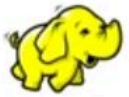
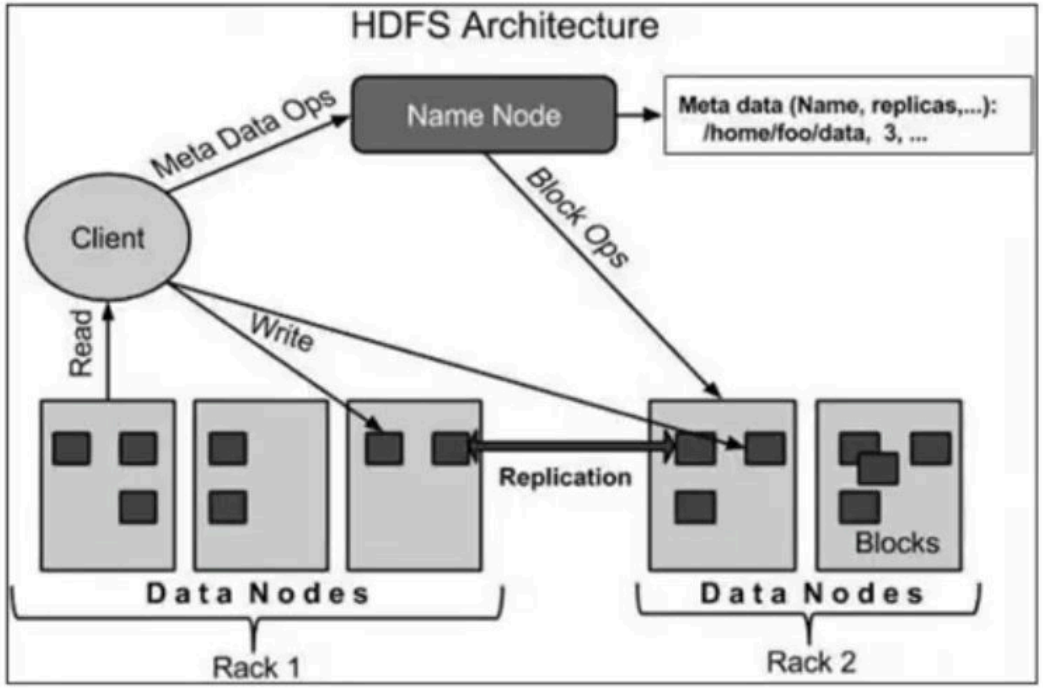
Design of HDFS 

Hadoop File System was developed using distributed file system design. It is run on commodity hardware. Unlike other distributed systems, HDFS is highly faulttolerant and designed using low-cost hardware.
HDFS holds very large amount of data and provides easier access. To store such huge data, the files are stored across multiple machines. These files are stored in redundant fashion to rescue the system from possible data losses in case of failure. HDFS also makes applications available to parallel processing.

Features of HDFS:

1. It is suitable for the distributed storage and processing.
2. Hadoop provides a command interface to interact with HDFS.
3. The built-in servers of namenode and datanode help users to easily check the status of cluster.
4. Streaming access to file system data.
5. HDFS provides file permissions and authentication.

HDFS Architecture



HDFS Concepts

Name Node: HDFS works in master-worker pattern where the name node acts as master. Name Node is controller and manager of HDFS as it knows the status and the metadata of all the files in HDFS; the metadata information being file permission, names and location of each block. The metadata are small, so it is stored in the memory of name node, allowing faster access to data.

Data Node: They store and retrieve blocks when they are told to; by client or name node. They report back to name node periodically, with list of blocks that they are storing. The data node being a commodity hardware also does the work of block creation, deletion and replication as stated by the name node.

Blocks: A Block is the minimum amount of data that it can read or write. HDFS blocks are 128 MB by default and this is configurable. Ff the file is in HDFS is smaller than block size, then it does not occupy full block? size, i.e. 5 MB of file stored in HDFS of block size 128 MB takes 5MB of space only.

Benefits and Challenges

Benefits

1. **High fault tolerance.**
2. **Very Large Files:** Files should be of hundreds of megabytes, gigabytes or more.
3. **Commodity Hardware:**It works on low cost hardware.
4. **Streaming Data Access**
5. **Fast**
6. **Flexible**

Challenges

1. The Difficulty in Finding Root Cause of Problems.
2. Inefficient Cluster Utilization.
3. The business impact of Hadoop inefficiencies.
4. No real-time data processing
5. Issue with small files

File size

- HDFS supports large files and large numbers of files.
- A typical HDFS cluster has lens of millions of files.
- A typical file is 100 MB or larger.
- The NameNode maintains the namespace metadata of the file such as the filename, directory name, user, permissions etc.
- A file is considered small if its size is much less than the block size. For example, if the block size is 128MB and the file size is1MB to 50MB, the file is considered a small file.

Block size

- HDFS stores files into fixed-size blocks.
- HDFS data blocks size is 128MB.
- Blocks size can be configured as per our requirements.
- Hadoop distributes these blocks on different slave machines, and the master machine stores the metadata about blocks location.

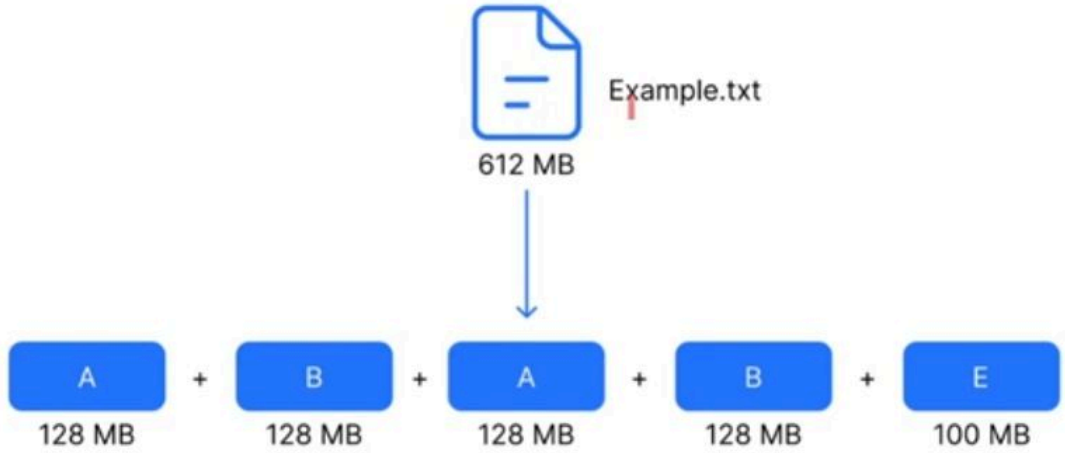
Why are blocks in HDFS huge?

1. To minimize the cost of seek: For the large size blocks, time taken to transfer the data from disk can be longer as compared to the time taken to start the block.
2. If blocks are small, there will be too many blocks in Hadoop HDFS and thus too much metadata to store. Managing such a huge number of blocks and metadata will create overhead and lead to traffic in a network.

Conclusion:

We can conclude that the HDFS data blocks are blocked-sized chunks having size 128 MB by default. We can configure this size as per our requirements. The files smaller than the block size do not occupy the full block size. The size of HDFS data blocks is large in order to reduce the cost of seek and network traffic.

Data block in 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.

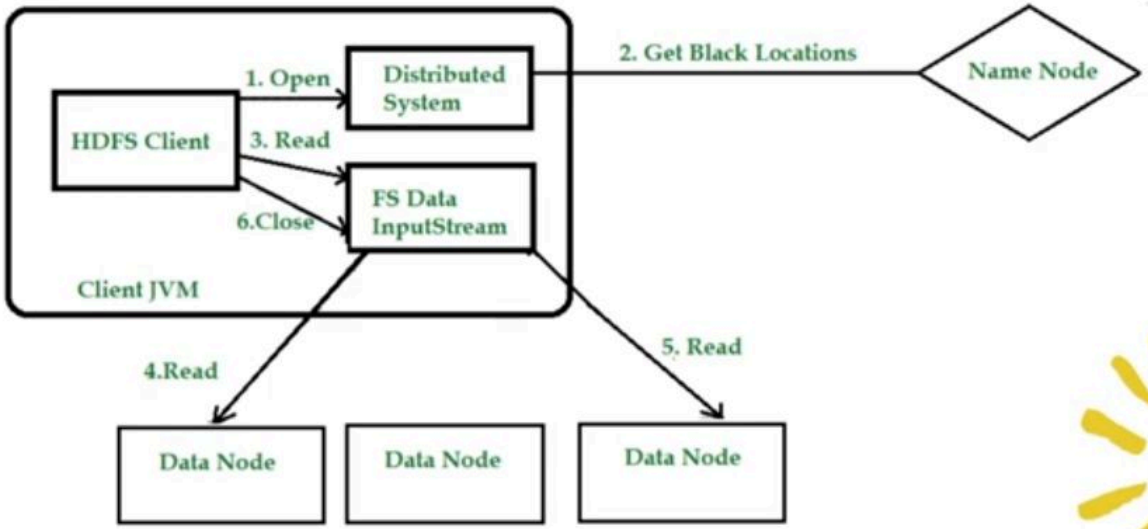
Data replication

Why is replication done in HDFS?

Replication in HDFS increases the availability of Data at any point of time. If any node containing a block of data which is used for processing crashes, we can get the same block of data from another node this is because of replication.

- Replication ensures the availability of the data.
- 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.
- Replication is one of the major factors in making the HDFS a Fault tolerant system.

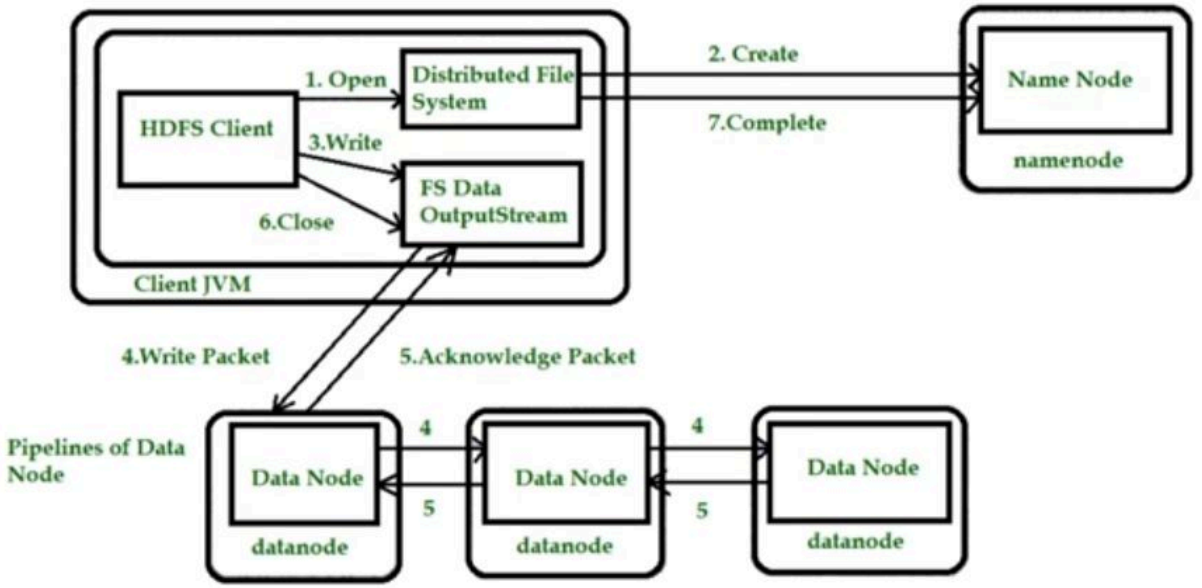
HDFS read files



HDFS 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.
- **Step 3:** The client then calls read() on the stream. DFSInputStream, which has stored the info node addresses for the primary (closest) few blocks within 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.

HDFS write files



HDFS 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:** The client writes data, the DFSOutputStream splits it into packets, which it writes to an indoor queue called the info queue.
- **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.

Scoop vs Flume vs HDFS

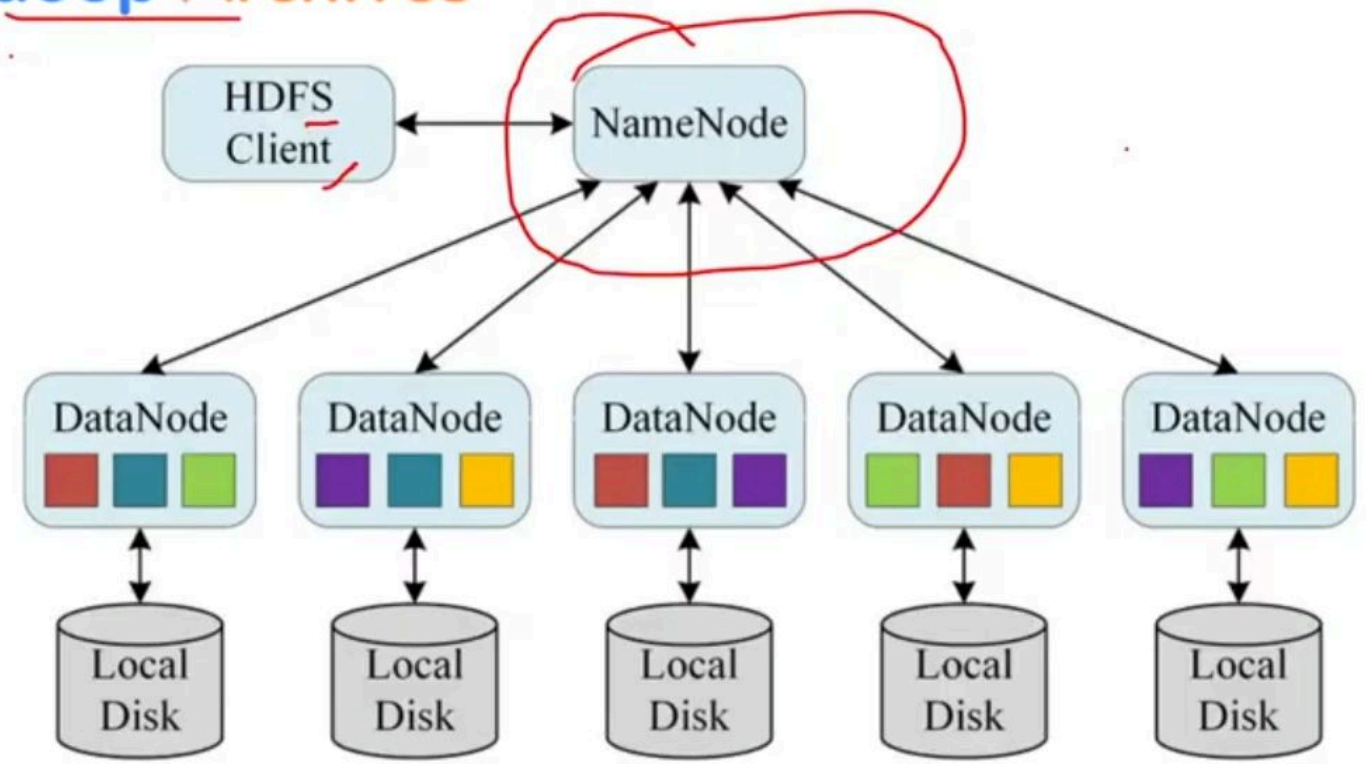
Sqoop	Flume	HDFS
Sqoop is designed for bulk data importing into Hadoop and exporting back to RDBMS.	Flume helps in transferring the streaming data directly into HDFS.	It is a distributed file system used by Hadoop to store the actual data
Sqoop has a connector based architecture and allows connectors to connect with the right data sources to fetch the data.	Flume has an agent-based architecture and a piece of code is written to fetch data.	HDFS comes with a distributed architecture in which the entire data is distributed across the nodes.
Using the Sqoop import command we can import data into HDFS.	Data flows to HDFS using channels.	This is a storage destination.
Here the data load cannot be driven by an event.	Here the data load can be driven by an event.	It is just designed to store the data delivered from multiple sources.
The data importing from structured data sources becomes simple using Sqoop. Because Sqoop connectors know the exact process to connect with data sources and fetches the data	Flume loads the streaming data such as log files of a web server, tweets generated on Twitter, etc.	HDFS consists of in-built commands for data storage.

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.
- 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.

Hadoop Archives



Compression

Data compression at various stages in Hadoop:

1. Compressing input files
2. Compressing the map output
3. Compressing output files

Hadoop compression formats

- **Deflate:** It is the compression algorithm whose implementation is zlib. Defalte compression algorithm is also used by gzip compression tool. Filename extension is .deflate.

- **gzip:** gzip compression is based on Deflate compression algorithm. Gzip compression is not as fast as LZO or snappy but compresses better so space saving is more. Gzip is not splitable. Filename extension is .gz.

- **bzip2:** Using bzip2 for compression will provide higher compression ratio but the compressing and decompressing speed is slow. Bzip2 is splittable. Filename extension is .bz2.

Compression

Hadoop compression formats:

- Snappy: The Snappy compressor from Google provides fast compression and decompression but compression ratio is less. Snappy is not splittable. Extension is .snappy.

- LZO: LZO, just like snappy is optimized for speed so compresses and decompresses faster but compression ratio is less. LZO is not splittable by default but you can index the lzo files as a pre-processing step to make them splittable. Filename extension is .lzo.

Codecs(compressor-decompressor) in Hadoop:

There are different codec classes for different compression formats

- Deflate – org.apache.hadoop.io.compress.DefaultCodec
- Gzip – org.apache.hadoop.io.compress.GzipCodec
- Bzip2 – org.apache.hadoop.io.compress.Bzip2Codec
- Snappy – org.apache.hadoop.io.compress.SnappyCodec
- LZO – com.hadoop.compression.lzo.LzoCodec

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).
- 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 and File-based data structures

- Apache Avro is a language-neutral data serialization system.
- Avro creates binary structured format that is both compressible and splitable. Hence it can be efficiently used as the input to Hadoop MapReduce jobs.
- Avro provides rich data structures. For example, you can create a record that contains an array, an enumerated type, and a sub record.
- Avro is a preferred tool to serialize data in Hadoop.

Features of Avro:

- Avro is a language-neutral data serialization system.
- It can be processed by many languages (currently C, C++, C#, Java, Python, and Ruby).
- Avro creates binary structured format that is both compressible and splitable.
- Avro creates a self-describing file named Avro Data File, in which it stores data along with its schema in the metadata section.
- Avro is also used in Remote Procedure Calls (RPCs). During RPC, client and server exchange schemas in the connection handshake.

Avro and File-based data structures

General Working of Avro:

- **Step 1** – Create schemas. Here you need to design Avro schema according to your data.
- **Step 2** – Read the schemas into your program. It is done in two ways –

  By Generating a Class Corresponding to Schema –

  Compile the schema using Avro. This generates a class file corresponding to the schema.

  By Using Parsers Library –

  You can directly read the schema using parsers library.
- **Step 3** – Serialize the data using the serialization API provided for Avro, which is found in the package org.apache.avro.specific.
- **Step 4** – Deserialize the data using deserialization API provided for Avro, which is found in the package org.apache.avro.specific.