HDFS Tutorial: Read & Write Commands using Java API

By **Execute Java Online**, www.guru99.com
January 14th, 2017

Hadoop comes with a distributed file system called **HDFS** (**HADOOP Distributed File Systems**) HADOOP based applications make use of HDFS. HDFS is designed for storing very large data files, running on clusters of commodity hardware. It is fault tolerant, scalable, and extremely simple to expand.

Do you know? When data exceeds the capacity of storage on a single physical machine, it becomes essential to divide it across number of separate machines. File system that manages storage specific operations across a network of machines is called as **distributed file system**.

In this tutorial we will learn,

HDFS cluster primarily consists of a **NameNode** that manages the file system **Metadata** and a **DataNodes** that stores the **actual data**.

- **NameNode:** NameNode can be considered as a master of the system. It maintains the file system tree and the metadata for all the files and directories present in the system. Two files **'Namespace image'** and the **'edit log'** are used to store metadata information. Namenode has knowledge of all the datanodes containing data blocks for a given file, however, it does not store block locations persistently. This information is reconstructed every time from datanodes when the system starts.
- **DataNode :** DataNodes are slaves which reside on each machine in a cluster and provide the actual storage. It is responsible for serving, read and write requests for the clients.

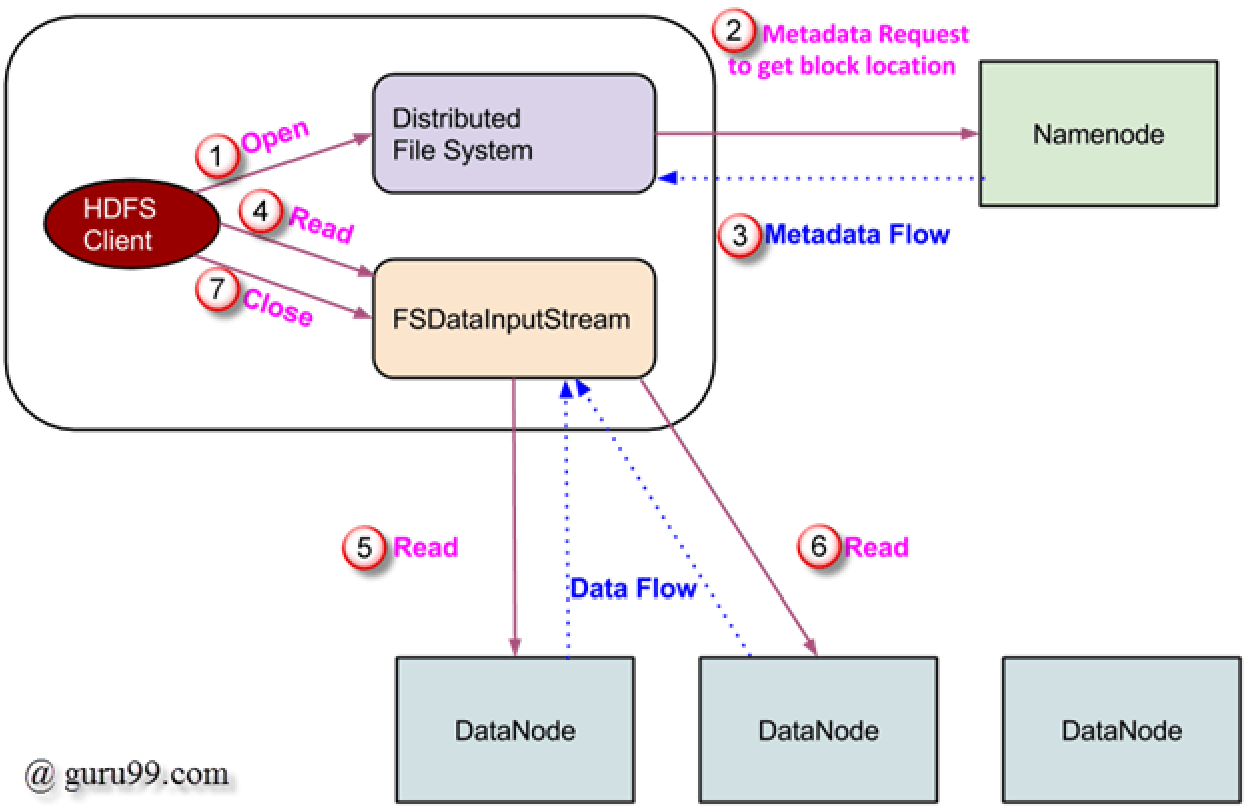
Read/write operations in HDFS operate at a block level. Data files in HDFS are broken into block-sized chunks, which are stored as independent units. Default block-size is 64 MB.

HDFS operates on a concept of data replication wherein multiple replicas of data blocks are created and are distributed on nodes throughout a cluster to enable high availability of data in the event of node failure.

Do you know? A file in HDFS, which is smaller than a single block, does not occupy a block's full storage.

Read Operation In HDFS

Data read request is served by HDFS, NameNode and DataNode. Let's call reader as a 'client'. Below diagram depicts file read operation in Hadoop.

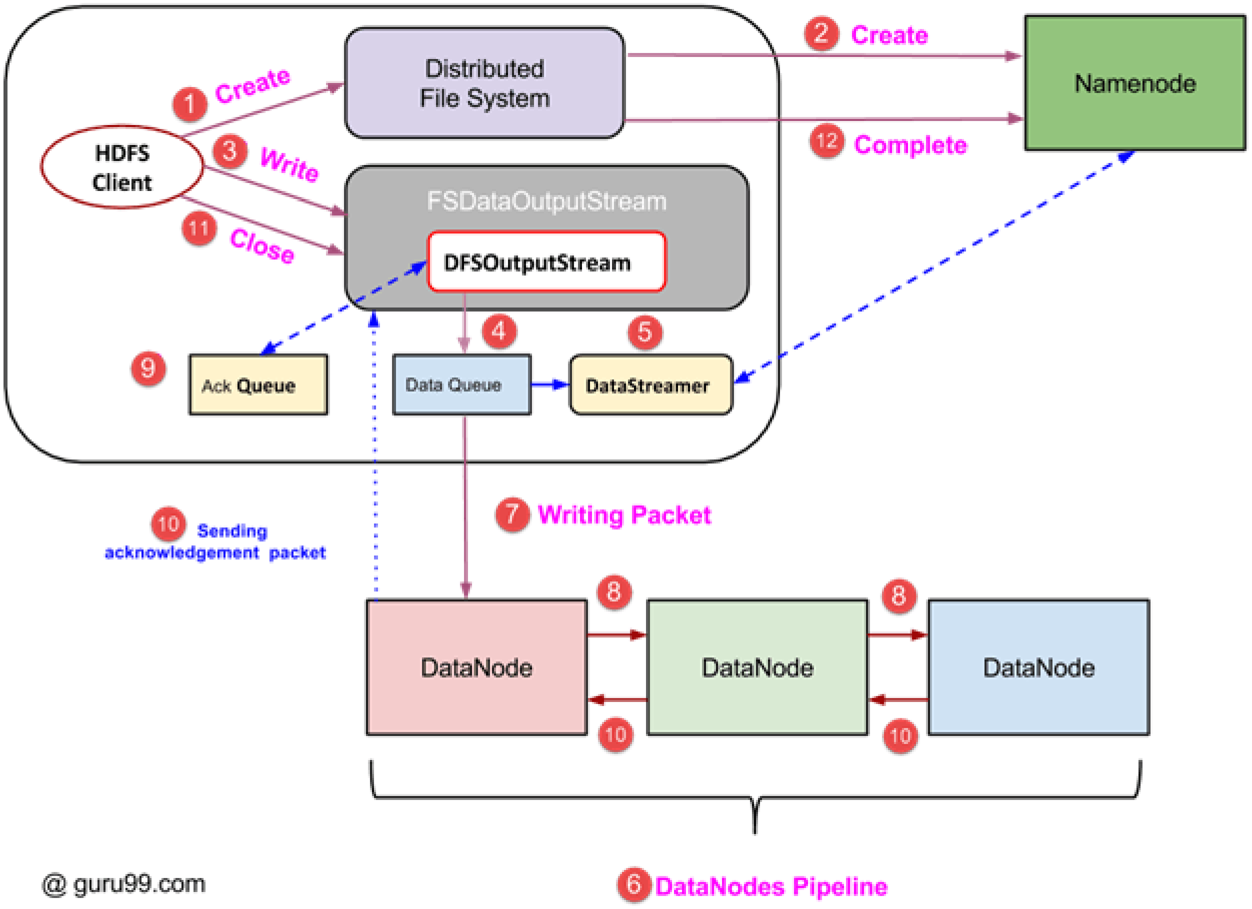


1. Client initiates read request by calling **'open()'** method of FileSystem object; it is an object of type **DistributedFileSystem**.
2. This object connects to namenode using RPC and gets metadata information such as the locations of the blocks of the file. Please note that these addresses are of first few block of file.
3. In response to this metadata request, addresses of the DataNodes having copy of that block, is returned back.
4.

5. Data is read in the form of streams wherein client invokes **'read()'** method repeatedly. This process of **read()** operation continues till it reaches end of block.
6. Once end of block is reached, DFSInputStream closes the connection and moves on to locate the next DataNode for the next block
7. Once client has done with the reading, it calls **close()** method.

Write Operation In HDFS

In this section, we will understand how data is written into HDFS through files.



1. Client initiates write operation by calling 'create()' method of DistributedFileSystem object which creates a new file - Step no. 1 in above diagram.
2. DistributedFileSystem object connects to the NameNode using RPC call and initiates new file creation. However, this file create operation does not associate any blocks with the file. It is the responsibility of NameNode to verify that the file (which is being created) does not exist already and client has correct permissions to create new file. If file already exists or client does not have sufficient permission to create a new file, then **IOException** is thrown to client. Otherwise, operation succeeds and a new record for the file is created by the NameNode.

3. Once new record in NameNode is created, an object of type FSDataOutputStream is returned to the client. Client uses it to write data into the HDFS. Data write method is invoked (step 3 in diagram).
4. FSDataOutputStream contains DFSOutputStream object which looks after communication with DataNodes and NameNode. While client continues writing data, **DFSOutputStream** continues creating packets with this data. These packets are en-queued into a queue which is called as **DataQueue**.
5. There is one more component called **DataStreamer** which consumes this **DataQueue**. DataStreamer also asks NameNode for allocation of new blocks thereby picking desirable DataNodes to be used for replication.
6. Now, the process of replication starts by creating a pipeline using DataNodes. In our case, we have chosen replication level of 3 and hence there are 3 DataNodes in the pipeline.
7. The DataStreamer pours packets into the first DataNode in the pipeline.
8. Every DataNode in a pipeline stores packet received by it and forwards the same to the second DataNode in pipeline.
9. Another queue, 'Ack Queue' is maintained by DFSOutputStream to store packets which are waiting for acknowledgement from DataNodes.
10. Once acknowledgement for a packet in queue is received from all DataNodes in the pipeline, it is removed from the 'Ack Queue'. In the event of any DataNode failure, packets from this queue are used to reinitiate the operation.
11. After client is done with the writing data, it calls close() method (Step 9 in the diagram) Call to close(), results into flushing remaining data packets to the pipeline followed by waiting for acknowledgement.
12. Once final acknowledgement is received, NameNode is contacted to tell it that the file write operation is complete.

Access HDFS using JAVA API

In this section, we try to understand Java interface used for accessing Hadoop's file system.

In order to interact with Hadoop's filesytem programmatically, Hadoop provides multiple JAVA classes. Package named org.apache.hadoop.fs contains classes useful in manipulation of a file in Hadoop's filesystem. These operations include, open, read, write, and close. Actually, file API for Hadoop is generic and can be extended to interact with other filesystems other than HDFS.

Reading a file from HDFS, programmatically

Object java.net.URL is used for reading contents of a file. To begin with, we need to make Java recognize Hadoop's hdfs URL scheme. This is done by calling **setURLStreamHandlerFactory** method on URL object and an instance of FsUrlStreamHandlerFactory is passed to it. This method needs to be executed only once per JVM, hence it is enclosed in a static block.

An example code is-

```
public class URLCat {
    static {
        URL.setURLStreamHandlerFactory(new
FsUrlStreamHandlerFactory());
    }
    public static void main(String[] args) throws Exception {
        InputStream in = null;
        try {
            in = new URL(args[0]).openStream();
            IOUtils.copyBytes(in, System.out, 4096, false);
        } finally {
            IOUtils.closeStream(in);
        }
    }
}
```

This code opens and reads contents of a file. Path of this file on HDFS is passed to the program as a commandline argument.

Access HDFS Using COMMAND-LINE INTERFACE

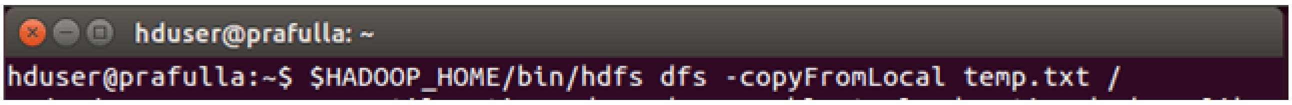
This is one of the simplest way to interact with HDFS. Command-line interface has support for filesystem operations like read file, create directories, moving files, deleting data, and listing directories.

We can run **'$HADOOP_HOME/bin/hdfs dfs -help'** to get detailed help on every command. Here, **'dfs'** is a shell command of HDFS which supports multiple subcommands.

Some of the widely used commands are listed below along with some details of each one.

1. Copy a file from local filesystem to HDFS

$HADOOP_HOME/bin/hdfs dfs -copyFromLocal temp.txt /

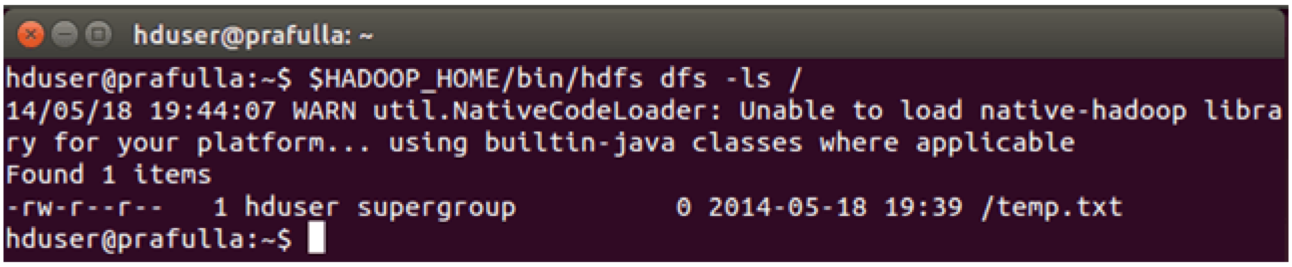


```
hduser@prafulla: ~
hduser@prafulla:~$ $HADOOP_HOME/bin/hdfs dfs -copyFromLocal temp.txt /
```

This command copies file temp.txt from local filesystem to HDFS.

2. We can list files present in a directory using **-ls**

$HADOOP_HOME/bin/hdfs dfs -ls /

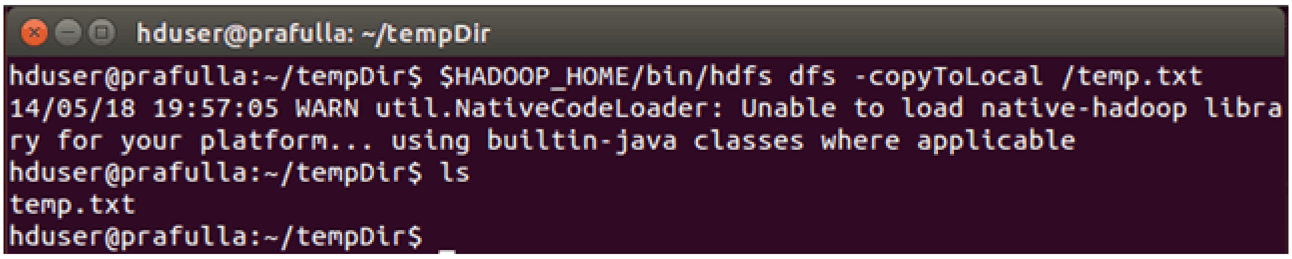


```
hduser@prafulla: ~
hduser@prafulla:~$ $HADOOP_HOME/bin/hdfs dfs -ls /
14/05/18 19:44:07 WARN util.NativeCodeLoader: Unable to load native-hadoop libra
ry for your platform... using builtin-java classes where applicable
Found 1 items
-rw-r--r--   1 hduser supergroup          0 2014-05-18 19:39 /temp.txt
hduser@prafulla:~$ 
```

We can see a file **'temp.txt'** (copied earlier) being listed under **' / '** directory.

3. Command to copy a file to local filesystem from HDFS

$HADOOP_HOME/bin/hdfs dfs -copyToLocal /temp.txt

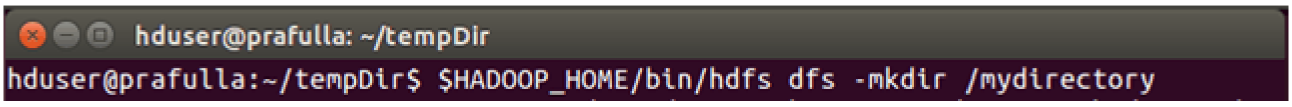


```
hduser@prafulla: ~/tempDir
hduser@prafulla:~/tempDir$ $HADOOP_HOME/bin/hdfs dfs -copyToLocal /temp.txt
14/05/18 19:57:05 WARN util.NativeCodeLoader: Unable to load native-hadoop libra
ry for your platform... using builtin-java classes where applicable
hduser@prafulla:~/tempDir$ ls
temp.txt
hduser@prafulla:~/tempDir$ _
```

We can see **temp.txt** copied to local filesystem.

4. Command to create new directory

$HADOOP_HOME/bin/hdfs dfs -mkdir /mydirectory



```
hduser@prafulla: ~/tempDir
hduser@prafulla:~/tempDir$ $HADOOP_HOME/bin/hdfs dfs -mkdir /mydirectory
```

Check whether directory is created or not. Now, you should know how to do it ;-)