

USN: 

**CMR Institute of Technology, Bangalore**  
**II - INTERNAL ASSESSMENT**

Semester: 4-CBCS 2018  
Subject: **BIG DATA ANALYTICS (18MCA454)**  
Faculty: Ms Gomathi T

Date: 21 May 2021  
Time: 02:00 PM - 03:30 PM  
Max Marks: 50

<b>PART A</b>					
<i>Answer any 1 question(s)</i>					
Q.No		Marks	CO	PO	BT/CL
1	Define HDFS and explain HDFS concepts in detail.	10	CO4	PO3	L2
2	Explain HDFS Federation and HDFS High-availability	10	CO4	PO3	L2
<b>PART B</b>					
<i>Answer any 1 question(s)</i>					
Q.No		Marks	CO	PO	BT/CL
3	Write a java program to read data using the filesystem API and explain the reading process in detail.	10	CO4	PO3	L3
4	Write a java program to copy a local file to a Hadoop File System and explain the process in detail.	10	CO4	PO4	L3
<b>PART C</b>					
<i>Answer any 1 question(s)</i>					
Q.No		Marks	CO	PO	BT/CL
5	Demonstrate the anatomy of a file read with a neat diagram.	10	CO4	PO3	L2
6	Demonstrate the anatomy of a file write with a neat diagram.	10	CO4	PO3	L2
<b>PART D</b>					
<i>Answer any 1 question(s)</i>					
Q.No		Marks	CO	PO	BT/CL
7	Write a program to display files from a hadoop filesystem on standard output using a URLStreamHandler and demonstrate the steps in detail.	10	CO4	PO4	L3
8	Write a program to display the files from a Hadoop filesystem on standard output twice using seek and demonstrate the steps in detail.	10	CO4	PO4	L4
<b>PART E</b>					
<i>Answer any 1 question(s)</i>					
Q.No		Marks	CO	PO	BT/CL
9	Write a program to show the file status for a collection of paths in Hadoop Filesystem and illustrate the steps involved in getting the output	10	CO4	PO3	L4
10	Write a program for excluding paths that matches a regular expression and illustrate the steps involved in getting the output.	10	CO4	PO3	L4

## 1. Define HDFS and explain HDFS concepts.

HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware. Very large files “Very large” in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data.

### **Streaming data access**

HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, then various analyses are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record.

### **Commodity hardware**

Hadoop doesn't require expensive, highly reliable hardware to run on. It's designed to run on clusters of commodity hardware (commonly available hardware available from multiple vendors) for which the chance of node failure across the cluster is high, at least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

HDFS is not best fit for:

### **Low-latency data access:**

Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS. Remember, HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency.

### **Lots of small files:**

Since the namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode. As a rule of thumb, each file, directory, and block takes about 150 bytes. So, for example, if you had one million files, each taking one

block, you would need at least 300 MB of memory. While storing millions of files is feasible, billions is beyond the capability of current hardware.

***Multiple writers, arbitrary file modifications:***

Files in HDFS may be written to by a single writer. Writes are always made at the end of the file. There is no support for multiple writers, or for modifications at arbitrary offsets in the file.

**HDFS concepts:**

A disk has a block size, which is the minimum amount of data that it can read or write. Filesystems for a single disk build on this by dealing with data in blocks, which are an integral multiple of the disk block size. Filesystem blocks are typically a few kilobytes in size, while disk blocks are normally 512 bytes. This is generally transparent to the filesystem user who is simply reading or writing a file—of whatever length. However, there are tools to perform filesystem maintenance, such as `df` and `fsck`, that operate on the filesystem block level.

HDFS, too, has the concept of a block, but it is a much larger unit—64 MB by default. Like in a filesystem for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units.

**Name nodes and Data nodes:**

An HDFS cluster has two types of node operating in a master-worker pattern: a namenode (the master) and a number of datanodes (workers). The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts.

A client accesses the filesystem on behalf of the user by communicating with the namenode and datanodes. The client presents a POSIX-like filesystem interface, so the user code does not need to know about the namenode and datanode to function. Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing.

## 2. Explain HDFS Federation and HDFS High-availability

### HDFS Federation

The name node keeps a reference to every file and block in the file system in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling.

HDFS Federation, introduced in the 0.23 release series, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace. For example, one namenode might manage all the files rooted under /user, say, and a second namenode might handle files under /share.

Under federation, each namenode manages a namespace volume, which is made up of the metadata for the namespace, and a block pool containing all the blocks for the files in the namespace. Namespace volumes are independent of each other, which means namenodes do not communicate with one another, and furthermore the failure of one namenode does not affect the availability of the namespaces managed by other namenodes.

To access a federated HDFS cluster, clients use client-side mount tables to map file paths to namenodes.

This is managed in configuration using the ViewFileSystem, and viewfs:// URIs.

### HDFS High-availability:

The combination of replicating namenode metadata on multiple filesystems, and using the secondary namenode to create checkpoints protects against data loss, but does not provide high-availability of the filesystem. 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.

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

### 3. Write a program to copy a local file to a hadoop filesystem

```

Public class FileCopuWithProgress{
Public static void main(String[] args) throws exception{
String localsrc=args[0];
String dst=args[1];
InputStream in=new BufferedInputStream(new FileInputStream(localsrc));
Configuration conf=new Configuration();
FileSystem fs=FileSystem.get(URI.create(dst),conf);
OutputStream out=fs.create(new path(dst),new Progressable(){
Public void progress(){ system.out.print(".");
}});
IOUtils.copyBytes(in, out, 4096, false);
}
}

```

### 4. Write a program to read data using the filesystem API

```

Public class FileSystemCat{
Public static void main(String[] args) throws exception{
String uri=args[0];
Configuration conf=new Configuration();
FileSystem fs=FileSystem.get(URI.create(uri),conf);
InputStream in=null;
Try
{
In=fs.open(new Path(uri));
IOUtils.copyBytes(in, System.out, 4096, false);
}
}

```

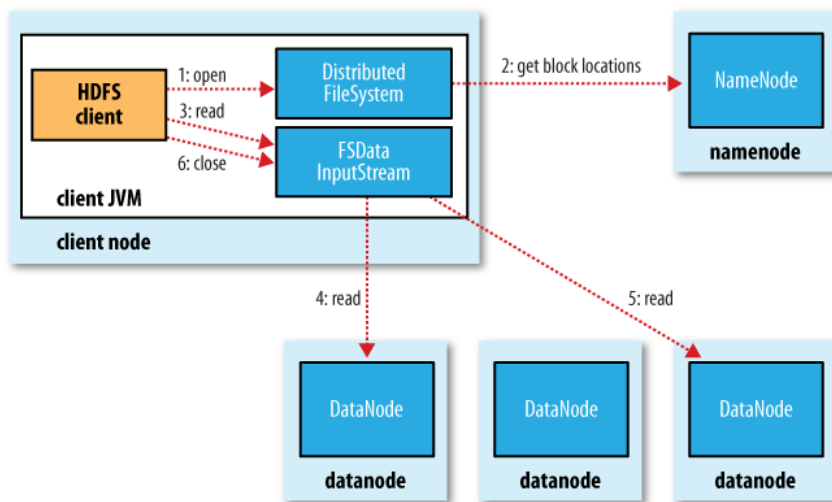
```

}
Finally{IOUtils.closestream(in);
}
}
}
}

```

**5. Demonstrate the anatomy of a file read with a neat diagram**

To get an idea of how data flows between the client interacting with HDFS, the namenode and the datanodes, consider Figure which shows the main sequence of events when reading a file.



The client opens the file it wishes to read by calling open() on the FileSystem object, which for HDFS is an instance of DistributedFileSystem (step 1 in Figure).

DistributedFileSystem calls the namenode, using RPC, to determine the locations of the blocks for the first few blocks in the file (step 2).

For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Furthermore, the datanodes are sorted according to their proximity to the node. If the client is itself a datanode (in the case of a MapReduce task, for instance), then it will read from the local datanode, if it hosts a copy of the block. The DistributedFileSystem returns an FSDDataInputStream (an input stream that supports file seeks) to the client for it to read data from. FSDDataInputStream in turn wraps a DFSInputStream, which manages the datanode and namenode I/O.

The client then calls `read()` on the stream (step 3). `DFSInputStream`, which has stored the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file. Data is streamed from the datanode back to the client, which calls `read()` repeatedly on the stream (step 4).

When the end of the block is reached, `DFSInputStream` will close the connection to the datanode, then find the best datanode for the next block (step 5). This happens transparently to the client, which from its point of view is just reading a continuous stream. Blocks are read in order with the `DFSInputStream` opening new connections to datanodes as the client reads through the stream. It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed. When the client has finished reading, it calls `close()` on the `FSDatInputSteam`(step 6)

## **6. Demonstrate the anatomy of a file write with a neat diagram.**

The case we're going to consider is the case of creating a new file, writing data to it, then closing the file. See Figure 3-4. The client creates the file by calling `create()` on `DistributedFileSystem` (step 1 in Figure 3-4). `DistributedFileSystem` makes an RPC call to the namenode to create a new file in the filesystem's namespace, with no blocks associated with it (step 2). The namenode 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 namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an `IOException`.

The `DistributedFileSystem` returns an `FSDatOutputSteam` for the client to start writing data to. Just as in the read case, `FSDatOutputSteam` wraps a `DFSOutput Stream`, which handles communication with the datanodes and namenode.

As the client writes data (step 3), `DFSOutputSteam` splits it into packets, which it writes to an internal queue, called the data queue. The data queue is consumed by the Data Streamer, whose responsibility it is to ask the namenode to allocate new blocks by

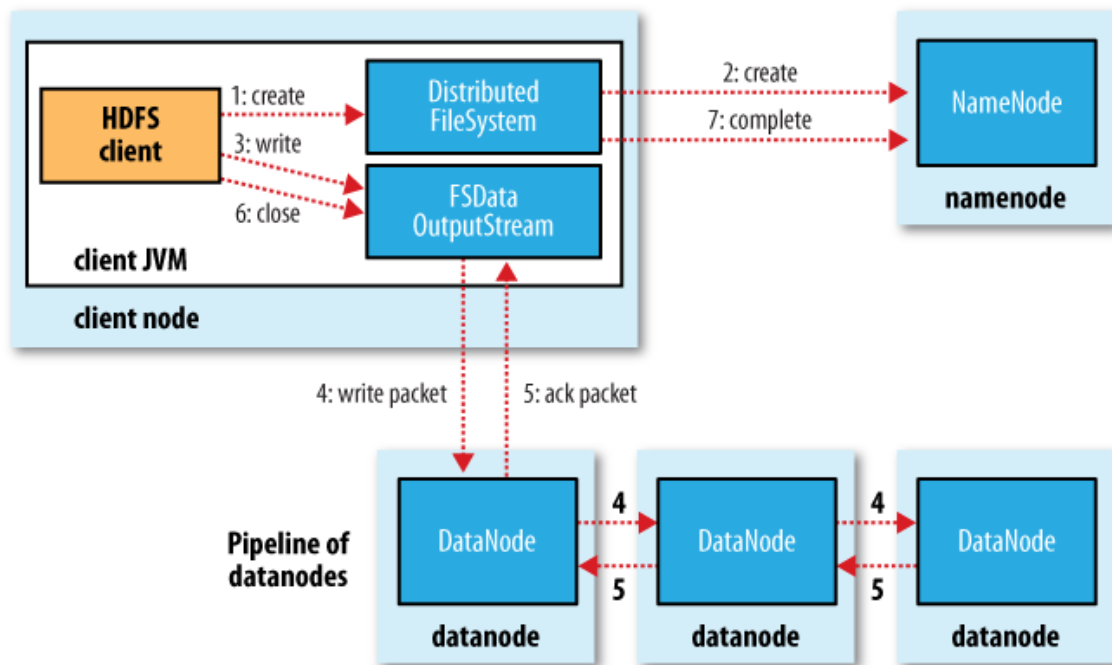
picking a list of suitable datanodes to store the replicas. The list of datanodes forms a

pipeline—we'll assume the replication level is three, so there are three nodes in the

pipeline. The `DataStreamer` streams the packets to the first datanode in the pipeline,

which stores the packet and forwards it to the second datanode in the pipeline. Similarly, the second datanode stores the packet and forwards it to the third (and last)

datanode in the pipeline (step 4).



4

7. Write a program to display files from a hadoop filesystem on standard output using a URLStreamHandler.

*Example 3-1. Displaying files from a Hadoop filesystem on standard output using a URLStreamHandler*

```
public class URLLCat {

    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);
        }
    }
}
```



**8. Write a program to display the files from a Hadoop filesystem on standard output twice using seek.**

```
Public class FileSystemDoubleCat
{
Public static void Main(string[] args) throws Exception
{
String uri=args[0];
Configuration conf= new Configuration();
FileSystem fs=FileSystem.get(URI.create(uri),conf);
FSDataStream in = null;
try
{
in = fs.Open(new Path(uri));
IOUtils.CopyBytes(in,System.Out,4096,false);
In.Seek(0);
IOUtils.CopyBytes(in,System.Out,4096,false);
}
Finally{IOUtils.CloseStream(in);}
}
}
```

**9. Write a program to show the file status for a collection of paths in Hadoop Filesystem**

```
Public class Liststatus
{
Public static void main(string[] args) throws Exception{
String uri=args[0];
Configuration conf=new Configuration();
FileSystem fs=FileSystem.get(URI.create(uri),conf);

Path[] paths=new Path[args.length];
```

```
For(int i=0;i<paths.length;i++) {  
Paths[i]=new path(args[i]);  
}  
Filestatus[] status=fs.listStatus(paths);  
Path[] listedpaths=FileUtil.stat2Paths(status);  
For(Path p:listedpaths){ System.out.println(p);}}
```

**10. Write a program for excluding paths that matches a regular expression.10**

```
Public class RegexExcludePathFilter implements PathFilter  
{  
Private final string regex;  
Public RegexExcludePathFilter(string regex)  
{  
this.regex=regex;  
}  
Public boolean accept(Path path)  
{ return !path.toString().matches(regex);  
}  
}
```