



Hadoop Distributed File System: Introduction

This lesson gives a brief introduction to the Hadoop Distributed File System.

We'll cover the following




- Goal
- What is Hadoop Distributed File System (HDFS)?
- Background
- APIs

Goal#

Design a distributed system that can store huge files (terabyte and larger). The system should be scalable, reliable, and highly available.

What is Hadoop Distributed File System (HDFS)?#

HDFS is a distributed file system and was built to store unstructured data. It is designed to store huge files reliably and stream those files at high bandwidth to user applications.

HDFS is a variant and a simplified version of the Google File System (GFS).  lot of HDFS architectural decisions are inspired by GFS design. HDFS is built

around the idea that the most efficient data processing pattern is a **write-once, read-many-times** pattern.



Background#

[Apache Hadoop](#) is a software framework that provides a distributed file storage system and distributed computing for analyzing and transforming very large data sets using the [MapReduce](#) programming model. HDFS is the default file storage system in Hadoop. It is designed to be a **distributed, scalable, fault-tolerant** file system that primarily caters to the needs of the **MapReduce** paradigm.

Both HDFS and GFS were built to store very large files and scale to store petabytes of storage. Both were built for handling batch processing on huge data sets and were designed for data-intensive applications and not for end-users. Like GFS, HDFS is also not POSIX-compliant and is not a mountable file system on its own. It is typically accessed via HDFS clients or by using application programming interface (API) calls from the Hadoop libraries.

Given the current HDFS design, the following types of applications are not a good fit for HDFS:

1. Low-latency data access:

HDFS is optimized for high throughput (which may come at the expense of latency). Therefore, applications that need low-latency data access will not work well with HDFS.

2. Lots of small files:

HDFS has a central server called NameNode, which holds all the filesystem metadata in memory. This limits the number of files in the filesystem by the amount of memory on the NameNode. Although storing millions of files is feasible, billions are beyond the capability of the current hardware.



3. No concurrent writers and arbitrary file modifications:

Contrary to GFS, multiple writers cannot concurrently write to



file. Furthermore, writes are always made at the end of the file, in an append-only fashion; **there is no support for modifications at arbitrary offsets in a file.**

APIs#

HDFS does not provide standard POSIX-like APIs. Instead, it exposes user-level APIs. In HDFS, files are organized hierarchically in directories and identified by their pathnames. HDFS supports the usual file system operations, e.g., files and directories can be **created**, **deleted**, **renamed**, **moved**, and **symbolic links** can be created. All **read** and **write** operations are done in an append-only fashion.

[← Back](#)[Next →](#)[Mock Interview: GFS](#)[High-level Architecture](#)[✓ Completed](#)[⚠ Report an Issue](#)



High-level Architecture

This lesson gives a brief overview of HDFS's architecture.

We'll cover the following

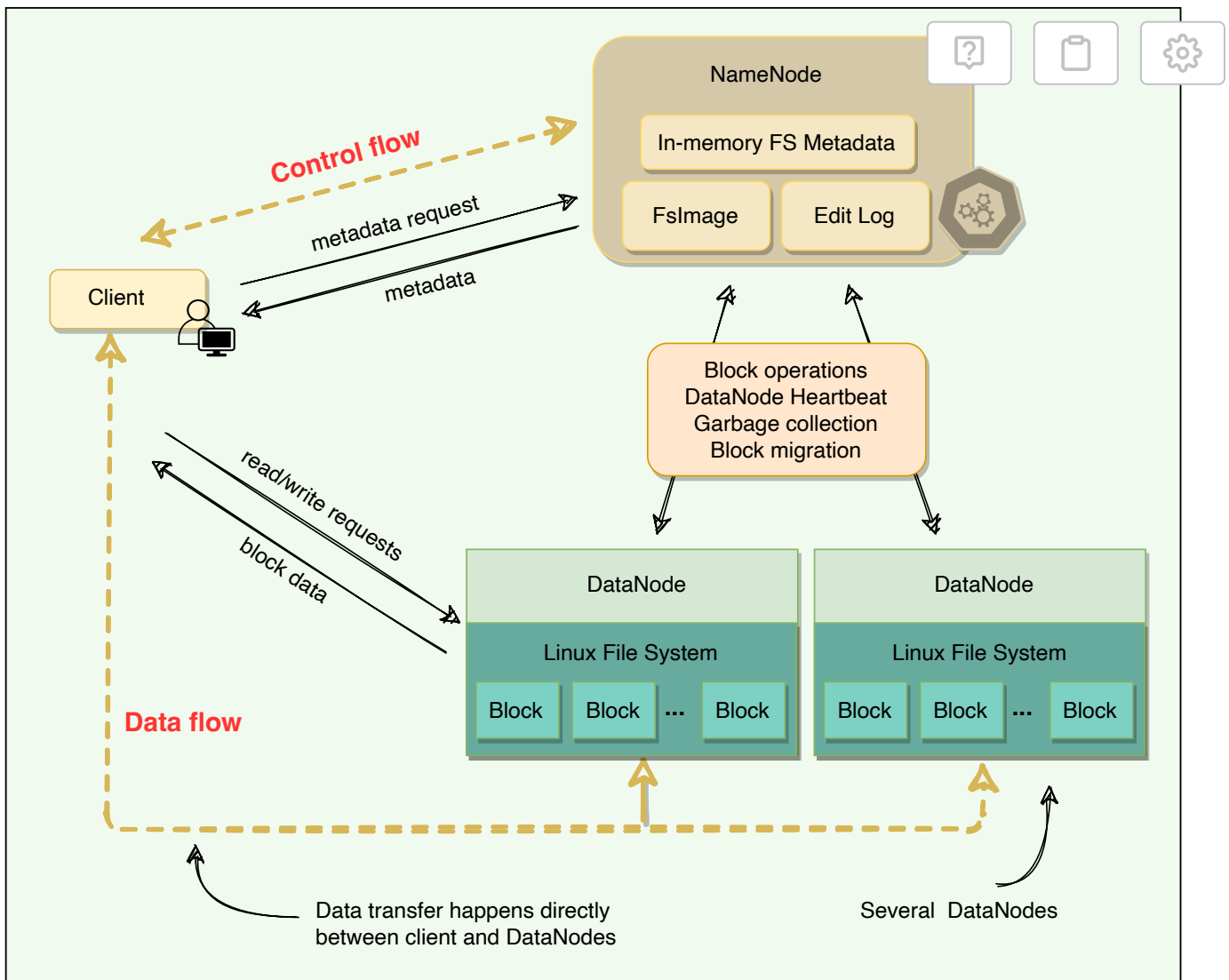


- HDFS architecture
- Comparison between GFS and HDFS

HDFS architecture#

All files stored in HDFS are broken into multiple fixed-size blocks, where each block is 128 megabytes in size by default (configurable on a per-file basis). Each file stored in HDFS consists of two parts: the **actual file data** and the **metadata**, i.e., how many block parts the file has, their locations and the total file size, etc. HDFS cluster primarily consists of a **NameNode** that manages the file system metadata and **DataNodes** that store the actual data.



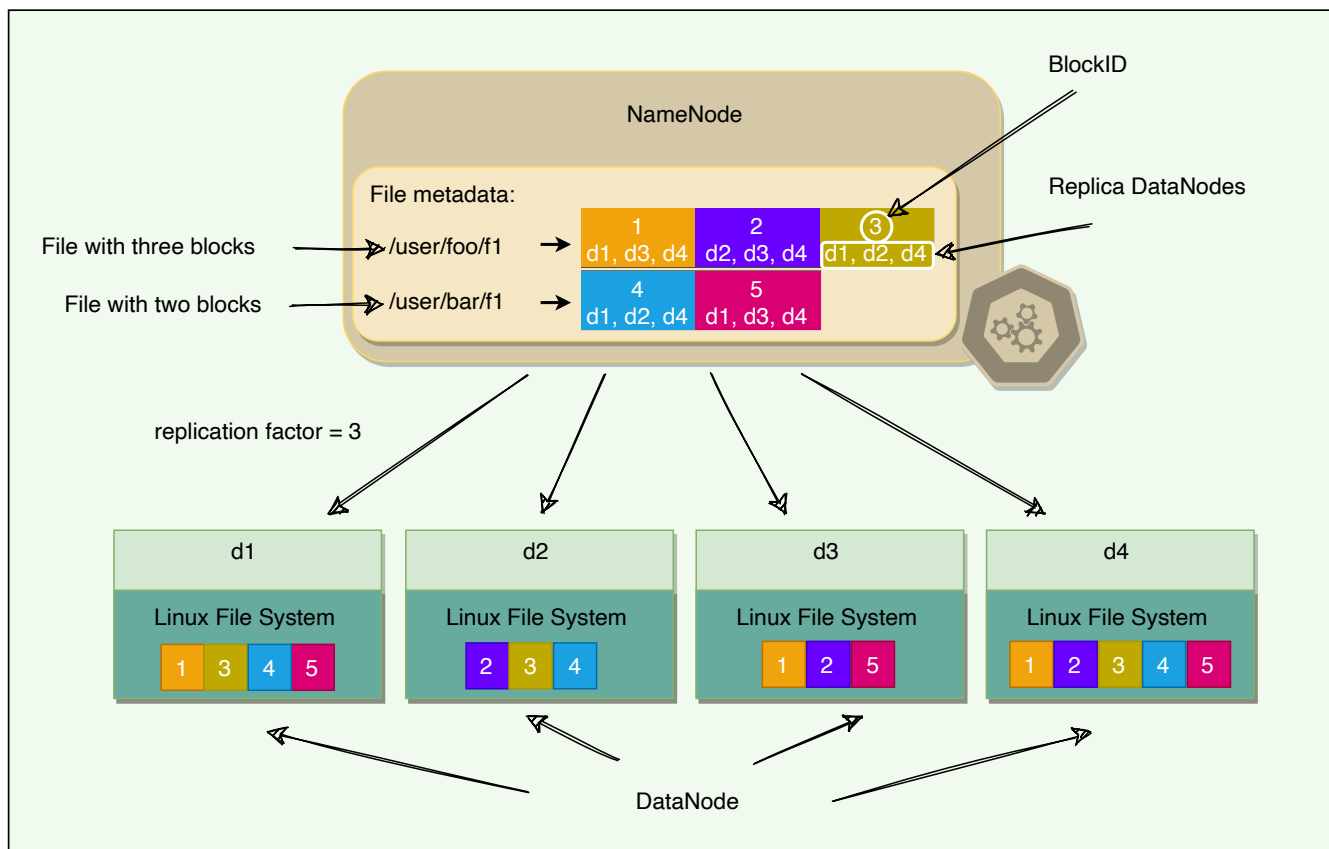


HDFS high-level architecture

- All blocks of a file are of the same size except the last one.
- HDFS uses **large block sizes** because it is designed to store extremely large files to enable MapReduce jobs to process them efficiently.
- Each block is identified by a unique 64-bit ID called **BlockID**.
- All read/write operations in HDFS operate at the block level.
- DataNodes store each block in a separate file on the local file system and provide read/write access.
- When a DataNode starts up, it scans through its local file system and sends the list of hosted data blocks (called BlockReport) to the NameNode.



- The NameNode maintains two on-disk data structures to represent the system's state: an **FsImage** file and an **EditLog**. FsImage is a checkpoint of the file system metadata at some point in time, while the EditLog is a log of all of the file system metadata transactions since the image file was last created. These two files help NameNode to recover from failure.
- User applications interact with HDFS through its client. HDFS Client interacts with NameNode for metadata, but all data transfers happen directly between the client and DataNodes.
- To achieve high-availability, HDFS creates multiple copies of the data and distributes them on nodes throughout the cluster.



HDFS block replication

Comparison between GFS and HDFS#



HDFS architecture is similar to GFS, although there are differences in terminology. Here is the comparison between the two file systems:



	GFS	HDFS
Storage node	ChunkServer	DataNode
File part	Chunk	Block
File part size	Default chunk size is 64MB (adjustable)	Default block size is 128MB
Metadata Checkpoint	Checkpoint image	FsImage
Write ahead log	Operation log	Edit log
Platform	Linux	Cross-platform
Language	Developed in C++	Developed in Java
Available Implementation	Only used internally by Google	OpenSource
Monitoring	Master receives HeartBeat from ChunkServers	NameNode receives HeartBeat from DataNodes
Concurrency	Follow multiple writers and multiple readers model	Does not support multiple writers, follows the write-once mode
File Operations	Append and random writes are possible	Only append is possible
Garbage Collection	Any deleted file is renamed into a particular folder to be garbage collected later	Any deleted file is renamed into a particular folder to be garbage collected later
Communication	<p>RPC over TCP is used for communication with the master</p> <p>To minimize latency, pipelining and streaming are used over TCP for data transfer.</p>	<p>RPC over TCP is used for communication with the master</p> <p>For data transfer, pipelining and streaming are used over TCP</p>
Cache Management	<p>Client cache metadata</p> <p>Client or ChunkServer does not cache file data</p> <p>ChunkServers rely on the buffer cache in Linux to maintain frequently accessed data in memory</p>	<p>HDFS uses distributed caching</p> <p>User-specified paths are cached in the DataNode's memory (block cache)</p> <p>The cache could be private (for one user) or public (for all users)</p>

		<div><div><div><div>?</div></div><div><div>📄</div></div><div><div>⚙️</div></div></div><div>users of the system</div></div>
Replication Strategy	<p>Chunk replicas are spread across the racks. Master automatically replicates the chunks.</p> <p>By default, three copies of each chunk are stored. User can specify a different replication factor.</p> <p>The master re-replicates a chunk replica as soon as the number of available replicas falls below a user-specified number.</p>	<p>The HDFS has an automatic replication system.</p> <p>By default, two copies of each chunk are stored at two different Data Nodes in the same rack, and a third copy is stored on a Data Node in a different rack (for reliability).</p> <p>User can specify a different replication factor.</p>
File system Namespace	<p>Files are organized hierarchically in directories and identified by pathnames.</p>	<p>HDFS supports a traditional hierarchical file organization. Users can create directories to store files.</p> <p>HDFS also supports object storage systems such as Amazon S3 and Google Cloud Storage.</p>
Database	<p>Bigtable uses GFS as its storage engine.</p>	<p>HBase uses HDFS as its storage engine.</p>

← Back

Next →

Hadoop Distributed File System: Intro...

Deep Dive

✔ Completed

⚠ Report an Issue





Deep Dive

Let's explore some of HDFS's design components.

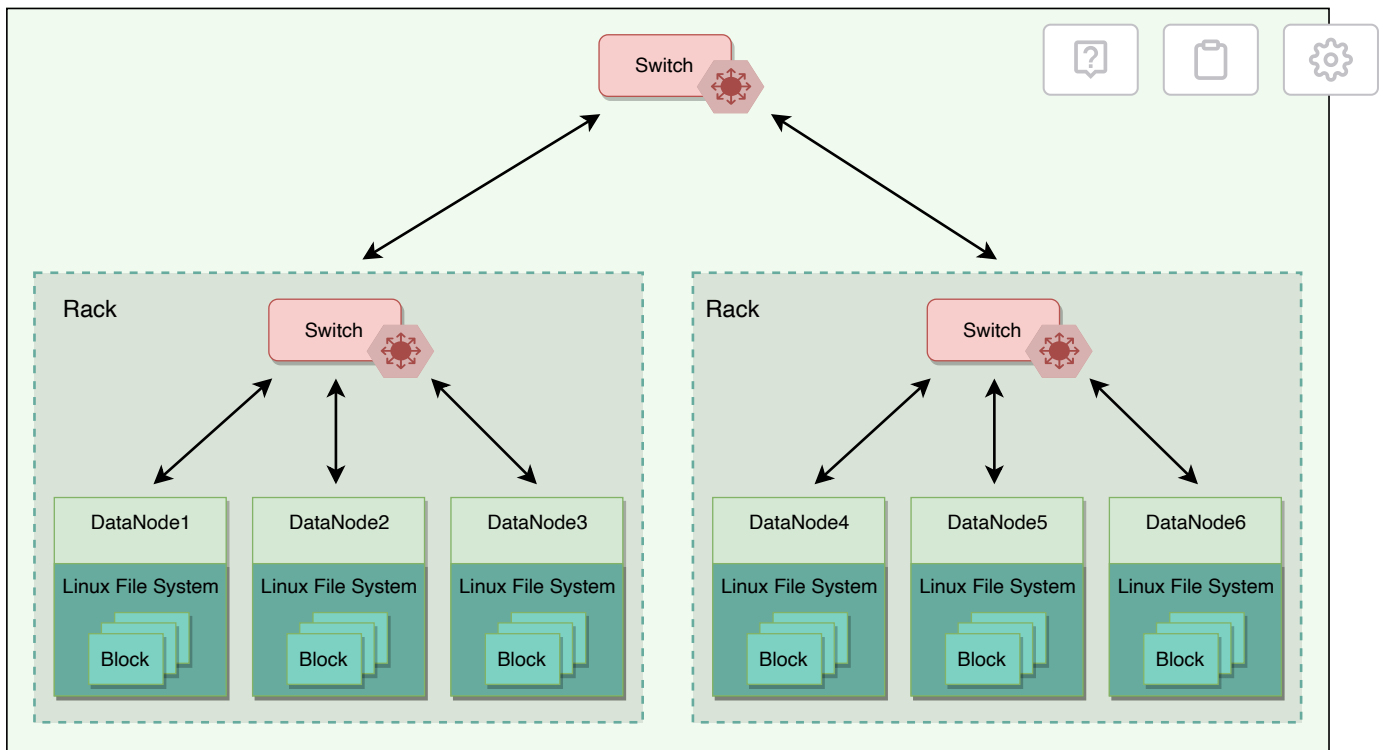
We'll cover the following ^

- Cluster topology
- Rack aware replication
- Synchronization semantics
- HDFS consistency model

Cluster topology#

A typical data center contains many racks of servers connected using switches. A common configuration for Hadoop clusters is to have about 30 to 40 servers per rack. Each rack has a dedicated gigabit switch that connects all of its servers and an uplink to a core switch or router, whose bandwidth is shared by many racks in the data center, as shown in the following figure.





HDFS cluster topology

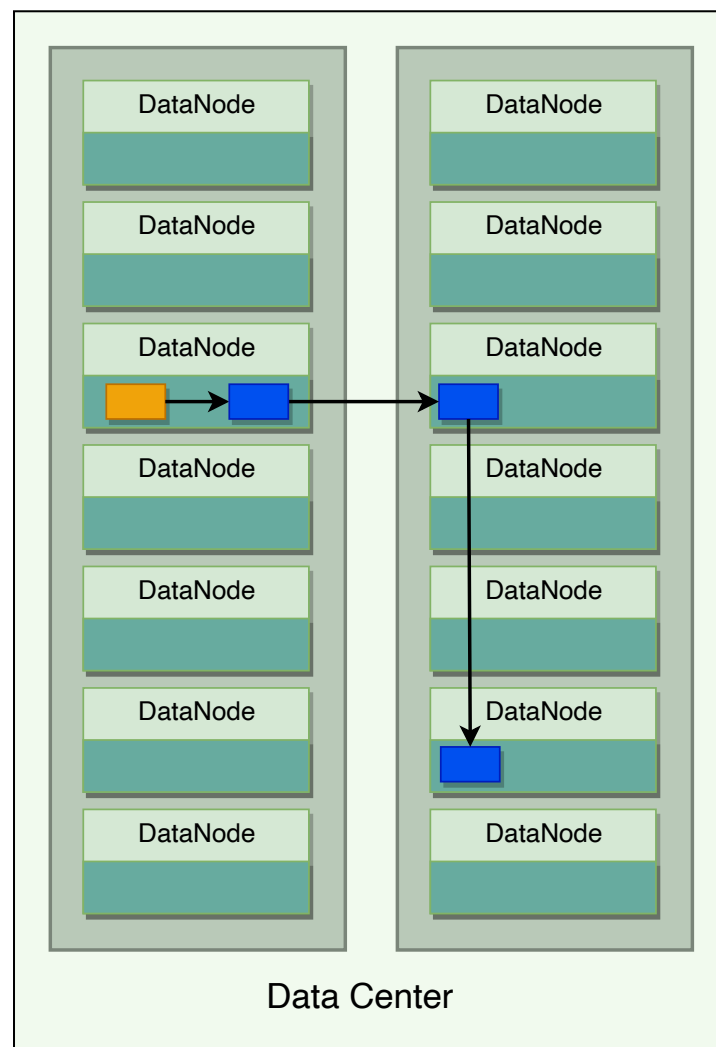
When HDFS is deployed on a cluster, each of its servers is configured and mapped to a particular rack. The network distance between servers is measured in hops, where one hop corresponds to one link in the topology. Hadoop assumes a tree-style topology, and the distance between two servers is the sum of their distances to their closest common ancestor.

In the above figure, the distance between Node 1 and itself is zero hops (the case when two processes are communicating on the same node). Node 1 and Node 2 are two hops away, while the distance between Node 3 and Node 4 is four hops.

Rack aware replication#

The placement of replicas is critical to HDFS reliability and performance. HDFS employs a rack-aware replica placement policy to improve data reliability, availability, and network bandwidth utilization. If the replication factor is three, HDFS attempts to place the first replica on the same node as ☀ the client writing the block. In case a client process is not running in the




HDFS cluster, a node is chosen at random. The second replica is written to a random node on a different rack from the first (i.e., off-rack replica). The third replica of the block is then written to another random node on the same rack as the second. Additional replicas are written to random nodes in the cluster, but the system tries to avoid placing too many replicas on the same rack. The figure below illustrates the replica placement for a triple-replicated block in HDFS. The idea behind HDFS's replica placement is to be able to tolerate node and rack failures. For example, when an entire rack goes offline due to power or networking problems, the requested block can still be located at a different rack.



Rack-aware replication

The default HDFS replica placement policy can be summarized as follows:



1. No DataNode will contain more than one replica of any   
2. If there are enough racks available, no rack will contain more than two replicas of the same block.

Following this rack-aware replication scheme slows the write operation as the data needs to be replicated onto different racks, but this is an intentional tradeoff between reliability and performance that HDFS made.

Synchronization semantics#

Early versions of HDFS followed strict **immutable semantics**. Once a file was written, it could never again be re-opened for writes; files could still be deleted. However, current versions of HDFS support append. This is still quite limited in the sense that existing binary data once written to HDFS cannot be modified in place.

This design choice in HDFS was made because some of the most common MapReduce workloads follow the **write once, read many data-access** pattern. MapReduce is a restricted computational model with predefined stages. The reducers in MapReduce write independent files to HDFS as output. HDFS focuses on fast read access for multiple clients at a time.

HDFS consistency model#

HDFS follows a **strong consistency model**. As stated above, each data block written to HDFS is replicated to multiple nodes. To ensure strong consistency, a write is declared successful only when all replicas have been written successfully. This way, all clients see the same (and consistent) view of the file. Since HDFS does not allow multiple concurrent writers to write to an HDFS file, implementing strong consistency becomes a relatively easy task.



 **Back**

High-level Architecture







Next

Anatomy of a Read Operation

 **Completed**

 Report an Issue





Anatomy of a Read Operation

We'll cover the following



- HDFS read process
- Short circuit read

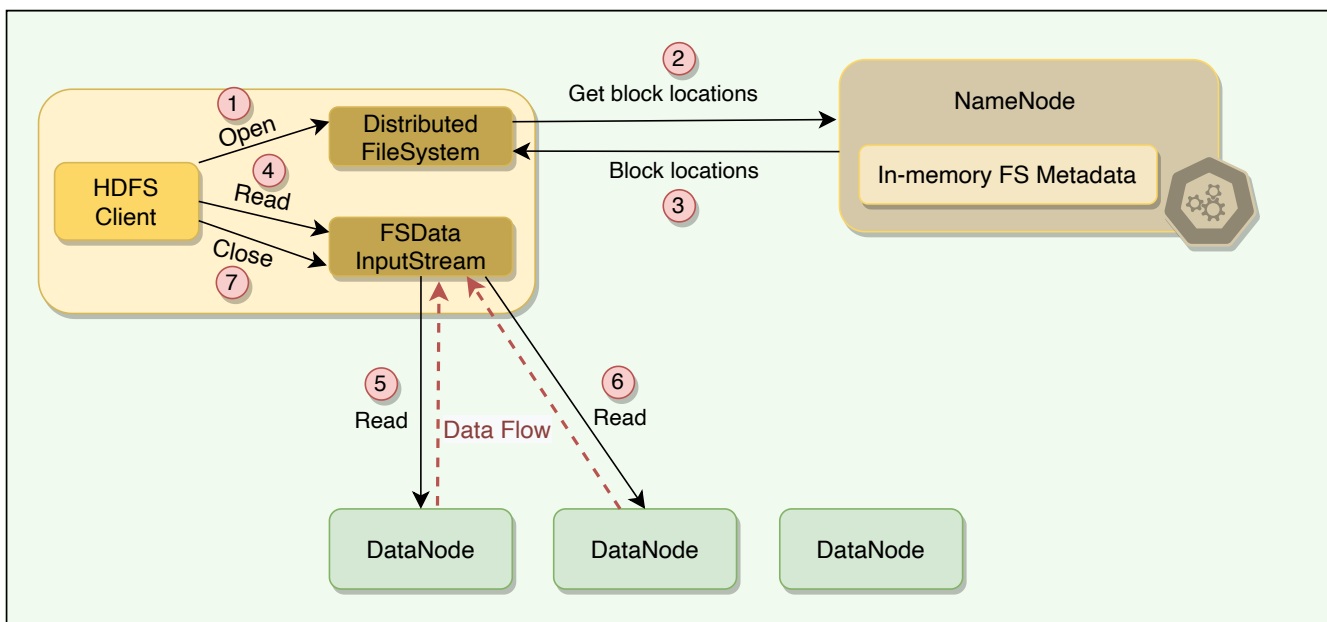
HDFS read process#

HDFS read process can be outlined as follows:

1. When a file is opened for reading, HDFS client initiates a read request, by calling the `open()` method of the `Distributed FileSystem` object. The client specifies the file name, start offset, and the read range length.
2. The `Distributed FileSystem` object calculates what blocks need to be read based on the given offset and range length, and requests the locations of the blocks from the `NameNode`.
3. `NameNode` has metadata for all blocks' locations. It provides the client a list of blocks and the locations of each block replica. As the blocks are replicated, `NameNode` finds the closest replica to the client when providing a particular block's location. The closest locality of each block is determined as follows:
 - If a required block is within the same node as the client, it is preferred.
 - Then, the block in the same rack as the client is preferred.
 - Finally, an off-rack block is read.



4. After getting the block locations, the client calls the `read()` method of `FSDa` `InputStream`, which takes care of all the interactions with the `DataNodes`. In step 4 in the below diagram, once the client invokes the `read()` method, the input stream object establishes a connection with the closest `DataNode` with the first block of the file.
5. The data is read in the form of streams. As the data is streamed, it is passed to the requesting application. Hence, the block does not have to be transferred in its entirety before the client application starts processing it.
6. Once the `FSDa` `InputStream` receives all data of a block, it closes the connection and moves on to connect the `DataNode` for the next block. It repeats this process until it finishes reading all the required blocks of the file.
7. Once the client finishes reading all the required blocks, it calls the `close()` method of the input stream object.



The anatomy of a read operation

Short circuit read#



As we saw above, the client reads the data directly from DataNode. The client uses TCP sockets for this. If the data and the client are on the same machine, HDFS can directly read the file bypassing the DataNode. This scheme is called short circuit read and is quite efficient as it reduces overhead and other processing resources.

[← Back](#)[Next →](#)[Deep Dive](#)[Anatomy of a Write Operation](#) **Completed** [Report an Issue](#)



Anatomy of a Write Operation

This lesson explains how HDFS handles a write operation.

We'll cover the following



- HDFS write process

HDFS write process#

HDFS write process can be outlined as follows:

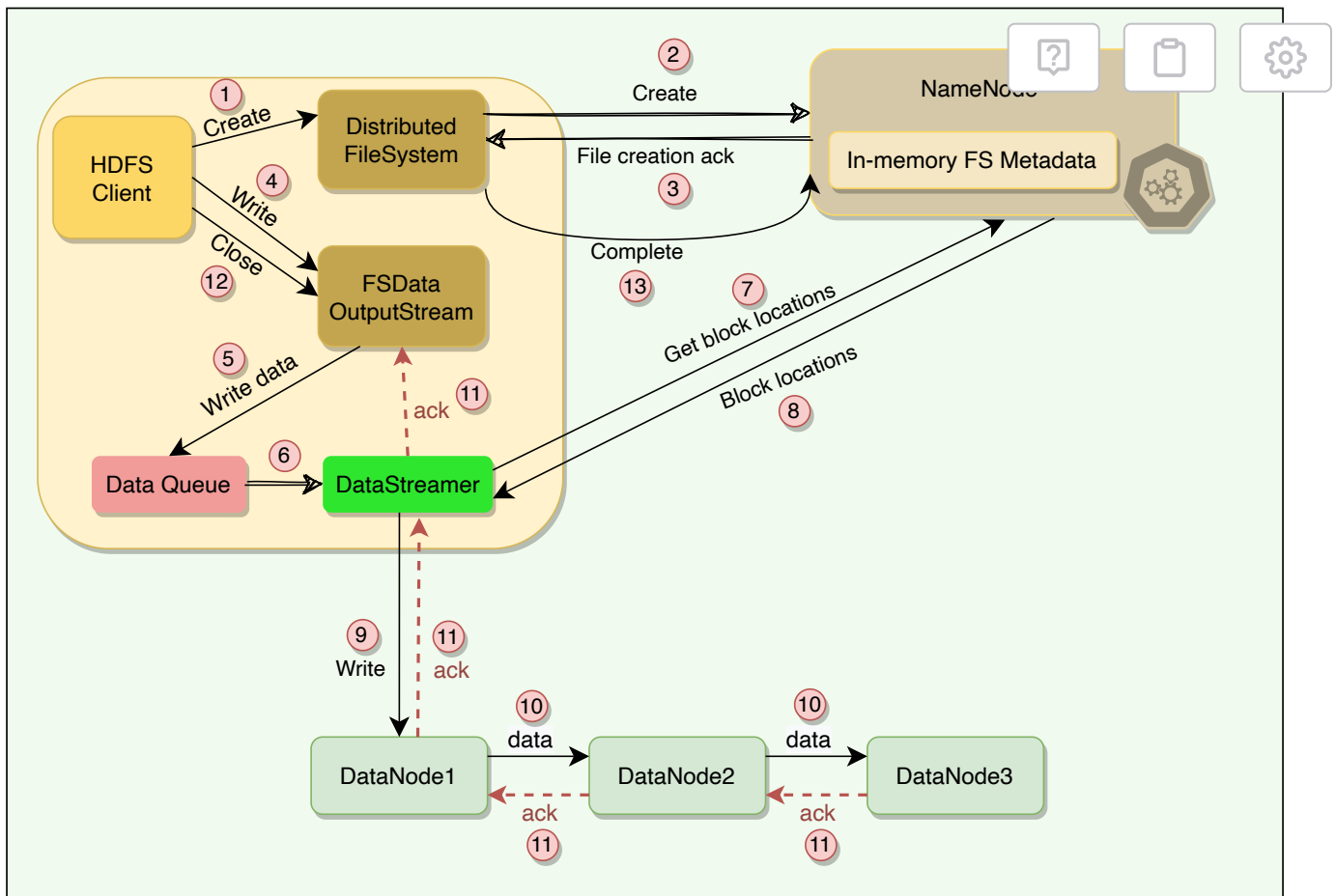
1. HDFS client initiates a write request by calling the `create()` method of the `Distributed FileSystem` object.
2. The `Distributed FileSystem` object sends a file creation request to the `NameNode`.
3. The `NameNode` verifies that the file does not already exist and that the client has permission to create the file. If both these conditions are verified, the `NameNode` creates a new file record and sends an acknowledgment.
4. The client then proceeds to write the file using `FSDa Output Stream`
5. The `FSDa Output Stream` writes data to a local queue called 'Data Queue.' The data is kept in the queue until a complete block of data is accumulated.
6. Once the queue has a complete block, another component called `DataStreamer` is notified to manage data transfer to the `DataNode`.





7. DataStreamer first asks the NameNode to allocate a new DataNodes, thereby picking desirable DataNodes to be used for replication.
8. The NameNode provides a list of blocks and the locations of each block replica.
9. Upon receiving the block locations from the NameNode, the DataStreamer starts transferring the blocks from the internal queue to the nearest DataNode.
10. Each block is written to the first DataNode, which then pipelines the block to other DataNodes in order to write replicas of the block. This way, the blocks are replicated during the file write itself. It is important to note that HDFS does not acknowledge a write to the client until all the replicas for that block have been written by the DataNodes.
11. Once the DataStreamer finishes writing all blocks, it waits for acknowledgments from all the DataNodes.
12. Once all acknowledgments are received, the client calls the `close()` method of the `OutputStream`.
13. Finally, the Distributed FileSystem contacts the NameNode to notify that the file write operation is complete. At this point, the NameNode commits the file creation operation, which makes the file available to be read. If the NameNode dies before this step, the file is lost.



[< Back](#)[Next >](#)[Anatomy of a Read Operation](#)[Data Integrity & Caching](#)☒ Completed[Report an Issue](#)



Data Integrity & Caching

Let's explore how HDFS ensures data integrity and implements caching.

We'll cover the following




- Data integrity
 - Block scanner
- Caching

Data integrity#

Data Integrity refers to ensuring the correctness of the data. When a client retrieves a block from a DataNode, the data may arrive corrupted. This corruption can occur because of faults in the storage device, network, or the software itself. HDFS client uses checksum to verify the file contents. When a client stores a file in HDFS, it computes a checksum of each block of the file and stores these checksums in a separate hidden file in the same HDFS namespace. When a client retrieves file contents, it verifies that the data it received from each DataNode matches the checksum stored in the associated checksum file. If not, then the client can opt to retrieve that block from another replica.

Block scanner#

A block scanner process periodically runs on each DataNode to scan blocks stored on that DataNode and verify that the stored checksums match the  block data. Additionally, when a client reads a complete block and checksum

verification succeeds, it informs the DataNode. The DataNode



verification of the replica. Whenever a client or a block scanner detects a corrupt block, it notifies the NameNode. The NameNode marks the replica as corrupt and initiates the process to create a new good replica of the block.

Caching#

Normally, blocks are read from the disk, but for frequently accessed files, blocks may be explicitly cached in the DataNode's memory, in an off-heap block cache. HDFS offers a Centralized Cache Management scheme to allow its users to specify paths to be cached. Clients can tell the NameNode which files to cache. NameNode communicates with the DataNodes that have the desired blocks on disk and instructs them to cache the blocks in off-heap caches.

Centralized cache management in HDFS has many significant advantages:

1. Explicitly specifying blocks for caching prevents the eviction of frequently accessed data from memory. This is particularly important as most of the HDFS workloads are bigger than the main memory of the DataNode.
2. Because the NameNode manages DataNode caches, applications can query the set of cached block locations when making MapReduce task placement decisions. Co-locating a task with a cached block replica improves read performance.
3. When a DataNode has cached a block, clients can use a new, more efficient, zero-copy read API. As the block is already in memory and its checksum verification has already been done by the DataNode, clients can incur essentially zero overhead when using this new API.
4. Centralized caching can improve overall cluster memory utilization. When relying on the OS buffer cache at each DataNode, repeated reads of a block will result in all 'n' replicas of the block being pulled into the



of a block will result in all n replicas of the block being pulled into the



buffer cache. With centralized cache management, a user can explicitly specify only ‘ m ’ of the ‘ n ’ replicas, saving ‘ $n-m$ ’ memory.

[← Back](#)[Next →](#)[Anatomy of a Write Operation](#)[Fault Tolerance](#) **Completed** [Report an Issue](#)



Fault Tolerance

Let's explore what techniques HDFS uses for fault tolerance.

We'll cover the following



- How does HDFS handle DataNode failures?
 - Replication
 - HeartBeat
- What happens when the NameNode fails?
 - FsImage and EditLog
 - Metadata backup

How does HDFS handle DataNode failures?#

Replication#

When a DataNode dies, all of its data becomes unavailable. HDFS handles this data unavailability through replication. As stated earlier, every block written to HDFS is replicated to multiple (default three) DataNodes. Therefore, if one DataNode becomes inaccessible, its data can be read from other replicas.

HeartBeat#



The NameNode keeps track of DataNodes through a heartbeat mechanism. Each DataNode sends periodic heartbeat messages (every few seconds) to the NameNode. If a DataNode dies, the heartbeats will stop, and the NameNode will detect that the DataNode has died. The NameNode will then mark the DataNode as dead and will no longer forward any read/write request to that DataNode. Because of replication, the blocks stored on that DataNode have additional replicas on other DataNodes. The NameNode performs regular status checks on the file system to discover under-replicated blocks and performs a **cluster rebalance** process to replicate blocks that have less than the desired number of replicas.




What happens when the NameNode fails?#

FsImage and EditLog#

The NameNode is a **single point of failure (SPOF)**. A NameNode failure will bring the entire file system down. Internally, the NameNode maintains two on-disk data structures that store the file system's state: an **FsImage** file and an **EditLog**. FsImage is a checkpoint (or the image) of the file system metadata at some point in time, while the EditLog is a log of all of the file system metadata transactions since the image file was last created. All incoming changes to the file system metadata are written to the EditLog. At periodic intervals, the EditLog and FsImage files are merged to create a new image file snapshot, and the edit log is cleared out.

Metadata backup#

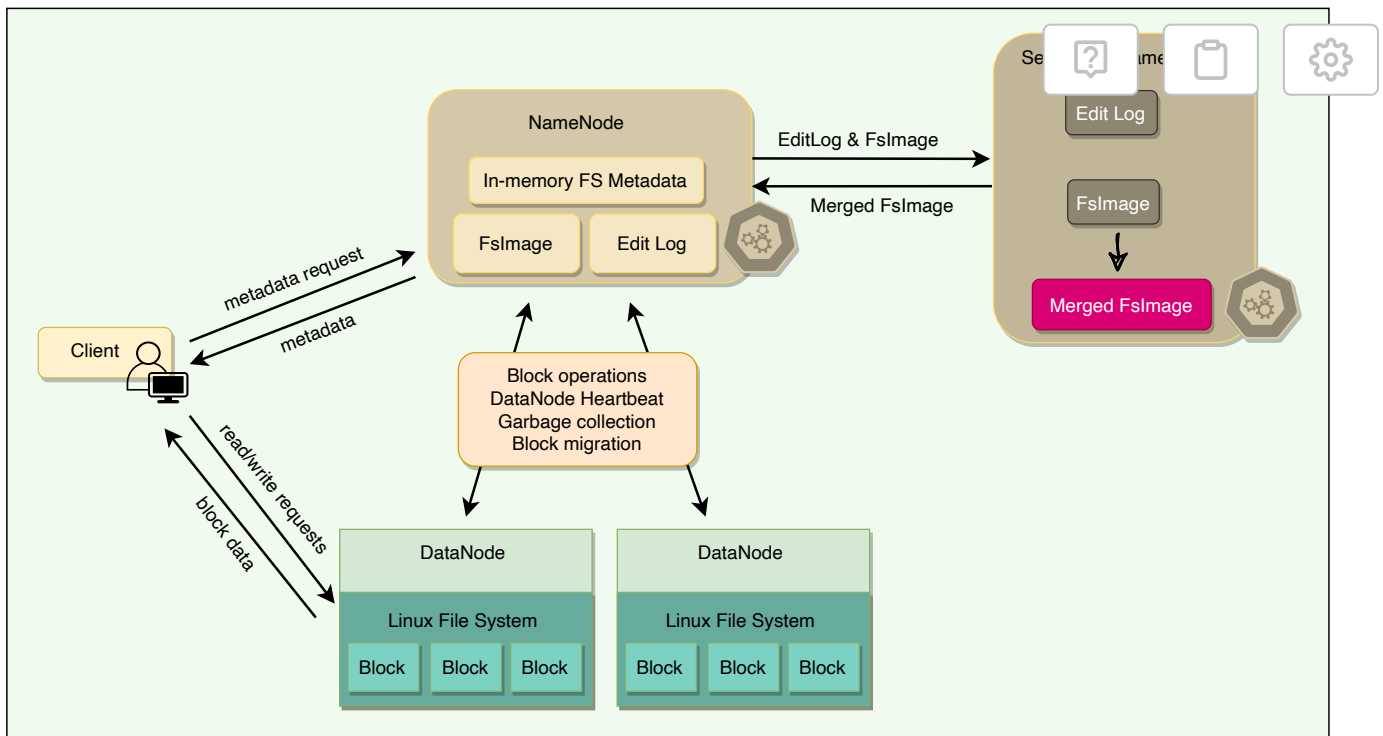
On a NameNode failure, the metadata would be unavailable, and a disk failure on the NameNode would be catastrophic because the file metadata would be lost since there would be no way of knowing how to reconstruct 

the files from the blocks on the DataNodes. For this reason, it is important to make the NameNode resilient to failure, and HDFS provides two mechanisms for this:



1. The first way is to back up and store multiple copies of FsImage and EditLog. The NameNode can be configured to maintain multiple copies of the files. Any update to either the FsImage or EditLog causes each copy of the FsImages and EditLogs to get updated synchronously and atomically. A common configuration is to maintain one copy of these files on a local disk and one on a remote Network File System (NFS) mount. This synchronous updating of multiple copies of the FsImage and EditLog may degrade the rate of namespace transactions per second that a NameNode can support. However, this degradation is acceptable because even though HDFS applications are very data-intensive, they are not metadata-intensive.
2. Another option provided by HDFS is to run a **Secondary NameNode**, which despite its name, is not a backup NameNode. Its main role is to help primary NameNode in taking the checkpoint of the filesystem. Secondary NameNode periodically merges the namespace image with the EditLog to prevent the EditLog from becoming too large. The secondary NameNode runs on a separate physical machine because it requires plenty of CPU and as much memory as the NameNode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the NameNode failure. However, the state of the secondary NameNode lags behind that of the primary, so in the event of total failure of the primary, data loss is almost inevitable. The usual course of action, in this case, is to copy the NameNode's metadata files that are on NFS to the secondary and run it as the new primary.





Role of primary and secondary NameNode

[← Back](#)[Next →](#)[Data Integrity & Caching](#)[HDFS High Availability \(HA\)](#)☒ Completed[Report an Issue](#)



HDFS High Availability (HA)

Let's learn how HDFS achieves high availability.

We'll cover the following



- HDFS high availability architecture
 - QJM
 - Zookeeper
- Failover and fencing
 - Fencing

HDFS high availability architecture#

Although NameNode's metadata is copied to multiple file systems to protect against data loss, it still does not provide high availability of the filesystem. If the NameNode fails, no clients will be able to read, write, or list files, because the NameNode is the sole repository of the metadata and the file-to-block mapping. In such an event, the whole Hadoop system would effectively be out of service until a new NameNode is brought online.

To recover from a failed NameNode scenario, an administrator will start a new primary NameNode with one of the filesystem metadata replicas and configure DataNodes and clients to use this new NameNode. The new NameNode is not able to serve requests until it has

1. loaded its namespace image into memory,
2. replayed its EditLog, and



3. received enough block reports from the DataNodes.



On large clusters with many files and blocks, it can take half an hour or more to perform a cold start of a NameNode. Furthermore, this long recovery time is a problem for routine maintenance. In fact, because an unexpected failure of the NameNode is rare, the case for planned downtime is actually more important in practice.

To solve this problem, Hadoop, in its 2.0 release, added support for HDFS High Availability (HA). In this implementation, there are two (or more) NameNodes in an active-standby configuration. At any point in time, exactly one of the NameNodes is in an active state, and the others are in a Standby state. The active NameNode is responsible for all client operations in the cluster, while the Standby is simply acting as a follower of the active, maintaining enough state to provide a fast failover when required.

For the Standby nodes to keep their state synchronized with the active node, HDFS made a few architectural changes:


- The NameNodes must use highly available shared storage to share the EditLog (e.g., a Network File System (NFS) mount from a Network Attached Storage (NAS)).
- When a standby NameNode starts, it reads up to the end of the shared EditLog to synchronize its state with the active NameNode, and then continues to read new entries as the active NameNode writes them.
- DataNodes must send block reports to all the NameNodes because the block mappings are stored in a NameNode's memory, and not on disk.
- Clients must be configured to handle NameNode failover, using a mechanism that is transparent to users. Client failover is handled transparently by the client library. The simplest implementation uses client-side configuration to control failover. The HDFS URI uses a logical hostname which is mapped to multiple NameNode addresses, and the client library tries each NameNode address until the operation succeeds. ☀

There are two choices for the highly available shared storage (as described above), or a **Quorum Journal Manager (QJM)**.



QJM#

The sole purpose of the QJM is to provide a highly available EditLog. The QJM runs as a group of journal nodes, and each edit must be written to a quorum (or majority) of the journal nodes. Typically, there are three journal nodes, so that the system can tolerate the loss of one of them. This arrangement is similar to the way [ZooKeeper](#) works, although it is important to realize that the QJM implementation does not use ZooKeeper.

 **Note:** HDFS High Availability does use ZooKeeper for electing the active NameNode. More details on this later. QJM process runs on all NameNodes and communicates all EditLog changes to journal nodes using RPC.

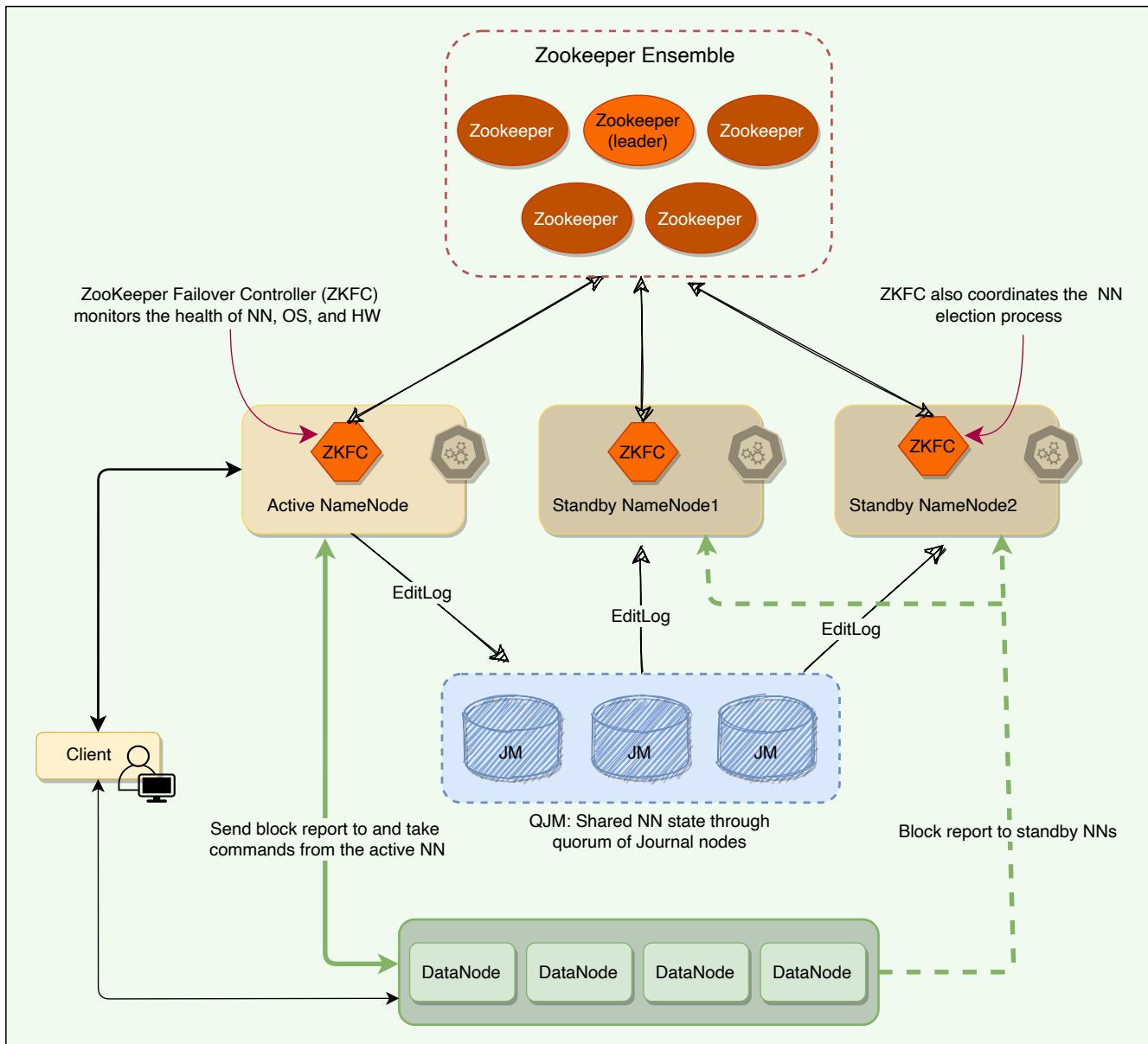
Since the Standby NameNodes have the latest state of the metadata available in memory (both the latest EditLog and an up-to-date block mapping), any standby can take over very quickly (in a few seconds) if the active NameNode fails. However, the actual failover time will be longer in practice (around a minute or so) because the system needs to be conservative in deciding that the active NameNode has failed.

In the unlikely event of the Standbys being down when the active fails, the administrator can still do a cold start of a Standby. This is no worse than the non-HA case.

Zookeeper#

The ZKFailoverController (ZKFC) is a ZooKeeper client that runs on each NameNode and is responsible for coordinating with the Zookeeper and also





Failover and fencing#



NameNode for failures (using Heartbeat), and triggers a failover when the active NameNode fails.



Graceful failover: For routine maintenance, an administrator can manually initiate a failover. This is known as a graceful failover, since the failover controller arranges an orderly transition from the active NameNode to the Standby.

Ungraceful failover: In the case of an ungraceful failover, however, it is impossible to be sure that the failed NameNode has stopped running. For example, a slow network or a network partition can trigger a failover transition, even though the previously active NameNode is still running and thinks it is still the active NameNode.

The HA implementation uses the mechanism of **Fencing** to prevent this “**split-brain**” scenario and ensure that the previously active NameNode is prevented from doing any damage and causing corruption.

Fencing#

Fencing is the idea of putting a fence around a previously active NameNode so that it cannot access cluster resources and hence stop serving any read/write request. To apply fencing, the following two techniques are used:

- **Resource fencing:** Under this scheme, the previously active NameNode is blocked from accessing resources needed to perform essential tasks. For example, revoking its access to the shared storage directory (typically by using a vendor-specific NFS command), or disabling its network port via a remote management command.
- **Node fencing:** Under this scheme, the previously active NameNode is blocked from accessing all resources. A common way of doing this is to power off or reset the node. This is an effective method of keeping it



from accessing anything at all. This technique is also called

“Shoot The Other Node In The Head.”



To learn more about automatic failover, take a look at [Apache documentation](#).

← Back

Fault Tolerance

Next →

HDFS Characteristics

✓ Completed

⚠ Report an Issue





HDFS Characteristics

This lesson will explore some important aspects of HDFS architecture.

We'll cover the following ^

- Security and permission
- HDFS federation
- Erasure coding
- HDFS in practice

Security and permission#

HDFS provides a permissions model for files and directories which is similar to POSIX. Each file and directory is associated with an **owner** and a **group**. Each file or directory has separate permissions for the owner, other users who are members of a group, and all other users. There are three types of permission:

1. **Read permission (r)**: For files, r permission is required to read a file. For directories, r permission is required to list the contents of a directory.
2. **Write permission (w)**: For files, w permission is required to write or append to a file. For a directory, w permission is required to create or delete files or directories in it.
3. **Execute permission (x)**: For files, x permission is ignored as we cannot execute a file on HDFS. For a directory, x permission is required to access a child of the directory.

access a child of the directory.




HDFS also provides optional support for POSIX ACLs (Access Control Lists) to augment file permissions with finer-grained rules for specific named users or named groups.

HDFS federation#

The NameNode keeps the metadata of the whole namespace in memory, which means that on very large clusters with many files, the memory becomes the limiting factor for scaling. A more serious problem is that a single NameNode, serving all metadata requests, can become a performance bottleneck. To help resolve these issues, HDFS Federation was introduced in the 2.x release, which allows a cluster to scale by adding NameNodes, each of which manages a portion of the filesystem namespace. For example, one NameNode might manage all the files rooted under `/user`, and a second NameNode might handle files under `/share`. Under federation:

- All NameNodes work independently. No coordination is required between NameNodes.
- DataNodes are used as the common storage by all the NameNodes.
- A NameNode failure does not affect the availability of the namespaces managed by other NameNodes.
- To access a federated HDFS cluster, clients use client-side mount tables to map file paths to NameNodes.

Multiple NameNodes running independently can end up generating the same **64-bit Block IDs** for their blocks. To avoid this problem, a namespace uses one or more **Block Pools**, where a unique ID identifies each block pool in a cluster. A block pool belongs to a single namespace and does not cross the namespace boundary. The extended block ID, which is a tuple of (Block Pool ID, Block ID), is used for block identification in HDFS Federation. 



Erasure coding#

By default, HDFS stores three copies of each block, resulting in a 200% overhead (to store two extra copies) in storage space and other resources (e.g., network bandwidth). Compared to this default replication scheme, Erasure Coding (EC) is probably the biggest change in HDFS in recent years. EC provides the same level of fault tolerance with much less storage space. In a typical EC setup, the storage overhead is no more than 50%. This fundamentally doubles the storage space capacity by bringing down the replication factor from 3x to 1.5x.

Under EC, data is broken down into fragments, expanded, encoded with redundant data pieces, and stored across different DataNodes. If, at some point, data is lost on a DataNode due to corruption, etc., then it can be reconstructed using the other fragments stored on other DataNodes. Although EC is more CPU intensive, it greatly reduces the storage needed for reliably storing a large data set.

For more details on how EC works, see [blog](#) or [wiki](#).

HDFS in practice#

Although HDFS was primarily designed to support Hadoop MapReduce jobs by providing a DFS for the Map and Reduce operations, HDFS has found many uses with big-data tools.

HDFS is used in several Apache projects that are built on top of the Hadoop framework, including Pig, Hive, HBase, and Giraph. HDFS support is also included in other projects, such as GraphLab.



The primary advantages of HDFS include the following:



- **High bandwidth for MapReduce workloads:** Large Hadoop clusters (thousands of machines) are known to continuously write up to one terabyte per second using HDFS.
- **High reliability:** Fault tolerance is a primary design goal in HDFS. HDFS replication provides high reliability and availability, particularly in large clusters, in which the probability of disk and server failures increases significantly.
- **Low costs per byte:** Compared to a dedicated, shared-disk solution such as a SAN, HDFS costs less per gigabyte because storage is collocated with compute servers. With SAN, we have to pay additional costs for managed infrastructure, such as the disk array enclosure and higher-grade enterprise disks, to manage hardware failures. HDFS is designed to run with commodity hardware, and redundancy is managed in software to tolerate failures.
- **Scalability:** HDFS allows DataNodes to be added to a running cluster and offers tools to manually rebalance the data blocks when cluster nodes are added, which can be done without shutting the file system down.

The primary disadvantages of HDFS include the following:

- **Small file inefficiencies:** HDFS is designed to be used with large block sizes (128MB and larger). It is meant to take large files (hundreds of megabytes, gigabytes, or terabytes) and chunk them into blocks, which can then be fed into MapReduce jobs for parallel processing. HDFS is inefficient when the actual file sizes are small (in the kilobyte range). Having a large number of small files places additional stress on the NameNode, which has to maintain metadata for all the files in the file system. Typically, HDFS users combine many small files into larger ones using techniques such as sequence files. A sequence file can be understood as a container of binary key-value pairs where the file



understood as a container of binary key value pairs, where the file name is the key, and the file contents are the value.



- **POSIX non-compliance:** HDFS was not designed to be a POSIX-compliant, mountable file system; applications will have to be either written from scratch or modified to use an HDFS client. Workarounds exist that enable HDFS to be mounted using a [FUSE](#) driver, but the file system semantics do not allow writes to files once they have been closed.
- **Write-once model:** The write-once model is a potential drawback for applications that require concurrent write accesses to the same file. However, the latest version of HDFS now supports file appends.

In short, HDFS is a good option as a storage backend for distributed applications that follow the MapReduce model or have been specifically written to use HDFS. HDFS can be used efficiently with a small number of large files rather than a large number of small files.

[← Back](#)[Next →](#)[HDFS High Availability \(HA\)](#)[Summary: HDFS](#) **Completed**[Report an Issue](#)



Summary: HDFS

Here is a quick summary of HDFS for you!

We'll cover the following







- Summary
- System design patterns
- References and further reading

Summary#

- HDFS is a scalable distributed file system for large, distributed data-intensive applications.
- HDFS uses commodity hardware to reduce infrastructure costs.
- HDFS provides APIs for usual file operations like create, delete, open, close, read, and write.
- Random writes are not possible; writes are always made at the end of the file in an append-only fashion.
- HDFS does not support multiple concurrent writers.
- An HDFS cluster consists of a **single NameNode** and **multiple DataNodes** and is accessed by multiple clients.
- **Block:** Files are broken into fixed-size blocks (default 128MB), and blocks are replicated across a number of DataNodes to ensure fault-tolerance. The block size and the replication factor are configurable.
- DataNodes store blocks on local disk as Linux files.



- NameNode server is the coordinator of an HDFS cluster and is

- **NameNode** server is the coordinator of an HDFS cluster and is responsible for keeping track of all filesystem metadata.   
- NameNode keeps all metadata in memory for faster operations. For fault-tolerance and in the event of NameNode crash, all metadata changes are written to the disk onto an **EditLog**. This EditLog can also be replicated on a remote filesystem (e.g., NFS) or a secondary NameNode.
- The NameNode does not keep a persistent record of which DataNodes have a replica of a given block. Instead, the NameNode asks each DataNode about what blocks it holds at NameNode startup and whenever a DataNode joins the cluster.
- **FsImage**: The NameNode state is periodically serialized to disk and then replicated, so that on recovery, a NameNode may load the checkpoint into memory, replay any subsequent operations from the edit log, and be available again very quickly.
- **HeartBeat**: The NameNode communicates with each DataNode through Heartbeat messages to pass instructions and collect its state.
- **Client**: User applications interact with HDFS through its client. HDFS Client interacts with NameNode for metadata, but all data transfers happen directly between the client and DataNodes.
- **Data Integrity**: Each DataNode uses checksumming to detect the corruption of stored data.
- **Garbage Collection**: Any deleted file is renamed to a hidden name to be garbage collected later.
- **Consistency**: HDFS is a strongly consistent file system. Each data block is replicated to multiple nodes, and a write is declared to be successful only after all the replicas have been written successfully.
- **Cache**: For frequently accessed files, the blocks may be explicitly cached in the DataNode's memory, in an off-heap block cache.
- **Erasure coding**: HDFS uses erasure coding to reduce replication overhead. 



System design patterns#

Here is a summary of system design patterns used in HDFS.

- **Write-Ahead Log:** For fault tolerance and in the event of NameNode crash, all metadata changes are written to the disk onto an EditLog which is a write-ahead log.
- **HeartBeat:** The HDFS NameNode periodically communicates with each DataNode in HeartBeat messages to give it instructions and collect its state.
- **Split-Brain:** ZooKeeper is used to ensure that only one NameNode is active at any time. Fencing is used to put a fence around a previously active NameNode so that it cannot access cluster resources and hence stop serving any read/write request.
- **Checksum:** Each DataNode uses checksumming to detect the corruption of stored data.

References and further reading#

- [HDFS paper](#)
- [HDFS High Availability \(HA\)architecture](#)
- [Apache HDFS Architecture](#)
- [Distributed File Systems: A Survey](#)

[← Back](#)

HDFS Characteristics

[Next → !\[\]\(4688aadfd656ded00cd6bdfae55089a9_img.jpg\)](#)

Quiz: HDFS



 Report an Issue

