HDFS

"The value of an idea lies in using of it"

-Siva Kumar Bhuchipalli

www.hadooptutorial.info/

Contents

Content Creation
Creative Design
Strategy
Digital Marketing
Public Relations
Social Media

HDFS Introduction

HDFS is a distributed file system implemented on Hadoop's framework designed to store vast amount of data on low cost commodity hardware and ensuring high speed process on data.

Hadoop Distributed File System design is based on the design of Google File System. It's notion is "Write Once Read Multiple times".

HDFS Objectives

❖Able to store vast amount of data probably in Tera bytes or Peta bytes by spreading the data across a number of machines on cluster.

❖Storing data reliably, and in fault-tolerant manner by maintaining data replication to cope with loss of individual machines in the cluster

❖Able to process the data locally by moving the computation/processing to data nodes instead of bringing data from data nodes to computation server.

HDFS Limitations

❖As HDFS is designed on the notion of "Write Once, Read multiple times", once a file is written to HDFS, Then it can't be updated. But delete, append, and read Operations can be performed on HDFS files.

❖HDFS is not suitable for large number of small sized files but best suits for large sized files.
Because file system namespace (fsimage file)maintained by Namenode is limited by it's main memory capacity.

HDFS Design Concepts

Important components in HDFS Architecture:

BLOCKS

Name Node

Data Nodes

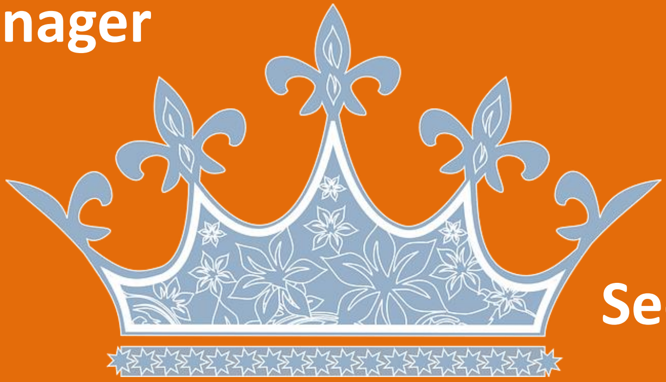
5 Daemons

Name Node

Resource Manager

Data Node

Node Manager



Secondary Name Node

Blocks

- HDFS is a block structured file system. Each HDFS file is broken into blocks of **fixed size usually 128 MB** which are stored across various data nodes on the cluster. Each of these blocks is stored as a separate file on local file system on data nodes (Commodity machines on cluster).

- Thus to access a file on HDFS, multiple data nodes need to be referenced and the list of the data nodes which need to be accessed is determined by the file system metadata stored on Name Node.

- So, any HDFS client trying to access/read a HDFS file, will get block information from Name Node first, and then based on the block id’s and locations, data will be read from corresponding data nodes/computer machines on cluster.

```
$ hadoop fsck / -files -blocks
```

BLOCKS

Advantages of Blocks

- **Quick Seek Time:**

  By default, HDFS Block Size is 128 MB which is much larger than any other file system. In HDFS, large block size is maintained to reduce the seek time for block access.

- **Ability to Store Large Files:**

  Another benefit is that, there is no need to store all blocks of a file on the same disk or node. So, a file's size can be larger than the size of a disk or node.

- **How Fault Tolerance is achieved with HDFS Blocks:**

  HDFS blocks feature suits well with the replication for providing fault tolerance and availability.

  - ✓ By default each block is replicated to three separate machines. This feature insures blocks against corrupted blocks or disk or machine failure.
  - ✓ If a block becomes unavailable, a copy can be read from another machine. And a block that is no longer available due to corruption or machine failure can be replicated from its alternative machines to other live machines to bring the replication factor back to the normal level (3 by default).

Name Node

- Name Node plays a Master role in Master/Slaves Architecture where as Data Nodes acts as slaves.
- File System metadata is stored on Name Node.
- File System Metadata contains majorly, File names, File Permissions and locations of each block of files. Thus, Metadata is relatively small in size and fits into Main Memory of a computer machine. So, it is stored in Main Memory of Name Node to allow fast access.

- Name Node is the single point of contact for accessing files in HDFS and it determines the block ids and locations for data access.

Important Components of Name Node

- FsImage: It is a file on Name Node's Local File System containing entire HDFS file system namespace (including mapping of blocks to files and file system properties)

- EditLog: It is a Transaction Log residing on Name Node's Local File System and contains a record/entry for every change that occurs to File System Metadata.

Since Name node is the only communication gateway for clients/users/applications to get the actual data residing on data nodes, It is a single point of failure and in fact, if this machine fails by any chance, then all the files on the HDFS file system would be lost since there is no other way of knowing how to reconstruct the files from the blocks on the data nodes.

So it is very important to keep the Name Node resilient to failure. One of the methods for this, is maintaining Secondary Name Node

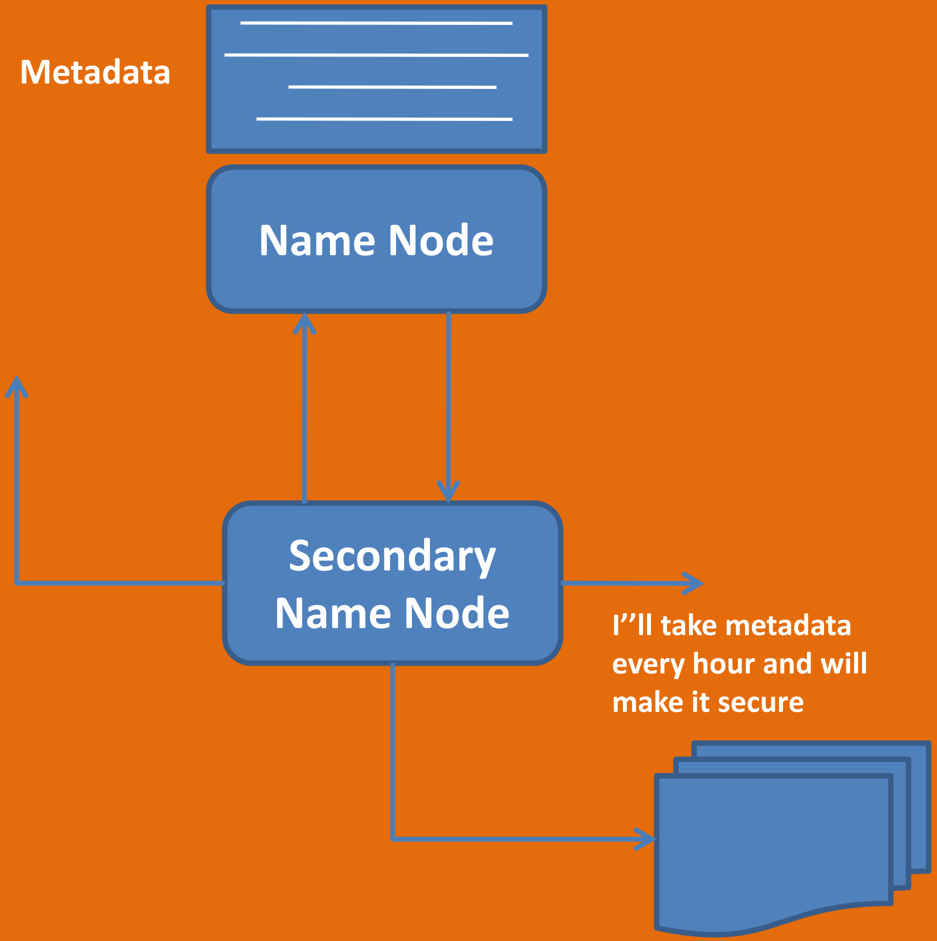
Data Node

- Slave Machines which stores the actual data of HDFS files
- Each data node on a cluster periodically sends a heartbeat message to the name node which is used by the name node to discover the data node failures based on missing heartbeats.

- The name node marks data nodes without recent heartbeats as dead, and does not dispatch any new I/O requests to them. Because data located at a dead data node is no longer available to HDFS
- Data node death may cause the replication factor of some blocks to fall below their specified values.
- Name node constantly tracks which blocks must be re-replicated, and initiates replication whenever necessary.

Thus all the blocks on a dead data node are re-replicated on other live data nodes and replication factor remains normal.

Secondary Name Node



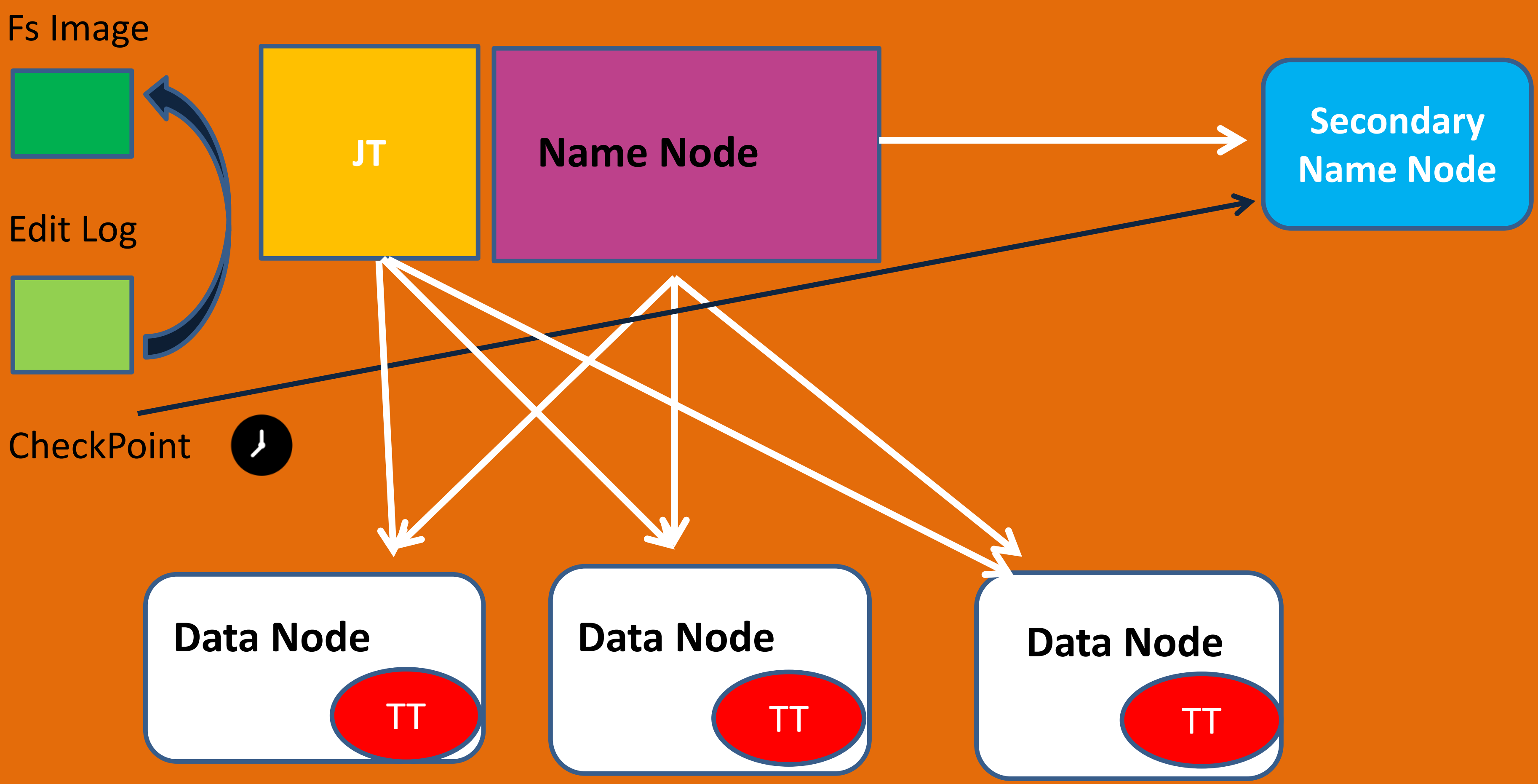
❖In HDFS 1.0, not a hot standby for the NameNode

❖By Default connects to NameNode every hour*

❖Housekeeping, backup of Name Node metadata

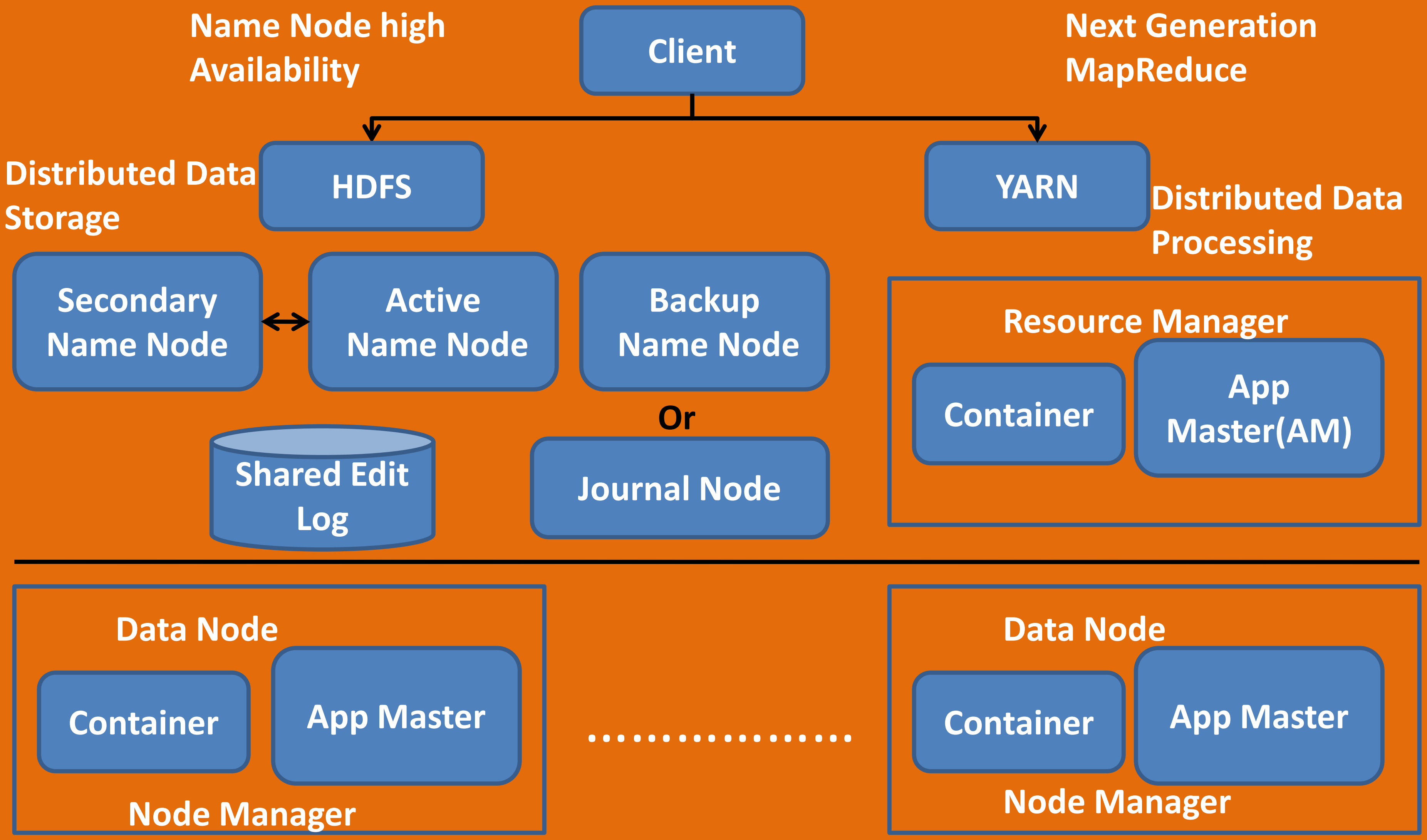
❖Saved metadata is used to bring up the Primary Name Node in case of its failure

HDFS 1.0 Architecture



TT---Task Tracker
JT---Job Tracker

HDFS 2.0 Architecture



HDFS WEB UI

http://namenode:50070/	- NameNode
http://namenode:50010/	- DataNode

Name Node

File Metadata:
/home/user/Mohan.txt → 1,2,3
/home/user/Sastry.txt →4,5

r=3

Hdfs-site.xml
↓
dfs.replication

3	
5	
2	

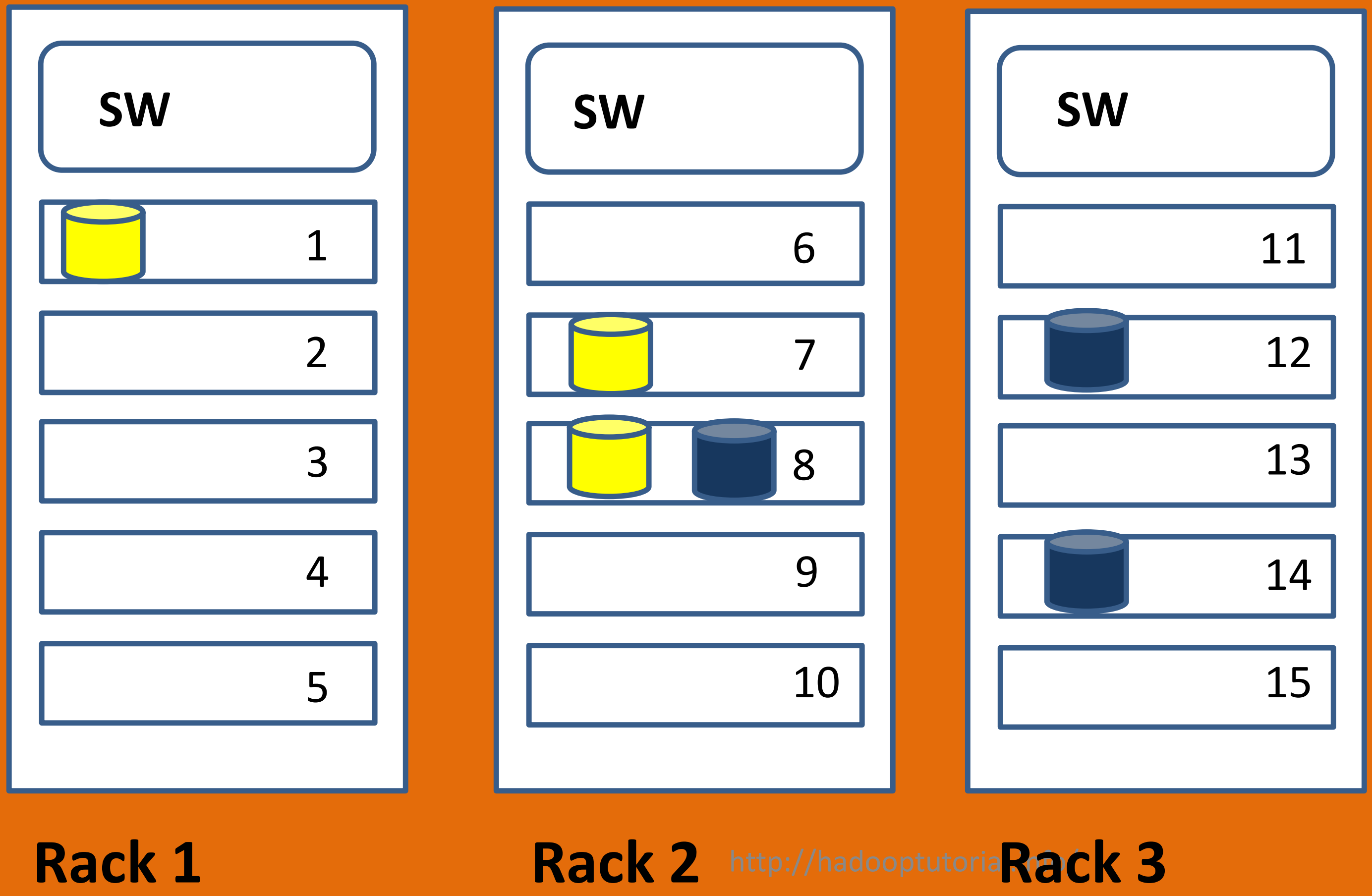
4	1
	2
	5

4	1
2	
5	3

1	
	4
	3

Data Nodes

Rack Awareness



http://hadooptutoria

Name Node

metadata

File.txt=

Blk A:
DN: 1, 7, 8

Blk B:
DN: 8, 12 ,14

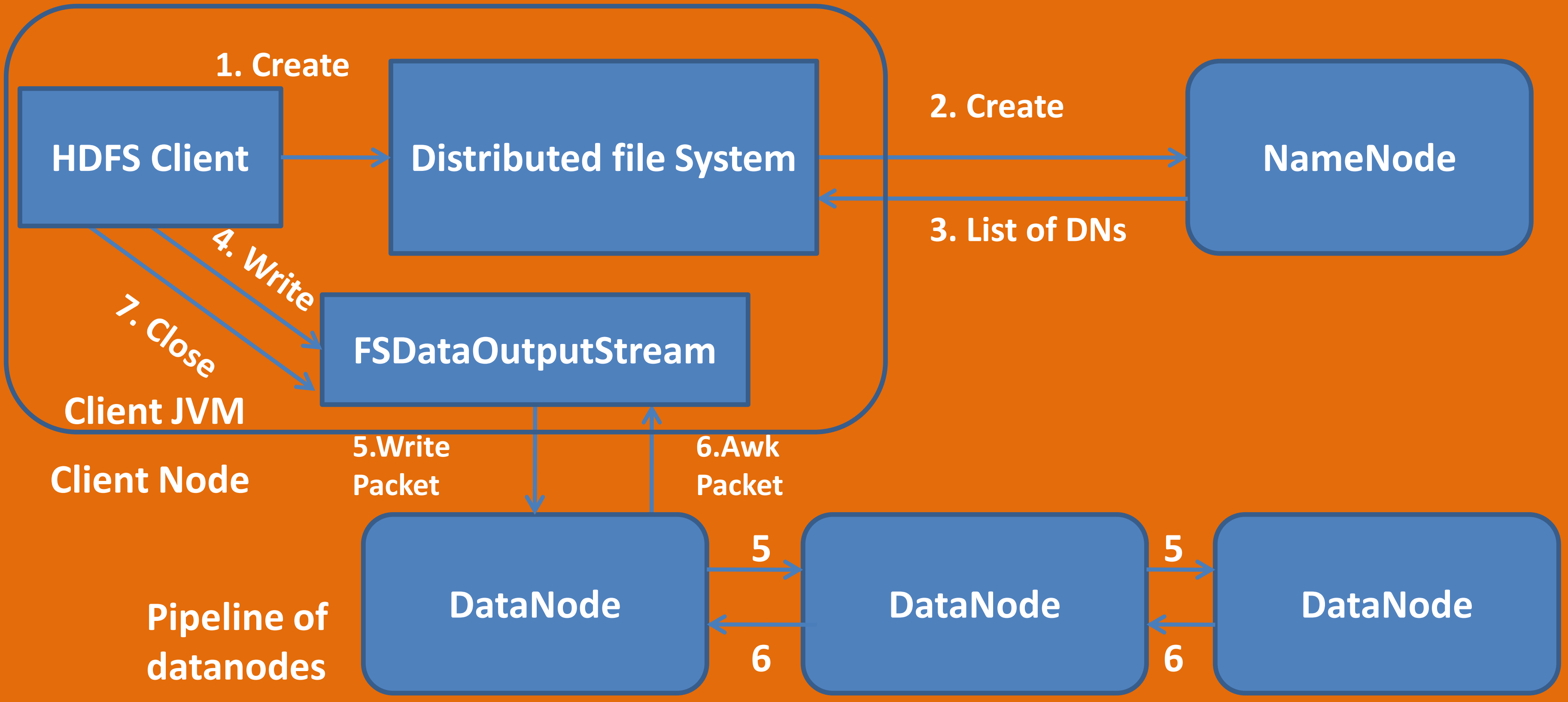
Rack Awareness

Rack1:
DN:1,2,3,4,5

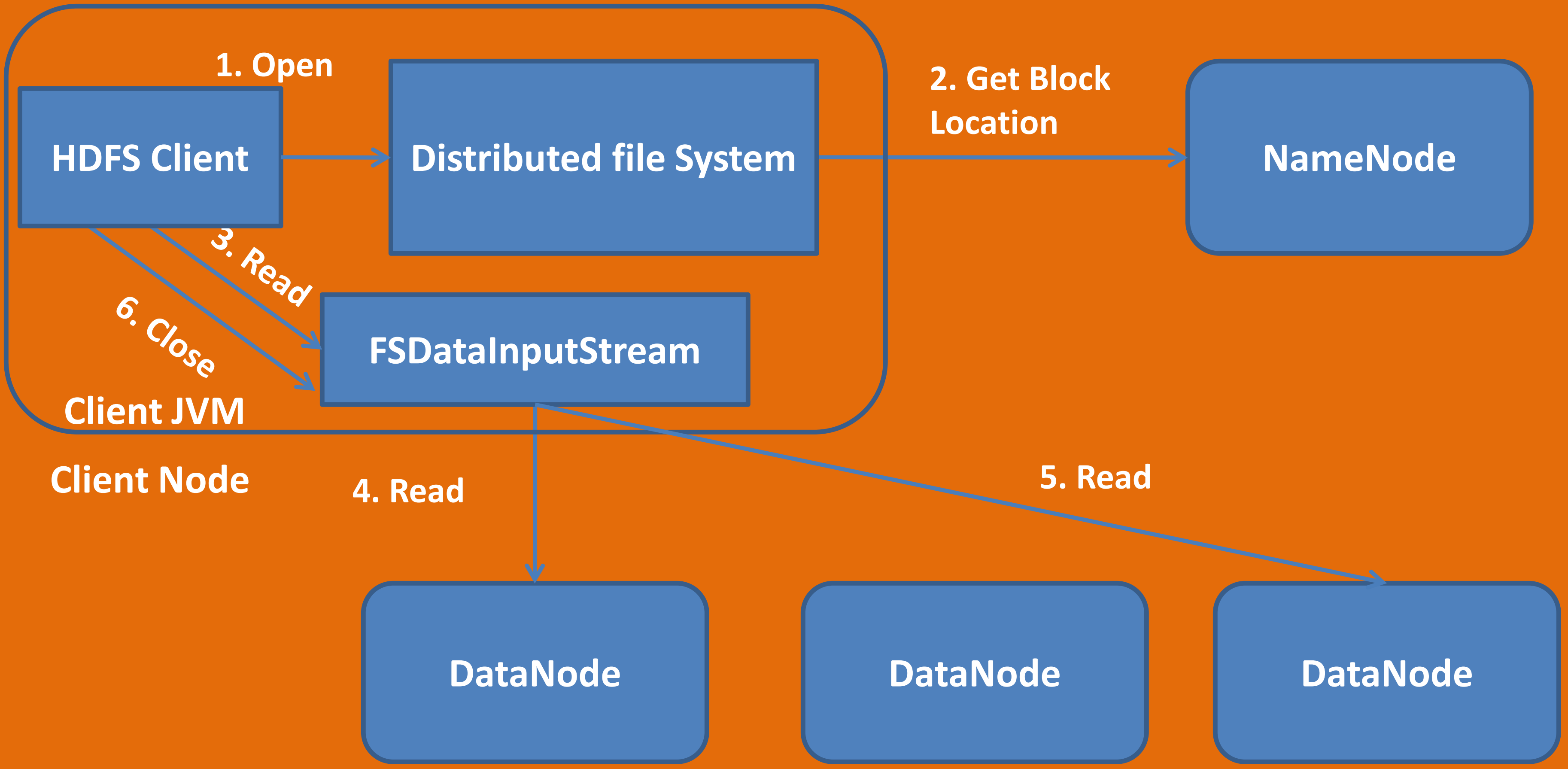
Rack2:
DN:6,7,8,9,10

Rack3:
DN:11,12,13,14,15

Anatomy Of File Write Operation In HDFS



Anatomy Of File Read Operation In HDFS



Safe Mode

Safe Mode in hadoop is a maintenance state of NameNode during which NameNode doesn't allow any changes to the file system.

During Safe Mode, HDFS cluster is read-only and doesn't replicate or delete blocks. At the start up of Name Node

- **It loads the file system namespace from the last saved fsimage into its main memory and the edits log file.**
- **Merges edits log file onto fsimage**
- **And then it receives block reports containing information about block locations from all data nodes**

Name Node enters safe mode automatically during its start up.

HDFS Rebalance

Whenever a new data node is added to the existing HDFS cluster or a data node is removed from the cluster then some of the data nodes in the cluster will have more/less blocks compared to other data nodes.

In such cases, to make all the data nodes space is uniformly utilized for blocks distribution, HDFS rebalance will be triggered by Hadoop Administrator.

Rebalancer is a administration tool in HDFS, to balance the distribution of blocks uniformly across all the data nodes in the cluster.
Rebalancing will be done on demand only. It will not get triggered automatically.

HDFS administrator issues this command on request to balance the cluster.

```
$ hdfs balancer
```

If a Rebalancer is triggered, NameNode will scan entire data node list and when

- Under-utilized data node is found, it moves blocks from over-utilized data nodes to this current data node
- If Over-utilized data node is found, it moves blocks from this data node to other under-utilized.
- Spreads different replicas of a block across the racks so that cluster can survive loss of an entire rack.

HDFS FileSystem Commands

1. mkdir

Similar to Unix mkdir command, it is used for creating directories in HDFS.

```
$ hadoop fs -mkdir  [-p] <path>
```

2. ls

Similar to Unix ls command, it is used for listing directories in HDFS.

The -lsr command can be used for recursive listing.

```
$ hadoop fs -ls [-d] [-h] [-R] <path>
```

-d Parent Directories are listed as plain files.

-h Formats the sizes of files in a human-readable fashion rather than a number of bytes.

-R Recursively list the contents of directories.

- List the contents that match the specified file pattern.
- If path is not specified, the contents of /user/<currentUser> will be listed.

- Directory entries are of the form:

permissions - userId groupId sizeOfDirectory(in bytes) modificationDate(yyyy-MM-dd HH:mm) directoryName

- Result of File entries will be of the form:

permissions numberOfReplicas userId groupId sizeOfFile(in bytes) modificationDate(yyyy-MM-dd HH:mm) fileName

```
$ hadoop fs -ls /
$ hadoop fs -lsr /
```

3. put

Copies files from local files system to HDFS. This is similar to -copyFromLocal Commands.

```
$ hadoop fs -put [-f] [-p] <localsrc> ... <dst>
```

- Copying fails if the file already exists, unless the -f flag is given.
- Passing -p preserves access and modification times, ownership and the mode. Passing -f overwrites the destination if it already exists

```
$ hadoop fs -put sample.txt /user/data/
```

4. get

- Copies files from HDFS to local file system.
- This is similar to -copyToLocal command.

```
$ hadoop fs -get /user/data/sample.txt workspace/
```

5. cat

Similar to UNIX cat command, it is used for displaying contents of the file.

```
$ hadoop fs -cat /user/data/sample.txt
```

6. cp

Similar to Unix cp command, it is used for copying files from one directory to another within HDFS.

```
$ hadoop fs -cp /user/data/sample.txt /user/hadoop
$ hadoop fs -cp /user/data/sample.txt /user/test/in
```

7. mv

Similar to Unix mv command, it is used for moving a file from one directory to another within HDFS.

```
$ hadoop fs -mv /user/hadoop/sample.txt /user/test/
```

8. rm

Similar to Unix rm command, it is used for removing a file from HDFS. The command -rmr can be used for recursive delete.

```
$ hadoop fs -rm [-f] [-r|-R] [-skipTrash] <src>
```

-skipTrash	option bypasses trash, if enabled, and immediately deletes <src>
-f	If the file does not exist, do not display a diagnostic message or modify the exit status to reflect an error.
-[rR]	Recursively deletes directories

```
$ hadoop fs -rm -r /user/test/sample.txt
```

Note:

Directories can't be deleted by -rm command. We need to use -rm -r (recursive remove) command to delete directories and files inside them. Only files can be deleted by -rm command.

9. getmerge

- It is used for merging a list of files in one directory on HDFS into a single file on local file system.

```
$ hadoop fs -getmerge /user/data
```

10. setrep

- This command is used to change the replication factor of a file to a specific instead of the default of replication factor for the remaining in HDFS.

- If <path> is a directory then the command recursively changes the replication factor of all files under the directory tree rooted at <path>.

```
$ hadoop fs -setrep [-R] [-w] <replication factor number> <file/path name>
```

-w It requests that the command waits for the replication to complete. This can potentially take a very long time.

-R It is accepted for backwards compatibility. It has no effect.

```
$ hadoop fs -setrep 2 /user/hadoop/
```

11. touchz

This command can be used to create a file of zero length in HDFS.

```
$ hadoop fs -touchz URI
```

12. test

This command can be used to test a hdfs file's existence or zero length or is it a directory. Syntax is

```
$ hadoop fs -test -[defsz] /user/test/test.txt
```

Options:

-d return 0 if <path> is a directory.
-e return 0 if <path> exists.
-f return 0 if <path> is a file.
-s return 0 if file <path> is greater than zero bytes in size.
-z return 0 if file <path> is zero bytes in size, else return 1.

13. expunge

This command is used to empty the trash in hadoop file system.

```
$ hadoop fs –expunge
```

14. appendToFile

Appends the contents of all the given local files to the given destination file on HDFS.

- ❑ The destination file will be created if it does not exist.
- ❑ If <localSrc> is –, then the input is read from stdin.

```
$ hadoop fs -appendToFile <local files separated by space> <hdfs destination file>
```

15. tail

Shows the last 1KB of the file.

```
$ hadoop fs -tail [-f] <file>
```

16. df

Shows the capacity, free and used space of the filesystem.

```
$ hadoop fs -df [-h] [<path> ...]
```

-h Formats the sizes of files in a human-readable fashion

17. du

Show the amount of space, in bytes, used by the files that match the specified file pattern.

```
$ hadoop fs -du [-s] [-h] <path>
```

18. count

Count the number of directories, files and bytes under the paths that match the specified file pattern.

```
$ hadoop fs -count [-q] <path>
```

The output columns are:

```
DIR_COUNT FILE_COUNT CONTENT_SIZE FILE_NAME or
QUOTA REMAINING_QUOTA SPACE_QUOTA REMAINING_SPACE_QUOTA
DIR_COUNT FILE_COUNT CONTENT_SIZE FILE_NAME
```

19. chgrp

Changes group of a file or path.

```
$ hadoop fs -chgrp [-R] groupname <path>
```

20. chmod

Changes permissions of a file. This works similar to the Linux shell's chmod command with a few exceptions.

```
$ hadoop fs -chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH
```

-R modifies the files recursively. This is the only option currently supported.
<MODE> Mode is the same as mode used for the shell's command. The only letters recognized are 'rwx', e.g. a+r,g-w,+rwx,o=r.
<OCTALMODE> Mode specifed in 3 or 4 digits. If 4 digits, the first may be 1 or 0 to turn the sticky bit on or off, respectively. Unlike the shell command, it is not possible to specify only part of the mode, e.g. 754 is same as u=rwx,g=rx,o=r

HDFS Distributed File Copy

Hadoop provides HDFS Distributed File copy (distcp) tool for copying large amounts of HDFS files within or in between HDFS clusters.

It is implemented based on Mapreduce framework and thus it submits a map-only mapreduce job to parallelize the copy process. Usually this tool is useful for copying files between clusters from production to development environments.

Basix Syntax is,

```
$hadoop distcp hdfs://namenodeX/src hdfs://namenodeY/dest
```

```
hadoop distcp -skipcrccheck -update
hdfs://lherhegapd034.enterprisenet.org:8020/user/babusi02/dimensions
hdfs://lherhegapq015.enterprisenet.org:8020/user/babusi02/ndx_met
```

Note:

hadoop's distcp is the better option to copy bulk data/ huge number of files from one machine to another (or cluster to cluster) than using fs -put or fs -cp commands.

dfsadmin-HDFS Administration Command

dfsadmin (distributed file system administration) command is used for file system administration activities like getting file system report, enter/leave safemode, refreshing nodes in the cluster and HDFS upgrade etc.

Below are the list of command options available with dfsadmin command.

- -report
- -safemode<enter|leave|get|wait>
- -refreshNodes
- -finalizeUpgrade
- -metasave<filename>
- -fetchImage<local file>
- -printTopology

Hadoop Archive Files

Hadoop archive files or HAR files are facility to pack HDFS files into archives. This is the best option for storing large number of small sized files in HDFS as storing large number of small sized files directly in HDFS is not very efficient.

The advantage of har files is that, these files can be directly used as input files in Map reduce jobs.

HAR File Creation :

```
$ hadoop archive -archiveName NAME -p <parent path> <src>* <dest>
```

Here the NAME should be in *.har format and src path should be relative to parent path. dest will be relative to user directory.

HAR File Deletion :

```
$ hadoop fs -rmr /user/hadoop1/archive/test.har
```

Limitation Of HAR Files

❖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.

❖Archives are immutable. Once an archive is created, to add or remove files from/to archive we need to re-create the archive.

❖HAR files can be used as input to Map Reduce but there is no archive-aware Input Format that can pack multiple files into a single Map Reduce split, so processing lots of small files, even in a HAR file will require lots of map tasks which are inefficient.

JAVA API

File System API Path

- ❖ Hadoop's Path object represents a file or a directory
  - Not java.io.File which tightly couples to local filesystem
- ❖ Path is really a URI on the FileSystem
  - ✓ HDFS: hdfs://localhost/user/file1
  - ✓ Local: file:///user/file1
- Examples:
  - ✓ new Path("/test/file1.txt");
  - ✓ new Path("hdfs://localhost:9000/test/file1.txt");

Hadoop Configuaration Project

- Configuration object stores clients' and servers' configuration
  - Very heavily used in Hadoop
    - HDFS, MapReduce, HBase, etc...
- Simple key-value paradigm
  - Wrapper for java.util.Properties class
- Several construction options
  - ✓ Configuration conf1 = new Configuration();
  - ✓ Configuration conf2 = new Configuration(conf1);
  - Configuration object conf2 is seeded with configurations of conf1 object

Getting properties is simple!

- Get the property
  - String nnName = conf.get("fs.default.name");
    - returns null if property doesn't exist
- Get the property and if doesn't exist return the provided default
  - String nnName = conf.get("fs.defaultFS","hdfs://localhost:9000");

There are also typed versions of these methods:

getBoolean, getInt, getFloat, etc...

Example: int prop = conf.getInt("file.size");

Reading Data From HDFS

- Create File System
- Open Input Stream to a Path
- Copy bytes using IOUtils
- Close Stream

Create File System

- ❖ FileSystem fs = FileSystem.get(new Configuration());

  If you run with yarn command, DistributedFileSystem (HDFS) will be created

  - ✓ Utilizes fs.defaultFS property from configuration
  - ✓ Recall that Hadoop framework loads core-site.xml which sets property to hdfs (hdfs://localhost:8020)

Open Input Stream To Path

```
…
InputStream input = null;
try {
input = fs.open(fileToRead);
…
```

- ❖ fs.open returns

  org.apache.hadoop.fs.FSDataInputStream

  Another FileSystem implementation will return their own custom implementation of InputStream
- ❖ Opens stream with a default buffer of 4k
- ❖ If you want to provide your own buffer size use

  fs.open(Path f, int bufferSize)

Copy Bytes Using IOUtils

```
IOUtils.copyBytes(inputStream, outputStream, buffer);
```

- Copy bytes from InputStream to OutputStream
- Hadoop's IOUtils makes the task simple

buffer parameter specifies number of bytes to buffer at a time.

```
...
} finally {
IOUtils.closeStream(input);
...
```

Utilize IOUtils to avoid boiler plate code that catches IOException

Reading Line by Line

```
Path pt=new Path(args[0]);
FileSystem fs = FileSystem.get(new Configuration());
BufferedReader br=new BufferedReader(new InputStreamReader(fs.open(pt)));
String line=br.readLine();
```

ReadFile.java Example

```
public class ReadFile
        {
        public static void main(String[] args)
        throws IOException
                {
                Path fileToRead = new Path("/training/data/readMe.txt");
                FileSystem fs = FileSystem.get(new Configuration());
                InputStream input = null;
                try
                        {
                        input = fs.open(fileToRead);
                        IOUtils.copyBytes(input, System.out, 4096);
                        }
                finally
                        {
                        IOUtils.closeStream(input);
                        }
                }
        }
```

1: Open File System

2: Open InputStream

3: Copy from Input to Output Stream

4: Close Stream

```
$ yarn jar HadoopSamples.jar hdfs.ReadFile readme.txt
```

Write Data

1. Create FileSystem instance
2. Open OutputStream
   - ✓ FSDataOutputStream in this case.
   - ✓ Open a stream directly to a Path from FileSystem.
   - ✓ Creates all needed directories on the provided path.
3. Copy data using IOUtils

WriteToFile.java Example

```
public class WriteToFile
{
        public static void main(String[] args) throws IOException
        {
        String textToWrite = "Hello HDFS! Elephants are awesome!\n";
        InputStream in = new BufferedInputStream(
        new ByteArrayInputStream(textToWrite.getBytes()));
        Path toHdfs = new Path("/training/workspace/writeMe.txt");
        Configuration conf = new Configuration();

        FileSystem fs = FileSystem.get(conf);        <-- 1: Create FileSystem instance

        FSDataOutputStream out = fs.create(toHdfs);  <-- 2: Open OutputStream

        IOUtils.copyBytes(in, out, conf);            <-- 3: Copy Data
        }
}
```

Run WriteToFile

```
$ yarn jar $PLAY_AREA/HadoopSamples.jar hdfs.WriteToFile
$ hdfs dfs -cat /training/playArea/writeMe.txt
Hello HDFS! Elephants are awesome!
```

FileSystem: Writing Data

- Append to the end of the existing file
  - ✓ Optional support by concrete FileSystem
  - ✓ HDFS supports
- No support for writing in the middle of the file
  - ✓ FileSystem's create and append methods have overloaded version that take callback interface to notify client of the progress

← Report progress to the screen

Delete Data

```
FileSystem.delete(Path path, Boolean recursive)
```

If recursive == true then non-empty directory will be deleted otherwise IOException is emitted

```
Path toDelete =
new Path("/training/playArea/writeMe.txt");
boolean isDeleted = fs.delete(toDelete, false);
System.out.println("Deleted: " + isDeleted);
```

```
$ yarn jar $PLAY_AREA/HadoopSamples.jar hdfs.DeleteFile
Deleted: true

$ yarn jar $PLAY_AREA/HadoopSamples.jar hdfs.DeleteFile
Deleted: false
```

File was already deleted by previous run

File System: mkdirs

- Create a directory - will create all the parent directories

```
Configuration conf = new Configuration();
Path newDir = new Path("/training/playArea/newDir");
FileSystem fs = FileSystem.get(conf);
boolean created = fs.mkdirs(newDir);
System.out.println(created);
```

```
$ yarn jar $PLAY_AREA/HadoopSamples.jar hdfs.MkDir true
```

File System: listStatus

- Browse the FileSystem with listStatus() methods

```
Path path = new Path("/");
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
FileStatus [] files = fs.listStatus(path);
for (FileStatus file : files ){
System.out.println(file.getPath().getName());
}
```

```
$ yarn jar $PLAY_AREA/HadoopSamples.jar
hdfs.SimpleLs
training
user
```

List files under "/" for HDFS

LsWithPathFilter.java example

```
FileSystem fs = FileSystem.get(conf);

FileStatus [] files = fs.listStatus(path, new PathFilter()
{
        @Override
        public boolean accept(Path path)
        {
        if (path.getName().equals("user"))
                {
                return false;
                }
                return true;
         }
};


for (FileStatus file : files )
{
System.out.println(file.getPath().getName());
}
```

Restrict result of listStatus() by supplying PathFilter object

Do not show path whose name equals to "user"

Run LsWithPathFilter Example

```
$ yarn jar $PLAY_AREA/HadoopSamples.jar hdfs.SimpleLs
training
User
```

```
$yarn jar $PLAY_AREA/HadoopSamples.jar hdfs.LsWithPathFilter
training
```

Any .... Questions????????

THANK