Introduction to Hadoop: Introducing hadoop, Why hadoop, Why not RDBMS, RDBMS Vs Hadoop, History of Hadoop, Hadoop overview, Use case of Hadoop, HDFS (Hadoop Distributed File System),Processing data with Hadoop, Managing resources and applications with Hadoop YARN(Yet Another Resource Negotiator). Introduction to Map Reduce Programming: Introduction, Mapper, Reducer, Combiner, Partitioner, Searching, Sorting, Compression.

Introducing Hadoop

- Hadoop is an open-source framework designed for distributed storage and processing of large datasets using clusters of computers.
- It divides large data into smaller parts and processes them across multiple computers at the same time.
- It is part of the Apache Software Foundation and is widely used for Big Data processing.

Need for Hadoop

Why can't we use traditional databases?

Big Data Problems: Companies like Google, Facebook, and Amazon generate terabytes of data every second.

Slow Processing: A single computer cannot handle such huge data efficiently.

Storage Limitations: Traditional databases have limits on how much data they can store.

Hadoop helps to **store and process** large-scale data **efficiently and quickly**.

Hadoop has 3 main components:

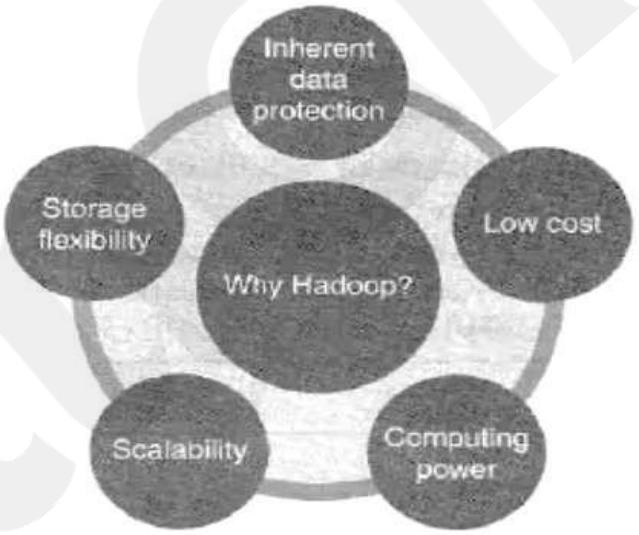
HDFS (Hadoop Distributed File System) – Like **Google Drive**, it stores data across multiple computers.
MapReduce – Like **group work**, it processes data in parallel and then combines results.
YARN (Yet Another Resource Negotiator) – Like a **manager**, it assigns tasks to different computers.

Why Hadoop?

The key consideration is:

Its capability to handle massive amounts of data, different categories of data - fairly quickly.

The other considerations are:



Low cost:

Hadoop is an open-source framework and uses commodity hardware (commodity hard- ware is relatively inexpensive and low-cost, widely available, general-purpose computers) to store enormous quantities of data.

Computing power:

Hadoop is based on distributed computing model which processes very large volumes of data fairly quickly. The more the number of computing nodes, the more the processing power at hand.

Scalability:

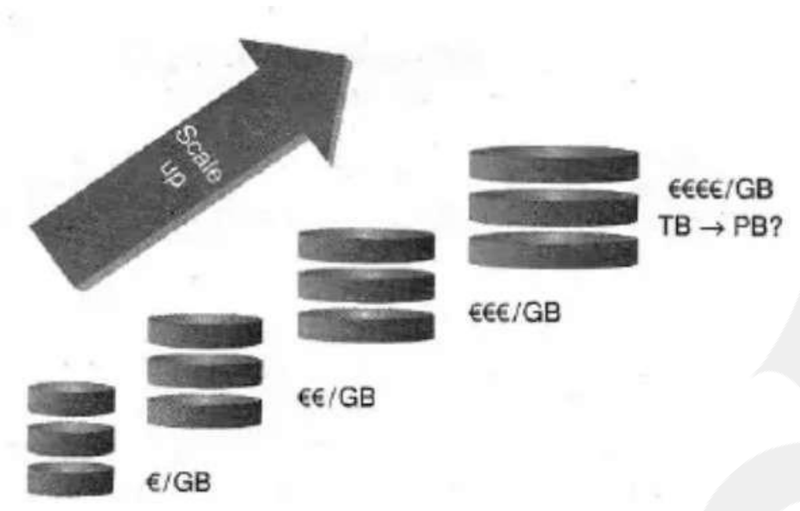
This boils down to simply adding nodes as the system grows and requires much less administration.

Storage flexibility: Unlike the traditional relational databases, in Hadoop data need not be pre-processed before storing it. Hadoop provides the convenience of storing as much data as one needs and also the added flexibility of deciding later as to how to use the stored data. In Hadoop, one can store unstructured data like images, videos, and free-form text.

Inherent data protection: Hadoop protects data and executing applications against hardware failure. If a node fails, it automatically redirects the jobs that had been assigned to this node to the other functional and available nodes and ensures that distributed computing does not fail. It goes a step further to store multiple copies (replicas) of the data on various nodes across the cluster.

Why not RDBMS?

- RDBMS is not suitable for storing and processing large files, images, and videos.
- RDBMS is not a good choice when it comes to advanced analytics involving machine learning.
- It calls for huge investment as the volume of data shows an upward trend.

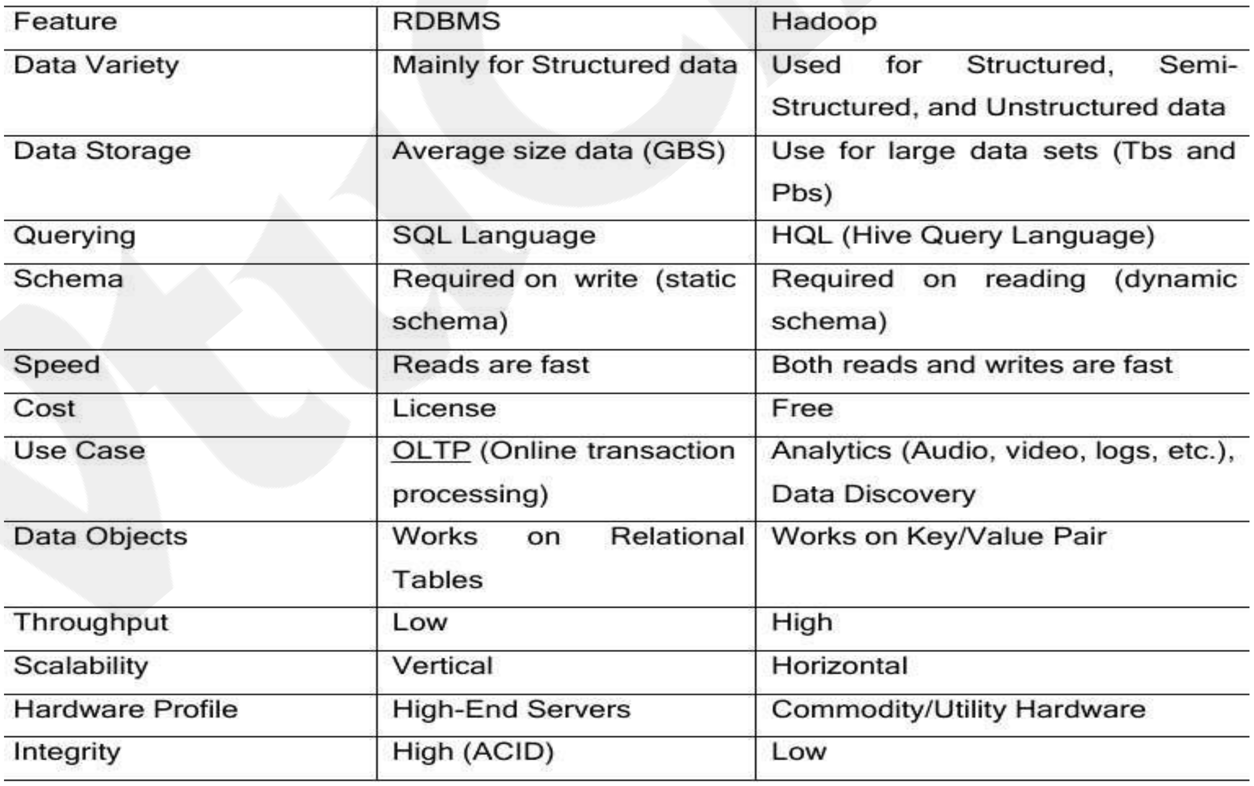


RDBMS vs Hadoop

PARAMETERS	RDBMS	HADOOP
System	Relational Database Management System.	Node Based Flat Structure.
Data	Suitable for structured data	Suitable for structured, unstructured data. Supports variety of data formats in real time such as XML, JSON, text based flat file formats, etc.
Processing	OLTP(Online Transaction Processing)	Analytical, Big Data Processing
Choice	When the data needs consistent	Big Data processing, which does not require any consistent

	relationship.	relationships between data.
Processor	Needs expensive hardware or high-end processors to store huge volumes of data.	In a Hadoop Cluster, a node requires only a processor, a network card, and few hard drives.
Cost	Cost around $10,000 to $14,000 per terabytes of storage.	Cost around $4,000 per terabytes of storage.

Other differences between RDBMS and Hadoop.



Feature	RDBMS	Hadoop
Data Variety	Mainly for Structured data	Used for Structured, Semi-Structured, and Unstructured data
Data Storage	Average size data (GBS)	Use for large data sets (Tbs and Pbs)
Querying	SQL Language	HQL (Hive Query Language)
Schema	Required on write (static schema)	Required on reading (dynamic schema)
Speed	Reads are fast	Both reads and writes are fast
Cost	License	Free
Use Case	OLTP (Online transaction processing)	Analytics (Audio, video, logs, etc.), Data Discovery
Data Objects	Works on Relational Tables	Works on Key/Value Pair
Throughput	Low	High
Scalability	Vertical	Horizontal
Hardware Profile	High-End Servers	Commodity/Utility Hardware
Integrity	High (ACID)	Low

When to choose between NoSQL and Hadoop?

Choose **Hadoop** for large-scale **batch processing, big data analytics, and cost-effective distributed storage** with HDFS.

NoSQL is better for **real-time transactions, flexible schema, and low-latency applications** like web apps and IoT.

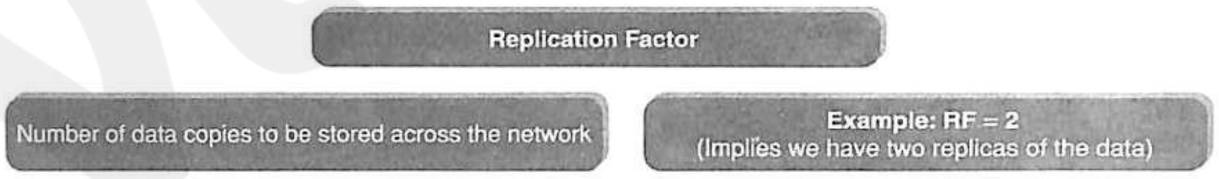
Distributed Computing Challenges:

Although there are several challenges with distributed computing, we will focus on two major challenges.

Hardware Failure

In a distributed system, several servers are networked together. This implies that more often than not, there may be a possibility of hardware failure. And when such a failure does happen, how does one retrieve the data that was stored in the system? Just to explain further - a regular hard disk may fail once in 3 years. And when you have 1000 such hard disks, there is a possibility of at least a few being down every day.

Hadoop has an answer to this problem in Replication Factor (RF). Replication Factor connotes the number of data copies of a given data item/data block stored across the network.

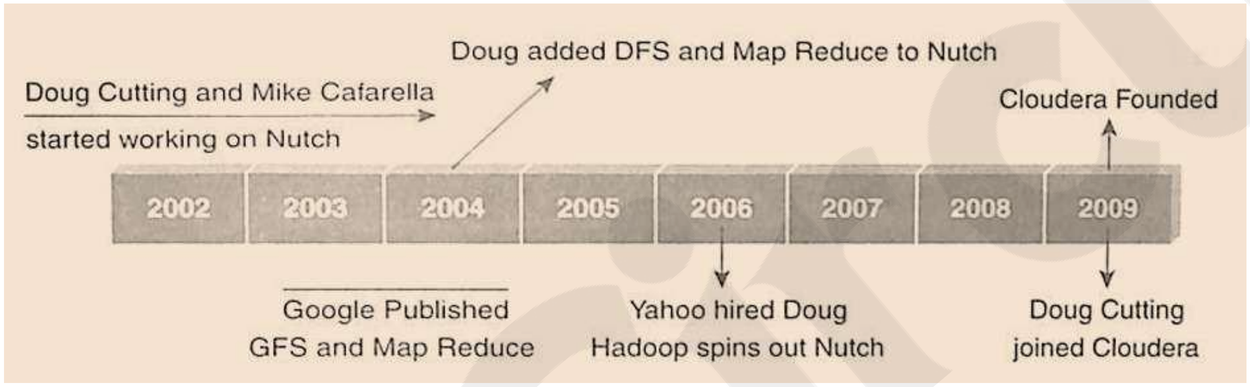

Replication Factor
Number of data copies to be stored across the network
Example: RF = 2
(Implies we have two replicas of the data)

How to Process This Gigantic Store of Data?

In a distributed system, the data is spread across the network on several machines. A key challenge here is to integrate the data available on several machines prior to processing it.

Hadoop solves this problem by using MapReduce Programming. It is a programming model to process the data (MapReduce programming will be discussed a little later).

History of Hadoop


Doug added DFS and Map Reduce to Nutch
Doug Cutting and Mike Cafarella started working on Nutch
Cloudera Founded
2002
2003
2004
2005
2006
2007
2008
2009
Google Published GFS and Map Reduce
Yahoo hired Doug Hadoop spins out Nutch
Doug Cutting joined Cloudera

The image is a timeline depicting key events in the history of Hadoop and related technologies. Here's a breakdown of the events shown:

1. **2002** - Doug Cutting and Mike Cafarella started working on **Nutch**, an open-source web crawler.
2. **2003** - The development of Nutch continued.
3. **2004** - Google published papers on **Google File System (GFS)** and **MapReduce**, which inspired further developments in distributed computing.
4. **2004-2005** - Doug Cutting added **Distributed File System (DFS)** and **MapReduce** to Nutch, enhancing its capabilities for large-scale data processing.
5. **2006** - Yahoo hired Doug Cutting, and **Hadoop was spun out from Nutch**, becoming a separate open-source project focused on distributed data storage and processing.

6. **2007-2008** - Hadoop continued to evolve and gain adoption.
7. **2009** - **Cloudera was founded**, a company focused on commercializing Hadoop solutions.
8. **2009** - Doug Cutting joined Cloudera, further contributing to the development and expansion of Hadoop-based technologies.

Hadoop Overview

Open-source software framework to store and process massive amounts of data in a distributed fashion on large clusters of commodity hardware. Basically, Hadoop accomplishes two tasks:

1. Massive data storage.

2. Faster data processing.

Key Aspects of Hadoop

Open source software: It is free to download, use and contribute to.

Framework: Means everything that you will need to develop and execute and application is provided - programs, tools, etc..

Distributed: Divides and stores data across multiple computers. Computation/Processing is done in parallel across multiple connected nodes.

Massive storage: Stores colossal amounts of data across nodes of low-cost commodity hardware.

Faster processing: Large amounts of data is processed in parallel, yielding quick response.

Hadoop components

Hadoop Core Components

1. HDFS:

(a) Storage component.

(b) Distributes data across several nodes.

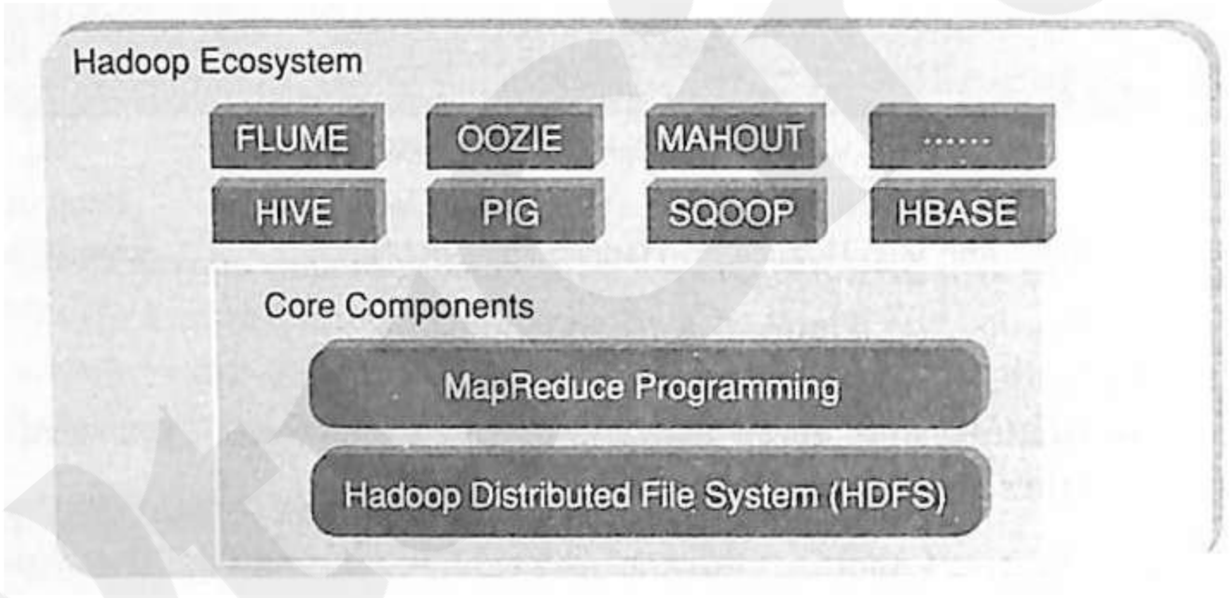
(c) Natively redundant.

2. MapReduce:

(a) Computational framework.

(b) Splits a task across multiple nodes.

(c) Processes data in parallel.



Hadoop Ecosystem: Hadoop Ecosystem are support projects to enhance the functionality of Hadoop Core Components. The Eco Projects are as follows:

1. HIVE

2. PIG

3. SQOOP

4. HBASE

5. FLUME

6. OOZIE

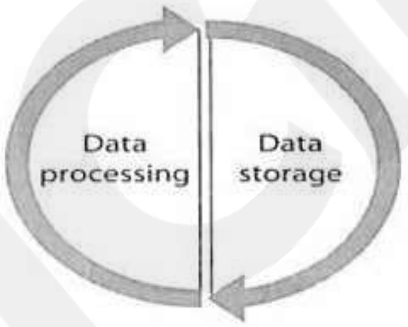
7. MAHOUT

Hadoop Conceptual Layer

It is conceptually divided into:

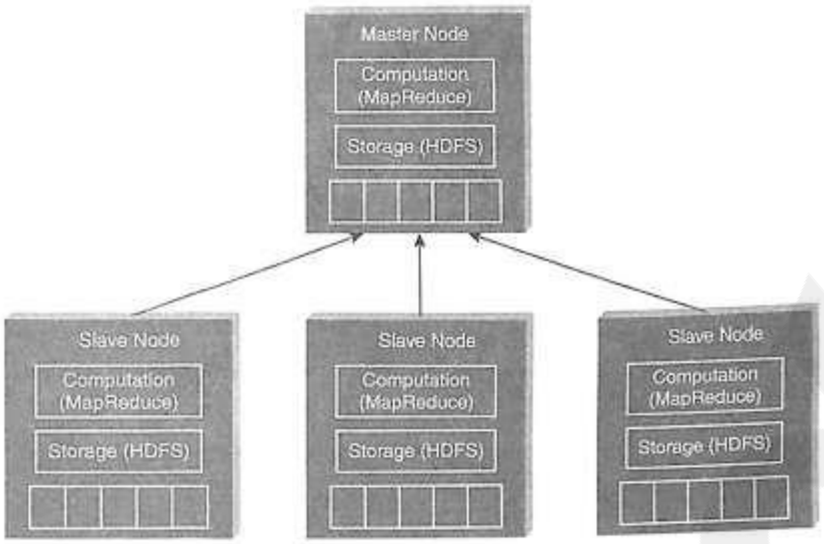
Data Storage Layer which stores huge volumes of data and

Data Processing Layer which processes data in parallel to extract richer and meaningful insights from data.



High-Level Architecture of Hadoop

- Hadoop is a distributed Master-Slave Architecture.
- Master node is known as **NameNode** and slave nodes are known as **DataNodes**.
- Figure depicts the Master-Slave Architecture of Hadoop Framework.



The key components of the Master Node are:

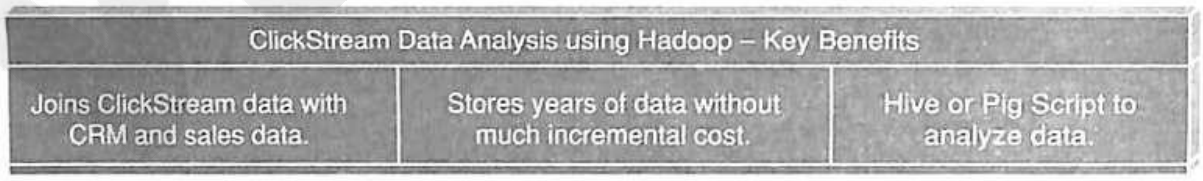
1. Master HDFS: Its main responsibility is partitioning the data storage across the slave nodes. It also keeps track of locations of data on DataNodes.

2. Master MapReduce: It decides and schedules computation task on slave nodes.

Use case of Hadoop

ClickStream Data

ClickStream data (mouse clicks) helps you to understand the purchasing behavior of customers. ClickStream analysis helps online marketers to optimize their product web pages, promotional content, etc. to improve their business.



ClickStream Data Analysis using Hadoop – Key Benefits		
Joins ClickStream data with CRM and sales data.	Stores years of data without much incremental cost.	Hive or Pig Script to analyze data.

The ClickStream analysis, as shown in above figure, using Hadoop provides three key benefits:

1. Hadoop helps to join ClickStream data with other data sources such as Customer Relationship Management Data (Customer Demographics Data, Sales Data, and Information on Advertising Campaigns). This additional data often provides the much needed information to understand cus- tomer behavior.

2. Hadoop's scalability property helps you to store years of data without ample incremental cost. This helps you to perform temporal or year over year analysis on ClickStream data which your competitors may miss.

3. Business analysts can use Apache Pig or Apache Hive for website analysis. With these tools, you can organize ClickStream data by user session, refine it, and feed it to visualization or analytics tools.

HDFS(Hadoop Distributed file system)

Some key Points of Hadoop Distributed File System are as follows:

1. Storage component of Hadoop.

2. Distributed File System.

3. Modeled after Google File System.

4. Optimized for high throughput (HDFS leverages large block size and moves computation where data is stored).

5. You can create multiple copies of a file as per configuration, ensuring reliability. This replication enhances fault tolerance for both software and hardware failures.

6. Re-replicates data blocks automatically on nodes that have failed.

7. You can realize the power of HDFS when you perform read or write on large files (gigabytes and larger).

8. It operates above native file systems like ext3 and ext4, as illustrated in the figure below. This abstraction enables additional functionality and flexibility in data management.

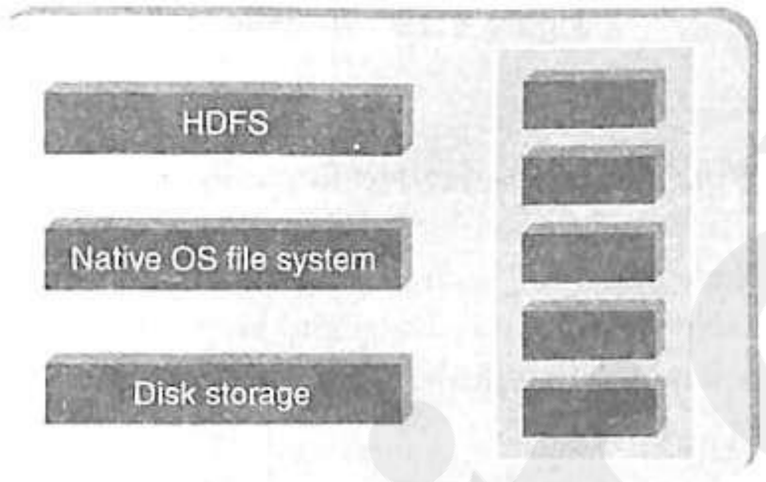
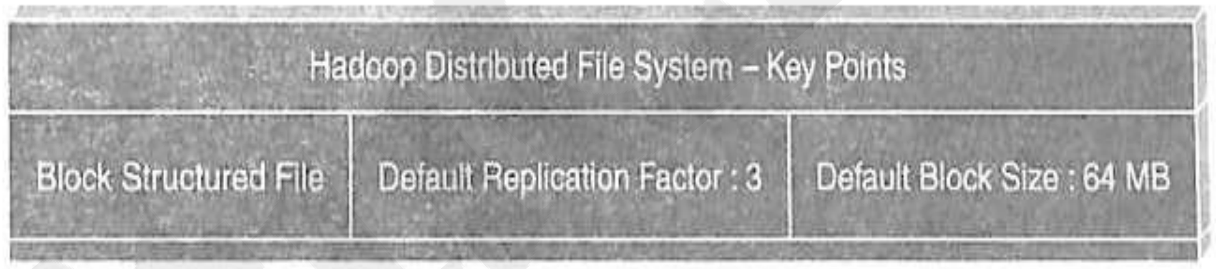


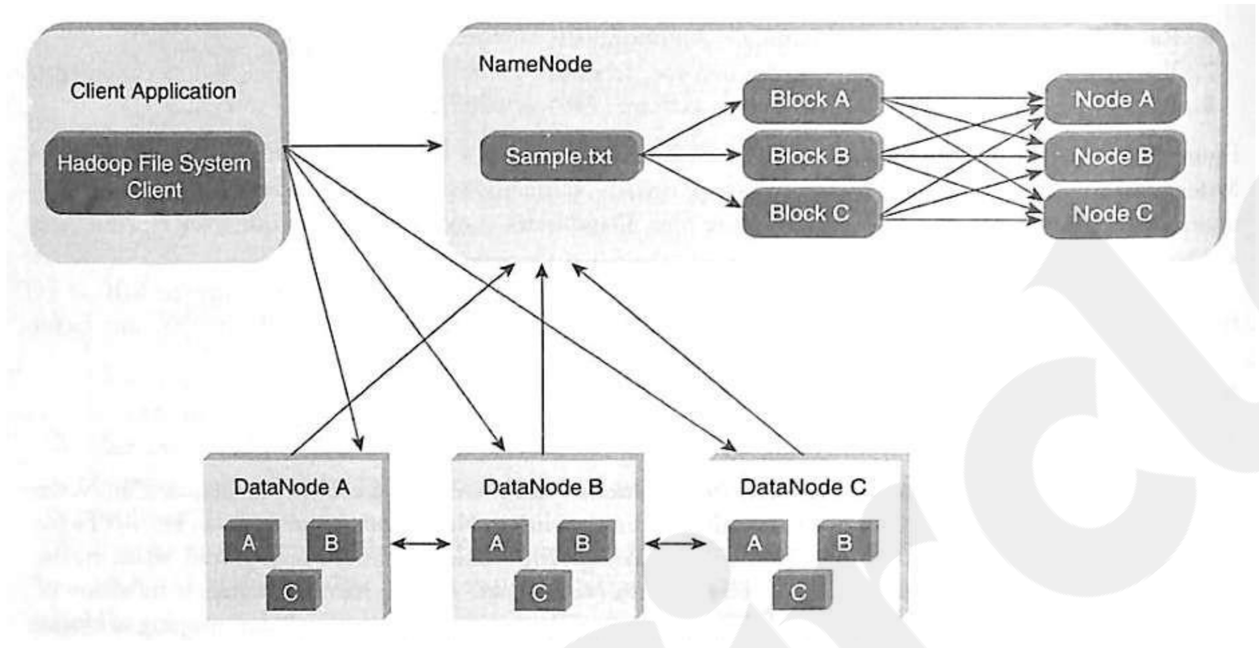
Figure below describes important key points of HDFS.



Hadoop Distributed File System – Key Points		
Block Structured File	Default Replication Factor : 3	Default Block Size : 64 MB

The figure highlights key aspects of the **Hadoop Distributed File System (HDFS)**. It mentions that HDFS uses a **block-structured file system**, has a **default replication factor of 3** for fault tolerance, and a **default block size of 64 MB** for efficient storage and processing.

Figure below describes Hadoop Distributed File System Architecture.



The figure illustrates the **Hadoop Distributed File System (HDFS) architecture**, showing how a file is stored and managed across multiple nodes.

Key Components:

Client Application

The client interacts with **HDFS** through the **Hadoop File System Client** to read/write data.

NameNode

Stores metadata about files (e.g., locations of blocks, filenames).

Manages file-to-block mapping (e.g., *Sample.txt* is split into **Block A, Block B, and Block C**).

Assigns blocks to different **DataNodes** for storage.

DataNodes

Actual storage locations for blocks.

Each block is **replicated** across multiple DataNodes (ensuring fault tolerance).

The figure shows **three DataNodes (A, B, and C)**, each storing different copies of **Blocks A, B, and C** based on the replication factor.

Working of HDFS:

A file is split into **fixed-size blocks** (e.g., **64MB or 128MB**).

These blocks are distributed across different **DataNodes** for redundancy.

The **NameNode** manages block locations but does not store actual data.

The **Hadoop File System Client** communicates with the **NameNode** to fetch block locations and then retrieves data from **DataNodes**.

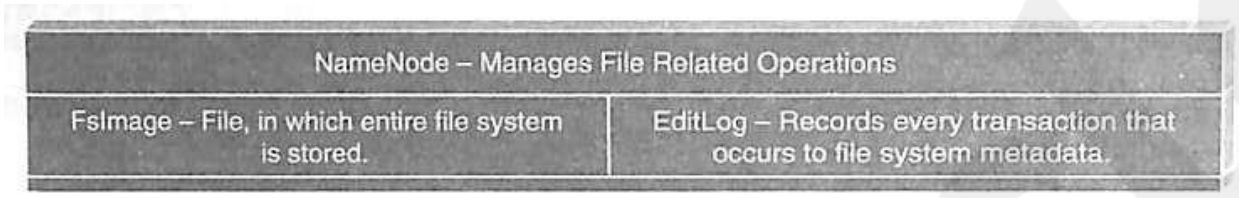
This design ensures **high availability, fault tolerance, and parallel processing** for large-scale data applications.

HDFS Daemons

NameNode

- HDFS breaks a large file into smaller pieces called blocks.
- NameNode uses a rack ID to identify DataNodes in the rack.
- A rack is a collection of DataNodes within the cluster.
- NameNode keeps tracks of blocks of a file as it is placed on various DataNodes. NameNode manages file-related operations such as read, write, create, and delete. Its main job is managing the File System Namespace.
- A file system namespace is collection of files in the cluster.

- NameNode stores HDFS namespace. File system namespace includes mapping of blocks to file, file properties and is stored in a file called FsImage.
- NameNode uses an EditLog (transaction log) to record every transaction that happens to the file system metadata. Refer Figure below.



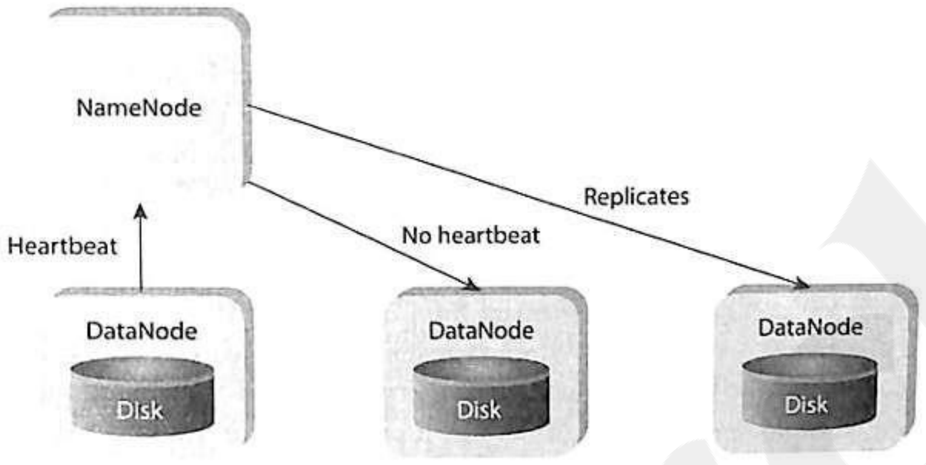
NameNode – Manages File Related Operations	
FsImage – File, in which entire file system is stored.	EditLog – Records every transaction that occurs to file system metadata.

- When NameNode starts up, it reads FsImage and EditLog from disk and applies all transactions from the EditLog to in-memory representation of the FsImage.
- Then it flushes out new version of FsImage on disk and truncates the old EditLog because the changes are updated in the FsImage.
- There is a single NameNode percluster.

DataNode

- There are multiple DataNodes per cluster.
- During Pipeline read and write DataNodes communicate with each other.
- A DataNode also continuously sends "heartbeat" message to NameNode to ensure the connectivity between the NameNode and DataNode.
- In case there is no heartbeat from a DataNode, the NameNode replicates that DataNode within the cluster and keeps on running as if nothing had happened.

- The figure illustrates the **HDFS NameNode and DataNode communication** using **heartbeats** for monitoring and data replication.



Heartbeat Mechanism:

Active **DataNodes send heartbeats** to the **NameNode** to indicate they are functioning.

If a **DataNode stops sending heartbeats (fails or disconnects)**, the NameNode assumes it is **unavailable.**

Data Replication:

When a DataNode fails (**no heartbeat detected**), the **NameNode triggers replication** of its data to another DataNode.

This ensures **fault tolerance and high availability** of data in HDFS.

The mechanism helps **detect node failures** and **automatically redistribute data**, ensuring Hadoop's **reliability and resilience.**

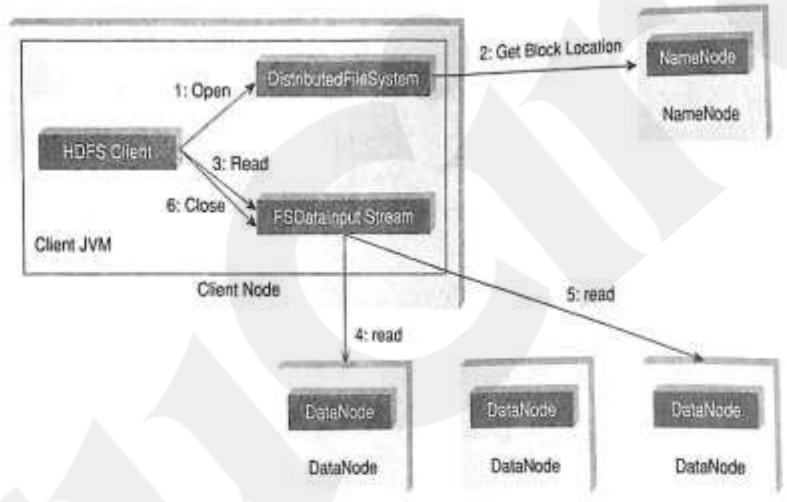
Secondary NameNode

- The Secondary NameNode takes a snapshot of HDFS metadata at intervals specified in the Hadoop configuration.

- Since the memory requirements of Secondary NameNode are the same as NameNode, it is better to run NameNode and Secondary NameNode on different machines.
- In case of failure of the NameNode, the Secondary NameNode can be configured manually to bring up the cluster. However, the Secondary NameNode does not record any real-time changes that happen to the HDFS metadata.

Anatomy of File read

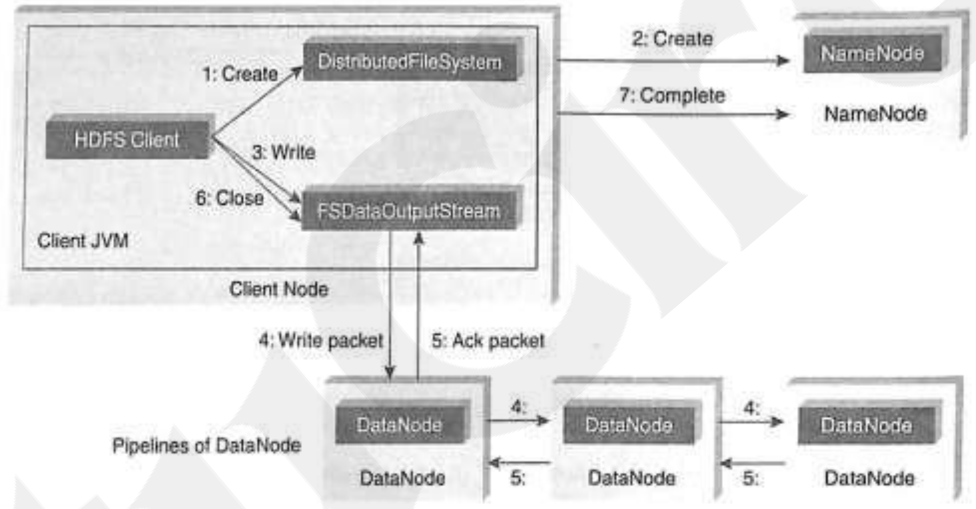
The figure below describes the anatomy of file read



1. The client opens the file that it wishes to read from by calling open() on the DistributedFileSystem.
2. DistributedFileSystem communicates with the NameNode to get the location of data blocks. NameNode returns with the addresses of the DataNodes that the data blocks are stored on. Subsequent to this, the DistributedFileSystem returns an FSDataInputStream to client to read from the file.

3. Client then calls read() on the stream DFSInputStream, which has addresses of the DataNodes for the first few blocks of the file, connects to the closest DataNode for the first block in the file.
4. Client calls read() repeatedly to stream the data from the DataNode.
5. When end of the block is reached, DFSInputStream closes the connection with the DataNode. It repeats the steps to find the best DataNode for the next block and subsequent blocks.
6. When the client completes the reading of the file, it calls close() on the FSDataInputStream to close the connection.

Anatomy of File write



The steps involved in anatomy of File Write are as follows:

1. The client calls create() on DistributedFileSystem to create a file.

2. An RPC(Remote Procedure Call) call to the NameNode happens through the DistributedFileSystem to create a new file. The NameNode performs various checks to create a new file (checks whether such a file exists or not). Initially, the NameNode creates a file without associating any data blocks to the file. The DistributedFileSystem returns an FSDataOutputStream to the client to perform write.

3. As the client writes data, data is split into packets by DFSOutputStream, which is then written to an internal queue, called data queue. DataStreamer consumes the data queue. The DataStreamer requests the NameNode to allocate new blocks by selecting a list of suitable DataNodes to store replicas. This list of DataNodes makes a pipeline. Here, we will go with the default replication factor of three, so there will be three nodes in the pipeline for the first block.

4. DataStreamer streams the packets to the first DataNode in the pipeline. It stores packet and forwards it to the second DataNode in the pipeline. In the same way, the second DataNode stores the packet and forwards it to the third DataNode in the pipeline.

5. In addition to the internal queue, DFSOutputStream also manages an "Ack queue" of packets that are waiting for the acknowledgement by DataNodes. A packet is removed from the "Ack queue" only if it is acknowledged by all the DataNodes in the pipeline.

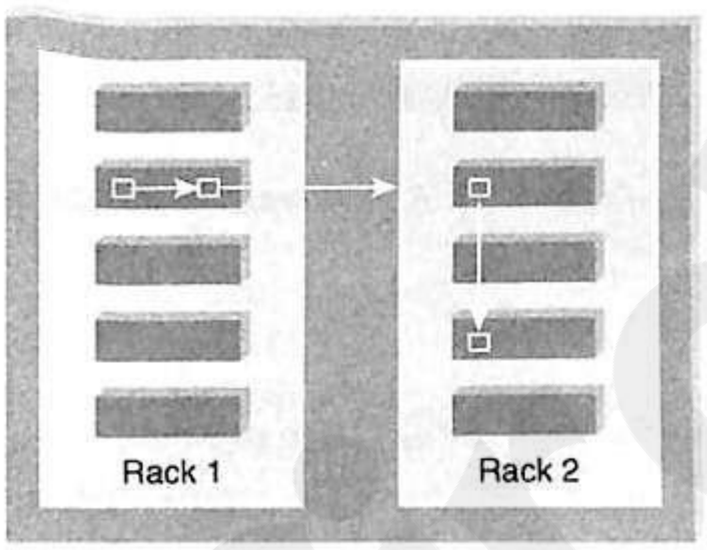
6. When the client finishes writing the file, it calls close() on the stream.

7. This flushes all the remaining packets to the DataNode pipeline and waits for relevant acknowledgments before communicating with the NameNode to inform the client that the creation of the file is complete.

Replica Placement Strategy

Hadoop Default Replica Placement Strategy:

As per the Hadoop Replica Placement Strategy, first replica is placed on the same node as the client. Then it places second replica on a node that is present on different rack. It places the third replica on the same rack as second, but on a different node in the rack. Once replica locations have been set, a pipeline is built. This strategy provides good reliability. Figure below describes the typical replica pipeline.



Working with HDFS Commands

Objective: To get the list of directories and files at the root of HDFS.

hadoop fs –ls /

The command **hadoop fs -ls /** is used in **Hadoop Distributed File System (HDFS)** to list the contents of the root directory (/).

hadoop fs → Interface to interact with the Hadoop filesystem.

-ls → Lists files and directories in the specified path.

/ → Refers to the root directory in HDFS.

Objective: To get the list of complete directories and files of HDFS.

hadoop fs –ls –R /

The command **hadoop fs -ls -R /** is used to recursively list all files and directories in **Hadoop Distributed File System (HDFS)** starting from the root

(/). This command is useful for searching files, checking storage structure, and debugging file locations in HDFS.

hadoop fs → Interface to interact with HDFS.

-ls → Lists files and directories in the specified path.

-R (Recursive) → Lists files and directories **recursively**, including all subdirectories and their contents.

/ → Specifies the root directory of HDFS.

Objective: To create a directory (say, sample) in HDFS.

hadoop fs –mkdir /sample

The command **hadoop fs -mkdir /sample** is used to **create a new directory** named **sample** in the **Hadoop Distributed File System (HDFS)** root (/). It organizes data in HDFS by creating folders, stores files in structured directories and Prepares directories for Hadoop jobs.

hadoop fs → Interface to interact with HDFS.

-mkdir → Creates a new directory.

/sample → Specifies the directory path to be created in HDFS.

Objective: To copy a file from local file system to HDFS.

hadoop fs -put /root/sample/test.txt /sample/test.txt

The command **hadoop fs -put /root/sample/test.txt /sample/test.txt** is used to **upload a file** from the local file system to the **Hadoop Distributed File System (HDFS)**.

hadoop fs → Interface to interact with HDFS.

-put → Copies a file from the local file system to HDFS.

/root/sample/test.txt → The **source file** located in the local file system.

/sample/test.txt → The **destination path** in HDFS where the file will be stored.

Use Cases:

Uploading data files to HDFS for **processing with Hadoop/Spark**.

Transferring log files, CSVs, or datasets for **big data analysis.**

Storing files in a distributed manner for **fault tolerance and scalability**.

Objective: To copy a file from HDFS to local file system.

hadoop fs -get /sample/test.txt /local/path/testsample.txt

hadoop fs → Interface to interact with HDFS.

-get → Downloads a file from **HDFS to the local file system**.

/sample/test.txt → **Source file** in HDFS.

/local/path/testsample.txt → **Destination path** in the local file system where the file will be stored.

Use Cases:

Retrieving processed results from **HDFS to local storage**.

Copying data files from **HDFS for analysis on a local machine**.

Exporting files for **backup or sharing**.

Objective: To copy a file from local file system to HDFS via copyFromLocal command.

hadoop fs -copyFromLocal /root/sample/test.txt /sample/testsample.txt

hadoop fs -copyFromLocal: Copies a local file to the Hadoop Distributed File System (HDFS).

/root/sample/test.txt: The **source file** (on the local file system).

/sample/testsample.txt: The **destination file** (on HDFS).

Objective: To copy a file from Hadoop file system to local file system via copyToLocal command.

hadoop fs -copyToLocal /sample/test.txt /root/sample/testsample1.txt

hadoop fs -copyToLocal: Copies a file **from HDFS to the local file system**.

/sample/test.txt: The **source file** (on HDFS).

/root/sample/testsample1.txt: The **destination file** (on the local system).

Objectives To display the contents of an HDFS file on console.

hadoop fs -cat /sample/test.txt

Displays the **contents** of /sample/test.txt (stored in HDFS) on the terminal.

Objective: To copy a file from one directory to another on HDFS.

hadoop fs -cp /sample/test.txt /sample1/

hadoop fs -cp → Copies a file **within HDFS**.

/sample/test.txt → Source file in HDFS.

/sample1/ → Destination directory in HDFS. The trailing / ensures it is treated as a directory.

Objective: To remove a directory from HDFS.

hadoop fs -rm -r /sample1

hadoop fs -rm → Removes files from HDFS.

-r → Recursively deletes a **directory and its contents**.

/sample1 → The target directory to be deleted.

Special Features of HDFS

1. **Data Replication:** There is absolutely no need for a client application to track all blocks. It directs the client to the nearest replica to ensure high performance.

2. **Data Pipeline:** A client application writes a block to the first DataNode in the pipeline. Then this DataNode takes over and forwards the data to the next node in the pipeline. This process continues for all the data blocks, and subsequently all the replicas are written to the disk.

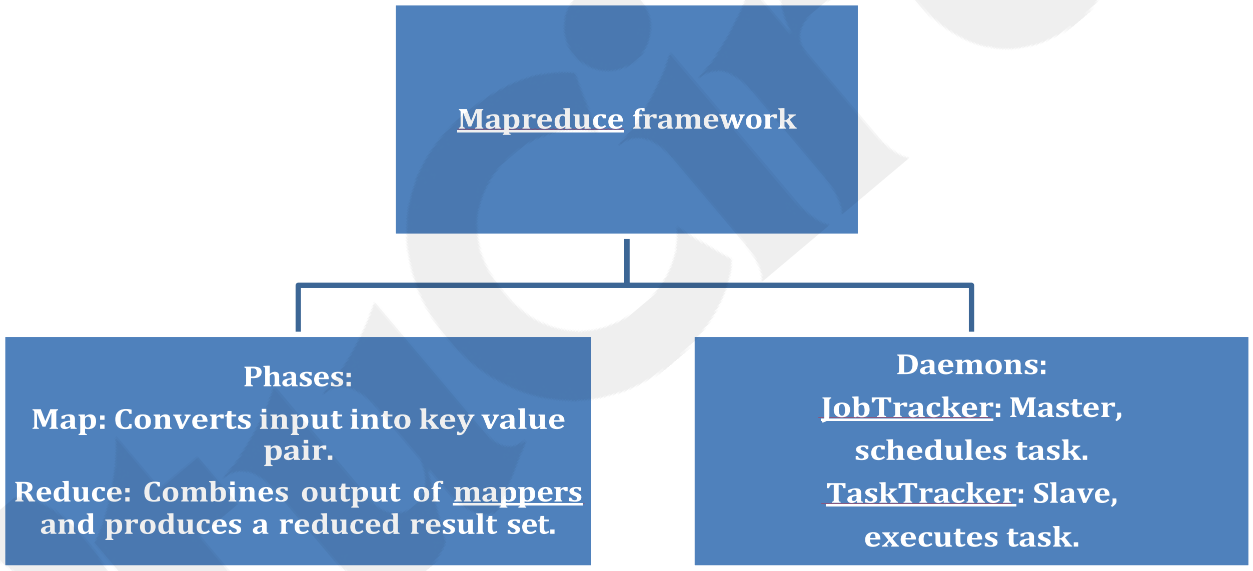
PROCESSING DATA WITH HADOOP

MapReduce Programming is a software framework. MapReduce Programming helps you to process massive amounts of data in parallel.

In MapReduce Programming, the input dataset is split into independent chunks. Map tasks process these independent chunks completely in a parallel manner. The output produced by the map tasks serves as intermediate data and is stored on the local disk of that server. The output of the mappers are automatically shuffled and sorted by the framework. MapReduce Framework sorts the output based on keys. This sorted output becomes the input to the reduce tasks. Reduce task provides reduced output by combining the out- put of the various mappers. Job inputs and outputs are stored in a file system. MapReduce framework also takes care of the other tasks such as scheduling, monitoring, re-executing failed tasks, etc.

Hadoop Distributed File System and MapReduce Framework run on the same set of nodes. This config- uration allows effective scheduling of tasks on the nodes where data is present (Data Locality). This in turn results in very high throughput.

There are two daemons associated with MapReduce Programming. A single master JobTracker per cluster and one slave TaskTracker per cluster-node. The JobTracker is responsible for scheduling tasks to the TaskTrackers, monitoring the task, and re-executing the task just in case the TaskTracker fails. The TaskTracker executes the task.

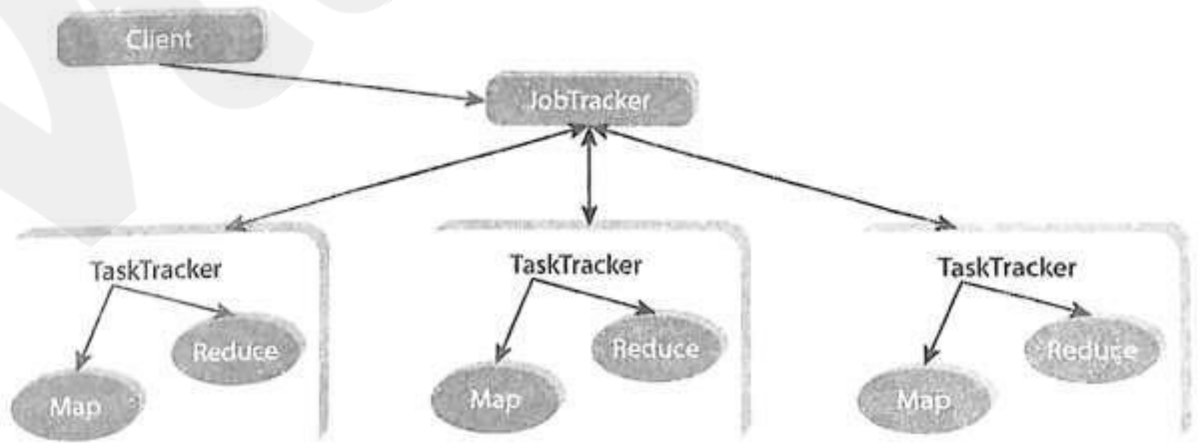

Mapreduce framework
Phases:
Map: Converts input into key value pair.
Reduce: Combines output of mappers and produces a reduced result set.
Daemons:
JobTracker: Master, schedules task.
TaskTracker: Slave, executes task.

The MapReduce functions and input/output locations are implemented via the MapReduce applications. These applications use suitable interfaces to construct the job. The application and the job parameters together are known as job configuration. Hadoop job client submits job (jar/executable, etc.) to the JobTracker. Then it is the responsibility of Job Tracker to schedule tasks to the slaves. In addition to scheduling, it also monitors the task and provides status information to the job-client.

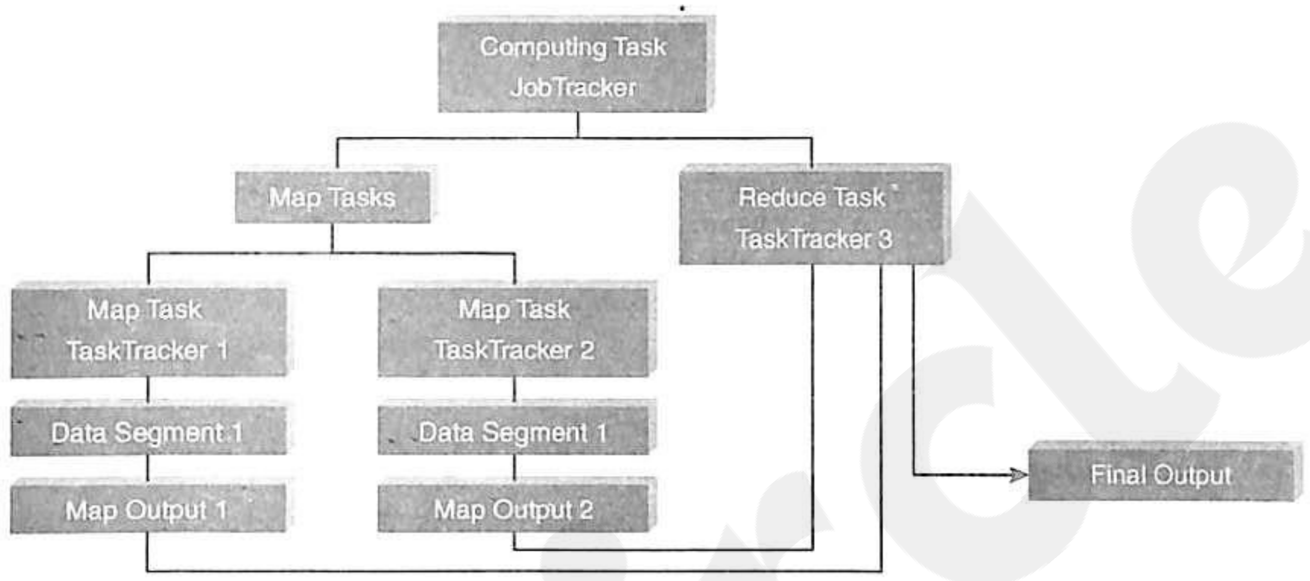
Mapreduce Daemons:

1. JobTracker: It provides connectivity between Hadoop and your application. When you submit code to cluster, Job Tracker creates the execution plan by deciding which task to assign to which node. It also monitors all the running tasks. When a task fails, it automatically re-schedules the task to a different node after a predefined number of retries. Job Tracker is a master daemon responsible for executing overall MapReduce job. There is a single Job Tracker per Hadoop cluster.

2. TaskTracker: This daemon is responsible for executing individual tasks that is assigned by the Job Tracker. There is a single TaskTracker per slave and spawns multiple Java Virtual Machines (JVMs) to handle multiple map or reduce tasks in parallel. TaskTracker continuously sends heartbeat message to JobTracker. When the Job Tracker fails to receive a heartbeat from a TaskTracker, the Job Tracker assumes that the TaskTracker has failed and resubmits the task to another available node in the cluster. Once the client submits a job to the Job Tracker, it partitions and assigns diverse MapReduce tasks for each Task Tracker in the cluster.



How does Mapreduce work?



The above figure shows how the mapreduce programming works which is fundamental to Hadoop's data processing framework. Here's a step-by-step explanation of the workflow illustrated in the figure:

1. **Computing Task (JobTracker)**:
   - The **JobTracker** is responsible for managing and scheduling jobs across the Hadoop cluster.
   - It assigns **Map Tasks** and **Reduce Tasks** to available **TaskTrackers** (worker nodes).
2. **Map Tasks Execution**:
   - The **JobTracker** assigns **Map Tasks** to different **TaskTrackers** (worker nodes).
   - Each **TaskTracker** processes a **data segment** (a portion of the input data).
   - The map function runs on these segments to generate intermediate results called **Map Outputs**.
3. **Reduce Task Execution**:

- The intermediate **Map Outputs** are sent to a **Reduce Task**, which runs on another **TaskTracker (TaskTracker 3)**.
- The **Reducer** aggregates, sorts, and processes the intermediate results to generate the final output.

4. **Final Output Generation**:
   - The processed data from the **Reduce Task** is then written to the final output location.

- **JobTracker** is the master node that assigns and monitors tasks.
- **TaskTrackers** execute individual **Map** and **Reduce tasks**.
- **Map tasks** process raw data and generate intermediate outputs.
- **Reduce task** processes the intermediate outputs and produces the final result.
- This model enables parallel processing, making it efficient for handling large-scale data.

This figure effectively represents the distributed nature of Hadoop's **MapReduce** framework, where tasks are divided and executed across multiple nodes to enhance performance and scalability.

MAPREDUCE PROGRAMMING ARCHITECTURE

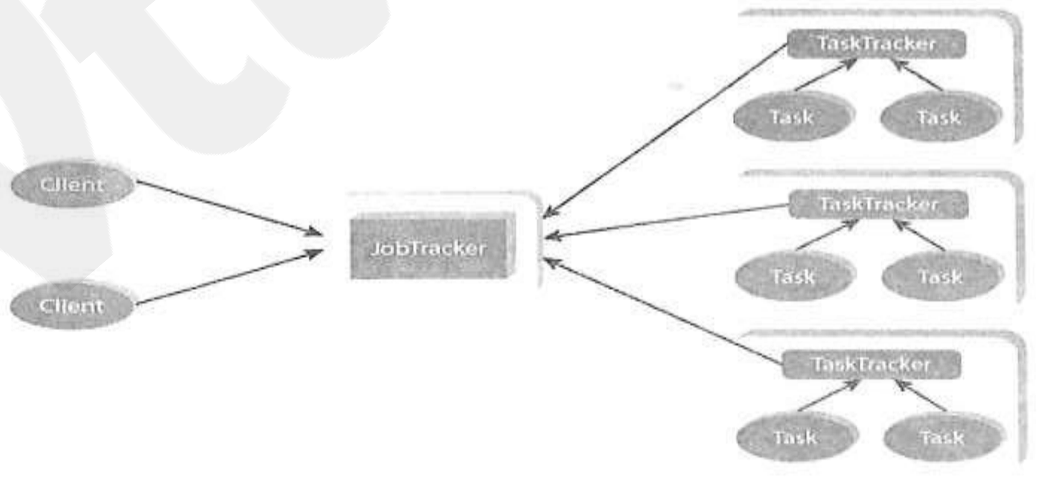


Figure above depicts Job the mapreduce programming architecture.

The following steps describe how MapReduce performs its task.

1. First, the input dataset is split into multiple pieces of data (several small subsets).

2. Next, the framework creates a master and several workers processes and executes the worker processes remotely.

3. Several map tasks work simultaneously and read pieces of data that were assigned to each map task. The map worker uses the map function to extract only those data that are present on their server and generates key/value pair for the extracted data.

4. Map worker uses partitioner function to divide the data into regions. Partitioner decides which reducer should get the output of the specified mapper.

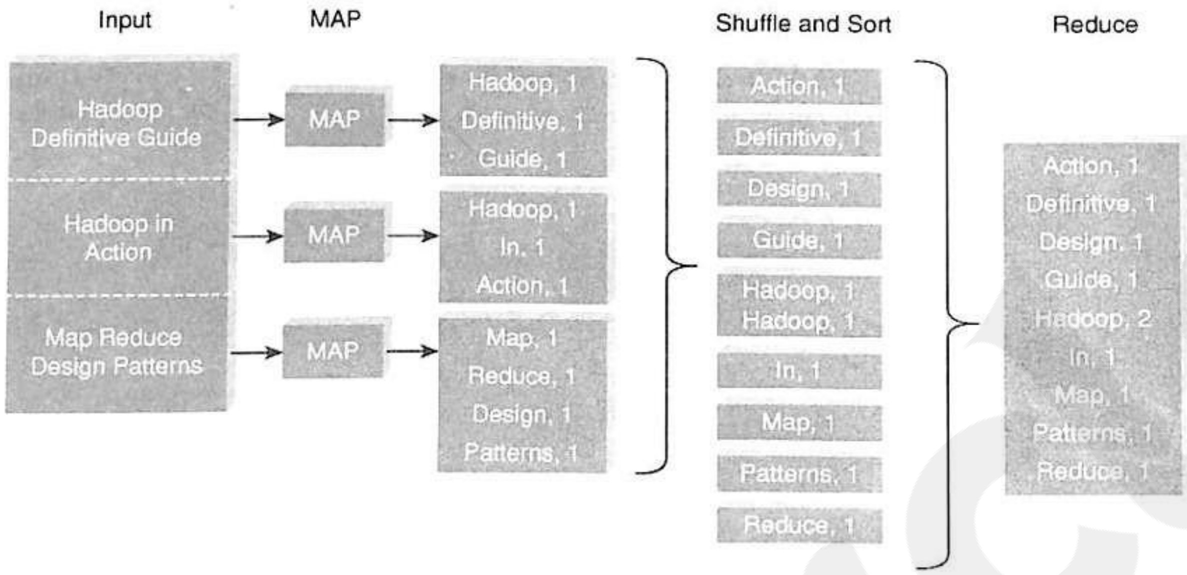
5. When the map workers complete their work, the master instructs the reduce workers to begin their work. The reduce workers in turn contact the map workers to get the key/value data for their partition. The data thus received is shuffled and sorted as per keys.

6. Then it calls reduce function for every unique key. This function writes the output to the file.

7. When all the reduce workers complete their work, the master transfers the control to the user program.

MapReduce Example

The famous example for MapReduce Programming is Word Count. For example, consider you need to count the occurrences of similar words across 50 files. You can achieve this using MapReduce Programming. Refer Figure.



Word Count MapReduce Programming using Java

The MapReduce Programming requires three things.

1. Driver Class: This class specifies Job Configuration details.

2. Mapper Class: This class overrides the Map Function based on the problem statement.

3. Reducer Class: This class overrides the Reduce Function based on the problem statement.

Wordcounter.java: Driver Program

```
package com.app;
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

```
public class WordCounter {
    public static void main(String[] args) throws IOException, InterruptedException,
ClassNotFoundException {
        // Create a new Job
        Job job = Job.getInstance();
        job.setJobName("wordcounter");

        // Set the Jar file containing this class
        job.setJarByClass(WordCounter.class);

        // Set Mapper and Reducer classes
        job.setMapperClass(WordCounterMap.class);
        job.setReducerClass(WordCounterRed.class);

        // Set output key-value types
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        // Set input and output paths
        FileInputFormat.addInputPath(job, new Path("/sample/word.txt"));
        FileOutputFormat.setOutputPath(job, new Path("/sample/wordcount"));

        // Execute the job and wait for completion
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Mapper Class: WordCounterMap.java

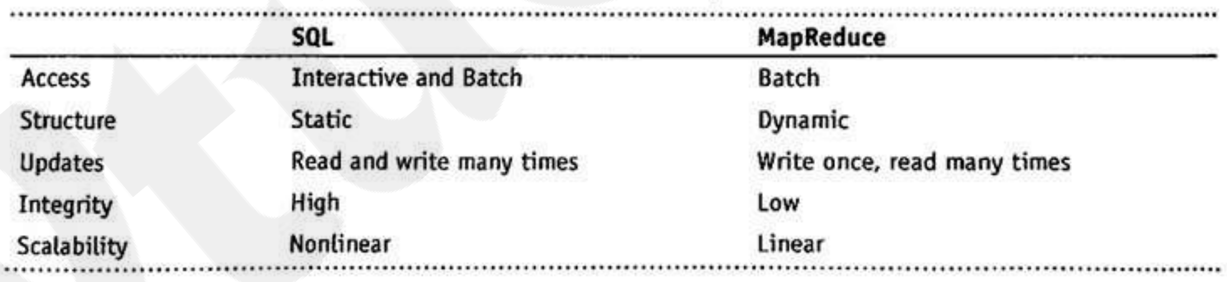
```
package com.app;
```

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

public class WordCounterMap extends Mapper<Object, Text, Text, IntWritable>
{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
        // Convert line to lowercase and split by non-word characters
        String[] words = value.toString().toLowerCase().split("\\W+");

          for (String w : words) {
              if (!w.isEmpty()) {
                  word.set(w);
                  context.write(word, one); // Emit (word, 1)
              }
          }
    }
}
```

Reducer Class: WordCounterRed.java

```
package com.app;

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

public class WordCounterRed extends Reducer<Text, IntWritable, Text,
IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException {
        int sum = 0;

        // Sum up all counts for each word
        for (IntWritable val : values) {
            sum += val.get();
        }

        // Emit (word, total_count)
        context.write(key, new IntWritable(sum));
    }
}
```

Difference between SQL and Mapreduce



	SQL	MapReduce
Access	Interactive and Batch	Batch
Structure	Static	Dynamic
Updates	Read and write many times	Write once, read many times
Integrity	High	Low
Scalability	Nonlinear	Linear

MANAGING RESOURCES AND APPLICATIONS WITH HADOOP YARN (YET ANOTHER RESOURCE NEGOTIATOR)

Apache Hadoop YARN is a sub-project of Hadoop 2.x. Hadoop 2.x is YARN-based architecture. It is a general processing platform. YARN is not constrained to MapReduce only. You can run multiple applications in Hadoop 2.x in which all applications share a common resource management. Now Hadoop can be used for various types of processing such as Batch, Interactive, Online, Streaming, Graph, and others.

Limitations of Hadoop 1.0 Architecture

In Hadoop 1.0, HDFS and MapReduce are Core Components, while other components are built around the core.

1. Single NameNode is responsible for managing entire namespace for Hadoop Cluster.
2. It has a restricted processing model which is suitable for batch-oriented MapReduce jobs.
3. Hadoop MapReduce is not suitable for interactive analysis.
4. Hadoop 1.0 is not suitable for machine learning algorithms, graphs, and other memory intensive algorithms.
5. MapReduce is responsible for cluster resource management and data processing.

In this Architecture, map slots might be "full", while the reduce slots are empty and vice versa. This causes resource utilization issues. This needs to be improved for proper resource utilization.

HDFS Limitation

NameNode saves all its file metadata in main memory. Although the main memory today is not as small and as expensive as it used to be two decades ago, still there is a limit on the number of objects that one can have in the memory on a single NameNode. The NameNode can quickly become overwhelmed with load on the system increasing.

In Hadoop 2.x, this is resolved with the help of HDFS Federation.

Hadoop 2: HDFS

HDFS 2 consists of two major components: (a) namespace, (b) blocks storage service. Namespace service takes care of file-related operations, such as creating files, modifying files, and directories. The block storage service handles data node cluster management, replication.

HDFS 2 Features

1. Horizontal scalability.
2. High availability.

HDFS Federation uses multiple independent NameNodes for horizontal scalability. NameNodes are independent of each other. It means, NameNodes does not need any coordination with each other. The DataNodes are common storage for blocks and shared by all NameNodes. All DataNodes in the cluster registers with each NameNode in the cluster.

High availability of NameNode is obtained with the help of Passive Standby NameNode. In Hadoop 2.x, Active-Passive NameNode handles failover automatically. All namespace edits are recorded to a shared NFS storage and there is a single writer at any point of time. Passive NameNode reads edits from shared storage and keeps updated metadata information. In case of Active NameNode failure, Passive NameNode becomes an Active NameNode automatically. Then it starts writing to the shared storage. Figure below describes the Active-Passive NameNode interaction.

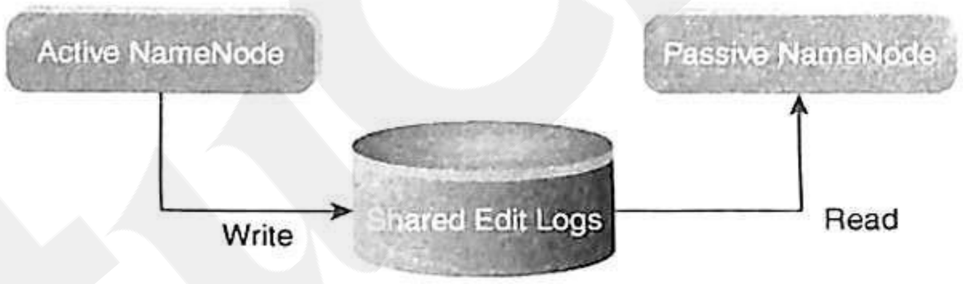
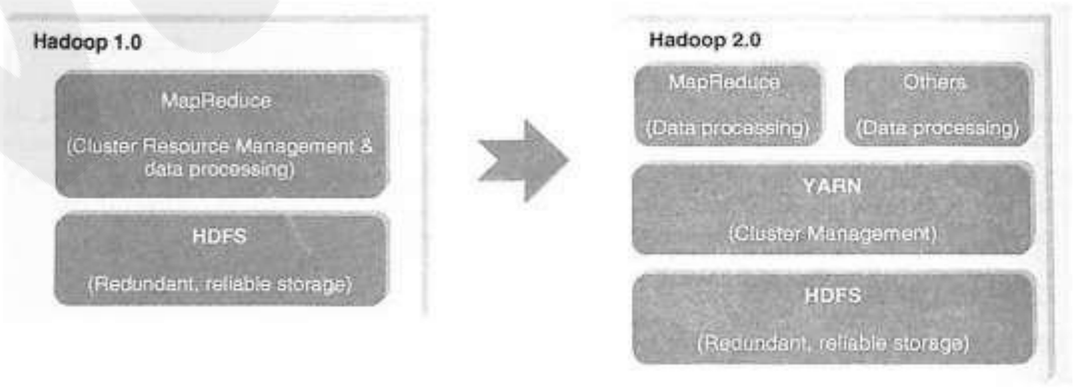
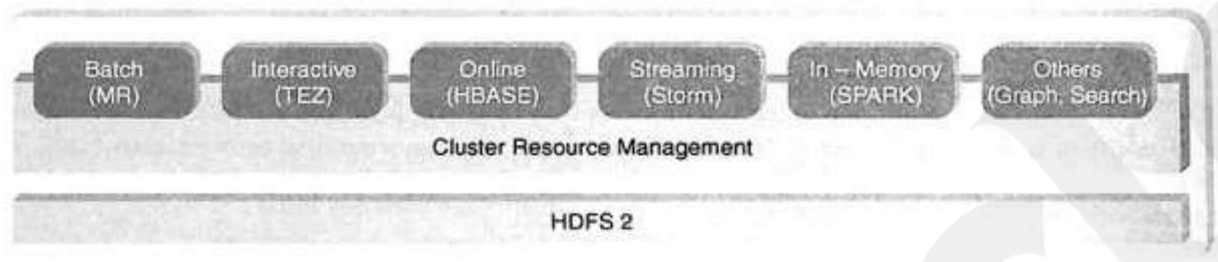


Figure below depicts Hadoop 1.0 and Hadoop 2.0 architecture.



Hadoop 2 YARN: Taking Hadoop beyond Batch

YARN helps us to store all data in one place. We can interact in multiple ways to get predictable performance and quality of services. This was originally architected by Yahoo. Refer Figure below.



Fundamental Idea

The fundamental idea behind this architecture is splitting the Job Tracker responsibility of resource management and Job Scheduling/Monitoring into separate daemons. Daemons that are part of YARN Architecture are described below.

1. A Global ResourceManager: Its main responsibility is to distribute resources among various applica- tions in the system. It has two main components:

(a) Scheduler: The pluggable scheduler of ResourceManager decides allocation of resources to var- ious running applications. The scheduler is just that, a pure scheduler, meaning it does NOT monitor or track the status of the application.

(b) ApplicationManager: ApplicationManager does the following:

• Accepting job submissions.

• Negotiating resources (container) for executing the application specific ApplicationMaster. Restarting the ApplicationMaster in case of failure.

2. NodeManager: This is a per-machine slave daemon. NodeManager responsibility is launching the application containers for application execution. NodeManager monitors the resource usage such as memory, CPU, disk, network, etc. It then reports the usage of resources to the global ResourceManager.

3. Per-application ApplicationMaster: This is an application-specific entity. Its responsibility is to negotiate required resources for execution from the ResourceManager. It works along with the NodeManager for executing and monitoring component tasks.

Basic Concepts

Application:

1. Application is a job submitted to the framework.
2. Example - MapReduce Job.

Container:

1. Basic unit of allocation.
2. Fine-grained resource allocation across multiple resource types (Memory, CPU, disk, network, etc.)

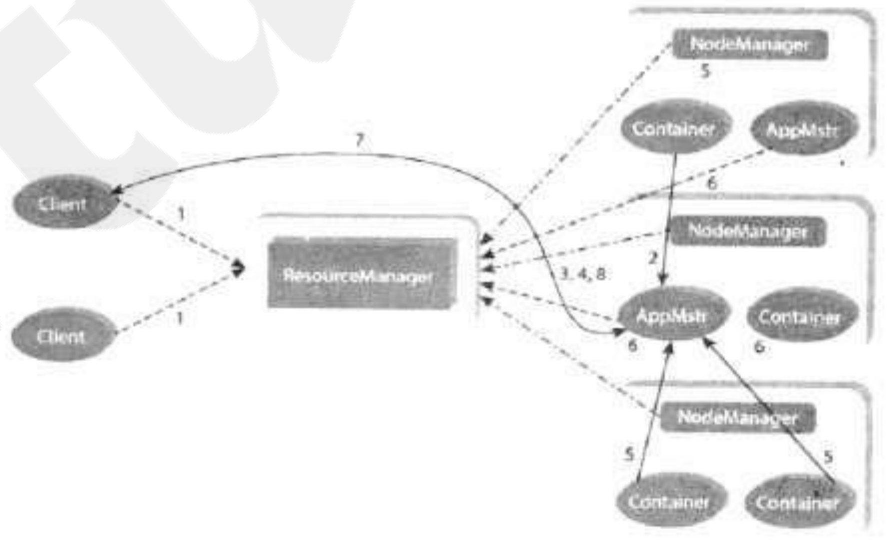
(a) container_0 = 2GB, ICPU

(b) container_1 = 1GB, 6 CPU

3. Replaces the fixed map/reduce slots.

YARN ARCHITECTURE

The figure below shows YARN architecture.



The steps involved in YARN architecture is as shown below:

1. A client program submits the application which includes the necessary specifications to launch the application-specific ApplicationMaster itself.
2. The ResourceManager launches the ApplicationMaster by assigning some container.
3. The ApplicationMaster, on boot-up, registers with the ResourceManager. This helps the client program to query the ResourceManager directly for the details.
4. During the normal course, ApplicationMaster negotiates appropriate resource containers via the resource-request protocol.
5. On successful container allocations, the ApplicationMaster launches the container by providing the container launch specification to the NodeManager.
6. The NodeManager executes the application code and provides necessary information such as progress, status, etc. to it's ApplicationMaster via an application-specific protocol.
7. During the application execution, the client that submitted the job directly communicates with the ApplicationMaster to get status, progress updates, etc. via an application-specific protocol.
8. Once the application has been processed completely, ApplicationMaster deregisters with the ResourceManager and shuts down, allowing its own container to be repurposed.

INTRODUCTION TO MAPREDUCE PROGRAMMING

Introduction

In MapReduce Programming, Jobs (Applications) are split into a set of map tasks and reduce tasks. Then these tasks are executed in a distributed fashion on Hadoop cluster. Each task processes small subset of data that has been assigned to it. This way, Hadoop distributes the load across the cluster. MapReduce job takes a set of files that is stored in HDFS (Hadoop Distributed File System) as input.

Map task takes care of loading, parsing, transforming, and filtering. The responsibility of reduce task is grouping and aggregating data that is produced by map tasks to generate final output. Each map task is broken into the following phases:

1. RecordReader.
2. Mapper.
3. Combiner.
4. Partitioner.

The output produced by map task is known as intermediate keys and values. These intermediate keys and values are sent to reducer. The reduce tasks are broken into the following phases:

1. Shuffle.
2. Sort.
3. Reducer.
4. Output Format.

Hadoop assigns map tasks to the DataNode where the actual data to be processed resides. This ensures data locality. Data locality means that data is not moved over network; only computational code is moved to process data which saves network bandwidth.

Mapper

A mapper maps the input key-value pairs into a set of intermediate key-value pairs. Maps are individual tasks that have the responsibility of transforming input records into intermediate key-value pairs.

1. RecordReader: RecordReader converts a byte-oriented view of the input (as generated by the Input- Split) into a record-oriented view and presents it to the Mapper tasks. It presents the tasks with keys and values. Generally the key is the positional information and value is a chunk of data that constitutes the record.

2. Map: Map function works on the key-value pair produced by RecordReader and generates zero or more intermediate key-value pairs. The MapReduce decides the key-value pair based on the context.

3. Combiner: It is an optional function but provides high performance in terms of network bandwidth and disk space. It takes intermediate key-value pair provided by mapper and applies user-specific aggregate function to only that mapper. It is also known as local reducer.

4. Partitioner: The partitioner takes the intermediate key-value pairs produced by the mapper, splits them into shard, and sends the shard to the particular reducer as per the user-specific code. Usually, the key with same values goes to the same reducer. The partitioned data of each map task is written to the local disk of that machine and pulled by the respective reducer.

Reducer

The primary chore of the Reducer is to reduce a set of intermediate values (the ones that share a common key) to a smaller set of values. The Reducer has three primary phases: Shuffle and Sort, Reduce, and Output Format.

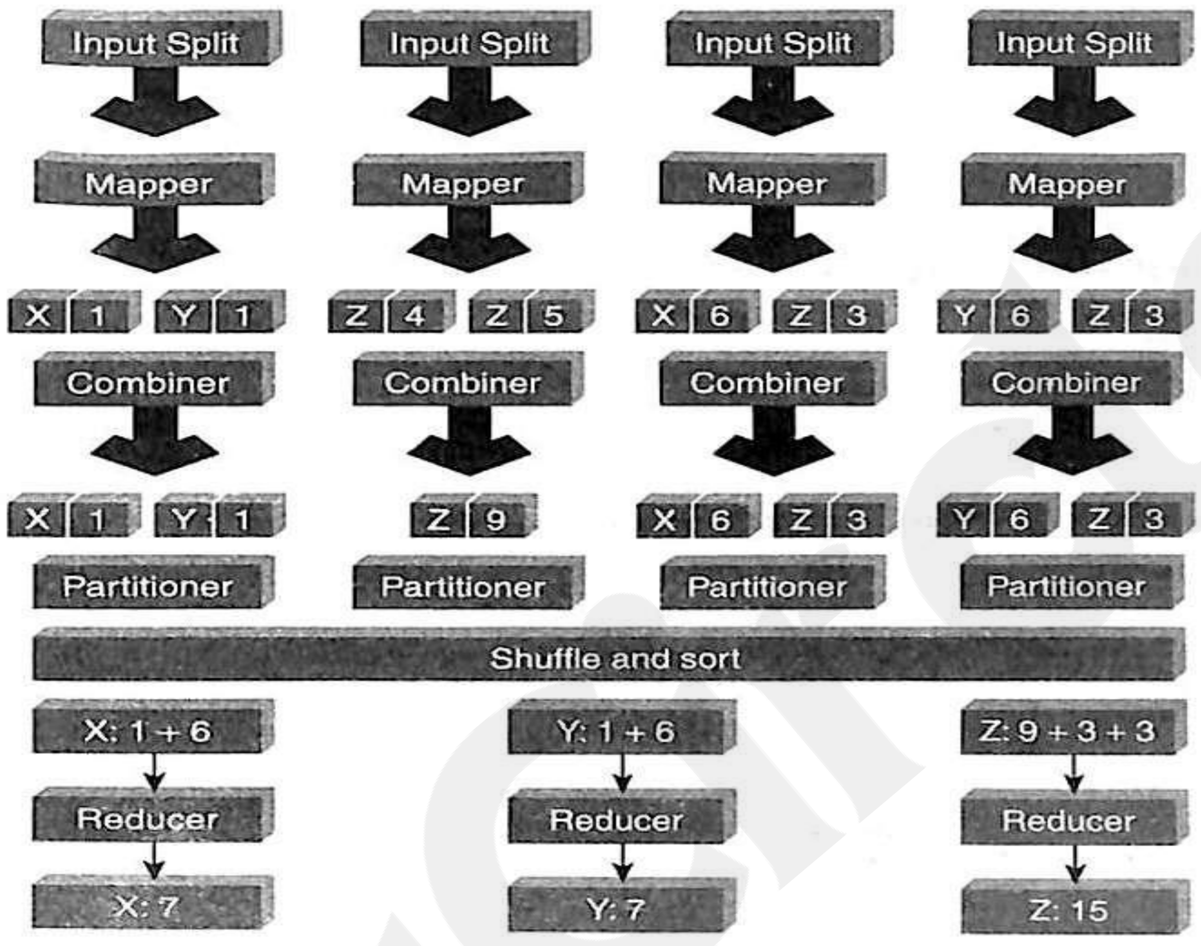
1. Shuffle and Sort: This phase takes the output of all the partitioners and downloads them into the local machine where the reducer is running. Then these individual data pipes are sorted by keys which produce larger data list. The main purpose of this sort is grouping similar words so that their values can be easily iterated over by the reduce task.

2. Reduce: The reducer takes the grouped data produced by the shuffle and sort phase, applies reduce function, and processes one group at a time. The reduce function iterates all the values associated with that key. Reducer function provides various operations such as aggregation, filtering, and com- bining data. Once it is done, the output (zero or more key-value pairs) of reducer is sent to the output format.

3. Output Format: The output format separates key-value pair with tab (default) and writes it out to a file using record writer.

Figure describes the chores of Mapper, Combiner, Partitioner, and Reducer for the word count problem.

The Word Count problem has been discussed under "Combiner" and "Partitioner".



Combiner

It is an optimization technique for MapReduce Job. Generally, the reducer class is set to be the combiner class. The difference between combiner class and reducer class is as follows:

1. Output generated by combiner is intermediate data and it is passed to the reducer.
2. Output of the reducer is passed to the output file on disk.

The sections have been designed as follows:

Objective: What is it that we are trying to achieve here?

Input Data: What is the input that has been given to us to act upon?

Act: Output:

The actual statement/command to accomplish the task at hand. The result/output as a consequence of executing the statement.

Objective

To write a **MapReduce** program that counts the occurrences of words in a given text file and uses a **Combiner** for optimization.

1. Input Data

The input file contains sentences such as:

Welcome to Hadoop Session

Introduction to Hadoop

Introducing Hive

Hive Session

Pig Session

2. MapReduce Execution Flow

A. Mapper Phase

The **Mapper** reads the input file and splits it into words.It emits key-value pairs in the form **(word, 1)**.

Example output from Mapper:

(Hadoop, 1)

(Session, 1)

(Introduction, 1)

(Hadoop, 1)

(Introducing, 1)

(Hive, 1)

(Hive, 1)

(Session, 1)

(Pig, 1)

(Session, 1)

B. Combiner Phase (Optimization)

The **Combiner** runs after the Mapper to **locally aggregate** word counts before passing data to the Reducer.

This reduces the amount of data transferred across the network.

Example output from Combiner:

(Hadoop, 2)

(Session, 3)

(Hive, 2)

(Pig, 1)

(Introduction, 1)

(Introducing, 1)

C. Shuffle and Sort Phase

The framework groups all occurrences of the same word together.

D. Reducer Phase

The **Reducer** receives the grouped data and sums up the counts.

Example output from Reducer:

Hadoop 2

Hive 2

Introducing 1

Introduction 1

Pig 1

Session 3

3. Driver Program Implementation

The **Combiner class is set** in the driver program:

job.setCombinerClass(WordCounterRed.class);

The **input and output paths** are configured:

FileInputFormat.addInputPath(job, new Path("/mapreducedemos/lines.txt"));
FileOutputFormat.setOutputPath(job,newPath("/mapreducedemos/output/wordcount/"));

The MapReduce job is executed using:

hadoop jar <jar_name> <driver_class> <input_path> <output_path>

4. Output Storage

The reducer output is stored in **part-r-00000** under the output directory:

/mapreducedemos/output/wordcount/part-r-00000

Partitioner

- The partitioning phase happens **after the map phase and before the reduce phase**.
- The **number of partitions equals the number of reducers**.
- The **default partitioner** in Hadoop is the **hash partitioner**, but custom partitioners can be implemented.

Objective of the Exercise

- Implement a **MapReduce program** to count word occurrences.
- Use a **custom partitioner** to divide words based on their starting alphabet.
- This ensures that words beginning with the same letter are sent to the same reducer.

Input Data Example

- Welcome to Hadoop Session
- Introduction to Hadoop
- Introducing Hive
- Hive Session
- Pig Session

Implementation of the Custom Partitioner (WordCountPartitioner.java)

- Extends Partitioner<Text, IntWritable>.
- Extracts the **first letter** of each word.
- Assigns a **partition number** based on this letter.
- Uses a switch-case structure for letters A–Z, with a **default partition for non-alphabet characters.**

```
import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import  org.apache.hadoop.mapreduce.Partitioner;


public class WordCountPartitioner extends Partitioner<Text, IntWritable>
{

    @Override

    public int getPartition(Text key, IntWritable value, int numPartitions) {

        String word = key.toString();

        char alphabet = word.toUpperCase().charAt(0);

        int partitionNumber = 0;


        switch  (alphabet)  {

            case 'A': partitionNumber = 1; break;

            case 'B': partitionNumber = 2; break;

            case 'C': partitionNumber = 3; break;

            case 'D': partitionNumber = 4; break;

            case 'E': partitionNumber = 5; break;

            case 'F': partitionNumber = 6; break;

            case 'G': partitionNumber = 7; break;

            case 'H': partitionNumber = 8; break;

            case 'I': partitionNumber = 9; break;

            case 'J': partitionNumber = 10; break;

            case 'K': partitionNumber = 11; break;
```

```
                case 'L': partitionNumber = 12; break;
                case 'M': partitionNumber = 13; break;
                case 'N': partitionNumber = 14; break;
                case 'O': partitionNumber = 15; break;
                case 'P': partitionNumber = 16; break;
                case 'Q': partitionNumber = 17; break;
                case 'R': partitionNumber = 18; break;
                case 'S': partitionNumber = 19; break;
                case 'T': partitionNumber = 20; break;
                case 'U': partitionNumber = 21; break;
                case 'V': partitionNumber = 22; break;
                case 'W': partitionNumber = 23; break;
                case 'X': partitionNumber = 24; break;
                case 'Y': partitionNumber = 25; break;
                case 'Z': partitionNumber = 26; break;
                default: partitionNumber = 0; // Non-alphabet words
            }

            return partitionNumber;
        }
    }
```

Sets **27 reducers** (one for each letter A-Z and a default).

Uses setPartitionerClass to specify the custom partitioner.

job.setNumReduceTasks(27);

job.setPartitionerClass(WordCountPartitioner.class);

// Input and Output Paths

FileInputFormat.addInputPath(job, new Path("/mapreducedemos/lines.txt"));

FileOutputFormat.setOutputPath(job,newPath("/mapreducedemos/output/wordcou ntpartitioner/"));

- The output directory contains **27 partitioned files**.
- Each partition file corresponds to words starting with a specific letter.
- Example: part-r-00008 corresponds to words starting with **'H'**.

Each line is broken into words:

- Welcome
- to
- Hadoop
- Session
- Introduction
- to
- Hadoop
- Introducing
- Hive
- Hive
- Session
- Pig
- Session

After mapping, each word gets a count of 1:(Words emitted by the mapper)

- Welcome → 1

- to → 1
- Hadoop → 1
- Session → 1
- Introduction → 1
- to → 1
- Hadoop → 1
- Introducing → 1
- Hive → 1
- Hive → 1
- Session → 1
- Pig → 1
- Session → 1

Each word is assigned to a reducer based on its first letter.

Word	First Letter	Partition Number
Welcome	W	23
to	T	20
Hadoop	H	8
Session	S	19
Introduction	I	9
Introducing	I	9
Hive	H	8
Pig	P	16

Each reducer generates an output file with words starting with a specific letter.

File: part-r-00008 (Reducer for 'H')

```
Hadoop    2

Hive      2
```

File: part-r-00009 (Reducer for 'I')

```
Introduction    1

Introducing     1
```

File: part-r-00016 (Reducer for 'P')

```
Pig    1
```

File: part-r-00019 (Reducer for 'S')

```
Session    3
```

File: part-r-00020 (Reducer for 'T')

```
to    2
```

File: part-r-00023 (Reducer for 'W')

```
Welcome    1
```

Final Output Directory

```
/mapreducedemos/output/wordcountpartitioner/

  ├── part-r-00008  (Words starting with H)

  ├── part-r-00009  (Words starting with I)

  ├── part-r-00016  (Words starting with P)

  ├── part-r-00019  (Words starting with S)

  ├── part-r-00020  (Words starting with T)

  ├── part-r-00023  (Words starting with W)
```

Searching

Objective

Implement a **MapReduce program** to **search for a specific keyword** in a file.

The program will output lines containing the keyword along with **file name and position.**

Input Data (student.csv)

1001,John,45

1002,Jack,39

1003,Alex,44

1004,Smith,38

1005,Bob,33

Each row contains **ID, Name, and Score**.

The **keyword to search is "Jack"**.

The MapReduce job consists of **three components:**

Driver Program (WordSearcher.java)

Mapper Class (WordSearchMapper.java)

Reducer Class (WordSearchReducer.java)

(a) Driver Program (WordSearcher.java)

This **configures the Hadoop job** and specifies the Mapper, Reducer, and input/output paths.

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;
import  org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import  org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordSearcher {

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

        Configuration conf = new Configuration();

        Job job = new Job(conf, "Word Search");

        job.setJarByClass(WordSearcher.class);

        job.setMapperClass(WordSearchMapper.class);

        job.setReducerClass(WordSearchReducer.class);

        job.setOutputKeyClass(Text.class);

        job.setOutputValueClass(Text.class);

        job.setInputFormatClass(TextInputFormat.class);

        job.setOutputFormatClass(TextOutputFormat.class);

        // Setting the keyword to search

        job.getConfiguration().set("keyword", "Jack");

        // Set input and output paths

        FileInputFormat.setInputPaths(job,        new     Path("/mapreduce/student.csv"));
FileOutputFormat.setOutputPath(job,newPath("/mapreduce/output/search/"));

        job.setNumReduceTasks(1);

        // Run the job and exit

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }

}
```

 Configures the job, sets the keyword, and specifies input and output paths.
 Uses only one reducer for simplicity.

(b) Mapper Class (WordSearchMapper.java)

The **Mapper scans each line**, searches for the **keyword**, and **outputs matching lines**.

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.mapreduce.Mapper;

import    org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class WordSearchMapper extends Mapper<LongWritable, Text, Text, Text> {

    static String keyword;

    static int pos = 0;

    @Override

    protected void setup(Context context) throws IOException, InterruptedException {

        // Get the keyword from the configuration

        Configuration configuration = context.getConfiguration();

        keyword = configuration.get("keyword");

    }

    @Override

    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
        // Get file name
        FileSplit fileSplit = (FileSplit) context.getInputSplit();
        String fileName = fileSplit.getPath().getName();
        // Track word position
        Integer wordPos;
        pos++;
        // If line contains the keyword, write output
        if (value.toString().contains(keyword)) {
            wordPos = pos;
            context.write(value, new Text(fileName + "," + wordPos.toString()));
        }
    }
}
```

 Extracts the keyword from the job configuration (setup method).
 Searches each line (map method) and writes matches with file name & position.

(c) Reducer Class (WordSearchReducer.java)

The **Reducer simply writes** the filtered results from the Mapper.

```
import java.io.IOException;
import org.apache.hadoop.io.Text;
import  org.apache.hadoop.mapreduce.Reducer;
public class WordSearchReducer extends Reducer<Text, Text, Text, Text> {
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
        for (Text value : values) {
            context.write(key, value);
        }
    }
}
```

 Receives filtered results and writes them as final output.
 No additional processing needed since the Mapper does the filtering.

After running the job, the **output is stored in Hadoop HDFS** at:

/mapreduce/output/search/part-r-00000

Content of part-r-00000:

1002,Jack,39 student.csv,5

 The program correctly identifies and outputs the row that contains "Jack".

Sorting

Objective to write a Mapreduce program to sort data by student name.

Input data (stored in student.csv file)

1001,John,45

1002,Jack,39

1003,Alex,44

1004,Smith,38

1005,Bob,33

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import  org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import   org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class SortStudNames {

    // Mapper Class
    public static class SortMapper extends Mapper<LongWritable, Text, Text, Text> {
        protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
            String[] token = value.toString().split(",");
            context.write(new Text(token[1]), new Text(token[0] + "," + token[2])); // Key:
Name, Value: ID,Score
        }
    }

    // Reducer Class
    public static class SortReducer extends Reducer<Text, Text, NullWritable, Text> {
        public void reduce(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
            for (Text details : values) {
                context.write(NullWritable.get(),    new    Text(key.toString()    +    ","    +
details.toString()));
            }
        }
    }
    // Driver Class
    public static void main(String[] args) throws IOException, InterruptedException, ClassNotFoundException {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "Sort Students by Name");
        job.setJarByClass(SortStudNames.class);
        job.setMapperClass(SortMapper.class);
        job.setReducerClass(SortReducer.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);
        job.setOutputKeyClass(NullWritable.class);
        job.setOutputValueClass(Text.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Output:

```
Alex,1003,44
Bob,1005,33
Jack,1002,39
John,1001,45
Smith,1004,38
```

Command to run the program:

hadoop jar StudentSort.jar SortStudNames /mapreduce/student.csv /mapreduce/output

After execution, check output:

hdfs dfs -cat /mapreduce/output/part-r-00000

COMPRESSION

In MapReduce programming, you can compress the MapReduce output file. Compression provides two benefits as follows:

1. Reduces the space to store files.
2. Speeds up data transfer across the network.

You can specify compression format in the Driver Program as shown below:

```
conf.setBoolean("mapred.output.compress", true);
conf.setClass("mapred.output.compression.codec",GzipCodec.class,CompressionCodec.class);
```

Here, codec is the implementation of a compression and decompression algorithm. GzipCodec is the compression algorithm for gzip. This compresses the output file.

*******END*******