HDFS

HDFS (Hadoop Distributed File System) is the storage layer of Hadoop, designed to store and manage large volumes of data across distributed clusters. It is optimized for scalability, fault tolerance, and high-throughput data access.

HDFS is a distributed file system that:

- Stores data across multiple machines in a cluster.
- Provides high-throughput access to data, even for large files.
- Is designed to handle hardware failures gracefully.
- Is optimized for batch processing rather than real-time access.

Assumptions and Goals

1. **Hardware Failure:** Hardware failure is the norm rather than the exception. An HDFS instance may consist of hundreds or thousands of server machines, each storing part of the file system's data. The fact that there are a huge number of components and that each component has a non-trivial probability of failure means that some component of HDFS is always non-functional. Therefore, detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

2. **Streaming Data Access:** Applications that run on HDFS need streaming access to their data sets. They are not general purpose applications that typically run on general purpose file systems. HDFS is designed more for batch processing rather than interactive use by users. The emphasis is on high throughput of data access rather than low latency of data access. POSIX imposes many hard requirements that are not needed for applications that are targeted for HDFS. POSIX semantics in a few key areas has been traded to increase data throughput rates.

3. **Large Data Sets:** Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size. Thus, HDFS is tuned to support large files. It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster. It should support tens of millions of files in a single instance.

4. **Simple Coherency Model:** HDFS applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed except for appends and truncates. Appending the content to the end of the files is supported but cannot be updated at arbitrary point. This assumption simplifies data coherency issues and enables high throughput data access. A MapReduce application or a web crawler application fits perfectly with this model.

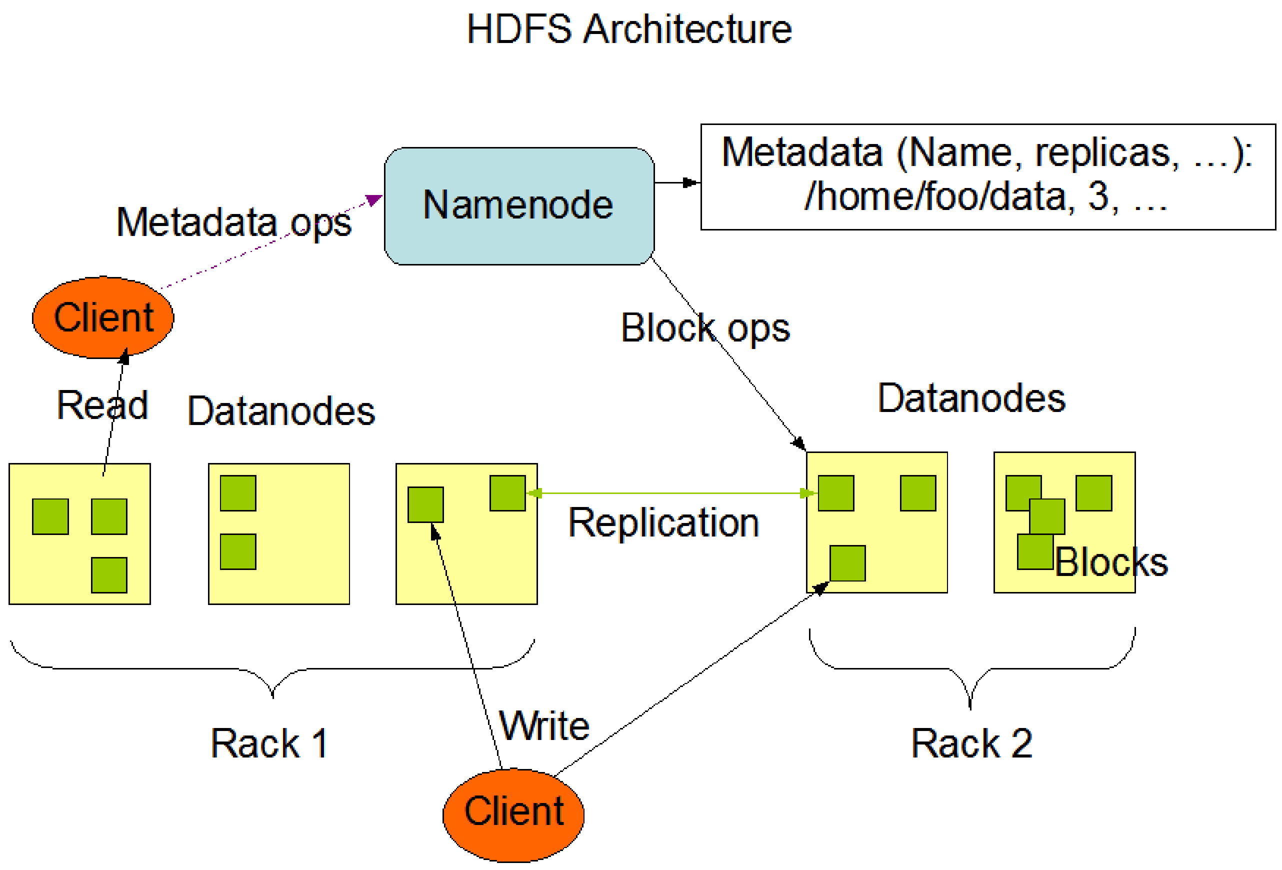
5. **"Moving Computation is Cheaper than Moving Data":** A computation requested by an application is much more efficient if it is executed near the data it operates on. This is especially true when the size of the data set is huge. This minimizes network congestion and increases the overall throughput of the system. The assumption is that it is often better to migrate the computation closer to where the data is located rather than moving the data to where the application is running. HDFS provides interfaces for applications to move themselves closer to where the data is located.

6. **Portability Across Heterogeneous Hardware and Software Platforms:** HDFS has been designed to be easily portable from one platform to another. This facilitates widespread adoption of HDFS as a

platform of choice for a large set of applications.

Architecture of HDFS

- HDFS has a master/slave architecture. An HDFS cluster consists of a single ***NameNode***, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of ***DataNodes***, usually one per node in the cluster, which manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes.



- The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

- The NameNode and DataNode are pieces of software designed to run on commodity machines. These machines typically run a GNU/Linux operating system (OS). HDFS is built using the Java language; any machine that supports Java can run the NameNode or the DataNode software. Usage of the highly portable Java language means that HDFS can be deployed on a wide range of machines.

- A typical deployment has a dedicated machine that runs only the NameNode software. Each of the other machines in the cluster runs one instance of the DataNode software. The architecture does not preclude running multiple DataNodes on the same machine but in a real deployment that is rarely the case.

HDFS Namespace

HDFS is designed to provide a robust and scalable file system for distributed storage. Its hierarchical organization, support for user quotas and access permissions, and the role of the NameNode in managing the file system namespace and replication factor all contribute to its efficiency and reliability.

Hierarchical File Organization

HDFS supports a traditional hierarchical file organization, similar to many other file systems. This means:

- **Directories and Files**: Users and applications can create directories and store files within these directories.
- **File System Namespace**: The hierarchy of the file system namespace allows for operations such as creating and removing files, moving files from one directory to another, and renaming files.
- **User Quotas and Access Permissions**: HDFS supports user quotas, which limit the amount of data a user can store, and access permissions to control who can read, write, or execute files.

Reserved Paths and Naming Conventions

While HDFS follows the naming conventions of typical file systems, it reserves certain paths and names for specific features:

Reserved Paths: Paths like /.reserved and .snapshot are reserved for special purposes. Transparent Encryption and Snapshots: These features use reserved paths to provide additional functionality, such as encrypting data transparently and creating snapshots of the file system for backup and recovery.

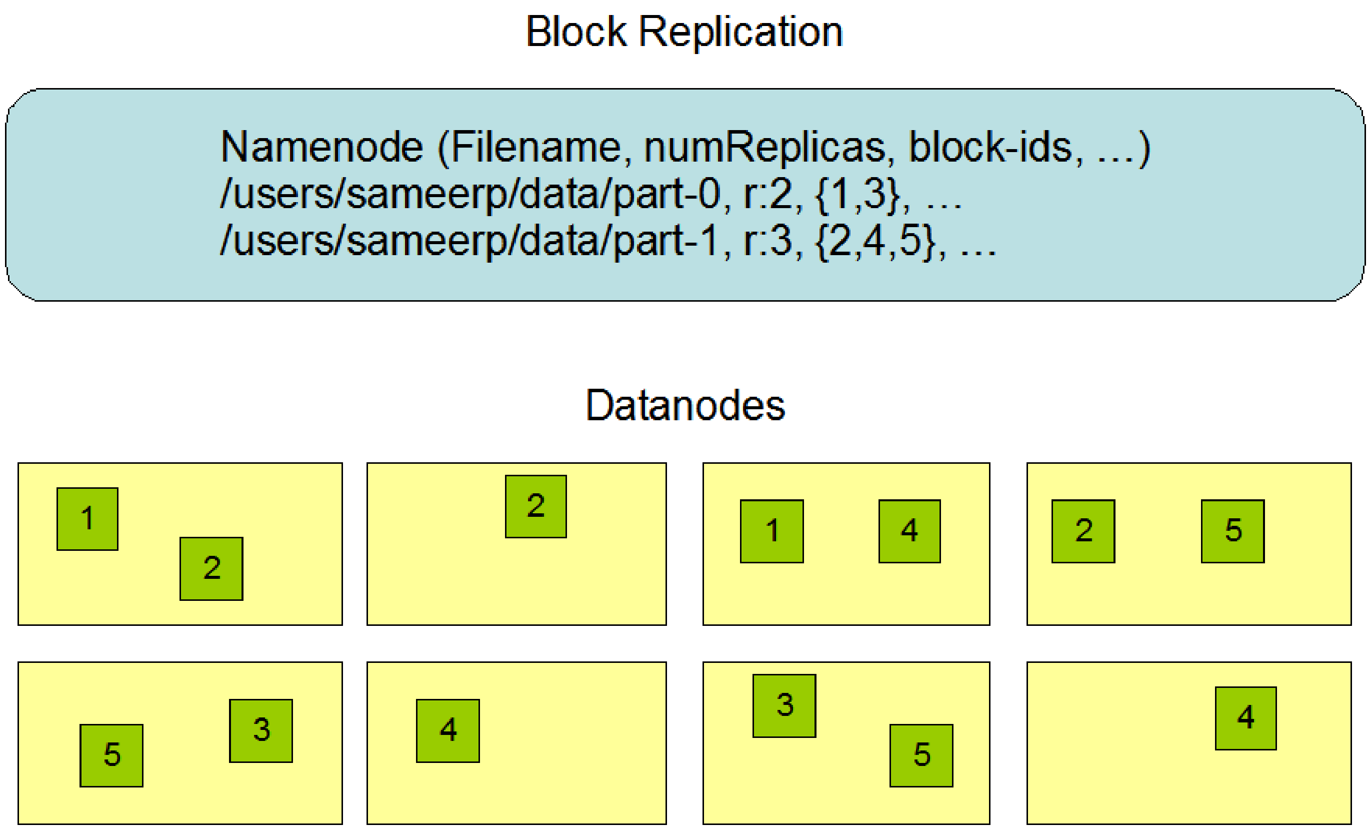
Role of the NameNode

The NameNode is a critical component of HDFS, responsible for maintaining the file system namespace:

- **Recording Changes**: Any changes to the file system namespace or its properties are recorded by the NameNode.
- **Replication Factor**: Applications can specify the number of replicas (copies) of a file that HDFS should maintain. This replication factor ensures data redundancy and reliability. The NameNode stores this information and manages the replication process.

DATA Replication

- HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file.


Block Replication
Namenode (Filename, numReplicas, block-ids, …)
/users/sameerp/data/part-0, r:2, {1,3}, …
/users/sameerp/data/part-1, r:3, {2,4,5}, …
Datanodes
1
2
2
1
4
2
5
5
3
4
3
5
4

HDFS File Storage and Fault Tolerance:

Purpose: HDFS is designed to store very large files across a cluster of machines in a distributed manner, ensuring fault tolerance and reliability.

- File Storage:
  - Files in HDFS are split into blocks (e.g., 128 MB by default).
  - Blocks are replicated across multiple machines (DataNodes) for fault tolerance.
  - The block size and replication factor (number of copies) can be configured per file.
- Block Handling:
  - All blocks except the last one are of equal size.
  - The last block can be smaller than the configured block size if it is not completely filled.
  - Variable-length block support: With features like append and hsync, users can start a new block without filling the last one to the full block size.
- Replication:
  - Each block is replicated according to the specified replication factor.
  - Replication factor can be defined during file creation and modified later.
  - Write-once model for files, allowing only one writer at a time, but appends and truncates are allowed.
- Role of the NameNode:
  - The NameNode handles metadata, including block locations and the replication strategy.
  - The NameNode receives periodic Heartbeats (to check DataNode status) and Blockreports (which list the blocks stored by each DataNode).

Replica Placement and Rack-Aware Policy:

- Replica Placement Strategy:

  - The placement of replicas is crucial for optimizing data reliability, availability, and network utilization in HDFS.
  - The placement policy is rack-aware, ensuring replicas are distributed across different racks to improve reliability and performance.

- Rack Awareness:

  - HDFS nodes often span multiple racks in a data center.
  - Inter-rack communication is slower compared to intra-rack communication, so replica placement aims to minimize inter-rack traffic during writes.

- Default Replica Placement Policy:

  - Replication Factor = 3:
    - One replica is placed on the local DataNode if the writer is on that DataNode.
    - Another replica is placed on a random DataNode within the same rack.
    - The third replica is placed on a different rack to ensure fault tolerance.
    - This strategy optimizes write performance by minimizing inter-rack write traffic, while still ensuring data reliability.
  - Replication Factor > 3:
    - For higher replication factors, additional replicas are placed randomly, ensuring that the number of replicas per rack does not exceed an upper limit.
    - The upper limit is typically (replicas - 1) / racks + 2.

- Storage Policies and Rack-Aware Decisions:

  - Storage Types and Policies: After supporting different storage types (e.g., SSD, HDD), the NameNode takes these policies into account while placing replicas.
  - Node Selection: The NameNode first tries to place replicas based on rack awareness. If no suitable nodes are available (due to storage type constraints), it looks for nodes with fallback storage types.

- Replica Selection and Read Performance:

  - Read Performance Optimization:
    - HDFS tries to minimize network bandwidth consumption and read latency by serving read requests from the closest replica to the client.
    - If a replica exists on the same rack as the reader, that replica is preferred.
    - For clusters spanning multiple data centers, local replicas (within the same data center) are preferred over remote replicas.

- Replica Placement for Reads:

  - The system ensures that the placement of blocks is efficient for both write and read operations.

- Block Placement Policies:

  - HDFS has several pluggable block placement policies, with the default being the BlockPlacementPolicyDefault.
  - Users can choose different policies based on their infrastructure and use cases.

    - The current default policy attempts to distribute replicas in a way that improves write performance without compromising data reliability.

- Safemode in HDFS: On startup, the NameNode enters Safemode, a special state where no block replication happens. During Safemode, the NameNode waits for Heartbeats and Blockreports from the DataNodes. The Safemode Process is as follow:

    1. The NameNode checks that minimum replica requirements are met for each block.
    2. A block is considered safely replicated when the required number of replicas have been confirmed.
    3. Once a configurable percentage of blocks have safely replicated, and after an additional 30 seconds, the NameNode exits Safemode.

  After exiting Safemode, the NameNode identifies any blocks that still have fewer than the required replicas and starts replicating those blocks to ensure sufficient redundancy.

How HDFS Works?

1. File Storage

- Files are split into fixed-size blocks (default: 128 MB or 256 MB).
- Each block is stored on multiple DataNodes (replication factor: 3 by default).
- The NameNode tracks the location of each block.

2. Write Operation

- The client contacts the NameNode to request file creation.
- The NameNode allocates DataNodes for storing the file blocks.
- The client writes data to the first DataNode, which replicates it to other DataNodes in a pipeline.

3. Read Operation

- The client contacts the NameNode to get the block locations.
- The client reads data directly from the nearest DataNode(s).

4. Replication

- Each block is replicated across multiple DataNodes for fault tolerance.
- Replication factor is configurable (default: 3).

Use Cases for HDFS

1. Big Data Storage: Stores petabytes of data for analytics and machine learning.

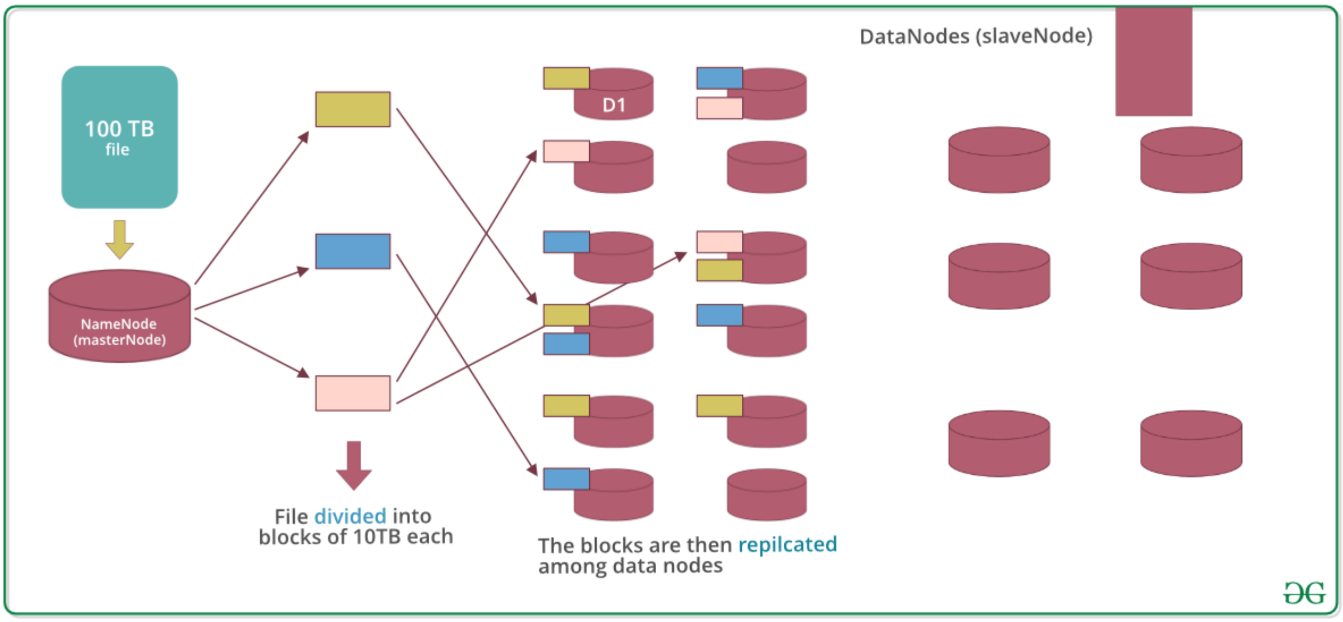
2. Batch Processing: Works seamlessly with frameworks like MapReduce, Spark, and Hive.

3. Data Warehousing: Acts as the storage layer for tools like Apache Hive and Apache Impala.

4. Log Storage: Ideal for storing and analyzing large volumes of log data.

Data storage in HDFS

Now let’s see how the data is stored in a distributed manner.


DataNodes (slaveNode)
100 TB file
NameNode (masterNode)
D1
File divided into blocks of 10TB each
The blocks are then repilcated among data nodes

Lets assume that 100TB file is inserted, then masternode(namenode) will first divide the file into blocks of 10TB (default size is 128 MB in Hadoop 2.x and above). Then these blocks are stored across different datanodes(slavenode). Datanodes(slavenode)replicate the blocks among themselves and the information of what blocks they contain is sent to the master. Default replication factor is 3 means for each block 3 replicas are created (including itself). In hdfs.site.xml we can increase or decrease the replication factor i.e we can edit its configuration here.

Note: MasterNode has the record of everything, it knows the location and info of each and every single data nodes and the blocks they contain, i.e. nothing is done without the permission of masternode.