# Comprehensive Lecture Notes: HDFS Architecture, High Availability, and Scalability

Based on Lecture by Dr. Tariq Mahmood

Fall 2025

## Contents

## 1 Introduction to HDFS (Hadoop Distributed File System)

The Hadoop Distributed File System (HDFS) is the underlying storage layer of the Hadoop ecosystem, designed to store vast amounts of data reliably across clusters of commodity hardware.

### 1.1 Core Characteristics

- **WORM Pattern (Write-Once, Read-Many):** HDFS is optimized for data that is written once and rarely modified, but read multiple times.
  - *Implication:* Once a file is created and closed, it cannot be overwritten or edited in the middle. Data integrity is prioritized over mutability.
  - *Example:* A daily server log file. Once the day ends and the log is rotated, it serves as a historical record for analysis and is never modified again.
- **Commodity Hardware:** Unlike traditional supercomputers, HDFS runs on "commodity" hardware—affordable, widely available components (e.g., standard servers with multi-core processors and 16-64 GB RAM).
  - **Storage Media:** HDFS utilizes standard Hard Disk Drives (HDD) for DataNodes due to their cost-effectiveness for bulk storage. However, **Solid State Drives (SSD)** are recommended for the NameNode to handle high-speed random metadata operations.
- **Latency Profile:** HDFS is designed for **batch processing** (high throughput), not low latency.
  - It is **not** suitable for applications requiring immediate response times, such as online gaming, video streaming, or real-time IoT control systems.
  - *Modern Evolution:* While HDFS itself is high-latency, engines like Apache Spark or Flink running on top of HDFS can provide near real-time processing capabilities.

### 1.2 The Small Files Problem

HDFS is not designed to handle a large number of very small files efficiently due to the architecture of the NameNode.

**Memory Constraints:** The NameNode stores the metadata for the entire file system in its Random Access Memory (RAM). The limit to the number of files in the cluster is governed by the NameNode's memory capacity, not the disk space of the cluster.

**Rule of Thumb (ROT):**

- Each block metadata takes $\approx$ 150 bytes.
- Each file inode takes $\approx$ 150 bytes.
- **Total per file (single block):** $\approx$ 300 bytes.

**Calculation Example:** If a cluster stores 1 million small files, the memory requirement is:

$$1,000,000 \text{ files} \times 300 \text{ bytes/file} = 300,000,000 \text{ bytes} \approx 300 \text{ MB} \tag{1}$$

While 300 MB is manageable, scaling to billions of files would require hundreds of gigabytes of RAM, which becomes physically and economically infeasible. Additionally, processing millions of small files degrades computation performance (e.g., in MapReduce) due to the overhead of starting tasks for small amounts of data.

## 2 HDFS Architecture Components

### 2.1 The NameNode (Master)

The NameNode is the centerpiece of an HDFS file system. It keeps the directory tree of all files in the file system, and tracks where across the cluster the file data is kept.

- **FsImage:** A stored snapshot (checkpoint) of the file system metadata.
- **EditLog:** A transaction log recording every change (write, delete, rename) that occurs after the snapshot.
- **State Recovery:** Upon startup, the NameNode loads the FsImage and applies the EditLog to reconstruct the current state of the file system.

### 2.2 The DataNodes (Workers)

DataNodes are the workhorses of the file system.

- They store and retrieve data blocks when instructed by clients or the NameNode.
- They report back to the NameNode periodically with lists of blocks they are storing (Block Reports) and send "Heartbeats" to indicate they are alive.

### 2.3 The Secondary NameNode

**Crucial Distinction:** The Secondary NameNode is **not** a backup or standby node for the Active NameNode. It does not provide High Availability (HA).

**Function:** Its primary role is to perform periodic **checkpointing**.

1. It downloads the current FsImage and EditLog from the Active NameNode (via HTTP).
2. It merges the EditLog into the FsImage in its own memory.
3. It creates a new `fsimage.checkpoint` file.
4. It uploads this new image back to the Active NameNode.

*Benefit:* This prevents the EditLog from growing indefinitely, which would otherwise make restarting the NameNode incredibly slow (as it would have to replay a massive log).

## 3 High Availability (HA) in HDFS

### 3.1 The Limitation of Hadoop 1.x

In early versions of Hadoop, the NameNode was a Single Point of Failure (SPOF).

- If the NameNode failed, the entire cluster became unavailable.
- The Secondary NameNode could not take over immediately.
- Restoring the NameNode from a cold start could take 30 minutes or more on large clusters (loading image + replaying logs + waiting for block reports).

### 3.2 Hadoop 2.x Architecture: Active and Standby

Hadoop 2 introduced native High Availability. This involves running two NameNodes:

1. **Active NameNode:** Handles all client operations.
2. **Standby NameNode:** Acts as a hot backup, maintaining synchronized state with the Active node to facilitate fast failover.

#### 3.2.1 Shared Edits via JournalNodes

To keep the Standby node synchronized, HDFS uses a group of lightweight daemons called **JournalNodes**.

- **Mechanism:** The Active NameNode writes modifications to a quorum of JournalNodes. The Standby NameNode constantly reads these edits and applies them to its own namespace.
- **Quorum:** A majority of JournalNodes must be available to write a log. There must be an odd number of JournalNodes (usually 3 or 5) to tolerate failures (e.g., $(N-1)/2$ failures).

#### 3.2.2 Failover Controller (ZKFC)

Automatic failover is managed by **ZooKeeper** and the **ZooKeeper Failover Controller (ZKFC)**.

- **ZooKeeper:** A distributed coordination service that maintains a lock to decide which NameNode is Active.
- **ZKFC:** A client process running on each NameNode machine. It monitors the health of the NameNode and manages the session with ZooKeeper. If the Active NameNode becomes unresponsive, ZKFC terminates the session, triggering ZooKeeper to elect the Standby as the new Active.

## 4 Scalability: HDFS Federation

Even with High Availability, a single NameNode limits the total number of files a cluster can store because all metadata must reside in memory. **HDFS Federation** solves this by horizontally scaling the metadata layer.

### 4.1 Concept

Federation allows multiple NameNodes to function independently within the same cluster.

- **Namespace Volume:** Each NameNode manages a specific slice of the file system (e.g., one manages `/user`, another manages `/projects`).
- **Block Pool:** The underlying storage (DataNodes) is shared. A DataNode stores blocks for *all* NameNodes in the federation.
- **Independence:** Failures are isolated. If the NameNode for `/user` goes down, the NameNode for `/projects` remains unaffected.

### 4.2 Routing Mechanisms

To present these separate namespaces as a unified file system to the client, two approaches are used:

1. **ViewFs (Client-Side):**
   - The client utilizes a "View File System." The mount table (mapping `/data` to Cluster A and `/logs` to Cluster B) acts like a virtual file system.
   - *Drawback:* Requires configuration changes on every client machine whenever a new mount point is added.
2. **Router-Based Federation (Server-Side):**
   - Introduced by Microsoft in HDFS 2.9.
   - Adds a software layer (Router) that centralizes the namespace management.
   - Clients connect to the Router, which transparently forwards requests to the correct NameNode. This removes the need for client-side config updates.

## 5 Case Study: Uber's HDFS Optimization

Uber manages over 100 petabytes of data, with huge query volumes from Presto and Hive. As they scaled, they encountered severe bottlenecks in HDFS performance.

### 5.1 Challenges Faced

1. **Read Bottleneck:** Over 50% of HDFS traffic came from read-only Presto queries. The Active NameNode became a bottleneck, with RPC queue times exceeding 500ms (normal is 10ms).
2. **Small Files:** Streaming ingestion pipelines generated millions of tiny files, clogging NameNode memory.
3. **Garbage Collection (GC):** Large Java heaps on NameNodes led to long "stop-the-world" GC pauses, freezing the cluster.

### 5.2 Uber's Solutions

#### 5.2.1 1. Observer NameNode

To address the read bottleneck, Uber implemented the **Observer NameNode**.

- **Role:** A read-only replica of the NameNode.
- **Mechanism:** It reads edit logs from JournalNodes (like the Standby) but allows clients to read directly from it.
- **Consistency:** It provides "eventual consistency." Clients willing to accept a slight lag (ms) can read from the Observer, offloading massive read traffic from the Active NameNode.
- **Result:** Improved NameNode throughput by nearly 100%.

#### 5.2.2 2. Handling Small Files (Stitching)

Uber developed libraries (e.g., **Hoodie**) and a "Stitcher" service.

- Instead of storing raw small files from streams, the Stitcher merges them into larger files (typically >1GB).
- This significantly reduces the metadata object count on the NameNode.

#### 5.2.3 3. Operational Improvements

- **Spotlight:** A real-time audit log analysis tool (using Flink/Kafka) to identify and automatically kill abusive queries or users causing slowdowns.
- **GC Tuning:** Optimized JVM settings using Concurrent Mark Sweep (CMS) to perform aggressive old-generation collection, preventing long pauses.
- **Quotas:** Enforced strict namespace quotas (allocation based on 256MB/file ratio) to force developers to optimize file generation.

#### 5.2.4 4. Tiered Storage

To manage costs, Uber implemented a tiered approach:

- **Hot Tier:** High-performance hardware for recent, frequently accessed data.
- **Warm Tier:** High-density storage (250TB+ servers) for older data. This tier utilizes **Erasure Coding** (EC) rather than standard 3x replication to save space while maintaining fault tolerance.

## 6 Summary

HDFS has evolved from a simple batch storage system to a sophisticated, highly available distributed architecture. While the NameNode memory limitation persists as a primary bottleneck, strategies like Federation, ViewFs, and the Observer NameNode allow HDFS to scale to the exabyte level, as demonstrated by Uber's infrastructure evolution.