Hortonworks Data Platform

HDFS Administration

(August 29, 2016)

docs.cloudera.com

Hortonworks Data Platform: HDFS Administration

The Hortonworks Data Platform, powered by Apache Hadoop, is a massively scalable and 100% open source platform for storing, processing and analyzing large volumes of data. It is designed to deal with data from many sources and formats in a very quick, easy and cost-effective manner. The Hortonworks Data Platform consists of the essential set of Apache Hadoop projects including MapReduce, Hadoop Distributed File System (HDFS), HCatalog, Pig, Hive, HBase, ZooKeeper and Ambari. Hortonworks is the major contributor of code and patches to many of these projects. These projects have been integrated and tested as part of the Hortonworks Data Platform release process and installation and configuration tools have also been included.

Unlike other providers of platforms built using Apache Hadoop, Hortonworks contributes 100% of our code back to the Apache Software Foundation. The Hortonworks Data Platform is Apache-licensed and completely open source. We sell only expert technical support, training and partner-enablement services. All of our technology is, and will remain, free and open source.

Please visit the Hortonworks Data Platform page for more information on Hortonworks technology. For more information on Hortonworks services, please visit either the Support or Training page. Feel free to contact us directly to discuss your specific needs.

Table of Contents

List of Tables

1. ACLs on HDFS

This guide describes how to use Access Control Lists (ACLs) on the Hadoop Distributed File System (HDFS). ACLs extend the HDFS permission model to support more granular file access based on arbitrary combinations of users and groups.

1.1. Configuring ACLs on HDFS

Only one property needs to be specified in the hdfs-site.xml file in order to enable ACLs on HDFS:

- **dfs.namenode.acls.enabled**

  Set this property to "true" to enable support for ACLs. ACLs are disabled by default. When ACLs are disabled, the NameNode rejects all attempts to set an ACL.

  Example:

```
<property>
  <name>dfs.namenode.acls.enabled</name>
  <value>true</value>
</property>
```

1.2. Using CLI Commands to Create and List ACLs

Two new sub-commands are added to FsShell: `setfacl` and `getfacl`. These commands are modeled after the same Linux shell commands, but fewer flags are implemented. Support for additional flags may be added later if required.

- **setfacl**

  Sets ACLs for files and directories.

  Example:

```
-setfacl [-bkR] {-m|-x} <acl_spec> <path>
```

```
-setfacl --set <acl_spec> <path>
```

  Options:

Table 1.1. ACL Options

Option	Description
`-b`	Remove all entries, but retain the base ACL entries. The entries for User, Group, and Others are retained for compatbility with Permission Bits.
`-k`	Remove the default ACL.
`-R`	Apply operations to all files and directories recursively.
`-m`	Modify the ACL. New entires are added to the ACL, and existing entries are retained.
`-x`	Remove the specified ACL entires. All other ACL entries are retained.

Option	Description
`--set`	Fully replace the ACL and discard all existing entries. The acl_spec must include entries for User, Group, and Others for compatibility with Permission Bits.
`<acl_spec>`	A comma-separated list of ACL entries.
`<path>`	The path to the file or directory to modify.

Examples:

```
hdfs dfs -setfacl -m user:hadoop:rw- /file
hdfs dfs -setfacl -x user:hadoop /file
hdfs dfs -setfacl -b /file
hdfs dfs -setfacl -k /dir
hdfs dfs -setfacl --set user::rw-,user:hadoop:rw-,group::r--,other::r-- /
file
hdfs dfs -setfacl -R -m user:hadoop:r-x /dir
hdfs dfs -setfacl -m default:user:hadoop:r-x /dir
```

Exit Code:

Returns 0 on success and non-zero on error.

- **getfacl**

  Displays the ACLs of files and directories. If a directory has a default ACL, `getfacl` also displays the default ACL.

  Usage:

  ```
  -getfacl [-R] <path>
  ```

  Options:

  Table 1.2. getfacl Options

Option	Description
`-R`	List the ACLs of all files and directories recursively.
`<path>`	The path to the file or directory to list.

  Examples:

  ```
  hdfs dfs -getfacl /file
  hdfs dfs -getfacl -R /dir
  ```

  Exit Code:

  Returns 0 on success and non-zero on error.

1.3. ACL Examples

Before the implementation of Access Control Lists (ACLs), the HDFS permission model was equivalent to traditional UNIX Permission Bits. In this model, permissions for each file or directory are managed by a set of three distinct user classes: Owner, Group, and Others. There are three permissions for each user class: Read, Write, and Execute. Thus, for any file system object, its permissions can be encoded in 3*3=9 bits. When a user attempts to access

a file system object, HDFS enforces permissions according to the most specific user class applicable to that user. If the user is the owner, HDFS checks the Owner class permissions. If the user is not the owner, but is a member of the file system object's group, HDFS checks the Group class permissions. Otherwise, HDFS checks the Others class permissions.

This model can sufficiently address a large number of security requirements. For example, consider a sales department that would like a single user – Bruce, the department manager – to control all modifications to sales data. Other members of the sales department need to view the data, but must not be allowed to modify it. Everyone else in the company (outside of the sales department) must not be allowed to view the data. This requirement can be implemented by running chmod 640 on the file, with the following outcome:

```
-rw-r-----1 brucesales22K Nov 18 10:55 sales-data
```

Only Bruce can modify the file, only members of the sales group can read the file, and no one else can access the file in any way.

Suppose that new requirements arise. The sales department has grown, and it is no longer feasible for Bruce to control all modifications to the file. The new requirement is that Bruce, Diana, and Clark are allowed to make modifications. Unfortunately, there is no way for Permission Bits to address this requirement, because there can be only one owner and one group, and the group is already used to implement the read-only requirement for the sales team. A typical workaround is to set the file owner to a synthetic user account, such as "salesmgr," and allow Bruce, Diana, and Clark to use the "salesmgr" account via sudo or similar impersonation mechanisms. The drawback with this workaround is that it forces complexity onto end-users, requiring them to use different accounts for different actions.

Now suppose that in addition to the sales staff, all executives in the company need to be able to read the sales data. This is another requirement that cannot be expressed with Permission Bits, because there is only one group, and it is already used by sales. A typical workaround is to set the file's group to a new synthetic group, such as "salesandexecs," and add all users of "sales" and all users of "execs" to that group. The drawback with this workaround is that it requires administrators to create and manage additional users and groups.

Based on the preceding examples, you can see that it can be awkward to use Permission Bits to address permission requirements that differ from the natural organizational hierarchy of users and groups. The advantage of using ACLs is that it enables you to address these requirements more naturally, in that for any file system object, multiple users and multiple groups can have different sets of permissions.

Example 1: Granting Access to Another Named Group

To address one of the issues raised in the preceding section, we will set an ACL that grants Read access to sales data to members of the "execs" group.

- Set the ACL:

```
> hdfs dfs -setfacl -m group:execs:r-- /sales-data
```

- Run **getfacl** to check the results:

```
> hdfs dfs -getfacl /sales-data
# file: /sales-data
# owner: bruce
```

```
# group: sales
user::rw-
group::r--
group:execs:r--
mask::r--
other::---
```

- If we run the "ls" command, we see that the listed permissions have been appended with a plus symbol (+) to indicate the presence of an ACL. The plus symbol is appended to the permissions of any file or directory that has an ACL.

```
> hdfs dfs -ls /sales-data
Found 1 items
-rw-r-----+ 3 bruce sales 0 2014-03-04 16:31 /sales-data
```

The new ACL entry is added to the existing permissions defined by the Permission Bits. As the file owner, Bruce has full control. Members of either the "sales" group or the "execs" group have Read access. All others do not have access.

Example 2: Using a Default ACL for Automatic Application to New Children

In addition to an ACL enforced during permission checks, there is also the separate concept of a default ACL. A default ACL can only be applied to a directory – not to a file. Default ACLs have no direct effect on permission checks for existing child files and directories, but instead define the ACL that new child files and directories will receive when they are created.

Suppose we have a "monthly-sales-data" directory that is further subdivided into separate directories for each month. We will set a default ACL to guarantee that members of the "execs" group automatically get access to new subdirectories as they get created each month.

- Set a default ACL on the parent directory:

```
> hdfs dfs -setfacl -m default:group:execs:r-x /monthly-sales-data
```

- Make subdirectories:

```
> hdfs dfs -mkdir /monthly-sales-data/JAN
> hdfs dfs -mkdir /monthly-sales-data/FEB
```

- Verify that HDFS has automatically applied the default ACL to the subdirectories:

```
> hdfs dfs -getfacl -R /monthly-sales-data
# file: /monthly-sales-data
# owner: bruce
# group: sales
user::rwx
group::r-x
other::---
default:user::rwx
default:group::r-x
default:group:execs:r-x
default:mask::r-x
default:other::---

# file: /monthly-sales-data/FEB
# owner: bruce
```

```
# group: sales
user::rwx
group::r-x
group:execs:r-x
mask::r-x
other::---
default:user::rwx
default:group::r-x
default:group:execs:r-x
default:mask::r-x
default:other::---

# file: /monthly-sales-data/JAN
# owner: bruce
# group: sales
user::rwx
group::r-x
group:execs:r-x
mask::r-x
other::---
default:user::rwx
default:group::r-x
default:group:execs:r-x
default:mask::r-x
default:other::---
```

Example 3: Blocking Access to a Sub-Tree for a Specific User

Suppose there is a need to immediately block access to an entire sub-tree for a specific user. Applying a named user ACL entry to the root of that sub-tree is the fastest way to accomplish this without accidentally revoking permissions for other users.

- Add an ACL entry to block user Diana's access to "monthly-sales-data":

```
> hdfs dfs -setfacl -m user:diana:--- /monthly-sales-data
```

- Run **getfacl** to check the results:

```
> hdfs dfs -getfacl /monthly-sales-data
# file: /monthly-sales-data
# owner: bruce
# group: sales
user::rwx
user:diana:---
group::r-x
mask::r-x
other::---
default:user::rwx
default:group::r-x
default:group:execs:r-x
default:mask::r-x
default:other::---
```

It is important to keep in mind the order of evaluation for ACL entries when a user attempts to access a file system object:

- If the user is the file owner, the Owner Permission Bits are enforced.
- Else, if the user has a named user ACL entry, those permissions are enforced.

- Else, if the user is a member of the file's group or any named group in an ACL entry, then the union of permissions for all matching entries are enforced. (The user may be a member of multiple groups.)
- If none of the above are applicable, the Other Permission Bits are enforced.

In this example, the named user ACL entry accomplished our goal because the user is not the file owner and the named user entry takes precedence over all other entries.

1.4. ACLS on HDFS Features

POSIX ACL Implementation

ACLs on HDFS have been implemented with the POSIX ACL model. If you have ever used POSIX ACLs on a Linux file system, the HDFS ACLs work the same way.

Compatibility and Enforcement

HDFS can associate an optional ACL with any file or directory. All HDFS operations that enforce permissions expressed with Permission Bits must also enforce any ACL that is defined for the file or directory. Any existing logic that bypasses Permission Bits enforcement also bypasses ACLs. This includes the HDFS super-user and setting dfs.permissions to "false" in the configuration.

Access Through Multiple User-Facing Endpoints

HDFS supports operations for setting and getting the ACL associated with a file or directory. These operations are accessible through multiple user-facing endpoints. These endpoints include the FsShell CLI, programmatic manipulation through the FileSystem and FileContext classes, WebHDFS, and NFS.

User Feedback: CLI Indicator for ACLs

The plus symbol (+) is appended to the listed permissions of any file or directory with an associated ACL. To view, use the `ls -l` command.

Backward-Compatibility

The implementation of ACLs is backward-compatible with existing usage of Permission Bits. Changes applied via Permission Bits (`chmod`) are also visible as changes in the ACL. Likewise, changes applied to ACL entries for the base user classes (Owner, Group, and Others) are also visible as changes in the Permission Bits. Permission Bit and ACL operations manipulate a shared model, and the Permission Bit operations can be considered a subset of the ACL operations.

Low Overhead

The addition of ACLs will not cause a detrimental impact to the consumption of system resources in deployments that choose not to use ACLs. This includes CPU, memory, disk, and network bandwidth.

Using ACLs does impact NameNode performance. It is therefore recommended that you use Permission Bits, if adequate, before using ACLs.

ACL Entry Limits

The number of entries in a single ACL is capped at a maximum of 32. Attempts to add ACL entries over the maximum will fail with a user-facing error. This is done for two reasons: to simplify management, and to limit resource consumption. ACLs with a very high number of entries tend to become difficult to understand, and may indicate that the requirements are better addressed by defining additional groups or users. ACLs with a very high number of entries also require more memory and storage, and take longer to evaluate on each permission check. The number 32 is consistent with the maximum number of ACL entries enforced by the "ext" family of file systems.

Symlinks

Symlinks do not have ACLs of their own. The ACL of a symlink is always seen as the default permissions (777 in Permission Bits). Operations that modify the ACL of a symlink instead modify the ACL of the symlink's target.

Snapshots

Within a snapshot, all ACLs are frozen at the moment that the snapshot was created. ACL changes in the parent of the snapshot are not applied to the snapshot.

Tooling

Tooling that propagates Permission Bits will not propagate ACLs. This includes the `cp -p` shell command and `distcp -p`.

1.5. Use Cases for ACLs on HDFS

ACLs on HDFS supports the following use cases:

Multiple Users

In this use case, multiple users require Read access to a file. None of the users are the owner of the file. The users are not members of a common group, so it is impossible to use group Permission Bits.

This use case can be addressed by setting an access ACL containing multiple named user entries:

```
ACLs on HDFS supports the following use cases:
```

Multiple Groups

In this use case, multiple groups require Read and Write access to a file. There is no group containing all of the group members, so it is impossible to use group Permission Bits.

This use case can be addressed by setting an access ACL containing multiple named group entries:

```
group:sales:rw-
group:execs:rw-
```

Hive Partitioned Tables

In this use case, Hive contains a partitioned table of sales data. The partition key is "country". Hive persists partitioned tables using a separate subdirectory for each distinct value of the partition key, so the file system structure in HDFS looks like this:

```
user
`-- hive
 `-- warehouse
 `-- sales
 |-- country=CN
 |-- country=GB
 `-- country=US
```

All of these files belong to the "salesadmin" group. Members of this group have Read and Write access to all files. Separate country groups can run Hive queries that only read data for a specific country, such as "sales_CN", "sales_GB", and "sales_US". These groups do not have Write access.

This use case can be addressed by setting an access ACL on each subdirectory containing an owning group entry and a named group entry:

```
country=CN
group::rwx
group:sales_CN:r-x

country=GB
group::rwx
group:sales_GB:r-x

country=US
group::rwx
group:sales_US:r-x
```

Note that the functionality of the owning group ACL entry (the group entry with no name) is equivalent to setting Permission Bits.

Important

Storage-based authorization in Hive does not currently consider the ACL permissions in HDFS. Rather, it verifies access using the traditional POSIX permissions model.

Default ACLs

In this use case, a file system administrator or sub-tree owner would like to define an access policy that will be applied to the entire sub-tree. This access policy must apply not only to the current set of files and directories, but also to any new files and directories that are added later.

This use case can be addressed by setting a default ACL on the directory. The default ACL can contain any arbitrary combination of entries. For example:

```
default:user::rwx
default:user:bruce:rw-
default:user:diana:r--
default:user:clark:rw-
default:group::r--
```

```
default:group:sales::rw-
default:group:execs::rw-
default:others::---
```

It is important to note that the default ACL gets copied from the directory to newly created child files and directories at time of creation of the child file or directory. If you change the default ACL on a directory, that will have no effect on the ACL of the files and subdirectories that already exist within the directory. Default ACLs are never considered during permission enforcement. They are only used to define the ACL that new files and subdirectories will receive automatically when they are created.

Minimal ACL/Permissions Only

HDFS ACLs support deployments that may want to use only Permission Bits and not ACLs with named user and group entries. Permission Bits are equivalent to a minimal ACL containing only 3 entries. For example:

```
user::rw-
group::r--
others::---
```

Block Access to a Sub-Tree for a Specific User

In this use case, a deeply nested file system sub-tree was created as world-readable, followed by a subsequent requirement to block access for a specific user to all files in that sub-tree.

This use case can be addressed by setting an ACL on the root of the sub-tree with a named user entry that strips all access from the user.

For this file system structure:

```
dir1
`-- dir2
 `-- dir3
 |-- file1
 |-- file2
 `-- file3
```

Setting the following ACL on "dir2" blocks access for Bruce to "dir3,""file1,""file2," and "file3":

```
user:bruce:---
```

More specifically, the removal of execute permissions on "dir2" means that Bruce cannot access "dir2", and therefore cannot see any of its children. This also means that access is blocked automatically for any new files added under "dir2". If a "file4" is created under "dir3", Bruce will not be able to access it.

ACLs with Sticky Bit

In this use case, multiple named users or named groups require full access to a shared directory, such as "/tmp". However, Write and Execute permissions on the directory also give users the ability to delete or rename any files in the directory, even files created by other users. Users must be restricted so that they are only allowed to delete or rename files that they created.

This use case can be addressed by combining an ACL with the sticky bit. The sticky bit is existing functionality that currently works with Permission Bits. It will continue to work as expected in combination with ACLs.

2. Archival Storage

This section describes how to use storage policies to assign files and directories to archival storage types.

2.1. Introduction

Archival storage lets you store data on physical media with high storage density and low processing resources.

Implementing archival storage involves the following steps:

1. Shut down the DataNode.
2. Assign the ARCHIVE storage type to DataNodes designed for archival storage.
3. Set HOT, WARM, or COLD storage policies on HDFS files and directories.
4. Restart the DataNode.

If you update a storage policy setting on a file or directory, you must use the HDFS mover data migration tool to actually move blocks as specified by the new storage policy.

2.2. HDFS Storage Types

HDFS storage types can be used to assign data to different types of physical storage media. The following storage types are available:

- **DISK** – Disk drive storage (default storage type)
- **ARCHIVE** – Archival storage (high storage density, low processing resources)
- **SSD** – Solid State Drive
- **RAM_DISK** – DataNode Memory

If no storage type is assigned, DISK is used as the default storage type.

2.3. Storage Policies: Hot, Warm, and Cold

You can store data on DISK or ARCHIVE storage types using the following preconfigured storage policies:

- **HOT**– Used for both storage and compute. Data that is being used for processing will stay in this policy. When a block is HOT, all replicas are stored on DISK. There is no fallback storage for creation, and ARCHIVE is used for replication fallback storage.
- **WARM** - Partially HOT and partially COLD. When a block is WARM, the first replica is stored on DISK, and the remaining replicas are stored on ARCHIVE. The fallback storage for both creation and replication is DISK, or ARCHIVE if DISK is unavailable.

- **COLD** - Used only for storage, with limited compute. Data that is no longer being used, or data that needs to be archived, is moved from HOT storage to COLD storage. When a block is COLD, all replicas are stored on ARCHIVE, and there is no fallback storage for creation or replication.

The following table summarizes these replication policies:

Policy ID	Policy Name	Replica Block Placement (for n replicas)	Fallback storage for creation	Fallback storage for replication
12	HOT (default)	Disk: n	<none>	ARCHIVE
8	WARM	Disk: 1, ARCHIVE: n-1	DISK, ARCHIVE	DISK, ARCHIVE
4	COLD	ARCHIVE: n	<none>	<none>

Note

Currently, storage policies cannot be edited.

2.4. Configuring Archival Storage

Use the following steps to configure archival storage:

1. Shut down the DataNode, using the applicable commands in Controlling HDP Services Manually.

2. Assign the ARCHIVE Storage Type to the DataNode.

   You can use the `dfs.datanode.data.dir` property in the `/etc/hadoop/conf/hdfs-site.xml` file to assign the ARCHIVE storage type to a DataNode.

   The `dfs.datanode.data.dir` property determines where on the local filesystem a DataNode should store its blocks.

   If you specify a comma-delimited list of directories, data will be stored in all named directories, typically on different devices. Directories that do not exist are ignored. You can specify that each directory resides on a different type of storage: DISK, SSD, ARCHIVE, or RAM_DISK.

   To specify a DataNode as DISK storage, specify [DISK] and a local file system path. For example:

   ```
   <property>
     <name>dfs.datanode.data.dir</name>
     <value>[DISK]/grid/1/tmp/data_trunk</value>
   </property>
   ```

   To specify a DataNode as ARCHIVE storage, insert [ARCHIVE] at the beginning of the local file system path. For example:

   ```
   <property>
     <name>dfs.datanode.data.dir</name>
     <value>[ARCHIVE]/grid/1/tmp/data_trunk</value>
   </property>
   ```

3. Set or Get Storage Policies. To set a storage policy on a file or a directory:

```
hdfs storagepolicies -setStoragePolicy <path> <policyName>
```

Arguments:

Table 2.1. Setting Storage Policy

Argument	Description
<path>	The path to a directory or file.
<policyName>	The name of the storage policy.

Example:

```
hdfs storagepolicies -setStoragePolicy /cold1 COLD
```

To get the storage policy of a file or a directory:

```
hdfs storagepolicies -getStoragePolicy <path>
```

Argument:

Table 2.2. Getting Storage Policy

Argument	Description
<path>	The path to a directory or file.

Example:

```
hdfs storagepolicies -getStoragePolicy /cold1
```

4. Start the DataNode, using the applicable commands in Controlling HDP Services Manually.

5. Use Mover to Apply Storage Policies:

When you update a storage policy setting on a file or directory, the new policy is not automatically enforced. You must use the HDFS `mover` data migration tool to actually move blocks as specified by the new storage policy.

The `mover` data migration tool scans the specified files in HDFS and checks to see if the block placement satisfies the storage policy. For the blocks that violate the storage policy, it moves the replicas to a different storage type in order to fulfill the storage policy requirements.

Command:

```
hdfs mover [-p <files/dirs> | -f <local file name>]
```

Arguments:

Table 2.3. HDFS Mover Arguments

Arguments	Description
-p <files/dirs>	Specify a space-separated list of HDFS files/directories to migrate.

Arguments	Description
-f <local file>	Specify a local file containing a list of HDFS files/ directories to migrate.

Note

Note that when both `-p` and `-f` options are omitted, the default path is the root directory.

3. Centralized Cache Management in HDFS

This section provides instructions on setting up and using centralized cache management in HDFS. Centralized cache management enables you to specify paths to directories or files that will be cached by HDFS, thereby improving performance for applications that repeatedly access the same data.

3.1. Overview

Centralized cache management in HDFS is an explicit caching mechanism that enables you to specify paths to directories or files that will be cached by HDFS. The NameNode will communicate with DataNodes that have the desired blocks available on disk, and instruct the DataNodes to cache the blocks in off-heap caches.

Centralized cache management in HDFS offers many significant advantages:

- Explicit pinning prevents frequently used data from being evicted from memory. This is particularly important when the size of the working set exceeds the size of main memory, which is common for many HDFS workloads.
- Because DataNode caches are managed by the NameNode, applications can query the set of cached block locations when making task placement decisions. Co-locating a task with a cached block replica improves read performance.
- When a block has been cached by a DataNode, clients can use a new, more efficient, zero-copy read API. Since checksum verification of cached data is done once by the DataNode, clients can incur essentially zero overhead when using this new API.
- Centralized caching can improve overall cluster memory utilization. When relying on the operating system buffer cache on each DataNode, repeated reads of a block will result in all `n` replicas of the block being pulled into the buffer cache. With centralized cache management, you can explicitly pin only `m` of the `n` replicas, thereby saving `n-m` memory.

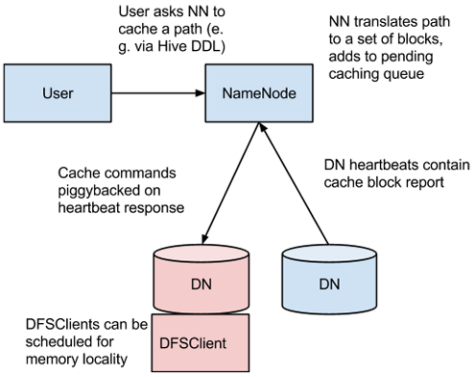
3.2. Caching Use Cases

Centralized cache management is useful for:

- **Files that are accessed repeatedly –** For example, a small fact table in Hive that is often used for joins is a good candidate for caching. Conversely, caching the input of a once-yearly reporting query is probably less useful, since the historical data might only be read once.
- **Mixed workloads with performance SLAs –** Caching the working set of a high priority workload ensures that it does not compete with low priority workloads for disk I/O.

3.3. Caching Architecture

The following figure illustrates the centralized cached management architecture.



In this architecture, the NameNode is responsible for coordinating all of the DataNode off-heap caches in the cluster. The NameNode periodically receives a cache report from each DataNode. The cache report describes all of the blocks cached on the DataNode. The NameNode manages DataNode caches by piggy-backing cache and uncache commands on the DataNode heartbeat.

The NameNode queries its set of Cache Directives to determine which paths should be cached. Cache Directives are persistently stored in the fsimage and edit logs, and can be added, removed, and modified via Java and command-line APIs. The NameNode also stores a set of Cache Pools, which are administrative entities used to group Cache Directives together for resource management, and to enforce permissions.

The NameNode periodically re-scans the namespace and active Cache Directives to determine which blocks need to be cached or uncached, and assigns caching work to DataNodes. Re-scans can also be triggered by user actions such as adding or removing a Cache Directive or removing a Cache Pool.

Cache blocks that are under construction, corrupt, or otherwise incomplete are not cached. If a Cache Directive covers a symlink, the symlink target is not cached.

Currently, caching can only be applied to directories and files.

3.4. Caching Terminology

Cache Directive

A Cache Directive defines the path that will be cached. Paths can point either directories or files. Directories are cached non-recursively, meaning only files in the first-level listing of the directory will be cached.

Cache Directives also specify additional parameters, such as the cache replication factor and expiration time. The replication factor specifies the number of block replicas to cache. If multiple Cache Directives refer to the same file, the maximum cache replication factor is applied.

The expiration time is specified on the command line as a time-to-live (TTL), which represents a relative expiration time in the future. After a Cache Directive expires, it is no longer taken into consideration by the NameNode when making caching decisions.

Cache Pool

A Cache Pool is an administrative entity used to manage groups of Cache Directives. Cache Pools have UNIX-like permissions that restrict which users and groups have access to the pool. Write permissions allow users to add and remove Cache Directives to the pool. Read permissions allow users to list the Cache Directives in a pool, as well as additional metadata. Execute permissions are unused.

Cache Pools are also used for resource management. Cache Pools can enforce a maximum memory limit, which restricts the aggregate number of bytes that can be cached by directives in the pool. Normally, the sum of the pool limits will approximately equal the amount of aggregate memory reserved for HDFS caching on the cluster. Cache Pools also track a number of statistics to help cluster users track what is currently cached, and to determine what else should be cached.

Cache Pools can also enforce a maximum time-to-live. This restricts the maximum expiration time of directives being added to the pool.

3.5. Configuring Centralized Caching

Native Libraries

In order to lock block files into memory, the DataNode relies on native JNI code found in `libhadoop.so`. Be sure to enable JNI if you are using HDFS centralized cache management.

Configuration Properties

Configuration properties for centralized caching are specified in the `hdfs-site.xml` file.

Required Properties

Currently, only one property is required:

- `dfs.datanode.max.locked.memory` This property determines the maximum amount of memory (in bytes) that a DataNode will use for caching. The "locked-in-memory size" ulimit (`ulimit -l`) of the DataNode user also needs to be increased to exceed this parameter (for more details, see the following section on). When setting this value, remember that you will need space in memory for other things as well, such as the DataNode and application JVM heaps, and the operating system page cache. Example:

```
<property>
  <name>dfs.datanode.max.locked.memory</name>
  <value>268435456</value>
</property>
```

Optional Properties

The following properties are not required, but can be specified for tuning.

- `dfs.namenode.path.based.cache.refresh.interval.ms` The NameNode will use this value as the number of milliseconds between subsequent cache path re-scans. By default, this parameter is set to 300000, which is five minutes. Example:

```
<property>
  <name>dfs.namenode.path.based.cache.refresh.interval.ms</name>
```

```
  <value>300000</value>
</property>
```

- `dfs.time.between.resending.caching.directives.ms` The NameNode will use this value as the number of milliseconds between resending caching directives. Example:

```
<property>
  <name>dfs.time.between.resending.caching.directives.ms</name>
  <value>300000</value>
</property>
```

- `dfs.datanode.fsdatasetcache.max.threads.per.volume` The DataNode will use this value as the maximum number of threads per volume to use for caching new data. By default, this parameter is set to 4. Example:

```
<property>
  <name>dfs.datanode.fsdatasetcache.max.threads.per.volume</name>
  <value>4</value>
</property>
```

- `dfs.cachereport.intervalMsec` The DataNode will use this value as the number of milliseconds between sending a full report of its cache state to the NameNode. By default, this parameter is set to 10000, which is 10 seconds. Example:

```
<property>
  <name>dfs.cachereport.intervalMsec</name>
  <value>10000</value>
</property>
```

- `dfs.namenode.path.based.cache.block.map.allocation.percent` The percentage of the Java heap that will be allocated to the cached blocks map. The cached blocks map is a hash map that uses chained hashing. Smaller maps may be accessed more slowly if the number of cached blocks is large. Larger maps will consume more memory. The default value is 0.25 percent. Example:

```
<property>
  <name>dfs.namenode.path.based.cache.block.map.allocation.percent</name>
  <value>0.25</value>
</property>
```

OS Limits

If you get the error "Cannot start datanode because the configured max locked memory size...is more than the datanode's available RLIMIT_MEMLOCK ulimit," that means that the operating system is imposing a lower limit on the amount of memory that you can lock than what you have configured. To fix this, you must adjust the `ulimit -l` value that the DataNode runs with. This value is usually configured in `/etc/security/limits.conf`, but this may vary depending on what operating system and distribution you are using.

You have correctly configured this value when you can run `ulimit - l` from the shell and get back either a higher value than what you have configured or the string "unlimited", which indicates that there is no limit. Typically, `ulimit -l` returns the memory lock limit in kilobytes (KB), but `dfs.datanode.max.locked.memory` must be specified in bytes.

For example, if the value of `dfs.datanode.max.locked.memory` is set to 128000 bytes:

```
<property>
  <name>dfs.datanode.max.locked.memory</name>
  <value>128000</value>
</property>
```

Set the `memlock` (max locked-in-memory address space) to a slightly higher value. For example, to set memlock to 130 KB (130,000 bytes) for the hdfs user, you would add the following line to `/etc/security/limits.conf`.

```
hdfs - memlock 130
```

Note

The information in this section does not apply to deployments on Windows. Windows has no direct equivalent of `ulimit -l`.

3.6. Using Cache Pools and Directives

You can use the Command-Line Interface (CLI) to create, modify, and list Cache Pools and Cache Directives via the `hdfs cacheadmin` subcommand.

Cache Directives are identified by a unique, non-repeating, 64-bit integer ID. IDs will not be reused even if a Cache Directive is removed.

Cache Pools are identified by a unique string name.

You must first create a Cache Pool, and then add Cache Directives to the Cache Pool.

Cache Pool Commands

- `addPool` – Adds a new Cache Pool.

  Usage:

  ```
  hdfs cacheadmin -addPool <name> [-owner <owner>] [-group <group>]
  [-mode <mode>] [-limit <limit>] [-maxTtl <maxTtl>]
  ```

  Options:

Table 3.1. Cache Pool Add Options

Option	Description
`<name`	The name of the pool.
`<owner`	The user name of the owner of the pool. Defaults to the current user.
`<group`	The group that the pool is assigned to. Defaults to the primary group name of the current user.
`<mode`	The UNIX-style permissions assigned to the pool. Permissions are specified in octal (e.g. 0755). Pool permissions are set to 0755 by default.
`<limit`	The maximum number of bytes that can be cached by directives in the pool, in aggregate. By default, no limit is set.
`<maxTtl`	The maximum allowed time-to-live for directives being added to the pool. This can be specified in seconds, minutes, hours, and days (e.g. 120s, 30m, 4h, 2d). Valid

Option	Description
	units are [smhd]. By default, no maximum is set. A value of "never" specifies that there is no limit.

- `modifyPool` – Modifies the metadata of an existing Cache Pool.

  Usage:

```
hdfs cacheadmin -modifyPool <name> [-owner <owner>] [-group <group>]
[-mode <mode>] [-limit <limit>] [-maxTtl <maxTtl>]
```

  Options:

Table 3.2. Cache Pool Modify Options

Option	Description
`name`	The name of the pool to modify.
`owner`	The user name of the owner of the pool.
`group`	The group that the pool is assigned to.
`mode`	The UNIX-style permissions assigned to the pool. Permissions are specified in octal (e.g. 0755).
`limit`	The maximum number of bytes that can be cached by directives in the pool, in aggregate.
`maxTtl`	The maximum allowed time-to-live for directives being added to the pool. This can be specified in seconds, minutes, hours, and days (e.g. 120s, 30m, 4h, 2d). Valid units are [smdh]. By default, no maximum is set. A value of "never" specifies that there is no limit.

- `removePool` – Removes a Cache Pool. This command also "un-caches" paths that are associated with the pool.

  Usage:

```
hdfs cacheadmin -removePool <name>
```

  Options:

Table 3.3. Cache Pool Remove Options

Option	Description
`name`	The name of the Cache Pool to remove.

- `listPools` – Displays information about one or more Cache Pools, such as name, owner, group, permissions, and so on.

  Usage:

```
hdfs cacheadmin -listPools [-stats] [<name>]
```

  Options:

Table 3.4. Cache Pools List Options

Option	Description
`stats`	Displays additional Cache Pool statistics.

Option	Description
name	If specified, lists only the named Cache Pool.

- `help` – Displays detailed information about a command.

  Usage:

  ```
  hdfs cacheadmin -help <command-name>
  ```

  Options:

Table 3.5. Cache Pool Help Options

Option	Description
<command-name	Displays detailed information for the specified command name. If no command name is specified, detailed help is displayed for all commands.

Cache Directive Commands

- `addDirective` – Adds a new Cache Directive.

  Usage:

  ```
  hdfs cacheadmin -addDirective -path <path> -pool <pool-name> [-force]
  [-replication <replication>] [-ttl <time-to-live>]
  ```

  Options:

Table 3.6. Cache Pool Add Directive Options

Option	Description
<path>	The path to the cache directory or file.
<pool-name>	The Cache Pool to which the Cache Directive will be added. You must have Write permission for the Cache Pool in order to add new directives.
<-force>	Skips checking of the Cache Pool resource limits.
<-replication>	The cache replication factor to use. Default setting is 1.
<time-to-live>	How long the directive is valid. This can be specified in minutes, hours and days (e.g. 30m, 4h, 2d). Valid units are [smdh]. A value of "never" indicates a directive that never expires. If unspecified, the directive never expires.

- `removeDirective` – Removes a Cache Directive.

  Usage:

  ```
  hdfs cacheadmin -removeDirective <id>
  ```

  Options:

Table 3.7. Cache Pools Remove Directive Options

Option	Description
<id>	The ID of the Cache Directive to remove. You must have Write permission for the pool that the directive

Option	Description
	belongs to in order to remove it. You can use the `-listDirectives` command to display a list of Cache Directive IDs.

- `removeDirectives` – Removes all of the Cache Directives in a specified path.

  Usage:

  ```
  hdfs cacheadmin -removeDirectives <path>
  ```

  Options:

  Table 3.8. Cache Pool Remove Directives Options

Option	Description
`<path>`	The path of the Cache Directives to remove. You must have Write permission for the pool that the directives belong to in order to remove them. You can use the `-listDirectives` command to display a list of Cache Directives.

- `listDirectives` – Returns a list of Cache Directives.

  Usage:

  ```
  hdfs cacheadmin -listDirectives [-stats] [-path <path>] [-pool <pool>]
  ```

  Options:

  Table 3.9. Cache Pools List Directives Options

Option	Description
`<path>`	Lists only the Cache Directives in the specified path. If there is a Cache Directive in the <path> that belongs to a Cache Pool for which you do not have Read access, it will not be listed.
`<pool>`	Lists on the Cache Directives in the specified Cache Pool.
`<-stats>`	Lists path-based Cache Directive statistics.

4. Configuring HDFS Compression

This section describes how to configure HDFS compression on Linux.

Linux supports `GzipCodec`, `DefaultCodec`, `BZip2Codec`, `LzoCodec`, and `SnappyCodec`. Typically, `GzipCodec` is used for HDFS compression. Use the following instructions to use `GZipCodec`.

- **Option I:** To use `GzipCodec` with a one-time only job:

```
hadoop jar hadoop-examples-1.1.0-SNAPSHOT.jar sort sbr"-Dmapred.compress.
map.output=true" sbr"-Dmapred.map.output.compression.codec=org.apache.
hadoop.io.compress.GzipCodec"sbr "-Dmapred.output.compress=true" sbr"-
Dmapred.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec"sbr
 -outKey org.apache.hadoop.io.Textsbr -outValue org.apache.hadoop.io.Text
 input output
```

- **Option II:** To enable `GzipCodec` as the default compression:

  - Edit the `core-site.xml` file on the NameNode host machine:

```
<property>
  <name>io.compression.codecs</name>
  <value>org.apache.hadoop.io.compress.GzipCodec,
    org.apache.hadoop.io.compress.DefaultCodec,com.hadoop.compression.lzo.
LzoCodec,
    org.apache.hadoop.io.compress.SnappyCodec</value>
<description>A list of the compression codec classes that can be used
    for compression/decompression.</description>
</property>
```

  - Edit the `mapred-site.xml` file on the JobTracker host machine:

```
<property>
  <name>mapreduce.map.output.compress</name>
  <value>true</value>
</property>

<property>
  <name>mapreduce.map.output.compress.codec</name>
  <value>org.apache.hadoop.io.compress.GzipCodec</value>
</property>

<property>
  <name>mapreduce.output.fileoutputformat.compress.type</name>
  <value>BLOCK</value>
</property>
```

  - (Optional) - Enable the following two configuration parameters to enable job output compression. Edit the `mapred-site.xml` file on the Resource Manager host machine:

```
<property>
  <name>mapreduce.output.fileoutputformat.compress</name>
  <value>true</value>
</property>

<property>
  <name>mapreduce.output.fileoutputformat.compress.codec</name>
  <value>org.apache.hadoop.io.compress.GzipCodec</value>
</property>
```

- Restart the cluster using the applicable commands in Controlling HDP Services Manually.

5. Configuring Rack Awareness On HDP

Use the following instructions to configure rack awareness on an HDP cluster.

5.1. Create a Rack Topology Script

Topology scripts are used by Hadoop to determine the rack location of nodes. This information is used by Hadoop to replicate block data to redundant racks.

1. Create a topology script and data file. The topology script must be executable.

Sample Topology Script Named rack-topology.sh

```
#!/bin/bash

# Adjust/Add the property "net.topology.script.file.name"
# to core-site.xml with the "absolute" path the this
# file. ENSURE the file is "executable".

# Supply appropriate rack prefix
RACK_PREFIX=default

# To test, supply a hostname as script input:
if [ $# -gt 0 ]; then

CTL_FILE=${CTL_FILE:-"rack_topology.data"}

HADOOP_CONF=${HADOOP_CONF:-"/etc/hadoop/conf"}

if [ ! -f ${HADOOP_CONF}/${CTL_FILE} ]; then
 echo -n "/$RACK_PREFIX/rack "
 exit 0
fi

while [ $# -gt 0 ] ; do
 nodeArg=$1
 exec< ${HADOOP_CONF}/${CTL_FILE}
 result=""
 while read line ; do
 ar=( $line )
 if [ "${ar[0]}" = "$nodeArg" ] ; then
 result="${ar[1]}"
 fi
 done
 shift
 if [ -z "$result" ] ; then
 echo -n "/$RACK_PREFIX/rack "
 else
 echo -n "/$RACK_PREFIX/rack_$result "
 fi
done

else
 echo -n "/$RACK_PREFIX/rack "
fi
```

Sample Topology Data File Named rack_topology.data

```
# This file should be:
# - Placed in the /etc/hadoop/conf directory
# - On the Namenode (and backups IE: HA, Failover, etc)
# - On the Job Tracker OR Resource Manager (and any Failover JT's/RM's)
# This file should be placed in the /etc/hadoop/conf directory.

# Add Hostnames to this file. Format <host ip> <rack_location>
192.168.2.10 01
192.168.2.11 02
192.168.2.12 03
```

2. Copy both of these files to the `/etc/hadoop/conf` directory on all cluster nodes.
3. Run the `rack-topology.sh` script to ensure that it returns the correct rack information for each host.

5.2. Add the Topology Script Property to core-site.xml

1. Stop HDFS using the applicable commands in the "Controlling HDP Services Manually" section of Installing HDP Manually
2. Add the following property to `core-site.xml`:

```
<property>
  <name>net.topology.script.file.name</name>
  <value>/etc/hadoop/conf/rack-topology.sh</value>
</property>
```

By default the topology script will process up to 100 requests per invocation. You can also specify a different number of requests with the `net.topology.script.number.args` property. For example:

```
<property>
  <name>net.topology.script.number.args</name>
  <value>75</value>
</property>
```

5.3. Restart HDFS and MapReduce

Restart HDFS and MapReduce using the applicable commands in Controlling HDP Services Manually.

5.4. Verify Rack Awareness

After the services have started, you can use the following methods to verify that rack awareness has been activated:

1. Look in the NameNode logs located in `/var/log/hadoop/hdfs/`. For example: `hadoop-hdfs-namenode-sandbox.log`. You should see an entry like this:

```
014-01-13 15:58:08,495 INFO org.apache.hadoop.net.NetworkTopology: Adding
```

```
                        a new node: /rack01/<ipaddress>
```

2. The Hadoop `fsck` command should return something like the following (if there are two racks):

```
Status: HEALTHY  Total size: 123456789 B  Total dirs: 0  Total files: 1
Total blocks (validated): 1 (avg. block size 123456789 B)
Minimally replicated blocks: 1 (100.0 %)  Over-replicated blocks: 0 (0.0 %)
 Under-replicated blocks: 0 (0.0 %)  Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 3  Average block replication: 3.0  Corrupt
blocks: 0  Missing replicas: 0 (0.0 %)  Number of data-nodes: 40  Number of
racks: 2  FSCK ended at Mon Jan 13 17:10:51 UTC 2014 in 1 milliseconds
```

3. The Hadoop `dfsadmin -report` command will return a report that includes the rack name next to each machine. The report should look something like the following excerpted example:

```
[bsmith@hadoop01 ~]$ sudo -u hdfs hadoop dfsadmin -report
Configured Capacity: 19010409390080 (17.29 TB) Present Capacity:
18228294160384 (16.58 TB) DFS Remaining: 5514620928000 (5.02 TB) DFS
Used: 12713673232384 (11.56 TB) DFS Used%: 69.75% Under replicated blocks:
181 Blocks with corrupt replicas: 0  Missing blocks: 0
-------------------------------------------------  Datanodes available:
5 (5 total, 0 dead)  Name: 192.168.90.231:50010 (h2d1.hdp.local) Hostname:
h2d1.hdp.local Rack: /default/rack_02 Decommission Status : Normal
Configured Capacity: 15696052224 (14.62 GB) DFS Used: 314380288
(299.82 MB) Non DFS Used: 3238612992 (3.02 GB) DFS Remaining: 12143058944
(11.31 GB) DFS Used%: 2.00% DFS Remaining%: 77.36%
Configured Cache Capacity: 0 (0 B) Cache Used: 0 (0 B) Cache Remaining: 0
(0 B) Cache Used%: 100.00% Cache Remaining%: 0.00% Last contact: Thu Jun 12
11:39:51 EDT 2014
```

6. Customizing HDFS

6.1. Customize the HDFS Home Directory

By default, the HDFS home directory is set to `/user/<user_name>`. You can use the `dfs.user.home.base.dir` property to customize the HDFS home directory.

1. In the `hdfs-site.xml` file, set the following property:

```
<property>
  <name>dfs.user.home.base.dir</name>
  <value>/user</value>
  <description>Base directory of user home.</description>
</property>
```

   Where `<value>` is the path to the new home directory.

6.2. Set the Size of the NameNode Edits Directory

You can use the following `hdfs-site.xml` properties to control the size of the directory that holds the NameNode `edits` directory.

- `dfs.namenode.num.checkpoints.retained` – The number of image checkpoint files that are retained in storage directories. All edit logs necessary to recover an up-to-date namespace from the oldest retained checkpoint are also retained.
- `dfs.namenode.num.extra.edits.retained` – The number of extra transactions that should be retained beyond what is minimally necessary for a NameNode restart. This can be useful for audit purposes, or for an HA setup where a remote Standby Node may have been offline for some time and require a longer backlog of retained edits in order to start again.

7. Hadoop Archives

The Hadoop Distributed File System (HDFS) is designed to store and process large data sets, but HDFS can be less efficient when storing a large number of small files. When there are many small files stored in HDFS, these small files occupy a large portion of the namespace. As a result, disk space is under-utilized because of the namespace limitation.

Hadoop Archives (HAR) can be used to address the namespace limitations associated with storing many small files. A Hadoop Archive packs small files into HDFS blocks more efficiently, thereby reducing NameNode memory usage while still allowing transparent access to files. Hadoop Archives are also compatible with MapReduce, allowing transparent access to the original files by MapReduce jobs.

7.1. Introduction

The Hadoop Distributed File System (HDFS) is designed to store and process large (terabytes) data sets. For example, a large production cluster may have 14 PB of disk space and store 60 million files.

However, storing a large number of small files in HDFS is inefficient. A file is generally considered to be "small" when its size is substantially less than the HDFS block size, which is 256 MB by default in HDP. Files and blocks are name objects in HDFS, meaning that they occupy namespace (space on the NameNode). The namespace capacity of the system is therefore limited by the physical memory of the NameNode.

When there are many small files stored in the system, these small files occupy a large portion of the namespace. As a consequence, the disk space is underutilized because of the namespace limitation. In one real-world example, a production cluster had 57 million files less than 256 MB in size, with each of these files taking up one block on the NameNode. These small files used up 95% of the namespace but occupied only 30% of the cluster disk space.

Hadoop Archives (HAR) can be used to address the namespace limitations associated with storing many small files. HAR packs a number of small files into large files so that the original files can be accessed transparently (without expanding the files).

HAR increases the scalability of the system by reducing the namespace usage and decreasing the operation load in the NameNode. This improvement is orthogonal to memory optimization in the NameNode and distributing namespace management across multiple NameNodes.

Hadoop Archive is also compatible with MapReduce — it allows parallel access to the original files by MapReduce jobs.

7.2. Hadoop Archive Components

HAR Format Data Model

The Hadoop Archive data format has the following layout:

```
foo.har/_masterindex //stores hashes and offsets
foo.har/_index //stores file statuses
foo.har/part-[1..n] //stores actual file data
```

The file data is stored in multipart files, which are indexed in order to retain the original separation of data. Moreover, the file parts can be accessed in parallel by MapReduce programs. The index files also record the original directory tree structures and file status.

HAR File System

Most archival systems, such as tar, are tools for archiving and de-archiving. Generally, they do not fit into the actual file system layer and hence are not transparent to the application writer in that the archives must be expanded before use.

The Hadoop Archive is integrated with the Hadoop file system interface. The `HarFileSystem` implements the `FileSystem` interface and provides access via the `har://` scheme. This exposes the archived files and directory tree structures transparently to users. Files in a HAR can be accessed directly without expanding them.

For example, if we have the following command to copy an HDFS file to a local directory:

```
hdfs dfs –get hdfs://namenode/foo/file-1 localdir
```

Suppose a Hadoop Archive `bar.har` is created from the `foo` directory. With the HAR, the command to copy the original file becomes:

```
hdfs dfs –get har://namenode/bar.har/foo/file-1 localdir
```

Users only need to change the URI paths. Alternatively, users may choose to create a symbolic link (from `hdfs://namenode/foo` to `har://namenode/bar.har/foo` in the example above), and then even the URIs do not need to be changed. In either case, `HarFileSystem` will be invoked automatically to provide access to the files in the HAR. Because of this transparent layer, HAR is compatible with the Hadoop APIs, MapReduce, the FS shell command-line interface, and higher-level applications such as Pig, Zebra, Streaming, Pipes, and DistCp.

Hadoop Archiving Tool

Hadoop Archives can be created using the Hadoop archiving tool. The archiving tool uses MapReduce to efficiently create Hadoop Archives in parallel. The tool can be invoked using the command:

```
hadoop archive -archiveName name -p <parent> <src>* <dest>
```

A list of files is generated by traversing the source directories recursively, and then the list is split into map task inputs. Each map task creates a part file (about 2 GB, configurable) from a subset of the source files and outputs the metadata. Finally, a reduce task collects metadata and generates the index files.

7.3. Creating a Hadoop Archive

The Hadoop archiving tool can be invoked using the following command:

```
hadoop archive -archiveName name -p <parent> <src>* <dest>
```

Where `-archiveName` is the name of the archive you would like to create. The archive name should be given a .har extension. The `<parent>` argument is used to specify the relative path to the location where the files are to be archived in the HAR.

Example

```
hadoop archive -archiveName foo.har -p /user/hadoop dir1 dir2 /user/zoo
```

This example creates an archive using `/user/hadoop` as the relative archive directory. The directories `/user/hadoop/dir1` and `/user/hadoop/dir2` will be archived in the `/user/zoo/foo.har` archive.

Archiving does not delete the source files. If you would like to delete the input files after creating an archive to reduce namespace, you must manually delete the source files.

Although the hadoop archive command can be run from the host file system, the archive file is created in the HDFS file system from directories that exist in HDFS. If you reference a directory on the host file system rather than in HDFS, you will get the following error:

```
The resolved paths set is empty. Please check whether the srcPaths exist,
 where srcPaths
 = [</directory/path>]
```

To create the HDFS directories used in the preceding example, use the following series of commands:

```
hdfs dfs -mkdir /user/zoo
hdfs dfs -mkdir /user/hadoop
hdfs dfs -mkdir /user/hadoop/dir1
hdfs dfs -mkdir /user/hadoop/dir2
```

7.4. Looking Up Files in Hadoop Archives

The `hdfs dfs -ls` command can be used to look up files in Hadoop archives. Using the example `/user/zoo/foo.har` archive created in the previous section, use the following command to list the files in the archive:

```
hdfs dfs -ls har:///user/zoo/foo.har/
```

This command returns:

```
har:///user/zoo/foo.har/dir1
har:///user/zoo/foo.har/dir2
```

These archives were created with the following command:

```
hadoop archive -archiveName foo.har -p /user/hadoop dir1 dir2 /user/zoo
```

If you change the command to:

```
hadoop archive -archiveName foo.har -p /user/ hadoop/dir1 hadoop/dir2 /user/
zoo
```

And then run the following command:

```
hdfs dfs -ls -R har:///user/zoo/foo.har
```

The following output is returned:

```
har:///user/zoo/foo.har/hadoop
har:///user/zoo/foo.har/hadoop/dir1
har:///user/zoo/foo.har/hadoop/dir2
```

Note that the modified parent argument causes the files to be archived relative to `/user/` rather than `/user/hadoop`.

7.5. Hadoop Archives and MapReduce

To use Hadoop Archives with MapReduce, you must reference files slightly differently than with the default file system. If you have a Hadoop Archive stored in HDFS in `/user/ zoo/foo.har`, you must specify the input directory as `har:///user/zoo/foo.har` to use it as a MapReduce input. Since Hadoop Archives are exposed as a file system, MapReduce is able to use all of the logical input files in Hadoop Archives as input.

8. JMX Metrics APIs for HDFS Daemons

You can use the following methods to access HDFS metrics using the Java Management Extensions (JMX) APIs.

Use the HDFS Daemon Web Interface

You can access JMX metrics through the web interface of an HDFS daemon. This is the recommended method.

For example, use the following command format to access the NameNode JMX:

```
curl -i http://localhost:50070/jmx
```

You can use the `qry` parameter to fetch only a particular key:

```
curl -i http://localhost:50070/jmx?qry=Hadoop:service=NameNode,name=
NameNodeInfo
```

Directly Access the JMX Remote Agent

This method requires that the JMX remote agent is enabled with a JVM option when starting HDFS services.

For example, the following JVM options in `hadoop-env.sh` are used to enable the JMX remote agent for the NameNode. It listens on port 8004 with SSL disabled. The user name and password are saved in the `mxremote.password` file.

```
export HADOOP_NAMENODE_OPTS="-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.password.file=$HADOOP_CONF_DIR/jmxremote.
password
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.port=8004 $HADOOP_NAMENODE_OPTS"
```

Details about related settings can be found here. You can also use the jmxquery tool to retrieve information through JMX.

Hadoop also has a built-in JMX query tool, `jmxget`. For example:

```
hdfs jmxget -server localhost -port 8004 -service NameNode
```

Note that `jmxget` requires that authentication be disabled, as it does not accept a user name and password.

Using JMX can be challenging for operations personnel who are not familiar with JMX setup, especially JMX with SSL and firewall tunnelling. Therefore, it is generally recommended that you collect JXM information through the web interface of HDFS daemons rather than directly accessing the JMX remote agent.

9. Memory as Storage (Technical Preview)

This chapter describes how to use DataNode memory as storage in HDFS.

Note

This feature is a technical preview and considered under development. Do not use this feature in your production systems. If you have questions regarding this feature, contact Support by logging a case on our Hortonworks Support Portal at https://support.hortonworks.com.

9.1. Introduction

HDFS supports efficient writes of large data sets to durable storage, and also provides reliable access to the data. This works well for batch jobs that write large amounts of persistent data.

Emerging classes of applications are driving use cases for writing smaller amounts of temporary data. Using DataNode memory as storage addresses the use case of applications that want to write relatively small amounts of intermediate data sets with low latency.

Writing block data to memory reduces durability, as data can be lost due to process restart before it is saved to disk. HDFS attempts to save replica data to disk in a timely manner to reduce the window of possible data loss.

DataNode memory is referenced using the RAM_DISK storage type and the LAZY_PERSIST storage policy.

Using DataNode memory as HDFS storage involves the following steps:

1. Shut down the DataNode.
2. Mount a portion of DataNode memory for use by HDFS.
3. Assign the RAM_DISK storage type to the DataNode, and enable short-circuit reads.
4. Set the LAZY_PERSIST storage policy on the HDFS files and directories that will use memory as storage.
5. Restart the DataNode.

If you update a storage policy setting on a file or directory, you must use the HDFS `mover` data migration tool to actually move blocks as specified by the new storage policy.

Memory as storage represents one aspect of YARN resource management capabilities that includes CPU scheduling, CGroups, node labels, and archival storage.

9.2. HDFS Storage Types

HDFS storage types can be used to assign data to different types of physical storage media. The following storage types are available:

- **DISK** – Disk drive storage (default storage type)
- **ARCHIVE** – Archival storage (high storage density, low processing resources)
- **SSD** – Solid State Drive
- **RAM_DISK** – DataNode Memory

If no storage type is assigned, DISK is used as the default storage type.

9.3. The LAZY_PERSIST Memory Storage Policy

You can store data on configured DataNode memory using the LAZY_PERSIST storage policy.

For LAZY_PERSIST, the first replica is stored on RAM_DISK (DataNode memory), and the remaining replicas are stored on DISK. The fallback storage for both creation and replication is DISK.

The following table summarizes these replication policies:

Policy ID	Policy Name	Block Placement (for n replicas)	Fallback storage for creation	Fallback storage for replication
15	LAZY_PERSIST	RAM_DISK: 1, DISK:*n*-1	DISK	DISK

Note

Currently, storage policies cannot be edited.

9.4. Configuring Memory as Storage

Use the following steps to configure DataNode memory as storage:

1. Shut Down the DataNode

Shut down the DataNode using the applicable commands in Controlling HDP Services Manually.

2. Mount a Portion of DataNode Memory for HDFS

To use DataNode memory as storage, you must first mount a portion of the DataNode memory for use by HDFS.

For example, you would use the following commands to allocate 2GB of memory for HDFS storage:

```
sudo mkdir -p /mnt/hdfsramdisk
sudo mount -t tmpfs -o size=2048m tmpfs /mnt/hdfsramdisk
Sudo mkdir -p /usr/lib/hadoop-hdfs
```

3. Assign the RAM_DISK Storage Type and Enable Short-Circuit Reads

Edit the following properties in the `/etc/hadoop/conf/hdfs-site.xml` file to assign the RAM_DISK storage type to DataNodes and enable short-circuit reads.

- The `dfs.name.dir` property determines where on the local filesystem a DataNode should store its blocks. To specify a DataNode as RAM_DISK storage, insert [RAM_DISK] at the beginning of the local file system mount path and add it to the `dfs.name.dir` property.
- To enable short-circuit reads, set the value of `dfs.client.read.shortcircuit` to `true`.

For example:

```
<property>
  <name>dfs.data.dir</name>
  <value>file:///grid/3/aa/hdfs/data/,[RAM_DISK]file:///mnt/hdfsramdisk/</
value>
</property>

<property>
  <name>dfs.client.read.shortcircuit</name>
  <value>true</value>
</property>

<property>
  <name>dfs.domain.socket.path</name>
  <value>/var/lib/hadoop-hdfs/dn_socket</value>
</property>

<property>
  <name>dfs.checksum.type</name>
  <value>NULL</value>
</property>
```

4. Set the LAZY_PERSIST Storage Policy on Files or Directories

Set a storage policy on a file or a directory.

Command:

```
hdfs dfsadmin -setStoragePolicy <path> <policyName>
```

Arguments:

- **<path>** - The path to a directory or file.
- **<policyName>** - The name of the storage policy.

Example:

```
hdfs dfsadmin -setStoragePolicy /memory1 LAZY_PERSIST
```

Get the storage policy of a file or a directory.

Command:

```
hdfs dfsadmin -getStoragePolicy <path>
```

Arguments:

- **<path>** - The path to a directory or file.

Example:

```
hdfs dfsadmin -getStoragePolicy /memory1 LAZY_PERSIST
```

5. Start the DataNode

Start the DataNode using the applicable commands in the Controlling HDP Services Manually section of the HDP Reference Guide.

Using Mover to Apply Storage Policies

When you update a storage policy setting on a file or directory, the new policy is not automatically enforced. You must use the HDFS `mover` data migration tool to actually move blocks as specified by the new storage policy.

The `mover` data migration tool scans the specified files in HDFS and checks to see if the block placement satisfies the storage policy. For the blocks that violate the storage policy, it moves the replicas to the applicable storage type in order to fulfill the storage policy requirements.

Command:

```
hdfs mover [-p <files/dirs> | -f <local file name>]
```

Arguments:

- **-p<files/dirs>** - Specify a space-separated list of HDFS files/directories to migrate.
- **-f<local file>** - Specify a local file list containing a list of HDFS files/directories to migrate.

Note

When both `-p` and `-f` options are omitted, the default path is the root directory.

10. Running DataNodes as Non-Root

This chapter describes how to run DataNodes as a non-root user.

10.1. Introduction

Historically, part of the security configuration for HDFS involved starting the DataNode as the root user, and binding to privileged ports for the server endpoints. This was done to address a security issue whereby if a MapReduce task was running and the DataNode stopped running, it would be possible for the MapReduce task to bind to the DataNode port and potentially do something malicious. The solution to this scenario was to run the DataNode as the root user and use privileged ports. Only the root user can access privileged ports.

You can now use Simple Authentication and Security Layer (SASL) to securely run DataNodes as a non-root user. SASL is used to provide secure communication at the protocol level.

Important

Make sure to execute a migration from using root to start DataNodes to using SASL to start DataNodes in a very specific sequence across the entire cluster. Otherwise, there could be a risk of application downtime.

In order to migrate an existing cluster that used root authentication to start using SASL instead, first ensure that HDP 2.2 or later has been deployed to all cluster nodes as well as any external applications that need to connect to the cluster. Only the HDFS client in versions HDP 2.2 and later can connect to a DataNode that uses SASL for authentication of data transfer protocol, so it is vital that all callers have the correct version before migrating. After HDP 2.2 or later has been deployed everywhere, update the configuration of any external applications to enable SASL. If an HDFS client is enabled for SASL, it can connect successfully to a DataNode running with either root authentication or SASL authentication. Changing configuration for all clients guarantees that subsequent configuration changes on DataNodes will not disrupt the applications. Finally, each individual DataNode can be migrated by changing its configuration and restarting. It is acceptable to temporarily have a mix of some DataNodes running with root authentication and some DataNodes running with SASL authentication during this migration period, because an HDFS client enabled for SASL can connect to both.

10.2. Configuring DataNode SASL

Use the following steps to configure DataNode SASL to securely run a DataNode as a non-root user:

1. Shut Down the DataNode

Shut down the DataNode using the applicable commands in Controlling HDP Services Manually.

2. Enable SASL

Configure the following properties in the `/etc/hadoop/conf/hdfs-site.xml` file to enable DataNode SASL.

The `dfs.data.transfer.protection` property enables DataNode SASL. You can set this property to one of the following values:

- `authentication` – Establishes mutual authentication between the client and the server.
- `integrity` – in addition to authentication, it guarantees that a man-in-the-middle cannot tamper with messages exchanged between the client and the server.
- `privacy` – in addition to the features offered by authentication and integrity, it also fully encrypts the messages exchanged between the client and the server.

In addition to setting a value for the `dfs.data.transfer.protection` property, you must set the `dfs.http.policy` property to `HTTPS_ONLY`. You must also specify ports for the DataNode RPC and HTTP Servers.

Note

For more information on configuring SSL, see the sections "Creating and Managing SSL Certificates" and "Enabling SSL for HDP Components" in the Hadoop Security Guide.

For example:

```
<property>
  <name>dfs.data.transfer.protection</name>
  <value>integrity</value>
</property>

<property>
  <name>dfs.datanode.address</name>
  <value>0.0.0.0:10019</value>
</property>

<property>
  <name>dfs.datanode.http.address</name>
  <value>0.0.0.0:10022</value>
</property>

<property>
  <name>dfs.http.policy</name>
  <value>HTTPS_ONLY</value>
</property>
```

Note

If you are already using the following encryption setting:

`dfs.encrypt.data.transfer=true`

This is similar to:

`dfs.data.transfer.protection=privacy`

These two settings are mutually exclusive, so you should not have both of them set. However, if both are set, `dfs.encrypt.data.transfer` will not be used.

3. Update Environment Settings

Edit the following setting in the `/etc/hadoop/conf/hadoop-env.sh` file, as shown below:

```
#On secure datanodes, user to run the datanode as after dropping privileges
export HADOOP_SECURE_DN_USER=
```

The `export HADOOP_SECURE_DN_USER=hdfs` line enables the legacy security configuration, and must be set to an empty value in order for SASL to be enabled.

4. Start the DataNode

Start the DataNode services using the applicable commands in Controlling HDP Services Manually.

11. Short Circuit Local Reads On HDFS

In HDFS, reads normally go through the DataNode. Thus, when a client asks the DataNode to read a file, the DataNode reads that file off of the disk and sends the data to the client over a TCP socket. "Short-circuit" reads bypass the DataNode, allowing the client to read the file directly. Obviously, this is only possible in cases where the client is co-located with the data. Short-circuit reads provide a substantial performance boost to many applications.

11.1. Prerequisites

To configure short-circuit local reads, you must enable `libhadoop.so`. See Native Libraries for details on enabling this library.

11.2. Configuring Short-Circuit Local Reads on HDFS

To configure short-circuit local reads, add the following properties to the `hdfs-site.xml` file. Short-circuit local reads need to be configured on both the DataNode and the client.

Property Name	Property Value	Description
dfs.client.read.shortcircuit	true	Set this to true to enable short-circuit local reads.
dfs.domain.socket.path	/var/lib/hadoop-hdfs/ dn_socket	The path to the domain socket. Short-circuit reads make use of a UNIX domain socket. This is a special path in the file system that allows the client and the DataNodes to communicate. You will need to set a path to this socket. The DataNode needs to be able to create this path. On the other hand, it should not be possible for any user except the hdfs user or root to create this path. For this reason, paths under */var/run* or */var/lib* are often used. In the file system that allows the client and the DataNodes to communicate. You will need to set a path to this socket. The DataNode needs to be able to create this path. On the other hand, it should not be possible for any user except the hdfs user or root to create this path. For this reason, paths under */var/run* or */var/lib* are often used.
dfs.client.domain.socket.data.traffic	false	This property controls whether or not normal data traffic will be passed through the UNIX domain socket. This feature has not been certified with HDP releases, so it is recommended that you set the value of this property to `false`. Abnormal data traffic will be passed through the UNIX domain socket.
dfs.client.use.legacy.blockreader.local	false	Setting this value to `false` specifies that the new version (based on

Property Name	Property Value	Description
		HDFS-347) of the short-circuit reader is used. This new new short-circuit reader implementation is supported and recommended for use with HDP. Setting this value to `true` would mean that the legacy short-circuit reader would be used.
dfs.datanode.hdfs-blocks-metadata.enabled	true	Boolean which enables back-end DataNode-side support for the experimental `DistributedFileSystem#getFile` `VBlockStorageLocationsAPI`.
dfs.client.file-block-storage-locations.timeout	60	Timeout (in seconds) for the parallel RPCs made in `DistributedFileSystem` `#getFileBlockStorageLocations()`. This property is deprecated but is still supported for backward compatibility
dfs.client.file-block-storage-locations.timeout.millis	60000	Timeout (in milliseconds) for the parallel RPCs made in `DistributedFileSystem` `#getFileBlockStorageLocations()`. This property replaces `dfs.client.file-block-storage-locations.timeout`, and offers a finer level of granularity.
dfs.client.read.shortcircuit. skip.checksum	false	If this configuration parameter is set, short-circuit local reads will skip checksums. This is normally not recommended, but it may be useful for special setups. You might consider using this if you are doing your own checksumming outside of HDFS.
dfs.client.read.shortcircuit. streams.cache.size	256	The DFSClient maintains a cache of recently opened file descriptors. This parameter controls the size of that cache. Setting this higher will use more file descriptors, but potentially provide better performance on workloads involving many seeks.
dfs.client.read.shortcircuit. streams.cache.expiry.ms	300000	This controls the minimum amount of time (in milliseconds) file descriptors need to sit in the client cache context before they can be closed for being inactive for too long.

The XML for these entries:

```
<configuration>

<property>
  <name>dfs.client.read.shortcircuit</name>
```

```
  <value>true</value>
</property>

<property>
  <name>dfs.domain.socket.path</name>
  <value>/var/lib/hadoop-hdfs/dn_socket</value>
</property>

<property>
  <name>dfs.client.domain.socket.data.traffic</name>
  <value>false</value>
</property>

<property>
  <name>dfs.client.use.legacy.blockreader.local</name>
  <value>false</value>
</property>

<property>
  <name>dfs.datanode.hdfs-blocks-metadata.enabled</name>
  <value>true</value>
</property>

<property>
  <name>dfs.client.file-block-storage-locations.timeout.millis</name>
  <value>60000</value>
</property>

<property>
  <name>dfs.client.read.shortcircuit.skip.checksum</name>
  <value>false</value>
</property>

<property>
  <name>dfs.client.read.shortcircuit.streams.cache.size</name>
  <value>256</value>
</property>

<property>
  <name>dfs.client.read.shortcircuit.streams.cache.expiry.ms</name>
  <value>300000</value>
</property>

</configuration>
```

12. Accidental Deletion Protection

This chapter describes accidental deletion protection using HDFS features.

12.1. Preventing Accidental Deletion of Files

You can prevent accidental deletion of files by enabling the `Trash` feature for HDFS. For additional information regarding HDFS trash configuration, see HDFS Architecture.

You might still cause irrecoverable data loss if the `-skipTrash` and `-R` options are accidentally used on directories with a large number of files. You can obtain an additional layer of protection by using the `-safely` option to the **fs shell -rm** command. The **fs shell -rm** command checks the hadoop.shell.safely.delete.limit.num.files property from `core-site.xml` file, even if you specify `-skipTrash`. By specifying the `-safely` option, the **-rm** command requires that you confirm if the number of files to be deleted is greater than the limit specified by the assigned value. The default limit for value is `100`, referring to 100 files.

This confirmation warning is disabled if `value` is set at `0` or the `-safely` is not specified to the **-rm** command.

To enable the hadoop.shell.safely.delete.limit.num.files property, add the following lines to `core-site.xml`:

```
<property>
<name>hadoop.shell.safely.delete.limit.num.files</name>
<value>100</value>
<description>Used by -safely option of hadoop fs shell -rm command to avoid
accidental deletion of large directories.</description>
</property>
```

In the following example, the hadoop.shell.safely.delete.limit.num.files property with an associated value of `10` has been added to `core-site.xml` with `-skipTrash` . In this example, **fs shell -r** prompts deletion of a directory with only `10` files. It does not prompt if `trash` is enabled and `-skipTrash` is not.

```
[ambari-qa@c6405 current]$ hdfs dfs -ls -R /tmp/test1
-rw-r--r--   3 ambari-qa hdfs       2413 2016-10-20 20:57 /tmp/test1/capacity-
scheduler.xml
-rw-r--r--   3 ambari-qa hdfs       4435 2016-10-20 20:57 /tmp/test1/core-
site.xml
-rw-r--r--   3 ambari-qa hdfs       1308 2016-10-20 20:57 /tmp/test1/hadoop-
policy.xml
-rw-r--r--   3 ambari-qa hdfs       8071 2016-10-20 20:57 /tmp/test1/hdfs-
site.xml
-rw-r--r--   3 ambari-qa hdfs       3518 2016-10-20 20:57 /tmp/test1/kms-acls.
xml
-rw-r--r--   3 ambari-qa hdfs       5511 2016-10-20 20:57 /tmp/test1/kms-site.
xml
-rw-r--r--   3 ambari-qa hdfs       7339 2016-10-20 20:57 /tmp/test1/mapred-
site.xml
-rw-r--r--   3 ambari-qa hdfs        884 2016-10-20 20:57 /tmp/test1/ssl-
client.xml
```

```
-rw-r--r--   3 ambari-qa hdfs       1000 2016-10-20 20:57 /tmp/test1/ssl-
server.xml
-rw-r--r--   3 ambari-qa hdfs      20349 2016-10-20 20:57 /tmp/test1/yarn-
site.xml
[ambari-qa@c6405 current]$ hdfs dfs -rm -R /tmp/test1
16/10/20 20:58:37 INFO fs.TrashPolicyDefault: Moved: 'hdfs://c6403.ambari.
apache.org:8020/tmp/test1' to trash at: hdfs://c6403.ambari.apache.org:8020/
user/ambari-qa/.Trash/Current/tmp/test1
```

The following example deletes files without prompting or moving to the trash:

```
[ambari-qa@c6405 current]$ hdfs dfs -ls -R /tmp/test2
-rw-r--r--   3 ambari-qa hdfs       2413 2016-10-20 20:59 /tmp/test2/capacity-
scheduler.xml
-rw-r--r--   3 ambari-qa hdfs       4435 2016-10-20 20:59 /tmp/test2/core-
site.xml
-rw-r--r--   3 ambari-qa hdfs       1308 2016-10-20 20:59 /tmp/test2/hadoop-
policy.xml
-rw-r--r--   3 ambari-qa hdfs       8071 2016-10-20 20:59 /tmp/test2/hdfs-
site.xml
-rw-r--r--   3 ambari-qa hdfs       3518 2016-10-20 20:59 /tmp/test2/kms-acls.
xml
-rw-r--r--   3 ambari-qa hdfs       5511 2016-10-20 20:59 /tmp/test2/kms-site.
xml
-rw-r--r--   3 ambari-qa hdfs       7339 2016-10-20 20:59 /tmp/test2/mapred-
site.xml
-rw-r--r--   3 ambari-qa hdfs        884 2016-10-20 20:59 /tmp/test2/ssl-
client.xml
-rw-r--r--   3 ambari-qa hdfs       1000 2016-10-20 20:59 /tmp/test2/ssl-
server.xml
-rw-r--r--   3 ambari-qa hdfs      20349 2016-10-20 20:59 /tmp/test2/yarn-
site.xml
[ambari-qa@c6405 current]$ hdfs dfs -rm -R -skipTrash /tmp/test2
Deleted /tmp/test2
```

The following example prompts for you to confirm file deletion if the number of files to be deleted is greater than the value specified to `hadoop.shell.safely.delete.limit.num.files`:

```
[ambari-qa@c6405 current]$ hdfs dfs -ls -R /tmp/test3
-rw-r--r--   3 ambari-qa hdfs       2413 2016-10-20 21:00 /tmp/test3/capacity-
scheduler.xml
-rw-r--r--   3 ambari-qa hdfs       4435 2016-10-20 21:00 /tmp/test3/core-
site.xml
-rw-r--r--   3 ambari-qa hdfs       1308 2016-10-20 21:00 /tmp/test3/hadoop-
policy.xml
-rw-r--r--   3 ambari-qa hdfs       8071 2016-10-20 21:00 /tmp/test3/hdfs-
site.xml
-rw-r--r--   3 ambari-qa hdfs       3518 2016-10-20 21:00 /tmp/test3/kms-acls.
xml
-rw-r--r--   3 ambari-qa hdfs       5511 2016-10-20 21:00 /tmp/test3/kms-site.
xml
-rw-r--r--   3 ambari-qa hdfs       7339 2016-10-20 21:00 /tmp/test3/mapred-
site.xml
-rw-r--r--   3 ambari-qa hdfs        884 2016-10-20 21:00 /tmp/test3/ssl-
client.xml
-rw-r--r--   3 ambari-qa hdfs       1000 2016-10-20 21:00 /tmp/test3/ssl-
server.xml
-rw-r--r--   3 ambari-qa hdfs      20349 2016-10-20 21:00 /tmp/test3/yarn-
site.xml
```

```
[ambari-qa@c6405 current]$ hdfs dfs -rm -R -skipTrash -safely /tmp/test3
Proceed deleting 10 files? (Y or N) N
Delete aborted at user request.
```

Warning

Using the `-skipTrash` option without the `-safely` option is not recommended, as files will be deleted immediately and without warning.

13. WebHDFS Administrator Guide

Use the following instructions to set up WebHDFS:

1. Set up WebHDFS. Add the following property to the `hdfs-site.xml` file

```
<property>
  <name>dfs.webhdfs.enabled</name>
  <value>true</value>
</property>
```

If running a secure cluster, follow the steps listed below.

1. Create an HTTP service user principal using the command given below:

```
kadmin: addprinc -randkey HTTP/$<Fully_Qualified_Domain_Name>@$<Realm_Name>.
COM
```

where:

Create an HTTP service user principal using the command given below:

```
kadmin: addprinc -randkey HTTP/$<Fully_Qualified_Domain_Name>@$<Realm_Name>.
COM
```

where:

- `Fully_Qualified_Domain_Name`: Host where NameNode is deployed
- `Realm_Name`: Name of your Kerberos realm

2. Create keytab files for the HTTP principals.

```
kadmin: xst -norandkey -k /etc/security/spnego.service.keytab HTTP/
$<Fully_Qualified_Domain_Name>
```

3. Verify that the keytab file and the principal are associated with the correct service.

```
klist –k -t /etc/security/spnego.service.keytab
```

4. Add the following properties to the `hdfs-site.xml`file.

```
<property>
  <name>dfs.web.authentication.kerberos.principal</name>
  <value>HTTP/$<Fully_Qualified_Domain_Name>@$<Realm_Name>.COM</value>
</property>
<property>
  <name>dfs.web.authentication.kerberos.keytab</name>
  <value>/etc/security/spnego.service.keytab</value>
</property>
```

where:

- `Fully_Qualified_Domain_Name`: Host where NameNode is deployed
- `Realm_Name`: Name of your Kerberos realm

5. Restart the NameNode and DataNode services using the applicable commands in Controlling HDP Services Manually.

14. Backing Up HDFS Metadata

This chapter focuses on understanding and backing up HDFS metadata.

14.1. Introduction to HDFS Metadata Files and Directories

HDFS metadata represents the structure of HDFS directories and files in a tree. It also includes the various attributes of directories and files, such as ownership, permissions, quotas, and replication factor.

14.1.1. Files and Directories

Warning

Do not attempt to modify metadata directories or files. Unexpected modifications can cause HDFS downtime, or even permanent data loss. This information is provided for educational purposes only.

Persistence of HDFS metadata broadly consist of two categories of files:

fsimage — Contains the complete state of the file system at a point in time. Every file system modification is assigned a unique, monotonically increasing transaction ID. An `fsimage` file represents the file system state after all modifications up to a specific transaction ID.

edits file — Contains a log that lists each file system change (file creation, deletion or modification) that was made after the most recent `fsimage`.

Checkpointing is the process of merging the content of the most recent `fsimage`, with all `edits` applied after that `fsimage` is merged, to create a new `fsimage`. Checkpointing is triggered automatically by configuration policies or manually by HDFS administration commands.

14.1.1.1. Namenodes

The following example shows an HDFS metadata directory taken from a NameNode. This shows the output of running the tree command on the metadata directory, which is configured by setting dfs.namenode.name.dir in `hdfs-site.xml`.

```
data/dfs/name
### current#
### VERSION#
### edits_0000000000000000001-0000000000000000007
# ### edits_0000000000000000008-0000000000000000015
# ### edits_0000000000000000016-0000000000000000022
# ### edits_0000000000000000023-0000000000000000029
# ### edits_0000000000000000030-0000000000000000030
# ### edits_0000000000000000031-0000000000000000031
# ### edits_inprogress_0000000000000000032
```

```
# ### fsimage_0000000000000000030
# ### fsimage_0000000000000000030.md5
# ### fsimage_0000000000000000031
# ### fsimage_0000000000000000031.md5
# ### seen_txid
### in_use.lock
```

In this example, the same directory has been used for both `fsimage` and `edits`. Alternative configuration options are available that allow separating `fsimage` and `edits` into different directories. Each file within this directory serves a specific purpose in the overall scheme of metadata persistence:

VERSION	Text file that contains the following elements:	
	layoutVersion	Version of the HDFS metadata format. When you add new features that require a change to the metadata format, you change this number. An HDFS upgrade is required when the current HDFS software uses a layout version that is newer than the current one.
	namespaceID/clusterID/ blockpoolID	Unique identifiers of an HDFS cluster. These identifiers are used to prevent DataNodes from registering accidentally with an incorrect NameNode that is part of a different cluster. These identifiers also are particularly important in a federated deployment. Within a federated deployment, there are multiple NameNodes working independently. Each NameNode serves a unique portion of the namespace

		(`namespaceID`) and manages a unique set of blocks (`blockpoolID`). The `clusterID` ties the whole cluster together as a single logical unit. This structure is the same across all nodes in the cluster.
	storageType	Always `NAME_NODE` for the NameNode, and never `JOURNAL_NODE`.
	cTime	Creation time of file system state. This field is updated during HDFS upgrades.
edits_start transaction ID-end transaction ID		Finalized and unmodifiable edit log segments. Each of these files contains all of the edit log transactions in the range defined by the file name. In an High Availability deployment, the standby can only read up through the finalized log segments. The standby NameNode is not up-to-date with the current edit log in progress. When an HA failover happens, the failover finalizes the current log segment so that it is completely caught up before switching to active.
fsimage_end transaction ID		Contains the complete metadata image up through . Each `fsimage` file also has a corresponding .md5 file containing a MD5 checksum, which HDFS uses to guard against disk corruption.
seen_txid		Contains the last transaction ID of the last checkpoint (merge of `edits` into an `fsimage`) or edit log roll (finalization of current `edits_inprogress` and creation of a new one). This is not the last transaction ID accepted by the NameNode. The file is not updated on every transaction, only on a checkpoint or an edit log roll. The purpose of this file is to try to identify if `edits` are missing during startup. It is possible to configure the NameNode to use separate directories for `fsimage` and `edits` files. If the `edits` directory accidentally gets deleted, then all transactions since the last checkpoint would go away, and the NameNode starts up using just `fsimage` at an old state. To guard against this, NameNode startup also checks `seen_txid` to verify

	that it can load transactions at least up through that number. It aborts startup if it cannot verify the load transactions.
in_use.lock	Lock file held by the NameNode process, used to prevent multiple NameNode processes from starting up and concurrently modifying the directory.

14.1.1.2. Journalnodes

In an HA deployment, `edits` are logged to a separate set of daemons called JournalNodes. A JournalNode's metadata directory is configured by setting dfs.journalnode.edits.dir. The JournalNode contains a VERSION file, multiple `edits__files` and an `edits_inprogress_`, just like the NameNode. The JournalNode does not have `fsimage` files or `seen_txid`. In addition, it contains several other files relevant to the HA implementation. These files help prevent a split-brain scenario, in which multiple NameNodes could think they are active and all try to write `edits`.

committed-txid	Tracks last transaction ID committed by a NameNode.
last-promised-epoch	Contains the "epoch," which is a monotonically increasing number. When a new NameNode, starts as active, it increments the epoch and presents it in calls to the JournalNode. This scheme is the NameNode's way of claiming that it is active and requests from another NameNode, presenting a lower epoch, must be ignored.
last-writer-epoch	Contains the epoch number associated with the writer who last actually wrote a transaction.
paxos	Specifies the directory that temporary files used in the implementation of the Paxos distributed consensus protocol. This directory often appears as empty.

14.1.1.3. Datanodes

Although DataNodes do not contain metadata about the directories and files stored in an HDFS cluster, they do contain a small amount of metadata about the DataNode itself and its relationship to a cluster. This shows the output of running the tree command on the DataNode's directory, configured by setting dfs.datanode.data.dir in hdfs-site.xml.

```
data/dfs/data/
### current
# ### BP-1079595417-192.168.2.45-1412613236271
# # ### current
# # # ### VERSION
# # # ### finalized
# # # # ### subdir0# # # # ### subdir1
# # # # ### blk_1073741825
# # # # ### blk_1073741825_1001.meta
# # # ### lazyPersist
# # # ### rbw
# # ### dncp_block_verification.log.curr
# # ### dncp_block_verification.log.prev
# # ### tmp
```

```
# ### VERSION
### in_use.lock
```

The purpose of these files are as follows:

BP-random integer-NameNode-IP address-creation time	Top level directory for datanodes. The naming convention for this directory is significant and constitutes a form of cluster metadata. The name is a block pool ID. “BP” stands for “block pool,” the abstraction that collects a set of blocks belonging to a single namespace. In the case of a federated deployment, there are multiple “BP” sub-directories, one for each block pool. The remaining components form a unique ID: a random integer, followed by the IP address of the NameNode that created the block pool, followed by creation time.
VERSION	Text file containing multiple properties, such as layoutVersion, clusterId and cTime, which is much like the NameNode and JournalNode. There is a VERSION file tracked for the entire DataNode as well as a separate VERSION file in each block pool sub-directory. In addition to the properties already discussed earlier, the DataNode’s VERSION files also contain: storageType — `storageType` field is set to `DATA_NODE`. blockpoolID — Repeats the block pool ID information encoded into the sub-directory name.
finalized/rbw	Both `finalized` and `rbw` contain a directory structure for block storage. This holds numerous block files, which contain HDFS file data and the corresponding .meta files, which contain checksum information. `rbw` stands for “replica being written”. This area contains blocks that are still being written to by an HDFS client. The finalized sub-directory contains blocks that are not being written to by a client and have been completed.
lazyPersist	HDFS is incorporating a new feature to support writing transient data to memory, followed by lazy persistence to disk in the background. If this feature is in use, then a lazyPersist sub-directory is present and used for lazy persistence of in-memory blocks to disk. We’ll cover this exciting new feature in greater detail in a future blog post.
scanner.cursor	File to which the "cursor state" is saved. The DataNode runs a block scanner which periodically does checksum verification of each block file on disk.

	This scanner maintains a "cursor," representing the last block to be scanned in each block pool slice on the volume, and called the "cursor state."
in_use.lock	Lock file held by the DataNode process, used to prevent multiple DataNode processes from starting up and concurrently modifying the directory.

14.1.2. HDFS Commands

You can use the following HDFS commands to manipulate metadata files and directories:

hdfs namenode	Automatically saves a new checkpoint at NameNode startup. As stated earlier, checkpointing is the process of merging any outstanding edit logs with the latest `fsimage`, saving the full state to a new `fsimage` file, and rolling `edits`. Rolling `edits` means finalizing the current `edits_inprogress` and starting a new one.
hdfs dfsadmin -safemode enter, hdfs dfsadmin -saveNamespace	Saves a new checkpoint (similar to restarting NameNode) while the NameNode process remains running. The NameNode must be in safe mode, and all attempted write activity fails while this command runs.
hdfs dfsadmin -rollEdits	Manually rolls `edits`. Safe mode is not required. This can be useful if a standby NameNode is lagging behind the active NameNode and you want it to get caught up more quickly. The standby NameNode can only read finalized edit log segments, not the current in progress `edits` file.
hdfs dfsadmin -fetchImage	Downloads the latest `fsimage` from the NameNode. This can be helpful for a remote backup type of scenario.

14.1.2.1. Configuration Properties

dfs.namenode.name.dir	Specifies where on the local filesystem the DFS name node stores the name table (`fsimage`). If this is a comma-delimited list of directories then the name table is replicated in all of the directories, for redundancy.
dfs.namenode.edits.dir	Specifies where on the local filesystem the DFS name node stores the transaction (`edits`) file. If this is a comma-delimited list of directories, the transaction file is replicated in all of the directories, for redundancy. The default value is set to the same value as `dfs.namenode.name.dir`.
dfs.namenode.checkpoint.period	Specifies the number of seconds between two periodic checkpoints.

dfs.namenode.checkpoint.txns	The standby creates a checkpoint of the namespace every `dfs.namenode.checkpoint.txns` transactions, regardless of whether `dfs.namenode.checkpoint.period` has expired.
dfs.namenode.checkpoint.check.period	Specifies how frequently to query for the number of un-checkpointed transactions.
dfs.namenode.num.checkpoints.retained	Specifies the number of image checkpoint files to be retained in storage directories. All edit logs necessary to recover an up-to-date namespace from the oldest retained checkpoint are also retained.
dfs.namenode.num.extra.edits.retained	Specifies the number of extra transactions which are retained beyond what is minimally necessary for a NN restart. This can be useful for audit purposes or for an HA setup where a remote Standby Node might have been offline and need to have a longer backlog of retained `edits` to start again.
dfs.namenode.edit.log.autoroll.multiplier.threshold	Specifies how often an active namenode rolls its own edit log. The actual threshold (in number of `edits`) is determined by multiplying this value by dfs.namenode.checkpoint.txns. This prevents extremely large edit files from accumulating on the active namenode, which can cause timeouts during namenode start-up and pose an administrative hassle. This behavior is intended as a fail-safe for when the standby fails to roll the edit log by the normal checkpoint threshold.
dfs.namenode.edit.log.autoroll.check.interval.ms	Specifies the time in milliseconds that an active namenode checks if it needs to roll its edit log.
dfs.datanode.data.dir	Determines where on the local filesystem an DFS data node should store its blocks. If this is a comma-delimited list of directories, then data is stored in all named directories, typically on different devices. Directories that do not exist are ignored. Heterogeneous storage allows specifying that each directory resides on a different type of storage: `DISK`, SSD, `ARCHIVE` or `RAM_DISK`.

14.2. Backing Up HDFS Metadata

You can backup of HDFS metadata without taking down either HDFS or the NameNodes.

14.2.1. Get Ready to Backup the HDFS Metadata

- Regardless of the solution, a full, up-to-date continuous backup of the namespace is not possible. Some of the most recent data is always lost. HDFS is not an Online Transaction

Processing (OLTP) system. Most data can be easily recreated if you re-run Extract, Transform, Load (ETL) or processing jobs.

- Normal NameNode failures are handled by the Standby NameNode. Doing so creates a safety-net for the very unlikely case where both master NameNodes fail.
- In the case of both NameNode failures, you can start the NameNode service with the most recent image of the namespace.
- Name Nodes maintain the namespace as follows:
  - Standby NameNodes keep a namespace image in memory based on `edits` available in a storage ensemble in Journal Nodes.
  - Standby NameNodes make a namespace checkpoint and saves a `fsimage_*` to disk.
  - Standby NameNodes transfer the `fsimage` to the primary NameNodes using HTTP.

Both NameNodes write `fsimages` to disk in the following sequence:

- NameNodes write the namespace to a file `fsimage.ckpt_*` on disk.
- NameNodes creates a `fsimage_*.md5` file.
- NameNodes moves the file `fsimage.ckpt_*` to `fsimage_.*`.

The process by which both NameNodes write `fsimages` to disk ensures that:

- The most recent namespace image on disk in a `fsimage_*` file is on the standby NameNode.
- Any `fsimage_*` file on disk is finalized and does not receive updates.

14.2.2. Perform a Backup of the HDFS Metadata

Use the following procedure to backup HDFS metadata without affecting the availability of NameNode:

1. Make sure the Standby NameNode checkpoints the namespace to `fsimage_` once per hour.
2. Deploy monitoring on both NameNodes to confirm that checkpoints are triggering regularly. This helps reduce the amount of missing transactions in the event that you need to restore from a backup containing only `fsimage` files without subsequent edit logs. It is good practice to monitor this anyway, because huge uncheckpointed edit logs can cause long delays after a NameNode restart while it replays those transactions.
3. Back up the most recent "`fsimage_*`" and "`fsimage_*.md5`" from the standby NameNode periodically. Try to keep the latest version of the file on another machine in the cluster.
4. Back up the `VERSION` file from the standby NameNode.

15. Balancing in HDFS

This chapter describes the HDFS Balancer, a tool for balancing data across the storage devices of an HDFS cluster.

15.1. Overview of the HDFS Balancer

The HDFS Balancer is a tool for balancing the data across the storage devices of a HDFS cluster. The HDFS Balancer was originally designed to run slowly so that the balancing activities would not affect normal cluster activities and the running of jobs. As of HDP 2.3.4, the HDFS Balancer was redesigned.

With the redesign, the HDFS Balancer runs faster, though it can also be configured to run slowly. You can also specify the source datanodes, to free up the spaces in particular datanodes. You can use a block distribution application to pin its block replicas to particular datanodes so that the pinned replicas are not moved for cluster balancing.

15.2. Why HDFS Data Becomes Unbalanced

Data stored in HDFS can become unbalanced for any of the following reasons:

- Addition of Datanodes

  When new datanodes are added to a cluster, newly created blocks are written to these datanodes from time to time. The existing blocks are not moved to them without using the HDFS Balancer.

- Behavior of the Client

  In some cases, a client application might not write data uniformly across the datanode machines. A client application might be skewed in writing data, and might always write to some particular machines but not others. HBase is an example of such a client application. In other cases, the client application is not skewed by design, for example, MapReduce or YARN jobs.

  The data is skewed so that some of the jobs write significantly more data than others. When a Datanode receives the data directly from the client, it stores a copy to its local storage for preserving data locality. The datanodes receiving more data generally have higher storage utilization.

- Block Allocation in HDFS

  HDFS uses a constraint satisfaction algorithm to allocate file blocks. Once the constraints are satisfied, HDFS allocates a block by randomly selecting a storage device from the candidate set uniformly. For large clusters, the blocks are essentially allocated randomly in a uniform distribution, provided that the client applications write data to HDFS uniformly across the datanode machines. Uniform random allocation might not result in a uniform data distribution because of randomness. This is generally not a problem when the cluster has sufficient space. The problem becomes serious when the cluster is nearly full.

15.3. Configurations and CLI Options

You can configure the HDFS Balancer by changing the configuration or by using CLI options.

15.3.1. Configuring the Balancer

dfs.datanode.balance.max.concurrent.moves — Limits the maximum number of concurrent block moves that a Datanode is allowed for balancing the cluster. If you set this configuration in a Datanode, the Datanode throws an exception when the limit is exceeded. If you set this configuration in the HDFS Balancer, the HDFS Balancer schedules concurrent block movements within the specified limit. The Datanode setting and the HDFS Balancer setting can be different. As both settings impose a restriction, an effective setting is the minimum of them.

It is recommended that you set this to the highest possible value in Datanodes and adjust the runtime value in the HDFS Balancer to gain the flexibility. The default value is 5.

You can reconfigure without Datanode restart. Follow these steps to reconfigure a Datanode:

1. Change the value of dfs.datanode.balance.max.concurrent.moves in the configuration xml file on the Datanode machine.
2. Start a reconfiguration task. Use the **hdfs dfsadmin -reconfig datanode <dn_addr>:<ipc_port> start** command.

For example, suppose a Datanode has 12 disks. You can set the configuration to 24, a small multiple of the number of disks, in the Datanodes. Setting it to a higher value might not be useful, and only increases disk contention. If the HDFS Balancer is running in a maintenance window, the setting in the HDFS Balancer can be the same, that is, 24, to use all the bandwidth. However, if the HDFS Balancer is running at same time as other jobs, you set it to a smaller value, for example, 5, in the HDFS Balancer so that there is bandwidth available for the other jobs.

dfs.datanode.balance.bandwidthPerSec — Limits the bandwidth in each Datanode using for balancing the cluster. Changing this configuration does not require restarting Datanodes. Use the **dfsadmin -setBalancerBandwidth** command.

The default is `1048576 (=1MB/s)`.

dfs.balancer.moverThreads	Limits the number of total concurrent moves for balancing in the entire cluster. Set this property to the number of threads in the HDFS Balancer for moving blocks. Each block move requires a thread. The default is `1000`.
dfs.balancer.max-size-to-move	With each iteration, the HDFS Balancer chooses datanodes in pairs and moves data between the datanode pairs. Limits the maximum size of data that the HDFS Balancer moves between a chosen datanode pair. If you increase this configuration when the network and disk are not saturated, increases the data transfer between the datanode pair in each iteration while the duration of an iteration remains about the same. The default is `10737418240` (`10GB`).
dfs.balancer.getBlocks.size	Specifies the total data size of the block list returned by a `getBlocks(..)`. When the HDFS Balancer moves a certain amount of data between source and destination datanodes, it repeatedly invokes the `getBlocks(..) rpc` to the Namenode to get lists of blocks from the source datanode until the required amount of data is scheduled. The default is `2147483648` (`2GB`).
dfs.balancer.getBlocks.min-block-size	Specifies the minimum block size for the blocks used to balance the cluster. The default is `10485760` (`10MB`)
dfs.datanode.block-pinning.enabled	Specifies if block-pinning is enabled. When you create a file, a user application can specify a list of favorable datanodes by way of the file creation API in DistributedFileSystem. The namenode uses its best effort, allocating blocks to the favorable datanodes. If `dfs.datanode.block-pinning.enabled` is set to true, if a block replica is written to a favorable datanode, it is “pinned” to that datanode. The pinned replicas are not moved for cluster balancing to keep them stored in the specified favorable datanodes. This feature is useful for block distribution aware user applications such as HBase. The default is `false`.

15.3.2. Using the Balancer CLI Commands

Balancing Policy, Threshold, and Blockpools

[-policy <policy>]	Specifies which policy to use to determine if a cluster is balanced. The two supported policies are `blockpool` and `datanode`. blockpool means that the cluster is balanced if each pool in each node is balanced. datanode means that a cluster is balanced if each datanode is balanced. blockpool is a more strict policy than datanode in the sense that the blockpool requirement implies the datanode requirement. The default policy is datanode.
[-threshold <threshold>]	Specifies a number in [1.0, 100.0] representing the acceptable threshold of the percentage of storage capacity so that storage utilization outside the average +/- the threshold is considered as over/under utilized. The default threshold is 10.0.
[-blockpools <comma-separated list of blockpool ids>]	Specifies a list of block pools on which the HDFS Balancer runs. If the list is empty, the HDFS Balancer runs on all existing block pools. The default value is an empty list.

Include and Exclude Lists

[-include [-f <hosts-file> | <comma-separated list of hosts>]]	When the include list is non-empty, only the datanodes specified in the list are balanced by the HDFS Balancer. An empty include list means including all the datanodes in the cluster. The default value is an empty list.
[-exclude [-f <hosts-file> | <comma-separated list of hosts>]]	The datanodes specified in the exclude list are excluded so that the HDFS Balancer does not balance those datanodes. An empty exclude list means that no datanodes are excluded. When a datanode is specified in both in the include list and the exclude list, the datanode is excluded. The default value is an empty list.

Idle-Iterations and Run During Upgrade

[-idleiterations <idleiterations>]	Specifies the number of consecutive iterations in which no blocks have been moved before the HDFS Balancer terminates with the NO_MOVE_PROGRESS exit status. Specify `-1` for infinite iterations. The default is `5`.
[-runDuringUpgrade]	If specified, the HDFS Balancer runs even if there is an ongoing HDFS upgrade. If not specified, the HDFS

Balancer terminates with the `UNFINALIZED_UPGRADE` exit status.

When there is no ongoing upgrade, this option has no effect. It is usually not desirable to run HDFS Balancer during upgrade. To support rollback, blocks being deleted from HDFS are moved to the internal trash directory in datanodes and not actually deleted. Running the HDFS Balancer during upgrading cannot reduce the usage of any datanode storage.

Source Datanodes

[-source [-f <hosts-file> | <comma-separated list of hosts>]]

Specifies the source datanode list. The HDFS Balancer selects blocks to move from only the specified datanodes. When the list is empty, all the datanodes are chosen as a source. The option can be used to free up the space of some particular datanodes in the cluster. Without the -source option, the HDFS Balancer can be inefficient in some cases.

The default value is an empty list.

The following table shows an example, where the average utilization is 25% so that D2 is within the 10% threshold. It is unnecessary to move any blocks from or to D2. Without specifying the source nodes, HDFS Balancer first moves blocks from D2 to D3, D4 and D5, since they are under the same rack, and then moves blocks from D1 to D2, D3, D4 and D5. By specifying D1 as the source node, HDFS Balancer directly moves blocks from D1 to D3, D4 and D5.

Table 15.1. Example of Utilization Movement

Datanodes (with the same capacity)	Utilization	Rack
D1	95%	A
D2	30%	B
D3, D4, and D5	0%	B

15.3.3. Recommended Configurations

This section provides recommended configurations for Background and Fast modes.

The HDFS Balancer can run in either Background or Fast modes.

Background Mode — HDFS Balancer runs as a background process. The cluster serves other jobs and applications at the same time.

Fast Mode — HDFS Balancer runs at maximum (fast) speed.

Table 15.2. Datanode Configuration Properties

Property	Default	Background Mode	Fast Mode
dfs.datanode.balance.max.concurrent.moves	5	4 x (# of disks)	4 x (# of disks)
dfs.datanode.balance.max.concurrent.moves	10485760 (10 MB)	use default	10737418240 (10 GB)

Table 15.3. Balancer Configuration Properties

Property	Default	Background Mode	Fast Mode
dfs.datanode.balance.max.concurrent.moves	5	# of disks	4 x (# of disks)
dfs.balancer.moverThreads	1000	use default	20,000
dfs.balancer.max-size-to-move	10737418240 (10 GB)	1073741824 (1GB)	107374182400 (100 GB)
dfs.balancer.getBlocks.min-block-size	10485760 (10 MB)	use default	104857600 (100 MB)

15.4. Cluster Balancing Algorithm

This section describes the algorithm deployed by the HDFS Balancer to balance the cluster. The HDFS Balancer runs in iterations. Each iteration consists of the following steps:

15.4.1. Step 1: Storage Group Classification

The HDFS Balancer first invokes the `getLiveDatanodeStorageReport` rpc to the Namenode to the storage report for all the storages in all Datanodes. The storage report contains storage utilization information such as capacity, dfs used space, remaining space, and so forth, for each storage in each datanode.

A Datanode can contain multiple storages and the storages can have different storage types. A storage group `Gi,T` is defined to be the group of all the storages with the same storage type T in Datanode `i`. For example, `Gi,DISK` is the storage group of all the DISK storages in Datanode i. For each storage type T in each datanode i, HDFS Balancer computes Storage Group Utilization (%)

$U_{i,T}$ = 100% (storage group used space)/(storage group capacity),

and Average Utilization (%)

$U_{avg,T}$ = 100% * (sum of all used spaces)/(sum of all capacities).

Let # be the threshold parameter (default is 10%) and $G_{I,T}$ be the storage group with storage type T in datanode I.

```
                    Over-Utilized:       {G_{i,T} :  U_{avg,T} + # < U_{i,T}},
Average + Threshold
 ------------------------------------------------------------------------------
                    Above-Average: {G_{i,T} :  U_{avg,T} < U_{i,T} <= U_{avg,T} + #},
Average
 ------------------------------------------------------------------------------
                    Below-Average: {G_{i,T} :  U_{avg,T} - # <= U_{i,T} <= U_{avg,T}},
Average - Threshold
 ------------------------------------------------------------------------------
                    Under-Utilized:      {G_{i,T} :  U_{i,T} < U_{avg,T} - # }.
```

Roughly speaking, a storage group is over-utilized (under-utilized) if its utilization is larger (smaller) than the average plus (minus) threshold. A storage group is above-average (below-average) if its utilization is larger (smaller) than average but within the threshold.

If there are no over-utilized storages and no under-utilized storages, the cluster is said to be balanced. The HDFS Balancer terminates with a SUCCESS state. Otherwise, it continues with the following steps.

15.4.2. Step 2: Storage Group Pairing

The HDFS Balancer selects over-utilized or above-average storage as source storage, and under-utilized or below-average storage as target storage. It pairs a source storage group with a target storage group (source # target) in the following priority order:

- Same-Rack (where the source and the target storage reside in the same rack)

  Over-Utilized # Under-Utilize

  Over-Utilized # Below-Average

  Above-Average # Under-Utilized

- Any (where the source and target storage do not reside in the same rack)

  Over-Utilized # Under-Utilized

  Over-Utilized # Below-Average

  Above-Average # Under-Utilized

15.4.3. Step 3: Block Move Scheduling

For each source-target pair, the HDFS Balancer chooses block replicas from the source storage groups and schedules block moves. A block replica in a source datanode is a good candidate if it satisfies all of the following conditions:

- The storage type of the block replica in the source datanode is the same as the target storage type.
- The storage type of the block replica is not already scheduled.
- The target does not already have the same block replica.
- The number of racks of the block is not reduced after the move.

Logically, the HDFS Balancer schedules a block replica to be “moved” from a source storage group to a target storage group. In practice, a block usually has multiple replicas. The block move can be done by first copying the replica from a proxy, which can be any storage group containing one of the replicas of the block, to the target storage group, and then deleting the replica in the source storage group.

After a candidate block in the source datanode is specified, the HDFS Balancer selects a storage group containing the same replica as the proxy. The HDFS Balancer selects the closest storage group as the proxy in order to minimize the network traffic.

When it is impossible to schedule a move, the HDFS Balancer terminates with a NO_MOVE_BLOCK exit status.

15.4.4. Step 4: Block Move Execution

The HDFS Balancer dispatches a scheduled block move by invoking the DataTransferProtocol `.replaceBlock(..)` method to the target datanode. It specifies the proxy, and the source as delete-hint in the method call. The target datanode copies the replica directly from the proxy to its local storage. When the copying process has been completed, the target datanode reports the new replica to the Namenode with the `delete-hint`. Namenode uses the `delete-hint` to delete the extra replica, that is, delete the replica stored in the source.

After all block moves are dispatched, the HDFS Balancer waits until all the moves are completed. Then, the HDFS Balancer continues running a new iteration and repeats all of the steps. If the scheduled moves fail for 5 consecutive iterations, the HDFS Balancer terminates with a NO_MOVE_PROGRESS exit status.

15.5. Exit Status

The following table shows exit statuses for the HDFS Balancer:

Table 15.4. Exit Statuses for the HDFS Balancer

Status	Value	Description
SUCCESS	0	The cluster is balanced. There are no over or under-utilized storages, with regard to the specified threshold.
ALREADY_RUNNING	-1	Another HDFS Balancer is running.
NO_MOVE_BLOCK	-2	The HDFS Balancer is not able to schedule a move.
NO_MOVE_PROGRESS	-3	All of the scheduled moves have failed for 5 consecutive iterations.
IO_EXCEPTION	-4	An IOException occurred.
ILLEGAL_ARGUMENTS	-5	An illegal argument in the command or configuration occurred.
INTERUPTED	-6	The HDFS Balancer process was interrupted.
UNFINALIZED_UPGRADE	-7	The cluster is being upgraded.