HDFS

Hadoop Distributed File System

Before going into depth of HDFS, let us discuss a problem statement.

If we have 100TB data, How will we design a system to store it? Let's take 2 minutes to find out possible solutions and then we will discuss it.

One possible solution is to build network-attached storage or storage area network. We can buy hundred 1TB hard disks and mount them to hundred subfolders as shown in the image. What will be the challenges in this approach? Let us take 2 minutes to find out challenges and then we will discuss them.

Let us discuss the challenges.

How will we handle failover and backups?

Failover means switching to a redundant or standby hard disk upon the failure of any hard disk. For backup, we can put extra hard disks or build a RAID i.e. redundant array of independent disks for every hard disk in the system but still it will not solve the problem of failover which is really important for real-time applications.

How will we distribute the data uniformly?

Distributing the data uniformly across the hard disks is really important so that no single disk will be overloaded at any point in time.

Is it the best use of available resources?

There may be other small size hard disks available with us but we may not be able to add them to NAS or SAN because huge files can not be stored in these smaller hard disks. Therefore we will need to buy new bigger hard disks.

How will we handle frequent access to files? What if most of the users want to access the files stored in one of the hard disks. File access speed will be really slow in that case and apparently no user will be able to access the file due to congestion.

How will we scale out?

Scaling out means adding new hard disks when we need more storage. When we will add more hard disks, data will not be uniformly distributed as old hard disks will have more data and newly added hard disks will have less or no data.

To solve above problems Hadoop comes with a distributed filesystem called HDFS. We may sometimes see references to "DFS" informally or in older documentation or configurations.

HDFS namenodes and datamodes

HDFS

An HDFS cluster has two types of nodes: one namenode also known as the master and multiple datanodes

An HDFS cluster consists of many machines. One of these machines is designated as namenode and other machines act as datanodes. Please note that we can also have datanode on the machine where namenode service is running. By default, namenode metadata service runs on port 8020

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 in RAM and persisted 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.

Datanodes are the workhorses of the filesystem. While namenode keeps the index of which block is stored in which datanode, datanodes store the actual data. In short, datanodes do not know the name of a file and namenode does not know what is inside a file.

As a rule of thumb, namenode should be installed on the machine having bigger RAM as all the metadata for files and directories is stored in RAM. We will not be able to store many files in HDFS if RAM is not big as there will not be enough space for metadata to fit in RAM. Since data nodes store actual data, datanodes should be run on the machines having the bigger disk.

HDFS - Replication

Let’s understand the concept of blocks in HDFS. When we store a file in HDFS, the file gets split into the chunks of 128MB block size. Except for the last block all other blocks will have 128 MB in size. The last block may be less than or equal to 128MB depending on file size. This default block size is configurable.

Let’s say we want to store a 560MB file in HDFS. This file will get split into 4 blocks of 128 MB and one block of 48 MB

What are the advantages of splitting the file into blocks? It helps fitting big file into smaller disks. It leaves less unused space on the every datanode as many 128MB blocks can be stored on the each datanode. It optimizes the file transfer. Also, it distributes the load to multiple machines. Let’s say a file is stored on 10 data nodes, whenever a user accesses the file, the load gets distributed to 10 machines instead of one machine.

It is same like when we download a movie using torrent. The movie file gets broken down into multiple pieces and these pieces get downloaded from multiple machines parallelly. It helps in downloading the file faster.

Let’s understand the HDFS replication. Each block has multiple copies in HDFS. A big file gets split into multiple blocks and each block gets stored to 3 different data nodes. The

default replication factor is 3. Please note that no two copies will be on the same data node. Generally, first two copies will be on the same rack and the third copy will be off the rack (A rack is an almirah where we stack the machines in the same local area network). It is advised to set replication factor to at least 3 so that even if something happens to the rack, one copy is always safe.

We can set the default replication factor of the file system as well as of each file and directory individually. For files which are not important we can decrease the replication factor and for files which are very important should have high replication factor.

Whenever a datanode goes down or fails, the namenode instructs the datanodes which have copies of lost blocks to start replicating the blocks to the other data nodes so that each file and directory again reaches the replication factor assigned to it.

Design and limitations

HDFS is designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware. Let's understand the design of HDFS

It is designed for very large files. "Very large" in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size.

It is designed for streaming data access. It 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 the source, and 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

It is designed for commodity hardware. Hadoop doesn't require expensive, highly reliable hardware. It's designed to run on the commonly available hardware that can be obtained from multiple vendors. HDFS is designed to carry on working without a noticeable interruption to the user in case of hardware failure.

It is also worth knowing the applications for which HDFS does not work so well.

HDFS does not work well 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. HDFS is optimized for delivering high throughput and this may be at the expense of latency.

HDFS is not a good fit if we have a lot of small files. Because 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

If we have multiple writers and arbitrary file modifications, HDFS will not a good fit. Files in HDFS are modified by a single writer at any time.

Writes are always made at the end of the file, in the append-only fashion.

There is no support for modifications at arbitrary offsets in the file.

File reading and writing

When a user wants to read a file, the client will talk to namenode and namenode will return the metadata of the file. The metadata has information about the blocks and their locations.

When the client receives metadata of the file, it communicates with the datanodes and accesses the data sequentially or parallelly. This way there is no bottleneck in namenode as client talks to namenode only once to get the metadata of the files.

HDFS by design makes sure that no two writers write the same file at the same time by having singular namenode.

If there are multiple namenodes, and clients make requests to these different namenodes, the entire filesystem can get corrupted. This is because these multiple requests can write to a file at the same time.

Let’s understand how files are written to HDFS. When a user uploads a file to HDFS, the client on behalf of the user tells the namenode that it wants to create the file. The namenode replies back with the locations of datanodes where the file can be written. Also, namenode creates a temporary entry in the metadata.

The client then opens the output stream and writes the file to the first datanode. The first datanode is the one which is closest to the client machine. If the client is on a machine which is also a datanode, the first copy will be written on this machine.

Once the file is stored on one datanode, the data gets copied to the other datanodes simultaneously. Also, once the first copy is completely written, the datanode informs the client that the file is created.

The client then confirms to the namenode that the file has been created. The namenode crosschecks this with the datanodes and updates the entry in the metadata successfully.

Now, lets try to understand what happens while reading a file from HDFS.

When a user wants to read a file, the HDFS client, on behalf of the user, talk to the namenode.

The Namenode provides the locations of various blocks of this file and their replicas instead of giving back the actual data.

Out of these locations, the client chooses the datanodes closer to it. The client talks to these datanodes directly and reads the data from these blocks.

The client can read blocks of the file either sequentially or simultaneously.

Namenode Backup & Failover

The metadata is maintained in the memory as well as on the disk. On the disk, it is kept in two parts: namespace image and edit logs.

The namespace image is created on demand while edit logs are created whenever there is a change in the metadata. So, at any point, to get the current state of the metadata, edit logs need to be applied on the image.

Since the metadata is huge, writing it to the disk on every change may be time consuming. There fore, saving just the change makes it extremely fast.

Without the namenode, the HDFS cannot be used at all. This is because we do not know which files are stored in which datanodes. Therefore it is very important to make the namenode resilient to failures. Hadoop provides various approaches to safeguard the namenode.

The first approach is to maintain a copy of the metadata on NFS - Network File System. Hadoop can be configured to do this. These modifications to the metadata happen either both on NFS and Locally or nowhere.

In the second approach to making the namenode resilient, we run a secondary namenode on a different machine.

The main role of the secondary namenode is to periodically merge the namespace image with edit logs to prevent the edit logs from becoming too large.

When a namenode fails, we have to first prepare the latest namespace image and then bring up the secondary namenode.

This approach is not good for production applications as there will be a downtime until the secondary namenode is brought online. With this method, the namenode is not highly available.

To make the namenode resilient, Hadoop 2.0 added support for high availability.

This is done using multiple namenodes and zookeeper. Of these namenodes, one is active and the rest are standby namenodes. The standby namenodes are exact replicas of the active namenode.

The datanodes send block reports to both the active and the standby namenodes to keep all namenodes updated at any point-in-time.

If the active namenode fails, a standby can take over very quickly because it has the latest state of metadata.

zookeeper helps in switching between the active and the standby namenodes.

The namenode maintains the reference to every file and block in the memory. If we have too many files or folders in HDFS, the metadata could be huge. Therefore the memory of namenode may become insufficient.

To solve this problem, HDFS federation was introduced in Hadoop 2 dot 0.

In HDFS Federation, we have multiple namenodes containing parts of metadata.

The metadata - which is the information about files folders - gets distributed manually in different namenodes. This distribution is done by maintaining a mapping between folders and namenodes.

This mapping is known as mount tables.

In this diagram, the mount table is defining /mydata1 folder is in namenode1 and /mydata2 and /mydata3 are in namenode2

Mount table is not a service. It is a file kept along with and referred from the HDFS configuration file.

The client reads mount table to find out which folders belong to which namenode.

It routes the request to read or write a file to the namenode corresponding to the file's parent folder.

The same pool of datanodes is used for storing data from all namenodes.

Lets us discuss what is metadata. Following attributes get stored in metadata

- List of files
- List of Blocks for each file
- List of DataNode for each block
- File attributes, e.g. access time, replication factor, file size, file name, directory name
- Transaction logs or edit logs store file creation and file deletion timestamps.

Uploading Files

To upload files from our local machine to HDFS, we can use Hue. Let’s upload a file from our local machine to HDFS using Hue. Login to Hue, click on file browser. On the top left, you can see your home directory in HDFS. Please note that in HDFS you have permissions to create files and directories only in your home directory. Apart from your home directory, you have read permissions on /data and /dataset directories which contain dataset and code provided by CloudxLab.

Let’s upload a file. Click on “Upload”, select the type of file. In our case, we are going to upload normal text file so we will select “Files”. Click on “Select Files” and let's select the file from the local machine. We’ve successfully uploaded the file from our local machine to HDFS.

We can see the user, owner and permissions of the uploaded file in Hue File browser. HDFS file permissions are same as Unix file permissions.

Copy file from terminal

Create a file test.txt using nano or vi editor. Run command hadoop fs -copyFromLocal test.txt. This command will copy the file from CloudxLab Linux console to CloudxLab HDFS. To verify if file is copied to HDFS, please type hadoop fs -ls test.txt To see content of file in HDFS, please type hadoop fs -cat test.txt

More commands

To access the files in HDFS, we can type any of the commands displayed on the screen.

To see which datanodes have blocks of sample.txt, use the following command:

hdfs fsck -blocks -locations -racks -files sample.txt

Blocks are located on datanodes having private ips 172.31.53.48, 172.31.38.183 and 172.31.37.42

By default, Every file is having a replication factor of 3 on CloudxLab. To change the replication factor of sample.txt to 1, we can run

hadoop fs -setrep -w 1 /user/abhinav9884/sample.txt

Now if we check the blocks, we will see that the Average block replication is 1. If you want to increase your space quota on HDFS, please decrease the replication factor of your home directory in HDFS