1. **HDFS FEDERATION AND HIGH AVAILABILITY:**

**WHY HDFS FEDERATION:**

HDFS has two main layers:

* **Namespace**
  + Consists of directories, files and blocks.
  + It supports all the namespace related file system operations such as create, delete, modify and list files and directories.
* **Block Storage Service**, which has two parts:
  + Block Management (performed in the Namenode)
    - Provides Datanode cluster membership by handling registrations, and periodic heart beats.
    - Processes block reports and maintains location of blocks.
    - Supports block related operations such as create, delete, modify and get block location.
    - Manages replica placement, block replication for under replicated blocks, and deletes blocks that are over replicated.

The prior HDFS architecture allows only a **single namespace** for the entire cluster. In that configuration, a single Namenode manages the namespace.

**HDFS Federation addresses this limitation by adding support for multiple Namenodes/namespaces to HDFS.**

**HIGH AVAILABILITY**

Before Hadoop 2 comes to the picture, Hadoop clusters were living with the fact that Name Node has placed limits on the degree to which they could scale. Some of the clusters were able to scale beyond 3,000 or 4,000 nodes. NameNode’s require to maintain records for each block of data stored in the cluster turned out to be the most significant factor limiting the greater cluster growth. When we have too many blocks, it becomes increasingly difficult for the NameNode to scale up as the Hadoop cluster scales out.

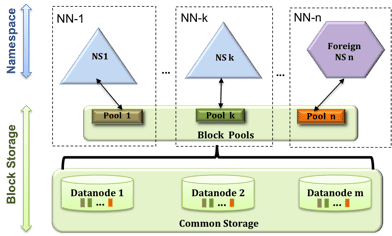
The solution to expanding Hadoop clusters indefinitely is to federate the NameNode. Specifically, we must set it up so that we have multiple NameNode instances running on their own, dedicated master nodes and then making every NameNode responsible only for the data file blocks in its own name space.

Often in Hadoop’s infancy, a great amount of discussion was focused on the NameNode’s representation of a **single point of failure** (SPOF's). Hadoop, entirely, has always had a robust and failure-tolerant architecture design, with the exception only in this key area. As we already know, without the NameNode, there’s no Hadoop cluster. Thus federation has overcame this limitation

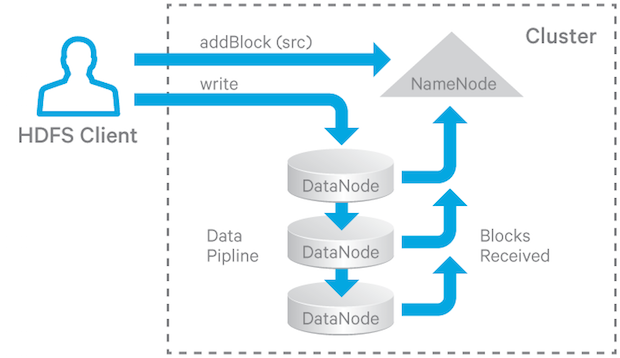
**HDFS FEDERATION**

* In order to scale the name service horizontally, federation uses multiple independent Namenodes/namespaces.
* The Namenodes are **federated**; the Namenodes are **independent** and **do not require coordination with each other**.

* The Datanodes are used as common storage for blocks by all the Namenodes. Each Datanode registers with all the Namenodes in the cluster. Datanodes send periodic heartbeats and block reports. They also handle commands from the Namenodes.
* A **Block Pool** is a set of blocks that belong to a single namespace. Datanodes store blocks for all the block pools in the cluster.
* It is managed independently of other block pools. This allows a namespace to generate Block IDs for new blocks without the need for coordination with the other namespaces. The failure of a namenode does not prevent the datanode from serving other namenodes in the cluster.
* A Namespace and its block pool together are called **Namespace Volume**. It is a self-contained unit of management. When a namenode/namespace is deleted, the corresponding block pool at the datanodes is deleted. Each namespace volume is upgraded as a unit, during cluster upgrade.



**2. How HDFS handles failures while writing data:**



In HDFS, files are divided into blocks, and file access follows multi-reader, single-writer semantics. To meet the fault-tolerance requirement, multiple replicas of a block are stored on different DataNodes. The number of replicas is called the replication factor. When a new file block is created, or an existing file is opened for append, the HDFS write operation creates a pipeline of DataNodes to receive and store the replicas (the replication factor generally determines the number of DataNodes in the pipeline).

There are three stages of a write pipeline:

1. ***Pipeline setup*.** The client sends a Write\_Block request along the pipeline and the last DataNode sends an acknowledgement back. After receiving the acknowledgement, the pipeline is ready for writing.
2. ***Data streaming*.** The data is sent through the pipeline in packets. The client buffers the data until a packet is filled up, and then sends the packet to the pipeline. If the client calls hflush(), then even if a packet is not full, it will nevertheless be sent to the pipeline and the next packet will not be sent until the acknowledgement of the previous hflush’ed packet is received by the client.
3. ***Close***.The client waits until all packets have been acknowledged and then sends a close request.

***RECOVERY***:

1. When a DataNode in the pipeline detects an error (for example, a checksum error or a failure to write to disk), that DataNode takes itself out of the pipeline by closing up all TCP/IP connections. If the data is deemed not corrupted, it also writes buffered data to the relevant block and checksum (METADATA) files.
2. When the client detects the failure, it stops sending data to the pipeline, and reconstructs a new pipeline using the remaining good DataNodes. As a result, all replicas of the block are bumped up to a new GS.
3. The client resumes sending data packets with this new GS. If the data sent has already been received by some of the DataNodes, they just ignore the packet and pass it downstream in the pipeline.