01 Hadoop Interview Questions -Kirkyagami 👨‍💻🕵️‍♂️

1. What is *Hadoop* and what are its *core components*?

Apache Hadoop is a robust, open-source platform that facilitates distributed storage and processing of vast datasets across clusters of computers. It provides a cost-effective, powerful, and scalable foundation for Big Data analytics.

Core Components

Hadoop Distributed File System (HDFS)

- **Purpose**: Designed for high-speed access to application data and redundantly storing and managing large volumes of data.
- **Key Features**: Fault tolerance through data replication, high throughput for data access, data integrity, and coherency.
- **HDFS Components**: NameNode (manages the file system namespace and regulates access to files), DataNodes (store and manage data within the file system), Secondary NameNode (performs periodic checkpoints of the namespace).

Yet Another Resource Negotiator (YARN)

- **Purpose**: Serves as a distributed resource management system for allocating computational resources in Hadoop clusters.
- **Key Features**: Allows multiple data processing engines like MapReduce, Spark, and others to run on Hadoop in a shared manner.
- **YARN Components**: ResourceManager (manages and monitors cluster resources), NodeManager (manages resources on individual nodes), ApplicationMaster (coordinates execution of a particular application or job), Containers (virtualized resources where application code runs).

MapReduce

- **Purpose**: A data processing model that processes large data sets across a Hadoop cluster in a distributed and parallel manner.
- **Key Features**: Implements data distribution, data processing, and data aggregation phases.
- **MapReduce Components**: Mapper (processes input data and generates key-value pairs), Reducer (aggregates the key-value pairs generated by the Mappers),

Partitioner (distributes the key-value pairs across Reducers), Combiner (performs local aggregation on the Map output before it's shuffled to the Reducers).

Other Hadoop Ecosystem Components

Hadoop's rich ecosystem comprises tools and frameworks that extend its functionality to various Big Data tasks:

- **Apache Hive**: A data warehouse infrastructure that provides data summarization and ad hoc querying using a SQL-like language called HiveQL. It translates queries to MapReduce jobs.
- **Apache HBase**: A NoSQL database designed to operate on top of HDFS. It's capable of real-time read/write access to Big Data.
- **Apache ZooKeeper**: A centralized service for distributed systems that enables synchronization and group services, such as configuration management and distributed locks.
- **Apache Oozie**: A workflow scheduler to manage Apache Hadoop jobs.
- **Apache Mahout**: A library of scalable machine learning algorithms that can be run on Hadoop. It allows easy implementation of simple Hadoop workflows.
- **Apache Pig**: A platform for analyzing large datasets. It provides a high-level language, Pig Latin, which automates common data manipulation operations.

Hadoop is highly flexible and compatible with a wide array of hardware vendors and cloud service providers, making it a favorite choice for efficient Big Data management and analysis.

2. Explain the concept of a *Hadoop Distributed File System (HDFS)* and its *architecture*.

Hadoop Distributed File System (HDFS) is a distributed, fault-tolerant, and scalable file system designed to reside on commodity hardware. It's part of the core Hadoop ecosystem and provides **reliable** data storage for distributed data processing tasks.

Key Design Principles

- **Rack Awareness**: HDFS is aware of machine racks, enabling optimized data replication.
- **Write Once, Read Many**: It's better suited for applications where files are written once but read multiple times.

- **Data Locality**: HDFS aims to process data on the same nodes where it is stored to minimize network congestion.

High-Level Concepts

- **NameNode**: Serves as the **master node**, keeping track of the file system structure and metadata like permissions and data block locations.
- **DataNode**: Multiple nodes serve as **slaves**, storing the actual data blocks and responding to requests from clients.
- **Client**: Applications that use HDFS. They can be run as a library or a separate utility, interacting with Namenode and Datanodes.

HDFS uses **block storage**, typically with a default block size of 128 MB (configurable) and a **replication mechanism** to ensure data redundancy. By default, it makes three copies of each block.

Data Operations Workflow

1. **File Write**: The client splits the file into blocks and sends them to DataNodes for storage. The NameNode records the blocks' locations.
2. **Block Replication**: DataNodes replicate blocks for redundancy.
3. **File Read**: The client fetches block locations from the NameNode and reads directly from the DataNodes.
4. **File Deletion**: The client requests file deletion from the NameNode, which updates metadata and expedites block deletion on DataNodes.

Redundancy and Fault Tolerance

HDFS employs both **data mirroring** across DataNodes and **block replication** to ensure that each data block is safely stored across the cluster. This approach minimizes data loss due to node failures.

Even if a single node or a few nodes fail, data remains accessible using the available replicas. When a node becomes inaccessible, the NameNode redistributes the blocks stored on that node.

Limitations

- **Latency with Small Files**: Not optimized for high-throughput or low-latency requirements, especially with small files.

- **Primary Consistency**: Ensures only primary consistency, which might not suffice for applications needing strong consistency.

3. How does *MapReduce* programming model work in *Hadoop*?

MapReduce is the foundation of Hadoop's parallel data processing system. Its **two-step methodology** involves mapping data into key-value pairs and then reducing based on those keys.

Key Components

- **Input Data**: Gets divided into smaller chunks for processing.
- **Mapper**: Operates on these data chunks, typically filtering, transforming, and sorting them.
- **Shuffle and Sort**: Collects and organizes mapped data before passing it to the reducers.
- **Reducer**: Processes and aggregates the data, finalizing the output.

Workflow

1. **Data Splitting**: Hadoop splits data files into manageable blocks (default: 128MB).
2. **Mapper Input**: Each block is processed by multiple mappers, creating intermediate key-value pairs.
3. **Data Shuffling**: Key-value pairs are shuffled, sorted, and grouped to ensure data of the same key goes to the same reducer.
4. **Reducer Input**: Reducers handle key groups separately, computing each group's output.

Code Example: Word Count

Here is the mapper code:

```
from mrjob.job import MRJob

class WordCount(MRJob):
    def mapper(self, _, line):
        words = line.split()
        for word in words:
            yield word, 1
```

```
if __name__ == '__main__':
    WordCount.run()
```

And the corresponding reducer:

```
from mrjob.job import MRJob

class WordCount(MRJob):
    def reducer(self, key, values):
        yield key, sum(values)

if __name__ == '__main__':
    WordCount.run()
```

Under the Hood: The Hadoop Framework

- **Job Tracker**: Manages all jobs in the Hadoop cluster.
- **Task Tracker**: Each node runs a task tracker to execute Map and Reduce tasks.
- **Data Locality**: Hadoop tries to process data on the same node where it's stored, minimizing data transfer over the network.

4. What is *YARN*, and how does it improve *Hadoop's resource management*?

YARN, short for "Yet Another Resource Negotiator," is Hadoop's latest **resource management** platform. YARN represents a crucial advancement over its predecessor, the TaskTracker/JobTracker system, by offering a more flexible, scalable, and precise resource assignment and management.

YARN Architecture

YARN has a **distributed architecture** comprising three main components:

1. **Resource Manager (RM)**: The primary master daemon. It oversees resource allocation and job scheduling across the Hadoop cluster. It's composed of two sub-components: the Scheduler and the ApplicationManager.
2. **Node Manager (NM)**: Per-cluster-node slave daemon. This manager is responsible for monitoring resources on individual cluster nodes and for being responsive to Resource Manager requests.

3. **Application Master (AM)**: A per-application framework-specific master. It orchestrates the task execution and resource management for a particular application.

Key Components & Task Flow

- **Resource Manager (RM)**: This represents the global resource scheduler. It's in charge of handling job submissions, negotiates resources, and works to avoid over-commitment of resources.
- **Node Manager (NM)**: This daemon is present on all nodes within the cluster, monitoring resource usage (CPU, memory, etc.) and reporting the same to the Resource Manager.
- **Application Master (AM)**: A specialized resource manager for a single application. It handles the task of requesting resources from the Resource Manager and processing them from Node Managers.

The typical resource management flow in a YARN setup involves these general steps:

1. **Job Submission and Initialization**: The client submits the job to the Resource Manager. The Resource Manager deploys the Application Master specific to the application.
2. **Resource Acquisition**: The Application Master, after initialization, requests the resources it needs from the Resource Manager.
3. **Task Execution**: The Application Master directs the Node Manager to launch containers, which are the entities where the actual tasks (mappers or reducers, in the case of MapReduce) are run.
4. **Client Communication**: The Resource Manager and Application Master constantly update the client about the job status.

Container Flexibility

The main distinguishing feature of YARN is its use of the "container concept" for allocating resources. A **container** is a unified unit of CPU, memory, storage, and network resources, which is what YARN allocates to an application or a task.

The flexibility of containers sets YARN apart, as it allows systems like MapReduce and other applications to request the specific resources they need (e.g., CPU cores or memory). This is in contrast to the initial approach, where MapReduce tasks were restricted to fixed-size resource slots.

YARN's Advantages

1. **Improved Task Scheduling**: YARN enhances task scheduling by allowing applications to request specific resources and schedule tasks promptly.
2. **Resource Management Flexibility**: With containers, applications are no longer restricted to a predetermined number of slots for resources. They can tailor requests based on their unique requirements.
3. **Multi-Tenancy Capability**: YARN supports the simultaneous execution of multiple workloads, making it possible for different applications to exist and operate within the same Hadoop cluster.
4. **Enhanced Ecosystem Integration**: Both new and existing processing frameworks can seamlessly integrate with the YARN infrastructure, adding further functionality to the Hadoop ecosystem.
5. **Simplified Cluster Maintenance**: Node Manager and Application Master roles permit easy handling of resources and applications at the cluster node level and within applications, respectively.

5. Explain the role of the *Namenode* and *Datanode* in *HDFS*.

The **Namenode** and **Datanodes** form the backbone of HDFS, **managing metadata and data storage**, respectively.

Namenode

The **Namenode** primarily functions as the **metadata repository** for the file system. It maintains file-to-block mapping, file permissions, and the hierarchical directory structure.

The Namenode's key responsibilities include:

1. **Metadata Storage**: It stores metadata such as block IDs, permissions, and modification times.
2. **Namespace Operations**: It manages the file system namespace, including directory and file creation, deletion, and renaming. It also handles attribute modifications.
3. **Access Control**: The Namenode enforces access permissions and authenticates clients.

Checkpointing is the process of merging edit logs with the filesystem metadata on disk. Least frequently modified data is checked less often. This ensures that the namenode remains responsive.

Datanodes

Datanodes are responsible for storing and managing the actual data in HDFS. They execute file I/O operations requested by clients and the namenode.

The main functions of Datanodes are:

1. Block Operations: They perform read, write, and delete operations on the data blocks, and report block health and integrity to the Namenode.
2. Heartbeats: Datanodes send periodic heartbeats to the Namenode, indicating their operational status.
3. Block Reports: They periodically report the list of blocks that they are hosting to the Namenode.

In case of block loss or corruption, the Datanode is responsible for replicating or recovering the block.

Client-Server Model

HDFS employs a client-server model for read and write operations.

- Read: The client requests data from Datanodes, which operate directly on the files and send blocks to the client.
- Write: The client communicates with the Namenode for file metadata and then directly with Datanodes.

High Availability in Namenode

In a distributed environment, keeping a single point of failure is not desirable, which is why there is a mechanism for High Availability with multiple active namenodes (machines).

- Active Namenode: Serves read and write requests.
- Standby Namenode: Maintains enough state information to take over in case of active Namenode failure.

This is achieved via a mechanism called Quorum Journal, which ensures that both Namenodes see the same transactions.

Data Replication for Fault Tolerance

HDFS stores multiple copies of data blocks across Datanodes to ensure high data availability and reliability. The system is configurable to determine the replication level. Usually, the default number of replicas is three.

When a modification happens, the client asks the Namenode for a list of Datanodes to store the new data. Once the writing Datanodes confirm, the operation is considered successful.

6. What is a *Rack Awareness algorithm* in *HDFS*, and why is it *important*?

In Hadoop Distributed File System (HDFS), the concept of Rack Awareness refers to HDFS's ability to understand the network topology and the physical location of its nodes. The primary goal of Rack Awareness is to optimize data storage reliability and data transfer efficiency (reducing inter-rack data traffic).

Multi-level Topology Knowledge

HDFS leverages a two-level topology consisting of `Racks` and `Nodes`:

1. Racks: Represent physical groups or locations of nodes, often within the same data center. They serve as a defining structure that organizes and groups nodes.
2. Nodes: Indicate individual machines or servers that store data and comprise Hadoop clusters.

By utilizing this generated network, Hadoop actively places blocks of data across different racks to achieve fault tolerance and performance

Rack Awareness in Hadoop's Transfer Modes

HDFS nodes optimize data transfer during block movement, particularly during operations such as rebalancing, replication, and block recovery. This results in a reduction in unnecessary and potentially costly inter-rack data traffic.

Why Is Rack Awareness Important?

1. Fault Tolerance:
   - Minimizes probability of losing data due to rack-wide failures.
   - Guarantees at least one replica of a block exists on an adjacent rack (except in the case of just one rack).
2. Network Efficiency:
   - Reduces bottlenecks and network congestion because data can be read closer to the requesting node, from the local rack where possible.

- Lessens the risk of latency and data transfer overhead caused by inter-rack nodes.

3. **Inter-Rack Balancing**:
    - Distributes blocks equitably across racks, ensuring efficient use of storage and resources.
    - Facilitates uniform data access and load distribution.

Key Algorithms

HDFS uses **Rack Awareness Algorithms** to control how data placement and data transfer operations work across nodes and racks. Latter versions of HDFS come with dynamic rack discovery techniques to handle more complex and evolving data center configurations.

The default rack allocation strategy employs a simple *Round-Robin* approach to distribute blocks among the data cluster's racks, offering a fundamental yet efficient methodology for data storage and transfer operations. Different strategies, whether static or dynamic, including manual rack planning, might be more fitting for specific scenarios or requirements.

Rack Awareness Configuration

You can set and modify the Rack Awareness policies and configurations in your Hadoop setup using HDFS-specific configuration files, such as `hdfs-site.xml`.

For instance, you can make use of the following Rack Awareness configurations to:

- Define the number of replicas managed within HDFS.
- Outline the number of racks and their locations within your datacenter.
- Assign nodes to particular racks and guarantee the right configuration of the clusters.
- Allocate nodes fine-tuned to specific racks fitting your rack-based storage or networking needs.

7. What are some of the *characteristics* that differentiate *Hadoop* from traditional *RDBMS*?

Both Hadoop and relational database management systems (RDBMS) are used for **data management**. However, they differ in the types of data they handle and their underlying data models.

Key Characteristics

Data Model

- **Hadoop**: Utilizes a schema-on-read model, allowing for more flexible handling of unstructured or semi-structured data.
- **RDBMS**: Implements a schema-on-write model, requiring data to adhere to pre-defined structures before ingestion.

Data Storage and Management

- **Hadoop**: Distributes and stores data across clusters in a scalable and fault-tolerant manner. It's suitable for managing large volumes of both structured and unstructured data.
- **RDBMS**: Stores data in a structured, tabular format across different tables. While data scaling can be vertical to an extent, horizontal scaling can be complex relative to Hadoop.

Data Processing Paradigms

- **Hadoop**: Primarily employs batch processing with tools such as MapReduce. However, with advancements such as YARN and Spark, it also supports real-time and interactive processing.
- **RDBMS**: Designed for transactional processing and predominantly utilizes online transaction processing (OLTP) systems. With the advent of in-memory engines and modern architectures, some RDBMS now support real-time analytics as well.

Overall Flexibility

- **Hadoop**: Offers greater flexibility in terms of data analytics workflows. Tools such as Hive, HBase, and Impala allow for various types of data access and query processing.
- **RDBMS**: Designed with a specific structure that caters well to certain types of read and write operations. Changes to schema can be challenging, and reporting on unstructured or semi-structured data may be limited.

Primary Use Cases

- **Hadoop**: Ideal for processing high volumes of unstructured or semi-structured data, and is often the go-to choice for big data analytics tasks.

- **RDBMS**: Suited for handling structured data, is a core component of enterprise systems like customer relationship management (CRM), and is commonly used for traditional point-of-sale (POS) systems.

Data Duplication and Consistency

- **Hadoop**: Offers eventual consistency and, in some cases, the concept of "read-your-write" consistency. Duplicating data across nodes enables parallel processing and fault tolerance.
- **RDBMS**: Emphasizes immediate consistency to ensure data integrity. While replication and failover can be supported, they often require sophisticated setups and can involve some latency.