Scaling HDFS cluster using Namenode Federation

Sanjay Radia, Suresh Srinivas
Yahoo!

1 Introduction

Terminology:

Federated HDFS, Namenode Federation	A Federated HDFS or Namenode Federation, in general, allows multiple namespaces within a HDFS cluster and cooperation across clusters. In this document, since it covers the first phase of the project, Federated HDFS is limited to mean Multiple namespaces within a HDFS cluster.
Horizontal Scaling	Scale by adding additional units. (e.g. add more servers – In HDFS, the storage and IO bandwidth is scaled by adding more datanodes).
HDFS Cluster	***Current Model:*** HDFS Cluster includes a single namespace (namenode) and multiple datanodes. ***New Model:*** HDFS Cluster is multiple namespaces (namenodes) shared the storage provided by multiple datanodes.
Namespace	A hierarchical naming structure of directories and files
Namenode	A server that stores and provides access to a namespace
Namespace volume	The namespace and the set of blocks it references. This is an independent unit of management.
Vertical Scaling	Scale by using a larger unit (e.g. a larger server, more memory, more cores, etc)

HDFS cluster has a single namenode that manages the file system namespace. The current limitation that a cluster can contain only a single namenode results in the following issues:

1. ***Scalability:*** Namenode maintains the entire file system metadata in memory. The size of the metadata is limited by the physical memory available on the node. This results in the following issues:
   a. Scaling storage – while storage can be scaled by adding more datanodes/disks to the datanodes, since more storage results in more metadata, the total storage file system can handle is limited by the metadata size.
   b. Scaling the namespace – the number of files and directories that can be created is limited by the memory on namenode.

   To address these issues larger block sizes are encouraged, creating a smaller number of larger files or using tools like the hadoop archive (har).
2. ***Performance Scalability:*** File system operations throughput is limited to the throughput of a single namenode.
3. ***Isolation:*** No isolation for a multi-tenant environment. An experimental client application that puts high load on the central namenode can impact a production application.
4. ***Availability:*** While the design does not prevent building a failover mechanism, when a failure occurs the entire namespace and hence the entire cluster is down.

Limits to Vertical Scaling of Namenode

To support more files and storage capacity, the current approach is to vertically scale the namenode by increasing the java heap for the namenode process. A cluster of 10.4PB and 3700 datanodes is currently configured with java heap at 40GB. As the demand for cluster capacity increases, the cluster will run into namenode physical memory limitation (64GB on some machines). Optimizing the namenode to use memory more efficiently is expensive both in development effort and complexity added to the code. Further, using larger java heap results in GC issues, and longer startup time. It also makes debugging JVM related problems harder; since such a large JVM is not supported well by tools such as jhat.

1.1 Background

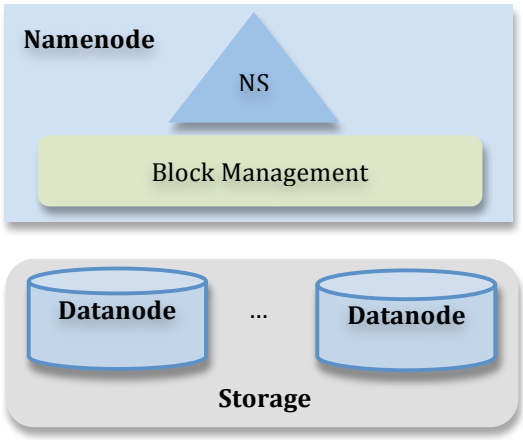
Currently HDFS architecture has 2 main layers:

- Namespace management - manages namespace consisting of directories, files and blocks. It supports file system operations such as creation/modification/deletion and listing of files and directories.
- Block storage has two parts:
    - Block management – manages the membership of datanodes in the cluster for block storage. Supports block related operations such as creation/deletion/modification/getting location of blocks, replica placement and block replication to satisfy replication factor and placement.
    - Physical storage and access to block data across the network.

The current HDFS implementation is structured as follows:

- The namenode implements - Namespace management and Block management
- The datanodes provide the physical storage and access to block data. Datanodes register with the block management layer in the namenode to form the storage layer for the HDFS cluster.
    - Note although we think of the datanode as registering and communicating with the namenode, it is really communicating with the block management layer inside the namenode and not with the namespace management layer.
- Hence the block storage subsystem is implemented partly in the datanodes and partly in the namenode.

Within the namenode implementation, the internal java APIs do not provide clean separation.



Block Identification: A file is one or more blocks with replicas stored on datanodes. Each block is identified with a unique cluster-wide block ID (64-bit number). Current system is a system with a single namespace, using a single pool of blocks for storage.

2 Federated HDFS and Block Storage Architecture Overview

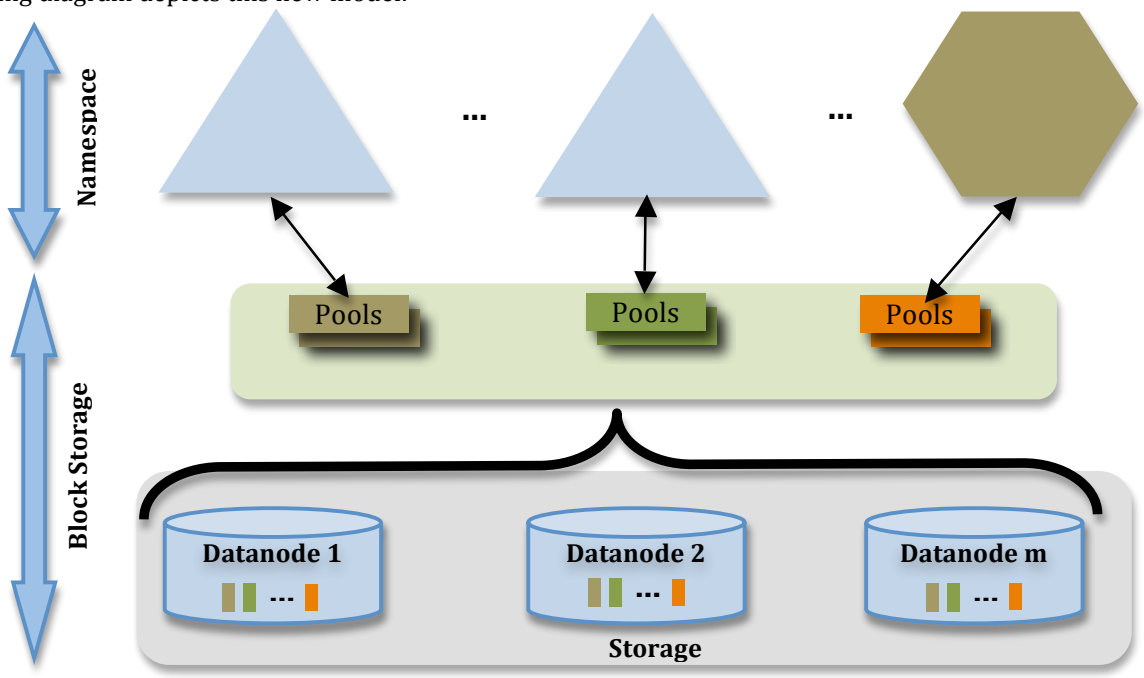
In the proposed architecture, to improve scalability, multiple namenodes/namespaces are used in the cluster. Each namespace volume uses **all** the datanodes in one or more sets of blocks called "block pools".

HDFS Cluster definition:

Current HDFS	**Federated HDFS**

• A single HDFS namespace implemented inside a single namenode
• A single pool of storage implemented using multiple datanodes
	• Multiple independent HDFS namespaces each implemented on a separate namenode.
	• A single pool of storage implemented using multiple datanodes
	  ○ Data nodes are not partitioned – a datanode can provide storage to all namenodes
	  ○ The storage contains independent pools of blocks - each pool of blocks is owned by a single name node.

The following diagram depicts this new model:



Salient features of Block Storage:

- A Block Pool is an independent set of blocks that belongs to a single entity (e.g. namespace). A block pool is managed independently of other pools, allowing owning entities to generate Block IDs for new blocks without the need for co-ordination with other entities.
- Block management – manages the membership of datanodes in the cluster for block storage. Supports operations such as creation/deletion/modification/getting location of blocks, replica placement and block replication to satisfy replication factor and placement.
- Datanodes provide a shared storage layer and store blocks belonging to **all** the block pools.
- Datanodes maintain the block ownership at the block pool level and not at the individual block level.
- Each datanode communicates with the block management layer:
    - Registers and sends periodic heartbeats
    - Sends block reports for each block pool.
    - Accepts commands for block management (copy blocks, delete blocks, etc.)

Salient features of Multiple Namespaces:

- A Namespace and its block pool together are called **Namespace volume**. It is a self-contained unit of management.
    - It can be garbage collected independently - When a namenode/namespace is deleted, the corresponding block pool in the datanodes can be deleted.
    - Namespace Volumes don't need to coordinate with other namespace volumes.

2.1 Benefits

Benefits of Block Storage Layer:

1. Separating the block storage layer enables:
    a. Implementation of a non-HDFS namespace using the block storage layer.
    b. Other applications such as HBase can use block storage layer directly.
    c. Separation of block storage layer enables future work such as distributed namespace.
2. Multiple applications share the common datanode pool for storage resulting in better storage utilization compared to partitioning datanodes among the applications. Additional details of separation of block storage layer are discussed in Appendix B.
3. We are investigating special block pools for MapReduce temp storage (details soon).

Benefits and drawbacks of multiple namespaces:

1. Provides a horizontal scaling solution for namespace and namespace operations and hence for the entire HDFS file system.
2. Multiple namenodes enable partitioning customers to different namespaces/namenodes for better isolation, availability and manageability.

3. However, there is a drawback with this approach. One needs to manage multiple namespaces and namenodes. To provide user transparency, one can use symbolic links and/or client-side mount tables namespaces (See HDFS-1053). The operations staff has to partition the current single namespace into multiple namespaces and run multiple namenode servers.

2.2 High Level Design

Terminology:

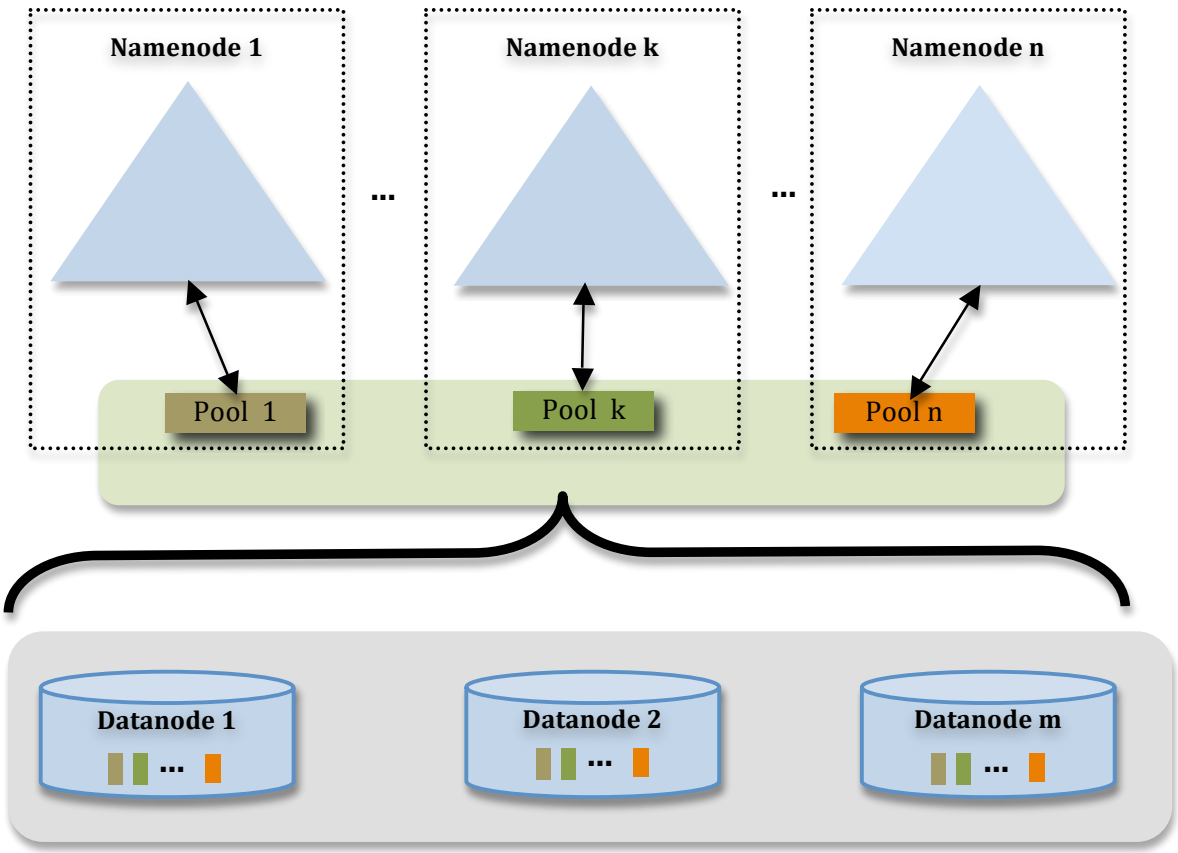
BP	Block Pool
Birthmark	An identifier that is globally unique across all the clusters to uniquely identify an entity.

The architecture overview of the previous section defined the layers of abstraction without discussing server boundaries. This section provides a concrete realization of the architecture, defining which parts of the abstraction layers reside in which servers. The design, depicted in the picture below has the following properties:

1. Each namespace uses only one block pool. (The general architecture allows multiple block pools in namespace but such generalization will not be supported in the first phase.)
2. As in current HDFS, a namenode implements two layers:
    a. The namespace layer that manages a single namespace
    b. The block management layer that manages the pool of blocks used by the namespace.

    That is, a namenode manages a single namespace volume.
    Even though the namenode continues to have the namespace and the block management layers, the goal is to cleanly separate the two layers. This separation will help in future if we need to move block management out of namenode, into separate layer. For the discussion related to having a separate block management servers, see Appendix B.

3. Each namenode is completely independent of other namenodes.
4. The entire cluster is upgraded or rolled back as a whole, as done currently. Rolling upgrades is not a goal for this.



3 Managing Namespace Volumes and Block Pools

The following requirements needs to be considered when designing block pool functionality:

1. A block pool, identified by its *BlockPoolID* belongs to a single namespace. Any violation of this rule is an error and the system must detect this and take appropriate action or avoid it in the first place.
2. DN may come back after a very long time holding a stale block pool that has been no longer used because its namespace was deleted.
3. When a datanode is moved intentionally or accidentally to another cluster must be detected and the blocks pools on it must not conflict with the block pools on the new cluster.
4. Optional: the design must simplify merging of two clusters.

3.1 Identifiers

Block Pool ID:

A block pool ID identifies a block pool. It is a unique ID across all clusters.. When a new namespace is created (as part of the format operation of a NN) a unique id is generated and stored persistently by the namenodes. Using a unique *BlockPoolID* simplifies the creation, as it is done on the fly on the namenodes instead of an error prone alternative of an administrator configuring it. Namenodes persist the *BlockPoolID* information on the disk and reuse it in subsequent restarts.

The following table describes the identifiers used in the current cluster and the new federated HDFS:

Current HDFS		**Federated HDFS**	
Identifier and when created	**Purpose of identifier**	**Identifier and when created**	**Purpose of identifier**
		ClusterID • Birthmark of cluster – globally unique ID created when a cluster is created. Admin runs a command to create and store in config file.	Detects a DN or NN configured to belong to the wrong cluster
		• *ClusterID* has an associated cluster name.	Used for logging and report generation.
NamespaceID • Birthmark of the NN and effectively the cluster. • Created when NN is formatted.	Detect a NN or DN configured to wrong cluster? All communication from DN to NN includes *NamespaceID*. A DN request with non matching *NamespaceID* is rejected by the namenode.	*NamespaceID* • Birthmark of NN used to show ownership of a block pool. • Created when NN is formatted.	Globally used in future when a BP is moved from one NN to another
StorageID • Birthmark of DN • Created when DN is started for the first time.	Allows IP of DN to change in a running cluster.	*StorageID* • Same • Same	Same
BlockID – created by NN when block is created.	Identifies a block	*ExtendedBlockID*	Identifies a block
		BlockPoolID • Created when NN is formatted (later when new BPs are added to an existing Namespace/NN	Identifies a block pool.
		BlockID	Identifies a blocks within a

		• Created by NN when block is created.	BP.

3.1.1 FAQ

1. Why do we need *ClusterID*?
   a. Cannot solve it merely with global BlockPoolIDs or NamespaceIDs - In federation setup, a DN talks to multiple NNs. If a DN is accidentally moved to another cluster, the DN continues to keep its old blocks and creates new block pools for the NNs in the new cluster. The NamespaceID that previously prevented such moves will not work in this case.
2. Why is *BlockPoolID* globally unique?
   a. Rather than assure the uniqueness of block pool ID via admin configuration, a unique id can be generated programmatically.
   b. Clearly it has to be unique within a cluster. It is equally easy to make them globally unique or cluster unique. Only cost is # of bits.
   c. Allows deletion of a block pool without worrying about accidental reuse – indeed our first solution was to make it cluster unique and we were then forced to add a BP-birthmark.
   d. Allows cluster merges.
   e. If BPID is not unique, due to the possibility of reusing a deleted BPID, two instances of block pools might conflict. An associated birthmark is needed to distinguish the two instances.
3. Why not use *NamespaceID* as the block pool id (i.e. get rid of *BlockPoolID*)
   a. Wrong abstraction layer – the block layer does not know or give a damn about who uses it above. Hence its name should be *BlockPoolID*. (You can argue to get rid of NamespaceID).
   b. In future we want to allow a namespace to have multiple bps.
   c. In future we want to be able to move a bp from one NN to another. A that time the NamespaceID uniquely defines the single owner.

3.2 Namespace Volume management

3.2.1 Current Cluster and Namespace management

Current Cluster setup:
Currently, cluster is setup by specifying namenode information in core-site.xml/hdfs-site.xml. All the nodes in the cluster share this configuration. Datanodes use this configuration to communicate with the primary namenode.

Primary NN is setup with the following files:
1. *dfs.include* – list of datanodes that are allowed to register with the namenode and hence join the cluster.
2. *dfs.exclude* – list of datanodes excluded from the cluster. A datanode that has already registered with the NN, when moves to this file is decommissioned.

Additionally the following files are used for starting/stopping the cluster:
1. *masters* – includes information about the secondary namenode. Startup scripts start secondary NN on these nodes.
2. *slaves* – includes the list of all the datanodes. Startup script starts datanode process on these nodes.

Current Namespace creation:
When a namenode is formatted a new namespace is created identified by *NamespaceID*. Namenode persists the *NamespaceID* and sends to all the datanodes in registration response. Datanode persists the *NamespaceID* and from then on, it works only with that namenode.

Current Namespace deletion:
Reformatting namenodes and all the DNs deletes the namenode and the entire HDFS cluster. There is no way to do global reformat of DNs via the NN.

3.2.2 Federated Cluster and Namespace Volume management

3.2.2.1 Cluster setup

A HDFS cluster is initialized when the very first namespace volume of the cluster is created. As part of formatting a NN with the "-newCluster" option, it will generate a unique *ClusterID* and a unique *BlockPoolID*, which are persisted on the namenode.
- Subsequent NN must be given the same *ClusterID* during its format to be in the same cluster.
- Each DN discovers the *ClusterID* when it registers and from then on "sticks" to this cluster.

- If at any point a NN or a DN tries to join another cluster, the DNs or the NNs in that cluster will reject registration.

The following files are used for configuring the cluster:

File	Description	Used By			
		Client	NN	DN	BN
hdfs_site.xml	Configuration file that overrides *hdfs_default.xml*. This includes all the configuration parameters common to all the nodes in the cluster and the clients.	Y xxx N?	Y	Y	Y
hdfs_nn.xml	Configuration that lists namenodes and the corresponding backup nodes.	N	N	Y	Y
hdfs_client.xml	Used by only the clients to HDFS. Typically used for overriding ***fs.default.name*** configuration at the client.	Y	N	N	N
dfs.include.xml *dfs.exclude.xml*	Same as today.	N	Y	N	N

The following files are used only the cluster startup scripts (they are not read by the HDFS code):

1. *masters* – includes information about all the namenodes and the corresponding secondary namenodes. Startup scripts start Primary or Secondary NNs on these nodes.
2. *slaves* – includes the list of all the datanodes. Startup scripts start datanode process on these nodes.

3.2.2.2 Namespace Volume creation

1. When a namenode is formatted a new namespace and corresponding block pool is created. Namenode persists the *NamespaceID* and *BlockPoolID*.
2. Datanode discovers the *NamespaceID*, *BlockPoolID* and *ClusterID* of a namenode during pre-registration handshake. If the datanode is registering with the namenode for the first time, it formats a new block pool and records *NamespaceID* and other block pool relevant information.

3.2.2.3 Adding a Namespace Volume (namenode) to the cluster

1. At the new NN, run the format-NN command and supply it the *ClusterID*. Namenode generates a unique *BlockPoolID* and persists the *ClusterID* and *BlockPoolID* in its metadata
3. DNs in the cluster are given the NN-refresh signal to reread the list of NN config file.
    - Each DN registers with the new NN and create a new directory for the NN's Blockpool.
    - If a datanode is down, it gets the latest configuration when it starts back up and registers with the newly added namenode.

3.2.2.4 Adding a new datanodes to the cluster

1. Cluster configuration files *dfs.include are* updated to include the new datanodes.
2. The new datanodes are formatted and started.
3. The new datanodes read the NN list config file and register with each namenodes.
    a. On the registration with first NN, datanodes learn *ClusterID* and become part of the cluster.
    b. For each NN it creates the directory to store the blocks of the NN's block pool.

3.2.2.5 Namespace Volume deletion

A Namespace Volume is deleted as follows:

1. Start cluster and delete all the files of volume on the namenode.
2. Reformat NN.
3. Go to each DN and issue the delete BP command. If the block pool has no blocks on the datanode, it is deleted. Command can also be run with an option-***forced*** to delete the block pool even if it has blocks in it.

3.2.2.6 Move a namespace from one namenode to another namenode within a cluster

1. Stop the namenode.

2. Copy required configuration and data directories (**TBD**) with metadata from one namenode to another namenode.
3. The DNS name and/or address of the namenode are updated in the config file and or the DNS server as appropriate.
4. Ensure the old namenode is stopped. Start the new namenode.

To move a namespace volume to a different cluster, distcp needs to be used to copy the data to a different namenode in version 1.

3.2.2.7 Move part of a namespace from one namenode to another namenode

1. Copy using tools such as distcp.
2. Delete the sub namespace from the first namenode.

In the future, we may support creating a copy-on-write snapshot with a different block pool ID and then moving the snapshot.

3.2.2.8 Move a namespace to another cluster

1. Copy the namespace to another cluster using tools such as distcp.
2. Delete the namespace from the first cluster.

3.2.2.9 Merging two clusters into a single Federated Cluster.

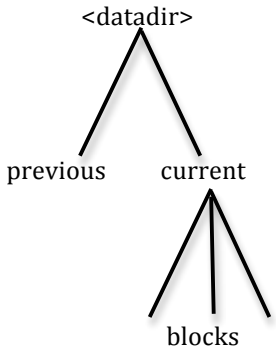
Merging of two clusters is done in following steps:

1. A new tool will be provided to rename the *ClusterID* of a cluster. For one of the cluster, the *ClusterID* is renamed to that of the other cluster it is merging with. Both the clusters are shutdown.
2. The *includes* and *excludes* file from both the clusters is merged together.
3. Both the clusters are started.
4. Optionally balancer is run to balance the storage utilization.
5. Client side mount tables are updated to provide transparent access to both the namespaces.

3.3 Block Storage

Current Structure:

Datanodes store the block data on disks on the local file system. The blocks are stored under the data directory in a directory structure as shown below:



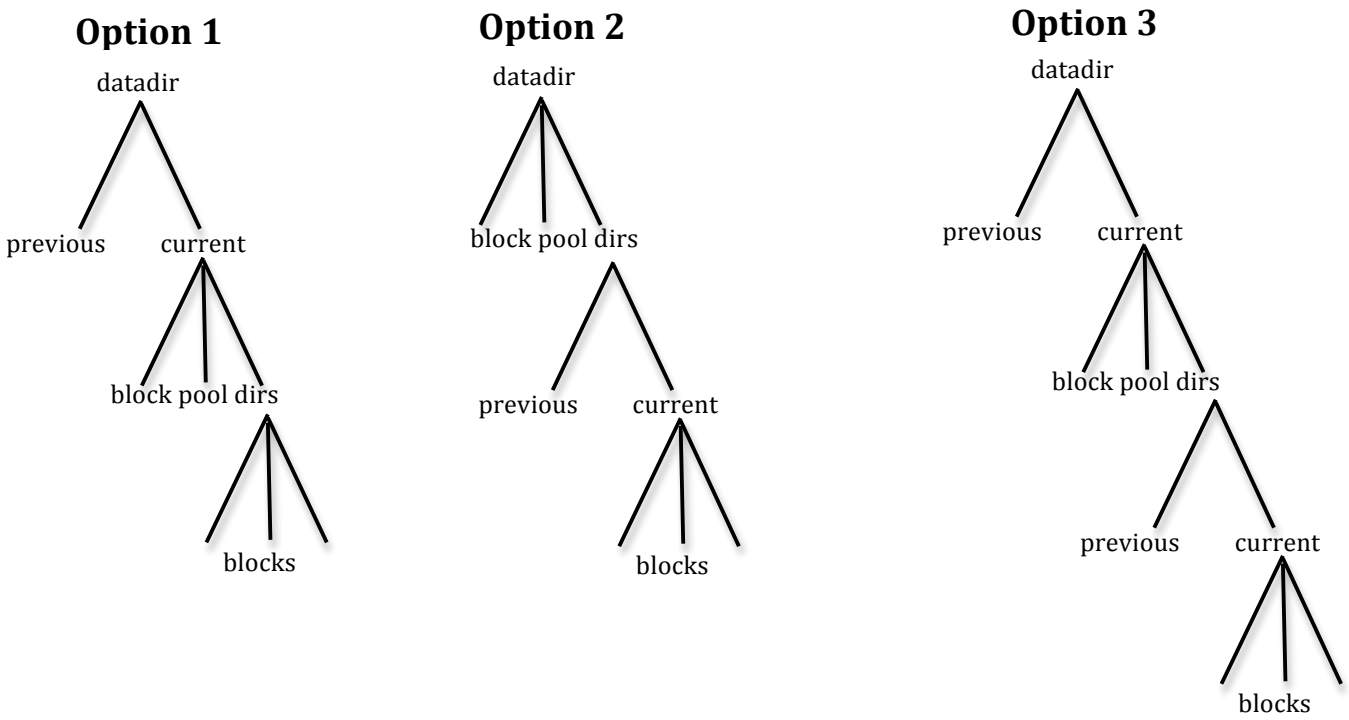
The directories *previous* and *current* are used during upgrades as described in HADOOP-702. To avoid data loss/corruption during the upgrade, a snapshot of the file system data is created both on the namenode and the datanodes for rollback purposes.

During snapshot, on datanode:

1. `<datadir>/current` is moved to `<datadir>/previous`
2. Datanode metadata files are created under `<datadir>/current`.
3. Block files are created in `<datadir>/current` by hard linking to block files in `previous`.

During rollback, `previous` is moved back to `current` to restore the file system from the snapshot.

Block Storage with Federation:
Datanode storage directory hierarchy needs to be changed to include block pool IDs. There are several choices for the directory structure, as shown below:



Option 1:
This can create snapshot only at the datanode level. Individual namenode upgrade will result in a snapshot for all the block pools. This solution is not ideal to maintain independence of namenodes from each other and possible rolling upgrades in the future.

Option 2:
Enables snapshot at block pool level and solves the problems described with option 1. However for the first upgrade, in order to rollback, `previous` directory under <datadir> is still needed. This is needed only for rollback. Finalizing the upgrade will delete the directory.

Option 3:
In addition to the features provided by Option 2, this option enables the datanode level snapshot. In future if datanode upgrade is decouple from the namenode, a datanode level snapshot can be created. This also allows each namenode to independently create snapshots to backup data at a block pool level.

3.4 Single Checkpointer

Currently for every primary namenode, a checkpointing namenode (either backup or secondary namenode) is used. Checkpointer periodically merges the *fsimage* and *editlog*, to generate a new *fsimage.* To reduce the number of nodes, a single node would be used for checkpointing for all the namenodes. The details of single checkpointer will be finalized during the implementation phase.

3.5 Network Partition and Federated Cluster

Due to network segmentation, a namenode could lose a large number of datanodes. This could trigger a replication storm and fill up available datanodes. This can occur in the current HDFS cluster. In federated environment, this problem could be more severe, as network segmentation might result in each namenode seeing different set of datanodes. This problem is addressed by HDFS-779, which puts namenode back into safemode, on losing large number of replicas.

4 Cluster Management

4.1 Web UI

4.1.1 Namenode Web UI

Cluster Summary:
The existing Cluster Summary section will be enhanced to show ClusterID, block pool ID and storage used by the block pool, along with the overall cluster storage utilization.

Live Nodes:
The column **Blocks** indicates number of blocks stored on the DN. This will change to indicate the number of blocks belonging to the block pool of the NN. A new column **Block Pool Used** and **Block Pool Used %** will be indicate the storage used for the block pool and % of total storage of the DN used for the block pool.

Decommissioning status:
Decommissioning status shows the decommissioning status of each of the block pools and the entire datanode.

4.1.2 Cluster Web UI

Namenode servlets and JSPs will be changed to provide the information displayed on Namenode UI as a structured data (XML?). This information will be used to build a Cluster Web UI that will include the following information:

1. Cluster Summary: Show overall cluster utilization, list of namenodes, and for each namenode - block pools owned by it, storage utilization of the block pools, missing block information, number of live/dead/decommissioning datanodes.
2. Clicking on the namenode will take the administrator to namenode Web UI.

4.2 Upgrade

4.2.1 Upgrade Mechanism

Currently upgrading a namenode triggers upgrade of the entire cluster. Datanodes connect to the namenode and when the *ctime* or layout version does not match, take a snapshot of the blocks and perform local upgrade. Entire cluster is upgraded to a new release; no rolling upgrades can be performed.

Under Federation, in phase 1, we do not allow mixed cluster – i.e. some NNs or some DNs running different software versions. Each NN will upgrade separately, and DNs upgrade the namenode's block pools, during registration with the namenode. The upgraded is performed at block pool level as shown below:

1. A datanode gets the software version of the namenode in pre-registration handshake. If the datanode is running a different software version, it does not register with the namenodes running older software version. Datanode periodically retries connecting to these namenodes to check if the namenode has been upgraded.
2. During datanode pre-registration handshake with an upgraded namenode, if the *ctime* of the corresponding block pool is not equal to that of the namenode *ctime, a* snapshot of the block pool is taken and the block pool is locally upgraded. The other block pools are not changed.
3. During rollback, the datanode rolls back all the block pools. If a namenode in the cluster is not rolled back and is still running newer version, the datanode will not register with it. Datanode periodically retries connection to these namenode to check if the namenode has been rolled back.

Note:
Consider a situation where all but one NN is upgraded. The last one could not be upgraded because the NN was not reachable or the server had some problems. An admin could choose to run production jobs on such a partially upgraded cluster, with one namespace missing. Later the last NN is ready to be upgraded and is upgraded by the admin. While retrying connection to namenode in step 1, datanodes connect to the newly upgraded namenode, resulting in snapshot creation and upgrade of the block pool. This could impact the performance of running jobs. To avoid this, an administrative option may be needed not to retry connection to a namenode until the next maintenance window.

4.2.2 Upgrading to Federation release

Namenode changes:

1. Namenode when it first comes up, creates a *BlockPoolID* and adds *BlockPoolID* and *ClusterID* into VERSION file.
2. Namenode stores, in memory, all the blocks it has under the newly created *BlockPoolID*.

Datanode changes:

1. After startup, stores the new *ClusterID*
2. During first registration sends the *StorageID* and *BlockPoolID*= null
3. Gets registration response with *ClusterID*, *NamespaceID* and *BlockPoolID*. If the system does not have any *BlockPoolID*, DN moves the existing blocks under the new *BlockPoolID*.
4. Block report sent post registration reports all the existing blocks under the new *BlockPoolID*.

Rolling back remains the same and is done by moving the <datadir>/previous to <datadir>/current.

4.2.3 Backward compatibility

Changing BlockID to *ExtendedBlockID*, to include *BlockPoolID*, must not affect the user application. The impact of this change must be limited the input/output streams and should not be visible to the applications.

4.3 Decommissioning

Decommissioning currently works as follows:

1. Nodes to be decommissioned are added to *excludes* file and node list is refreshed at the namenode.
2. NN marks the datanode state as *decommission_in_progress* and starts replications for blocks in those nodes.
3. Datanode state in web UI and other tools are given as **decommission in progress**.
4. When the replication of the replicas is complete, the datanode is marked as decommissioned. It eventually shuts down.

With federation, a datanode is decommissioned when all the blocks in all the block pools are replicated. New tools will be provided:

d. To start decommissioning.
e. To query decommissioning status.

This tool can be run from any machine with access to the cluster and has cluster configuration (like gateway machines).

New decommissioning process:

1. Datanodes to be decommissioned are added to *excludes* file. The tool to start decommissioning is run with *excludes* file. The tool connects to each namenode over ssh, copies *excludes* file and runs "refresh nodes" command. This triggers decommissioning on each namenode as it is done currently.
2. NNs replicate the blocks from the block pools they own on the DN. When no more replication is pending, namenode updates the datanode state as *decommissioned*. **Namenodes do not shutdown datanodes after decommissioning is complete.**
3. A tool can be run to query decommissioning status. The tool talks to all the namenodes and provides the consolidated state of datanode as:
   - Decommissioned – if all the namenodes indicate the state as *decommissioned*.
   - Decommissioning Started – if all the namenodes indicate the state of datanode as either *decommissioned* or *decommission_in_progress*.
   - Decomissioning Partially Started– if some of the namenodes has the datanode in normal state, i.e. not *decommissioned* or *decommission_in_progress.* This could happen either if a new namenode was added to the cluster after starting the decommissioning or if a namenode was not reachable to start decommissioning or if a namenode is restarted while decommissioning. Admin needs to run the tool to start decommissioning again. Since this operation is idempotent, this affects only the namenode where decommissioning is started.
4. Tools will be provided to query the decommissioning status and to shutdown datanodes that are decommissioned.

4.4 Distributed Upgrade

Distributed upgrade is an upgrade that is not locally performed, as it requires co-ordination between the nodes in the cluster. Example – when CRC storage was moved to meta data files, datanodes communicated with each other to ensure computed CRC are same for the blocks, during upgrade.

There is no need for distributed upgrade when moving to Federated HDFS. In future, projects that need distributed upgrade, need to consider the existence of multiple namenodes, in the cluster.

4.5 Balancer

Current balancer mechanism:

Currently balancer works with a single NN, by getting the list blocks, datanodes and utilization information of the datanodes. Balancer is run with balancing threshold t% as input. The datanode storage is balanced when the following condition is met:

$$\bar{c} - t <= d_i <= \bar{c} + t$$

d_i is the % storage utilization of every DN in the cluster
$\bar{c}$ is the % average storage utilization in the cluster
t is the % balancing threshold

Balancer with Federation:
With block pools, the goal is to:
1. Balance the datanode storage as currently done.
2. In addition, balance every block pool to meet the following condition:

$$\bar{c} - t <= b_i <= \bar{c} + t$$

b_i is the % storage utilization for a block pool i on every DN in the cluster
$\bar{c}$ is the $ average storage utilization in the cluster
t is the % balancing threshold

Balancer mechanism:
Balancer balances the block pool storage. When all the block pools are balanced, the datanodes are also balanced. The balancing algorithm is shown below:

```
while (cluster_balanced is not true) {
    // Balance one block pool at a time – as much as possible
    for (block pool b : block pool list) {
        if (b is not balanced)
           balance b as much as possible or for time unit x;
    }
    if (all block pools are balanced)
        cluster_balanced = true;
}
```

4.6 Cluster startup/shutdown

HDFS has scripts such as start-all.sh and start-dfs.sh that starts or stops the namenodes and datanodes that are part of the cluster. This is done using *slaves* file that lists all the datanodes that are part of the cluster. Starting and stopping of the cluster can be done from any of the nodes that are in the cluster.

With block pools following scripts are required:
1. Ability to stop and start all the datanodes in the cluster.
2. Ability to stop and start all the namenodes, specific list of namenodes or a single namenode.
3. Ability to stop and start the whole cluster (all the datanodes and all the namenodes)

5 Security

Job tracker currently gets delegation tokens on behalf of the user, for the submitted jobs. A delegation token for every NN in the cluster is needed while submitting the job. Also tasks must use appropriate delegation token corresponding to the NN, while sending requests to a NN. DFSClient already handles this.

Client side mount tables must also work with multiple delegation tokens. The new file system that handles client side namespace must choose the right delegation token when accessing a namenode.

DFSClient already works uses appropriate access token when accessing datanode. Access token will need a change to include the block pool ID along with block ID information.

Appendix A - Use Cases

Following use cases have been considered in the document:

Use case 1. Creating a new Federated HDFS Cluster
(Actor – Grid operations or any user installing the system)
See section 4.2.2.1

Use case 2. Adding datanodes to a cluster
(Actor – Grid operations or any user installing the system)
See section 4.2.2.4

Use case 3. Adding a namenode/namespace volume to a cluster
(Actor – Grid operations or any user installing the system)
See section 4.2.2.3

Use case 4. Delete a namenode/namespace volume from a cluster
(Actor – Grid operations or any user installing the system)
See section 4.2.2.5

Use case 5. Datanode is misconfigured and accidentally moved to a cluster
(Actor – Grid operations or any user installing the system)

Use case 6. Move a namespace from one namenode to a different namenode with in the cluster
(Actor – Grid operations or any user installing the system)
See section 4.2.2.6

Use case 7. Move a part of namespace from one namenode to a different namenode with in the cluster
(Actor – Grid operations or any user installing the system)
See section 4.2.2.7

Use case 8. Move a namespace from one cluster to another cluster.
(Actor – Grid operations or any user installing the system)
See section 4.2.2.8

Use case 9. Upgrade cluster from old release to a new release with federation
(Actor – Grid operations or any user installing the system)
See section 5.2.2

Use case 10. Rollback a cluster from release with federation back to the previous release
(Actor – Grid operations or any user installing the system)
See section 5.2.2

Use case 11. Decommissioning datanodes
(Actor – Grid operations or any user installing the system)
See section 5.4

Use case 12. Balancing storage utilization
(Actor – Grid operations or any user installing the system)
See section 5.5

Use case 13. Centralized cluster monitoring
(Actor – Grid operations or any user installing the system)
See section 5.1

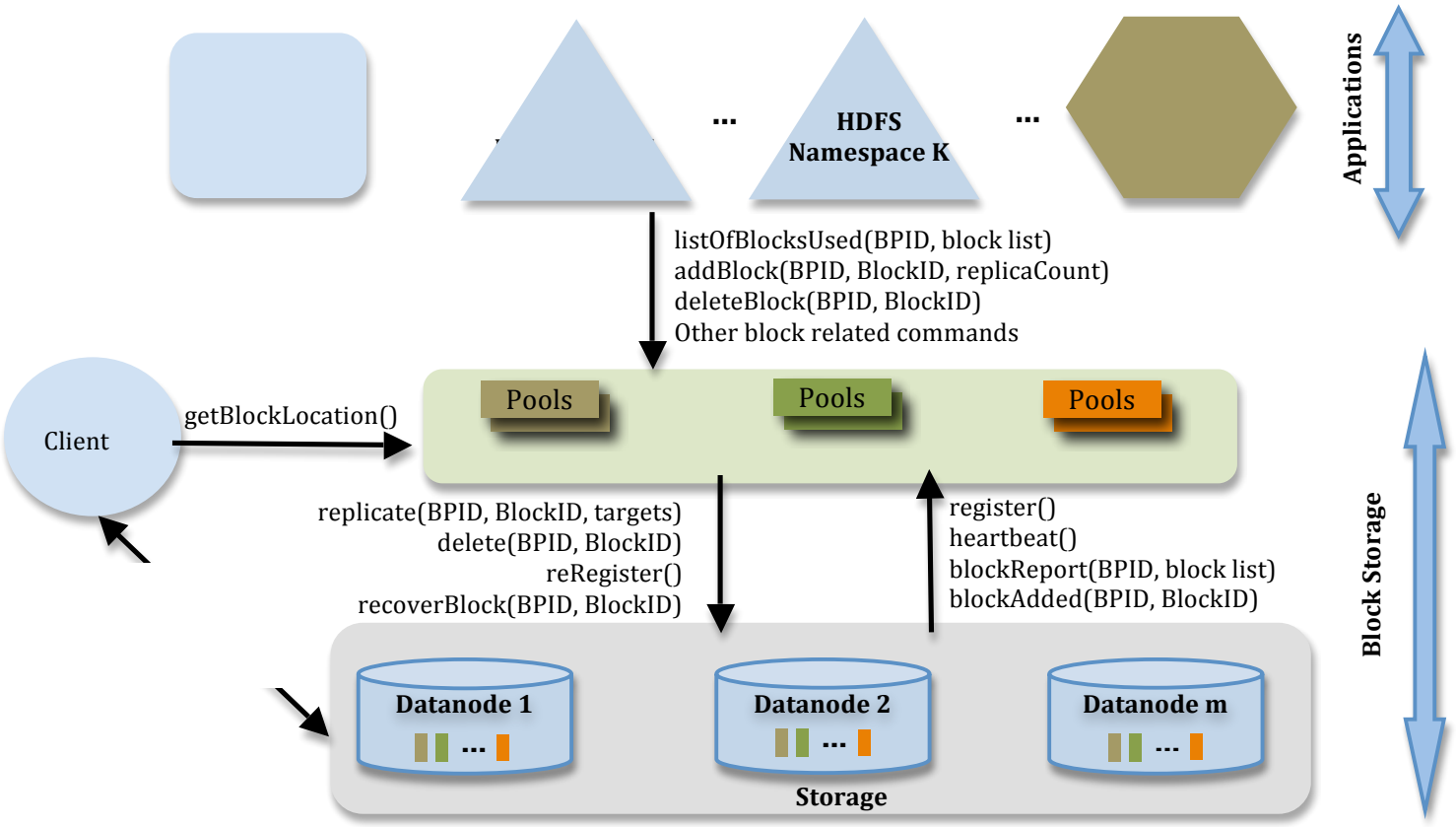
Use case 14. Merging two cluster into a single federated HDFS cluster
(Actor – Grid operations or any user installing the system)
Section 4.2.2.9

Following use cases are not considered in the document:

Use case 15. Splitting a federated cluster into multiple clusters.

Appendix B - Separating Block Storage Layer

Block Storage is a separate layer with in HDFS, used by namespaces to store blocks. See section 1.1 for details. Block storage layer with some of its interface methods is as shown below:



This separation has several advantages:

1. Clean separation of namespace from the block storage layer.
2. Other namespaces and applications can directly use the block storage layer for storage, without the need for HDFS namespace/namenode.

Based on where the block management lies, two solutions are possible:

1. **Block Storage interface as a library** - Block management is available as a library that gets included in the application. For example, in the namenode the block management is included with the application.
2. **Block Storage as a service** - Block management is done as a service by a separate set of nodes. These block management nodes along with the datanodes become a *storage cluster*.

Separating the storage layer as a service solves many problems.

1. Scaling block management is simpler.
    a. Given that the block ID space is flat, simpler mechanism based on hashing could be used for distributing the block pools/blocks among multiple block management nodes.
2. Applications need to scale only the functionality it provides without requiring to scale the block management embedded in it. This also insulates the application from block management head aches such as replication storm, dealing with loss of datanodes etc.
3. Applications are independent of each other and do not co-ordinate with each other. However nodes doing the block management can co-ordinate with each other and have more complete picture of the storage cluster. This has the following benefits:
    a. Simplifies/reduces the number of messages from datanodes to block management, such as, registration, heartbeats.
    b. Block management layer can more effectively co-ordinate balancing the storage, decommissioning datanodes and can deal with network partitions.
4. Block management layer provides central cluster web UI, management of the cluster, starting and stopping of the cluster.

Appendix C – Namenode and Datanode metadata changes

Datanode storage directory:

Following files in datanode data directory will remain unchanged:

<data>/in_use.lock
<data>/storage

Following files/directories in datanode data directory will change to include bpid:

Current directory/files	**New directory/files**
<data>/tmp	<data>/tmp/bpid
	<data>/current/bpid/current/bp-<bpid>.meta with (*NamespaceID*)
<data>/current/finalized/<blocks and subdirectories>	<data>/current/bpid/current/finalized/<blocks and subdirectories>
<data>/current/rbw/<blocks and subdirectories>	<data>/current/bpid/current/rbw/<blocks and subdirectories>
<data>/current/VERSION	<data>/current/ID
	<data>/current/bpid/current/VERSION
<data>/previous/…	<data>/previous/… <data>/previous/bpid/previous/…

Datanode VERSION file:
Currently Datanode has a file <data>/current/VERSION that includes the following information:

- Type of node (DATA_NODE)
- *StorageID*
- *NamespaceID*
- *ctime*
- *LayoutVersion*

With federated HDFS this file is split into two files with the following information:

1. <data>/current/ID
   - Type of node (DATA_NODE)
   - *StorageID*
   - *ClusterID*
2. <data>/current>/bpid/VERSION (one for each block pool)
   - NamespaceID
   - *BlockPoolID*
   - *ctime*
   - *LayoutVersion*

Namenode VERSION file:
Currently namenode stores the following information:

- Type of node (NAME_NODE)
- NamespaceID
- *ctime*
- *LayoutVersion*

With federation, the VERSION file is split into the following files:

1. <data>/ID
   - *ClusterID*
   - NamespaceID
   - *BlockPoolID*
   - Type of node (NAME_NODE)
2. <data>/VERSION
   - *ctime*
   - *LayoutVersion*

Appendix D - Namespace Volume creation

1. Update hdfs_nn.xnl to include the DNS name of the new NN.
2. At the namenode: Run format-NN command. Namenode generates a unique *BlockPoolID* and persists it in *ID* file, which stores (*ClusterID, NamespaceID*, *BlockPoolID*, type of node). *VERSION* file includes (*ctime*, *LayoutVersion*). For details see Appendix C.
3. At the datanode: Send a refresh-NN signal to each DN. Each DN will register with new NNs.
   a. **First time registration with a new NN and discovering new namespace/block pool**
      - Datanode sends *versionRequest* to the namenode
      - Namenode sends in response NamespaceInfo that includes (*LayoutVersion, NamespaceID, BlockPoolID, ctime, BuildVersion, DistributedUpgradeVersion*).
      - If the Datanode does not have *ClusterID*, it saves it in *ID* file along with *StorageID* and *type of node*.
      - Datanode formats a new block pool and stores it in per block pool *VERSION* file, which stores (*LayoutVersion*, *NamespaceID*, *BlockPoolID*, *ctime*). For details, see Appendix C.
   b. **Subsequent registration with NN:**
      - Datanode sends versionRequest to namenode
      - Namenode sends in response NamespaceInfo that includes (*LayoutVersion*, *NamespaceID*, *BlockPoolID, ctime*, BuildVersion, DistributedUpgradeVersion).
      - If the *ClusterID* of the datanode does not match the one in the response, the datanode does not register with the namenode and prints a warning in the log.
      - For the given NamespaceID, if the *BlockPoolID* on the datanode does not match the BlockID in response, the datanode does not register with the namenode.
      - If all the IDs match, the datanode sends registration request with *BlockPoolID*, NamespaceID and *ClusterID*.
      - If the namenode sees any mismatch in IDs, it rejects the registration request from the datanode.
4. DN stores *BlockPoolID* in the version file (if it does not have it already). If a datanode already has the *BlockPoolID*, then it compares and stops registering with the namenode, if they do not match.

Appendix E – Future Improvements

This section captures further improvements to HDFS that was discussed during the design of this feature. These changes are related, but outside the scope of this feature.

HDFS Layout Version improvements:
Currently HDFS cluster uses a single layout version across both the namenode and datanodes. Any change in layout version, results in upgrade of both the namenode and the datanodes. This should be split into two separate layout versions. This helps in upgrade decision that is local to a node, independent of other nodes.

Decommissioning improvement:
Currently HDFS uses *excludes* files for tracking list of datanodes that are disallowed from joining the cluster. The same file is also used for triggering decommissioning datanodes in the cluster. Adding a registered datanode to *excludes* file results in decommissioning of that node.

A separate file for tracking nodes to be decommissioned is needed. This will remove the overloaded semantics of *excludes* file. Also a decommissioned node is not allowed to register with the namenode, during startup. A cluster restart during decommissioning results in such nodes joining the cluster and could result in corrupt files. This data loss can be prevented by allowing decommissioned datanodes to join the cluster as read-only replicas.