**UNIT – II**

**Hadoop Distributed File System**

**TOPICS:** The Design of HDFS, HDFS Concepts, Command Line Interface, Hadoop file system interfaces, Data flow, Data Ingestion with Sqoop and Hadoop archives, Hadoop I/O: Compression, Serialization, Avro and File-Based Data structures.

**THE Design of HDFS:**

**HDFS:**  
HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.

**Where to use HDFS**  
1.**Very Large Files:** Files should be of hundreds of megabytes, gigabytes or more.  
2.**Streaming Data Access:**The time to read whole data set is more important than latency in reading the first. HDFS is built on write-once and read-many-times pattern  
3.**Commodity Hardware:**It works on low cost hardware.

**Where not to use HDFS**

**1.Low Latency data access:** Applications that require very less time to access the first data should not use HDFS as it is giving importance to whole data rather than time to fetch the first record.

**2.Lots Of Small Files:**The name node contains the metadata of files in memory and if the files are small in size it takes a lot of memory for name node's memory which is not feasible.

**3.Multiple Writes:**It should not be used when we have to write multiple times.

**HDFS ARCHITECTURE**



**HDFS Concepts**

* Blocks
* Name Node
* Data Node
* Secondary Name Node
* Block Caching
* Rack Awareness
* HDFS High Availability
* HDFS Federation

**Blocks:**

* 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.
* HDFS, too, has the concept of a block, but it is a much larger unit — 128 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.
* Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block’s worth of underlying storage.

**Benefits:**

1. a file can be larger than any single disk in the network.
2. Making the unit of abstraction a block rather than a file simplifies the storage subsystem.
3. blocks fit well with replication for providing fault tolerance and availability.
4. If a block becomes unavailable, a copy can be read from another location in a way that is transparent to the client.
5. A block that is no longer available due to corruption or machine failure can be replicated from its alternative locations to other live machines to bring the replication factor back to the normal level.

**Name Node & Data Node:**

An HDFS cluster has two types of nodes 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, because this information is reconstructed from datanodes when the system starts.
* 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.
* Without the namenode, the filesystem cannot be used. In fact, if the machine running the namenode were obliterated, all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes.
* For this reason, it is important to make the namenode resilient to failure, and Hadoop provides two mechanisms for this.

Secondary Name Node:

* Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large.
* The secondary namenode usually runs on a separate physical machine because it requires plenty of CPU and as much memory as the namenode to perform the merge.
* It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing.
* However, the state of the secondary namenode lags that of the primary, so in the event of total failure of the primary, data loss is almost certain.
* The usual course of action in this case is to copy the namenode’s metadata files that are on NFS to the secondary and run it as the new primary.

Block Caching:

* A datanode reads blocks from disk, but for frequently accessed files the blocks may be explicitly cached in the datanode’s memory, in an off-heap *block cache*.
* By default, a block is cached in only one datanode’s memory, although the number is configurable on a per-file basis.
* Job schedulers (for MapReduce, Spark, and other frameworks) can take advantage of cached blocks by running tasks on the datanode where a block is cached, for increased read performance.

Hdfs Federation

* The namenode keeps a reference to every file and block in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling (see How Much Memory Does a Namenode Need?).
* HDFS federation, introduced in the 2.x 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.
* Block pool storage is not partitioned, however, so datanodes register with each namenode in the cluster and store blocks from multiple block pools.

HDFS High Availability:

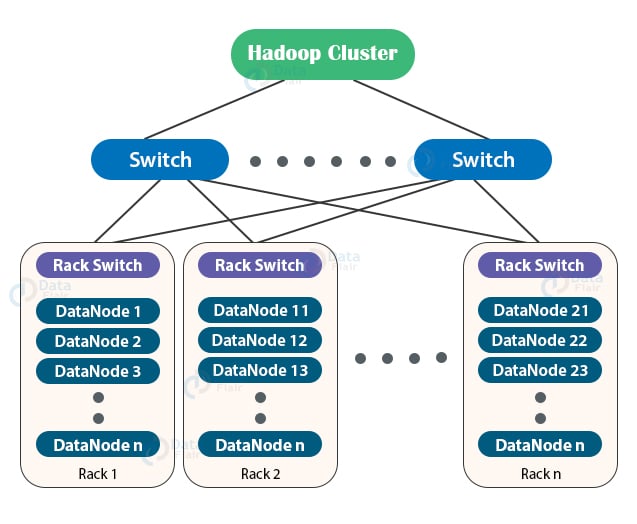
* The combination of replicating namenode metadata on multiple filesystems and using the secondary namenode to create checkpoints protects against data loss, but it does not provide high availability of the filesystem.
* The namenode is still a *single point of failure* (SPOF).
* If it did fail, all 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
  + (iii)received enough block reports from the datanodes to leave safe mode.
* On large clusters with many files and blocks, the time it takes for a namenode to start from cold can be 30 minutes or more.
* The long recovery time is a problem for routine maintenance, too.
* In fact, because unexpected failure of the namenode is so rare, the case for planned downtime is actually more important in practice.

Hadoop 2:

* Hadoop 2 remedied this situation by adding support for HDFS high availability (HA).
* In this implementation, there are a pair of namenodes in an active-standby configuration.
* In the event of the failure of the active namenode, the standby takes over its duties to continue servicing client requests without a significant interruption.
* A few architectural changes are needed to allow this to happen:
  + The namenodes must use highly available shared storage to share the edit log.
  + When a standby namenode comes up, it reads up to the end of the shared edit log to synchronize its state with the active namenode, and then continues to read new entries as they are written by the active namenode.
  + Datanodes must send block reports to both namenodes because the block mappings are stored in a namenode’s memory, and not on disk.
  + Clients must be configured to handle namenode failover, using a mechanism that is transparent to users.
  + The secondary namenode’s role is subsumed by the standby, which takes periodic checkpoints of the active namenode’s namespace.
* If the active namenode fails, the standby can take over very quickly (in a few tens of seconds) because it has the latest state available in memory: both the latest edit log entries and an up-to-date block mapping.
* The actual observed failover time will be longer in practice (around a minute or so), because the system needs to be conservative in deciding that the active namenode has failed.

Rack Awareness:

* HDFS stores files across multiple nodes (DataNodes) in a cluster. To get the maximum performance from Hadoop and to improve the network traffic during file read/write, NameNode chooses the DataNodes on the same rack or nearby racks for data read/write.
* Rack awareness is the concept of choosing the closer DataNode based on rack information.
* The **Rack** is the collection of around 40-50 DataNodes connected using the same network switch. If the network goes down, the whole rack will be unavailable.
* A large Hadoop cluster is deployed in multiple racks.



In a large Hadoop cluster, there are multiple racks. Each rack consists of DataNodes. Communication between the DataNodes on the same rack is more efficient as compared to the communication between DataNodes residing on different racks.

To reduce the network traffic during file [**read/write**](https://data-flair.training/blogs/hadoop-hdfs-data-read-and-write-operations/)**,** NameNode chooses the closest DataNode for serving the client read/write request. NameNode maintains **rack ids** of each DataNode to achieve this rack information. This concept of choosing the closest DataNode based on the rack information is known as **Rack Awareness**.

The reasons for the Rack Awareness in Hadoop are:

1. To reduce the network traffic while file read/write, which improves the cluster performance.
2. To achieve [fault tolerance](https://data-flair.training/blogs/learn-hadoop-hdfs-fault-tolerance/), even when the rack goes down (discussed later in this article).
3. Achieve high availability of data so that data is available even in unfavourable conditions.
4. To reduce the latency, that is, to make the file read/write operations done with lower delay.

NameNode uses a rack awareness algorithm while placing the replicas in HDFS.

### Advantages of Implementing Rack Awareness:

#### Preventing data loss against rack failure

#### Minimize the cost of write and maximize the read speed

#### Maximize network bandwidth and low latency

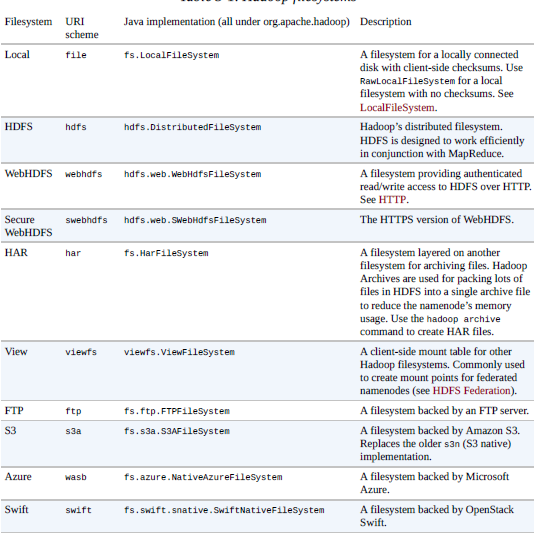
**COMMAND LINE INTERFACE**

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, such as Local FS, HFTP FS, S3 FS, and others.

* 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.**

**Hadoop file system interfaces:**

* Hadoop has an abstract notion of filesystems, of which HDFS is just one implementation.
* The Java abstract class org.apache.hadoop.fs.FileSystem represents the client interface to a filesystem in Hadoop, and there are several concrete implementations.



* Hadoop provides many interfaces to its filesystems, and it generally uses the URI scheme to pick the correct filesystem instance to communicate with.
* For example, the filesystem shell that we met in the previous section operates with all Hadoop filesystems.
* To list the files in the root directory of the local filesystem, type:

% **hadoop fs -ls file:///**

* Although it is possible (and sometimes very convenient) to run MapReduce programs that access any of these filesystems, when you are processing large volumes of data you should choose a distributed filesystem that has the data locality optimization, notably HDFS (see Scaling Out).

**Interfaces:**

* the Java API. The filesystem shell, for example, is a Java application that uses the Java FileSystem class to provide filesystem operations.
* These interfaces are most commonly used with HDFS, since the other filesystems in Hadoop typically have existing tools to access the underlying filesystem (FTP clients for FTP, S3 tools for S3, etc.), but many of them will work with any Hadoop filesystem.

**HTTP:**

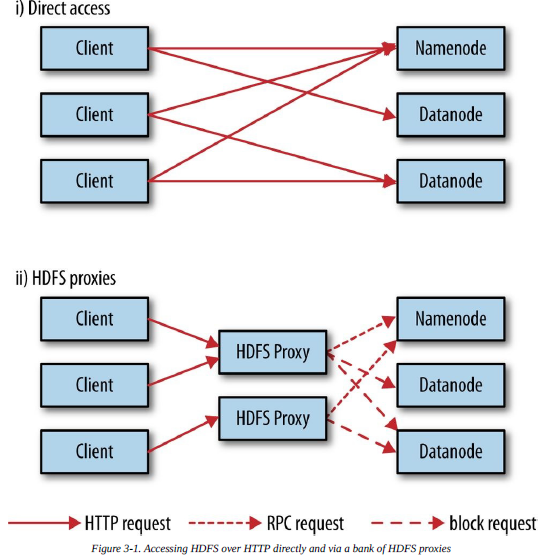
* There are two ways of accessing HDFS over HTTP: directly, where the HDFS daemons serve HTTP requests to clients; and via a proxy (or proxies), which accesses HDFS on the client’s behalf using the usual DistributedFileSystem API. Both use the WebHDFS protocol.

**First Case:**

* the embedded web servers in the namenode and datanodes act as WebHDFS endpoints. (WebHDFS is enabled by default, since dfs.webhdfs.enabled is set to true.)
* File metadata operations are handled by the namenode, while file read (and write) operations are sent first to the namenode, which sends an HTTP redirect to the client indicating the datanode to stream file data from

**Second Case:**

* HDFS over HTTP relies on one or more standalone proxy servers. (The proxies are stateless, so they can run behind a standard load balancer.) All traffic to the cluster passes through the proxy, so the client never accesses the namenode or datanode directly.
* This allows for stricter firewall and bandwidth-limiting policies to be put in place. It’s common to use a proxy for transfers between Hadoop clusters located in different data centers, or when accessing a Hadoop cluster running in the cloud from an external network.



**Other File System Interfaces:**

**C:**

* Hadoop provides a C library called *libhdfs* that mirrors the Java FileSystem interface (it was written as a C library for accessing HDFS, but despite its name it can be used to access any Hadoop filesystem).
* It works using the *Java Native Interface* (JNI) to call a Java filesystem client. There is also a *libwebhdfs* library that uses the WebHDFS interface

**NFS:**

* To mount HDFS on a local client’s filesystem using Hadoop’s NFSv3 gateway.

**FUSE:**

* *Filesystem in Userspace* (FUSE) allows filesystems that are implemented in user space to be integrated as Unix filesystems.
* Hadoop’s Fuse-DFS contrib module allows HDFS (or any Hadoop filesystem) to be mounted as a standard local filesystem.

**Data flow**

**Anatomy of a File Read:**

How the data flows between the client interacting with HDFS, the namenode,

and the datanodes. which shows the main sequence of events when reading a file.



**Step1:**

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

**Step2:**

* DistributedFileSystem calls the namenode, using remote procedure calls RPCs), to determine the locations of the first few blocks in the file.

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 client (according to the topology of the cluster’s network; see Network Topology and Hadoop).

If the client is itself a datanode (in the case of a MapReduce task, for instance), the client will read from the local datanode if that datanode hosts a copy of the block.

The DistributedFileSystem returns an FSDataInputStream (an input stream that

supports file seeks) to the client for it to read data from. FSDataInputStream in turn wraps a DFSInputStream, which manages the datanode and namenode I/O.

**Step3:**

* The client then calls read() on the stream.
* 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.

**Step4:**

* Data is streamed from the datanode back to the client, which calls read() repeatedly on the stream.

**Step5:**

* 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.

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.

**Step6:**

* When the client has finished reading, it calls close() on the FSDataInputStream.

During reading, if the DFSInputStream encounters an error while communicating with a datanode, it will try the next closest one for that block.

It will also remember datanodes that have failed so that it doesn’t needlessly retry them for later blocks.

The DFSInputStream also verifies checksums for the data transferred to it from the datanode.

If a corrupted block is found, the DFSInputStream attempts to read a replica of the block from another datanode; it also reports the corrupted block to the namenode.

One important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block.

**Anatomy of a File Write**

****

**Step1:**

* The client creates the file by calling create() on DistributedFileSystem

**Step2:**

* DistributedFileSystem makes an RPC call to the namenode to create a new file in the filesystem’s namespace, with no blocks associated with it.

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 istributedFileSystem returns an FSDataOutputStream for the client to start writing data to.

Just as in the read case, FSDataOutputStream wraps a DFSOutputStream, which handles communication with the datanodes and namenode.

**Step3:**

* As the client writes data the DFSOutputStream splits it into packets, which it writes to an internal queue called the *data queue.*
* The data queue is consumed by the DataStreamer, which is responsible for asking the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas.
* The list of datanodes forms a pipeline, and here 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 each packet and forwards it to the second datanode in the pipeline.

**Step4:**

* the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline.
* The DFSOutputStream also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the *ack queue*.

**Step5:**

* A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline.
* If any datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data.
* First, the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets.
* The current block on the good datanodes is given a new identity, which is communicated to the namenode, so that the partial block on the failed datanode will be deleted if the failed datanode recovers later on.
* The failed datanode is removed from the pipeline, and a new pipeline is constructed from the two good datanodes.
* The remainder of the block’s data is written to the good datanodes in the pipeline.
* The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node.
* Subsequent blocks are then treated as normal.

**Step6:**

* When the client has finished writing data, it calls close() on the stream.

**Step7:**

* This action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments before contacting the namenode to signal that the file is complete.

**Data Ingestion**

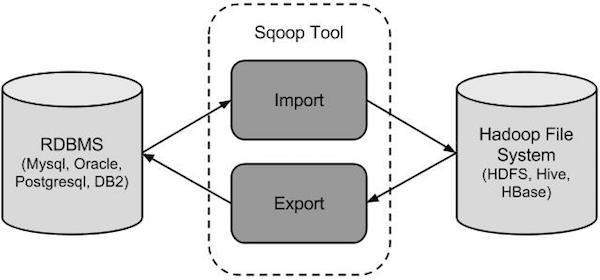
Hadoop Data ingestion is the beginning of your data pipeline in a data lake. It means taking data from various silo databases and files and putting it into Hadoop.

**Sqoop**

* Apache Sqoop(which is a portmanteau for “sql-to-hadoop”) is an open source tool that allows users to extract data from a structured data store into Hadoop for further processing. This processing can be done with MapReduce programs or other higher-level tools such as Hive, Pig or Spark.
* Sqoop can automatically create Hive tables from imported data from a RDBMS (Relational Database Management System) table.
* Sqoop can also be used to send data from Hadoop to a relational database, useful for sending results processed in Hadoop to an operational transaction processing system.

**Sqoop includes tools for the following operations:**

* Listing databases and tables on a database system
* Importing a single table from a database system, including specifying which columns to import and specifying which rows to import using a WHERE clause
* Importing data from one or more tables using a SELECT statement
* Incremental imports from a table on a database system (importing only what has changed since a known previous state)
* Exporting of data from HDFS to a table on a remote database system



Sqoop Import

The import tool imports individual tables from RDBMS to HDFS. Each row in a table is treated as a record in HDFS. All records are stored as text data in text files or as binary data in Avro and Sequence files.

Sqoop Export

The export tool exports a set of files from HDFS back to an RDBMS. The files given as input to Sqoop contain records, which are called as rows in table. Those are read and parsed into a set of records and delimited with user-specified delimiter.

**Hadoop I/O:**

**Data Integrity:**

* Users of Hadoop rightly expect that no data will be lost or corrupted during storage or processing.
* every I/O operation on the disk or network carries with it a small chance of introducing errors into the data that it is reading or writing, when the volumes of data flowing through the system are as large as the ones Hadoop is capable of handling, the chance of data corruption occurring is high.
* The usual way of detecting corrupted data is by computing a *checksum* for the data when it first enters the system, and again whenever it is transmitted across a channel that is unreliable and hence capable of corrupting the data.
* A commonly used error-detecting code is CRC-32 (32-bit cyclic redundancy check), which computes a 32-bit integer checksum for input of any size.
* CRC-32 is used for checksumming in Hadoop’s ChecksumFileSystem, while HDFS uses a more efficient variant called CRC-32C.

**Data Integrity in HDFS:**

* HDFS transparently checksums all data written to it and by default verifies checksums when reading data.
* A separate checksum is created for every dfs.bytes-per-checksum bytes of data. The default is 512 bytes, and because a CRC-32C checksum is 4 bytes long, the storage overhead is less than 1%.
* Datanodes are responsible for verifying the data they receive before storing the data and its checksum.
* When clients read data from datanodes, they verify checksums as well, comparing them with the ones stored at the datanodes.
* Each datanode keeps a persistent log of checksum verifications, so it knows the last time each of its blocks was verified.
* It is possible to disable verification of checksums by passing false to the
* setVerifyChecksum() method on FileSystem before using the open() method to read a file.
* The same effect is possible from the shell by using the -ignoreCrc option with the -get or the equivalent -copyToLocal command

**LocalFilesystem:**

* The Hadoop LocalFileSystem performs client-side checksumming. This means that when you write a file called *filename*, the filesystem client transparently creates a hidden file, *.filename.crc*, in the same directory containing the checksums for each chunk of the file.
* The chunk size is controlled by the file.bytes-per-checksum property, which defaults to 512 bytes.
* The chunk size is stored as metadata in the *.crc* file, so the file can be read back correctly even if the setting for the chunk size has changed.
* Checksums are verified when the file is read, and if an error is detected, LocalFileSystem throws a ChecksumException.
* This is accomplished by using RawLocalFileSystem in place of LocalFileSystem.
* To do this globally in an application, it suffices to remap the implementation for file URIs by setting the property fs.file.impl to the value org.apache.hadoop.fs.RawLocalFileSystem.
* Alternatively, you can directly create a RawLocalFileSystem instance, which may be useful if you want to disable checksum verification for only some reads.

for example:

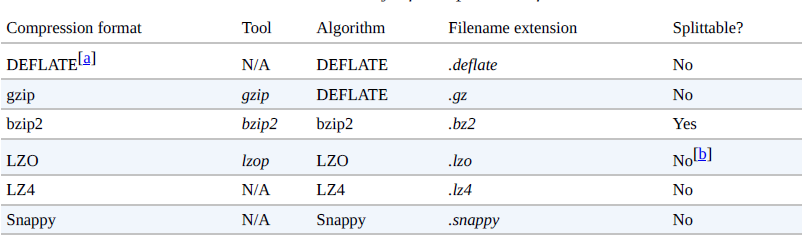
* + Configuration conf = ...
  + FileSystem fs = **new** RawLocalFileSystem();
  + fs.initialize(**null**, conf);

**ChecksumFileSystem:**

* LocalFileSystem uses ChecksumFileSystem to do its work, and this class makes it easy to add checksumming to other (nonchecksummed) filesystems, as ChecksumFileSystem is just a wrapper around FileSystem. The general idiom is as follows:
  + FileSystem rawFs = ...
  + FileSystem checksummedFs = **new** ChecksumFileSystem(rawFs);
* The underlying filesystem is called the *raw* filesystem, and may be retrieved using the getRawFileSystem() method on ChecksumFileSystem.
* ChecksumFileSystem has a few more useful methods for working with checksums, such as getChecksumFile() for getting the path of a checksum file for any file. Check the documentation for the others.
* If an error is detected by ChecksumFileSystem when reading a file, it will call its reportChecksumFailure() method.

**Compression:**

* File compression brings two major benefits:
  + it reduces the space needed to store files.
  + it speeds up data transfer across the network or to or from disk.
* When dealing with large volumes of data, both of these savings can be significant, so it pays to carefully consider how to use compression in Hadoop.
* There are many different compression formats, tools, and algorithms, each with different characteristics.



All compression algorithms exhibit a space/time trade-off:

* faster compression and decompression speeds usually come at the expense of smaller space savings.
* different options: –1 means optimize for speed, and -9 means optimize for space.
* For example, the following command creates a compressed file *file.gz* using the fastest compression method:
* % gzip -1 file
* gzip is a general purpose compressor and sits in the middle of the space/time trade-off.
* bzip2 compresses more effectively than gzip, but is slower. bzip2’s decompression speed is faster than its compression speed, but it is still slower than the other formats.
* LZO, LZ4, and Snappy, on the other hand, all optimize for speed and are around an order of magnitude faster than gzip.
* but compress less effectively. Snappy and LZ4 are also significantly faster than LZO for decompression.

**Codecs:**

* A *codec* is the implementation of a compression-decompression algorithm. In Hadoop, a codec is represented by an implementation of the CompressionCodec interface.
* GzipCodec encapsulates the compression and decompression algorithm for gzip.

**Compression format Hadoop CompressionCodec**

DEFLATE org.apache.hadoop.io.compress.DefaultCodec

gzip org.apache.hadoop.io.compress.GzipCodec

bzip2 org.apache.hadoop.io.compress.BZip2Codec

LZO com.hadoop.compression.lzo.LzopCodec

LZ4 org.apache.hadoop.io.compress.Lz4Codec

Snappy org.apache.hadoop.io.compress.SnappyCodec

**Hadoop compression codecs**

* The LZO libraries are GPL licensed and may not be included in Apache distributions, so for this reason the Hadoop codecs must be downloaded separately from Google.
* Compressing and decompressing streams with CompressionCodec CompressionCodec has two methods that allow you to easily compress or decompress data.
  + createOutputStream(OutputStream out)
  + CompressionOutputStream
  + createInputStream(InputStream in)
  + CompressionInputStream

**Example:**

*A program to compress data read from standard input and write it to standard output*

public class StreamCompressor {

public static void main(String[] args) throws Exception {

String codecClassname = args[0];

Class<?> codecClass = Class.forName(codecClassname);

Configuration conf = new Configuration();

CompressionCodec codec = (CompressionCodec)

ReflectionUtils.newInstance(codecClass, conf);

CompressionOutputStream out = codec.createOutputStream(System.out);

IOUtils.copyBytes(System.in, out, 4096, false);

out.finish();

}

}

**Command:**

“Text” using the StreamCompressor program with the GzipCodec, then decompresses it from standard input using *gunzip*:

% echo "Text" | hadoop StreamCompressor org.apache.hadoop.io.compress.GzipCodec \

| gunzip -Text Inferring CompressionCodecs using CompressionCodecFactory

% hadoop FileDecompressor file.gz

**Compression and Input Splits:**

* When considering how to compress data that will be processed by MapReduce, it is important to understand whether the compression format supports splitting.
* Consider an uncompressed file stored in HDFS whose size is 1 GB.
* With an HDFS block size of 128 MB, the file will be stored as eight blocks,.
* a MapReduce job using this file as input will create eight input splits.
* Each processed independently as input to a separate map task.

**WHICH COMPRESSION FORMAT SHOULD I USE?**

* Hadoop applications process large datasets, so you should strive to take advantage of compression.
* Which compression format you use depends on such considerations as file size, format, and the tools you are using for processing.
* Here are some suggestions, arranged roughly in order of most to least effective:
* Use a container file format such as sequence files (see the section), Avro datafiles (see the section), ORCFiles (see the section), or Parquet files (see the section), all of which support both compression and splitting. A fast compressor such as LZO, LZ4, or Snappy is generally a good choice.
* Use a compression format that supports splitting, such as bzip2 (although bzip2 is fairly slow), or one that can be indexed to support splitting, such as LZO.
* Split the file into chunks in the application, and compress each chunk separately using any supported compression format (it doesn’t matter whether it is splittable). In this case, you should choose the chunk size so that the compressed chunks are approximately the size of an HDFS block.
* Store the files uncompressed.
* For large files, you should not use a compression format that does not support splitting on the whole file, because you lose locality and make MapReduce applications very inefficient.

**Hadoop I/O Serialization**

* *Serialization* is the process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage.
* *Deserialization* is the reverse process of turning a byte stream back into a series of structured objects.
* Serialization is used in two quite distinct areas of distributed data processing: for interprocess communication and for persistent storage.
* In Hadoop, interprocess communication between nodes in the system is implemented using *remote procedure calls* (RPCs).
* The RPC protocol uses serialization to render the message into a binary stream to be sent to the remote node, which then deserializes the binary stream into the original message.

**Serialization for Storage Formats:**

In general, it is desirable that an RPC serialization format is:

**Compact:** A compact format makes the best use of network bandwidth, which is the most scarce resource in a data centre.

**Fast:** Interprocess communication forms the backbone for a distributed system, so it is essential that there is as little performance overhead as possible for the serialization and deserialization process.

**Extensible:** Protocols change over time to meet new requirements, so it should be straightforward to evolve the protocol in a controlled manner for clients and servers.

For example, it should be possible to add a new argument to a method call and have the new servers accept messages in the old format (without the new argument) from old clients.

**Interoperable:** For some systems, it is desirable to be able to support clients that are written in different languages to the server, so the format needs to be designed to make this possible.

**Serialization is used in two quite distinct areas of distributed data processing:**

* Interprocess communication

When a client calls a function or subroutine from one pc to the pc in-network or server, that calling is a remote procedure call.

* Persistent storage

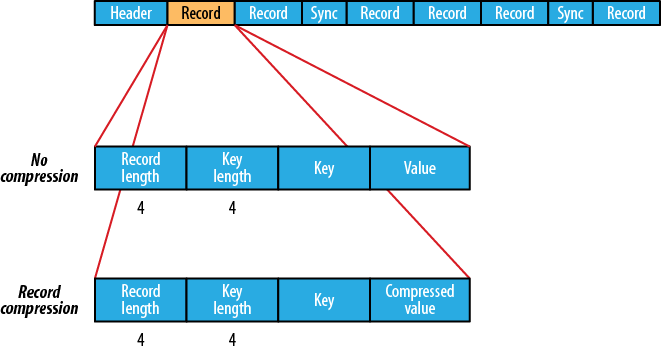
It is better than java's inbuilt serialization as java serialization isn't compact. Serialization and Deserialization of data help maintain and manage corporate decisions for effective use of resources and data available in Data warehouse or any other database -writable - language specific to java.

File-Based Datastructures:

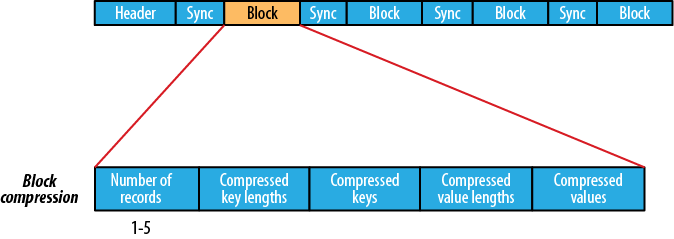
a specialized data structure to hold your data. For doing MapReduce-based processing, putting each blob of binary data into its own file doesn’t scale, so Hadoop developed a number of higher-level containers for these situations.

Sequence File:

* Imagine a logfile where each log record is a new line of text.
* If you want to log binary types, plain text isn’t a suitable format.
* Hadoop’s SequenceFile class fits the bill in this situation, providing a persistent data structure for binary key-value pairs.
* To use it as a logfile format, you would choose a key, such as timestamp represented by a LongWritable, and the value would be a Writable that represents the quantity being logged.
* SequenceFiles also work well as containers for smaller files. HDFS and MapReduce are optimized for large files, so packing files into a SequenceFile makes storing and processing the smaller files more efficient (Processing a whole file as a record contains a program to pack files into a SequenceFile).
* To create a SequenceFile, use one of its createWriter() static methods, which return a SequenceFile.Writer instance.
* Reading sequence files from beginning to end is a matter of creating an instance of SequenceFile.
* The hadoop fs command has a -text option to display sequence files in textual form.
* A sequence file consists of a header followed by one or more records.
* If no compression is enabled (the default), each record is made up of the record length (in bytes), the key length, the key, and then the value.



* The format for record compression is almost identical to that for no compression, except the value bytes are compressed using the codec defined in the header. Note that keys are not compressed.



* Block compression compresses multiple records at once; it is therefore more compact than and should generally be preferred over record compression because it has the opportunity to take advantage of similarities between records.

**MapFile:**

* A MapFile is a sorted SequenceFile with an index to permit lookups by key.
* The index is itself a SequenceFile that contains a fraction of the keys in the map (every 128th key, by default).
* The idea is that the index can be loaded into memory to provide fast lookups from the main data file, which is another SequenceFile containing all the map entries in sorted key order.
* MapFile offers a very similar interface to SequenceFile for reading and writing — the main thing to be aware of is that when writing using MapFile.Writer, map entries must be added in order, otherwise an IOException will be thrown.

**MapFile variants:**

Hadoop comes with a few variants on the general key-value MapFile interface:

* SetFile is a specialization of MapFile for storing a set of Writable keys. The keys must be added in sorted order.
* ArrayFile is a MapFile where the key is an integer representing the index of the element in the array and the value is a Writable value.
* BloomMapFile is a MapFile that offers a fast version of the get() method, especially for sparsely populated files.
* The implementation uses a dynamic Bloom filter for testing whether a given key is in the map.
* The test is very fast because it is in-memory, and it has a nonzero probability of false positives.
* Only if the test passes (the key is present) is the regular get() method called.

What is Avro?

Avro files are classified in to two types:

1.Data Serialization.

2. Data Exchange.

Avro files are language natural:

Hadoop Cluster

Avro file

Write Data In Java

Read Data in C++

Avro Serialization: Program can efficiently serialize data into files.

010101010101010101001

Hadoop

Bigdata

Sqoop

Hive

Pig

hbase

Avro Serialize

Message

01010101001010101001010

Data:100MB Binary Data (10 Mb)

Disk

Data storage is compact & efficient

Binary Data

100 Mb --------------------------🡪 10 Mb

Avro stores both the data definition and data together in one message or file.

Avro Stroes data and its definition(schema) together. Which helps program to understand the information stored is an file or message.

Data Definition ===🡺 Schema == Json

Data = Binary compact format data

Avro files include markers that can be used to splitting large datasets into subset (HDFS) blocks which it suitable for MapReduce processing.

Schema Evaluation:

It’s a key feature of Avro in this case schema can change over time for instance

* + - Adding a field
    - Removing a field

Still there is no problem for reading and writing data with changed schema or old schema.

Avro cleanly handles schema changes like:

* + - Missing fields
    - Newley added fields
    - Changed fields

1. Old program can read data [Based on New schema]
2. New program can read data [Based on old schema]
3. Apache Avro is a framework that allows you to serialize data in a format that has schema built in.
4. It serialize data in a compact binary format that does not require proxy objects or code generation.
5. Instead of using generated proxy libraries and string typing, Avro relies heavily on the schemas that are sent along with the serialize data.
6. Including schemas with Avro message allows any application to deserialize the data.
7. Avro schemas describe the format of the message and are defined using JavaScript object notation (JSON)