Big Data Analytics CSF443

UNIT – II

HADOOP DISTRIBUTED FILE SYSTEM (HDFS)

1. Design of HDFS:

- HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.
- There are Hadoop clusters running today that store petabytes of data.
- 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 a source, then various analyses are
  performed on that dataset over time.

- 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.
- 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.
- Files in HDFS may be written 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.

2. HDFS Concepts :

- **Blocks :**
- HDFS has the concept of a block, but it is a much larger unit—64 MB by default.

- Files in HDFS are broken into block-sized chunks, which are stored as independent units.
- Having a block abstraction for a distributed filesystem brings several benefits. :

1. A file can be larger than any single disk in the network. Nothing requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster.
2. Making the unit of abstraction a block rather than a file simplifies the storage subsystem. It simplifies the storage management (since blocks are a fixed size, it is easy to calculate how many can be stored on a given disk) and eliminating metadata concerns.
3. Blocks fit well with replication for providing fault tolerance and availability. To insure against corrupted blocks and disk and machine failure, each block is replicated to a small number of physically separate machines.

- HDFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks.
- **Namenodes and Datanodes:**
    - An HDFS cluster has two types of nodes operating in a master-worker pattern:

1. **A Namenode (the master) and**
2. **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
- The edit log.
- The namenode also knows the datanodes on which all the blocks for a given file are located.

3. Benefits and Challenges:

Benefits of HDFS:

- HDFS can store a large amount of information.

- HDFS is a simple & robust coherency model.
- HDFS is scalable and has fast access to required information.
- HDFS also serve a substantial number of clients by adding more machines to the cluster.
- HDFS provides streaming read access.
- HDFS can be used to read data stored multiple times, but the data will be written to the HDFS once.
- he recovery techniques will be applied very quickly.
- Hardware and operating systems portability across is heterogeneous commodities.
- High Economy by distributing data and processing across clusters of commodity personal computers.
- High Efficiency by distributing data, logic on parallel nodes to process it from where data is located.
- High Reliability by automatically maintaining multiple copies of data and automatically redeploying processing logic in the event of failures.

Challenges for HDFS:

- HDFS does not give any reliability if that machine goes down.
- An enormous number of clients must be handled if all the clients need the data stored on a single machine.
- Clients need to copy the data to their local machines before they can operate it.
- Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS.
- 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.
- Files in HDFS may be written by a single writer. Writers 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.

File Size :

- You can use the **Hadoop fs -ls** command to list files in the current directory as well as their details.

- The 5th column in the command output contains file size in bytes.
- You can also find file size using **hadoop fs -dus <path>**.
- For example, if a directory on HDFS named "/user/frylock/input" contains 100 files and you need the total size for all of those files you could run:
  hadoop fs -dus /user/frylock/input

- And you would get back the total size (in bytes) of all of the files in the "/user/frylock/input" directory.
- You can also use the following function to find the file size :

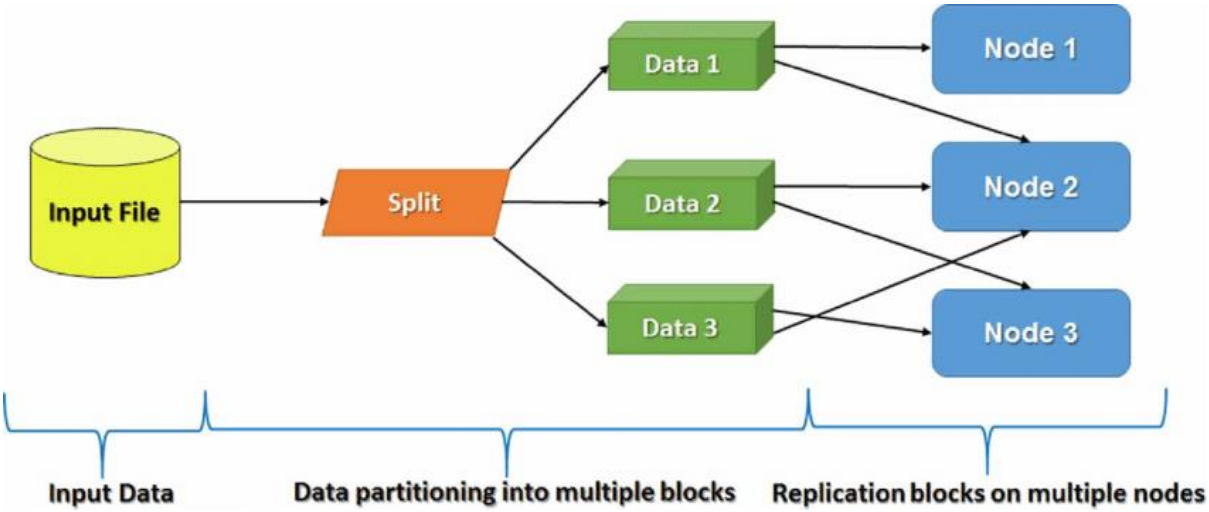
```
public class GetflStatus
{
    public long getflSize(String args) throws IOException,
FileNotFoundException
    {
        Configuration config = new Configuration();
        Path path = new Path(args);
        FileSystem hdfs = path.getFileSystem(config);
        ContentSummary cSummary = hdfs.getContentSummary(path);
        long length = cSummary.getLength();
        return length;
    }
}
```

HDFS Block abstraction:

- HDFS block size is usually 64MB-128MB and unlike other filesystems, a file smaller than the block size does not occupy the complete block size's worth of memory.
- The block size is kept so large so that less time is made doing disk seeks as compared to the data transfer rate.
- Why do we need block abstraction:

- Files can be bigger than individual disks.
- Filesystem metadata does not need to be associated with each and every block.
- Simplifies storage management - Easy to figure out the number of blocks which can be stored on each disk.
- Fault tolerance and storage replication can be easily done on a per-block basis.

Data Replication :



- Replication ensures the availability of the data.
- Replication is - making a copy of something and the number of times you make a copy of that particular thing can be expressed as its Replication Factor.

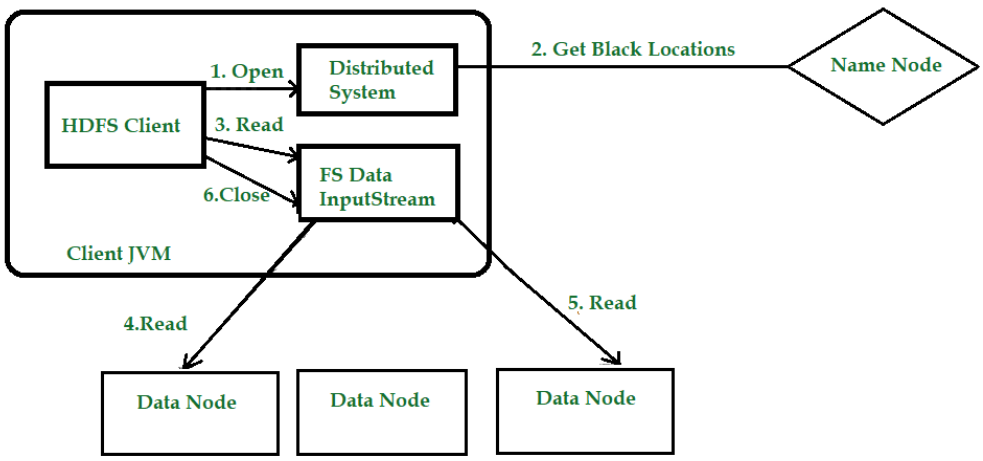
- As HDFS stores the data in the form of various blocks at the same time Hadoop is also configured to make a copy of those file blocks.
- By default, the Replication Factor for Hadoop is set to 3 which can be configured.
- We need this replication for our file blocks because for running Hadoop we are using commodity hardware (inexpensive system hardware) which can be crashed at any time.
- We are not using a supercomputer for our Hadoop setup.
- That is why we need such a feature in HDFS that can make copies of that file blocks for backup purposes, this is known as **fault tolerance**.

- For the big brand organization, the data is very much important than the storage, so nobody cares about this extra storage.
- You can configure the Replication factor in your hdfs-site.xml file.

```
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
```

How Does Hadoop Store , read , write files :

1. **Read Files :**



Step 1: The client opens the file he/she wishes to read by calling open() on the File System Object.
Step 2: Distributed File System(DFS) calls the name node, to determine the locations of the first few blocks in the file. For each block, the name node returns the addresses of the data nodes that have a copy of that block. The DFS returns an FSDataInputStream to the client for it to read data from.
Step 3: The client then calls read() on the stream. DFSInputStream, which has stored the info node addresses for the primary few blocks

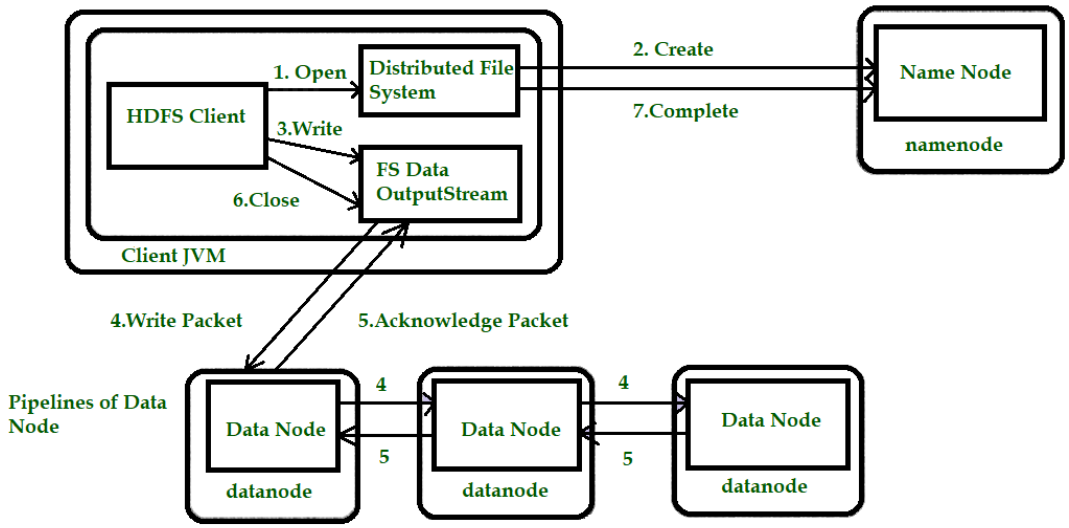
within the file, then connects to the primary (closest) data node for the primary block in the file.

Step 4: Data is streamed from the data node back to the client, which calls read() repeatedly on the stream.

Step 5: When the end of the block is reached, DFSInputStream will close the connection to the data node, then finds the best data node for the next block.

Step 6: When the client has finished reading the file, a function is called, close() on the FSDataInputStream.

1. **Write Files :**



Step 1: The client creates the file by calling create() on DistributedFileSystem(DFS).

Step 2: DFS makes an RPC call to the name node to create a new file in the file system's namespace, with no blocks associated with it. The name node 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 name node prepares a record of the new file; otherwise, the file can't be created. The DFS returns an FSDataOutputStream for the client to start out writing data to the file.

Step 3: Because the client writes data, the DFSOutputStream splits it into packets, which it writes to an indoor queue called the info queue. The data queue is consumed by the DataStreamer, which is liable for asking the name node to allocate new blocks by picking an inventory of suitable data nodes to store the replicas. The list of data nodes forms a pipeline. The DataStreamer streams the packets to the primary data node within the pipeline, which stores each packet and forwards it to the second data node within the pipeline.

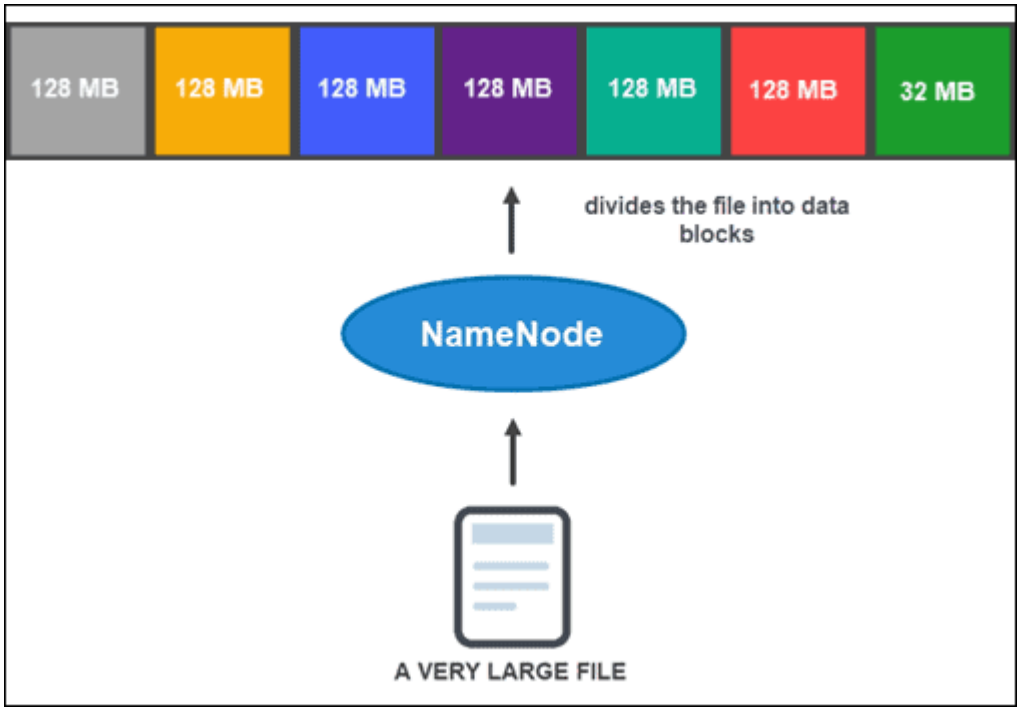
Step 4: Similarly, the second data node stores the packet and forwards it to the third (and last) data node in the pipeline.

Step 5: The DFSOutputStream sustains an internal queue of packets that are waiting to be acknowledged by data nodes, called an “ack queue”.

Step 6: This action sends up all the remaining packets to the data node pipeline and waits for acknowledgements before connecting to the name node to signal whether the file is complete or not.

1. **Store Files :**

- HDFS divides files into blocks and stores each block on a DataNode.
- Multiple DataNodes are linked to the master node in the cluster, the NameNode.
- The master node distributes replicas of these data blocks across the cluster.
- It also instructs the user where to locate wanted information.
- Before the NameNode can help you store and manage the data, it first needs to partition the file into smaller, manageable data blocks.
- This process is called **data block splitting**.



Java Interfaces to HDFS :

- Java code for writing file in HDFS :

```
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();
```

- Java code for reading file in HDFS :

```
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();
```

Command Line Interface :

- The HDFS can be manipulated through a Java API or through a command-line interface.
- The File System (FS) shell includes various shell-like commands that directly interact with the Hadoop Distributed File System (HDFS) as well as other file systems that Hadoop supports.
- Below are the commands supported :
- **appendToFile:** Append the content of the text file in the HDFS.
- **cat:** Copies source paths to stdout.
- **checksum: Returns the checksum information of a file.**
- **chgrp :** Change group association of files. The user must be the owner of files, or else a super-user.
- **chmod :** Change the permissions of files. The user must be the owner of the file, or else a super-user.
- **chown:** Change the owner of files. The user must be a super-user.
- **copyFromLocal: This command copies all the files inside the test folder in the edge node to the test folder in the HDFS.**

- **copyToLocal : This command copies all the files inside the test folder in the HDFS to the test folder in the edge node.**
- **count:** Count the number of directories, files and bytes under the paths that match the specified file pattern.
- **cp: Copy files from source to destination. This command allows multiple sources as well in which case the destination must be a directory.**
- **createSnapshot:** HDFS Snapshots are read-only point-in-time copies of the file system. Snapshots can be taken on a subtree of the file system or the entire file system. Some common use cases of snapshots are data backup, protection against user errors and disaster recovery.
- **deleteSnapshot:** Delete a snapshot from a snapshot table directory. This operation requires the owner privilege of the snapshottable directory.
- **df: Displays free space**
- **du: Displays sizes of files and directories contained in the given directory or the length of a file in case its just a file.**
- **expunge:** Empty the Trash.
- **find: Finds all files that match the specified expression and applies selected actions to them. If no path is specified then defaults to the current working directory. If no expression is specified then defaults to -print.**
- **get** Copy files to the local file system.
- **getfacl: Displays the Access Control Lists (ACLs) of files and directories. If a directory has a default ACL, then getfacl also displays the default ACL.**
- **getfattr: Displays the extended attribute names and values for a file or directory.**
- **getmerge :** Takes a source directory and a destination file as input and concatenates files in src into the destination local file.
- **help: Return usage output.**
- **ls: list files**
- **lsr: Recursive version of ls.**
- **mkdir: Takes path URI's as argument and creates directories.**

- **moveFromLocal: Similar to put command, except that the source localsrc is deleted after it's copied.**
- **moveToLocal: Displays a "Not implemented yet" message.**
- **mv: Moves files from source to destination. This command allows multiple sources as well in which case the destination needs to be a directory.**
- **put : Copy single src, or multiple srcs from local file system to the destination file system. Also reads input from stdin and writes to destination file system.**
- **renameSnapshot : Rename a snapshot. This operation requires the owner privilege of the snapshottable directory.**
- **rm : Delete files specified as args.**
- **rmdir : Delete a directory.**
- **rmr : Recursive version of delete.**
- **setfacl : Sets Access Control Lists (ACLs) of files and directories.**
- **setfattr : Sets an extended attribute name and value for a file or directory.**
- **setrep: Changes the replication factor of a file. If the path is a directory then the command recursively changes the replication factor of all files under the directory tree rooted at the path.**
- **stat : Print statistics about the file/directory at <path> in the specified format.**
- **tail: Displays the last kilobyte of the file to stdout.**
- **test : Hadoop fs -test -[defsz] URI.**
- **text: Takes a source file and outputs the file in text format. The allowed formats are zip and TextRecordInputStream.**
- **touchz: Create a file of zero length.**
- **truncate:** Truncate all files that match the specified file pattern to the specified length.
- **usage: Return the help for an individual command.**

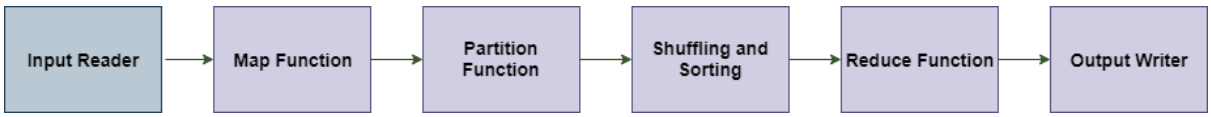
HDFS Interfaces :

Features of HDFS interfaces are :

1. Create new file
2. Upload files/folder
3. Set Permission
4. Copy
5. Move
6. Rename
7. Delete
8. Drag and Drop
9. HDFS File viewer

Data Flow :

- MapReduce is used to compute a huge amount of data.
- To handle the upcoming data in a parallel and distributed form, the data has to flow from various phases :



- **Input Reader :**
- The input reader reads the upcoming data and splits it into the data blocks of the appropriate size (64 MB to 128 MB).
- Once input reads the data, it generates the corresponding key-value pairs.
- The input files reside in HDFS.
- **Map Function :**
- The map function process the upcoming key-value pairs and generated the corresponding output key-value pairs.
- The mapped input and output types may be different from each other.
- **Partition Function :**
- The partition function assigns the output of each Map function to the appropriate reducer.
- The available key and value provide this function.
- It returns the index of reducers.
- **Shuffling and Sorting :**

- The data are shuffled between nodes so that it moves out from the map and get ready to process for reduce function.
- The sorting operation is performed on input data for Reduce function.
- **Reduce Function :**
- The Reduce function is assigned to each unique key.
- These keys are already arranged in sorted order.
- The values associated with the keys can iterate the Reduce and generates the corresponding output.
- **Output Writer :**
- Once the data flow from all the above phases, the Output writer executes.
- The role of the Output writer is to write the Reduce output to the stable storage.

Data Ingestion :

- Hadoop Data ingestion is the beginning of your data pipeline in a data lake.
- It means taking data from various silo databases and files and putting it into Hadoop.
- For many companies, it does turn out to be an intricate task.
- That is why they take more than a year to ingest all their data into the Hadoop data lake.
- The reason is, as Hadoop is open-source; there are a variety of ways you can ingest data into Hadoop.
- It gives every developer the choice of using her/his favourite tool or language to ingest data into Hadoop.
- Developers while choosing a tool/technology stress on performance, but this makes governance very complicated.
- **Sqoop :**
- Apache Sqoop (SQL-to-Hadoop) is a lifesaver for anyone who is experiencing difficulties in moving data from the data warehouse into the Hadoop environment.

- Apache Sqoop is an effective Hadoop tool used for importing data from RDBMS's like MySQL, Oracle, etc. into HBase, Hive or HDFS.
- Sqoop Hadoop can also be used for exporting data from HDFS into RDBMS.
- Apache Sqoop is a command-line interpreter i.e. the Sqoop commands are executed one at a time by the interpreter.
- **Flume :**
- Apache Flume is a service designed for streaming logs into the Hadoop environment.
- Flume is a distributed and reliable service for collecting and aggregating huge amounts of log data.
- With a simple and easy to use architecture based on streaming data flows, it also has tunable reliability mechanisms and several recoveries and failover mechanisms.

Hadoop Archives :

- Hadoop Archive is a facility that packs up small files into one compact HDFS block to avoid memory wastage of name nodes.
- Name node stores the metadata information of the HDFS data.
- If 1GB file is broken into 1000 pieces then namenode will have to store metadata about all those 1000 small files.
- In that manner,namenode memory will be wasted in storing and managing a lot of data.
- HAR is created from a collection of files and the archiving tool will run a MapReduce job.
- These Maps reduces jobs to process the input files in parallel to create an archive file.
- Hadoop is created to deal with large files data, so small files are problematic and to be handled efficiently.
- As a large input file is split into a number of small input files and stored across all the data nodes, all these huge numbers of records are to be stored in the name node which makes the name node inefficient.

- To handle this problem, Hadoop Archive has been created which packs the HDFS files into archives and we can directly use these files as input to the MR jobs.
- It always comes with ***.har** extension.
- **HAR Syntax :**
- `hadoop archive -archiveName NAME -p <parent path> <src>* <dest>`
- **Example :**
  `hadoop archive -archiveName foo.har -p /user/hadoop dir1 dir2 /user/zoo`

I/O Compression :

- In the Hadoop framework, where large data sets are stored and processed, you will need storage for large files.
- These files are divided into blocks and those blocks are stored in different nodes across the cluster so lots of I/O and network data transfer is also involved.
- In order to reduce the storage requirements and to reduce the time spent in-network transfer, you can have a look at data compression in the Hadoop framework.
- Using data compression in Hadoop you can compress files at various steps, at all of these steps it will help to reduce storage and quantity of data transferred.
- You can compress the input file itself.
- That will help you reduce storage space in HDFS.
- You can also configure that the output of a MapReduce job is compressed in Hadoop.
- That helps is reducing storage space if you are archiving output or sending it to some other application for further processing.

I/O Serialization :

- Serialization refers to the conversion of structured objects into byte streams for transmission over the network or permanent storage on a disk.

- Deserialization refers to the conversion of byte streams back to structured objects.
- Serialization is mainly used in two areas of distributed data processing :
- Interprocess communication
- Permanent storage
- We require I/O Serialization because :
- To process records faster (Time-bound).
- When proper data formats need to maintain and transmit over data without schema support on another end.
- When in the future, data without structure or format needs to process, complex Errors may occur.
- Serialization offers data validation over transmission.
- To maintain the proper format of data serialization, the system must have the following four properties -
- **Compact -** helps in the best use of network bandwidth
- **Fast -** reduces the performance overhead
- **Extensible -** can match new requirements
- **Inter-operable -** not language-specific

Avro :

- Apache Avro is a language-neutral data serialization system.
- Since Hadoop writable classes lack language portability, Avro becomes quite helpful, as it deals with data formats that can be processed by multiple languages.
- Avro is a preferred tool to serialize data in Hadoop.
- Avro has a schema-based system.
- A language-independent schema is associated with its read and write operations.
- Avro serializes the data which has a built-in schema.
- Avro serializes the data into a compact binary format, which can be deserialized by any application.
- Avro uses JSON format to declare the data structures.
- Presently, it supports languages such as Java, C, C++, C#, Python, and Ruby.

Security in Hadoop :

- Apache Hadoop achieves security by using Kerberos.
- At a high level, there are three steps that a client must take to access a service when using Kerberos.
- Thus, each of which involves a message exchange with a server.
- **Authentication** – The client authenticates itself to the authentication server. Then, receives a timestamped *Ticket-Granting Ticket (TGT)*.
- **Authorization** – The client uses the TGT to request a service ticket from the Ticket Granting Server.
- **Service Request** – The client uses the service ticket to authenticate itself to the server.

Administering Hadoop :

- The person who administers Hadoop is called HADOOP ADMINISTRATOR.
- Some of the common administering tasks in Hadoop are :
- Monitor health of a cluster
- Add new data nodes as needed
- Optionally turn on security
- Optionally turn on encryption
- Recommended, but optional, to turn on high availability
- Optional to turn on MapReduce Job History Tracking Server
- Fix corrupt data blocks when necessary
- Tune performance