**Q) List the use and responsibilities of following tools of Hadoop eco-system a. HDFS b. YARN c. FLUME d. SQOOP e. ZOOKEEPER f. HIVE**

**a. HDFS (Hadoop Distributed File System)**

**Use:**  
HDFS is the primary storage system used by Hadoop to store large datasets across multiple machines in a distributed fashion. It provides high-throughput access to application data and is capable of scaling up to thousands of nodes in a cluster.

**Responsibilities:**

1. **Storage of Large Datasets**: HDFS splits large datasets into blocks and stores them across different nodes for parallel processing.
2. **Fault Tolerance**: HDFS replicates data across multiple nodes to ensure data is not lost in case of hardware failures.
3. **Data Accessibility**: It allows seamless access to stored data via standard file system interfaces.
4. **High Throughput**: Designed for high-throughput rather than low-latency access to large datasets, making it ideal for batch processing.
5. **Scalability**: HDFS is designed to handle datasets that grow exponentially, making it scalable as data demands increase.

**b. YARN (Yet Another Resource Negotiator)**

**Use:**  
YARN is Hadoop’s cluster resource management system that handles job scheduling and resource allocation across the Hadoop cluster. It allows multiple applications to run simultaneously by dynamically managing resources.

**Responsibilities:**

1. **Resource Management**: YARN allocates resources like CPU and memory to various applications running on the Hadoop cluster.
2. **Job Scheduling**: It schedules jobs efficiently, managing the execution of tasks based on available resources.
3. **Multi-tenancy**: Enables running different types of applications (e.g., MapReduce, Spark) on the same cluster without conflict.
4. **Fault Tolerance**: YARN tracks and restarts failed jobs, ensuring smooth execution of applications.
5. **Dynamic Resource Allocation**: It ensures that resources are assigned to applications based on their current demands, improving cluster utilization.

**c. FLUME**

**Use:**  
Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data or streaming data from various sources into centralized storage, like HDFS.

**Responsibilities:**

1. **Data Ingestion**: Flume helps in ingesting data from multiple sources (web servers, social media, IoT devices) into HDFS or HBase.
2. **Streaming Data**: It is optimized for continuous and real-time data streams, handling large volumes of event data.
3. **Fault Tolerance**: Flume provides guaranteed delivery of data, even in case of node failure.
4. **Scalability**: It can be scaled horizontally by adding more agents to handle increasing data loads.
5. **Customizability**: Flume supports various custom sources, sinks, and channel types to meet different data ingestion needs.

**d. SQOOP**

**Use:**  
Sqoop is used for transferring data between Hadoop and relational databases (RDBMS). It imports data from databases like MySQL, Oracle, and PostgreSQL into HDFS and exports processed data back to the databases.

**Responsibilities:**

1. **Data Import**: Sqoop imports structured data from relational databases into Hadoop for further analysis.
2. **Data Export**: It exports processed data from HDFS back to the relational database for reporting or further use.
3. **Efficient Data Transfer**: Sqoop uses parallel processing to efficiently transfer large datasets to and from Hadoop, ensuring high performance.
4. **Data Transformation**: During import/export, it can also transform the data into a desired format (e.g., CSV, Avro).
5. **Schema Mapping**: Automatically generates schema for importing tables, ensuring smooth integration with relational databases.

**e. ZOOKEEPER**

**Use:**  
Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and group services within a Hadoop cluster. It is essential for coordinating distributed applications.

**Responsibilities:**

1. **Coordination**: It coordinates various distributed services and applications in a cluster to ensure they operate synchronously.
2. **Configuration Management**: Zookeeper stores and manages configuration information and distributes it across all nodes in the cluster.
3. **Leader Election**: Facilitates leader election in a distributed system to ensure one master node controls task execution while others are standby.
4. **Distributed Locking**: Provides a locking mechanism to avoid conflicts in accessing shared resources in a distributed environment.
5. **Fault Tolerance**: Zookeeper itself is highly available and ensures consistency across nodes even if one or more nodes fail.

**f. HIVE**

**Use:**  
Hive is a data warehouse infrastructure built on top of Hadoop that provides data summarization, querying, and analysis through an SQL-like interface (HiveQL). It allows users familiar with SQL to query data stored in HDFS without writing complex MapReduce programs.

**Responsibilities:**

1. **SQL-like Querying**: Hive allows users to write queries in HiveQL (similar to SQL) to retrieve and manipulate large datasets in HDFS.
2. **Data Warehousing**: Hive is primarily used for data warehousing tasks like data summarization, ad-hoc querying, and analysis.
3. **Schema on Read**: Hive applies a schema to data as it is read (rather than when it is written), making it flexible for analyzing unstructured or semi-structured data.
4. **Integration with Hadoop**: Hive translates HiveQL queries into MapReduce jobs, allowing data to be processed on Hadoop’s distributed computing framework.
5. **Extensibility**: Hive supports user-defined functions (UDFs) for custom data processing and analysis, enabling users to extend its capabilities.

**Q) Explain the responsibilities of Name node and Data nodes in HDFS. Differentiate between normal file system with HDFS.**

### ****Responsibilities of Name Node and Data Nodes in HDFS****

#### **Name Node**

The Name Node is the central part of HDFS and acts as the master server responsible for managing the metadata and file system namespace. It coordinates the storage and retrieval of the data blocks from the Data Nodes.

**Responsibilities of Name Node:**

1. **Metadata Storage**: Name Node stores metadata (e.g., file names, permissions, locations of blocks) for the entire file system in memory.
2. **Namespace Management**: It maintains the hierarchical structure of the file system, including directories, subdirectories, and files.
3. **Data Block Management**: Name Node tracks the locations of data blocks stored on different Data Nodes, ensuring they are correctly replicated and accessible.
4. **Data Block Replication**: It ensures data is fault-tolerant by replicating blocks across multiple Data Nodes according to the configured replication factor.
5. **Client Interaction**: When a client wants to read or write data, the Name Node provides the necessary block information and coordinates access to Data Nodes.
6. **Failover Support**: In case of Data Node failures, the Name Node ensures that the data is available by tracking replicas and initiating replication when necessary.

#### **Data Nodes**

Data Nodes serve as the workers in HDFS, responsible for storing actual data blocks. They regularly communicate with the Name Node to ensure the system functions smoothly.

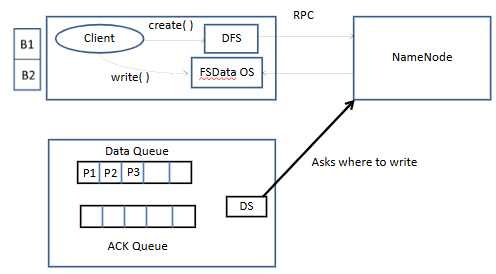
**Responsibilities of Data Nodes:**

1. **Storage of Data Blocks**: Data Nodes store and manage the physical storage of data blocks assigned by the Name Node.
2. **Data Block Reports**: They periodically send block reports to the Name Node, indicating the blocks they are storing.
3. **Data Replication**: Data Nodes participate in the replication process by replicating data to other Data Nodes as instructed by the Name Node.
4. **Data Read/Write**: They manage the actual data I/O operations, providing data to clients or accepting new data for storage.
5. **Heartbeat Signals**: Data Nodes send heartbeat signals to the Name Node at regular intervals to indicate they are functioning properly.
6. **Error Handling**: If a Data Node fails, its data blocks are replicated on other nodes by the Name Node to ensure fault tolerance.

**Q) Differences between Normal File System and HDFS**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Normal File System** | **HDFS (Hadoop Distributed File System)** |
| **Storage Architecture** | Single machine (centralized storage). | Distributed across multiple machines (nodes). |
| **Fault Tolerance** | Limited fault tolerance; data loss can occur on hardware failure. | High fault tolerance; data blocks are replicated across multiple nodes. |
| **Data Size Handling** | Handles small to medium-sized files. | Designed to handle very large datasets (GB, TB, PB). |
| **Data Access** | Local file system, accessed by one machine. | Distributed file system, accessible across the cluster. |
| **Block Size** | Small block size (e.g., 4KB, 8KB). | Large block size (default is 128MB or 256MB in HDFS). |
| **File System Type** | Hierarchical file system for general purpose use. | Designed for write-once, read-many access patterns for large data files. |
| **Replication** | No automatic replication; backup needed manually. | Automatic replication of data blocks for fault tolerance. |
| **Scalability** | Limited scalability as it is restricted to single machine storage capacity. | High scalability; can easily scale out by adding more Data Nodes. |
| **Data Processing** | Data processing is done on the same machine where the file system resides. | Data is processed in a distributed manner across multiple nodes (data locality concept). |
| **Metadata Handling** | Stores metadata with each file locally. | Name Node stores metadata in memory, while Data Nodes store only data. |
| **Use Cases** | Historical trend analysis, machine learning model training. | Real-time monitoring, alerting systems, and dashboards. |

**Q) With diagram show how write operation is performed in HDFS. List the steps to recover from failure during HDFS write operation.**



#### Step-by-Step Explanation of Write Operation in HDFS

1. **Client Request**:  
   The client sends a request to the **Name Node** to create a file and write data. The Name Node checks if the file already exists and whether the client has the appropriate permissions to create the file.
2. **Block Assignment**:  
   The Name Node splits the file into blocks (default block size is 128MB) and identifies a list of **Data Nodes** to store each block. It returns the list of Data Nodes where the replicas of the data blocks should be stored.
3. **Pipeline Setup**:  
   The client establishes a pipeline connection with the first Data Node (Replica 1). This Data Node, in turn, communicates with the second Data Node (Replica 2), and the second Data Node communicates with the third Data Node (Replica 3), forming a chain.
4. **Writing Data**:  
   The client starts writing data blocks. Each block is written to the first Data Node (Replica 1), which immediately forwards it to the second Data Node (Replica 2), and then it is forwarded to the third Data Node (Replica 3). This is done in a **pipelined** fashion, ensuring all replicas are written simultaneously.
5. **Acknowledgement**:  
   After the block is written to all replicas, the Data Nodes acknowledge the successful write. The third Data Node (Replica 3) sends an acknowledgment to the second Data Node (Replica 2), which forwards it to the first Data Node (Replica 1), and finally, the first Data Node sends the acknowledgment back to the client.
6. **File Close**:  
   After all blocks are written and acknowledged, the client informs the Name Node that the file has been closed. The Name Node then commits the file creation in its metadata.

### Steps to Recover from Failure during HDFS Write Operation

**In case of failure during the write operation in HDFS, the system provides mechanisms to ensure data integrity and recover from the failure.**

#### 1. Data Node Failure

* **If a Data Node fails during the write operation, the** pipeline is broken**. In such a case, the following actions are taken:**
  + **The client is notified about the failure.**
  + **The Name Node removes the failed Data Node from the pipeline and reassigns the write operation to a new Data Node to maintain the replication factor.**
  + **The pipeline resumes, and the data block is replicated to the new Data Node.**

#### 2. Client Failure

* **If the client fails or disconnects during the write operation, the incomplete data is removed, and the file remains** uncommitted**.**
  + **The** Name Node **waits for the client to re-establish the connection and continue the write operation.**
  + **If the client does not reconnect, the Name Node marks the file as** incomplete **and removes any partially written blocks to avoid corruption.**

#### 3. Network Failure

* **In the case of a network failure between the Data Nodes:**
  + **The current block being written is saved in the existing Data Nodes that have received the data.**
  + **Once the network recovers, the Name Node re-establishes the connection with new Data Nodes to ensure the block is fully replicated.**

#### 4. Name Node Failure

* **If the** Name Node **fails during the write operation:**
  + **HDFS uses a** secondary Name Node **or** Checkpoint Node **to recover metadata and file system state.**
  + **The system brings the Name Node back online, and the incomplete write operations can be restarted from where they left off.**

#### 5. Handling Data Corruption

* **HDFS uses** checksums **to verify data integrity. If any Data Node stores a corrupt block:**
  + **The Name Node detects this using** block reports**.**
  + **The corrupt block is deleted, and a new replica is created from a healthy copy on another Data Node.**

**Q) Explain how HDFS ensures high availability of data and services.**

HDFS (Hadoop Distributed File System) ensures **high availability** of data and services through a combination of redundancy, fault tolerance, and failover mechanisms. Here’s how it achieves this:

**1. Data Replication**

* **Replication Factor**: In HDFS, each data block is replicated across multiple Data Nodes. By default, a block is replicated **3 times**, though this factor can be adjusted based on needs.
  + **Primary Replica**: Stored on the node where the file is initially written.
  + **Secondary Replica**: Stored on a different node in a different rack to avoid data loss in case of rack failure.
  + **Tertiary Replica**: Stored on yet another node or rack for additional fault tolerance.
* **Purpose of Replication**: In case of node failure, the system can retrieve data from other replicas, ensuring the data is always available for processing.

**2. Fault Tolerance**

* **Automatic Failover**: If a Data Node fails, the Name Node automatically redirects the client to another node that holds the replica of the data.
* **Self-Healing Mechanism**: HDFS continuously monitors the health of Data Nodes. If a block replica is found to be missing or corrupted, the Name Node ensures that a new replica is created from the existing healthy copies.

**3. Name Node High Availability (HA)**

* Traditionally, HDFS had a **single Name Node**, creating a single point of failure. However, modern Hadoop versions implement **Name Node High Availability (HA)**, which eliminates this risk.
* **Active and Standby Name Nodes**:
  + Two Name Nodes operate in a highly available configuration: an **active** Name Node and a **standby** Name Node.
  + Both nodes share the same metadata stored on a **shared storage** system, such as a **JournalNode** or **Quorum-based Storage**.
  + In the event of a failure of the active Name Node, the standby Name Node automatically takes over without any data loss or downtime.
* **Failover Mechanism**:
  + A **failover controller** monitors the health of both Name Nodes. If the active Name Node goes down, the controller switches to the standby, making it the new active Name Node.

**4. Rack Awareness**

* **Rack-Aware Data Placement**: HDFS uses a **rack-awareness algorithm** to ensure replicas are placed on different racks.
* **Purpose**: In case of rack failure (e.g., network switch or power failure), data can still be accessed from replicas on different racks, ensuring the system remains available.
* **Optimized Network Bandwidth**: Storing replicas on different racks also optimizes the use of network bandwidth, ensuring efficient read and write operations even in case of rack failure.

**5. Heartbeat and Block Reports**

* **Heartbeat Mechanism**: Data Nodes send periodic **heartbeats** to the Name Node to inform it of their health and availability. If a Data Node stops sending heartbeats, the Name Node marks it as unavailable and begins replicating its data to other nodes.
* **Block Reports**: Data Nodes also send **block reports** to the Name Node, detailing the blocks they store. The Name Node uses these reports to ensure that all data blocks are accounted for and correctly replicated across the cluster.

**6. Checksum Verification**

* HDFS ensures data integrity by storing **checksums** for each block of data. Every time data is read, the checksum is validated to ensure the data hasn't been corrupted.
* If corruption is detected, HDFS can retrieve a valid replica from another Data Node and re-replicate the corrupt block, ensuring that the client only receives correct data.

**7. Backup and Recovery Mechanism**

* **Snapshots**: HDFS supports **snapshots**, which are read-only copies of the file system at a particular point in time. These can be used for backup and recovery in case of accidental data deletion or corruption.
* **Secondary Name Node/Checkpoint Node**: Although not a true backup, the **Secondary Name Node** (or **Checkpoint Node**) periodically saves a copy of the file system metadata and edits logs, ensuring that the Name Node can recover after a failure.

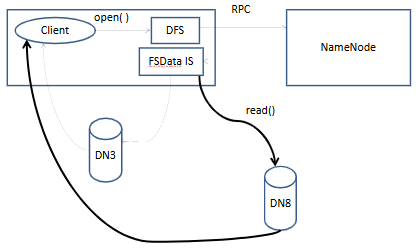
**8. Quorum-based Storage for Metadata (Journal Nodes)**

* To avoid a single point of failure for the metadata stored by the Name Node, HDFS uses **Quorum-based Journal Nodes**.
* **Quorum Mechanism**: When Name Nodes (both active and standby) update metadata, they write to a set of Journal Nodes. Only if a majority (quorum) of Journal Nodes have written the data does the operation succeed, ensuring the metadata is reliably stored.

**9. Data Locality for Efficient Processing**

* HDFS optimizes data processing by following the **data locality principle**, where the processing tasks are sent to the nodes where data is stored, reducing network traffic and improving the availability and performance of jobs.

**Q) With diagram show how read operation is performed in HDFS. Explain the steps involved in read operation.**

****

### ****Steps Involved in Read Operation****

#### **1. Client Request to Name Node**

* The **client** initiates the read operation by sending a **request** to the **Name Node** to access a file. The request includes the file name and its location in the HDFS file system.

#### **2. Block Location Retrieval from Name Node**

* The **Name Node** processes the request and checks its **metadata** to determine the locations of the blocks that make up the requested file.
* The Name Node returns a list of **Data Nodes** that hold replicas of each block of the file. The client is then informed of which Data Nodes hold the data.

#### **3. Connecting to Data Nodes**

* Once the client receives the block locations from the Name Node, it directly establishes a connection with the **closest Data Node** that holds the first block of the file.
* HDFS follows the **data locality principle**, which means the client attempts to read from the Data Node that is nearest (ideally within the same rack or network) to minimize data transfer time.

#### **4. Data Block Reading**

* The client starts reading the **data block** from the selected Data Node (say **Data Node 1** for Block 1).
* If the block is too large, the data is read in **chunks** using a stream.
* After finishing the read of the first block, the client proceeds to read the subsequent blocks in a similar manner by connecting to the respective Data Nodes (e.g., **Data Node 2** for Block 2, **Data Node 3** for Block 3).

#### **5. Block Replicas as Backup**

* If any of the Data Nodes holding the required block is **unavailable** or slow to respond, the client can request the block from one of its **replicas** stored on other Data Nodes.
* The client will use the closest available replica of the block to continue reading.

#### **6. Data Return to Client**

* After retrieving all the blocks of the file, the client assembles them in the correct sequence.
* The client now has the full content of the file, which can be used or processed further.

#### **7. No Interaction with Name Node During Data Transfer**

* After the initial block location request, the client communicates **directly** with the Data Nodes. The **Name Node** is not involved in the data transfer process.
* This design ensures **scalability** by offloading the Name Node from handling each data transfer, making HDFS suitable for handling a large number of concurrent read requests.

### ****Detailed Step-by-Step Process****

1. **Client File Read Request**:
   * The client sends a request to the Name Node to read a file. The Name Node responds with the locations of the blocks and their replicas across the Data Nodes.
2. **Data Node Selection**:
   * Based on the information from the Name Node, the client selects the nearest or most suitable Data Node for reading the blocks of data.
3. **Establishing Data Connection**:
   * The client establishes a TCP/IP connection with the first Data Node that holds the required block.
4. **Reading Data Blocks**:
   * The client reads the first block and continues to the next blocks, fetching them sequentially from the respective Data Nodes.
5. **Handling Failures**:
   * If a Data Node fails to respond or is slow, the client retries from another Data Node holding a replica of the block.
6. **Reassembling Data**:
   * The client reassembles the blocks in order and combines them to obtain the original file.
7. **End of Read Operation**:
   * The read operation is completed, and the client can use or process the data as needed.

**Q) Explain the terms heart-beat, edit-log and check-points.**

In HDFS, the terms **heartbeat**, **edit log**, and **checkpoint** are crucial for maintaining system health, data consistency, and fault tolerance. Here’s a detailed explanation of each term:

### ****1. Heartbeat****

#### **Definition:**

A **heartbeat** is a periodic signal sent by a Data Node to the Name Node to indicate that it is alive and functioning correctly.

#### **Purpose:**

* **Health Monitoring**: Ensures that the Name Node is aware of the health and status of all Data Nodes in the cluster.
* **Data Node Availability**: Helps in detecting and managing Data Nodes that have failed or become unreachable.

#### **Mechanism:**

* **Interval**: Data Nodes send heartbeats at regular intervals (default is every 3 seconds).
* **Response**: If the Name Node stops receiving heartbeats from a Data Node within a specified timeout period (default is 10 minutes), it marks the Data Node as **dead** and starts the process to replicate the data blocks stored on the failed node to other healthy nodes.

#### **Impact:**

* **Fault Detection**: Rapid detection of failures in the Data Nodes ensures minimal disruption and data loss.
* **Cluster Management**: Helps the Name Node manage and rebalance data across the cluster efficiently.

### ****2. Edit Log****

#### **Definition:**

The **edit log** is a file maintained by the Name Node that records all changes made to the file system metadata. It logs operations such as file creation, deletion, and modifications.

#### **Purpose:**

* **Metadata Consistency**: Ensures that changes to the file system are recorded sequentially and can be replayed to reconstruct the state of the file system in case of a failure.
* **Recovery**: Used during the recovery process to rebuild the file system metadata that might be lost or corrupted.

#### **Mechanism:**

* **Writing**: Each change to the file system metadata (e.g., creating a new file, modifying permissions) is written to the edit log before it is applied to the in-memory metadata.
* **Size Management**: The edit log grows over time as more changes are made. Regular checkpoints help manage the size of the edit log by merging it with the file system metadata.

#### **Impact:**

* **Durability**: Provides a durable record of all changes, which is critical for recovering from crashes or failures.
* **Recovery Process**: In case of Name Node failure, the edit log is used to replay changes and restore the file system to its most recent consistent state.

### ****3. Checkpoint****

#### **Definition:**

A **checkpoint** is a snapshot of the file system metadata and its state, including the edits recorded in the edit log. The checkpoint process consolidates the in-memory metadata and the edit log into a single file system image.

#### **Purpose:**

* **Metadata Recovery**: Reduces the amount of work needed to recover the file system by reducing the size of the edit log that needs to be replayed.
* **Efficiency**: Ensures that the Name Node can start up more quickly by applying a consolidated snapshot of the metadata rather than replaying a large edit log.

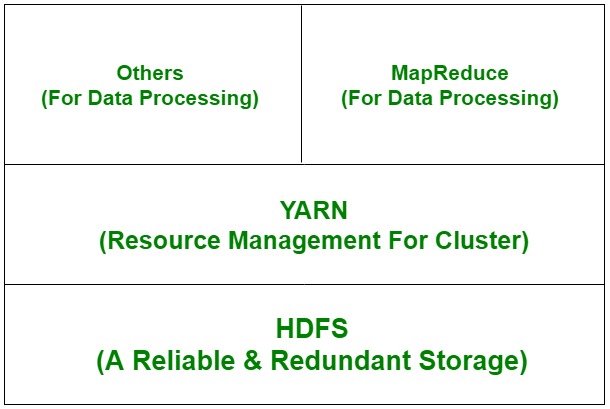
#### **Mechanism:**

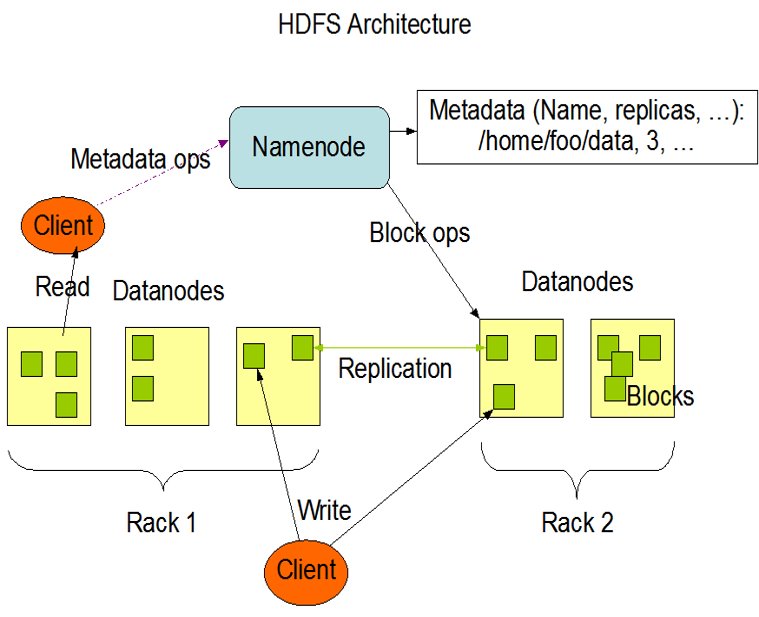
* **Checkpoint Node**: A secondary Name Node (Checkpoint Node) periodically performs the checkpointing process. It creates a new file system image by applying the changes from the edit log to a previous snapshot of the metadata.
* **Saving**: The checkpoint is saved in a shared storage system accessible to the active Name Node and is used to initialize the Name Node's state during recovery.

#### **Impact:**

* **System Recovery**: Facilitates faster recovery of the Name Node by reducing the need to replay the entire edit log.
* **Storage Management**: Helps manage the size of the edit log, preventing it from growing indefinitely and impacting performance.

**Q) With neat diagram, explain the architecture of HDFS.**



****

**Q) List the major components of YARN. Explain the responsibilities of YARN.**

**Major Components of YARN**

YARN (Yet Another Resource Negotiator) is a resource management layer of Hadoop that allows for more flexible and efficient resource management compared to the older MapReduce framework. It decouples the resource management and job scheduling functionalities, which enhances the scalability and performance of the Hadoop ecosystem.

Here are the major components of YARN:

1. **ResourceManager (RM)**
   * **Role**: Manages and allocates cluster resources.
   * **Function**:
     + **Resource Management**: Keeps track of available resources and allocates them to various applications.
     + **Scheduler**: Allocates resources to different applications based on various scheduling policies (e.g., Capacity Scheduler, Fair Scheduler).
     + **Application Management**: Receives and manages job submissions and resource requests from clients.
2. **NodeManager (NM)**
   * **Role**: Manages resources on individual nodes and handles container execution.
   * **Function**:
     + **Resource Monitoring**: Monitors resource usage (CPU, memory) on its node.
     + **Container Management**: Launches and manages containers as requested by the ResourceManager.
     + **Reporting**: Sends resource usage and health reports to the ResourceManager.
     + **Log Aggregation**: Manages and aggregates logs from running applications.
3. **ApplicationMaster (AM)**
   * **Role**: Manages the lifecycle of a specific application (job) and coordinates with the ResourceManager and NodeManagers.
   * **Function**:
     + **Application Coordination**: Manages the execution of the application by requesting resources from the ResourceManager and scheduling tasks.
     + **Task Management**: Coordinates tasks within the application and manages their execution in containers.
     + **Failure Recovery**: Handles failures by retrying tasks and managing fault tolerance for the application.
4. **Container**
   * **Role**: Provides a runtime environment for executing tasks of an application.
   * **Function**:
     + **Task Execution**: Contains the resources and environment necessary for running a specific task (e.g., a MapReduce task).
     + **Resource Isolation**: Ensures isolation between different tasks and applications by allocating dedicated resources (CPU, memory) for each container.
5. **ApplicationMaster (AM)**
   * **Role**: Manages the lifecycle of an application.
   * **Function**:
     + **Resource Negotiation**: Requests resources from the ResourceManager and monitors their allocation.
     + **Task Scheduling**: Schedules and manages the execution of tasks within allocated containers.
     + **Monitoring and Recovery**: Monitors the progress of the application and handles retries and recovery in case of task failures.

**Responsibilities of YARN**

YARN is designed to handle resource management and job scheduling in Hadoop clusters more efficiently and flexibly. Its primary responsibilities include:

1. **Resource Allocation**:
   * **Dynamic Resource Management**: Allocates cluster resources dynamically based on demand and application needs. The ResourceManager schedules resources based on predefined policies and application priorities.
   * **Fair Allocation**: Ensures resources are distributed fairly among competing applications, preventing any single application from monopolizing the cluster resources.
2. **Application Scheduling**:
   * **Job Scheduling**: Manages the scheduling and execution of applications (jobs) in the cluster. The ResourceManager uses various scheduling algorithms (e.g., Capacity Scheduler, Fair Scheduler) to allocate resources effectively.
   * **Application Coordination**: Coordinates the execution of applications by launching ApplicationMasters, which then handle task execution within the allocated containers.
3. **Resource Monitoring**:
   * **Node Resource Monitoring**: The NodeManager monitors and reports the resource usage of individual nodes (e.g., CPU, memory) to the ResourceManager. This helps in making informed decisions about resource allocation and ensuring efficient utilization of cluster resources.
   * **Application Monitoring**: The ApplicationMaster monitors the progress and health of tasks within an application, managing task execution and handling failures as needed.
4. **Fault Tolerance and Recovery**:
   * **Task Retry and Recovery**: The ApplicationMaster handles task failures by retrying failed tasks and managing fault tolerance within the application. This ensures that applications can recover from task failures and continue processing.
   * **Node Failure Handling**: The ResourceManager detects node failures and reallocates resources from failed nodes to other healthy nodes to maintain cluster availability and performance.
5. **Cluster Utilization**:
   * **Optimized Utilization**: YARN optimizes the utilization of cluster resources by enabling multiple applications to share the same cluster resources. It supports various types of applications beyond MapReduce, such as Spark, Tez, and custom applications.
   * **Resource Isolation**: Ensures that resources allocated to one application do not interfere with resources allocated to other applications, maintaining isolation and stability.
6. **Scalability**:
   * **Scaling**: YARN is designed to scale out to large clusters, handling thousands of nodes and applications efficiently. It decouples the resource management from the application execution, allowing for better scalability and resource utilization.

**Q) List out the responsibilities of Resource Manager in YARN.**

The **ResourceManager (RM)** in YARN (Yet Another Resource Negotiator) is a critical component responsible for managing the cluster's resources and coordinating the allocation of those resources to various applications. Here are the primary responsibilities of the ResourceManager:

**Responsibilities of ResourceManager**

1. **Resource Allocation and Management**:
   * **Resource Tracking**: Monitors and tracks the resources available in the cluster, including CPU, memory, and disk usage.
   * **Resource Scheduling**: Allocates cluster resources to applications based on predefined scheduling policies. This involves deciding how much resource to allocate to each application and when.
   * **Scheduling Policies**: Implements various scheduling policies such as Capacity Scheduler, Fair Scheduler, or custom policies to manage resource distribution and ensure fair access.
2. **Application Submission and Management**:
   * **Application Submission**: Receives and processes application submissions from clients. This includes accepting requests to run applications and initiating the process to allocate resources.
   * **Application Lifecycle Management**: Manages the lifecycle of applications, including launching the ApplicationMaster for each application and monitoring its status.
3. **Application Master Coordination**:
   * **ApplicationMaster Launch**: Coordinates the launching of ApplicationMasters for each application. Each ApplicationMaster is responsible for managing the execution of tasks within allocated containers.
   * **Resource Requests**: Handles resource requests from ApplicationMasters and allocates resources as per the application’s requirements and cluster availability.
4. **Resource Monitoring and Reporting**:
   * **Resource Utilization**: Monitors overall resource utilization and availability in the cluster. It keeps track of which resources are being used and by which applications.
   * **Status Reporting**: Provides status reports on resource allocation and application progress. It ensures that the cluster resources are being used efficiently.
5. **Failure Detection and Recovery**:
   * **Node Failure Detection**: Detects failures of NodeManagers and marks the associated resources as unavailable. The ResourceManager reallocates resources from failed nodes to other healthy nodes.
   * **Application Recovery**: Handles application failures by reassigning resources and, if necessary, restarting ApplicationMasters or retrying failed tasks.
6. **Queue Management**:
   * **Queue Configuration**: Manages and enforces resource quotas and policies for different queues (e.g., Capacity Scheduler queues or Fair Scheduler queues). This involves setting up and managing resource quotas, access policies, and priorities for different types of workloads.
   * **Queue Scheduling**: Schedules resources according to the policies defined for various queues, ensuring fair access and efficient utilization.
7. **Scalability and Load Balancing**:
   * **Scalability**: Ensures that the ResourceManager can handle a large number of nodes and applications, scaling efficiently with the growth of the cluster.
   * **Load Balancing**: Balances resource allocation across the cluster to avoid bottlenecks and ensure even distribution of resources.
8. **Integration with Other Components**:
   * **Coordination with NodeManager**: Interacts with NodeManagers to obtain updates on resource availability and health status. This interaction is crucial for accurate resource tracking and allocation.
   * **Integration with Other Frameworks**: Supports various frameworks beyond MapReduce, such as Apache Spark, Apache Tez, and custom applications, by providing a unified resource management interface.

**Q) Explain different schedulers used in YARN.**

YARN (Yet Another Resource Negotiator) supports various scheduling policies to manage resource allocation effectively across the cluster. The main schedulers used in YARN are:

### ****1. Capacity Scheduler****

#### **Overview:**

The Capacity Scheduler is designed to allocate resources to multiple queues based on configured capacities. It ensures that each queue gets its fair share of resources and that large applications do not monopolize the cluster.

#### **Key Features:**

* **Queue-based Allocation**: Resources are allocated to different queues, each associated with a particular application or set of applications.
* **Configured Capacities**: Each queue is assigned a specific capacity, which represents the percentage of the total cluster resources that the queue is entitled to.
* **Fair Sharing**: Within a queue, resources are allocated fairly among the applications.
* **Preemption**: Can preempt resources from lower-priority queues if higher-priority queues need more resources.

#### **Use Case:**

Ideal for environments where multiple teams or departments share the same cluster and need guaranteed resource allocations.

### ****2. Fair Scheduler****

#### **Overview:**

The Fair Scheduler aims to allocate resources equally among running applications to ensure that all applications receive a fair share of cluster resources over time.

#### **Key Features:**

* **Fair Allocation**: Resources are distributed fairly among all applications, ensuring that no single application gets more than its fair share.
* **Dynamic Adjustment**: Adjusts resource allocation dynamically based on the number of applications and their resource requirements.
* **Queue-based Configuration**: Supports hierarchical queues, allowing the configuration of parent-child relationships between queues.
* **Preemption**: Can preempt resources from applications that have received more than their fair share.

#### **Use Case:**

Suitable for scenarios where multiple applications need to share resources equally and where there is a need for dynamic resource allocation based on the current workload.

### ****3. FIFO (First In, First Out) Scheduler****

#### **Overview:**

The FIFO Scheduler is the simplest form of scheduling in YARN, where applications are allocated resources in the order they are submitted.

#### **Key Features:**

* **Simple Queue**: Applications are placed in a single queue and are processed in the order they arrive.
* **No Fairness**: Does not enforce fairness or capacity constraints. The first application to be submitted gets the resources first.
* **No Resource Guarantees**: No guarantees for resource allocation; resources are allocated on a first-come, first-served basis.

#### **Use Case:**

Suitable for small clusters or environments where simplicity is preferred and resource contention is not a major concern.

### ****4. CapacityScheduler with Queues****

#### **Overview:**

This is an extension of the Capacity Scheduler that uses hierarchical queue structures to provide more sophisticated resource allocation and management.

#### **Key Features:**

* **Hierarchical Queues**: Supports nested queue configurations, allowing for complex resource allocation strategies.
* **Dynamic Adjustments**: Queues can be dynamically adjusted to balance load and priorities.
* **Preemption**: Preemption policies can be configured to ensure high-priority queues get resources even if it means taking resources away from lower-priority queues.

#### **Use Case:**

Effective in environments where hierarchical resource allocation is needed, such as large organizations with multiple teams and projects requiring different levels of resource guarantees.

### ****5. Fair Scheduler with Capacity Guarantees****

#### **Overview:**

This scheduler combines the principles of the Fair Scheduler with capacity guarantees, ensuring both fairness and certain resource guarantees for different applications.

#### **Key Features:**

* **Fair Allocation**: Allocates resources fairly among applications.
* **Capacity Guarantees**: Ensures that each application or queue receives a minimum guaranteed amount of resources.
* **Dynamic Adjustment**: Balances resource allocation dynamically based on application needs and configured capacities.

#### **Use Case:**

Suitable for environments where both fairness and guaranteed resource allocation are critical, providing a balance between efficient resource use and meeting minimum requirements for applications.

**Q) Explain different types of failures in Map-Reduce jobs.**

MapReduce jobs can encounter various types of failures due to the distributed nature of the framework. Here are the different types of failures that can occur in MapReduce jobs and how they are generally handled:

**1. Task Failures**

**Description:** Task failures occur when an individual map or reduce task fails to complete successfully. This can happen due to several reasons, such as hardware issues, software bugs, or data corruption.

**Handling Mechanism:**

* **Task Retry:** The JobTracker (in Hadoop 1.x) or ResourceManager (in Hadoop 2.x/YARN) detects task failures and schedules the task to be retried on another node.
* **Task Re-execution:** If a task fails multiple times, the framework may re-execute the task on a different node to ensure completion. The default retry limit is typically three attempts.

**2. Node Failures**

**Description:** Node failures occur when a node in the Hadoop cluster becomes unresponsive or crashes, causing tasks running on that node to fail.

**Handling Mechanism:**

* **Heartbeat Monitoring:** The ResourceManager or JobTracker monitors heartbeats from nodes. If a node fails to send a heartbeat within a specified time, it is considered failed.
* **Task Reallocation:** The tasks running on the failed node are reassigned to other available nodes. The framework uses the data replication mechanism to ensure that the necessary data is available on other nodes.

**3. JobTracker/ResourceManager Failures**

**Description:** Failures of the JobTracker (in Hadoop 1.x) or ResourceManager (in Hadoop 2.x/YARN) can disrupt the management of job scheduling and resource allocation.

**Handling Mechanism:**

* **High Availability (HA) Setup:** In Hadoop 2.x/YARN, ResourceManager HA can be configured using a standby ResourceManager that takes over if the active ResourceManager fails. In Hadoop 1.x, JobTracker HA can be achieved by configuring a secondary JobTracker.
* **State Recovery:** The framework periodically saves the state of running jobs to disk. In the event of a failure, the framework can recover the job state from these saved checkpoints.

**4. Data Node Failures**

**Description:** Data node failures occur when a DataNode storing the data blocks becomes unresponsive or crashes, leading to potential data unavailability.

**Handling Mechanism:**

* **Data Replication:** HDFS replicates data blocks across multiple DataNodes to ensure fault tolerance. If a DataNode fails, the missing blocks are retrieved from other replicas.
* **Re-replication:** The NameNode detects under-replicated blocks and initiates the process of replicating these blocks to other DataNodes to maintain the desired replication factor.

**5. ApplicationMaster Failures**

**Description:** ApplicationMaster failures occur when the ApplicationMaster, which manages the lifecycle of a specific job, fails or becomes unresponsive.

**Handling Mechanism:**

* **Retry Mechanism:** YARN handles ApplicationMaster failures by retrying the ApplicationMaster on different nodes. The application state is recovered from the last checkpoint if possible.
* **State Persistence:** The state of running applications is periodically saved to facilitate recovery in case of ApplicationMaster failure.

**6. Network Failures**

**Description:** Network failures can cause communication issues between nodes, leading to task failures or delays.

**Handling Mechanism:**

* **Data Replication:** Data replication and block re-replication ensure that data is not lost due to network issues.
* **Task Retries:** The framework handles task failures caused by network issues by retrying the tasks on different nodes, ensuring that data is processed despite network disruptions.

**7. Disk Failures**

**Description:** Disk failures can cause data corruption or loss if the disks storing data blocks fail.

**Handling Mechanism:**

* **Data Replication:** HDFS replicates data blocks across multiple disks and nodes to mitigate the risk of data loss due to disk failures.
* **Re-replication:** The NameNode detects missing or corrupted blocks and initiates re-replication to ensure data integrity.

**Summary**

* **Task Failures:** Handled by retrying the failed tasks on other nodes.
* **Node Failures:** Managed by reallocating tasks to other nodes and using data replication.
* **JobTracker/ResourceManager Failures:** Addressed through high availability configurations and state recovery.
* **Data Node Failures:** Mitigated by data replication and re-replication.
* **ApplicationMaster Failures:** Managed by retrying the ApplicationMaster and recovering state from checkpoints.
* **Network Failures:** Handled by task retries and data replication.
* **Disk Failures:** Addressed through data replication and re-replication.

Each type of failure has specific handling mechanisms designed to ensure that MapReduce jobs can complete successfully despite various issues in the distributed environment.

**Q) Give different reasons for task failures in Map-Reduce job. Give the steps for recovering from Application master failures.**

**Reasons for Task Failures in MapReduce Jobs**

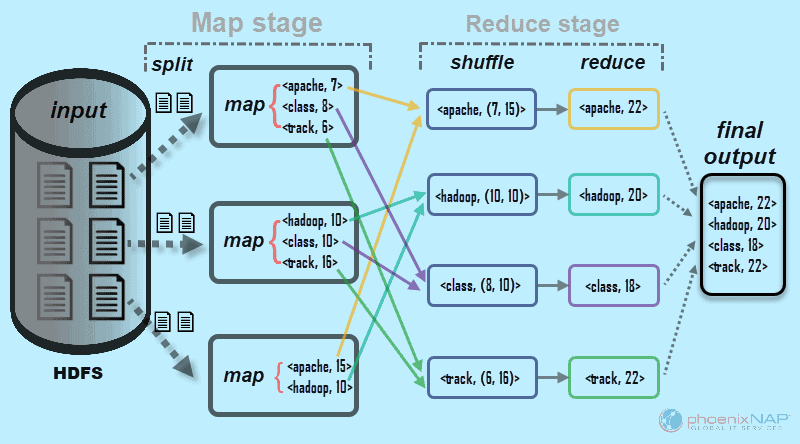
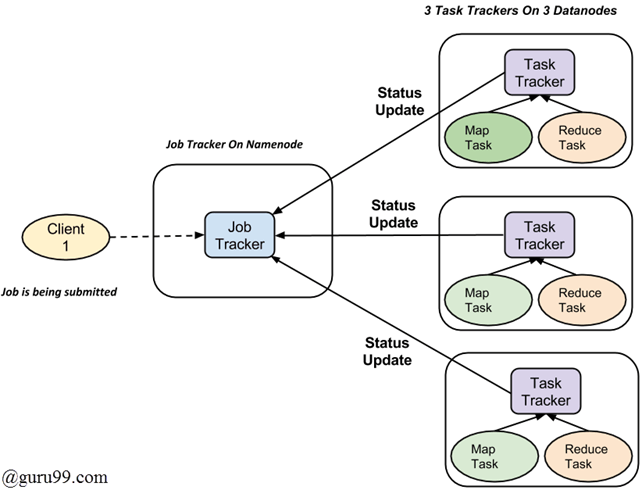
1. **Hardware Failures:**
   * **Disk Failures:** Physical disk issues can lead to data read/write errors.
   * **Memory Issues:** Insufficient memory or memory leaks can cause tasks to fail.
2. **Software Bugs:**
   * **Application Bugs:** Errors in the MapReduce application code can lead to task failures.
   * **Library or Dependency Issues:** Bugs in libraries or dependencies used by the application can cause failures.
3. **Data Corruption:**
   * **Corrupted Input Data:** Issues with the input data (e.g., corrupted files) can lead to task failures.
   * **Data Format Issues:** Incompatible data formats or parsing errors can cause tasks to fail.
4. **Resource Constraints:**
   * **CPU or Memory Limitations:** Tasks may fail if they exceed the allocated CPU or memory limits.
   * **Insufficient Disk Space:** Tasks may fail if there is not enough disk space to store intermediate data.
5. **Network Issues:**
   * **Network Partitions:** Network partitions or connectivity issues between nodes can cause tasks to fail.
   * **Slow Network:** High network latency or slow data transfer rates can lead to timeouts and task failures.
6. **Configuration Errors:**
   * **Misconfiguration:** Incorrect settings in job configuration files or Hadoop cluster configurations can lead to task failures.
   * **Resource Misallocation:** Inadequate resource allocation due to configuration issues can cause tasks to fail.
7. **Dependency Failures:**
   * **External Service Failures:** Dependencies on external services (e.g., databases or APIs) that become unavailable can lead to task failures.
8. **File System Issues:**
   * **HDFS Failures:** Issues with the Hadoop Distributed File System (HDFS), such as corrupted blocks or name node failures, can impact task execution.

**Steps for Recovering from Application Master Failures**

In Hadoop 2.x with YARN, the ApplicationMaster (AM) is responsible for managing the lifecycle of a specific application. If the ApplicationMaster fails, recovery involves several steps:

1. **Detect Failure:**
   * **Heartbeat Monitoring:** The ResourceManager monitors heartbeats from the ApplicationMaster. If it stops receiving heartbeats or if the ApplicationMaster becomes unresponsive, it detects a failure.
2. **Restart ApplicationMaster:**
   * **Retry Mechanism:** YARN automatically restarts the ApplicationMaster on a different node. The failure of the ApplicationMaster triggers the ResourceManager to launch a new instance.
   * **ApplicationMaster Retry Policy:** The number of retries and the time between retries are configurable. By default, the ApplicationMaster is retried a few times before declaring a permanent failure.
3. **Recover Application State:**
   * **State Persistence:** YARN attempts to recover the application state from the last saved checkpoint. This includes tracking the progress of completed and pending tasks.
   * **Task Re-execution:** The new ApplicationMaster retrieves the state information and resumes or re-executes the tasks that were previously in progress. Tasks that were completed before the failure are not re-executed.
4. **Handle Data and Resource Reallocation:**
   * **Resource Reallocation:** The new ApplicationMaster requests resources from the ResourceManager to re-run or continue running tasks.
   * **Data Recovery:** Data that was processed before the failure is retained, and intermediate data is recovered from the data nodes if necessary.
5. **Notify and Update Users:**
   * **Failure Notification:** Users are notified about the application failure and recovery attempts. Logs and status updates are provided to help diagnose issues and understand the recovery process.
6. **Post-Failure Actions:**
   * **Diagnostic Logging:** Analyze logs to determine the cause of the ApplicationMaster failure and prevent future occurrences. Logs include both application-specific logs and system logs.
   * **Configuration Review:** Review and adjust configurations if necessary to mitigate issues that led to the failure. This might include tuning resource settings or adjusting retry policies.

**Q) With neat diagram, show how map reduce jobs are executed in Hadoop environment.**

### **Execution of MapReduce Jobs in Hadoop**

1. **Job Submission:**
   * The user submits a MapReduce job to the Hadoop cluster using the command-line interface (CLI) or through a client application.
2. **Job Initialization:**
   * **JobTracker (Hadoop 1.x) / ResourceManager (Hadoop 2.x/YARN):** Receives the job submission request and creates a job context that includes job configuration, input splits, and other metadata.
3. **Job Scheduling:**
   * **JobTracker (Hadoop 1.x) / ResourceManager (Hadoop 2.x/YARN):** Schedules the job’s tasks (map and reduce) across the cluster nodes. It assigns the Map tasks to available DataNodes that hold the input data and the Reduce tasks to nodes based on resource availability.
4. **Map Task Execution:**
   * **Mapper:** Each Map task processes an input split and applies the map function to produce intermediate key-value pairs. These tasks run on the DataNodes where the input data is stored to reduce data transfer overhead.
   * **Local Disk Storage:** Intermediate key-value pairs are written to local disk as files.
5. **Shuffle and Sort:**
   * **Map Output:** The intermediate key-value pairs are sorted and shuffled based on the keys.
   * **Shuffle Phase:** The framework transfers the sorted map output to the appropriate Reduce tasks. This involves partitioning the data and sending it over the network to the nodes running Reduce tasks.
6. **Reduce Task Execution:**
   * **Reducer:** Each Reduce task processes the intermediate key-value pairs, applies the reduce function, and generates the final output.
   * **Output Storage:** The final output is written to HDFS (Hadoop Distributed File System).
7. **Job Completion:**
   * **JobTracker / ResourceManager:** Monitors the progress of Map and Reduce tasks, handles task retries in case of failures, and manages job completion.
   * **Job Status:** The framework updates the job status and provides completion information to the user.

**Q)Give the responsibilities of a. Application Master b. Application Manager c. Node Manager d. Resource Manager**

### ****Responsibilities in the YARN Architecture****

#### **a. Application Master**

**Responsibilities:**

1. **Application Lifecycle Management:** Manages the lifecycle of a specific application, from submission to completion. This includes requesting resources, monitoring progress, and handling failures.
2. **Resource Request and Negotiation:** Requests resources from the ResourceManager and negotiates resource allocation for the application's tasks.
3. **Task Scheduling:** Schedules the execution of tasks (Map and Reduce tasks) and manages their execution across different nodes.
4. **Application Monitoring:** Monitors the progress and status of tasks, handles task failures, and re-assigns tasks if necessary.
5. **Application State Management:** Manages the state of the application, including tracking completed and pending tasks, and recovering from failures.
6. **Reporting:** Provides status updates and logs about the application's progress and completion status to the ResourceManager and user.

#### **b. Application Manager**

(Note: In Hadoop terminology, there is no specific component called "Application Manager" distinct from "Application Master". The term "Application Manager" might be a confusion with "Application Master". In general YARN architecture, Application Master fulfills the responsibilities related to managing an application's lifecycle.)

#### **c. Node Manager**

**Responsibilities:**

1. **Resource Monitoring:** Monitors the resources (CPU, memory, disk) available on the node and reports them to the ResourceManager.
2. **Container Management:** Manages the lifecycle of containers, including container creation, resource allocation, and execution of tasks within the containers.
3. **Task Execution:** Executes tasks (Map and Reduce tasks) within the containers, according to the instructions from the Application Master.
4. **Health Reporting:** Periodically sends heartbeat signals and resource usage information to the ResourceManager to confirm that the node is active and functioning properly.
5. **Log Aggregation:** Collects and aggregates logs generated by tasks running on the node, and makes them available to the Application Master and users.

#### **d. Resource Manager**

**Responsibilities:**

1. **Resource Allocation:** Manages and allocates cluster resources among various applications based on their requirements and priorities.
2. **Scheduler Management:** Employs different schedulers (e.g., Capacity Scheduler, Fair Scheduler) to manage the allocation of resources to different queues and applications.
3. **Resource Tracking:** Monitors resource usage across the cluster and ensures efficient utilization of resources.
4. **Application Coordination:** Coordinates with Application Masters to ensure that resources are allocated appropriately and that tasks are scheduled for execution.
5. **Failure Management:** Detects and handles node failures or resource-related issues by reallocating resources and tasks as needed.
6. **Job Submission Management:** Accepts job submissions from users, schedules them, and manages their execution through the Application Masters.

With example, explain Hadoop MapReduce process.  
The Hadoop MapReduce process involves several stages where large datasets are processed in parallel across a distributed cluster. Here's a detailed explanation with an example of how the MapReduce process works.

**Example: Word Count**

**Objective:** Count the number of occurrences of each word in a large collection of text files.

**Steps in the Hadoop MapReduce Process**

1. **Job Submission:**
   * **User Input:** The user submits a Word Count job to the Hadoop cluster using the Hadoop CLI or a programmatic interface.
   * **Job Configuration:** The job specifies input data (e.g., text files in HDFS), output location, and the Mapper and Reducer classes.
2. **Job Initialization:**
   * **ResourceManager (YARN):** Receives the job and allocates resources for it.
   * **ApplicationMaster:** Initializes the job by creating a job context, defining input splits, and setting up the Mapper and Reducer tasks.
3. **Map Phase:**
   * **Input Splitting:** The input data is split into smaller chunks (splits). Each split is processed by a separate Mapper.
   * **Mapper Task Execution:** Each Mapper reads a chunk of input data and processes it. For the Word Count example:
     + **Input Data:** Text lines.
     + Mapper Code:

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

private final static IntWritable one = new IntWritable(1);

private Text word = new Text();

public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

String line = value.toString();

StringTokenizer itr = new StringTokenizer(line);

while (itr.hasMoreTokens()) {

word.set(itr.nextToken());

context.write(word, one);

} } }

* **Output:** The Mapper emits intermediate key-value pairs, such as ("word1", 1) and ("word2", 1).

**Shuffle and Sort Phase:**

* **Shuffle:** The intermediate key-value pairs produced by all Mappers are shuffled and sorted by key. This ensures that all occurrences of the same key (word) are grouped together.
* **Sort:** The pairs are sorted by key, preparing them for the Reduce phase.

**Reduce Phase:**

* **Reducer Task Execution:** Each Reducer processes a group of key-value pairs with the same key. For the Word Count example:
  + Reducer Code:

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

private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.get(); }

result.set(sum);

context.write(key, result);

} }

* **Output:** The Reducer emits final key-value pairs with the word and its total count, such as ("word1", 5) and ("word2", 3).

**Job Completion:**

* **Output Storage:** The final output is written to HDFS in the specified output directory.
* **ResourceManager and ApplicationMaster:** Monitor the job’s progress, handle failures, and update the job status. Once the job is completed, the ApplicationMaster reports the results to the user.