Hortonworks Data Platform

HDFS Administration Guide

(Mar 1, 2016)

docs.hortonworks.com

Hortonworks Data Platform: HDFS Administration Guide

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 Figures

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.
`lt;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 the Controlling HDP Services Manually section of the HDP Reference Guide.

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]file:///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]file:///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 dfsadmin -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 dfsadmin -setStoragePolicy /cold1 COLD
```

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

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

Argument:

Table 2.2. Getting Storage Policy

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

Example:

```
hdfs dfsadmin -getStoragePolicy /cold1
```

4. Start the DataNode, using the applicable commands in the "Controlling HDP Services Manually" section of Installing HDP 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.

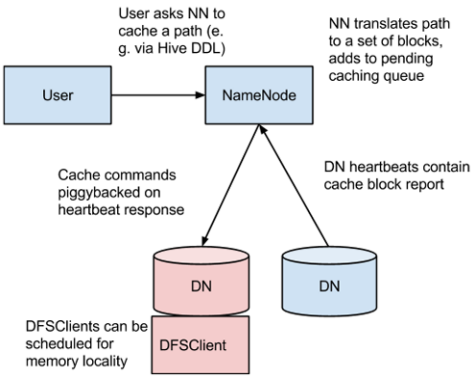
3.2. Caching Use Cases

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 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.
`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 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 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 belongs to in order to remove it. You can use the -

Option	Description
	`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 the "Controlling HDP Services Manually" section of Installing HDP 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. 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.

6.1. 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.

6.2. 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.

6.3. 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
```

6.4. 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`.

6.5. 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.

7. 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.

8. 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.

8.1. 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.

8.2. 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.

8.3. 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.

8.4. 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 the Controlling HDP Services Manually section of the HDP Reference Guide.

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.

9. Running DataNodes as Non-Root

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

9.1. 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.

9.2. 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 the "Controlling HDP Services Manually" section of HDP Reference Guide.

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 "Enable SSL on HDP Components" in the *HDP 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 the "Controlling HDP Services Manually" section of HDP Reference Guide.

10. 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.

10.1. Prerequisites

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

10.2. 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.

10.3. Short-Circuit Local Read Properties in hdfs-site.xml

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`.

Property Name	Property Value	Description
		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 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>
```

11. 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 the "Controlling HDP Services Manually" section of Installing HDP Manually.

12. HDFS "Data at Rest" Encryption

Encryption is a form of data security that is required in industries such as healthcare and the payment card industry. Hadoop provides several ways to encrypt stored data.

- The lowest level of encryption is volume encryption, which protects data after physical theft or accidental loss of a disk volume. The entire volume is encrypted; this approach does not support finer-grained encryption of specific files or directories. In addition, volume encryption does not protect against viruses or other attacks that occur while a system is running.
- Application level encryption (encryption within an application running on top of Hadoop) supports a higher level of granularity and prevents "rogue admin" access, but adds a layer of complexity to the application architecture.
- A third approach, HDFS data at rest encryption, encrypts selected files and directories stored ("at rest") in HDFS. This approach uses specially designated HDFS directories known as "encryption zones."

This chapter focuses on the third approach, HDFS data at rest encryption. The chapter is intended as an introductory quick start to HDFS data at rest encryption. Content will be updated regularly.

12.1. HDFS Encryption Overview

HDFS data at rest encryption implements end-to-end encryption of data read from and written to HDFS. End-to-end encryption means that data is encrypted and decrypted only by the client. HDFS does not have access to unencrypted data or keys.

HDFS encryption involves several elements:

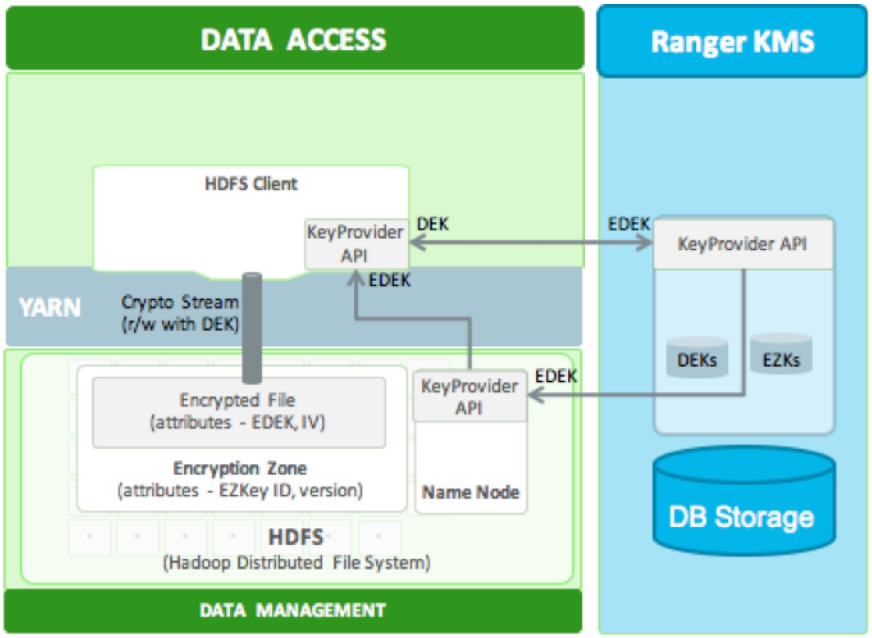
- **Encryption key**: A new level of permission-based access protection, in addition to standard HDFS permissions.
- **HDFS encryption zone**: A special HDFS directory within which all data is encrypted upon write, and decrypted upon read.
  - Each encryption zone is associated with an encryption key that is specified when the zone is created.
  - Each file within an encryption zone has a unique encryption key, called the "data encryption key" (DEK).
  - HDFS does not have access to DEKs. HDFS DataNodes only see a stream of encrypted bytes. HDFS stores "encrypted data encryption keys" (EDEKs) as part of the file's metadata on the NameNode.
  - Clients decrypt an EDEK and use the associated DEK to encrypt and decrypt data during write and read operations.
- **Ranger Key Management Service (Ranger KMS)**: An open source key management service based on Hadoop's `KeyProvider` API.

For HDFS encryption, the Ranger KMS has three basic responsibilities:

- Provide access to stored encryption zone keys.
- Generate and manage encryption zone keys, and create encrypted data keys to be stored in Hadoop.
- Audit all access events in Ranger KMS.

Note: This chapter is intended for security administrators who are interested in configuring and using HDFS encryption. For more information about Ranger KMS, see the Ranger KMS Administration Guide.

Figure 12.1. HDFS Encryption Components



Role Separation

Access to the key encryption/decryption process is typically restricted to end users. This means that encrypted keys can be safely stored and handled by HDFS, because the HDFS admin user does not have access to them.

This role separation requires two types of HDFS administrator accounts:

- HDFS service user: the system-level account associated with HDFS (`hdfs` by default).
- HDFS admin user: an account in the `hdfs` supergroup, which is used by HDFS administrators to configure and manage HDFS.

Important

For clear segregation of duties, we recommend that you restrict use of the `hdfs` account to system/interprocess use. Do not provide its password to

physical users. A (human) user who administers HDFS should only access HDFS through an admin user account created specifically for that purpose. For more information about creating an HDFS admin user, see Creating an HDFS Admin User.

Other services may require a separate admin account for clusters with HDFS encryption zones. For service-specific information, see Configuring HDP Services for HDFS Encryption.

12.2. Configuring and Starting the Ranger Key Management Service (Ranger KMS)

In a typical environment, a security administrator will set up the Ranger Key Management Service. For information about installing and configuring the Ranger KMS, see the Ranger KMS Administration Guide.

12.3. Configuring and Using HDFS Data at Rest Encryption

After the Ranger KMS has been set up and the NameNode and HDFS clients have been configured, an HDFS administrator can use the `hadoop key` and `hdfs crypto` command-line tools to create encryption keys and set up new encryption zones.

The overall workflow is as follows:

1. Create an HDFS encryption zone key that will be used to encrypt the file-level data encryption key for every file in the encryption zone. This key is stored and managed by Ranger KMS.

2. Create a new HDFS folder. Specify required permissions, owner, and group for the folder.

3. Using the new encryption zone key, designate the folder as an encryption zone.

4. Configure client access. The user associated with the client application needs sufficient permission to access encrypted data. In an encryption zone, the user needs file/directory access (through Posix permissions or Ranger access control), as well as access for certain key operations. To set up ACLs for key-related operations, see the Ranger KMS Administration Guide.

After permissions are set, Java API clients and HDFS applications with sufficient HDFS and Ranger KMS access privileges can write and read to/from files in the encryption zone.

12.3.1. Prepare the Environment

HDP supports hardware acceleration with Advanced Encryption Standard New Instructions (AES-NI). Compared with the software implementation of AES, hardware acceleration offers an order of magnitude faster encryption/decryption.

To use AES-NI optimization you need CPU and library support, described in the following subsections.

12.3.1.1. CPU Support for AES-NI optimization

AES-NI optimization requires an extended CPU instruction set for AES hardware acceleration.

There are several ways to check for this; for example:

```
$ cat /proc/cpuinfo | grep aes
```

Look for output with flags and 'aes'.

12.3.1.2. Library Support for AES-NI optimization

You will need a version of the `libcrypto.so` library that supports hardware acceleration, such as OpenSSL 1.0.1e. (Many OS versions have an older version of the library that does not support AES-NI.)

A version of the `libcrypto.so` libary with AES-NI support must be installed on HDFS cluster nodes and MapReduce client hosts – that is, any host from which you issue HDFS or MapReduce requests. The following instructions describe how to install and configure the `libcrypto.so` library.

RHEL/CentOS 6.5 or later

On HDP cluster nodes, the installed version of `libcrypto.so` supports AES-NI, but you will need to make sure that the symbolic link exists:

```
$ sudo ln -s /usr/lib64/libcrypto.so.1.0.1e /usr/lib64/
libcrypto.so
```

On MapReduce client hosts, install the `openssl-devel` package:

```
$ sudo yum install openssl-devel
```

12.3.1.3. Verifying AES-NI Support

To verify that a client host is ready to use the AES-NI instruction set optimization for HDFS encryption, use the following command:

```
hadoop checknative
```

You should see a response similar to the following:

```
15/08/12 13:48:39 INFO bzip2.Bzip2Factory: Successfully loaded & initialized
 native-bzip2 library system-native
14/12/12 13:48:39 INFO zlib.ZlibFactory: Successfully loaded & initialized
 native-zlib library
Native library checking:
hadoop:  true /usr/lib/hadoop/lib/native/libhadoop.so.1.0.0
zlib:    true /lib64/libz.so.1
snappy:  true /usr/lib64/libsnappy.so.1
lz4:     true revision:99
bzip2:   true /lib64/libbz2.so.1
openssl: true /usr/lib64/libcrypto.so
```

If you see `true` in the `openssl` row, Hadoop has detected the right version of `libcrypto.so` and optimization will work.

If you see `false` in this row, you do not have the correct version.

12.3.2. Create an Encryption Key

Create a "master" encryption key for the new encryption zone. Each key will be specific to an encryption zone.

Ranger supports AES/CTR/NoPadding as the cipher suite. (The associated property is listed under HDFS -> Configs in the Advanced hdfs-site list.)

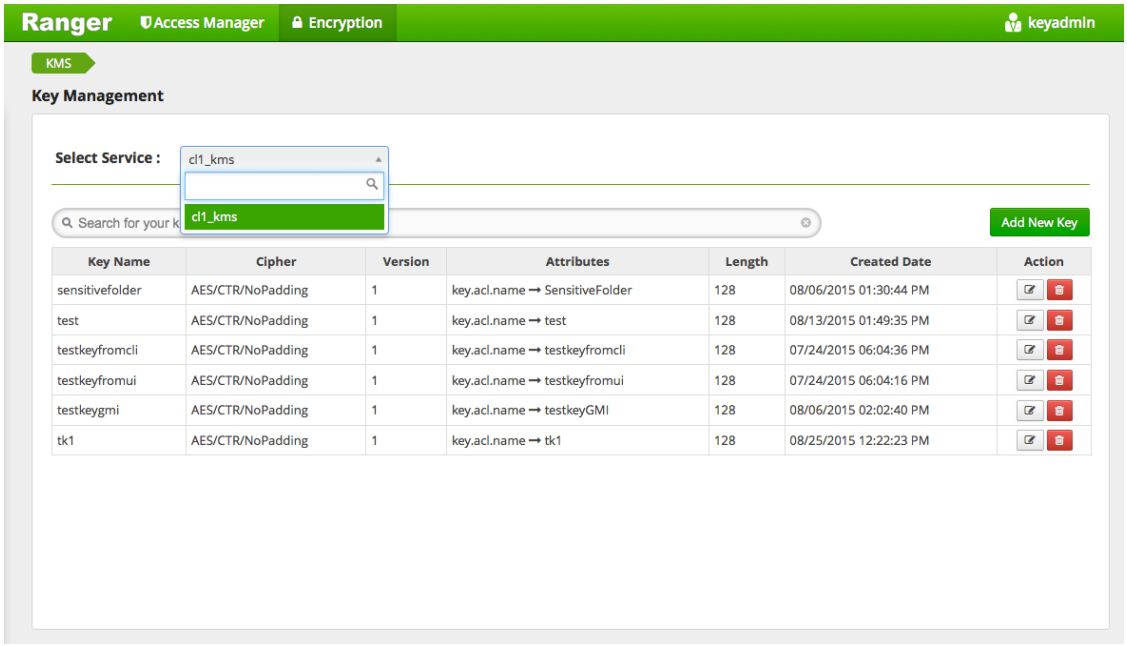
Key size can be 128 or 256 bits.

Recommendation: create a new superuser for key management. In the following examples, superuser `encr` creates the key. This separates the data access role from the encryption role, strengthening security.

Create an Encryption Key using Ranger KMS (Recommended)

In the Ranger Web UI screen:

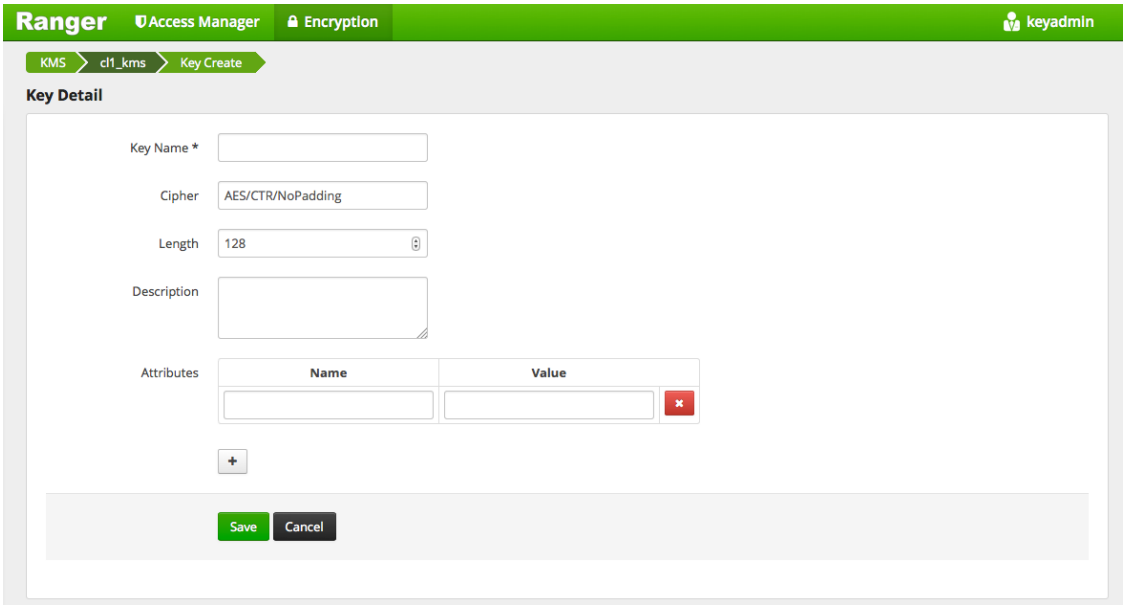
1. Choose the Encryption tab at the top of the screen.
2. Select the KMS service from the drop-down list.



To create a new key:

1. Click on "Add New Key":
2. Add a valid key name.

3. Select the cipher name. Ranger supports AES/CTR/NoPadding as the cipher suite.
4. Specify the key length, 128 or 256 bits.
5. Add other attributes as needed, and then save the key.



For information about rolling over and deleting keys, see Using the Ranger Key Management Service in the *Ranger KMS Administration Guide*.

Warning

Do not delete an encryption key while it is in use for an encryption zone. This will result in loss of access to data in that zone.

Create an Encryption Key using the CLI

The full syntax of the hadoop `key create` command is as follows:

```
[create <keyname> [-cipher <cipher>]
[-size <size>]
[-description <description>]
[-attr <attribute=value>]
[-provider <provider>]
[-help]]
```

Example:

```
# su - encr
```

```
# hadoop key create <key_name> [-size <number-of-bits>]
```

The default key size is 128 bits. The optional `-size` parameter supports 256-bit keys, and requires the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy File on all hosts in the cluster. For installation information, see the Ambari Security Guide.

Example:

```
# su - encr

# hadoop key create key1
```

To verify creation of the key, list the metadata associated with the current user:

```
# hadoop key list -metadata
```

For information about rolling over and deleting keys, see Using the Ranger Key Management Service in the *Ranger KMS Administration Guide*.

Warning

Do not delete an encryption key while it is in use for an encryption zone. This will result in loss of access to data in that zone.

12.3.3. Create an Encryption Zone

Each encryption zone must be defined using an empty directory and an existing encryption key. An encryption zone cannot be created on top of a directory that already contains data.

Recommendation: use one unique key for each encryption zone.

Use the `crypto createZone` command to create a new encryption zone. The syntax is:

```
-createZone -keyName <keyName> -path <path>
```

where:

- `-keyName`: specifies the name of the key to use for the encryption zone.
- `-path` specifies the path of the encryption zone to be created. It must be an empty directory.

Note

The `hdfs` service account can create zones, but cannot write data unless the account has sufficient permission.

Recommendation: Define a separate user account for the HDFS administrator, and do not provide access to keys for this user in Ranger KMS.

Steps:

1. As HDFS administrator, create a new empty directory. For example:

   ```
   # hdfs dfs -mkdir /zone_encr
   ```

2. Using the encryption key, make the directory an encryption zone. For example:

   ```
   # hdfs crypto -createZone -keyName key1 -path /zone_encr
   ```

   When finished, the NameNode will recognize the folder as an HDFS encryption zone.

3. To verify creation of the new encryption zone, run the `crypto -listZones` command as an HDFS administrator:

   `-listZones`

   You should see the encryption zone and its key. For example:

```
$ hdfs crypto -listZones
/zone-encr  key1
```

Note

The following property (in the `hdfs-default.xml` file) causes listZone requests to be batched. This improves NameNode performance. The property specifies the maximum number of zones that will be returned in a batch.

`dfs.namenode.list.encryption.zones.num.responses`

The default is 100.

To remove an encryption zone, delete the root directory of the zone. For example:

```
hdfs dfs -rm -R /zone_encr
```

12.3.4. Copy Files from/to an Encryption Zone

To copy existing files into an encryption zone, use a tool like `distcp`.

Note: for separation of administrative roles, do not use the `hdfs` user to create encryption zones. Instead, designate another administrative account for creating encryption keys and zones. See Creating an HDFS Admin User for more information.

The files will be encrypted using a file-level key generated by the Ranger Key Management Service.

DistCp Considerations

`DistCp` is commonly used to replicate data between clusters for backup and disaster recovery purposes. This operation is typically performed by the cluster administrator, via an HDFS superuser account.

To retain this workflow when using HDFS encryption, a new virtual path prefix has been introduced, `/.reserved/raw/`. This virtual path gives super users direct access to the underlying encrypted block data in the file system, allowing super users to `distcp` data without requiring access to encryption keys. This also avoids the overhead of decrypting and re-encrypting data. The source and destination data will be byte-for-byte identical, which would not be true if the data were re-encrypted with a new EDEK.

Warning

When using `/.reserved/raw/` to `distcp` encrypted data, make sure you preserve extended attributes with the `-px` flag. This is necessary because

encrypted attributes such as the EDEK are exposed through extended attributes; they *must* be preserved to be able to decrypt the file. For example:

```
sudo -u encr hadoop distcp -px hdfs:/cluster1-
namenode:50070/.reserved/raw/apps/enczone hdfs:/cluster2-
namenode:50070/.reserved/raw/apps/enczone
```

This means that if the `distcp` operation is initiated at or above the encryption zone root, it will automatically create a new encryption zone at the destination (if one does not already exist).

Recommendation: To avoid potential mishaps, first create identical encryption zones on the destination cluster.

Copying between encrypted and unencrypted locations

By default, `distcp` compares file system checksums to verify that data was successfully copied to the destination.

When copying between an unencrypted and encrypted location, file system checksums will not match because the underlying block data is different. In this case, specify the `-skipcrccheck` and `-update` flags to avoid verifying checksums.

12.3.5. Read and Write Files from/to an Encryption Zone

Clients and HDFS applications with sufficient HDFS and Ranger KMS permissions can read and write files from/to an encryption zone.

Overview of the client write process:

1. The client writes to the encryption zone.
2. The NameNode checks to make sure that the client has sufficient write access permissions. If so, the NameNode asks Ranger KMS to create a file-level key, encrypted with the encryption zone master key.
3. The Namenode stores the file-level encrypted data encryption key (EDEK) generated by Ranger KMS as part of the file's metadata, and returns the EDEK to the client.
4. The client asks Ranger KMS to decode the EDEK (to DEK), and uses the DEK to write encrypted data. Ranger KMS checks for permissions for the user before decrypting EDEK and producing the DEK for the client.

Overview of the client read process:

1. The client issues a read request for a file in an encryption zone.
2. The NameNode checks to make sure that the client has sufficient read access permissions. If so, the NameNode returns the file's EDEK and the encryption zone key version that was used to encrypt the EDEK.
3. The client asks Ranger KMS to decrypt the EDEK. Ranger KMS checks for permissions to decrypt EDEK for the end user.
4. Ranger KMS decrypts and returns the (unencrypted) data encryption key (DEK).

5. The client uses the DEK to decrypt and read the file.

The preceding steps take place through internal interactions between the DFSClient, the NameNode, and Ranger KMS.

In the following example, the `/zone_encr` directory is an encrypted zone in HDFS.

To verify this, use the `crypto -listZones` command (as an HDFS administrator). This command lists the root path and the zone key for the encryption zone. For example:

```
# hdfs crypto -listZones
/zone_encr  key1
```

Additionally, the `/zone_encr` directory has been set up for read/write access by the `hive` user:

```
# hdfs dfs -ls /
 …
drwxr-x---   - hive   hive            0 2015-01-11 23:12 /zone_encr
```

The `hive` user can, therefore, write data to the directory.

The following examples use the `copyFromLocal` command to move a local file into HDFS.

```
[hive@blue ~]# hdfs dfs -copyFromLocal web.log /zone_encr
[hive@blue ~]# hdfs dfs -ls /zone_encr
Found 1 items
-rw-r--r--   1 hive hive       1310 2015-01-11 23:28 /zone_encr/web.log
```

The `hive` user can read data from the directory, and can verify that the file loaded into HDFS is readable in its unencrypted form.

```
[hive@blue ~]# hdfs dfs -copyToLocal /zone_encr/web.log read.log
[hive@blue ~]# diff web.log read.log
```

Note

For more information about accessing encrypted files from Hive and other components, see Configuring HDP Services for HDFS Encryption.

Users without access to KMS keys will be able to see file names (via the -ls command), but they will not be able to write data or read from the encrypted zone. For example, the `hdfs` user lacks sufficient permissions, and cannot access the data in `/zone_encr`:

```
[hdfs@blue ~]# hdfs dfs -copyFromLocal install.log /zone_encr
copyFromLocal: Permission denied: user=hdfs, access=EXECUTE, inode="/
zone_encr":hive:hive:drwxr-x---

[hdfs@blue ~]# hdfs dfs -copyToLocal /zone_encr/web.log read.log
copyToLocal: Permission denied: user=hdfs, access=EXECUTE, inode="/
zone_encr":hive:hive:drwxr-x---
```

12.3.6. Delete Files from an Encryption Zone

You cannot move data from an Encryption Zone to a global Trash bin outside of the encryption zone.

To delete files from an encryption zone, use one of the following approaches:

1. When deleting the file via CLI, use the `-skipTrash` option. For example:

   ```
   hdfs dfs -rm /zone_name/file1 -skipTrash
   ```

2. When deleting the file via CLI, use the `-skipTrash` option. For example:

   ```
   hdfs dfs -rm /zone_name/file1 -skipTrash
   ```

3. (**Hive only**) Use PURGE, as in DROP TABLE ... PURGE. This skips the Trash bin even if the trash feature is enabled.

12.4. Configuring HDP Services for HDFS Encryption

The following HDP components support HDFS data at rest encryption:

- Hive
- HBase
- Sqoop
- YARN
- MapReduce
- Oozie
- WebHDFS

The following components do not currently support HDFS data at rest encryption:

- Hive on Tez
- Spark
- HDP Search
- Storm
- Accumulo
- Falcon

The remainder of this section describes scenarios and access considerations for accessing HDFS-encrypted files from supporting HDP components.

12.4.1. Hive

Recommendation: Store Hive data in an HDFS path called `/apps/hive`.

12.4.1.1. Configuring Hive Tables for HDFS Encryption

Before enabling encryption zones, decide whether to store your Hive tables across one zone or multiple encryption zones.

Single Encryption Zone

To configure a single encryption zone for your entire Hive warehouse:

1. Rename `/apps/hive` to `/apps/hive-old`
2. Create an encryption zone at `/apps/hive`
3. `distcp` all of the data from `/apps/hive-old` to `/apps/hive`.

To configure the Hive scratch directory (`hive.exec.scratchdir`) so that it resides inside the encryption zone:

1. Set the directory to `/apps/hive/tmp`.
2. Make sure that the permissions for `/apps/hive/tmp` are set to `1777`.

Multiple Encryption Zones

To access encrypted databases and tables with different encryption keys, configure multiple encryption zones.

For example, to configure two encrypted tables, `ez1.db` and `ez2.db`, in two different encryption zones:

1. Create two new encryption zones, `/apps/hive/warehouse/ez1.db` and `/apps/hive/warehouse/ez2.db`.
2. Load data into Hive tables `ez1.db` and `ez2.db` as usual, using `LOAD` statements. (For additional considerations, see "Loading Data into an Encrypted Table.")

12.4.1.2. Loading Data into an Encrypted Table

By design, HDFS-encrypted files cannot be moved or loaded from one encryption zone into another encryption zone, or from an encryption zone into an unencrypted directory. Encrypted files can only be copied.

Within an encryption zone, files can be copied, moved, loaded, and renamed.

Recommendations:

- When loading unencrypted data into encrypted tables (e.g., `LOAD DATA INPATH`), we recommend placing the source data (to be encrypted) into a landing zone within the destination encryption zone.
- An attempt to load data from one encryption zone into another will result in a copy operation. `Distcp` will be used to speed up the process if the size of the files being

copied is higher than the value specified by the `hive.exec.copyfile.maxsize` property. The default limit is 32 MB.

Here are two approaches for loading unencrypted data into an encrypted table:

- To load unencrypted data into an encrypted table, use the `LOAD DATA ...` statement.

  If the source data does not reside inside the encryption zone, the `LOAD` statement will result in a copy. If your data is already inside HDFS, though, you can use `distcp` to speed up the copying process.

- If the data is already inside a Hive table, create a new table with a `LOCATION` inside an encryption zone, as follows:

  ```
  CREATE TABLE encrypted_table [STORED AS] LOCATION ... AS SELECT *
  FROM <unencrypted_table>
  ```

  Note

  The location specified in the `CREATE TABLE` statement must be within an encryption zone. If you create a table that points `LOCATION` to an unencrypted directory, your data will not be encrypted. You must copy your data to an encryption zone, and then point `LOCATION` to that encryption zone.

If your source data is already encrypted, use the `CREATE TABLE` statement. Point `LOCATION` to the encrypted source directory where your data resides:

```
CREATE TABLE encrypted_table [STORED AS] LOCATION ... AS SELECT *
FROM <encrypted_source_directory>
```

This is the fastest way to create encrypted tables.

12.4.1.3. Encrypting Other Hive Directories

- **`LOCALSCRATCHDIR`** : The MapJoin optimization in Hive writes HDFS tables to a local directory and then uploads them to distributed cache. To enable encryption, either disable MapJoin (set `hive.auto.convert.join` to `false`) or encrypt the `local` Hive Scratch directory (`hive.exec.local.scratchdir`). **Performance note**: disabling MapJoin will result in slower join performance.

- **`DOWNLOADED_RESOURCES_DIR`**: Jars that are added to a user session and stored in HDFS are downloaded to `hive.downloaded.resources.dir`. If you want these Jar files to be encrypted, configure `hive.downloaded.resources.dir` to be part of an encryption zone. This directory needs to be accessible to the HiveServer2.

- **NodeManager Local Directory List**: Hive stores Jars and MapJoin files in the distributed cache, so if you'd like to use MapJoin or encrypt Jars and other resource files, the YARN configuration property NodeManager Local Directory List (`yarn.nodemanager.local-dirs`) must be configured to a set of encrypted local directories on all nodes.

  Alternatively, to disable MapJoin, set `hive.auto.convert.join` to `false`.

12.4.1.4. Additional Changes in Behavior with HDFS-Encrypted Tables

- Users reading data from read-only encrypted tables must have access to a temp directory that is encrypted with at least as strong encryption as the table.
- By default, temp datas related to HDFS encryption is written to a staging directory identified by the `hive-exec.stagingdir` property created in the `hive-site.xml` file? associated with the table folder.
- Previously, an `INSERT OVERWRITE` on a partitioned table inherited permissions for new data from the existing partition directory. With encryption enabled, permissions are inherited from the table.
- When using encryption with Trash enabled, table deletion operates differently than the default trash mechanism. For more information see Delete Files from an Encryption Zone.

12.4.2. HBase

HBase stores all of its data under its root directory in HDFS, configured with `hbase.rootdir`. The only other directory that the HBase service will read or write is `hbase.bulkload.staging.dir`.

On HDP clusters, `hbase.rootdir` is typically configured as `/apps/hbase/data`, and `hbase.bulkload.staging.dir` is configured as `/apps/hbase/staging`. HBase data, including the root directory and staging directory, can reside in an encryption zone on HDFS.

The HBase service user needs to be granted access to the encryption key in the Ranger KMS, because it performs tasks that require access to HBase data (unlike Hive or HDFS).

By design, HDFS-encrypted files cannot be bulk-loaded from one encryption zone into another encryption zone, or from an encryption zone into an unencrypted directory. Encrypted files can only be copied. An attempt to load data from one encryption zone into another will result in a copy operation. Within an encryption zone, files can be copied, moved, bulk-loaded, and renamed.

12.4.2.1. Recommendations

- Make the parent directory for the HBase root directory and bulk load staging directory an encryption zone, instead of just the HBase root directory. This is because HBase bulk load operations need to move files from the staging directory into the root directory.
- In typical deployments, `/apps/hbase` can be made an encryption zone.
- Do not create encryption zones as subdirectories under `/apps/hbase`, because HBase may need to rename files across those subdirectories.
- The landing zone for unencrypted data should always be within the destination encryption zone.

12.4.2.2. Steps

On a cluster without HBase currently installed:

1. Create the `/apps/hbase` directory, and make it an encryption zone.
2. Configure `hbase.rootdir=/apps/hbase/data`.
3. Configure `hbase.bulkload.staging.dir=/apps/hbase/staging`.

On a cluster with HBase already installed, perform the following steps:

1. Stop the HBase service.
2. Rename the `/apps/hbase` directory to `/apps/hbase-tmp`.
3. Create an empty `/apps/hbase` directory, and make it an encryption zone.
4. `DistCp -skipcrccheck -update` all data from `/apps/hbase-tmp` to `/apps/hbase`, preserving user-group permissions and extended attributes.
5. Start the HBase service and verify that it is working as expected.
6. Remove the `/apps/hbase-tmp` directory.

12.4.2.3. Changes in Behavior after HDFS Encryption is Enabled

The HBase bulk load process is a MapReduce job that typically runs under the user who owns the source data. HBase data files created as a result of the job are then bulk loaded in to HBase RegionServers. During this process, HBase RegionServers move the bulk-loaded files from the user's directory and move (rename) the files into the HBase root directory (`/apps/hbase/data`). When data at rest encryption is used, HDFS cannot do a rename across encryption zones with different keys.

Workaround: run the MapReduce job as the `hbase` user, and specify an output directory that resides in the same encryption zone as the HBase root directory.

12.4.3. Sqoop

Following are considerations for using Sqoop to import or export HDFS-encrypted data.

12.4.3.1. Recommendations

- **For Hive:**

  Make sure that you are using Sqoop with the `--target-dir` parameter set to a directory that is inside the Hive encryption zone. Specify the -D option after sqoop import.

  For example:

  ```
  sqoop import \

  -D sqoop.test.import.rootDir=<root-directory> \

  --target-dir <directory-inside-encryption-zone> \
  ```

```
<additional-arguments>
```

- **For append or incremental import:**

  Make sure that the `sqoop.test.import.rootDir` property points to the encryption zone specified in the `--target-dir` argument.

- **For HCatalog:**

  No special configuration is required.

12.4.4. MapReduce on YARN

Recommendation: Make `/apps/history` a single encryption zone. History files are moved between the `intermediate` and `done` directories, and HDFS encryption will not allow you to move encrypted files across encryption zones.

12.4.4.1. Steps

On a cluster with MapReduce over YARN installed, create the `/apps/history` directory and make it an encryption zone.

If `/apps/history` already exists and is not empty:

1. Create an empty `/apps/history-tmp` directory
2. Make `/apps/history-tmp` an encryption zone
3. Copy (`distcp`) all data from `/apps/history` into `/apps/history-tmp`
4. Remove `/apps/history`
5. Rename `/apps/history-tmp` to `/apps/history`

12.4.5. Oozie

12.4.5.1. Recommendations

A new Oozie administrator role (`oozie-admin`) has been created in HDP 2.3.

This role enables role separation between the Oozie daemon and administrative tasks. Both the `oozie-admin` role and the `oozie` role must be specified in the `adminusers.txt` file. This file is installed in HDP 2.3 with both roles specified. Both are also defined in Ambari 2.1 as well. Modification is only required if administrators choose to change the default administrative roles for Oozie.

If `oozie-admin` is used as the Oozie administrator user in your cluster, then the role is automatically managed by ambari.

If you plan to create an Oozie admin user other than `oozie-admin`, add the chosen username to `adminusers.txt` under the `$OOZIE_HOME/conf directory`.

Here is a sample `adminusers.txt` file:

```
#
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements.  See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership.  The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License.  You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
# Users should be set using following rules:
#
#     One user name per line
#     Empty lines and lines starting with '#' are ignored

oozie
oozie-admin
```

12.4.6. WebHDFS

12.4.6.1. Recommendations

WebHDFS is supported for writing and reading files to and from encryption zones.

12.4.6.1.1. Steps

To access encrypted files via WebHDFS, complete the following steps:

1. To enable WebHDFS in `hdfs-site.xml`, set the `dfs.webhdfs.enabled` property to true:

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

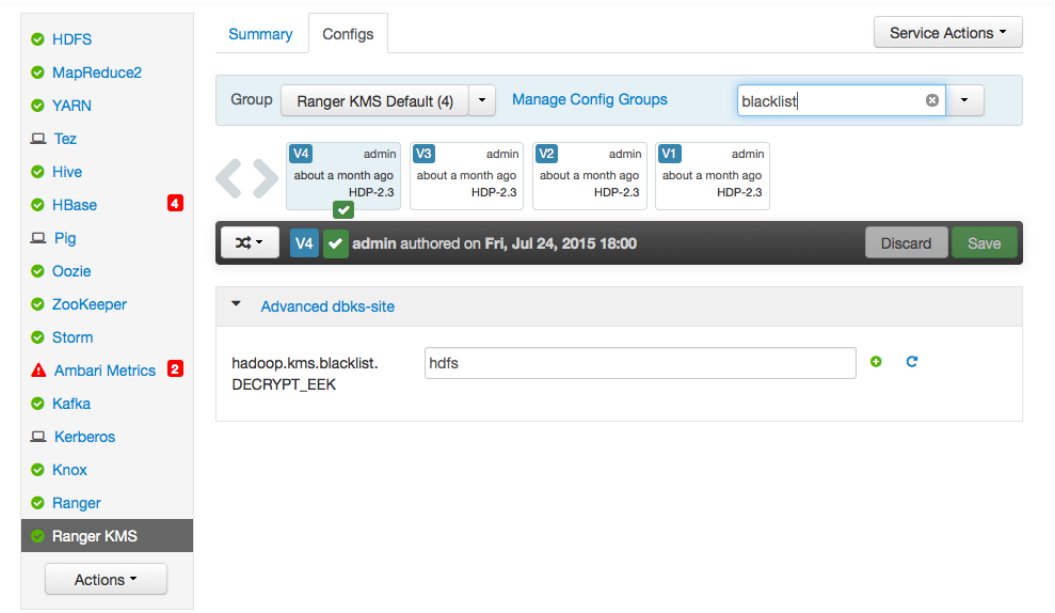
2. Make sure that you have separate HDFS administrative and service users, as described in Creating an HDFS Admin User.

3. KMS supports a blacklist and a whitelist for key access (through `kms-acls.xml`).

   By default the `hdfs` service user is included in the blacklist for decrypt_eek operations. To support WebHDFS, the HDFS service user must not be on the key access blacklist. Remove the HDFS service user from the blacklist:

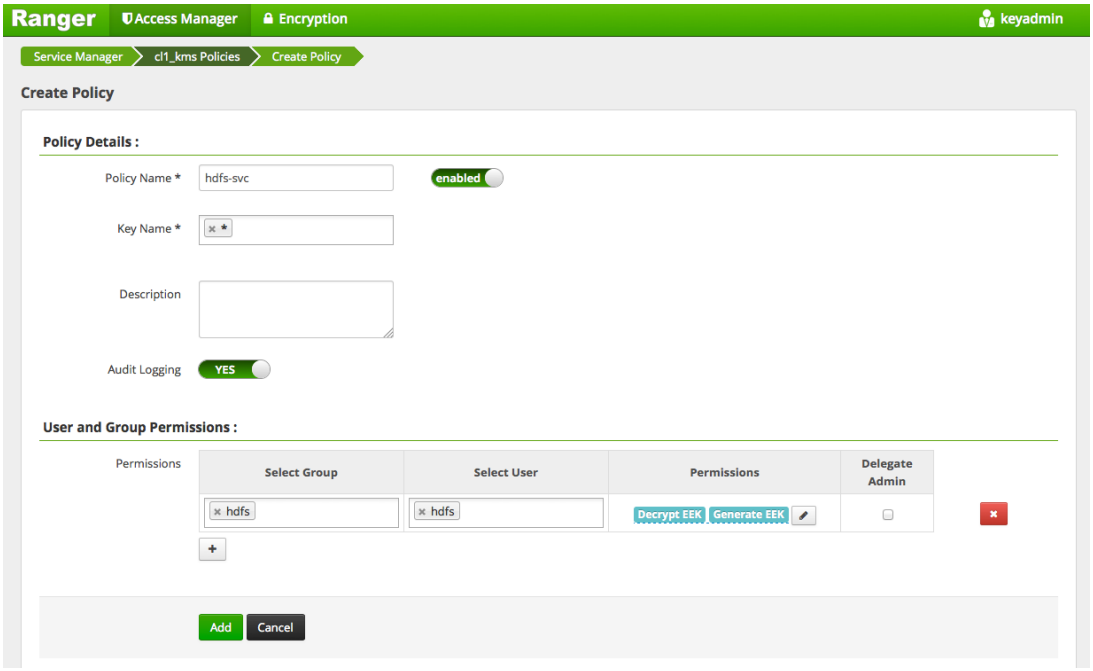
   a. To edit the blacklist using Ambari, go to Ranger KMS -> Configs, and search for "blacklist" or open the Advanced dbks-site list.

   b. Remove `hdfs` from the `hadoop.kms.blacklist.DECRYPT_EEK` property:



c. Restart Ranger KMS.

4. The HDFS service user must have GENERATE_EEK and DECRYPT_EEK permissions. To add the permissions using the Ranger Web UI, select the Access Manager tab-> Resource Based Policies (the default Access Manager view). Select the key store, select the policy, and click the edit icon. In the Permissions column click the edit icon and check the boxes for GenerateEEK and DecryptEEK. Then click Save.



5. Because the HDFS service user will have access to all keys, the HDFS service user should not be the administrative user. Specify a different administrative user in `hdfs-site.xml` for the administrative user.

For more information about operational tasks using Ranger KMS, see the Ranger KMS Administration Guide.

12.5. Appendix: Creating an HDFS Admin User

To capitalize on the capabilities of HDFS data at rest encryption, you will need two separate types of HDFS administrative accounts:

- HDFS administrative user: an account in the `hdfs` supergroup that is used to manage encryption keys and encryption zones. Examples in this chapter use an administrative user account named `encr`.
- HDFS service user: the system-level account traditionally associated with HDFS. By default this is user `hdfs` in HDP. This account owns the HDFS DataNode and NameNode processes.

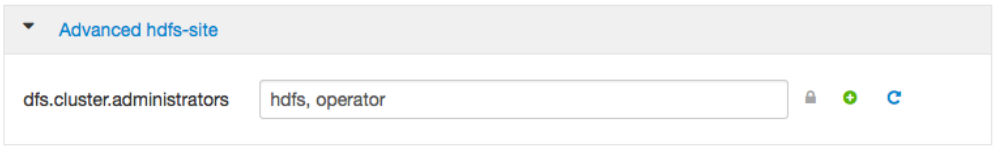
Important

This is a system-only account. Physical users should not be given access to this account.

Complete the following steps to create a new HDFS administrative user.

Note: These steps use sample values for group (`operator`) and user account (`opt1`).

1. Create a new group called `operator`.
2. Add a new user (for example, `opt1`) to the group.
3. Add principal `opt1@EXAMPLE.COM` and create a keytab.
4. Login as `opt1`, and do a `kinit` operation.
5. In Ambari, add `operator` to `dfs.permissions.superusergroup`.
6. In Ambari, add `hdfs, operator` to `dfs.cluster.administrators`:



7. Add `opt1` to the KMS blacklist. Set the corresponding property in Ambari:

   `hadoop.kms.blacklist.DECRYPT_EEK=opt1`

8. Restart HDFS.

Validation

Make sure the `opt1` account has HDFS administrative access:

```
hdfs dfsadmin -report
```

Make sure the `opt1` account cannot access encrypted files. For example, if `/data/test/file.txt` is in an encryption zone, the following command should return an error:

```
hdfs dfs -cat /data/test/file.txt
```

Additional Administrative User Accounts

If you plan to use HDFS data at rest encryption with YARN, we recommend that you create a separate administrative user account for YARN administration.

If you plan to use HDFS data at rest encryption with Oozie, refer to the Oozie section of this chapter.