The Hadoop Distributed File System

Persistent Storage on Single Systems

Persistent storage on disks is done using a block-oriented structure, and the file systems govern how data is stored on these blocks.

The desirable properties of these persistently stored files are

1. Long term existence
2. Shareability
3. Structure of the file

The file system provides a means to store data organised as files as well as a collection of functions to interface with these files. This is done by maintaining a set of attributes associated with the file. The main design of a file system is done by separating information into data and metadata, and mapping between these and blocks that store the files is designed on top.

A Network File System (NFS) is designed for a large number of small files, which handles scaling up, but not scaling out. A cluster is required to handle the load, not a single machine.

Persistent Storage on a Cluster - Distributed File Systems

A distributed file system is a file system running on a cluster of machines, and the Hadoop DFS is the open source version of the initial Google File System to handle distributed file systems.

The design of the HDFS is mainly to store very large files with streaming and data access patterns, running on clusters of commodity hardware. The key design principles are

1. Very large files
2. Read Mostly data
    a. Write once, read many times pattern
    b. Time to read the whole dataset is more important than the latency in reading the first record.
3. Commodity Hardware
    a. Off the shelf hardware
    b. Focus on scale out

Specialised hardware provides a large memory, fault tolerance and scaling up, but that is not explicitly required with the HDFS.

HDFS Internals

The file metadata usually stores

1. Filename
2. Access control information
3. Size
4. Location of file on disk

The HDFS stores this metadata on a separate server called the namenode, as the metadata is accessed less frequently. The file data is distributed across different machines, as the data is large and needs parallel computation, and each server storing this data is called a datanode.

A client querying the file required receives a datanode list for each block, and the client picks the first datanode for each block and reads the blocks sequentially. The information about the datanode list is provided by the namenode.

Writing a file into the HDFS is done by

1. First the client sends a create statement to the DFS which creates an entry for it on the namenode.
2. Parallel to that the client sends data to the filesystem output stream, which writes this data as packets across different datanodes, and sends back an acknowledgement, which makes the client close the stream/

HDFS follows a master-slave architecture where the namenode acts as a master and the datanodes represent the worker nodes of the infrastructure.

The HDFS Namenode

The namenode manages the filesystem namespace. It is a persistent store, which uses a filesystem tree structure to store data. Data is stored on an fsimage, which is not updated on every write, and also handles an edit log to keep track of requests.

The namenode allows to reconstruct the data in case of power failures on the datanodes. This also means that if the namenode fails in the execution state, it must be made fault tolerant as the filesystem tree cannot be reconstructed without it.

A couple of ways of doing this are

1. Backup the files that make up the persistent state by writing them to multiple filesystems, where these writes are synchronous and atomic
2. It is also possible to run a secondary namenode, which periodically merges the namespace image with the edit log and prevents the edit log from becoming too large. This causes some data loss in terms of primary namenode failure as it lags behind it. So a standby namenode may be used instead.

The HDFS Datanode

The datanodes represent the workhorses of the filesystem, which store and retrieve blocks when told by the client. They notify the namenode about the list of blocks in their store at periodic intervals to keep consistency.

The HDFS Block

The block size indicates the minimum data that can be read or written. Files in HDFS are broken into block sized chunks, which are stored as independent units. A file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage.

The benefits of having such a block abstraction is that a file can be larger than any single disk in the network. Having blocks simplifies the storage subsystem. Blocks also fit well with replication and provide fault tolerance.

Why 128 MB?

Time to read data from disk = seek time + rotational latency + transfer time.

The performance of read can be improved to reduce seek times. This is done when we have a larger block size, which makes transfer time greater than seek time, which is preferred as most of the time should be spent on transfers rather than seek latencies. A size of 128 MB is chosen as a tradeoff between number of blocks and reducing transfer time. The block size was mainly increased to

1. Improve namenode performance
2. Improve performance of MapReduce
3. Decrease the number of blocks created

Pitfalls of the HDFS

1. Low Latency Data Access
    a. High throughput is traded off with latency

2. Lots of small files
    a. A large number of files can result in large namenode sizes
3. Multiple writers
    a. Files in HDFS can be written to only by a single writer
    b. Writes are always made at the end of the file
    c. No support for multiple writers or modifications at offsets
4. Not POSIX compliant
    a. Cannot mount and use standard FS commands

HDFS High Availability

The objective of HDFS HA was to make the namenode resilient to failure. This is done through having an active-standby configuration, which is stored on shared storage and block mappings are stored in memory. On failure, the standby namenode takes over using the shared edit logs and in-memory block mappings.

In this

1. Namenodes must use highly available shared storage to share the edit log. When the standby namenode comes up, it reads till the end of the shared log to synchronize state with the active namenode and then continues to read in sync with the namenode
2. Datanodes must send block reports to both the namenodes as mappings are in memory
3. Clients must be configured to handle namenode failover
4. Secondary namenode's role is the standby's role, which takes periodic checkpoints of the active node's namespace