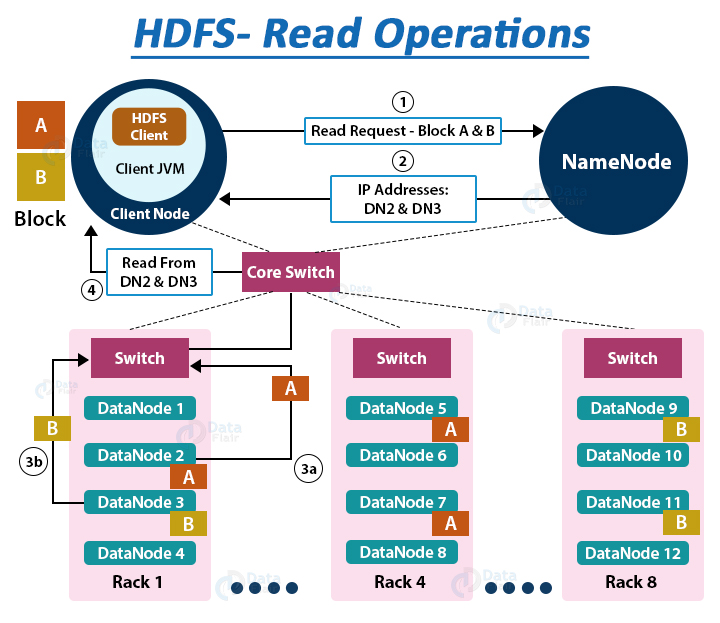
HDFS read operation

Suppose the HDFS client wants to read a file “File.txt”. Let the file be divided into two blocks say, A and B. The following steps will take place during the file read:

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2016/05/HDFS-read-operations.jpg)

1. The Client interacts with HDFS NameNode

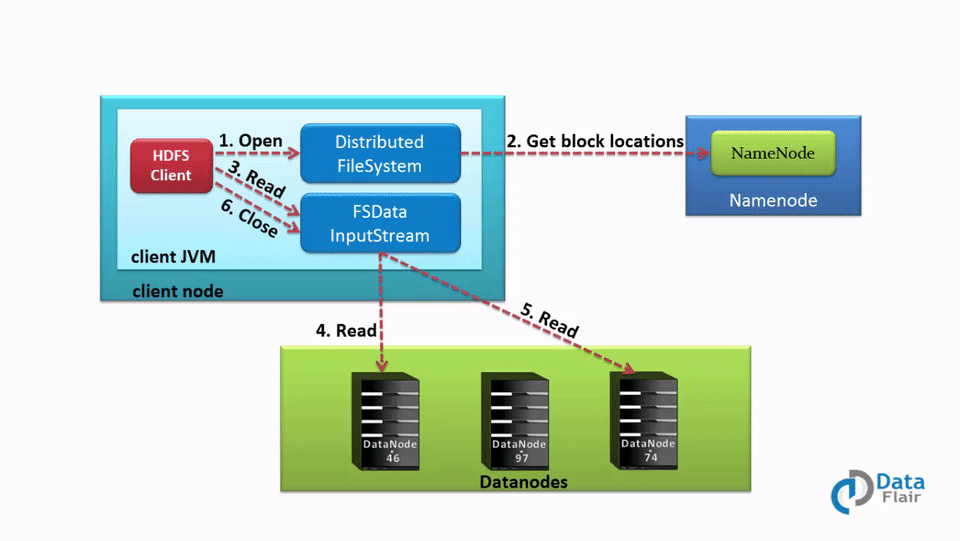
* As the NameNode stores the block’s metadata for the file “File.txt’, the client will reach out to NameNode asking locations of DataNodes containing data blocks.
* The NameNode first checks for required privileges, and if the client has sufficient privileges, the NameNode sends the locations of DataNodes containing blocks (A and B). NameNode also gives a **security token** to the client, which they need to show to the DataNodes for authentication. Let the NameNode provide the following list of IPs for block A and B – for block A, location of DataNodes D2, D5, D7, and for block B, location of DataNodes D3, D9, D11.

To perform various **HDFS operations** (read, write, copy, move, change permission, etc.) follow [**HDFS command list**](https://data-flair.training/blogs/top-hadoop-hdfs-commands-tutorial/).

2. The client interacts with HDFS DataNode

* After receiving the addresses of the DataNodes, the client directly interacts with the DataNodes. The client will send a request to the closest DataNodes (D2 for block A and D3 for block B) through the **FSDataInputstream** object. The **DFSInputstream** manages the interaction between client and DataNode.
* The client will show the security tokens provided by NameNode to the DataNodes and start reading data from the DataNode. The data will flow directly from the DataNode to the client.
* After reading all the required file blocks, the client calls close() method on the [FSDataInputStream](https://hadoop.apache.org/docs/r3.0.2/api/org/apache/hadoop/fs/FSDataInputStream.html) object.

Now let us see how internally read operation is carried out in Hadoop HDFS, how data flows between the client, the NameNode, and DataNodes during file read.

Internals of file read in HDFS[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2016/05/Data-Read-Mechanism-in-HDFS.gif)

1. In order to open the required file, the client calls the **open()** method on the **FileSystem object**, which for HDFS is an instance of DistributedFilesystem.
2. DistributedFileSystem then calls the NameNode using RPC to get the locations of the first few blocks of a file. For each [**data block**](https://data-flair.training/blogs/data-block/), NameNode returns the addresses of Datanodes that contain a copy of that block. Furthermore, the DataNodes are sorted based on their proximity to the client.
3. The DistributedFileSystem returns an **FSDataInputStream** to the client from where the client can read the data. FSDataInputStream in succession wraps a DFSInputStream. **DFSInputStream** manages the I/O of DataNode and NameNode.
4. Then the client calls the **read()** method on the**FSDataInputStream** object.
5. The DFSInputStream, which contains the addresses for the first few blocks in the file, connects to the closest DataNode to read the first block in the file. Then, the data flows from DataNode to the client, which calls read() repeatedly on the FSDataInputStream.
6. Upon reaching the end of the file, DFSInputStream closes the connection with that DataNode and finds the best suited DataNode for the next block.
7. If the DFSInputStream during reading, faces an error while communicating with a DataNode, it will try the other closest DataNode for that block. DFSInputStream will also remember DataNodes that have failed so that it doesn’t needlessly retry them for later blocks. Also, the DFSInputStream verifies checksums for the data transferred to it from the DataNode. If it finds any corrupt block, it reports this to the NameNode and reads a copy of the block from another DataNode.
8. When the client has finished reading the data, it calls **close()** on the **FSDataInputStream**.

How to Read a file from HDFS – Java Program

A sample code to read a file from HDFS is as follows (To perform HDFS read and write operations:

FileSystem fileSystem = FileSystem.get(conf);

Path path = **new** Path("/path/to/file.ext");

**if** (!fileSystem.exists(path)) {

System.out.println("File does not exists");

**return**;

}

FSDataInputStream **in** = fileSystem.open(path);

int numBytes = 0;

**while** ((numBytes = **in**.read(b))> 0) {

System.out.prinln((char)numBytes));// code to manipulate the data which is read

}

**in**.close();

out.close();

fileSystem.close();

Summary

So in this article, we have studied the data flow between client, DataNode, and NameNode during a client read request.

Now you have a pretty good idea about the HDFS file read operation and how the client interacts with DataNode and NameNode.

HDFS write operation

To write data in HDFS, the client first interacts with the **NameNode** to get permission to write data and to get IPs of **DataNodes** where the client writes the data. The client then directly interacts with the DataNodes for writing data. The DataNode then creates a replica of the data block to other DataNodes in the pipeline based on the replication factor.

**DFSOutputStream** in HDFS maintains two queues (data queue and ack queue) during the write operation.

1. The client interacts with HDFS NameNode

* To write a file inside the HDFS, the client first interacts with the NameNode. NameNode first checks for the client privileges to write a file. If the client has sufficient privilege and there is no file existing with the same name, NameNode then creates a record of a new file.
* NameNode then provides the address of all DataNodes, where the client can write its data. It also provides a security token to the client, which they need to present to the DataNodes before writing the block.
* If the file already exists in the HDFS, then file creation fails, and the client receives an **IO Exception**.

2. The client interacts with HDFS DataNode

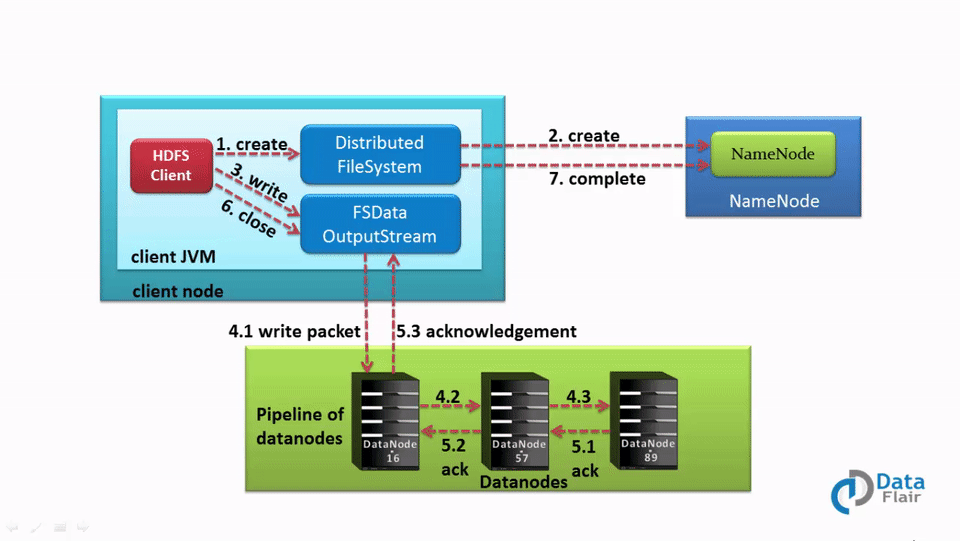
After receiving the list of the DataNodes and file write permission, the client starts writing data directly to the first DataNode in the list. As the client finishes writing data to the first DataNode, the DataNode starts making replicas of a block to other DataNodes depending on the replication factor.

If the replication factor is 3, then there will be a minimum of 3 copies of blocks created in different DataNodes, and after creating required replicas, it sends an acknowledgment to the client.

Thus it leads to the creation of a pipeline, and data replication to the desired value, in the cluster.

Internals of file write in Hadoop HDFS

Let us understand the HDFS write operation in detail. The following steps will take place while writing a file to the HDFS:

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2016/05/Data-Write-Mechanism-in-HDFS.gif)

**1.** The client calls the **create()** method on **DistributedFileSystem** to create a file.

**2.** DistributedFileSystem interacts with NameNode through the RPC call to create a new file in the filesystem namespace with no blocks associated with it.

**3.** The NameNode checks for the client privileges and makes sure that the file doesn’t already exist. If the client has sufficient privileges and no file with the same name exists, the NameNode makes a record of the new file. Otherwise, the client receives an I/O exception, and file creation fails. The DistributedFileSystem then returns an FSDataOutputStream for the client where the client starts writing data. [FSDataOutputstream](https://hadoop.apache.org/docs/r2.7.5/api/org/apache/hadoop/fs/FSDataOutputStream.html" \t "_blank), in turn, wraps a DFSOutputStream, which handles communication with the DataNodes and NameNode.

**4.** As the client starts writing data, the **DFSOutputStream** splits the client’s data into packets and writes it to an internal queue called the **data** **queue**. **DataStreamer**, which is responsible for telling the NameNode to allocate new blocks by choosing the list of suitable DataNode to store the replicas, uses this data queue.

The list of DataNode forms a pipeline. The number of DataNodes in the pipeline depends on the replication factor.

Suppose the replication factor is 3, so there are three nodes in the pipeline.

The DataStreamer streams the packet to the first DataNode in the pipeline, which stores each packet and forwards it to the second node in the pipeline. Similarly, the second DataNode stores the packet and transfers it to the next node in the pipeline (last node).

**5.** The **DFSOutputStream** also maintains another queue of packets, called **ack queue,** which is waiting for the acknowledgment from DataNodes.

Packet in the ack queue gets remove only when it receives an acknowledgment from all the DataNodes in the pipeline.

**6.** The client calls the **close()** method on the stream when he/she finishes writing data. Thus, before communicating the NameNode to signal about the file complete, the client close() method’s action pushes the remaining packets to the DataNode pipeline and waits for the acknowledgment.

**7.** As the Namenode already knows about the blocks (the file made of), so the NameNode only waits for blocks to be minimally replicated before returning successfully.

What happens if DataNode fails while writing a file in the HDFS?

While writing data to the DataNode, if DataNode fails, then the following actions take place, which is transparent to the client writing the data.

**1.** The pipeline gets closed, packets in the ack queue are then added to the front of the data queue making DataNodes downstream from the failed node to not miss any packet.

**2.** Then the current block on the alive DataNode gets a new identity. This id is then communicated to the NameNode so that, later on, if the failed DataNode recovers, the partial block on the failed DataNode will be deleted.

**3.** The failed DataNode gets removed from the pipeline, and a new pipeline gets constructed from the two alive DataNodes. The remaining of the block’s data is then written to the alive DataNodes, added in the pipeline.

**4.** The NameNode observes that the block is **under-replicated**, and it arranges for creating further copy on another DataNode. Other coming blocks are then treated as normal.

How to Write a file in HDFS – Java Program

A sample code to write a file to HDFS in Java is as follows:

FileSystem fileSystem = FileSystem.get(conf);

// Check if the file already exists

Path path = **new** Path("/path/to/file.ext");

**if** (fileSystem.exists(path)) {

System.out.println("File " + dest + " already exists");

**return**;

}

// Create a new file and write data to it.

FSDataOutputStream out = fileSystem.create(path);

InputStream **in** = **new** BufferedInputStream(**new** FileInputStream(

**new** File(source)));

byte[] b = **new** byte[1024];

int numBytes = 0;

**while** ((numBytes = **in**.read(b)) > 0) {

out.write(b, 0, numBytes);

}

// Close all the file descripters

**in**.close();

out.close();

fileSystem.close();

Summary

After reading this article, you have a good idea about the HDFS file write operation. From this article, we clearly understand the anatomy of file write in Hadoop.

The article has described the file write in detail along with the explanation of replicas creation during file write. We have also seen what happens if the DataNode fails while writing the file.