HDFS

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.

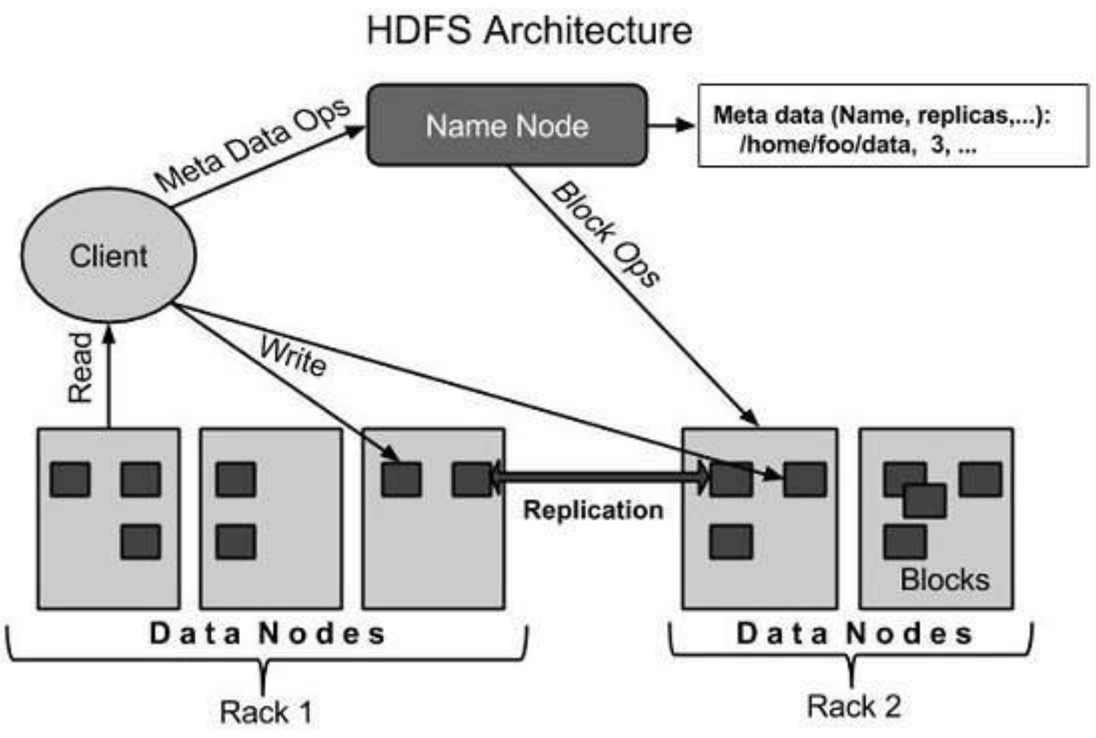
HDFS holds very large amount of data and provides easier access. To store such huge data, the files are stored across multiple machines. These files are stored in redundant fashion to rescue the system from possible data losses in case of failure. HDFS also makes applications available to parallel processing.

Introduction

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

Given below is the architecture of a Hadoop File System.



HDFS Architecture

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.

Installation and Shell

Starting HDFS

Initially you have to format the configured HDFS file system, open namenode HDFS server, and execute the following command.

```
$ hadoop namenode -format
```

After formatting the HDFS, start the distributed file system. The following command will start the namenode as well as the data nodes as cluster.

```
$ start-dfs.sh
```

Listing Files in HDFS

After loading the information in the server, we can find the list of files in a directory, status of a file, using 'ls'. Given below is the syntax of ls that you can pass to a directory or a filename as an argument.

```
$ $HADOOP_HOME/bin/hadoop fs -ls <args>
```

Inserting Data into HDFS

Assume we have data in the file called file.txt in the local system which is ought to be saved in the hdfs file system. Follow the steps given below to insert the required file in the Hadoop file system.

Step 1

You have to create an input directory.

```
$ $HADOOP_HOME/bin/hadoop fs -mkdir /user/input
```

Step 2

Transfer and store a data file from local systems to the Hadoop file system using the put command.

```
$ $HADOOP_HOME/bin/hadoop fs -put /home/file.txt /user/input
```

Step 3

You can verify the file using ls command.

```
$ $HADOOP_HOME/bin/hadoop fs -ls /user/input
```

Retrieving Data from HDFS

Assume we have a file in HDFS called outfile. Given below is a simple demonstration for retrieving the required file from the Hadoop file system.

Step 1

Initially, view the data from HDFS using cat command.

```
$ $HADOOP_HOME/bin/hadoop fs -cat /user/output/outfile
```

Step 2

Get the file from HDFS to the local file system using get command.

```
$ $HADOOP_HOME/bin/hadoop fs -get /user/output/ /home/hadoop_tp/
```

Shutting Down the HDFS

You can shut down the HDFS by using the following command.

```
$ stop-dfs.sh
```

JAVA API

Hadoop's org.apache.hadoop.fs.FileSystem is generic class to access and manage HDFS files/directories located in distributed environment. File's content stored inside datanode with multiple equal large sizes of blocks (e.g. 64 MB), and namenode keep the information of those blocks and Meta information. FileSystem read and stream by accessing blocks in sequence order. FileSystem first get blocks information from NameNode then open, read and close one by one. It opens first blocks once it complete then close and open next block. HDFS replicate the block to give higher reliability and scalability and if client is one of the datanode then it tries to access block locally if fail then move to other cluster datanode.

FileSystem uses FSDataOutputStream and FSDataInputStream to write and read the contents in stream. Hadoop has provided various implementation of FileSystem as described below:

- DistributedFileSystem: To access HDFS File in distributed environment
- LocalFileSystem: To access HDFS file in Local system
- FTPFileSystem: To access HDFS file FTP client
- WebHdfsFileSystem: To access HDFS file over the web

URI and Path:

Hadoop's URI locate file location in HDFS. It uses hdfs://host: port/location to access file through FileSystem.

Below code show how to create URI

```
hdfs://localhost:9000/user/joe/TestFile.txt
URI uri=URI.create ("hdfs://host: port/path");
```

Host and post on above uri could be configured in conf/core-site.xml file as below

```
<property><name>fs.default.name</name><value>hdfs://localhost:9000</value></
property>
```

Path consist URI and resolve the OS dependency in URI e.g. Windows uses \\path whereas linux uses //. It also uses to resolve parent child dependency.

It could be created as below

```
Path path=new Path (uri); //It constitute URI
```

Configuration

Configuration class passes the Hadoop configuration information to FileSystem. It loads the core-site and core-default.xml through class loader and keeps Hadoop configuration information such as fs.defaultFS, fs.default.name etc. You can create the Configuration class as below

```
Configuration conf = new Configuration ();
```

You can also set the configuration parameter explicitly as below

```
conf.set("fs.default.name", "hdfs://localhost:9000");
```

FileSystem

Below code describe how to create Hadoop's FileSystem

```
public static FileSystem get(Configuration conf)
public static FileSystem get(URI uri, Configuration conf)
public static FileSystem get(URI uri, Configuration conf, String user)
```

FileSystem uses NameNode to locate the DataNode and then directly access DataNodes block in sequence order to read the file. FileSystem uses Java IO FileSystem interface mainly DataInputStream and DataOutputStream for IO operation.

If you are looking to get local filesystem we can directly use getLocal method as mentioned below

```
public static LocalFileSystem getLocal(Configuration conf)
```

FSDataInputStream

FSDataInputStream wraps the DataInputStream and implements Seekable, PositionedReadable interfaces which provide method like getPos(), seek() method to provide Random Access on HDFS file.

FileSystem have open() method which return FSDataInputStream as below:

```
URI uri = URI.create ("hdfs://host: port/file path");
Configuration conf = new Configuration ();
FileSystem file = FileSystem.get (uri, conf);
FSDataInputStream in = file.open(new Path(uri));
```

Above method get FSDataInputStream with default buffer size 4096 byte i.e. 4KB. We can also define the buffer size while creating Input Stream as below code.

```
public abstract FSDataInputStream open(Path path, int sizeBuffer)
```

FSDaraInputStream implements seek (long pos) and getPos () method of Seekable interface.

```
public interface Seekable {
  void seek(long pos) throws IOException;
  long getPos() throws IOException;
 boolean seekToNewSource(long targetPos) throws IOException;
}
```

seek() method seek the file to the given offset from the start of the file so that read () will stream from that location whereas getPos() method will return the current position on the InputStream.

Below sample code uses seek (), getPos () and read() method

```
FileSystem file = FileSystem.get (uri, conf);
        FSDataInputStream in = file.open(new Path(uri));
        byte[] btbuffer = new byte[5];
        in.seek(5); // sent to 5th position
        Assert.assertEquals(5, in.getPos());
        in.read(btbuffer, 0, 5);//read 5 byte in byte array from offset 0
        System.out.println(new String(btbuffer));// &amp;amp;amp;quot; print
5 character from 5th position
      in.read(10,btbuffer, 0, 5);// print 5 character staring from 10th
position
```

FSDataInputStream also implements PositionedReadable, which provide read, & readFully method to read part of file content from seek position as mentioned below

```
read(long position, byte[] buffer, int offset, int length)
```

FSDataOutputStream

Filesystem's create () method return FSDataOutputStream, which use to create new HDFS file or write the content at the EOF. It doesn't provide seek because of HDFS limitation to write to content at the

EOF only. It wrap Java IO's DataOutputStream and add method such as getPos() to get the position of the file and write() to write the content at the last position.

Below method signature provide FSDataOutputStream:

Create method on FileSystem create file e.g.

```
public FSDataOutputStream create(Path f) create empty file.
public FSDataOutputStream append(Path f) will append existing file
```

Create method also pass Progressable interface to track the status during file creation.

```
public FSDataOutputStream create(Path f, Progressable progress)
```

FileStatus

As describe below code getStatus() method of FileSystem provide HDFS file's meta information of HDFS file

```
URI uri=URI.create(strURI);
        FileSystem fileSystem=FileSystem.get(uri,conf);
        FileStatus fileStatus=fileSystem.getFileStatus(new Path(uri));
        System.out.println("AccessTime:"+fileStatus.getAccessTime());
        System.out.println("AccessTime:"+fileStatus.getLen());
        System.out.println("AccessTime:"+fileStatus.getModificationTime());
        System.out.println("AccessTime:"+fileStatus.getPath());
```

If your uri is directory not file then listStatus() will give you array of FileStatus[] as below

```
public FileStatus[] listStatus(Path f)
```

Directories

FileSystem provide method public boolean mkdirs (Path f) to create directory and its entire necessary child directory if not available. It returns true if directory created successfully. This is not mandatory as whenever you create file it will try to create necessary sub directories.

Delete file

Delete method on FileSystem remove the file/directory permanently

```
public boolean delete(Path f, boolean recursive) throws IOException
```

If recursive is true then it will delete a non-empty directory

Hive

Hive Architecture

Installation

The following steps are required for installing Hive on your system. Let us assume the Hive archive is downloaded onto the /Downloads directory.

Extracting and verifying Hive Archive

The following command is used to verify the download and extract the hive archive:

```
$ tar zxvf apache-hive-0.14.0-bin.tar.gz
$ ls
```

On successful download, you get to see the following response:

```
apache-hive-0.14.0-bin apache-hive-0.14.0-bin.tar.gz
```

Copying files to /usr/local/hive directory

We need to copy the files from the super user "su -". The following commands are used to copy the files from the extracted directory to the /usr/local/hive" directory.

```
$ su -
passwd:

# cd /home/user/Download
# mv apache-hive-0.14.0-bin /usr/local/hive
# exit
```

Setting up environment for Hive

You can set up the Hive environment by appending the following lines to ~/.bashrc file:

```
export HIVE_HOME=/usr/local/hive
export PATH=$PATH:$HIVE_HOME/bin
export CLASSPATH=$CLASSPATH:/usr/local/Hadoop/lib/*:.
export CLASSPATH=$CLASSPATH:/usr/local/hive/lib/*:.
```

The following command is used to execute ~/.bashrc file.

```
$ source ~/.bashrc
```

Configuring Hive

To configure Hive with Hadoop, you need to edit the hive-env.sh file, which is placed in the $HIVE_HOME/conf directory. The following commands redirect to Hive config folder and copy the template file:

```
$ cd $HIVE_HOME/conf
$ cp hive-env.sh.template hive-env.sh
```

Edit the hive-env.sh file by appending the following line:

```
export HADOOP_HOME=/usr/local/hadoop
```

Hive installation is completed successfully. Now you require an external database server to configure Metastore.

Comparison with Traditional Database

HiveQL

Querying Data

Sorting and Aggregating

Map Reduce Scripts

Joins & Sub queries