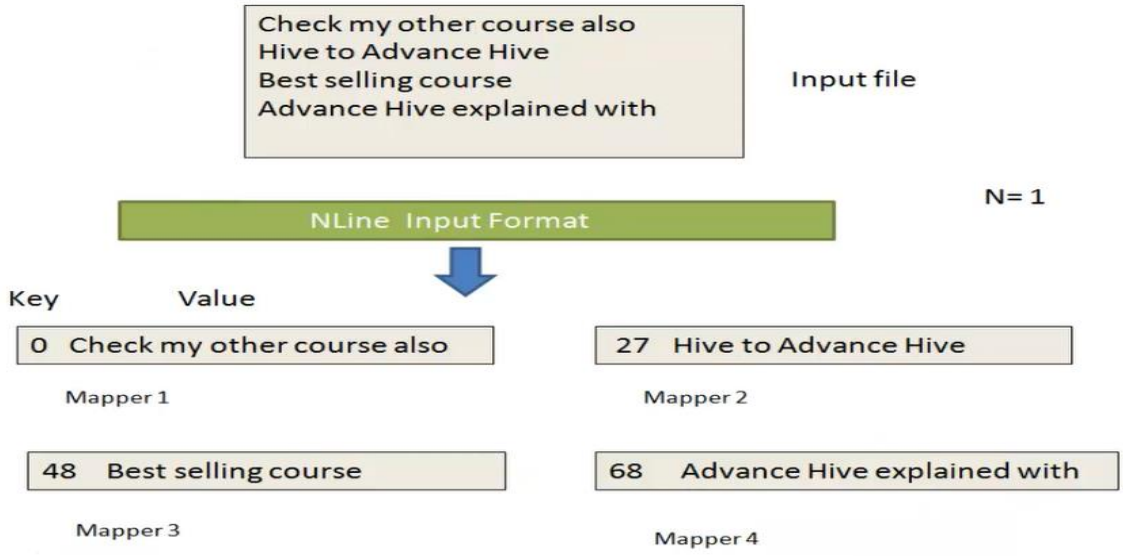
Text file input format



Key value text input format



Nline input format



Sequence File Input Format

- Format Used to read Sequence files.
- Reads the data from Intermediate Map files.
- Stores the data in form of binary key value pairs
- Keys and Values are User defined.

• *Sequence File as Text Input Format*: Converts key value pair to text objects by calling tostring() method.

•*Sequence File as Binary Input Format*: Retrieves the Sequence file's key and values as Binary objects.

Java Wrapper Classes

Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

The table below shows the primitive type and the equivalent wrapper class:

Wrapper classes (in java)	Primitive types	Box classes (in Hadoop)
Integer	int	IntWritable
Long	long	LongWritable
Float	float	FloatWritable
Double	Double	DoubleWritable
String	String	Text

All these box classes implemented writable and WritableComparable interface so that they can compare with each other.

Conversion:

int	IntWritable	(new IntWritable (int))
long	LongWritable	(new LongWritable (long))
float	FloatWritable	(new FloatWritable(float))
double	DoubleWritable	(new DoubleWritable(double))
String	Text	(new Text(string))

Writable and Writable Comparable:

- Writable is an Interface in Hadoop.
- It acts as wrapper to primitive data type of java.
- All the MapReduce datatypes must implement writable Interface.

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance.**	Interface **supports multiple inheritance.**
3) Abstract class **can have final, non-final, static and non-static variables.**	Interface has **only static and final variables.**
4) Abstract class **can provide the implementation of interface.**	Interface **can't provide the implementation of abstract class.**
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Writable Class Hadoop Data Types :

Hadoop provides classes that wrap the Java primitive types and implement the *WritableComparable* and *Writable* Interfaces. They are provided in the **org.apache.hadoop.io** package.

Primitive Writable Classes

These are Writable Wrappers for Java primitive data types and they hold a single primitive value that can be set either at construction or via a setter method.

All these primitive writable wrappers have get() and set() methods to read or write the wrapped value. Below is the list of primitive writable data types available in Hadoop.

- BooleanWritable
- ByteWritable
- IntWritable
- FloatWritable
- LongWritable
- DoubleWritable

References

Big Data: What it is and why it matters | SAS

Difference Between Structured, Unstructured & Semi-Structured Data (astera.com)

Characteristics of Big Data: Types & 5V's | upGrad blog

Real Time Big Data Applications in Various Domains | Edureka

Hadoop Ecosystem | Hadoop for Big Data and Data Engineering (analyticsvidhya.com)

Hadoop - HDFS Operations - Tutorialspoint

What is HDFS? Apache Hadoop Distributed File System | IBM

Learn about Hadoop Distributed File System Management (eduonix.com)

HDFS Data Block - Learn the Internals of Big Data Hadoop - TechVidvan

How HDFS achieves Fault Tolerance? (with practical example) - DataFlair (data-flair.training)

Rack Awareness in Hadoop HDFS - An Introductory Guide - DataFlair (data-flair.training)

Hadoop - File Blocks and Replication Factor - GeeksforGeeks

What is MapReduce in Hadoop? Architecture | Example (guru99.com)

Difference between Abstract class and Interface - Javatpoint

Java Wrapper Classes (w3schools.com)