HDFS

When a dataset outgrows the storage capacity of a single physical machine, it becomes necessary to partition it across a number of separate machines. Filesystems that manage the storage across a network of machines are called distributed filesystems.

Hadoop comes with a distributed filesystem called HDFS, which stands for Hadoop Distributed Filesystem.

The Design of HDFS :

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

Very large files:

"Very large" in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data.

Streaming data access :

HDFS 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 source, then various analyses are performed on that dataset over time.

Commodity hardware :

Hadoop doesn't require expensive, highly reliable hardware to run on. It's designed to run on clusters of commodity hardware (commonly available hardware available from multiple vendors3) for which the chance of node failure across the cluster is high, at least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

These are areas where HDFS is not a good fit today:

Low-latency data access :

Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS.

Lots of small files :

Since 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.

Multiple writers, arbitrary file modifications:

Files in HDFS may be written to by a single writer. Writes are always made at the end of the file. There is no support for multiple writers, or for modifications at arbitrary offsets in the file.

HDFS Concepts

Blocks:

HDFS has the concept of a block, but it is a much larger unit—64 MB by default. Files in HDFS are broken into block-sized chunks, which are stored as independent units.

Having a block abstraction for a distributed filesystem brings several benefits.:

The first benefit :

A file can be larger than any single disk in the network. There's nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster.

Second:

Making the unit of abstraction a block rather than a file simplifies the storage subsystem. The storage subsystem deals with blocks, simplifying storage management (since blocks are a fixed size, it is easy to calculate how many can be stored on a given disk) and eliminating metadata concerns.

Third:

Blocks fit well with replication for providing fault tolerance and availability. To insure against corrupted blocks and disk and machine failure, each block is replicated to a small number of physically separate machines (typically three).

Why Is a Block in HDFS So Large?

HDFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks. By making a block large enough, the time to transfer the data from the disk can be made to be significantly larger than the time to seek to the start of the block. Thus the time to transfer a large file made of multiple blocks operates at the disk transfer rate.

A quick calculation shows that if the seek time is around 10 ms, and the transfer rate is 100 MB/s, then to make the seek time 1% of the transfer time, we need to make the block size around 100 MB. The default is actually 64 MB, although many HDFS installations use 128 MB blocks. This figure will continue to be revised upward as transfer speeds grow with new generations of disk drives.

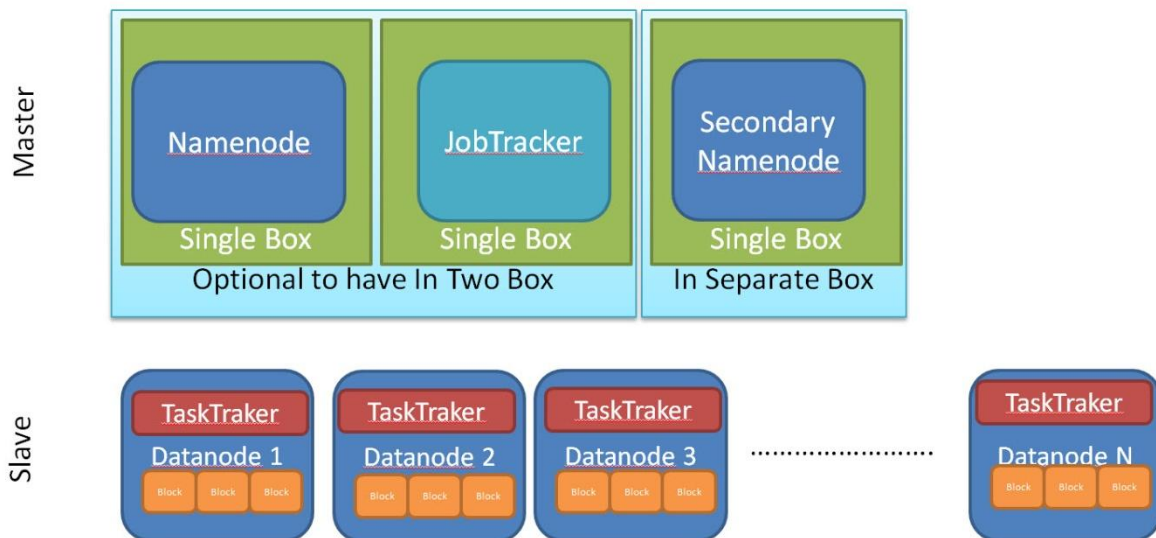
Namenodes and Datanodes:

An HDFS cluster has two types of node 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.

Apache Hadoop is designed to have Master Slave architecture:

Master: Namenode, JobTracker

Slave: {DataNode, TaskTraker}, ….. {DataNode, TaskTraker}



HDFS is one primary components of Hadoop cluster and HDFS is designed to have Master-slave architecture.

Master: NameNode

Slave: {Datanode}…..{Datanode}

- The Master (NameNode) manages the file system namespace operations like opening, closing, and renaming files and directories and determines the mapping of blocks to DataNodes along with regulating access to files by clients
- **Slaves (DataNodes)** are responsible for serving read and write requests from the file system's clients along with perform block creation, deletion, and replication upon instruction from the Master (NameNode).

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.

NameNode failure: if the machine running the namenode failed, 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.

What precautions HDFS is taking to recover file system in case of namenode failure:

The first way is to back up the files that make up the persistent state of the filesystem metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic. The usual configuration choice is to write to local disk as well as a remote NFS mount.

Second way:

It is also possible to run a secondary namenode, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. But this can shaped to act as primary namenode.

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 .

HDFS Federation, introduced in the 0.23 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.

Each Namenode 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 namenode is still a **single point of failure (SPOF),** since 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, and 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 0.23 release series of Hadoop remedies this situation by adding support for HDFS high-availability (HA). In this implementation there is a pair of namenodes in an activestandby 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.

- Datanodes must send block reports to both namenodes since the block mappings are stored in a namenode’s memory, and not on disk.
- Clients must be configured to handle namenode failover, which uses a mechanism that is transparent to users.

Failover and fencing:

The transition from the active namenode to the standby is managed by a new entity in the system called the **failover controller**. Failover controllers are pluggable, but the first implementation uses **ZooKeeper** to **ensure** that only **one namenode is active**.

Failover may also be initiated **manually by an adminstrator**, in the case of routine maintenance, for example. This is known as a **graceful failover**, since the failover controller arranges an orderly transition for both namenodes to switch roles.

In the case of an ungraceful failover, The HA implementation goes to great lengths to ensure that the **previously active namenode is prevented** from **doing any damage** and causing corruption—a method known as **fencing.**