3.1 What is Hadoop Distributed File System?

Hadoop File System was developed using distributed file system design. It is run on commodity hardware. Unlike other distributed systems, HDFS is highly fault tolerant and designed using low-cost hardware.

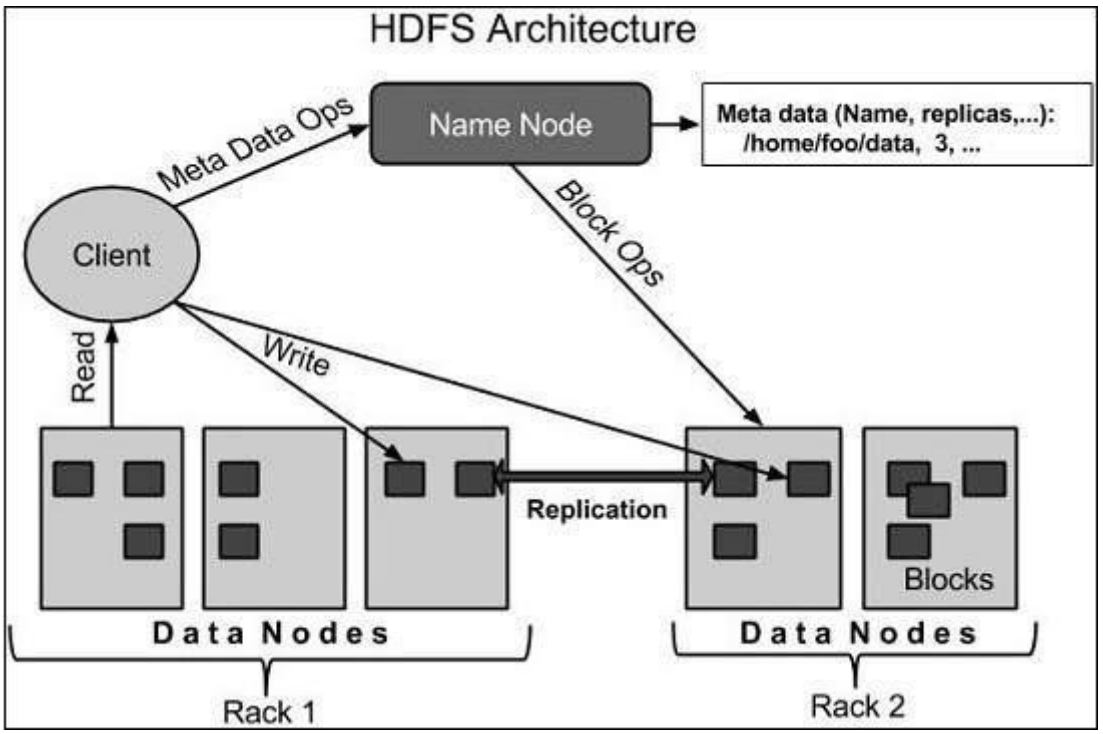
HDFS holds very large amount of data and provides easier access. To store such huge data, the files are stored across multiple machines. These files are stored in redundant fashion to rescue the system from possible data losses in case of failure. HDFS also makes applications available to parallel processing.

Features of HDFS

- It is suitable for the distributed storage and processing.
- Hadoop provides a command interface to interact with HDFS.
- The built-in servers of namenode and datanode help users to easily check the status of cluster.
- Streaming access to file system data.
- HDFS provides file permissions and authentication.

HDFS Architecture/ Design

Given below is the architecture of a Hadoop File System.



HDFS follows the master-slave architecture and it has the following elements.

Namenode

The namenode is the commodity hardware that contains the GNU/Linux operating system and the namenode software. It is a software that can be run on commodity hardware. The system having the namenode acts as the master server and it does the following tasks −

- Manages the file system namespace.
- Regulates client's access to files.
- It also executes file system operations such as renaming, closing, and opening files and directories.

Datanode

The datanode is a commodity hardware having the GNU/Linux operating system and datanode software. For every node (Commodity hardware/System) in a cluster, there will be a datanode. These nodes manage the data storage of their system.

- Datanodes perform read-write operations on the file systems, as per client request.
- They also perform operations such as block creation, deletion, and replication according to the instructions of the namenode.

Block

Generally the user data is stored in the files of HDFS. The file in a file system will be divided into one or more segments and/or stored in individual data nodes. These file segments are called as blocks. In other words, the minimum amount of data that HDFS can read or write is called a Block. The default block size is 64MB, but it can be increased as per the need to change in HDFS configuration.

Goals of HDFS

Fault detection and recovery − Since HDFS includes a large number of commodity hardware, failure of components is frequent. Therefore HDFS should have mechanisms for quick and automatic fault detection and recovery.

Huge datasets − HDFS should have hundreds of nodes per cluster to manage the applications having huge datasets.

Hardware at data − A requested task can be done efficiently, when the computation takes place near the data. Especially where huge datasets are involved, it reduces the network traffic and increases the throughput.

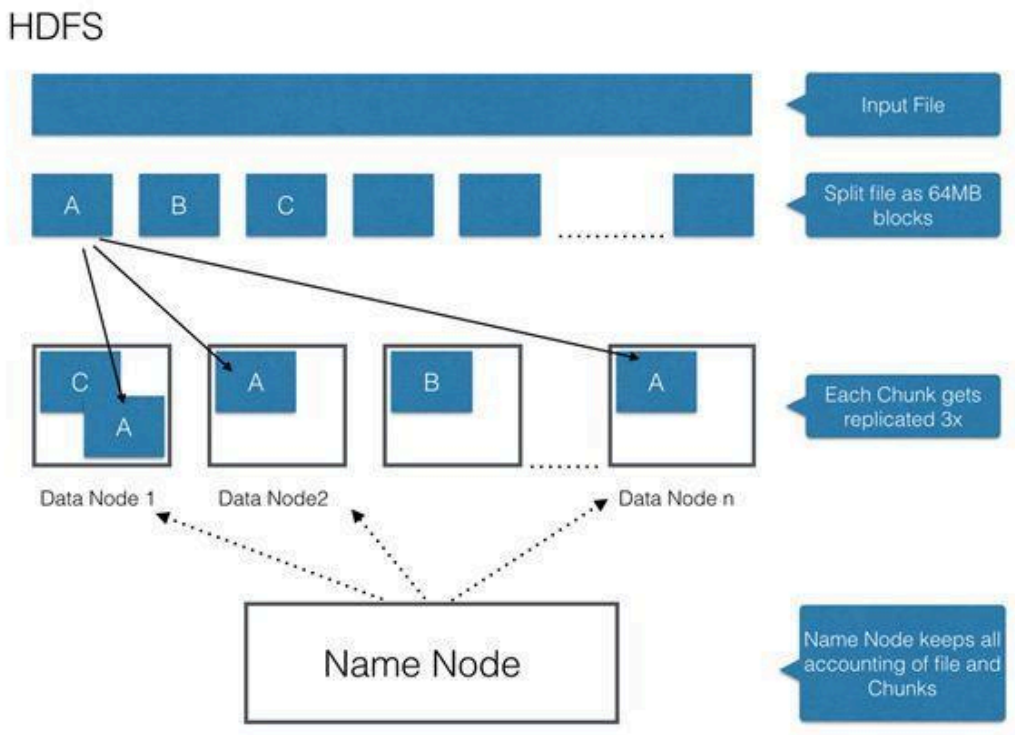
3.1.2 HDFS Concept

3.1.3 How files are stored in HDFS

Let's say we need to move a 1 Gig text file to HDFS.

1. HDFS will split the file into 64 MB blocks.
    a. The size of the blocks can be configured.
    b. An entire block of data will be used in the computation.
    c. Think of it as a sector on a hard disk.
2. Each block will be sent to 3 machines (data nodes) for storage.
    a. This provides reliability and efficient data processing.
    b. Replication factor of 3 is configurable.
    c. RAID configuration to store the data is not required.
    d. Since data is replicated 3 times the overall storage space is reduced a third.
3. The accounting of each block is stored in a central server, called a Name Node.
    a. A Name Node is a master node that keeps track of each file and its corresponding blocks and the data node locations.
    b. Map Reduce will talk with the Name Node and send the computation to the corresponding data nodes.
    c. The Name Node is the key to all the data and hence the Secondary Name node is used to improve the reliability of the cluster.

HDFS in Picture



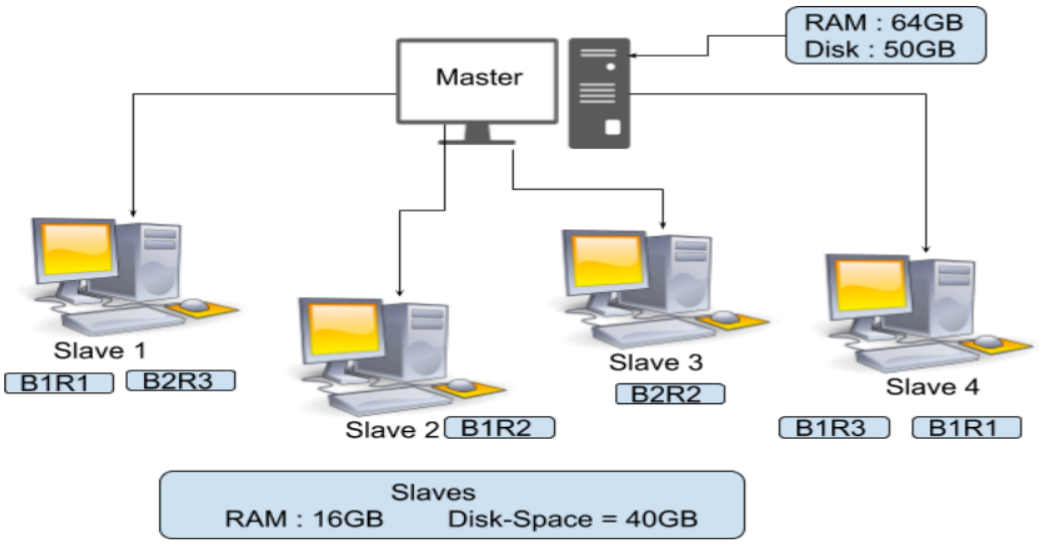
3.1.5 Replication factor

Replication ensures the availability of the data. Replication is nothing but making a copy of something and the number of times you make a copy of that particular thing can be expressed as its Replication Factor. As we have seen in File blocks that the HDFS stores the data in the form of various blocks at the same time Hadoop is also configured to make a copy of those file blocks. By default the Replication Factor for Hadoop is set to 3 which can be configured means you can change it Manually as per your requirement like in above example we have made 4 file blocks which means that 3 Replica or copy of each file block is made means total of 4×3 = 12 blocks are made for the backup purpose.

Now you might be getting a doubt that why we need this replication for our file blocks this is because for running Hadoop we are using commodity hardware (inexpensive system hardware) which can be crashed at any time. We are not using a supercomputer for our Hadoop setup. That is why we need such a feature in HDFS which can make copies of that file blocks for backup purposes, this is known as fault tolerance.

Now one thing we also need to notice that after making so many replica's of our file blocks we are wasting so much of our storage but for the big brand organization the data is very much important than the storage. So nobody care for this extra storage.

How does Replication work?



In the above image, you can see that there is a Master with RAM = 64GB and Disk Space = 50GB and 4 Slaves with RAM = 16GB, and disk Space = 40GB. Here you can observe that RAM for Master is more. It needs to be kept more because your Master is the one who is going to guide this slave so your Master has to process fast. Now suppose you have a file of size 150MB then the total file blocks will be 2 shown below.

128MB = Block 1

22MB = Block 2

As the replication factor by-default is 3 so we have 3 copies of this file block

FileBlock1-Replica1(B1R1) FileBlock2-Replica1(B2R1)

FileBlock1-Replica2(B1R2) FileBlock2-Replica2(B2R2)

FileBlock1-Replica3(B1R3) FileBlock2-Replica3(B2R3)

These blocks are going to be stored in our Slave as shown in the above diagram which means if suppose your Slave 1 crashed then in that case B1R1 and B2R3 get lost. But you can recover the B1 and B2 from other slaves as the Replica of this file blocks is already present in other slaves, similarly, if any other Slave got crashed then we can obtain that file block some other slave. Replication is going to increase our storage but Data is more necessary for us.

3.1.6 Name Node

3.1.7 Data Node

3.1.8 Secondary Name Node

3.1.9 Job Tracker

3.1.10 Task tracker

3.2 FS Image Edit-logs

FsImage is a file stored on the OS filesystem that contains the complete directory structure (namespace) of the HDFS with details about the location of the data on the Data Blocks and which blocks are stored on which node. This file is used by the NameNode when it is started.

EditLogs is a transaction log that records the changes in the HDFS file system or any action performed on the HDFS cluster such as addition of a new block, replication, deletion etc. In short, it records the changes since the last FsImage was created.

Every time the NameNode restarts, EditLogs are applied to FsImage to get the latest snapshot of the file system. But NameNode restarts are rare in production clusters. Because of this, you may encounter the following issues: .

- EditLog grows unwieldy in size, particularly where the NameNode runs for a long period of time without a restart;
- NameNode restart takes longer, as too many changes now have to be merged
- If the NameNode fails to restart (i.e., crashes), there will be significant data loss, as the FsImage used at the time of the restart is very old

Secondary Namenode helps to overcome the above issues by taking over the responsibility of merging EditLogs with FsImage from the NameNode.

- The Secondary NameNode obtains the FsImage and EditLogs from the NameNode at regular intervals.
- Secondary NameNoide loads both the FsImage and EditLogs to main memory and applies each operation from the EditLogs to the FsImage.
- Once a new FsImage is created, Secondary NameNode copies the image back to the NameNode.
- Namenode will use the new FsImage for the next restart, thus reducing startup time.

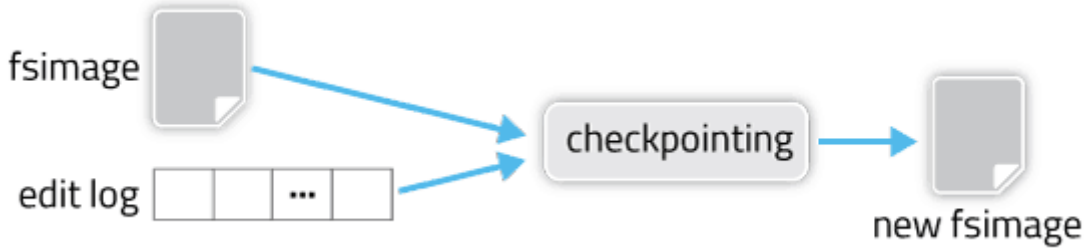
However, this seemingly fail-proof process is not without issues. Delays in the aforesaid process can cause a NameNode to startup without the latest FsImage at its disposal. Such delays can occur if:

- The Secondary NameNode takes too long to download the EditLogs from the NameNode;
- The NameNode is slow in uploading FsImages to the Secondary NameNode and/or in downloading the updated FsImages from the Secondary NameNode

3.2.1 Check-pointing Concept

A typical edit ranges from 10s to 100s of bytes, but over time enough edits can accumulate to become unwieldy. A couple of problems can arise from these large edit logs. In extreme cases, it can fill up all the available disk capacity on a node, but more subtly, a large edit log can substantially delay NameNode startup as the NameNode reapplies all the edits. This is where checkpointing comes in.

Checkpointing is a process that takes an fsimage and edit log and compacts them into a new fsimage. This way, instead of replaying a potentially unbounded edit log, the NameNode can load the final in-memory state directly from the fsimage. This is a far more efficient operation and reduces NameNode startup time.



However, creating a new fsimage is an I/O- and CPU-intensive operation, sometimes taking minutes to perform. During a checkpoint, the namesystem also needs to restrict concurrent access from other users. So, rather than pausing the active NameNode to perform a checkpoint, HDFS defers it to either the SecondaryNameNode or Standby NameNode, depending on whether NameNode high-availability is configured. The mechanics of checkpointing differs depending on if NameNode high-availability is configured; we'll cover both.

In either case though, checkpointing is triggered by one of two conditions: if enough time has elapsed since the last checkpoint (dfs.namenode.checkpoint.period), or if enough new edit log transactions have accumulated (dfs.namenode.checkpoint.txns). The checkpointing node periodically checks if either of these conditions is met (dfs.namenode.checkpoint.check.period), and if so, kicks off the checkpointing process.

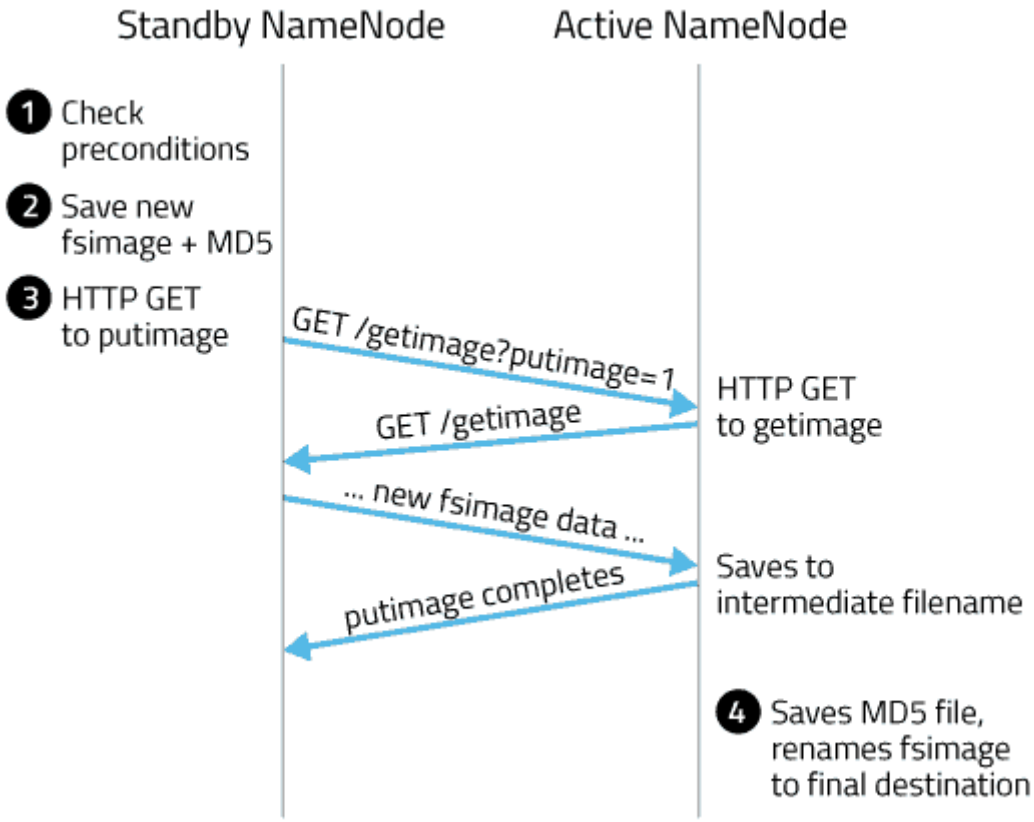
Checkpointing with a Standby NameNode

Checkpointing is actually much simpler when dealing with an HA setup, so let's cover that first.

When NameNode high-availability is configured, the active and standby NameNodes have a shared storage where edits are stored. Typically, this shared storage is an ensemble of three or more JournalNodes, but that's abstracted away from the checkpointing process.

The standby NameNode maintains a relatively up-to-date version of the namespace by periodically replaying the new edits written to the shared edits directory by the

active NameNode. As a result, checkpointing is as simple as checking if either of the two preconditions are met, saving the namespace to a new fsimage (roughly equivalent to running `hdfs dfsadmin -saveNamespace` on the command line), then transferring the new fsimage to the active namenode via HTTP.



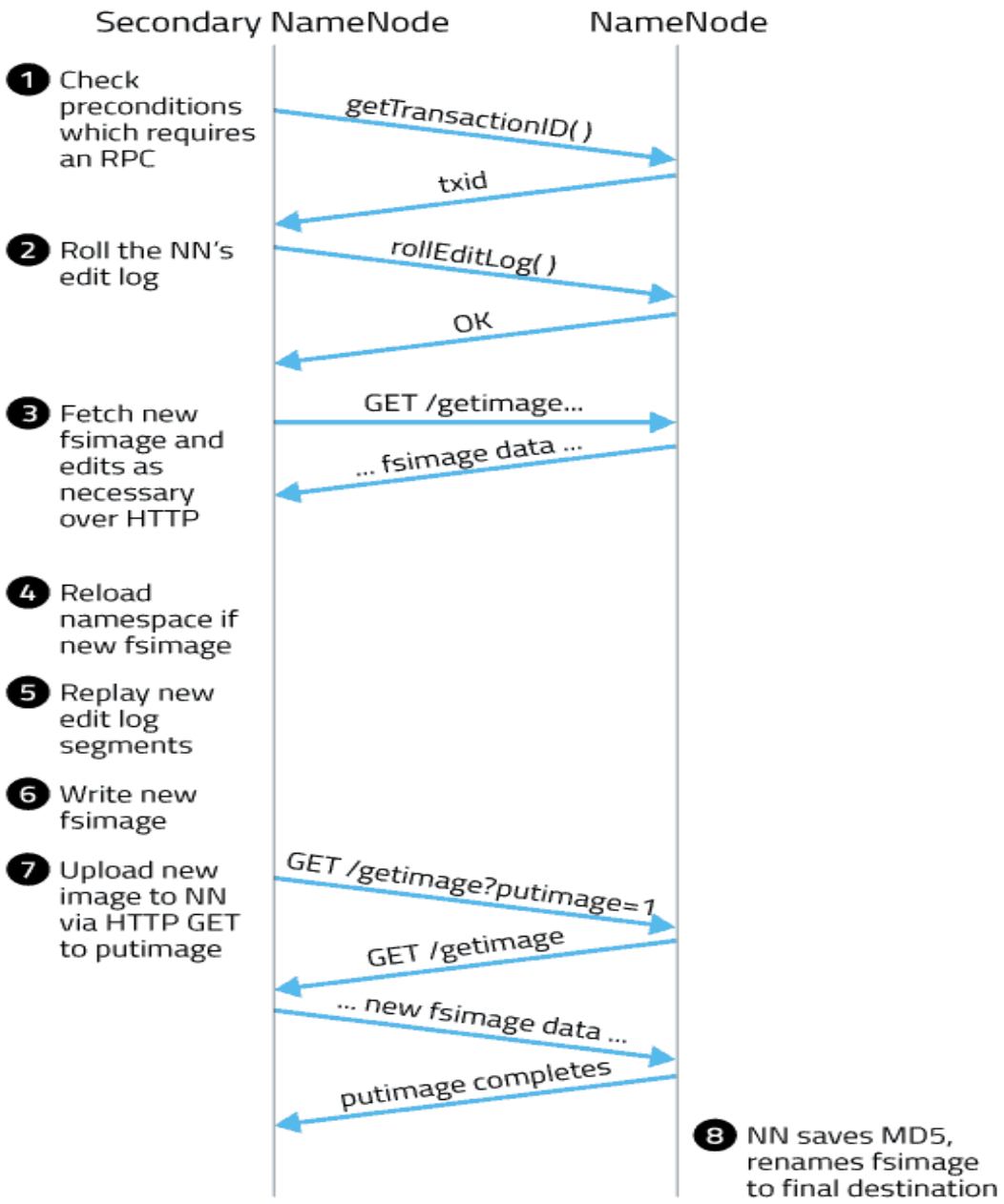
Here, Standby NameNode is abbreviated as SbNN and Active NameNode as ANN:

1. SbNN checks whether either of the two preconditions are met: elapsed time since the last checkpoint or number of accumulated edits.
2. SbNN saves its namespace to an a new fsimage with the intermediate name fsimage.ckpt_, where txid is the transaction ID of the most recent edit log transaction. Then, the SbNN writes an MD5 file for the fsimage, and renames the fsimage to fsimage_. While this is taking place, most other SbNN operations are blocked. This means administrative operations like NameNode failover or accessing parts of the SbNN's webui. Routine HDFS client operations (such as listing, reading, and writing files) are unaffected as these operations are serviced by the ANN.

3. SbNN sends an HTTP GET to the active NN's GetImageServlet at /getimage?putimage=1. The URL parameters also have the transaction ID of the new fsimage and the SbNN's hostname and HTTP port.
4. The active NN's servlet uses the information in the GET request to in turn do its own GET back to the SbNN's GetImageServlet. Similar to the standby, it first saves the new fsimage with the intermediate name fsimage.ckpt_, creates the MD5 file for the fsimage, and then renames the new fsimage to fsimage_.

Check pointing with a Secondary NameNode

In a non-HA deployment, checkpointing is done on the SecondaryNameNode rather than the standby NameNode. Since there isn't a shared edits directory or automatic tailing of the edit log, the SecondaryNameNode has to go through a few more steps first to refresh its view of the namespace before continuing down the same basic steps.



Here, the NameNode is abbreviated as NN and the SecondaryNameNode as 2NN:

1. 2NN checks whether either of the two preconditions are met: elapsed time since the last checkpoint or number of accumulated edits.
    a. In the absence of a shared edit directory, the most recent edit log transaction ID needs to be queried via an explicit RPC to the NameNode (NamenodeProtocol#getTransactionId).
2. 2NN triggers an edit log roll, which ends the current edit log segment and starts a new one. The NN can keep writing edits to the new segment while

the SNN compacts all the previous ones. This also returns the transaction IDs of the current fsimage and the edit log segment that was just rolled. Explicit triggering of an edit log roll is not necessary in an HA configuration, since the standby NameNode periodically rolls the edit log orthogonal to checkpointing.

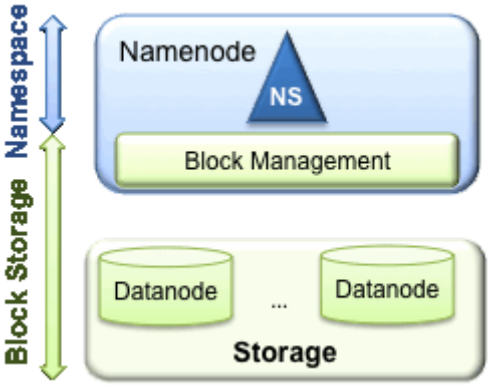
3. Given these two transaction IDs, the 2NN fetches new fsimage and edit files as needed via GET to the NN's GetImageServlet. The 2NN might already have some of these files from a previous checkpoint (such as the current fsimage).
4. If necessary, the 2NN reloads its namespace from a newly downloaded fsimage.
5. The 2NN replays the new edit log segments to catch up to the current transaction ID.From here, the rest is the same as in the HA case with a StandbyNameNode.
6. 2NN writes out its namespace to a new fsimage.
7. The 2NN contacts the NN via HTTP GET at /getimage?putimage=1, causing the NN's servlet to do its own GET to the 2NN to download the new fsimage.

3.4 HDFS federation

Federation enhances an existing Hadoop HDFS architecture. Prior HDFS architecture allows single namespace for the entire cluster. In that architecture, single NameNode manages namespace.

If NameNode fails, then whole cluster will be out of service. And the cluster will be unavailable until the NameNode restarts or brought on a separate machine.

HDFS Federation was introduced to overcome this limitation. It overcomes this by adding support for many NameNode/Namespaces to HDFS.



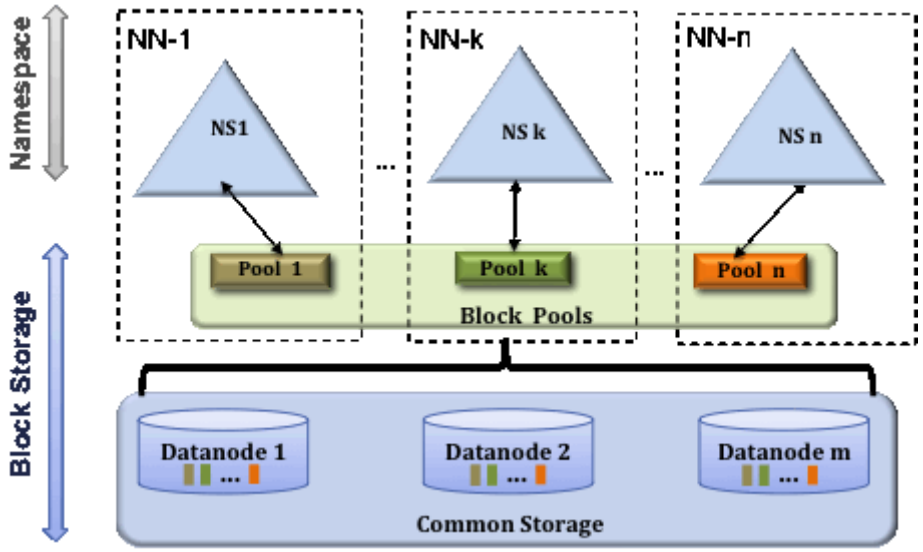
a) Namespace– This layer manages files, directories, and blocks. This layer supports basic file system operation such as creation, deletion of files.
b) Block Storage– It has two parts-
   - Block management – It supports block related operation such as creation, deletion of the blocks. It manages data nodes in the cluster and takes care of replication management.
   - Physical storage – This stores the blocks on the local file system and provides access to read or write operation. Follow this link to learn HDFS data read and write operation.

This current HDFS works fine for smaller setups. But, for large organizations where we need to take care of the huge amount of data has some limitation. Hadoop federation handles those limitations.

Multiple Namenodes/Namespaces

Federation uses many independent Namenode/namespaces to scale the name service horizontally. In HDFS Federation Architecture, at the bottom, datanodes are present. And datanodes are used as a common storage for blocks by all the namenodes.

Each datanodes registers with all the namenodes in the cluster. These datanodes send periodic heartbeats, block, report and handle command from the namenodes.



Block Pool

A Block Pool is a set of blocks that belong to a single namespace. Datanodes store blocks for all the block pools in the cluster. Each Block Pool is managed independently. This allows a namespace to generate Block IDs for new blocks without the need for coordination with the other namespaces. A Namenode failure does not prevent the Datanode from serving other Namenodes in the cluster.

A Namespace and its block pool together are called Namespace Volume. It is a self-contained unit of management. When a Namenode/namespace is deleted, the corresponding block pool at the Datanodes is deleted. Each namespace volume is upgraded as a unit, during cluster upgrade.

ClusterID

A ClusterID identifier is used to identify all the nodes in the cluster. When a Namenode is formatted, this identifier is either provided or auto generated. This ID should be used for formatting the other Namenodes into the cluster.

Benefits of HDFS Federation

HDFS Federation overcomes the limitations of prior HDFS architecture. Hence it provides:

1. Isolation – There is no isolation in single namenode in a multi-user environment. In HDFS federation different categories of application and users can be isolated to different namespaces by using many namenodes.

2. Namespace Scalability – In federation many namenodes horizontally scales up in the filesystem namespace.
3. Performance – We can improve Read/write operation throughput by adding more namenodes.

3.5 **HDFS High availability**

The namenode is still a single point of failure (SPOF), since if it did fail, all clients—including MapReduce jobs—would be unable 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 could be brought online.

To recover from a failed namenode in this situation, an administrator starts a new primary namenode with one of the filesystem metadata replicas, and configures datanodes and clients to use this new namenode.

The new namenode is not able to serve requests until it has

i) loaded its namespace image into memory,
ii) replayed its edit log, and
iii) received enough block reports from the datanodes to leave safe mode. On large clusters with many files and blocks, the time it takes for a namenode to start from cold can be 30 minutes or more.

The 0.23 release series of Hadoop remedies this situation by adding support for HDFS highavailability (HA). In this implementation there is a pair of namenodes in an activestandby configuration. In the event of the failure of the active namenode, the standby takes over its duties to continue servicing client requests without a significant interruption.

A few architectural changes are needed to allow this to happen:

- The namenodes must use highly-available shared storage to share the edit log.
- Datanodes must send block reports to both namenodes since the block mappings are stored in a namenode's memory, and not on disk.
- Clients must be configured to handle namenode failover, which uses a mechanism that is transparent to users.

Failover and fencing:

The transition from the active namenode to the standby is managed by a new entity in the system called the failover controller. Failover controllers are pluggable, but the first implementation uses ZooKeeper to ensure that only one namenode is active.

Failover may also be initiated manually by an adminstrator, in the case of routine maintenance, for example. This is known as a graceful failover, since the failover controller arranges an orderly transition for both namenodes to switch roles. In the case of an ungraceful failover, The HA implementation goes to great lengths to ensure that the previously active namenode is prevented from doing any damage and causing corruption—a method known as fencing.

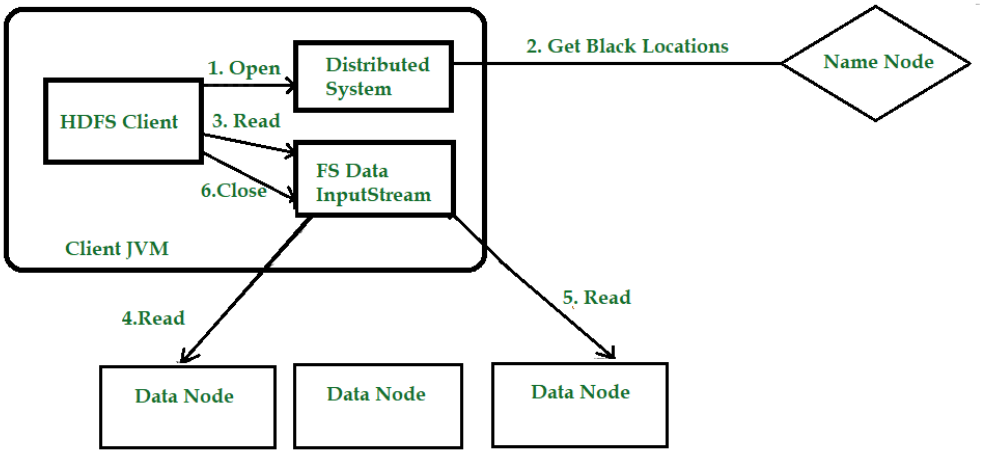
3.6 Architectural description for Hadoop Cluster

3.7 When to use or not to use HDFS, Block Allocation in Hadoop Cluster

3.8 HDFS Operations

Read operation in HDFS

Let's get an idea of how data flows between the client interacting with HDFS, the name node, and the data nodes with the help of a diagram. Consider the figure:



Step 1: The client opens the file it wishes to read by calling open() on the File System Object(which for HDFS is an instance of Distributed File System).

Step 2: Distributed File System(DFS) calls the name node, using remote procedure calls (RPCs), to determine the locations of the first few blocks in the file. For each block, the name node returns the addresses of the data nodes that have a copy of that block. The DFS returns an FSDataInputStream to the client for it to read data from. FSDataInputStream in turn wraps a DFSInputStream, which manages the data node and name node I/O.

Step 3: The client then calls read() on the stream. DFSInputStream, which has stored the info node addresses for the primary few blocks within the file, then connects to the primary (closest) data node for the primary block in the file.

Step 4: Data is streamed from the data node back to the client, which calls read() repeatedly on the stream.

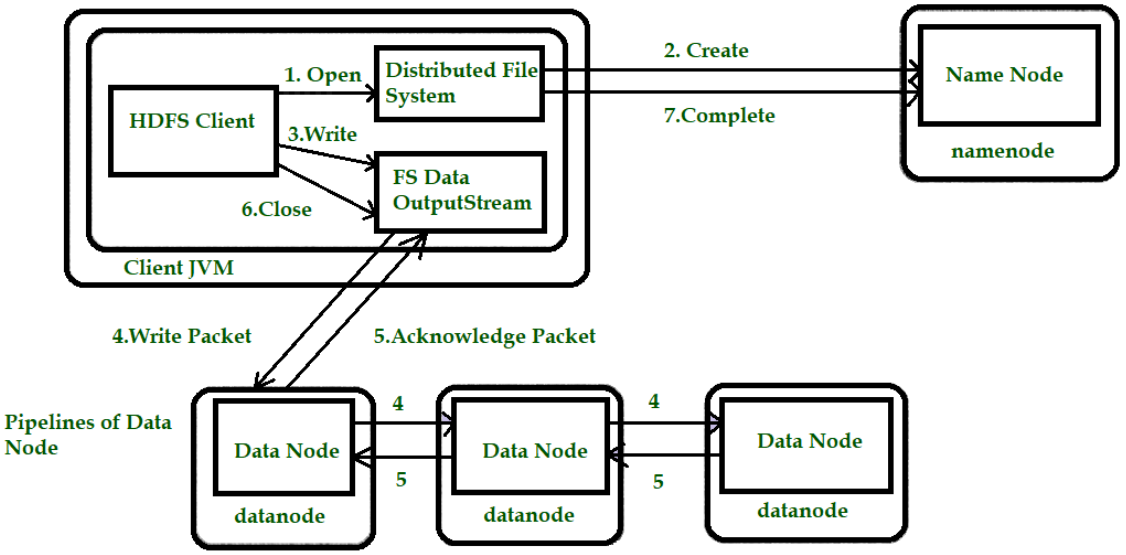
Step 5: When the end of the block is reached, DFSInputStream will close the connection to the data node, then finds the best data node for the next block. This happens transparently to the client, which from its point of view is simply reading an endless stream. Blocks are read as, with the DFSInputStream opening new connections to data nodes because the client reads through the stream. It will also call the name node to retrieve the data node locations for the next batch of blocks as needed.

Step 6: When the client has finished reading the file, a function is called, close() on the FSDataInputStream.

Write operation in HDFS

Next, we'll check out how files are written to HDFS. Consider figure 1.2 to get a better understanding of the concept.

Note: HDFS follows the Write once Read many times model. In HDFS we cannot edit the files which are already stored in HDFS, but we can append data by reopening the files.



Step 1: The client creates the file by calling create() on DistributedFileSystem(DFS).

Step 2: DFS makes an RPC call to the name node to create a new file in the file system's namespace, with no blocks associated with it. The name node performs various checks to make sure the file doesn't already exist and that the client has the right permissions to create the file. If these checks pass, the name node prepares a record of the new file; otherwise, the file can't be created and therefore the client is thrown an error i.e. IOException. The DFS returns an FSDataOutputStream for the client to start out writing data to.

Step 3: Because the client writes data, the DFSOutputStream splits it into packets, which it writes to an indoor queue called the info queue. The data queue is consumed by the DataStreamer, which is liable for asking the name node to allocate new blocks by picking an inventory of suitable data nodes to store the replicas. The list of data nodes forms a pipeline, and here we'll assume the replication level is three, so there are three nodes in the pipeline. The DataStreamer streams the packets to the primary data node within the pipeline, which stores each packet and forwards it to the second data node within the pipeline.

Step 4: Similarly, the second data node stores the packet and forwards it to the third (and last) data node in the pipeline.

Step 5: The DFSOutputStream sustains an internal queue of packets that are waiting to be acknowledged by data nodes, called an "ack queue".

Step 6: This action sends up all the remaining packets to the data node pipeline and waits for acknowledgments before connecting to the name node to signal whether the file is complete or not.

HDFS follows Write Once Read Many models. So, we can't edit files that are already stored in HDFS, but we can include them by again reopening the file. This design

allows HDFS to scale to a large number of concurrent clients because the data traffic is spread across all the data nodes in the cluster. Thus, it increases the availability, scalability, and throughput of the system.

Hadoop Archives

Hadoop archive is a facility which packs up small files into one compact HDFSblock to avoid memory wastage of name node.name node stores the metadata information of the the HDFS data.SO,say 1GB file is broken in 1000 pieces then namenode will have to store metadata about all those 1000 small files.In that manner,namenode memory willbe wasted it storing and managing a lot of data.

HAR is created from a collection of files and the archiving tool will run a MapReduce job.these Maps reduce jobs to process the input files in parallel to create an archive file.

Limitations of HAR Files:

1) Creation of HAR files will create a copy of the original files. So, we need as much disk space as size of original files which we are archiving.We can delete the original files after creation of archive to release some disk space.
2) Once an archive is created, to add or remove files from/to archive we need to re-create the archive.
3) HAR file will require lots of map tasks which are inefficient.

Data Integrity in HDFS

1) Data Integrity means to make sure that no data is lost or corrupted during storage or processing of the Data.
2) Since in Hadoop, amount of data being written or read is large in Volume, a chance of data corruption is more.
3) So in Hadoop checksum is computed when data written to the disk for the first time and again checked while reading data from the disk. If checksum matches the original checksum then it is said that data is not corrupted otherwise it is said to be corrupted.
4) Its just data detection error.
5) It is possible that it's the checksum that is corrupt, not the data, but this is very unlikely, because the checksum is much smaller than the data
6) HDFS uses a more efficient variant called CRC-32C to calculate checksum.

7) DataNodes are responsible for verifying the data they receive before storing the data and its checksum. Checksum is computed for the data that they receive from clients and from other DataNodes during replication
8) Hadoop can heal the corrupted data by copying one of the good replica to produce the new replica which is uncorrupt replica.
9) If a client detects an error when reading a block, it reports the bad block and the DataNodes it was trying to read from to the NameNode before throwing a Checksum Exception.
10) The NameNode marks the block replica as corrupt so it doesn't direct any more clients to it or try to copy this replica to another DataNodes.
11) It provides copy of the block another DataNodes which is to be replicated, so its replication factor is back at the expected level.
12) Once this has happened, the corrupt replica is deleted

Processing Unit:

What is MapReduce, History of MapReduce, How does MapReduce works, Input files, Input Format types Output Format Types, Text Input Format, Key Value Input Format, Sequence File Input Format, Input split, Record Reader, MapReduce overview, Mapper Phase, Reducer Phase, Sort and Shuffle Phase, Importance of MapReduce, Data Flow, Counters, Combiner Function, Partition Function, Joins, Map Side Join, Reduce Side Join, MapReduce Web UI, Job Scheduling, Task Scheduling, Fault Tolerance, Writing MapReduce Application, Driver Class, Mapper Class, Reducer Class, Serialization, File Based Data Structure, Writing a simple MapReduce program to Count Number of words, MapReduce Work Flows.