02 HDFS - NikhilSharma X KirkYagami

Hadoop Distributed File System (HDFS)

Table of Contents

1. Introduction to File Systems

1.1. What is a File System?

A file system is a fundamental component of an operating system that provides a structured way to store, organize, and retrieve data on storage devices. It acts as an abstraction layer, enabling users and applications to interact with data using logical names (files and directories) rather than physical disk locations. Without a file system, a storage device would simply be a large, undifferentiated block of data, making it impossible to distinguish one piece of information from another.

1.2. Requirements of a File System

Primary Requirements (for any file system):

- **Data Organization:** Provides a hierarchical structure (directories/folders) for logical grouping and efficient navigation of files.

- **Metadata Management:** Stores essential information about files and directories, such as file names, sizes, creation/modification timestamps, ownership, and permissions.
- **Access Control:** Implements security mechanisms to control user and process access to files and directories, based on permissions (read, write, execute).
- **Data Integrity:** Ensures the consistency and reliability of stored data, preventing corruption and loss. This often involves mechanisms like journaling.
- **Performance:** Aims for fast and efficient read/write operations to minimize latency for user applications.
- **Space Management:** Efficiently allocates and deallocates storage space on the underlying hardware, preventing fragmentation and maximizing utilization.

Additional Requirements for Distributed Systems (like HDFS):

- **Fault Tolerance:** The ability to continue operating normally even when individual hardware components (e.g., servers, disks) fail, without data loss or significant service interruption.
- **High Availability:** Minimizing downtime and ensuring continuous access to data and services, even during failures or maintenance.
- **Concurrent Access:** Supporting multiple users or applications simultaneously accessing and processing data.
- **Network Efficiency:** Optimizing data transfer across a network to reduce bottlenecks and improve overall system throughput, especially crucial for large datasets.
- **Scalability:** The ability to easily expand storage capacity and processing power by adding more nodes to the system as data volumes grow.

1.3. How File Systems Work

File systems perform a variety of operations to manage data:

- **File Creation:** When a new file is created, the file system allocates space on the storage device and creates metadata entries to describe the file.
- **File Access (Read/Write):** Applications interact with files through system calls (e.g., `open()`, `read()`, `write()`, `close()`). The file system translates these logical requests into physical disk operations.
- **File Organization:** Manages the directory structure, linking file names to their corresponding data blocks.
- **Space Management:** Tracks free and used blocks on the storage device, allocating new blocks for writes and reclaiming old blocks when files are deleted.
- **Caching:** Utilizes memory (RAM) to store frequently accessed data and metadata, significantly speeding up subsequent access.

1.4. Types of File Systems

File systems can be broadly categorized based on their architecture and typical use cases:

- **Local File Systems:** Designed for storage on a single machine.
    - **NTFS (New Technology File System - Windows):** Known for advanced features like journaling, compression, encryption, and robust access control.
    - **ext4 (Fourth Extended File System - Linux):** A popular journaling file system for Linux, offering large file support, improved performance, and reliability.
    - **APFS (Apple File System - macOS):** Optimized for solid-state drives (SSDs), offering features like snapshots, strong encryption, and space sharing.
    - **FAT32 (File Allocation Table 32 - Cross-platform):** An older, simpler file system with limitations on file and volume sizes, widely compatible across various operating systems.
- **Network File Systems (NFS):** Allow access to files over a network as if they were local.
    - **NFS (Network File System):** A distributed file system protocol widely used in Unix/Linux environments for sharing files across a network.
    - **SMB/CIFS (Server Message Block/Common Internet File System):** The primary protocol for file sharing in Windows environments.
    - **FTP/SFTP (File Transfer Protocol/Secure File Transfer Protocol):** Protocols primarily used for transferring files between computers, though they can facilitate basic remote file access.
- **Distributed File Systems:** Designed to store and manage data across a cluster of interconnected machines. These are crucial for large-scale data processing.
    - **GFS (Google File System):** Google's proprietary distributed file system, a foundational inspiration for HDFS.
    - **HDFS (Hadoop Distributed File System):** An open-source, highly fault-tolerant, and scalable distributed file system designed for big data applications.
    - **Ceph:** A highly scalable distributed storage system that provides object, block, and file storage interfaces.
    - **GlusterFS:** A scale-out network-attached storage file system, aggregating disparate storage resources into a single namespace.

---

2. Why HDFS? Addressing Traditional File System Limitations

Traditional file systems like FAT32, NTFS, ext3, ext4, HFS, and HFS+ are highly effective for single-machine environments but face significant limitations when

dealing with the scale and characteristics of "big data." HDFS was specifically designed to overcome these challenges.
Let's look at the limitations of traditional file systems, as highlighted in the provided image and how HDFS addresses them:

File System	File Limit	Volume Limit
Microsoft		
FAT32	4 GB	32 GB
NTFS	16 EB	16 EB
Apple		
HFS	2 GB	2 TB
HFS+	8 EB	8 EB
Linux		
ext3	2 TB	32 TB
ext4	16 TB	1 EB
XFS	8 EB	8 EB

Note: EB = Exabyte (1018 Bytes), TB = Terabyte (1012 Bytes), GB = Gigabyte (109 Bytes)

Why HDFS? (Addressing the "Why another file system?"):

- **Scalability (Horizontal vs. Vertical):**
  - Traditional file systems are designed for single-machine storage (vertical scaling), which quickly becomes a bottleneck as data grows into petabytes or exabytes.
  - **HDFS is designed for horizontal scalability**, distributing data across thousands of commodity machines. This allows for virtually unlimited storage and processing capacity simply by adding more nodes.
- **Fault Tolerance:**
  - In local file systems (ext4, XFS, NTFS), the failure of a single disk or server can lead to catastrophic data loss.
  - **HDFS inherently provides fault tolerance through data replication.** It stores multiple copies (default 3) of each data block on different nodes and racks, ensuring that even if several nodes fail, the data remains accessible and intact.
- **Data Locality:**
  - Traditional file systems do not consider where computation should occur relative to data. Moving massive datasets over a network for processing is inefficient.

  - **HDFS optimizes for data locality.** It brings the computation to where the data resides, minimizing network congestion and significantly improving processing efficiency for large analytical workloads.
- **Throughput Optimization:**
  - Local file systems are optimized for low-latency, random access patterns, which are crucial for interactive applications and operating system responsiveness.
  - **HDFS is optimized for high-throughput sequential access**, making it ideal for batch processing of very large datasets where the entire dataset is read or written sequentially.
- **Support for Large Files:**
  - While modern local file systems (NTFS, ext4, XFS, HFS+) can handle large individual files (up to exabytes), they become inefficient with a massive number of very large files spread across multiple physical disks and servers.
  - **HDFS is specifically engineered to store and manage files ranging from gigabytes to terabytes and even petabytes seamlessly across a distributed cluster.**
- **Streaming Data Access:**
  - Traditional file systems often focus on random access, where small chunks of data are accessed non-sequentially. This is inefficient for analytical tasks that involve reading entire datasets.
  - **HDFS is optimized for streaming data access**, making it perfectly suited for applications that perform sequential scans over large datasets, which is common in big data analytics.
- **Integration with Big Data Ecosystem:**
  - HDFS is the foundational storage layer for the entire Apache Hadoop ecosystem. It seamlessly integrates with other big data tools like MapReduce, Hive, Spark, HBase, and more, forming a complete big data processing pipeline.
  - Traditional file systems lack this inherent integration, making them less suitable for complex, multi-component big data solutions.
- **Economical Storage (Commodity Hardware):**
  - HDFS is designed to run on clusters of inexpensive, commodity hardware. This significantly reduces the cost of storing and processing massive amounts of data compared to relying on specialized, high-cost enterprise storage solutions.
- **Handling of Metadata (Dedicated NameNode):**
  - In traditional file systems, metadata management can become a bottleneck as the number of files scales.

- **HDFS uses a dedicated NameNode (master node) to manage all metadata efficiently in memory**, allowing it to handle a very large number of files and directories even in immense clusters.

In essence, HDFS provides a robust, scalable, and cost-effective solution specifically tailored for the unique demands of big data storage and processing, overcoming the inherent limitations of file systems designed for single-machine operation.

3. What is HDFS?

3.1. Definition

The **Hadoop Distributed File System (HDFS)** is a highly fault-tolerant, scalable, and high-throughput distributed file system designed to store and manage very large files (gigabytes to terabytes and beyond) across clusters of commodity hardware. It is the primary storage component of the Apache Hadoop framework.

3.2. Key Characteristics

HDFS is built upon a set of design principles that make it suitable for big data workloads:

Design Principles:

- **Hardware Failure is Normal:** HDFS is built with the explicit assumption that hardware components will fail. It includes mechanisms to detect and automatically recover from failures without data loss.
- **Streaming Data Access:** HDFS is optimized for "write-once, read-many" access patterns. It excels at batch processing large datasets where data is read sequentially, rather than low-latency random reads.
- **Large Datasets:** Designed to efficiently store and process files ranging from hundreds of megabytes to terabytes or even petabytes.
- **Simple Coherency Model:** Follows a "write-once, read-many" model. Once a file is written and closed, it cannot be modified. This simplifies consistency management and optimizes for high throughput.
- **Moving Computation is Cheaper than Moving Data:** HDFS promotes the concept of data locality. Instead of moving large datasets to computation nodes, it brings the computation (e.g., MapReduce tasks) to the nodes where the data is already stored, reducing network overhead.

Core Features:

- **Fault Tolerance:** Achieved primarily through data replication across multiple DataNodes and automatic recovery mechanisms.
- **High Throughput:** Enables rapid data processing by allowing parallel access to data blocks spread across many nodes.

- **Scalability:** Can scale horizontally by simply adding more commodity servers to the cluster, increasing both storage capacity and processing power.
- **Cost-Effective:** Runs on inexpensive, off-the-shelf hardware, significantly reducing infrastructure costs for large-scale data storage.
- **Data Locality:** Facilitates efficient processing by placing computation near the data, minimizing network traffic.

4. Hadoop vs HDFS

It's important to understand that HDFS is a core component *within* the broader Hadoop ecosystem.

4.1. Understanding the Relationship

Aspect	Hadoop	HDFS
Scope	A complete big data processing framework and ecosystem.	The distributed file system component of Hadoop.
Components	HDFS + MapReduce + YARN + Hadoop Common + various ecosystem projects (Hive, Spark, HBase, etc.)	Storage layer only.
Function	Provides a framework for distributed storage and processing of massive datasets.	Provides robust, scalable, and fault-tolerant distributed storage.
Dependencies	Includes HDFS as its underlying storage layer.	A fundamental part of the Hadoop ecosystem.

4.2. Hadoop Ecosystem Components

The Hadoop ecosystem is a rich collection of open-source projects designed to handle various aspects of big data:

Core Components:

- **HDFS (Hadoop Distributed File System):** The distributed storage layer.
- **YARN (Yet Another Resource Negotiator):** The resource management and job scheduling framework that allocates resources (CPU, memory) to applications running on the cluster.
- **MapReduce:** A programming model and processing framework for distributed batch processing of large datasets. (While still a core component, other processing engines like Spark are often preferred for new development due to their versatility and speed).
- **Hadoop Common:** A set of common utilities and libraries that support the other Hadoop modules.

Extended Ecosystem (examples):

- **Hive:** A data warehousing system built on top of Hadoop that allows users to query large datasets using an SQL-like language (HiveQL).
- **Pig:** A high-level platform for analyzing large datasets, offering a scripting language (Pig Latin) to express data analysis programs.
- **HBase:** A non-relational (NoSQL) distributed column-oriented database built on HDFS, providing real-time read/write access to large datasets.
- **Spark:** A fast and general-purpose cluster computing system that provides high-level APIs in Java, Scala, Python, and R, capable of performing batch processing, interactive queries, streaming, and machine learning.
- **Kafka:** A distributed streaming platform used for building real-time data pipelines and streaming applications.
- **ZooKeeper:** A centralized service for maintaining configuration information, naming, providing distributed synchronization, and group services. Used by Hadoop for High Availability.

5. Evolution of HDFS

HDFS has continuously evolved to meet the growing demands of big data, improving performance, fault tolerance, and manageability.

5.1. Version History and Key Changes

- **Hadoop 1.x (Initial Release - ~2011-2013):**
  - **Default Block Size:** 64MB.
  - **NameNode:** Single NameNode, which was a **Single Point of Failure (SPOF)**. If the NameNode went down, the entire HDFS cluster became unavailable.
  - **Maximum Cluster Size:** Limited to approximately 4,000 nodes.
  - **File Limit:** Scaled to around 100 million files per cluster due to NameNode memory limitations.
  - **Job Management:** `JobTracker` and `TaskTracker` were responsible for both resource management and job execution.
- **Hadoop 2.x (Major Overhaul - ~2013-2017):**
  - **Default Block Size:** Increased to 128MB. This was a significant change to optimize for larger file sizes and sequential I/O.
  - **High Availability (HA) NameNode:** Introduced an Active/Standby NameNode configuration, eliminating the SPOF.
  - **YARN Introduction:** Decoupled resource management (`ResourceManager`) and job scheduling (`ApplicationMaster`) from the MapReduce framework, allowing other processing engines (like Spark) to run on Hadoop.
  - **Maximum Cluster Size:** Significantly increased to around 10,000 nodes.
  - **Improved Scalability and Performance:** Overall architectural improvements led to better scalability and efficiency.

    - **HDFS Federation:** Enabled multiple independent NameNodes to manage different parts of the HDFS namespace, further improving scalability for extremely large numbers of files.
- **Hadoop 3.x (Continued Refinements - ~2017-Present):**
    - **Default Block Size:** Maintained at 128MB.
    - **Erasure Coding (EC) Support:** Introduced as an alternative to traditional replication for storage efficiency. EC can achieve the same level of fault tolerance with significantly less storage overhead (e.g., 1.5x vs 3x).
    - **More than 2 NameNodes in HA setup:** Allowed for a greater number of standby NameNodes for even higher resilience.
    - **Improved Container Management:** Enhancements to YARN for more efficient resource utilization.
    - **Enhanced Security Features:** Continued focus on security with more robust authentication and authorization mechanisms.
    - **Better Cloud Integration:** Improvements for deploying and managing Hadoop clusters in cloud environments.

5.2. Block Size Evolution Impact

The increase in the default HDFS block size from 64MB to 128MB (and sometimes larger, depending on the use case) was a strategic decision with several important implications:

Why Block Size Increased:

- **Reduced Metadata Overhead:** Each block requires metadata to be stored in the NameNode's memory. Larger block sizes mean fewer blocks for a given file, which translates to less memory consumption on the NameNode, allowing it to manage a greater number of files.
- **Better Sequential I/O Performance:** For large sequential reads, processing larger chunks of data at once improves throughput by reducing the overhead of starting and stopping I/O operations.
- **Improved Network Efficiency:** Fewer, larger blocks mean fewer individual network connections and handshakes are required to transfer a file, reducing network overhead.
- **Optimal for Batch Processing:** HDFS is designed for batch processing where entire files are often read. Larger blocks align well with this pattern, as the cost of seeking to a block is amortized over a larger data transfer.

Trade-offs of Large Block Size:

- **Inefficiency for Small Files:** If a file is smaller than the HDFS block size (e.g., a 1MB file stored with a 128MB block size), it still occupies an entire 128MB block on disk (though only 1MB of actual data is stored). This leads to wasted storage space and increased NameNode metadata overhead for many small files. This is known as the "Small Files Problem."

- **Less Efficient for Random Access:** While HDFS is optimized for sequential access, if an application needs to randomly access small portions of a very large file, larger block sizes can be less efficient as it might still need to read an entire large block to get a small piece of data.
- **Processing Granularity:** MapReduce tasks typically operate on block boundaries. Larger blocks mean fewer Map tasks for a given dataset, which might sometimes lead to less fine-grained parallelism if the dataset is not extremely large.

---

6. HDFS Architecture

HDFS operates on a classic **master-slave architecture**, which provides clear separation of concerns and facilitates scalability and fault tolerance.

6.1. Master-Slave Architecture

```
                     HDFS Cluster
                          |
             +------------+------------+
             |                         |
         NameNode                 DataNodes
        (Master)                  (Slaves)
             |                         |
     +-------+-------+         +-------+-------+
     |               |         |       |       |
   FSImage        EditLog    DataNode1 DataNode2 ... DataNodeN
                            (Blocks)  (Blocks)      (Blocks)
```

In this architecture:

- The **NameNode** is the central authority, managing the file system metadata and coordinating data storage. There is typically one active NameNode per cluster (with a standby for High Availability).
- **DataNodes** are the worker nodes, responsible for storing the actual data blocks and serving read/write requests. There are usually many DataNodes in a cluster.

6.2. NameNode (Master Node)

The NameNode is the brain of the HDFS cluster. It's a single point of truth for all metadata.

Primary Responsibilities:

- **Metadata Management:** Maintains the entire file system namespace (the directory tree and file metadata like permissions, timestamps, block allocation).

It doesn't store the actual data, only the pointers to where the data blocks are located on DataNodes.

- **Block Management:** Knows which DataNode stores which blocks for a given file, as well as the replication factor for each block.
- **Client Coordination:** Handles all client requests for file operations (opening, closing, renaming, deleting files, etc.).
- **DataNode Monitoring:** Regularly receives heartbeats and block reports from DataNodes to monitor their health and block inventory.
- **Access Control:** Enforces file permissions and security settings.

Key Components Stored and Managed by NameNode:

- **FSImage (File System Image):**
  - A complete, persistent snapshot of the HDFS namespace and block mapping.
  - Contains the entire directory tree, file and directory permissions, and the mapping of files to their respective blocks.
  - Stored on the NameNode's local disk.
  - Loaded into the NameNode's memory at startup, allowing for very fast metadata operations.
- **EditLog:**
  - A transaction log that records every change made to the file system namespace (e.g., file creation, deletion, rename, block addition).
  - All modifications are first written to the EditLog, ensuring that no changes are lost in case of a NameNode crash before they are persisted to FSImage.
  - Periodically, the NameNode (or a Secondary NameNode/JournalNode in HA setups) merges the EditLog with the FSImage to create a new, updated FSImage.
- **Block Reports:**
  - Periodic reports sent by DataNodes to the NameNode.
  - Each DataNode sends a list of all data blocks it currently stores.
  - The NameNode uses these reports to maintain an accurate and up-to-date mapping of blocks to DataNodes, detect missing blocks, and manage replica placement.

6.3. DataNode (Slave Nodes)

DataNodes are the workhorses of HDFS. They are responsible for storing the actual data.

Primary Responsibilities:

- **Data Storage:** Stores HDFS data blocks on its local file system (e.g., ext4, XFS).
- **Block Operations:** Performs low-level read, write, create, replicate, and delete operations on data blocks as instructed by the NameNode or directly by clients

(for data transfer).
- **Heartbeat Reporting:** Sends regular "heartbeat" messages (default: every 3 seconds) to the NameNode to signal its liveness and operational status.
- **Block Reporting:** Periodically sends "block reports" (default: every 6 hours) to the NameNode, listing all the data blocks it stores.
- **Direct Client Interaction:** After receiving block locations from the NameNode, clients interact directly with DataNodes to read and write data.

Key Features:
- Runs on commodity hardware, making HDFS cost-effective.
- Can be dynamically added to or removed from the cluster without stopping the entire system.
- If a DataNode stops sending heartbeats for a configurable period (default 10 minutes), the NameNode considers it dead and initiates re-replication of its blocks.

6.4. Blocks in HDFS

HDFS divides large files into smaller, fixed-size chunks called blocks.

Block Characteristics:
- **Default Size:** 128MB (configurable). This is significantly larger than typical file system blocks (e.g., 4KB or 8KB).
- **Replication:** Each block is replicated multiple times across different DataNodes to provide fault tolerance. The default replication factor is 3.
- **Distribution:** Blocks of a single file are distributed across multiple DataNodes throughout the cluster.
- **Checksums:** Each block has a checksum associated with it to verify data integrity during read operations. If a checksum mismatch is detected, the client will try to read a replica from another DataNode.
- **Immutable:** Once a block is written and closed, it cannot be modified. Any changes to a file result in new blocks being appended, or existing blocks being replaced with new ones (effectively a new version of the file).

Block Advantages:
- **Fault Tolerance:** Multiple copies of each block ensure that data remains available even if a DataNode fails.
- **Parallelism:** Since a large file is broken into many independent blocks, multiple clients or processing tasks can read and process different parts of the file simultaneously, leading to high throughput.
- **Load Distribution:** Blocks are spread across the cluster, balancing the storage and processing load across DataNodes.
- **Storage Efficiency for Large Files:** Files larger than the block size do not need to occupy contiguous space on a single disk or node, making storage management

flexible and efficient for very large files.

7. Rack Awareness

7.1. Concept and Importance

Rack Awareness is a crucial feature in HDFS that allows the NameNode to understand the physical network topology of the cluster. It knows which DataNodes are located on which network racks, which helps optimize data placement for both fault tolerance and network efficiency.

The primary goals of rack awareness are:

- **Improved Fault Tolerance:** Distributing replicas across different racks ensures that a rack-level failure (e.g., power outage to a rack, switch failure) does not lead to data loss or unavailability.
- **Optimized Network Traffic:** By minimizing cross-rack data transfers, especially during write operations, rack awareness reduces network congestion and improves overall performance.

7.2. Network Topology

```
                     HDFS Cluster
                          |
               +----------+----------+
               |                     |
            Rack 1                Rack 2
     (Top-of-Rack Switch)  (Top-of-Rack Switch)
               |                     |
       +-------+-------+     +-------+-------+
       |       |       |     |       |       |
 DataNode1 DataNode2 ... DataNodeX DataNodeY ...
```

In a typical data center, servers (DataNodes) are grouped into racks, and each rack has its own network switch (Top-of-Rack switch). Racks are then connected to a core network switch. Communication within a rack is generally faster and cheaper than communication between racks.

7.3. Rack-Aware Replica Placement

HDFS leverages rack awareness to implement an optimized replica placement policy, particularly for the default replication factor of 3. This policy aims to maximize fault tolerance while minimizing inter-rack network traffic.

Default Strategy (Replication Factor = 3):

1. **First Replica:**

- If the client writing the data is also a DataNode, the first replica is placed on that local DataNode.
- Otherwise, the first replica is placed on a randomly chosen DataNode.

2. **Second Replica:** Placed on a different DataNode within the *same rack* as the first replica.
3. **Third Replica:** Placed on a DataNode in a *different rack* from the first two replicas.

Benefits of this Strategy:

- **Fault Tolerance:**
  - **Node-level failure:** If a DataNode fails, two other replicas exist on different nodes.
  - **Rack-level failure:** If an entire rack fails (e.g., switch failure, power outage), two replicas of the data still exist on a different rack, ensuring data availability.
- **Network Efficiency:**
  - Writes: Only one block needs to traverse a cross-rack link to reach the third replica. The other two stay within the same rack.
  - Reads: Clients prefer to read from replicas on their local rack (if applicable) or from a DataNode close to them, reducing cross-rack network traffic.
- **Load Distribution:** Spreads blocks across racks, balancing storage and I/O load.

Distance Calculation (simplified, conceptual):

HDFS uses a network distance metric, where a lower number indicates closer proximity:

- **Distance = 0:** Between a node and itself.
- **Distance = 2:** Between two nodes on the same rack (data has to go up and down the rack switch).
- **Distance = 4:** Between two nodes on different racks but within the same cluster (data has to go up one rack switch, across the core switch, and down another rack switch).
- **Distance = 6 (or more):** Between nodes in different data centers/clusters.

7.4. Impact on Operations

- **Read Operations:**
  - When a client requests to read a block, the NameNode provides a list of DataNodes holding replicas, ordered by their network proximity to the client.
  - The client will try to read from the closest replica first (e.g., on the same node, then same rack). This reduces network latency and improves read performance.

- If a DataNode fails during a read, the client automatically switches to the next closest healthy replica.

- **Write Operations:**
  - The rack-aware placement strategy is applied when new blocks are written to ensure replicas are distributed optimally.
  - This strategy balances the need for high fault tolerance with minimizing the network overhead of replication, especially for cross-rack traffic.

8. HDFS Operations

Understanding how HDFS handles read and write operations is fundamental to grasping its architecture and performance characteristics.

8.1. Read Operation Flow

Reading a file in HDFS involves coordination between the client, NameNode, and DataNodes:

1. **Client Request:** An application or user initiates a request to read a file from HDFS.
2. **Metadata Query (Client to NameNode):** The HDFS client first contacts the NameNode, providing the path of the file it wishes to read.
3. **Location Response (NameNode to Client):** The NameNode, using its metadata (FSImage and EditLog), determines the file's block distribution. It returns to the client a list of all blocks that make up the file, along with the locations (DataNode IPs and ports) of all replicas for each block. The NameNode also provides this list ordered by network proximity to the client (due to rack awareness).
4. **Direct Data Access (Client to DataNodes):** The client, now armed with the block locations, connects directly to the DataNodes that hold the closest replicas for each block.
5. **Parallel Reading (DataNodes to Client):** The client reads blocks in parallel from multiple DataNodes, maximizing throughput. For each block, the client verifies its integrity using a checksum.
6. **File Assembly (Client):** As the blocks are streamed to the client, it reconstructs the original file from these smaller pieces.

Failure Handling During Read:

- **Block Corruption:** If a checksum validation fails for a block, indicating corruption, the client marks that replica as bad and automatically requests the same block from a different DataNode storing a healthy replica.
- **DataNode Failure:** If a DataNode becomes unresponsive during a read, the client transparently switches to another DataNode that holds a replica of the

same block from the list provided by the NameNode.

- **Network Issues:** Automatic retries with timeouts are implemented to handle transient network problems.

8.2. Write Operation Flow

Writing a file to HDFS is a more complex process due to the need for replication and consistency:

1. **Write Request (Client to NameNode):** The HDFS client initiates a request to create a new file and write data to it.
2. **Permission Check (NameNode):** The NameNode first performs a permission check to ensure the client has the necessary authorization to create the file.
3. **Block Allocation (NameNode to Client):** If permissions are granted, the NameNode determines the optimal DataNodes for the first block of the file, considering factors like replication factor (default 3), rack awareness, and DataNode capacity. It returns a list of target DataNodes to the client.
4. **Pipeline Setup (Client to DataNode1):** The client begins writing the first block of data. It breaks the data into small packets and streams them to the first DataNode in the chosen pipeline.
5. **Pipelined Replication (DataNode1 to DataNode2, DataNode2 to DataNode3):** As DataNode1 receives data packets, it simultaneously writes them to its local storage *and* forwards them to the second DataNode in the pipeline. DataNode2 does the same, writing locally and forwarding to DataNode3. This forms a data "pipeline."
6. **Acknowledgment (DataNodes to Client):** Once DataNode3 (the last in the pipeline) has successfully written the data and confirmed its checksum, it sends an acknowledgment back to DataNode2. DataNode2 then sends an acknowledgment to DataNode1, and finally, DataNode1 sends an acknowledgment back to the client. This ensures all replicas are written consistently.
7. **Repeat for All Blocks:** This entire process (steps 3-6) is repeated for every subsequent block until the entire file is written.
8. **File Close (Client to NameNode):** Once all blocks are written and acknowledged, the client sends a "file close" request to the NameNode. The NameNode then commits the file to the file system namespace, making it visible and accessible for reads.

Write Pipeline Advantages:

- **Efficiency:** Pipelined replication allows data to be replicated in parallel during the write operation, maximizing throughput.
- **Network Optimization:** Reduces network congestion by using a linear flow for replication rather than separate transfers from the client to each replica.

- **Fault Tolerance:** The pipeline can detect and handle DataNode failures during the write process. If a DataNode in the pipeline fails, the client and remaining DataNodes adapt to re-establish the pipeline.
- **Consistency:** Ensures that all replicas of a block are written simultaneously and consistently.

9. Fault Tolerance

Fault tolerance is a cornerstone of HDFS, enabling it to operate reliably even when individual hardware components inevitably fail.

9.1. Types of Failures

HDFS is designed to withstand various types of failures:

- **Hardware Failures:**
  - **DataNode Failure:** An individual server hosting a DataNode process crashes, becomes unresponsive, or is intentionally taken offline.
  - **NameNode Failure:** The master node (NameNode) crashes. This is a critical failure as it holds all metadata.
  - **Network Failure:** A network switch or router fails, leading to isolation of an entire rack or multiple nodes.
  - **Disk Failure:** An individual hard disk drive within a DataNode fails, leading to potential loss of blocks stored on that disk.
- **Software Failures:**
  - **Process Crashes:** The HDFS daemon processes (NameNode, DataNode) crash due to bugs or resource issues.
  - **Data Corruption:** Data blocks on disk get corrupted due to hardware issues, software bugs, or malicious activity.
  - **Configuration Errors:** Misconfigurations can lead to operational issues or instability.

9.2. Fault Tolerance Mechanisms

HDFS employs several mechanisms to ensure data integrity and availability in the face of failures:

- **Data Replication:**
  - **Multiple Copies:** The primary mechanism. Each HDFS block is replicated multiple times (default 3) and stored on different DataNodes, ideally across different racks (due to rack awareness).
  - **Automatic Re-replication:** If the NameNode detects that the number of replicas for a block has fallen below the desired replication factor (e.g., due to a DataNode failure), it automatically initiates the creation of new replicas on healthy DataNodes to restore the configured factor.

- **Replica Placement:** Strategic placement of replicas across different nodes and racks minimizes the impact of localized failures.
- **Configurable Factor:** The replication factor can be adjusted per file or globally based on the data's importance and storage cost considerations.

- **Health Monitoring:**
  - **Heartbeat Mechanism:** DataNodes send periodic "heartbeat" messages (default every 3 seconds) to the NameNode. This signals that the DataNode is alive and operational.
  - **Block Reports:** DataNodes send comprehensive "block reports" (default every 6 hours) listing all blocks they store. This allows the NameNode to maintain an accurate and up-to-date global view of block locations.
  - **Timeout Detection:** If the NameNode stops receiving heartbeats from a DataNode for a configurable period (default 10 minutes), it marks that DataNode as "dead."
  - **Health Checks:** Internal mechanisms and external monitoring tools continuously verify the health of HDFS components.
- **Automatic Recovery:**
  - **Dead Node Detection:** The NameNode's heartbeat monitoring quickly identifies failed DataNodes.
  - **Re-replication:** When a DataNode is marked dead, or when a block's replica count drops for any reason, the NameNode identifies which blocks need re-replication and instructs other healthy DataNodes to create new copies.
  - **Block Recovery:** If a client detects a corrupted block (e.g., via checksum mismatch), it reports it to the NameNode. The NameNode then marks that replica as corrupt and schedules its deletion and re-replication from a good replica.
  - **Metadata Recovery (for NameNode):** In the event of a NameNode failure (before HA was introduced), the FSImage and EditLog were used for recovery. With High Availability, the JournalNodes and a Standby NameNode provide much faster and automatic recovery.

9.3. Failure Scenarios and Responses

- **DataNode Failure:**
  1. The NameNode stops receiving heartbeats from the DataNode.
  2. After a timeout (e.g., 10 minutes), the DataNode is marked as "dead."
  3. All blocks previously stored on the dead DataNode are now considered "under-replicated" by the NameNode.
  4. The NameNode identifies these under-replicated blocks and schedules new replicas to be created on healthy DataNodes in the cluster.
  5. The replication factor for all affected blocks is restored.

- **Block Corruption:**
  1. During a client read operation, a checksum verification fails, indicating that the data block is corrupted.
  2. The client immediately reports the corrupted replica to the NameNode.
  3. The NameNode marks that specific replica as corrupted and avoids directing future read requests to it.
  4. The client attempts to read the same block from a healthy replica on another DataNode.
  5. The NameNode initiates the re-replication of the corrupted block from a good replica to maintain the desired replication factor. The corrupted block is eventually deleted.
- **Rack Failure:**
  1. An entire rack (and potentially its network switch) becomes unreachable.
  2. The NameNode detects that all DataNodes within that rack have stopped sending heartbeats.
  3. All blocks whose only remaining replicas were on the failed rack become "under-replicated."
  4. Due to rack awareness, replicas are usually spread across different racks. The NameNode uses the surviving replicas (on other racks) to re-replicate the affected blocks to new DataNodes on different, healthy racks.

---

10. High Availability (HA)

10.1. The Single Point of Failure Problem (Pre-Hadoop 2.x)

In Hadoop 1.x, the NameNode was a **Single Point of Failure (SPOF)**. This meant:

- **Availability Bottleneck:** If the NameNode crashed, the entire HDFS cluster would become unavailable for both read and write operations.
- **Manual Recovery:** Recovery from a NameNode failure typically involved manually restarting the NameNode, potentially recovering from the latest FSImage and EditLog. This was a time-consuming process.
- **Significant Downtime:** Depending on the size of the cluster and the amount of metadata, recovery could take anywhere from minutes to hours, leading to unacceptable downtime for critical big data applications.

10.2. High Availability Architecture

To address the SPOF issue, Hadoop 2.x introduced **NameNode High Availability (HA)**. This setup involves redundant NameNodes, ensuring that if one fails, another can quickly take over.

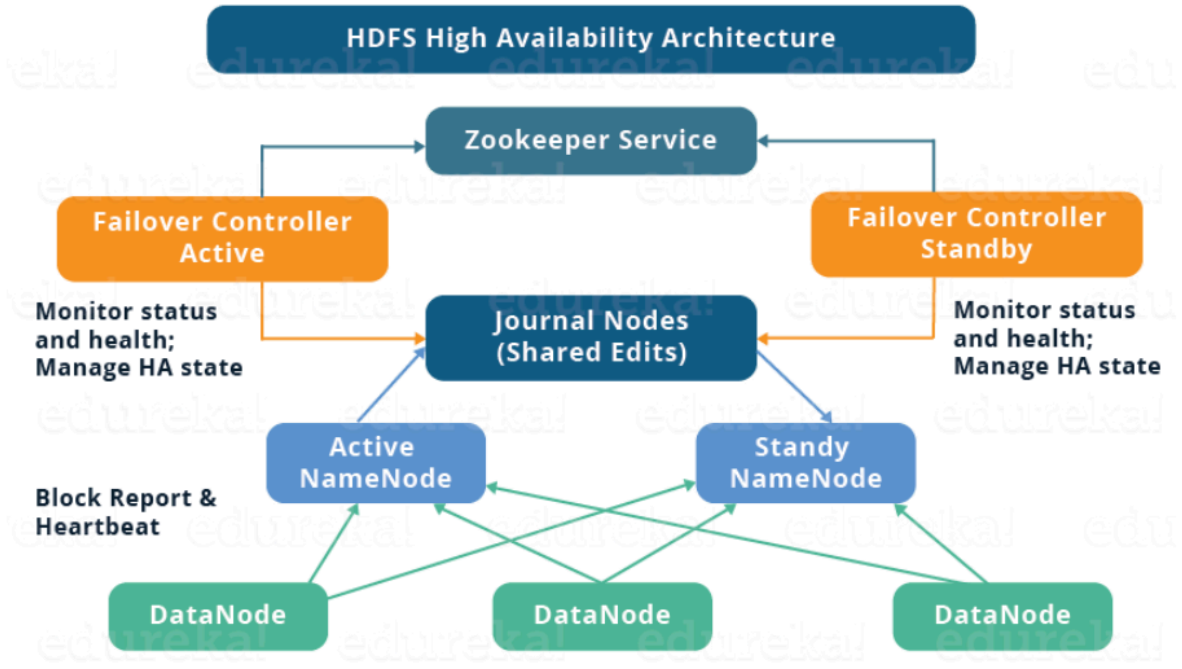
```
                 Shared Storage
                 (Journal Nodes)
                        |
                 +------+------+
                 |             |
           Active NameNode   Standby NameNode
                 |             |
                 +------+------+
                        |
                 DataNode Cluster
```

Key Components of HDFS HA:

- **Active NameNode:** The primary NameNode that handles all client operations (read/write requests) and interacts with DataNodes.
- **Standby NameNode:** A hot spare NameNode that keeps its state synchronized with the Active NameNode. It continuously receives updates to the EditLog and also has an up-to-date FSImage. It becomes the Active NameNode in case of a failover.
- **Journal Nodes (JNs):**
  - A group of lightweight daemon processes (typically 3, 5, or more for a quorum) that provide shared storage for the EditLog.
  - When the Active NameNode makes any change to the HDFS namespace, it writes the transaction to a majority of JournalNodes.
  - The Standby NameNode continuously reads from the JournalNodes to stay synchronized with the Active NameNode's state.
  - They ensure consistency between the Active and Standby NameNodes and prevent "split-brain" scenarios (where both NameNodes think they are active).
- **Automatic Failover (ZooKeeper Integration):**
  - **ZooKeeper:** A distributed coordination service that plays a crucial role in automatic failover. It acts as a shared election service and monitors the health of the Active NameNode.
  - **ZooKeeper QuorumPeer:** A cluster of ZooKeeper servers.
  - **Health Monitoring:** Both Active and Standby NameNodes maintain persistent sessions with the ZooKeeper cluster. The Active NameNode continuously sends heartbeats to ZooKeeper.
  - **Automatic Switchover:** If the Active NameNode fails (stops sending heartbeats to ZooKeeper), ZooKeeper detects this. The Standby NameNode, which has been closely monitoring the Active and keeping its state

synchronized via JournalNodes, is automatically elected and takes over as the new Active NameNode.

- **Fencing:** To prevent a "split-brain" scenario (where a failed Active NameNode recovers and tries to continue operating, leading to data inconsistency), fencing mechanisms are used. Fencing ensures that the old Active NameNode is truly shut down or prevented from causing harm before the new Active takes over (e.g., by killing its process, revoking its network access).


HDFS High Availability Architecture
Zookeeper Service
Failover Controller Active
Failover Controller Standby
Monitor status and health; Manage HA state
Journal Nodes (Shared Edits)
Monitor status and health; Manage HA state
Active NameNode
Standby NameNode
Block Report & Heartbeat
DataNode
DataNode
DataNode
edureka!

Link to the article: https://www.edureka.co/blog/how-to-set-up-hadoop-cluster-with-hdfs-high-availability/

10.3. HA Benefits and Trade-offs

Benefits of HDFS HA:

- **Elimination of SPOF:** The most significant benefit, as the cluster remains operational even if the primary NameNode fails.
- **Reduced Downtime:** Automatic failover dramatically reduces recovery time from hours to seconds or a few minutes, minimizing service interruption.
- **Improved Reliability:** Overall system availability and robustness are greatly enhanced.
- **Maintenance Windows:** Allows for planned maintenance (e.g., software upgrades, hardware replacement) on the NameNode without bringing down the entire cluster.

Trade-offs of HDFS HA:

- **Complexity:** The HA setup is significantly more complex to configure, deploy, and manage compared to a single NameNode setup.

- **Resource Overhead:** Requires additional hardware (for the Standby NameNode and JournalNodes) and introduces more running processes, consuming additional cluster resources.
- **Network Dependencies:** Relies heavily on the stability and performance of the network for communication between NameNodes, JournalNodes, and ZooKeeper.
- **Consistency Overhead:** There's a slight overhead for the Active NameNode to write to JournalNodes and for the Standby NameNode to continuously read from them to maintain synchronization.

11. Performance Characteristics

HDFS is specifically engineered for certain types of workloads, excelling in some areas while having limitations in others.

11.1. Performance Strengths

- **High Throughput:**
  - **Parallel Data Access:** By distributing large files into blocks across many DataNodes, HDFS allows multiple clients or processing tasks to read and write data in parallel.
  - **Large Block Sizes:** Optimized for sequential I/O. Larger blocks reduce the overhead associated with file system metadata and disk seeks, enabling faster data transfer rates for large files.
  - **Network Bandwidth Utilization:** Designed to saturate network links with data transfer, ensuring efficient use of cluster resources.
  - **Streaming Access Patterns:** Excel in scenarios where entire datasets are read or written sequentially (e.g., batch processing, analytics).
- **Scalability:**
  - **Linear Scaling:** HDFS can scale almost linearly in terms of storage capacity and processing power simply by adding more commodity nodes to the cluster.
  - **Petabyte-scale Data:** Capable of storing and managing datasets ranging from terabytes to petabytes and beyond.
  - **Thousands of Nodes:** Supports clusters with hundreds to thousands of DataNodes.
  - **Horizontal Scaling Model:** Contrasts with vertical scaling (upgrading a single server), offering a more cost-effective and flexible way to expand.

11.2. Performance Limitations

Understanding these limitations is crucial for choosing the right tool for the job.

- **Small Files Problem:**
  - **NameNode Memory Bottleneck:** Each file, regardless of its size, requires a metadata entry in the NameNode's memory. Millions of small files can

overwhelm the NameNode's memory, leading to performance degradation or out-of-memory errors.
  - **Wasted Block Space:** If a file is smaller than the HDFS block size (e.g., a 1KB file on a 128MB block), it still consumes an entire block of disk space (though only the 1KB of data is actual payload). This leads to inefficient storage utilization.
  - **Inefficient I/O:** Reading many small files involves numerous disk seeks and network connections, which is inefficient compared to continuous large sequential reads.
- **Random Access:**
  - **Optimized for Sequential Access:** HDFS is primarily optimized for sequential read and write operations.
  - **Inefficient Random Seeks:** For applications requiring random access (e.g., retrieving specific records from a large file without reading the whole file), HDFS can be inefficient due to the overhead of seeking to different blocks across the network.
  - **Not Suitable for OLTP:** HDFS is generally not suitable for Online Transaction Processing (OLTP) workloads that require low-latency, record-level random reads and writes (e.g., traditional relational databases).
- **Low Latency (High Throughput over Low Latency):**
  - **Batch Processing Focus:** HDFS prioritizes high aggregate throughput for large batch jobs over achieving very low latency for individual operations.
  - **Network Overhead:** The distributed nature of HDFS, with data transfers over the network, inherently introduces some latency compared to accessing data on a local disk.
  - **Not for Real-time Applications:** Not designed for real-time applications that demand immediate responses (e.g., interactive queries on single records, streaming analytics requiring millisecond latency). For such needs, other components like HBase or specialized streaming engines are used.

11.3. Optimization Strategies

To mitigate HDFS limitations and optimize its performance:

- **File Organization:**
  - **Combine Small Files:** Use tools like `hadoop archive` (HAR) or techniques like `SequenceFiles` or `Parquet` to combine many small files into larger HDFS files. This reduces NameNode metadata overhead and improves I/O efficiency.
  - **Optimal Block Size:** While 128MB is the default, adjust the block size based on your typical file sizes and processing patterns. Larger blocks for very large files, smaller for a mix of sizes (but still larger than traditional file system blocks).

- **Compression:** Apply compression (e.g., Gzip, Snappy, LZO) to reduce the actual data stored on disk and transferred over the network, improving both storage efficiency and I/O performance.
- **Efficient File Formats:** Utilize columnar storage formats like **Parquet** or **ORC**. These formats store data in a column-wise manner, allowing for efficient projection pushdown (reading only necessary columns) and predicate pushdown (filtering data before reading it), significantly improving query performance for analytical workloads.

- **Cluster Configuration:**
  - **NameNode Memory Sizing:** Ensure the NameNode has sufficient RAM to hold the entire file system namespace in memory, especially for clusters managing many files.
  - **Network Optimization:** Use high-bandwidth network connections (e.g., 10 Gigabit Ethernet) between DataNodes and especially between racks to minimize data transfer bottlenecks.
  - **Storage Balance:** Monitor and ensure that data is evenly distributed across DataNodes to prevent hot spots and maximize parallel I/O.
  - **Replication Tuning:** Adjust the replication factor based on the importance of data and your storage budget. While 3 is common, some non-critical data might tolerate 2, or highly critical data might require 4 or more. Consider **Erasure Coding** for cold data to save significant storage space while maintaining fault tolerance.

---

12. Summary

HDFS represents a fundamental shift in how large-scale data is stored and managed. It is specifically engineered to handle the challenges of "big data," moving beyond the limitations of traditional, single-machine file systems. Its robust design makes it an indispensable component of the big data ecosystem.

Key Takeaways:

- **Purpose-Built:** HDFS is designed for massive datasets, prioritizing high throughput for sequential reads and writes over low latency or random access.
- **Fault Tolerance through Replication:** Data is replicated across multiple nodes and racks to ensure availability and prevent data loss even in the face of hardware failures.
- **Rack Awareness for Efficiency:** Smart replica placement optimizes fault tolerance and reduces inter-rack network traffic.
- **High Availability Eliminates SPOF:** The HA architecture with Active/Standby NameNodes and Journal Nodes provides continuous service even if the primary NameNode fails.

- **Master-Slave Architecture:** Clear separation of responsibilities between the NameNode (metadata) and DataNodes (data storage).
- **Embraces Commodity Hardware:** Designed to run on inexpensive, off-the-shelf servers, making it cost-effective for petabyte-scale storage.

Best Use Cases for HDFS:

- **Large-scale data analytics and processing:** Ideal for batch processing frameworks like MapReduce, Spark, and Hive, where entire datasets are scanned.
- **Data archival and backup systems:** Cost-effective for storing large volumes of historical or infrequently accessed data.
- **ETL (Extract, Transform, Load) operations:** Efficiently handles the staging and processing of large datasets before loading them into data warehouses.
- **Machine learning training data storage:** Stores massive datasets required for training complex machine learning models.
- **Log aggregation and analysis:** Collects and processes vast amounts of log data from various sources for insights.

When NOT to Use HDFS:

- **Real-time, low-latency applications:** Not suitable for interactive applications requiring immediate responses (e.g., serving web pages, real-time dashboards for single queries).
- **Systems with many small files:** The "Small Files Problem" leads to excessive NameNode memory consumption and inefficient I/O.
- **Applications requiring frequent random access patterns:** HDFS is not optimized for random read/write operations (e.g., traditional transactional databases).
- **Single-user systems with small datasets:** Overkill and adds unnecessary complexity for personal use or small-scale data storage. A local file system is more appropriate.