HDFS Characteristics

This lesson will explore some important aspects of HDFS architecture.

We'll cover the following

Security and permission#

HDFS provides a permissions model for files and directories which is similar to POSIX. Each file and directory is associated with an **owner** and a **group**. Each file or directory has separate permissions for the owner, other users who are members of a group, and all other users. There are three types of permission:

1. **Read permission** (`r`): For files, `r` permission is required to read a file. For directories, `r` permission is required to list the contents of a directory.
2. **Write permission** (`w`): For files, `w` permission is required to write or append to a file. For a directory, `w` permission is required to create or delete files or directories in it.
3. **Execute permission** (`x`): For files, `x` permission is ignored as we cannot execute a file on HDFS. For a directory, `x` permission is required to access a child of the directory.

HDFS also provides optional support for POSIX ACLs (Access Control Lists) to augment file permissions with finer-grained rules for specific named users or named groups.

HDFS federation#

The NameNode keeps the metadata of the whole namespace in memory, which means that on very large clusters with many files, the memory becomes the limiting factor for scaling. A more serious problem is that a single NameNode, serving all metadata requests, can become a performance bottleneck. To help resolve these issues, HDFS Federation was introduced in the 2.x release, which 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`, and a second NameNode might handle files under `/share`. Under federation:

- All NameNodes work independently. No coordination is required between NameNodes.
- DataNodes are used as the common storage by all the NameNodes.
- A NameNode failure does not affect the availability of the namespaces managed by other NameNodes.
- To access a federated HDFS cluster, clients use client-side mount tables to map file paths to NameNodes.

Multiple NameNodes running independently can end up generating the same **64-bit Block IDs** for their blocks. To avoid this problem, a namespace uses one or more **Block Pools**, where a unique ID identifies each block pool in a cluster. A block pool belongs to a single namespace and does not cross the namespace boundary. The extended block ID, which is a tuple of (Block Pool ID, Block ID), is used for block identification in HDFS Federation.

Erasure coding#

By default, HDFS stores three copies of each block, resulting in a 200% overhead (to store two extra copies) in storage space and other resources (e.g., network bandwidth). Compared to this default replication scheme, Erasure Coding (EC) is probably the biggest change in HDFS in recent years. EC provides the same level of fault tolerance with much less storage space. In a typical EC setup, the storage overhead is no more than 50%. This fundamentally doubles the storage space capacity by bringing down the replication factor from 3x to 1.5x.

Under EC, data is broken down into fragments, expanded, encoded with redundant data pieces, and stored across different DataNodes. If, at some point, data is lost on a DataNode due to corruption, etc., then it can be reconstructed using the other fragments stored on other DataNodes. Although EC is more CPU intensive, it greatly reduces the storage needed for reliably storing a large data set.

For more details on how EC works, see blog (https://blog.cloudera.com/introduction-to-hdfs-erasure-coding-in-apache-hadoop/) or wiki (https://en.wikipedia.org/wiki/Erasure_code).

HDFS in practice#

Although HDFS was primarily designed to support Hadoop MapReduce jobs by providing a DFS for the Map and Reduce operations, HDFS has found many uses with big-data tools.

HDFS is used in several Apache projects that are built on top of the Hadoop framework, including Pig, Hive, HBase, and Giraph. HDFS support is also included in other projects, such as GraphLab.

The primary advantages of HDFS include the following:

- **High bandwidth for MapReduce workloads**: Large Hadoop clusters (thousands of machines) are known to continuously write up to one terabyte per second using HDFS.
- **High reliability**: Fault tolerance is a primary design goal in HDFS. HDFS replication provides high reliability and availability, particularly in large clusters, in which the probability of disk and server failures increases significantly.
- **Low costs per byte**: Compared to a dedicated, shared-disk solution such as a SAN, HDFS costs less per gigabyte because storage is collocated with compute servers. With SAN, we have to pay additional costs for managed infrastructure, such as the disk array enclosure and higher-grade enterprise disks, to manage hardware failures. HDFS is designed to run with commodity hardware, and redundancy is managed in software to tolerate failures.
- **Scalability**: HDFS allows DataNodes to be added to a running cluster and offers tools to manually rebalance the data blocks when cluster nodes are added, which can be done without shutting the file system down.

The primary disadvantages of HDFS include the following:

- **Small file inefficiencies**: HDFS is designed to be used with large block sizes (128MB and larger). It is meant to take large files (hundreds of megabytes, gigabytes, or terabytes) and chunk them into blocks, which can then be fed into MapReduce jobs for parallel processing. HDFS is inefficient when the actual file sizes are small (in the kilobyte range). Having a large number of small files places additional stress on the NameNode, which has to maintain metadata for all the files in the file system. Typically, HDFS users combine many small files into larger ones using techniques such as sequence files. A sequence file can be understood as a container of binary key-value pairs, where the file name is the key, and the file contents are the value.

- **POSIX non-compliance**: HDFS was not designed to be a POSIX-compliant, mountable file system; applications will have to be either written from scratch or modified to use an HDFS client. Workarounds exist that enable HDFS to be mounted using a FUSE (https://en.wikipedia.org/wiki/Filesystem_in_Userspace) driver, but the file system semantics do not allow writes to files once they have been closed.
- **Write-once model**: The write-once model is a potential drawback for applications that require concurrent write accesses to the same file. However, the latest version of HDFS now supports file appends.

In short, HDFS is a good option as a storage backend for distributed applications that follow the MapReduce model or have been specifically written to use HDFS. HDFS can be used efficiently with a small number of large files rather than a large number of small files.

← Back

HDFS High Availability (HA)

Next →

Summary: HDFS

Mark as Completed

Report an Issue