Hortonworks Data Platform

System Administration Guides

(Aug 29, 2014)

docs.hortonworks.com

Hortonworks Data Platform: System Administration Guides

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

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

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

Table of Contents

List of Tables

1. ACLs on HDFS

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

In this section:

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.

  Usage:

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

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

  Options:

Option	Description
`-b`	Remove all entries, but retain the base ACL entries. The entries for User, Group, and Others are retained for c
`-k`	Remove the default ACL.
`-R`	Apply operations to all files and directories recursively.
`-m`	Modify the ACL. New entries are added to the ACL, and existing entries are retained.
`-x`	Remove the specified ACL entries. All other ACL entries are retained.
`--set`	Fully replace the ACL and discard all existing entries. The `<acl_spec>` must include entries for User, Group, a Permission Bits.
`<acl_spec>`	A comma-separated list of ACL entries.
`<path>`	The path to the file or directory to modify.

Examples:

```
hdfs dfs -setfacl -m user:hadoop:rw- /file

hdfs dfs -setfacl -x user:hadoop /file

hdfs dfs -setfacl -b /file

hdfs dfs -setfacl -k /dir

hdfs dfs -setfacl --set user::rw-,user:hadoop:rw-,group::r--,other::r-- /
file

hdfs dfs -setfacl -R -m user:hadoop:r-x /dir

hdfs dfs -setfacl -m default:user:hadoop:r-x /dir
```

Exit Code:

Returns 0 on success and non-zero on error.

- **`getfacl`**

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

  Usage:

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

  Options:

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. ACLs Examples

In this section:

- Introduction: ACLs Versus Permission Bits
- Example 1: Granting Access to Another Named Group
- Example 2: Using a Default ACL for Automatic Application to New Children
- Example 3: Blocking Access to a Sub-Tree for a Specific User

1.3.1. Introduction: ACLs Versus Permission Bits

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 bruce  sales    22K 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.

1.3.2. 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 "+" symbol to indicate the presence of an ACL. The "+" 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.

1.3.3. 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::---
  ```

The default ACL is copied from the parent directory to a child file or directory when it is created. Subsequent changes to the default ACL of the parent directory do not alter the ACLs of existing children.

1.3.4. 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 all access to "monthly-sales-data" by user Diana:

  ```
  > 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::---
  ```

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

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

1. If the user is the file owner, the Owner Permission Bits are enforced.
2. Else, if the user has a named user ACL entry, those permissions are enforced.
3. 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.)
4. 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 ‘+’ symbol is appended to the listed permissions (the output of the `ls -l` command) of any file or directory with an associated ACL.

Backward-Compatibility

The implementation of ACLs is backward-compatible with existing usage of Permission Bits. Changes applied via Permission Bits (i.e., `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. In other words, 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
- Multiple Groups
- Hive Partitioned Tables
- Default ACLs
- Minimal ACL/Permissions Only
- Block Access to a Sub-Tree for a Specific User
- ACLs with Sticky Bit

1.5.1. 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:

```
user:bruce:r--
user:diana:r--
user:clark:r--
```

1.5.2. Multiple Groups

In this use case, multiple groups require Read and Write access to a file. There is no group containing the union of all of the groups' 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-
```

1.5.3. 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
```

A "salesadmin" group is the owning group for all of these files. Members of this group have Read and Write access to all files. Separate country-specific groups can run Hive queries that only read data for a specific country, e.g., "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.

1.5.4. 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.

1.5.5. 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::---
```

1.5.6. 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 newly-added files under "dir2". If a "file4" is created under "dir3", Bruce will not be able to access it.

1.5.7. ACLs with Sticky Bit

In this use case, multiple named users or named groups require full access to a general-purpose 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 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. Capacity Scheduler

This guide describes how to use the Capacity Scheduler to allocate shared cluster resources among users and groups.

In this section:

2.1. Introduction

The fundamental unit of scheduling in YARN is the *queue*. Each queue in the Capacity Scheduler has the following properties:

- A short queue name.
- A full queue path name.
- A list of associated child-queues and applications.
- The guaranteed capacity of the queue.
- The maximum capacity of the queue.
- A list of active users and their corresponding resource allocation limits.
- The state of the queue.
- Access control lists (ACLs) governing access to the queue. The following sections will describe how to configure these properties, and how Capacity Scheduler uses these properties to make various scheduling decisions.

2.2. Enabling Capacity Scheduler

To enable the Capacity Scheduler, set the following property in the `/etc/hadoop/conf/yarn-site.xml` file on the ResourceManager host:

Property	**Value**
`yarn.resourcemanager.scheduler.class`	`org.apache.hadoop.yarn.server.resourcema`

The settings for the Capacity Scheduler are contained in the `/etc/hadoop/conf/capacity-scheduler.xml` file on the ResourceManager host. The Capacity Scheduler reads this file when starting, and also when an administrator modifies the capacity-scheduler.xml file and then reloads the settings by running the following command:

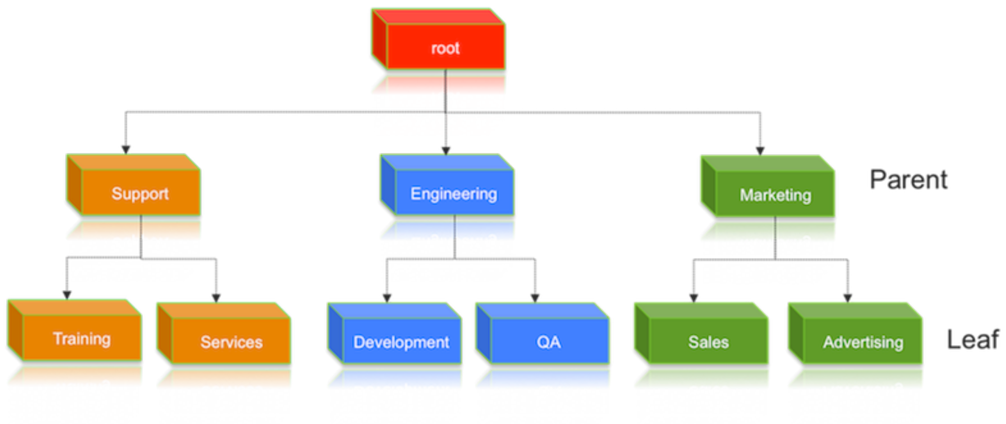
```
yarn rmadmin -refreshQueues
```

This command can only be run by cluster administrators. Administrator privileges are configured with the `yarn.admin.acl` property on the ResourceManager.

2.3. Setting up Queues

The fundamental unit of scheduling in YARN is a *`queue`*. The *`capacity`* of each queue specifies the percentage of cluster resources that are available for applications submitted to the queue. Queues can be set up in a hierarchy that reflects the database structure, resource requirements, and access restrictions required by the various organizations, groups, and users that utilize cluster resources.

For example, suppose that a company has three organizations: Engineering, Support, and Marketing. The Engineering organization has two sub-teams: Development and QA. The Support organization has two sub-teams: Training and Services. And finally, the Marketing organization is divided into Sales and Advertising. The following image shoes the queue hierarchy for this example:



Each child queue is tied to its parent queue with the `yarn.scheduler.capacity.<queue-path>.queues` configuration property in the `capacity-scheduler.xml` file. The top-level "support", "engineering", and "marketing" queues would be tied to the "root" queue as follows:

Property: `yarn.scheduler.capacity.root.queues`

Value: `support,engineering,marketing`

Example:

```
<property>
  <name>yarn.scheduler.capacity.root.queues</name>
  <value>support,engineering,marketing</value>
  <description>The top-level queues below root.</description>
</property>
```

Similarly, the children of the "support" queue would be defined as follows:

Property: `yarn.scheduler.capacity.support.queues`

Value: `training,services`

Example:

```
<property>
  <name>yarn.scheduler.capacity.support.queues</name>
  <value>training,services</value>
  <description>child queues under support</description>
</property>
```

The children of the "engineering" queue would be defined as follows:

Property: `yarn.scheduler.capacity.engineering.queues`

Value: `development,qa`

Example:

```
<property>
  <name>yarn.scheduler.capacity.engineering.queues</name>
  <value>development,qa</value>
  <description>child queues under engineering</description>
</property>
```

And the children of the "marketing" queue would be defined as follows:

Property: `yarn.scheduler.capacity.marketing.queues`

Value: `sales,advertising`

Example:

```
<property>
  <name>yarn.scheduler.capacity.marketing.queues</name>
  <value>sales,advertising</value>
  <description>child queues under marketing</description>
</property>
```

Hierarchical Queue Characteristics

- There are two types of queues: *`parent`* queues and *`leaf`* queues.

- Parent queues enable the management of resources across organizations and sub-organizations. They can contain more parent queues or leaf queues. They do not themselves accept any application submissions directly.
- Leaf queues are the queues that live under a parent queue and accept applications. Leaf queues do not have any child queues, and therefore do not have any configuration property that ends with ".queues".
- There is a top-level parent `root` queue that does not belong to any organization, but instead represents the cluster itself.
- Using parent and leaf queues, administrators can specify capacity allocations for various organizations and sub-organizations.

Scheduling Among Queues

Hierarchical queues ensure that guaranteed resources are first shared among the sub-queues of an organization before any remaining free resources are shared with queues belonging to other organizations. This enables each organization to have control over the utilization of its guaranteed resources.

- At each level in the hierarchy, every parent queue keeps the list of its child queues in a sorted manner based on demand. The sorting of the queues is determined by the currently used fraction of each queue’s capacity (or the full-path queue names if the reserved capacity of any two queues is equal).
- The root queue understands how the cluster capacity needs to be distributed among the first level of parent queues and invokes scheduling on each of its child queues.
- Every parent queue applies its capacity constraints to all of its child queues.
- Leaf queues hold the list of active applications (potentially from multiple users) and schedules resources in a FIFO (first-in, first-out) manner, while at the same time adhering to capacity limits specified for individual users.

2.4. Controlling Access to Queues with ACLs

Access-control lists (ACLs) can be used to restrict user and administrator access to queues. Application submission can really only happen at the leaf queue level, but an ACL restriction set on a parent queue will be applied to all of its descendant queues.

In the Capacity Scheduler, ACLs are configured by granting queue access to a list of users and groups with the `acl_submit_applications` property. The format of the list is "user1,user2 group1,group2" – a comma-separated list of users, followed by a space, followed by a comma-separated list of groups.

The value of `acl_submit_applications` can also be set to "*" (asterisk) to allow access to all users and groups, or can be set to " " (space character) to block access to all users and groups. ACLs for this property are inherited from the parent queue if not specified. The default is "*" for the root queue if not specified.

As mentioned previously, ACL settings on a parent queue are applied to all of its descendant queues. Therefore, if the parent queue uses the "*" (asterisk) value (or is not specified) to allow access to all users and groups, its child queues cannot restrict access. Similarly, before you can restrict access to a child queue, you must first set the parent queue to " " (space character) to block access to all users and groups.

For example, the following properties would set the root `acl_submit_applications` value to " " (space character) to block access to all users and groups, and also restrict access to its child "support" queue to the users "sherlock" and "pacioli" and the members of the "cfo-group" group:

```
<property>
  <name>yarn.scheduler.capacity.root.acl_submit_applications</name>
  <value> </value>
</property>

<property>
  <name>yarn.scheduler.capacity.root.support.acl_submit_applications</name>
  <value>sherlock,pacioli cfo-group</value>
</property>
```

A separate ACL can be used to control the administration of queues at various levels. Queue administrators can submit applications to the queue, kill applications in the queue, and obtain information about any application in the queue (whereas normal users are restricted from viewing all of the details of other users' applications).

Administrator ACLs are configured with the `acl_administer_queue` property. ACLs for this property are inherited from the parent queue if not specified. For example, the following properties would set the root `acl_administer_queue` value to " " (space character) to block access to all users and groups, and also grant administrator access to its child "support" queue to the users "sherlock" and "pacioli" and the members of the "cfo-group" group:

```
<property>
  <name>yarn.scheduler.capacity.root.acl_administer_queue</name>
  <value> </value>
</property>

<property>
  <name>yarn.scheduler.capacity.root.support.acl_administer_queue</name>
  <value>sherlock,pacioli cfo-group</value>
</property>
```

2.5. Managing Cluster Capacity with Queues

The Capacity Scheduler is designed to allow organizations to share compute clusters using the very familiar notion of FIFO (first-in, first-out) queues. YARN does not assign entire nodes to queues. Queues own a fraction of the capacity of the cluster, and this specified queue capacity can be fulfilled from any number of nodes in a dynamic fashion.

Scheduling is the process of matching resource requirements – of multiple applications from various users, and submitted to different queues at multiple levels in the queue hierarchy – with the free capacity available on the nodes in the cluster. Because total cluster capacity can vary, capacity configuration values are expressed as percents.

The `capacity` property can be used by administrators to allocate a percentage of cluster capacity to a queue. The following properties would divide the cluster resources between the Engineering, Support, and Marketing organizations in a 6:1:3 ratio (60%, 10%, and 30%).

Property: `yarn.scheduler.capacity.root.engineering.capacity`

Value: `60`

Property: `yarn.scheduler.capacity.root.support.capacity`

Value: `10`

Property: `yarn.scheduler.capacity.root.marketing.capacity`

Value: `30`

Now suppose that the Engineering group decides to split its capacity between the Development and QA sub-teams in a 1:4 ratio. That would be implemented by setting the following properties:

Property: `yarn.scheduler.capacity.root.engineering.development.capacity`

Value: `20`

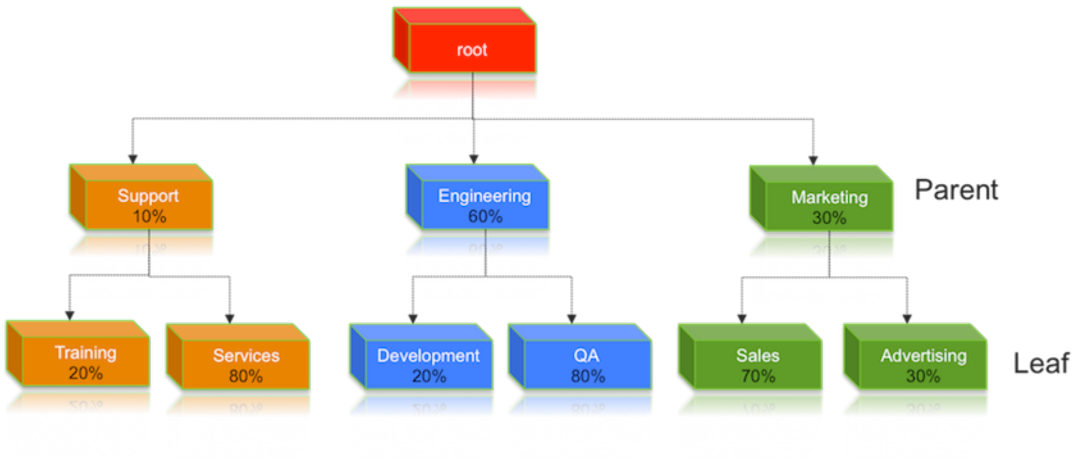
Property: `yarn.scheduler.capacity.root.engineering.qa.capacity`

Value: `80`

Note

The sum of capacities at any level in the hierarchy must equal 100%. Also, the capacity of an individual queue at any level in the hierarchy must be 1% or more (you cannot set a capacity to a value of 0).

The following image illustrates this cluster capacity configuration:



Resource Distribution Workflow

During scheduling, queues at any level in the hierarchy are sorted in the order of their current used capacity, and available resources are distributed among them starting with queues that are currently the most under-served. With respect to capacities alone, the resource scheduling has the following workflow:

- The more under-served a queue is, the higher the priority it receives during resource allocation. The most under-served queue is the queue with the least ratio of used capacity as compared to the total cluster capacity.
  - The used capacity of any parent queue is defined as the aggregate sum of used capacity of all of its descendant queues, recursively.
  - The used capacity of a leaf queue is the amount of resources used by the allocated Containers of all of the applications running in that queue.
- Once it is decided to give a parent queue the currently available free resources, further scheduling is done recursively to determine which child queue gets to use the resources – based on the previously described concept of used capacities.
- Further scheduling happens inside each leaf queue to allocate resources to applications in a FIFO order.
  - This is also dependent on locality, user level limits, and application limits.
  - Once an application within a leaf queue is chosen, scheduling also happens within the application. Applications may have different priorities of resource requests.
- To ensure elasticity, capacity that is configured but not utilized by any queue due to lack of demand is automatically assigned to the queues that are in need of resources.

Resource Distribution Workflow Example

Suppose our cluster has 100 nodes, each with 10 GB of memory allocated for YARN Containers, for a total cluster capacity of 1000 GB (1 TB). According to the previously described configuration, the Engineering organization is assigned 60% of the cluster capacity, i.e., an absolute capacity of 600 GB. Similarly, the Support organization is assigned 100 GB, and the Marketing organization gets 300 GB.

Under the Engineering organization, capacity is distributed between the Development team and the QA team in a in a 1:4 ratio. So Development gets 120 GB, and 480 GB is assigned to QA.

Now consider the following timeline of events:

- Initially, the entire "engineering" queue is free with no applications running, while the "support" and "marketing" queues are utilizing their full capacities.
- Users Sid and Hitesh first submit applications to the "development" leaf queue. Their applications are elastic and can run with either all of the resources available in the cluster, or with a subset of cluster resources (depending upon the state of the resource-usage).
  - Even though the "development" queue is allocated 120 GB, Sid and Hitesh are each allowed to occupy 120 GB, for a total of 240 GB.
  - This can happen despite the fact that the "development" queue is configured to be run with a capacity of 120 GB. Capacity Scheduler allows elastic sharing of cluster resources

for better utilization of available cluster resources. Since there are no other users in the "engineering" queue, Sid and Hitesh are allowed to use the available free resources.

- Next, users Jian, Zhijie and Xuan submit more applications to the "development" leaf queue. Even though each is restricted to 120 GB, the overall used capacity in the queue becomes 600 GB – essentially taking over all of the resources allocated to the "qa" leaf queue.
- User Gupta now submits an application to the "qa" queue. With no free resources available in the cluster, his application must wait.
  - Given that the "development" queue is utilizing all of the available cluster resources, Gupta may or may not be able to immediately get back the guaranteed capacity of his "qa" queue – depending upon whether or not preemption is enabled.
- As the applications of Sid, Hitesh, Jian, Zhijie, and Xuan finish running and resources become available, the newly available Containers will be allocated to Gupta's application. This will continue until the cluster stabilizes at the intended 1:4 resource usage ratio for the "development" and "qa" queues.

From this example, you can see that it is possible for abusive users to submit applications continuously, and thereby lock out other queues from resource allocation until Containers finish running or get preempted. To avoid this scenario, Capacity Scheduler supports limits on the elastic growth of any queue. For example, to restrict the "development" queue from monopolizing the "engineering" queue capacity, an administrator can set a the `maximum-capacity` property:

Property: `yarn.scheduler.capacity.root.engineering.development.maximum-capacity`

Value: `40`

Once this is set, users of the "development" queue can still go beyond their capacity of 120 GB, but they will not be allocated any more than 40% of the "engineering" parent queue's capacity (i.e., 40% of 600 GB = 240 GB).

The `capacity` and `maximum-capacity` properties can be used to control sharing and elasticity across the organizations and sub-organizations utilizing a YARN cluster. Administrators should balance these properties to avoid strict limits that result in a loss of utilization, and to avoid excessive cross-organization sharing.

Capacity and maximum capacity settings can be dynamically changed at run-time using `yarn rmadmin -refreshQueues`.

2.6. Setting User Limits

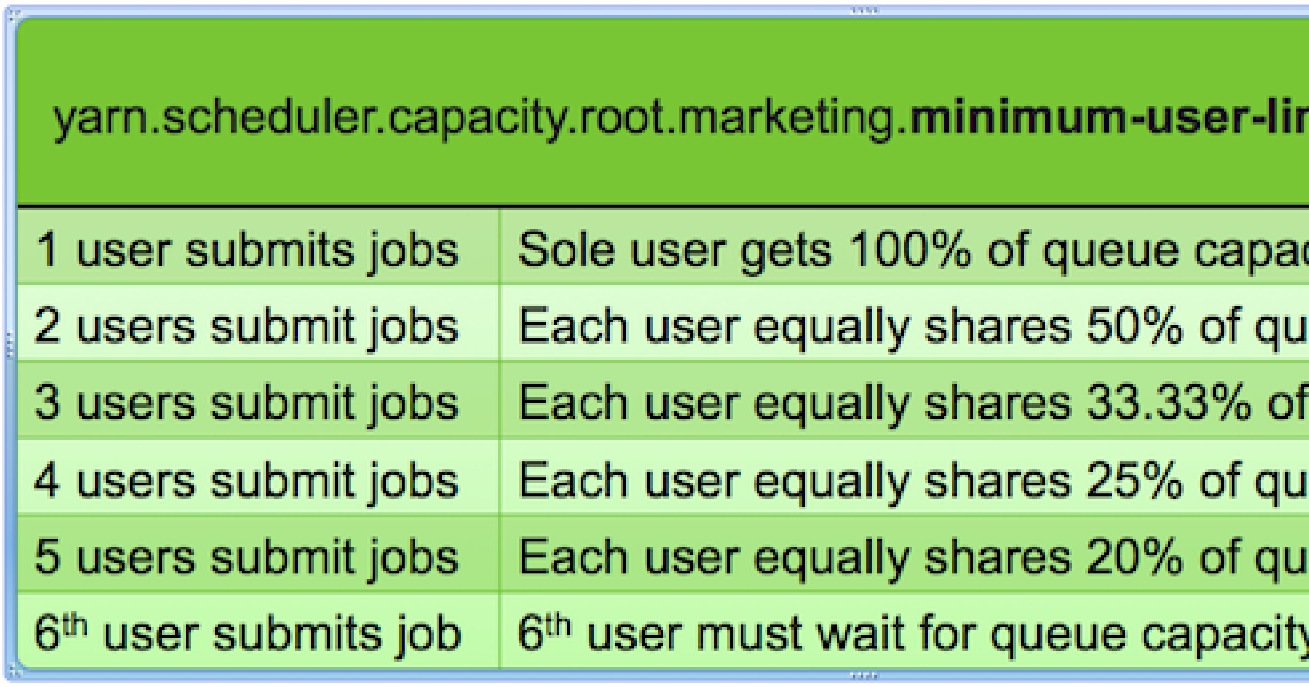
The `minimum-user-limit-percent` property can be used to set the minimum percentage of resources allocated to each leaf queue user. For example, to enable equal sharing of the "services" leaf queue capacity among five users, you would set the minimum-user-limit property to 20%:

Property: `yarn.scheduler.capacity.root.support.services.minimum-user-limit-percent`

Value: `20`

This setting determines the minimum limit that any user's share of the queue capacity can shrink to. Irrespective of this limit, any user can come into the queue and take more than his or her allocated share if there are idle resources available.

The following table shows how the queue resources are adjusted as users submit jobs to a queue with a `minimum-user-limit-percent` value of 20%:



yarn.scheduler.capacity.root.marketing.**minimum-user-li**	
1 user submits jobs	Sole user gets 100% of queue capac
2 users submit jobs	Each user equally shares 50% of qu
3 users submit jobs	Each user equally shares 33.33% of
4 users submit jobs	Each user equally shares 25% of qu
5 users submit jobs	Each user equally shares 20% of qu
6th user submits job	6th user must wait for queue capacity

- The Capacity Scheduler also manages resources for decreasing numbers of users. As users' applications finish running, other existing users with outstanding requirements begin to reclaim that share.

- Note that despite this sharing among users, the FIFO application scheduling order of Capacity Scheduler does not change. This guarantees that users cannot monopolize queues by submitting new applications continuously. Applications (and thus the corresponding users) that are submitted first always get a higher priority than applications that are submitted later.

Capacity Scheduler's leaf queues can also use the `user-limit-factor` property to control user resource allocations. This property denotes the fraction of queue capacity that any single user can consume up to a maximum value, regardless of whether or not there are idle resources in the cluster.

Property: `yarn.scheduler.capacity.root.support.user-limit-factor`

Value: `1`

The default value of "1" means that any single user in the queue can at maximum only occupy the queue's configured capacity. This prevents users in a single queue from monopolizing resources across all queues in a cluster. Setting the value to "2" would restrict the queue's users to twice the queue’s configured capacity. Setting it to a value of 0.5 would restrict any user from using resources beyond half of the queue capacity.

These settings can also be dynamically changed at run-time using `yarn rmadmin -refreshQueues`.

2.7. Application Reservations

The Capacity Scheduler is responsible for matching free resources in the cluster with the resource requirements of an application. Many times, a scheduling cycle occurs such that even though there are free resources on a node, they are not sized large enough to satisfy the application waiting for a resource at the head of the queue. This typically happens with high-memory applications whose resource demand for Containers is much larger than the typical application running in the cluster. This mismatch can lead to starving these resource-intensive applications.

The Capacity Scheduler *`reservations`* feature addresses this issue:

- When a node reports in with a finished Container, the Capacity Scheduler selects an appropriate queue to utilized the newly available resources based on capacity and maximum capacity settings.
- Within that selected queue, the Capacity Scheduler looks at the applications in a FIFO order along with the user limits. Once a needy application is found, the Capacity Scheduler tries to see if the requirements of that application can be met by the node’s free capacity.
- If there is a size mismatch, the Capacity Scheduler immediately creates a reservation on the node for the application’s required Container.
- Once a reservation is made for an application on a node, those resources are not used by the Capacity Scheduler for any other queue, application, or Container until the application reservation is fulfilled.
- The node on which a reservation is made reports back when enough Containers finish running such that the total free capacity on the node now matches the reservation size. When that happens, the Capacity Scheduler marks the reservation as fulfilled, removes it, and allocates a Container on the node.
- In some cases another node fulfills the resources required by the application, so the application no longer needs the reserved capacity on the first node. In this situation, the reservation is simply cancelled.

To prevent the number of reservations from growing in an unbounded manner, and to avoid any potential scheduling deadlocks, the Capacity Scheduler maintains only one active reservation at a time on each node.

2.8. Starting and Stopping Queues

Queues in YARN can be in two states: `RUNNING` or `STOPPED`. A `RUNNING` state indicates that a queue can accept application submissions, and a `STOPPED` queue does not accept application submissions. The default state of any configured queue is `RUNNING`.

In Capacity Scheduler, parent queues, leaf queues, and the root queue can all be stopped. For an application to be accepted at any leaf queue, all the queues in the hierarchy all the way up to the root queue must be running. This means that if a parent queue is stopped, all of the descendant queues in that hierarchy are inactive, even if their own state is `RUNNING`.

The following example sets the value of the `state` property of the "support" queue to RUNNING:

Property: `yarn.scheduler.capacity.root.support.state`

Value: `RUNNING`

Administrators can use the ability to stop and drain applications in a queue for a number of reasons, such as when decommissioning a queue and migrating its users to other queues. Administrators can stop queues at run-time, so that while current applications run to completion, no new applications are admitted. Existing applications can continue until they finish running, and thus the queue can be drained gracefully without any end-user impact.

Administrators can also restart the stopped queues by modifying the `state` configuration property and then refreshing the queue using `yarn rmadmin -refreshQueues` as previously described.

2.9. Setting Application Limits

To avoid system-thrash due to an unmanageable load – caused either by malicious users, or by accident – the Capacity Scheduler enables you to place a static, configurable limit on the total number of concurrently active (both running and pending) applications at any one time. The `maximum-applications` configuration property is used to set this limit, with a default value of 10,000:

Property: `yarn.scheduler.capacity.maximum-applications`

Value: `10000`

The limit for running applications in any specific queue is a fraction of this total limit, proportional to its capacity. This is a hard limit, which means that once this limit is reached for a queue, any new applications to that queue will be rejected, and clients will have to wait and retry later. This limit can be explicitly overridden on a per-queue basis with the following configuration property:

Property: `yarn.scheduler.capacity.<queue-path>.maximum-applications`

Value: `absolute-capacity * yarn.scheduler.capacity.maximum-applications`

There is another resource limit that can be used to set a maximum percentage of cluster resources allocated specifically to ApplicationMasters. The `maximum-am-resource-percent` property has a default value of 10%, and exists to avoid cross-application deadlocks where significant resources in the cluster are occupied entirely by the Containers running ApplicationMasters. This property also indirectly controls the number of concurrent running applications in the cluster, with each queue limited to a number of running applications proportional to its capacity.

Property: `yarn.scheduler.capacity.maximum-am-resource-percent`

Value: `0.1`

As with `maximum-applications`, this limit can also be overridden on a per-queue basis:

Property: `yarn.scheduler.capacity.<queue-path>.maximum-am-resource-percent`

Value: `0.1`

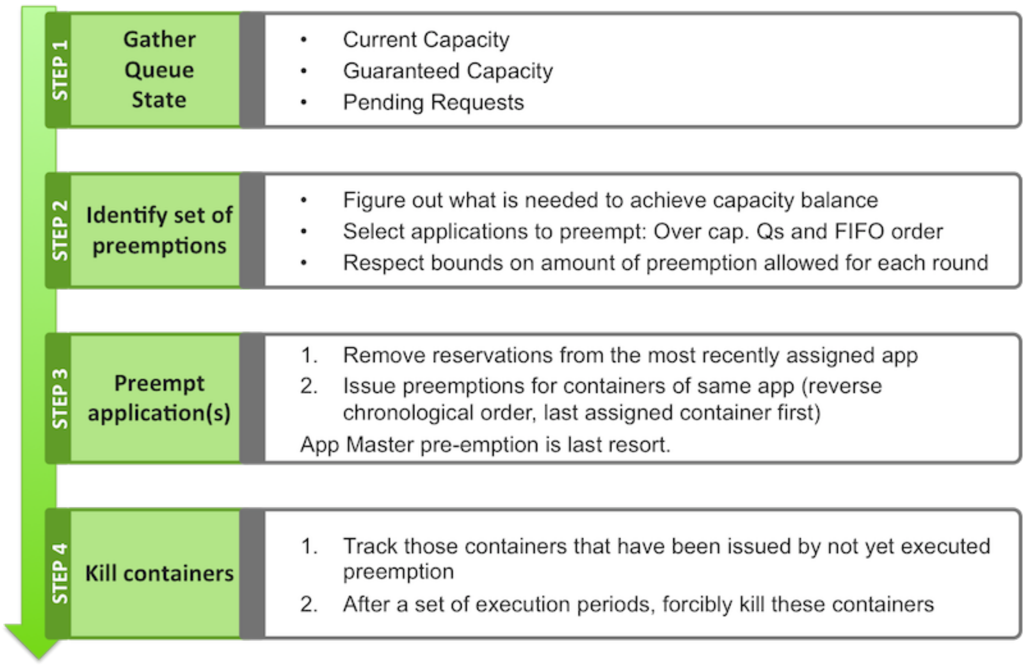
All of these limits ensure that no single application, user, or queue can cause catastrophic failure, or monopolize the cluster and cause excessive degradation of cluster performance.

2.10. Preemption

As mentioned previously, a scenario can occur in which a queue has a guaranteed level of cluster resources, but must wait to run applications because other queues are utilizing all of the available resources. If Preemption is enabled, higher-priority applications do not have to wait because lower priority applications have taken up the available capacity. With Preemption enabled, under-served queues can begin to claim their allocated cluster resources almost immediately, without having to wait for other queues' applications to finish running.

Preemption Workflow

Preemption is governed by a set of capacity monitor policies, which must be enabled by setting the `yarn.resourcemanager.scheduler.monitor.enable` property to "true". These capacity monitor policies apply Preemption in configurable intervals based on defined capacity allocations, and in as graceful a manner as possible. Containers are only killed as a last resort. The following image demonstrates the Preemption workflow:


STEP 1
Gather Queue State
• Current Capacity
• Guaranteed Capacity
• Pending Requests
STEP 2
Identify set of preemptions
• Figure out what is needed to achieve capacity balance
• Select applications to preempt: Over cap. Qs and FIFO order
• Respect bounds on amount of preemption allowed for each round
STEP 3
Preempt application(s)
1. Remove reservations from the most recently assigned app
2. Issue preemptions for containers of same app (reverse chronological order, last assigned container first)
App Master pre-emption is last resort.
STEP 4
Kill containers
1. Track those containers that have been issued by not yet executed preemption
2. After a set of execution periods, forcibly kill these containers

Preemption Configuration

The following properties in the `/etc/hadoop/conf/yarn-site.xml` file on the ResourceManager host are used to enable and configure Preemption.

- **Property:** `yarn.resourcemanager.scheduler.monitor.enable`

  Value: `true`

  Description: Setting this property to "true" enables Preemption. It enables a set of periodic monitors that affect the Capacity Scheduler. This default value for this property is "false" (disabled).

- **Property:** `yarn.resourcemanager.scheduler.monitor.policies`

  Value: `org.apache.hadoop.yarn.server.resourcemanager.monitor.capacity.ProportionalCa`

  Description: The list of SchedulingEditPolicy classes that interact with the scheduler. The only policy currently available for preemption is the “ProportionalCapacityPreemptionPolicy”.

- **Property:** `yarn.resourcemanager.monitor.capacity.preemption.monitoring_interval`

  Value: `3000`

  Description: The time in milliseconds between invocations of this policy. Setting this value to a longer time interval will cause the Capacity Monitor to run less frequently.

- **Property:** `yarn.resourcemanager.monitor.capacity.preemption.max_wait_before_kill`

  Value: `15000`

  Description: The time in milliseconds between requesting a preemption from an application and killing the container. Setting this to a higher value will give applications more time to respond to preemption requests and gracefully release Containers.

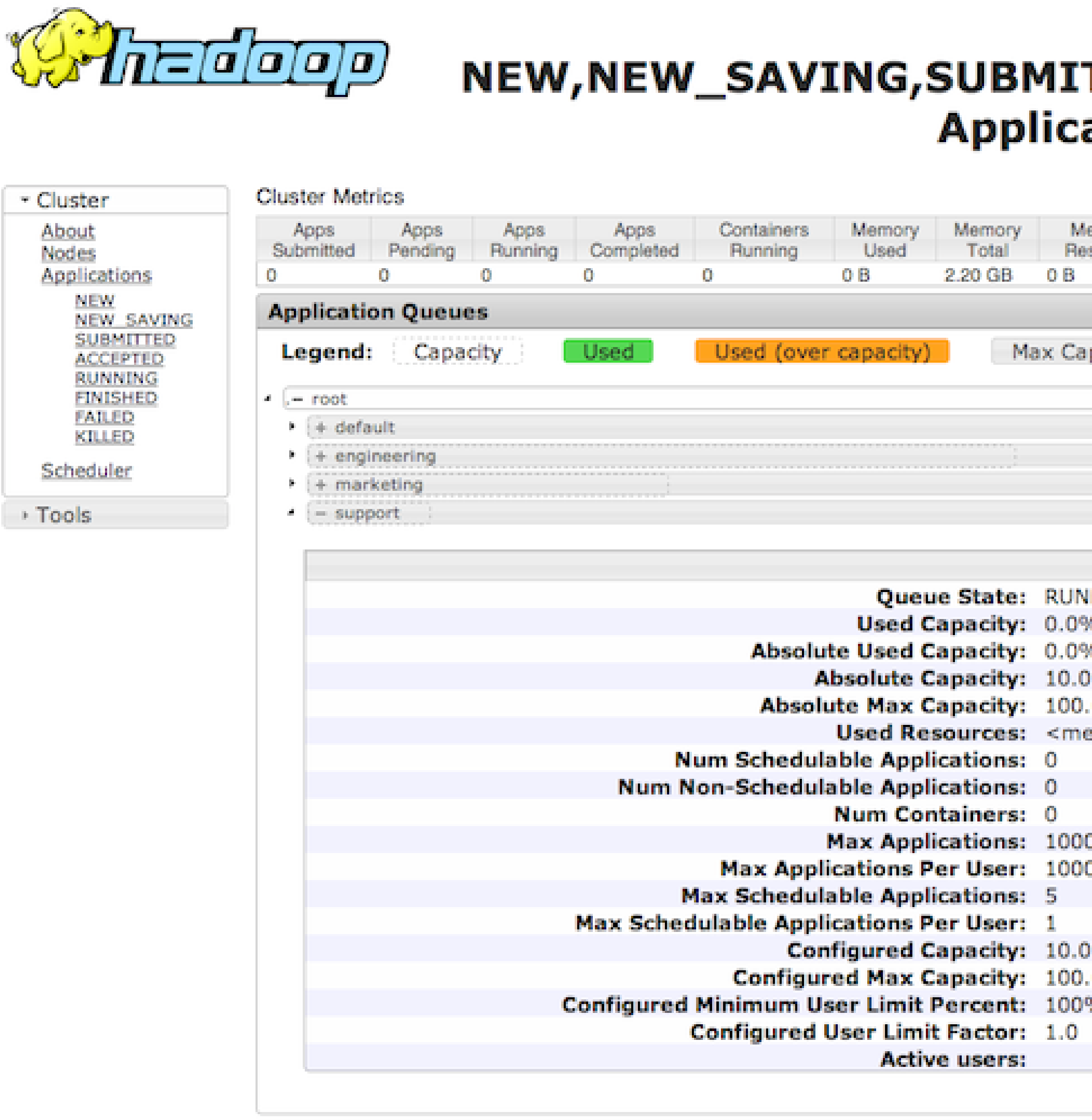
- **Property:** `yarn.resourcemanager.monitor.capacity.preemption.total_preemption_per_round`

  Value: `0.1`

  Description: The maximum percentage of resources preempted in a single round. You can use this value to restrict the pace at which Containers are reclaimed from the cluster. After computing the total desired preemption, the policy scales it back to this limit.

2.11. Scheduler User Interface

You can use the Scheduler page in the Hadoop User Interface (UI) page to view the status and settings of Capacity Scheduler queues. The following image show the Hadoop UI Scheduler page (`http://<hostname>:8088/cluster/scheduler`) with the "support" queue selected:



Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Me Res
0	0	0	0	0	0 B	2.20 GB	0 B

Application Queues

Legend: Capacity | Used | Used (over capacity) | Max Ca

- root
  - default
  - engineering
  - marketing
  - support

Queue State:	RUN
Used Capacity:	0.0%
Absolute Used Capacity:	0.0%
Absolute Capacity:	10.0
Absolute Max Capacity:	100.
Used Resources:	<me
Num Schedulable Applications:	0
Num Non-Schedulable Applications:	0
Num Containers:	0
Max Applications:	1000
Max Applications Per User:	1000
Max Schedulable Applications:	5
Max Schedulable Applications Per User:	1
Configured Capacity:	10.0
Configured Max Capacity:	100.
Configured Minimum User Limit Percent:	100%
Configured User Limit Factor:	1.0
Active users:	

3. Centralized Cache Management in HDFS

This guide 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.

In this document:

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 (OS) 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.

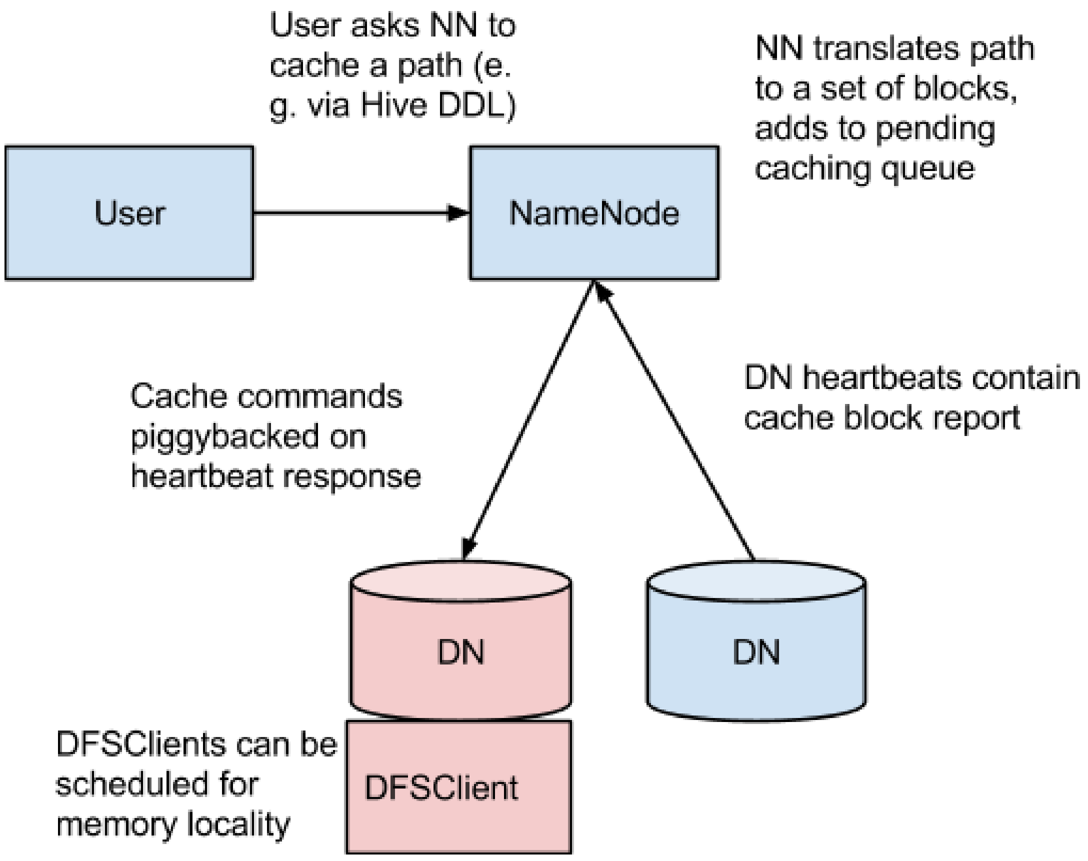
3.2. Caching Use Cases

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.


User asks NN to cache a path (e. g. via Hive DDL)
NN translates path to a set of blocks, adds to pending caching queue
User
NameNode
Cache commands piggybacked on heartbeat response
DN heartbeats contain cache block report
DN
DN
DFSClients can be scheduled for memory locality
DFSClient

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
- Configuration Properties
- OS Limits

3.5.1. 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.

3.5.2. 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 OS Limits). 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>
```

3.5.3. 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 will know that 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 with `dfs.datanode.max.locked.memory`, or the string "unlimited", which indicates that there is no limit. Note that it is typical for `ulimit -l` to output 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>
```

You would 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
```

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
- Cache Directive Commands

3.6.1. 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:

Option	**Description**
`<name>`	The name of the new 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 permissior
`<limit>`	The maximum number of bytes that can be cached by directives in the pool, in aggregate. By default, no limit
`<maxTtl>`	The maximum allowed time-to-live for directives being added to the pool. This can be specified in seconds, mi 120s, 30m, 4h, 2d. Valid units are [smhd]. By default, no maximum is set. A value of "never" specifies that the

- **`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:

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, mi 120s, 30m, 4h, 2d. Valid units are [smhd]. A value of "never" specifies that there is no limit.

- **`removePool`**

  Remove a Cache Pool. This also un-caches paths associated with the pool.

  Usage:

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

  Options:

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

- **`listPools`**

  Displays information about one or more Cache Pools, e.g., name, owner, group, permissions, etc.

  Usage:

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

  Options:

Option	Description
`-stats`	Display additional Cache Pool statistics.
`<name>`	If specified, list only the named Cache Pool.

- **`help`**

  Displays detailed information about a command.

  Usage:

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

Options:

Option	Description
`<command-name>`	Displays detailed information for the specified command name. If no command name is specified, detailed he

3.6.2. 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:

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 Po directives.
`<-force>`	Skips checking of Cache Pool resource limits.
`<replication>`	The UNIX-style permissions assigned to the pool. Permissions are specified in octal, e.g., 0755. Pool permissior
`<limit>`	The cache replication factor to use. Defaults to 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 indicates a directive that never expires. If unspecified, the directive never expires.

- **`removeDirective`**

  Removes a Cache Directive.

  Usage:

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

  Options:

Option	Description
`<id>`	The ID of the Cache Directive to remove. You must have Write permission for the pool that the directive belo can use the `-listDirectives` command to display a list of Cache Directive IDs.

- **`removeDirectives`**

  Removes all of the Cache Directives in a specified path.

  Usage:

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

  Options:

Option	Description

`<path>`	The path of the Cache Directives to remove. You must have Write permission for the pool that the directives 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:

Option	**Description**
`<path>`	List only the Cache Directives in this path. Note that if there is a Cache Directive in the `<path>` that belongs t not have Read access, it will not be listed.
`<pool>`	Only list the Cache Directives in the specified Cache Pool.
`<-stats>`	List path-based Cache Directive statistics.

4. Configuring Rack Awareness on HDP

Use the following instructions to configure rack awareness on a HDP cluster:

1. Create a Rack Topology Script
2. Add Properties to `core-site.xml`
3. Restart HDFS and MapReduce
4. Verify Rack Awareness

4.1. 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

   File name: `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

File name: `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.

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

1. Stop HDFS using the instructions on this page.

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>
```

4.3. Restart HDFS and MapReduce

Restart HDFS and MapReduce using the instructions on this page.

4.4. Verify Rack Awareness

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

- 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>
```

- 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
```

- 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 (partial) 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
```

5. Using DistCp to Copy Files

Hadoop DistCp (distributed copy) can be used to copy data between Hadoop clusters (and also within a Hadoop cluster). DistCp uses MapReduce to implement its distribution, error handling, and reporting. It expands a list of files and directories into map tasks, each of which copies a partition of the files specified in the source list.

In this guide:

5.1. Using DistCp

The most common use of DistCp is an inter-cluster copy:

```
hadoop distcp hdfs://nn1:8020/source hdfs://nn2:8020/destination
```

Where `hdfs://nn1:8020/source` is the data source, and `hdfs://nn2:8020/destination` is the destination. This will expand the name space under `/source` on NameNode "nn1" into a temporary file, partition its contents among a set of map tasks, and start copying from "nn1" to "nn2". Note that DistCp requires absolute paths.

You can also specify multiple source directories:

```
hadoop distcp hdfs://nn1:8020/source/a hdfs://nn1:8020/source/b hdfs://
nn2:8020/destination
```

Or specify multiple source directories from a file with the -f option:

```
hadoop distcp -f hdfs://nn1:8020/srclist hdfs://nn2:8020/destination
```

Where `srclist` contains:

```
hdfs://nn1:8020/source/a
hdfs://nn1:8020/source/b
```

DistCp from HDP-1.3.x to HDP-2.x

When using DistCp to copy from a HDP-1.3.x cluster to a HDP-2.x cluster, the format would be:

```
hadoop distcp hftp://<hdp 1.3.x namenode host>:50070/<folder path of source>
 hdfs://<hdp 2.x namenode host>/<folder path of target>
```

Here is an example of a DistCp copy from HDP 1.3.0 to HDP-2.0:

```
hadoop distcp hftp://namenodehdp130.test.com:50070/apps/hive/warehouse/db/
 hdfs://namenodehdp20.test.com/data/raw/
```

When copying from multiple sources, DistCp will abort the copy with an error message if two sources collide, but collisions at the destination are resolved based on the options specified. By default, files already existing at the destination are skipped (i.e. not replaced by the source file). A count of skipped files is reported at the end of each job, but it may be inaccurate if a copier failed for some subset of its files, but succeeded on a later attempt.

It is important that each NodeManager is able to communicate with both the source and destination file systems. For HDFS, both the source and destination must be running the same version of the protocol, or use a backwards-compatible protocol; see Copying Between Versions.

After a copy, you should generate and cross-check a listing of the source and destination to verify that the copy was truly successful. Since DistCp employs both Map/Reduce and the FileSystem API, issues in or between any of these three could adversely and silently affect the copy. Some have had success running with -update enabled to perform a second pass, but users should be acquainted with its semantics before attempting this.

It is also worth noting that if another client is still writing to a source file, the copy will likely fail. Attempting to overwrite a file being written at the destination should also fail on HDFS. If a source file is (re)moved before it is copied, the copy will fail with a FileNotFound exception.

5.2. Command Line Options

Flag	Description	Notes
`-p[rbugpca]`	Preserve r: replication number b: block size u: user g: group p: permission c: checksum-type a: ACL	Modification times are not preserved. Also, when `-update` is specified, status updates will **not** be synchronized unless the file sizes also differ (i.e. unless the file is recreated). If `-pa` is specified, DistCp also preserves the permissions because ACLs are a super-set of permissions.
`-i`	Ignore failures	This option will keep more accurate statistics about the copy than the default case. It also preserves logs from failed copies, which can be valuable for debugging. Finally, a failing map will not cause the job to fail before all splits are attempted.
`-log <logdir>`	Write logs to `<logdir>`	DistCp keeps logs of each file it attempts to copy as map output. If a map fails, the log output will not be retained if it is re-executed.
`-m <num_maps>`	Maximum number of simultaneous copies	Specify the number of maps to copy data. Note that more maps may not necessarily improve throughput.
`-overwrite`	Overwrite destination	If a map fails and `-i` is not specified, all the files in the split, not only those that failed, will be recopied. As discussed in the Usage documentation, it also changes the semantics for generating destination paths, so users should use this carefully.
`-update`	Overwrite if src size different from dst size	As noted in the preceding, this is not a “sync” operation. The only criterion examined is the source and destination file sizes; if they

Flag	Description	Notes
		differ, the source file replaces the destination file. As discussed in the Usage documentation, it also changes the semantics for generating destination paths, so users should use this carefully.
`-f <urilist_uri>`	Use list at `<urilist_uri>` as src list	This is equivalent to listing each source on the command line. The `urilist_uri` list should be a fully qualified URI.
`-filelimit <n>`	Limit the total number of files to be <= n	**Deprecated!** Ignored in DistCp v2.
`-sizelimit <n>`	Limit the total size to be <= n bytes	**Deprecated!** Ignored in DistCp v2.
`-delete`	Delete the files existing in the dst but not in src	The deletion is done by FS Shell. So the trash will be used, if it is enabled.
`-strategy {dynamic|uniformsize}`	Choose the copy-strategy to be used in DistCp.	By default, `uniformsize` is used. (i.e. Maps are balanced on the total size of files copied by each map. Similar to legacy.) If `dynamic` is specified, DynamicInputFormat is used instead. (This is described in the Architecture section, under InputFormats.)
`-bandwidth`	Specify bandwidth per map, in MB/second.	Each map will be restricted to consume only the specified bandwidth. This is not always exact. The map throttles back its bandwidth consumption during a copy, such that the **net** bandwidth used tends towards the specified value.
`-atomic {-tmp <tmp_dir>}`	Specify atomic commit, with optional tmp directory.	`-atomic` instructs DistCp to copy the source data to a temporary target location, and then move the temporary target to the final location atomically. Data will either be available at final target in a complete and consistent form, or not at all. Optionally, `-tmp` may be used to specify the location of the tmp-target. If not specified, a default is chosen. **Note:** `tmp_dir` must be on the final target cluster.
`-mapredSslConf <ssl_conf_file>`	Specify SSL Config file, to be used with HSFTP source	When using the hsftp protocol with a source, the security-related properties may be specified in a config file and passed to DistCp. `<ssl_conf_file>` needs to be in the classpath.
`-async`	Run DistCp asynchronously. Quits as soon as the Hadoop Job is launched.	The Hadoop Job-id is logged, for tracking.

5.3. Update and Overwrite

The DistCp `-update` option is used to copy files from a source that do not exist at the target, or that have different contents. The DistCp `-overwrite` option overwrites target files even if they exist at the source, or if they have the same contents.

The `-update` and `-overwrite` options warrant further discussion, since their handling of source-paths varies from the defaults in a very subtle manner.

Consider a copy from `/source/first/` and `/source/second/` to `/target/`, where the source paths have the following contents:

```
hdfs://nn1:8020/source/first/1
hdfs://nn1:8020/source/first/2
hdfs://nn1:8020/source/second/10
hdfs://nn1:8020/source/second/20
```

When DistCp is invoked without `-update` or `-overwrite`, the DistCp defaults would create directories `first/` and `second/`, under `/target`. Thus:

```
distcp hdfs://nn1:8020/source/first hdfs://nn1:8020/source/second hdfs://
nn2:8020/target
```

would yield the following contents in `/target`:

```
hdfs://nn2:8020/target/first/1
hdfs://nn2:8020/target/first/2
hdfs://nn2:8020/target/second/10
hdfs://nn2:8020/target/second/20
```

When either -`update` or `-overwrite` is specified, the **contents** of the source directories are copied to the target, and not the source directories themselves. Thus:

```
distcp -update hdfs://nn1:8020/source/first hdfs://nn1:8020/source/second
 hdfs://nn2:8020/target
```

would yield the following contents in `/target`:

```
hdfs://nn2:8020/target/1
hdfs://nn2:8020/target/2
hdfs://nn2:8020/target/10
hdfs://nn2:8020/target/20
```

By extension, if both source folders contained a file with the same name ("0", for example), then both sources would map an entry to `/target/0` at the destination. Rather than permit this conflict, DistCp will abort.

Now, consider the following copy operation:

```
distcp hdfs://nn1:8020/source/first hdfs://nn1:8020/source/second hdfs://
nn2:8020/target
```

With sources/sizes:

```
hdfs://nn1:8020/source/first/1 32
hdfs://nn1:8020/source/first/2 32
hdfs://nn1:8020/source/second/10 64
hdfs://nn1:8020/source/second/20 32
```

And destination/sizes:

```
hdfs://nn2:8020/target/1 32
hdfs://nn2:8020/target/10 32
hdfs://nn2:8020/target/20 64
```

Will effect:

```
hdfs://nn2:8020/target/1 32
hdfs://nn2:8020/target/2 32
hdfs://nn2:8020/target/10 64
hdfs://nn2:8020/target/20 32
```

`1` is skipped because the file-length and contents match. `2` is copied because it doesn't exist at the target. `10` and `20` are overwritten since the contents don't match the source.

If the `-update` option is used, `1` is overwritten as well.

5.4. DistCp and Security Settings

Security settings dictate whether DistCp should be run on the source cluster or the destination cluster. The general rule-of-thumb is that if one cluster is secure and the other is not secure, DistCp should be run from the secure cluster – otherwise there may be security-related issues.

When copying data from a secure cluster to an non-secure cluster, the following configuration setting is required for the DistCp client:

```
<property>
  <name>ipc.client.fallback-to-simple-auth-allowed</name>
  <value>true</value>
</property>
```

When copying data from a secure cluster to a secure cluster, the following configuration setting is required in the `core-site.xml` file:

```
<property>
  <name>hadoop.security.auth_to_local</name>
  <value></value>
  <description>Maps kerberos principals to local user names</description>
</property>
```

Secure-to-Secure: Kerberos Principal Name

- `distcp hdfs://hdp-2.0-secure hdfs://hdp-2.0-secure`

  One issue here is that the SASL RPC client requires that the remote server's Kerberos principal must match the server principal in its own configuration. Therefore, the same principal name must be assigned to the applicable NameNodes in the source and the destination cluster. For example, if the Kerberos principal name of the NameNode in the source cluster is `nn/host1@realm`, the Kerberos principal name of the NameNode in destination cluster must be `nn/host2@realm`, rather than `nn2/host2@realm`, for example.

Secure-to-Secure: ResourceManager Mapping Rules

When copying between two HDP2 secure clusters, or from HDP1 secure to HDP2 secure, further ResourceManager (RM) configuration is required if the two clusters have different realms. In order for DistCP to succeed, the same RM mapping rule must be used in both clusters.

For example, if secure Cluster 1 has the following RM mapping rule:

```
<property>
      <name>hadoop.security.auth_to_local</name>
      <value>
RULE:[2:$1@$0](rm@.*SEC1.SUP1.COM)s/.*/yarn/
DEFAULT
       </value>
</property>
```

And secure Cluster 2 has the following RM mapping rule:

```
<property>
      <name>hadoop.security.auth_to_local</name>
      <value>
RULE:[2:$1@$0](rm@.*BA.YISEC3.COM)s/.*/yarn/
DEFAULT
      </value>
</property>
```

The DistCp job from Cluster 1 to Cluster 2 will fail because Cluster 2 cannot resolve the RM principle of Cluster 1 correctly to the yarn user, because the RM mapping rule in Cluster 2 is different than the RM mapping rule in Cluster 1.

The solution is to use the same RM mapping rule in both Cluster 1 and Cluster 2:

```
<property>
      <name>hadoop.security.auth_to_local</name>
      <value>
RULE:[2:$1@$0](rm@.*SEC1.SUP1.COM)s/.*/yarn/
RULE:[2:$1@$0](rm@.*BA.YISEC3.COM)s/.*/yarn/
DEFAULT</value>
    </property>
```

5.5. DistCp and HDP Version

The HDP version of the source and destination clusters can determine which type of file systems should be used to read the source cluster and write to the destination cluster.

For example, when copying data from a 1.x cluster to a 2.x cluster, it is impossible to use “hdfs” for both the source and the destination, because HDP 1.x and 2.x have different RPC versions, and the client cannot understand both at the same time. In this case the WebHdfsFilesystem (webhdfs://) can be used in both the source and destination clusters, or the HftpFilesystem (hftp://) can be used to read data from the source cluster.

5.6. DistCp Data Copy Matrix: HDP1/HDP2 to HDP2

The following table provides a summary of configuration, settings, and results when using DistCp to copy data from HDP1 and HDP2 clusters to HDP2 clusters.

From	To	Source Configuration	Destination Configuration	DistCp Should be Run on...	Result
HDP 1.3	HDP 2.x	insecure + hdfs	insecure + webhdfs	HDP 1.3 (source)	success
HDP 1.3	HDP 2.x	secure + hdfs	secure + webhdfs	HDP 1.3 (source)	success
HDP 1.3	HDP 2.x	secure + hftp	secure + hdfs	HDP 2.x (destination)	success
HDP 1.3	HDP 2.1	secure + hftp	secure + swebhdfs	HDP 2.1 (destination)	success
HDP 1.3	HDP 2.x	secure + hdfs	insecure + webhdfs	HDP 1.3 (source)	Possible iss are discuss here.
HDP 2.0	HDP 2.x	secure + hdfs	insecure + hdfs	secure HDP 2.0 (source)	success
HDP 2.0	HDP 2.x	secure + hdfs	secure + hdfs	either HDP 2.0 or HDP 2.x (source or destination)	success
HDP 2.0	HDP 2.x	secure + hdfs	secure + webhdfs	HDP 2.0 (source)	success
HDP 2.0	HDP 2.x	secure + hftp	secure + hdfs	HDP 2.x (destination)	success

Note

For the above table:

- The term "secure" means that Kerberos security is set up.

- HDP 2.x means HDP 2.0 and HDP 2.1.
- hsftp is available in both HDP-1.x and HDP-2.x. It adds https support to hftp.

5.7. Copying Data from HDP-2.x to HDP-1.x Clusters

Copying Data from HDP-1.x to HDP-2.x Clusters is also supported, however, HDP-1.x is not aware of a new checksum introduced in HDP-2.x.

To copy data from HDP-2.x to HDP-1.x:

- Skip the checksum check during source 2.x –> 1.x.

  -or-
- Ensure that the file to be copied is in CRC32 before distcp 2.x –> 1.x.

5.8. DistCp Architecture

DistCp is comprised of the following components:

- DistCp Driver
- Copy-listing Generator
- InputFormats and MapReduce Components

5.8.1. DistCp Driver

The DistCp Driver components are responsible for:

- Parsing the arguments passed to the DistCp command on the command-line, via:
  - OptionsParser
  - DistCpOptionsSwitch
- Assembling the command arguments into an appropriate DistCpOptions object, and initializing DistCp. These arguments include:
  - Source-paths
  - Target location
  - Copy options (e.g. whether to update-copy, overwrite, which file attributes to preserve, etc.)
- Orchestrating the copy operation by:
  - Invoking the copy-listing generator to create the list of files to be copied.

- Setting up and launching the Hadoop MapReduce job to carry out the copy.
- Based on the options, either returning a handle to the Hadoop MapReduce job immediately, or waiting until completion.

The parser elements are executed only from the command-line (or if DistCp::run() is invoked). The DistCp class may also be used programmatically, by constructing the DistCpOptions object and initializing a DistCp object appropriately.

5.8.2. Copy-listing Generator

The copy-listing generator classes are responsible for creating the list of files/directories to be copied from source. They examine the contents of the source paths (files/directories, including wildcards), and record all paths that need copying into a SequenceFile for consumption by the DistCp Hadoop Job. The main classes in this module include:

1. **CopyListing**: The interface that should be implemented by any copy-listing generator implementation. Also provides the factory method by which the concrete CopyListing implementation is chosen.
2. **SimpleCopyListing**: An implementation of CopyListing that accepts multiple source paths (files/directories), and recursively lists all of the individual files and directories under each for copy.
3. **GlobbedCopyListing**: Another implementation of CopyListing that expands wildcards in the source paths.
4. **FileBasedCopyListing**: An implementation of CopyListing that reads the source path list from a specified file.

Based on whether a source file list is specified in the DistCpOptions, the source listing is generated in one of the following ways:

1. If there is no source file list, the GlobbedCopyListing is used. All wildcards are expanded, and all of the expansions are forwarded to the SimpleCopyListing, which in turn constructs the listing (via recursive descent of each path).
2. If a source file list is specified, the FileBasedCopyListing is used. Source paths are read from the specified file, and then forwarded to the GlobbedCopyListing. The listing is then constructed as described above.

You can customize the method by which the copy-listing is constructed by providing a custom implementation of the CopyListing interface. The behaviour of DistCp differs here from the legacy DistCp, in how paths are considered for copy.

The legacy implementation only lists those paths that must definitely be copied on to the target. E.g., if a file already exists at the target (and `-overwrite` isn't specified), the file is not even considered in the MapReduce copy job. Determining this during setup (i.e. before the MapReduce Job) involves file size and checksum comparisons that are potentially time consuming.

DistCp postpones such checks until the MapReduce job, thus reducing setup time. Performance is enhanced further since these checks are parallelized across multiple maps.

5.8.3. InputFormats and MapReduce Components

The InputFormats and MapReduce components are responsible for the actual copying of files and directories from the source to the destination path. The listing file created during copy-listing generation is consumed at this point, when the copy is carried out. The classes of interest here include:

- **UniformSizeInputFormat:** This implementation of org.apache.hadoop.mapreduce.InputFormat provides equivalence with Legacy DistCp in balancing load across maps. The aim of the UniformSizeInputFormat is to make each map copy roughly the same number of bytes. Therefore, the listing file is split into groups of paths, such that the sum of file sizes in each InputSplit is nearly equal to every other map. The splitting is not always perfect, but its trivial implementation keeps the setup time low.
- **DynamicInputFormat and DynamicRecordReader:** The DynamicInputFormat implements org.apache.hadoop.mapreduce.InputFormat, and is new to DistCp. The listing file is split into several “chunk files”, the exact number of chunk files being a multiple of the number of maps requested for in the Hadoop Job. Each map task is “assigned” one of the chunk files (by renaming the chunk to the task’s id), before the Job is launched. Paths are read from each chunk using the DynamicRecordReader, and processed in the CopyMapper. After all of the paths in a chunk are processed, the current chunk is deleted and a new chunk is acquired. The process continues until no more chunks are available. This “dynamic” approach allows faster map tasks to consume more paths than slower ones, thus speeding up the DistCp job overall.
- **CopyMapper:** This class implements the physical file copy. The input paths are checked against the input options (specified in the job configuration), to determine whether a file needs to be copied. A file will be copied only if at least one of the following is true:
  - A file with the same name does not exist at target.
  - A file with the same name exists at target, but has a different file size.
  - A file with the same name exists at target, but has a different checksum, and `-skipcrccheck` is not mentioned.
  - A file with the same name exists at target, but `-overwrite` is specified.
  - A file with the same name exists at target, but differs in block-size (and block-size needs to be preserved).
- **CopyCommitter:** This class is responsible for the commit phase of the DistCp job, including:
  - Preservation of directory permissions (if specified in the options)
  - Clean up of temporary files, work directories, etc.

5.9. DistCp Frequently Asked Questions

1. **Why does -update not create the parent source directory under a pre-existing target directory?** The behavior of -update and -overwriteis described in detail in the Using

DistCp section of this document. In short, if either option is used with a pre-existing destination directory, the **contents** of each source directory are copied over, rather than the source directory itself. This behavior is consistent with the legacy DistCp implementation.

2. **How does the new DistCp (version 2) differ in semantics from the legacy DistCp?**

   - Files that are skipped during copy previously also had their file-attributes (permissions, owner/group info, etc.) unchanged, when copied with Legacy DistCp. These are now updated, even if the file copy is skipped.

   - In Legacy DistCp, empty root directories among the source path inputs were not created at the target. These are now created.

3. **Why does the new DistCp (version 2) use more maps than legacy DistCp?** Legacy DistCp works by figuring out what files need to be actually copied to target before the copy job is launched, and then launching as many maps as required for copy. So if a majority of the files need to be skipped (because they already exist, for example), fewer maps will be needed. As a consequence, the time spent in setup (i.e. before the MapReduce job) is higher. The new DistCp calculates only the contents of the source paths. It doesn't try to filter out what files can be skipped. That decision is put off until the MapReduce job runs. This is much faster (vis-a-vis execution-time), but the number of maps launched will be as specified in the -m option, or 20 (the default) if unspecified.

4. **Why does DistCp not run faster when more maps are specified?** At present, the smallest unit of work for DistCp is a file. i.e., a file is processed by only one map. Increasing the number of maps to a value exceeding the number of files would yield no performance benefit. The number of maps launched would equal the number of files.

5. **Why does DistCp run out of memory?** If the number of individual files/directories being copied from the source path(s) is extremely large (e.g. 1,000,000 paths), DistCp might run out of memory while determining the list of paths for copy. This is not unique to the new DistCp implementation. To get around this, consider changing the -Xmx JVM heap-size parameters, as follows:

   ```
   bash$ export HADOOP_CLIENT_OPTS="-Xms64m -Xmx1024m"
   bash$ hadoop distcp /source /target
   ```

5.10. Appendix

In this section:

- Map Sizing
- Copying Between Versions of HDFS
- MapReduce and Other Side-Effects
- SSL Configurations for HSFTP Sources

5.10.1. Map Sizing

By default, DistCp makes an attempt to size each map comparably so that each copies roughly the same number of bytes. Note that files are the finest level of granularity, so increasing the number of simultaneous copiers (i.e. maps) may not always increase the number of simultaneous copies nor the overall throughput.

DistCp also provides a strategy to “dynamically” size maps, allowing faster DataNodes to copy more bytes than slower nodes. Using the dynamic strategy (explained in the Architecture), rather than assigning a fixed set of source files to each map task, files are instead split into several sets. The number of sets exceeds the number of maps, usually by a factor of 2-3. Each map picks up and copies all files listed in a chunk. When a chunk is exhausted, a new chunk is acquired and processed, until no more chunks remain.

By not assigning a source path to a fixed map, faster map tasks (i.e. DataNodes) are able to consume more chunks – and thus copy more data – than slower nodes. While this distribution isn’t uniform, it is fair with regard to each mapper’s capacity.

The dynamic strategy is implemented by the DynamicInputFormat. It provides superior performance under most conditions.

Tuning the number of maps to the size of the source and destination clusters, the size of the copy, and the available bandwidth is recommended for long-running and regularly run jobs.

5.10.2. Copying Between Versions of HDFS

For copying between two different versions of Hadoop, one will usually use HftpFileSystem. This is a read-only FileSystem, so DistCp must be run on the destination cluster (more specifically, on NodeManagers that can write to the destination cluster). Each source is specified as `hftp://<dfs.http.address>/<path>` (the default dfs.http.address is `<namenode>:50070`).

5.10.3. MapReduce and Other Side-Effects

As mentioned previously, should a map fail to copy one of its inputs, there will be several side-effects.

- Unless `-overwrite` is specified, files successfully copied by a previous map will be marked as “skipped” on a re-execution.
- If a map fails mapreduce.map.maxattempts times, the remaining map tasks will be killed (unless -i is set).
- If mapreduce.map.speculative is set final and true, the result of the copy is undefined.

5.10.4. SSL Configurations for HSFTP Sources

To use an HSFTP source (i.e. using the HSFTP protocol), a SSL configuration file needs to be specified (via the -mapredSslConf option). This must specify 3 parameters:

- ssl.client.truststore.location: The local file system location of the trust-store file, containing the certificate for the NameNode.
- ssl.client.truststore.type: (Optional) The format of the trust-store file.
- ssl.client.truststore.password: (Optional) Password for the trust-store file.

The following is an example of the contents of a SSL Configuration file:

```
<configuration>
  <property>
    <name>ssl.client.truststore.location</name>
    <value>/work/keystore.jks</value>
    <description>Truststore to be used by clients like distcp. Must be
 specified.</description>
  </property>

  <property>
    <name>ssl.client.truststore.password</name>
    <value>changeme</value>
    <description>Optional. Default value is "".</description>
  </property>

  <property>
    <name>ssl.client.truststore.type</name>
    <value>jks</value>
    <description>Optional. Default value is "jks".</description>
  </property>
</configuration>
```

The SSL configuration file must be in the classpath of the DistCp program.

6. Decommissioning Slave Nodes

Hadoop provides the decommission feature to retire a set of existing slave nodes (DataNodes, NodeManagers, or HBase RegionServers) in order to prevent data loss.

Slaves nodes are frequently decommissioned for maintainance. As a Hadoop administrator, you will decommission the slave nodes periodically in order to either reduce the cluster size or to gracefully remove dying nodes.

Use the following sections to decommission slave nodes in your cluster:

- Prerequisites
- Decommission DataNodes or NodeManagers
- Decommission HBase RegionServers

6.1. Prerequisites

- Ensure that the following property is defined in your `hdfs-site.xml` file.

  ```
  <property>
     <name>dfs.hosts.exclude</name>
     <value>$HADOOP_CONF_DIR/dfs.exclude</value>
     <final>true</final>
  </property>
  ```

  where *$HADOOP_CONF_DIR* is the directory for storing the Hadoop configuration files. For example, `/etc/hadoop/conf`.

- Ensure that the following property is defined in your `yarn-site.xml` file.

  ```
  <property>
     <name>yarn.resourcemanager.nodes.exclude-path</name>
     <value>$HADOOP_CONF_DIR/yarn.exclude</value>
     <final>true</final>
  </property>
  ```

  where *$HADOOP_CONF_DIR* is the directory for storing the Hadoop configuration files. For example, `/etc/hadoop/conf`.

6.2. Decommission DataNodes or NodeManagers

Nodes normally run both a DataNode and a NodeManager, and both are typically commissioned or decommissioned together.

With the replication level set to three, HDFS is resilient to individual DataNodes failures. However, there is a high chance of data loss when you terminate DataNodes without decommissioning them first. Nodes must be decommissioned on a schedule that permits replication of blocks being decommissioned.

On the other hand, if a NodeManager is shut down, the ResourceManager will reschedule the tasks on other nodes in the cluster. However, decommissioning a NodeManager may be required in situations where you want a NodeManager to stop to accepting new tasks, or when the tasks take time to execute but you still want to be agile in your cluster management.

6.2.1. Decommission DataNodes

Use the following instructions to decommission DataNodes in your cluster:

1. On the NameNode host machine, edit the *$HADOOP_CONF_DIR*/`dfs.exclude` file and add the list of DataNodes hostnames (separated by a newline character).

   where *$HADOOP_CONF_DIR* is the directory for storing the Hadoop configuration files. For example, `/etc/hadoop/conf`.

2. Update the NameNode with the new set of excluded DataNodes. On the NameNode host machine, execute the following command:

   ```
   su $HDFS_USER
   % hdfs dfsadmin -refreshNodes
   ```

   where *$HDFS_USER* is the user owning the HDFS services. For example, `hdfs`.

3. Open the NameNode web UI (http://*$NameNode_FQDN*`:50070`) and navigate to the **DataNodes** page.

   Check to see whether the state has changed to **Decommission In Progress** for the DataNodes being decommissioned.

4. When all the DataNodes report their state as **Decommissioned** (on the DataNodes page, or on the Decommissioned Nodes page at http://*$NameNode_FQDN*:8088/cluster/nodes/decommissioned), all of the blocks have been replicated. You can then shut down the decommissioned nodes.

5. If your cluster utilizes a `dfs.include` file, remove the decommissioned nodes from the *$HADOOP_CONF_DIR*/`dfs.include` file on the NameNode host machine, then execute the following command:

   ```
   su $HDFS_USER
   % hdfs dfsadmin -refreshNodes
   ```

   Note

   If no `dfs.include` file is specified, all DataNodes are considered to be included in the cluster (unless excluded in the `dfs.exclude` file). The `dfs.hosts` and `dfs.hosts.exclude` properties in `hdfs-site.xml` are used to specify the `dfs.include` and `dfs.exclude` files.

6.2.2. Decommission NodeManagers

Use the following instructions to decommission NodeManagers in your cluster:

1. On the NameNode host machine, edit the *$HADOOP_CONF_DIR*/`yarn.exclude` file and add the list of NodeManager hostnames (separated by a newline character).

where *$HADOOP_CONF_DIR* is the directory for storing the Hadoop configuration files. For example, `/etc/hadoop/conf`.

2. If your cluster utilizes a `yarn.include` file, remove the decommissioned nodes from the *$HADOOP_CONF_DIR*/`yarn.include` file on the ResourceManager host machine.

   Note

   If no `yarn.include` file is specified, all NodeManagers are considered to be included in the cluster (unless excluded in the `yarn.exclude` file). The `yarn.resourcemanager.nodes.include-path` and `yarn.resourcemanager.nodes.exclude-path` properties in `yarn-site.xml` are used to specify the `yarn.include` and `yarn.exclude` files.

3. Update the ResourceManager with the new set of NodeManagers. On the ResourceManager host machine, execute the following command:

```
su $YARN_USER
% yarn rmadmin -refreshNodes
```

   where *$YARN_USER* is the user owning the YARN services. For example, `yarn`.

6.3. Decommission HBase RegionServers

Use the following instructions to decommission HBase RegionServers in your cluster:

1. Decommission RegionServers.

   The preferred method of decommissioning RegionServers is to use the `graceful_stop.sh` script (Option I). This option gradually unloads Regions from the RegionServer, allowing the node to be terminated without impacting data availability. You can also terminate the RegionServer without first unloading its Regions (Option II). This will result in a short window of data unavailability as HBase's natural data recovery operations execute.

   - **Option I: Perform graceful stop**

     You can also use the following command to gracefully decommission a loaded RegionServer.

     Execute the following command from any host machine with HBase configuration installed:

```
su - $HBASE_USER
/usr/lib/hbase/bin/graceful_stop.sh $RegionServer.Hostname
```

     where *$HBASE_USER* is the user owning the HBase Services. For example, `hbase`.

     Important

     The value of *`$RegionServer.Hostname`* argument must match the hostname that HBase uses to identify RegionServers.

To find the hostname for a particular RegionServer, go to the HBase web UI and check the list of RegionServers in the HBase master UI. Typically, HBase Master uses hostnames but occassionally it can be the FQDN of a RegionServer.

- **Option II: Use `hbase-daemon.sh`**

  Execute the following command on the RegionServer that you want to decommission:

  Important

  It is important to execute the `hbase-daemon.sh` script on the RegionServer that you want to decommission.

  ```
  su - $HBASE_USER
  /usr/lib/hbase/bin/hbase-daemon.sh stop regionserver
  ```

  where *$HBASE_USER* is the user owning the HBase Services. For example, `hbase`.

  Note that Option II causes the RegionServer to close all the regions before shutting down.

2. Enable the load balancer.

   If you used the `graceful_stop.sh` script earlier, you may need to re-enable the Region Balancer. Do so using the `balance_switch` command from the shell. Pass the command `true` to enable the balancer, `false` to disable it. The command's return value is the state of the balancer **before** running the command. If `graceful_stop.sh` disabled the balancer earlier, enable it again like this:

   ```
   su - $HBASE_USER
   ```

   ```
   hbase shell
   ```

   ```
   hbase(main):001:0> balance_switch true
   false
   0 row(s) in 0.3590 seconds
   ```

   where *$HBASE_USER* is the user owning the HBase services. For example, `hbase`

7. Manually Add Slave Nodes to a HDP Cluster

Use the following instructions to manually add slave nodes to a HDP cluster:

7.1. Prerequisites

Ensure that the new slave nodes meet the following prerequisites:

- The following operating systems are supported:
  - 64-bit Red Hat Enterprise Linux (RHEL) 5 or 6
  - 64-bit CentOS 5 or 6
  - 64-bit SUSE Linux Enterprise Server (SLES) 11, SP1
- On each of your hosts:
  - yum (RHEL)
  - zypper (SLES)
  - rpm
  - scp
  - curl
  - wget
  - unzip
  - tar
  - pdsh
- Ensure that all of the ports listed here are available.
- To install Hive metastore or to use an external database for Oozie metastore, ensure that you deploy either a MySQL or an Oracle database in your cluster. For instructions, refer to this topic.
- Your system must have the correct JDK installed on all of the nodes in the cluster. For more information, see JDK Requirements.

7.2. Add Slave Nodes

Use the following instructions to manually add a slave node:

1. On each new slave node, configure the remote repository as described here.
2. On each new slave node, install HDFS as described here.
3. On each new slave node, install compression libraries as described here.
4. On each new slave node, create the DataNode and YARN NodeManager local directories as described in section 4.3 on this page.
5. Copy the Hadoop configurations to the new slave nodes and set appropriate permissions.
   - **Option I:** Copy Hadoop config files from an existing slave node.
     a. On an existing slave node, make a copy of the current configurations:
     ```
     tar zcvf hadoop_conf.tgz /etc/hadoop/conf
     ```
     b. Copy this file to each of the new nodes:
     ```
     rm -rf /etc/hadoop/conf
     cd /
     tar zxvf $location_of_copied_conf_tar_file/hadoop_conf.tgz
     chmod -R 755 /etc/hadoop/conf
     ```
   - **Option II:** Manually set up the Hadoop configuration files as described here.
6. On each of the new slave nodes, start the DataNode:
   ```
   su -l hdfs -c "/usr/lib/hadoop/sbin/hadoop-daemon.sh --config /etc/hadoop/
   conf start datanode"
   ```
7. On each of the new slave nodes, start the NodeManager:
   ```
   su - yarn -c "export HADOOP_LIBEXEC_DIR=/usr/lib/hadoop/libexec && /usr/lib/
   hadoop-yarn/sbin/yarn-daemon.sh --config /etc/hadoop/conf start nodemanager"
   ```
8. Optional - If you use a HDFS or YARN/ResourceManager `.include` file in your cluster, add the new slave nodes to the `.include` file, then run the applicable `refreshNodes` command.
   - To add new DataNodes to the `dfs.include` file:
     a. On the NameNode host machine, edit the `/etc/hadoop/conf/dfs.include` file and add the list of the new slave node hostnames (separated by newline character).

Note

If no `dfs.include` file is specified, all DataNodes are considered to be included in the cluster (unless excluded in the `dfs.exclude` file). The `dfs.hosts` and `dfs.hosts.exlude` properties in `hdfs-site.xml` are used to specify the `dfs.include` and `dfs.exclude` files.

b. On the NameNode host machine, execute the following command:

```
su -l hdfs -c "hdfs dfsadmin -refreshNodes"
```

- To add new NodeManagers to the `yarn.include` file:

a. On the ResourceManager host machine, edit the `/etc/hadoop/conf/yarn.include` file and add the list of the slave node hostnames (separated by newline character).

Note

If no `yarn.include` file is specified, all NodeManagers are considered to be included in the cluster (unless excluded in the `yarn.exclude` file). The `yarn.resourcemanager.nodes.include-path` and `yarn.resourcemanager.nodes.exclude-path` properties in `yarn-site.xml` are used to specify the `yarn.include` and `yarn.exclude` files.

b. On the ResourceManager host machine, execute the following command:

```
su -l yarn -c "yarn rmadmin -refreshNodes"
```

7.3. Add HBase RegionServer

Use the following instructions to manually add HBase RegionServer hosts:

1. On each of the new slave nodes, install HBase and ZooKeeper.

   - For RHEL/CentOS/Oracle Linux:

   ```
   yum install zookeeper hbase
   ```

   - For SLES:

   ```
   zypper install zookeeper hbase
   ```

   - For Ubuntu:

   ```
   apt-get install zookeeper hbase
   ```

2. On each of the new slave nodes, add the HDP repository to yum:

   - For RHEL/CentOS 5:

   ```
   wget -nv http://public-repo-1.hortonworks.com/HDP/centos5/2.x/GA/2.1-
   latest/hdp.repo -O /etc/yum.repos.d/hdp.repo
   ```

   - For RHEL/CentOS 6:

   ```
   wget -nv http://public-repo-1.hortonworks.com/HDP/centos6/2.x/GA/2.1-
   latest/hdp.repo -O /etc/yum.repos.d/hdp.repo
   ```

- For SLES:

  ```
  wget -nv http://public-repo-1.hortonworks.com/HDP/suse11/2.x/GA/2.1-
  latest/hdp.repo -O /etc/zypp.repos.d/hdp.repo
  ```

- For Ubuntu:

  ```
  wget http://public-repo-1.hortonworks.com/HDP/ubuntu12/2.x/hdp.list -O /
  etc/apt/sources.list.d/hdp.list
  ```

3. Copy the HBase configurations to the new slave nodes and set appropriate permissions.

   - **Option I:** Copy HBase config files from an existing slave node.

     a. On any existing slave node, make a copy of the current configurations:

     ```
     tar zcvf hbase_conf.tgz /etc/hbase/conf
     tar zcvf zookeeper_conf.tgz /etc/zookeeper/conf
     ```

     b. Copy this file to each of the new nodes:

     ```
     rm -rf /etc/hbase/conf
     mkdir -p /etc/hbase/conf
     cd /
     tar zxvf $location_of_copied_conf_tar_file/hbase_conf.tgz
     chmod -R 755 /etc/hbase/conf
     ```

     ```
     rm -rf /etc/zookeeper/conf
     mkdir -p /etc/zookeeper/conf
     cd /
     tar zxvf $location_of_copied_conf_tar_file/zookeeper_conf.tgz
     chmod -R 755 /etc/zookeeper/conf
     ```

   - **Option II:** Manually add Hadoop configuration files as described here.

4. On all of the new slave nodes, create the config directory, copy all the config files, and set the permissions:

   ```
   rm -r $HBASE_CONF_DIR ;
   mkdir -p $HBASE_CONF_DIR ;
   ```

   ```
   copy all the config files to $HBASE_CONF_DIR
   ```

   ```
   chmod a+x $HBASE_CONF_DIR/;
   chown -R $HBASE_USER:$HADOOP_GROUP $HBASE_CONF_DIR/../  ;
   chmod -R 755 $HBASE_CONF_DIR/../
   ```

   ```
   rm -r $ZOOKEEPER_CONF_DIR ;
   mkdir -p $ZOOKEEPER_CONF_DIR ;
   ```

   ```
   copy all the config files to $ZOOKEEPER_CONF_DIR
   ```

   ```
   chmod a+x $ZOOKEEPER_CONF_DIR/;
   chown -R $ZOOKEEPER_USER:$HADOOP_GROUP $ZOOKEEPER_CONF_DIR/../  ;
   chmod -R 755 $ZOOKEEPER_CONF_DIR/../
   ```

   where:

- *$HBASE_CONF_DIR* is the directory to store the HBase configuration files. For example, `/etc/hbase/conf`.
- *$HBASE_USER* is the user owning the HBase services. For example, `hbase`.
- *$HADOOP_GROUP* is a common group shared by services. For example, `hadoop`.
- *$ZOOKEEPER_CONF_DIR* is the directory to store the ZooKeeper configuration files. For example, `/etc/zookeeper/conf`
- *$ZOOKEEPER_USER* is the user owning the ZooKeeper services. For example, `zookeeper`.

5. Start HBase RegionServer node:

```
<login as $HBASE_USER>
/usr/lib/hbase/bin/hbase-daemon.sh --config $HBASE_CONF_DIR start
 regionserver
```

6. On the HBase Master host machine, edit the `/usr/lib/hbase/conf` file and add the list of slave nodes' hostnames. The hostnames must be separated by a newline character.

8. NameNode High Availability for Hadoop

The HDFS NameNode High Availability feature enables you to run redundant NameNodes in the same cluster in an Active/Passive configuration with a hot standby. This eliminates the NameNode as a potential single point of failure (SPOF) in an HDFS cluster.

Formerly, if a cluster had a single NameNode, and that machine or process became unavailable, the entire cluster would be unavailable until the NameNode was either restarted or started on a separate machine. This situation impacted the total availability of the HDFS cluster in two major ways:

- In the case of an unplanned event such as a machine crash, the cluster would be unavailable until an operator restarted the NameNode.
- Planned maintenance events such as software or hardware upgrades on the NameNode machine would result in periods of cluster downtime.

HDFS NameNode HA avoids this by facilitating either a fast failover to the new NameNode during machine crash, or a graceful administrator-initiated failover during planned maintenance.

This guide provides an overview of the HDFS NameNode High Availability (HA) feature, instructions on how to deploy Hue with an HA cluster, and instructions on how to enable HA on top of an existing HDP cluster using the Quorum Journal Manager (QJM) and Zookeeper Failover Controller for configuration and management. Using the QJM and Zookeeper Failover Controller enables the sharing of edit logs between the Active and Standby NameNodes.

Note

This guide assumes that an existing HDP cluster has been manually installed and deployed. If your existing HDP cluster was installed using Ambari, configure NameNode HA using the Ambari wizard, as described in the Ambari documentation.

This document includes the following sections:

8.1. Architecture

In a typical HA cluster, two separate machines are configured as NameNodes. In a working cluster, one of the NameNode machine is in the **Active** state, and the other is in the **Standby** state.

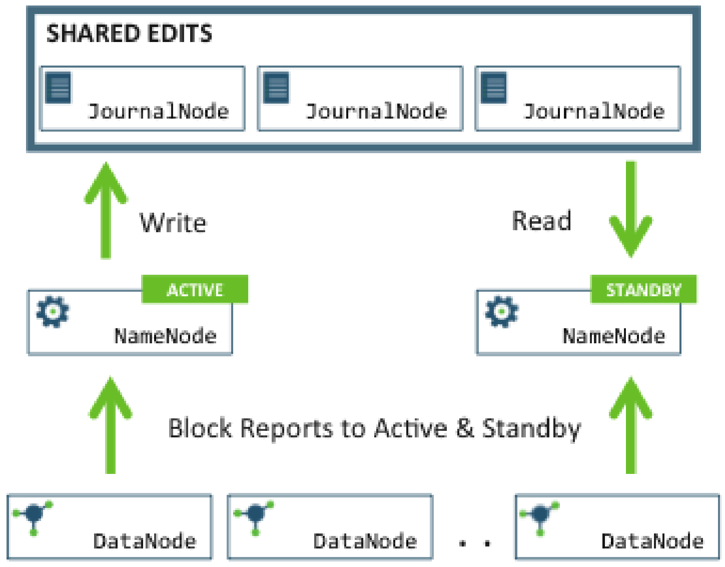
The Active NameNode is responsible for all client operations in the cluster, while the Standby acts as a slave. The Standby machine maintains enough state to provide a fast failover (if required).

In order for the Standby node to keep its state synchronized with the Active node, both nodes communicate with a group of separate daemons called **JournalNodes** (JNs). When the Active node performs any namespace modification, the Active node durably logs a modification record to a majority of these JNs. The Standby node reads the edits from the JNs and continuously watches the JNs for changes to the edit log. Once the Standby Node observes the edits, it applies these edits to its own namespace. When using QJM, JournalNodes acts the shared editlog storage. In a failover event, the Standby ensures that it has read all of the edits from the JounalNodes before promoting itself to the Active state. (This mechanism ensures that the namespace state is fully synchronized before a failover completes.)

Note

Secondary NameNode is not required in HA configuration because the Standby node also performs the tasks of the Secondary NameNode.

In order to provide a fast failover, it is also necessary that the Standby node have up-to-date information on the location of blocks in your cluster. To get accurate information about the block locations, DataNodes are configured with the location of both of the NameNodes, and send block location information and heartbeats to both NameNode machines.



It is vital for the correct operation of an HA cluster that only one of the NameNodes should be Active at a time. Failure to do so, would cause the namespace state to quickly diverge between the two NameNode machines thus causing potential data loss. (This situation is called as **split-brain scenario**.)

To prevent the **split-brain scenario**, the JournalNodes allow only one NameNode to be a writer at a time. During failover, the NameNode, that is to chosen to become active, takes over the role of writing to the JournalNodes. This process prevents the other NameNode from continuing in the Active state and thus lets the new Active node proceed with the failover safely.

8.2. Hardware Resources

Ensure that you prepare the following hardware resources:

- **NameNode machines**: The machines where you run Active and Standby NameNodes, should have exactly the same hardware. For recommended hardware for Hadoop, see Hardware recommendations for Apache Hadoop.
- **JournalNode machines**: The machines where you run the JournalNodes. The JournalNode daemon is relatively lightweight, so these daemons may reasonably be co-located on machines with other Hadoop daemons, for example the NameNodes or the YARN ResourceManager.

Note

There must be at least three JournalNode daemons, because edit log modifications must be written to a majority of JNs. This lets the system tolerate failure of a single machine. You may also run more than three JournalNodes, but in order to increase the number of failures that the system can tolerate, you must run an odd number of JNs, (i.e. 3, 5, 7, etc.).

Note that when running with `N` JournalNodes, the system can tolerate at most `(N - 1) / 2` failures and continue to function normally.

- **Zookeeper machines**: For automated failover functionality, there must be an existing Zookeeper cluster available. The Zookeeper service nodes can be co-located with other Hadoop daemons.

In an HA cluster, the Standby NameNode also performs checkpoints of the namespace state. Therefore, do not deploy a Secondary NameNode, CheckpointNode, or BackupNode in an HA cluster.

8.3. Deploy NameNode HA Cluster

HA configuration is backward compatible and works with your existing single NameNode configuration. The following instructions describe how to set up NameName HA on a manually-installed cluster. If you installed with Ambari and manage HDP on Ambari 1.4.1 or later, instead of these instructions use the Ambari documentation for the NameNode HA wizard.

Note

HA cannot accept HDFS cluster names that include an underscore (_).

To deploy a NameNode HA cluster:

1. Configure HA cluster
2. Deploy NameNode HA Cluster
3. OPTIONAL: Deploy Hue with HA Cluster

8.3.1. Configure NameNode HA Cluster

Add High Availability configurations to your HDFS configuration files. Start by taking the HDFS configuration files from the original NameNode in your HDP cluster, and use that as the base, adding the properties mentioned below to those files.

After you have added the configurations below, ensure that the same set of HDFS configuration files are propogated to all nodes in the HDP cluster. This ensures that all the nodes and services are able to interact with the highly available NameNode.

Add the following configuration options to your `hdfs-site.xml` file:

- **dfs.nameservices:**

  Choose an arbitrary but logical name (for example `mycluster`) as the value for `dfs.nameservices` option. This name will be used for both configuration and authority component of absolute HDFS paths in the cluster.

  ```
  <property>
    <name>dfs.nameservices</name>
    <value>mycluster</value>
    <description>Logical name for this new nameservice</description>
  </property>
  ```

  If you are also using HDFS Federation, this configuration setting should also include the list of other nameservices, HA or otherwise, as a comma-separated list.

- **dfs.ha.namenodes.[$nameservice ID]:**

  Provide a list of comma-separated NameNode IDs. DataNodes use this this property to determine all the NameNodes in the cluster.

  For example, for the nameservice ID `mycluster` and individual NameNode IDs `nn1` and `nn2`, the value of this property will be:

  ```
  <property>
    <name>dfs.ha.namenodes.mycluster</name>
    <value>nn1,nn2</value>
    <description>Unique identifiers for each NameNode in the nameservice</
  description>
  </property>
  ```

Note

Currently, a maximum of two NameNodes may be configured per nameservice.

- **dfs.namenode.rpc-address.[$nameservice ID].[$name node ID]:**

  Use this property to specify the fully-qualified RPC address for each NameNode to listen on.

  Continuning with the previous example, set the full address and IPC port of the NameNode process for the above two NameNode IDs - `nn1` and `nn2` .

  Note that there will be two separate configuration options.

```
<property>
  <name>dfs.namenode.rpc-address.mycluster.nn1</name>
  <value>machine1.example.com:8020</value>
</property>
<property>
  <name>dfs.namenode.rpc-address.mycluster.nn2</name>
  <value>machine2.example.com:8020</value>
</property>
```

- **dfs.namenode.http-address.[$nameservice ID].[$name node ID]:**

  Use this property to specify the fully-qualified HTTP address for each NameNode to listen on.

  Set the addresses for both NameNodes HTTP servers to listen on. For example:

```

<property>
<name>dfs.namenode.http-address.mycluster.nn1</name>
<value>machine1.example.com:50070</value>
</property>
<property>
<name>dfs.namenode.http-address.mycluster.nn2</name>
<value>machine2.example.com:50070</value>
</property>
```

Note

If you have Hadoop security features enabled, set the https-address for each NameNode.

- **dfs.namenode.shared.edits.dir:**

  Use this property to specify the URI that identifies a group of JournalNodes (JNs) where the NameNode will write/read edits.

  Configure the addresses of the JNs that provide the shared edits storage. The Active nameNode writes to this shared storage and the Standby NameNode reads from this location to stay up-to-date with all the file system changes.

Although you must specify several JournalNode addresses, **you must configure only one of these URIs** for your cluster.

The URI should be of the form:

```
qjournal://host1:port1;host2:port2;host3:port3/journalId
```

The Journal ID is a unique identifier for this nameservice, which allows a single set of JournalNodes to provide storage for multiple federated namesystems. You can reuse the nameservice ID for the journal identifier.

For example, if the JournalNodes for a cluster were running on the `node1.example.com`, `node2.example.com`, and `node3.example.com` machines and the nameservice ID were `mycluster`, you would use the following as the value for this setting:

```
<property>
  <name>dfs.namenode.shared.edits.dir</name>
  <value>qjournal://node1.example.com:8485;node2.example.com:8485;node3.
example.com:8485/mycluster</value>
</property>
```

Note

Note that the default port for the JournalNode is 8485.

- **dfs.client.failover.proxy.provider.[$nameservice ID]:**

  This property specifies the Java class that HDFS clients use to contact the Active NameNode. DFS Client uses this Java class to determine which NameNode is the current Active and therefore which NameNode is currently serving client requests.

  Use the **ConfiguredFailoverProxyProvider** implementation if you are not using a custom implementation.

  For example:

```
<property>
  <name>dfs.client.failover.proxy.provider.mycluster</name>
  <value>org.apache.hadoop.hdfs.server.namenode.ha.
ConfiguredFailoverProxyProvider</value>
</property>
```

- **dfs.ha.fencing.methods:**

  This property specifies a list of scripts or Java classes that will be used to fence the Active NameNode during a failover.

  It is important for maintaining correctness of the system that only one NameNode be in the Active state at any given time. Especially, when using the Quorum Journal Manager, only one NameNode will ever be allowed to write to the JournalNodes, so there is no potential for corrupting the file system metadata from a split-brain scenario. However, when a failover occurs, it is still possible that the previous Active NameNode could serve read requests to clients, which may be out of date until that NameNode shuts down when trying to write to the JournalNodes. For this reason, it is still recommended to configure some fencing methods even when using the Quorum Journal Manager. To improve the availability of the system in the event the fencing mechanisms fail, it is advisable to configure a fencing method which is guaranteed to return success as the last fencing method in the list. Note that if you choose to use no actual fencing methods, you must set some value for this setting, for example `shell(/bin/true)`.

The fencing methods used during a failover are configured as a carriage-return-separated list, which will be attempted in order until one indicates that fencing has succeeded. The following two methods are packaged with Hadoop: `shell` and `sshfence`. For information on implementing custom fencing method, see the `org.apache.hadoop.ha.NodeFencer` class.

- **sshfence:** SSH to the Active NameNode and kill the process.

  The *sshfence* option SSHes to the target node and uses *fuser* to kill the process listening on the service's TCP port. In order for this fencing option to work, it must be able to SSH to the target node without providing a passphrase. Ensure that you configure the **dfs.ha.fencing.ssh.private-key-files** option, which is a comma-separated list of SSH private key files.

  For example:

  ```
  <property>
    <name>dfs.ha.fencing.methods</name>
    <value>sshfence</value>
  </property>

  <property>
    <name>dfs.ha.fencing.ssh.private-key-files</name>
    <value>/home/exampleuser/.ssh/id_rsa</value>
  </property>
  ```

  Optionally, you can also configure a non-standard username or port to perform the SSH. You can also configure a timeout, in milliseconds, for the SSH, after which this fencing method will be considered to have failed. To configure non-standard username or port and timeout, see the properties given below:

  ```
  <property>
    <name>dfs.ha.fencing.methods</name>
    <value>sshfence([[username][:port]])</value>
  </property>
  <property>
    <name>dfs.ha.fencing.ssh.connect-timeout</name>
    <value>30000</value>
  </property>
  ```

- **shell:** Run an arbitrary shell command to fence the Active NameNode.

  The *shell* fencing method runs an arbitrary shell command:

  ```
  <property>
    <name>dfs.ha.fencing.methods</name>
    <value>shell(/path/to/my/script.sh arg1 arg2 ...)</value>
  </property>
  ```

The string between '(' and ')' is passed directly to a bash shell and may not include any closing parentheses.

The shell command will be run with an environment set up to contain all of the current Hadoop configuration variables, with the '_' character replacing any '.' characters in the configuration keys. The configuration used has already had any namenode-specific configurations promoted to their generic forms – for example **dfs_namenode_rpc-address** will contain the RPC address of the target node, even though the configuration may specify that variable as **dfs.namenode.rpc-address.ns1.nn1**.

Additionally, the following variables (referring to the target node to be fenced) are also available:

- *`$target_host`*: Hostname of the node to be fenced
- *`$target_port`*: IPC port of the node to be fenced
- *`$target_address`*: The combination of *`$target_host`* and *`$target_port`* as host:port
- *`$target_nameserviceid`*: The nameservice ID of the NN to be fenced
- *`$target_namenodeid`*: The namenode ID of the NN to be fenced

These environment variables may also be used as substitutions in the shell command. For example:

```
<property>
  <name>dfs.ha.fencing.methods</name>
  <value>shell(/path/to/my/script.sh --nameservice=$target_nameserviceid
 $target_host:$target_port)</value>
</property>
```

If the shell command returns an exit code of 0, the fencing is successful.

Note

This fencing method does not implement any timeout. If timeouts are necessary, they should be implemented in the shell script itself (for example, by forking a subshell to kill its parent in some number of seconds).

- **fs.defaultFS:**

  The default path prefix used by the Hadoop FS client. Optionally, you may now configure the default path for Hadoop clients to use the new HA-enabled logical URI. For example, for `mycluster` nameservice ID, this will be the value of the authority portion of all of your HDFS paths.

  Configure this property in the **core-site.xml** file:

```
<property>
  <name>fs.defaultFS</name>
  <value>hdfs://mycluster</value>
</property>
```

- **dfs.journalnode.edits.dir:**

  This is the absolute path on the JournalNode machines where the edits and other local state (used by the JNs) will be stored. You may only use a single path for this configuration. Redundancy for this data is provided by either running multiple separate JournalNodes or by configuring this directory on a locally-attached RAID array. For example:

  ```
  <property>
    <name>dfs.journalnode.edits.dir</name>
    <value>/path/to/journal/node/local/data</value>
  </property>
  ```

Note

See Creating Service Principals and Keytab files for HDP for instructions on configuring Kerberos-based security for Highly Available clusters.

8.3.2. Deploy NameNode HA Cluster

In this section, we use NN1 to denote the original NameNode in the non-HA setup, and NN2 to denote the other NameNode that is to be added in the HA setup.

Note

HA clusters reuse the `nameservice ID` to identify a single HDFS instance (that may consist of multiple HA NameNodes).

A new abstraction called `NameNode ID` is added with HA. Each NameNode in the cluster has a distinct `NameNode ID` to distinguish it.

To support a single configuration file for all of the NameNodes, the relevant configuration parameters are suffixed with both the **nameservice ID** and the **NameNode ID**.

1. **Start the JournalNode daemons** on those set of machines where the JNs are deployed. On each machine, execute the following command:

   ```
   su –l hdfs –c "/usr/lib/hadoop/sbin/hadoop-daemon.sh start journalnode"
   ```

2. Wait for the daemon to start on each of the JN machines.

3. **Initialize JournalNodes.**

   - At the **NN1** host machine, execute the following command:

     ```
     su –l hdfs –c "namenode -initializeSharedEdits -force"
     ```

This command formats all the JournalNodes. This by default happens in an interactive way: the command prompts users for “Y/N” input to confirm the format. You can skip the prompt by using option **-force** or **-nonInteractive**.

It also copies all the edits data after the most recent checkpoint from the edits directories of the local NameNode (**NN1**) to JournalNodes.

- At the host with the journal node (if it is separated from the primary host), execute the following command:

  ```
  su –l hdfs –c "namenode -initializeSharedEdits -force"
  ```

- Initialize HA state in ZooKeeper.

  Execute the following command on NN1:

  ```
  hdfs zkfc -formatZK -force
  ```

  This command creates a `znode` in ZooKeeper. The failover system stores uses this znode for data storage.

- Check to see if Zookeeper is running. If not, start Zookeeper by executing the following command on the ZooKeeper host machine(s).

  ```
  su - zookeeper -c "export  ZOOCFGDIR=/etc/zookeeper/conf ; export ZOOCFG=
  zoo.cfg ; source /etc/zookeeper/conf/zookeeper-env.sh ; /usr/lib/
  zookeeper/bin/zkServer.sh start"
  ```

- At the standby namenode host, execute the following command:

  ```
  su -l hdfs -c "namenode -bootstrapStandby -force"
  ```

4. Start NN1. At the NN1 host machine, execute the following command:

   ```
   su -l hdfs -c "/usr/lib/hadoop/sbin/hadoop-daemon.sh --config /etc/hadoop/
   conf start namenode"
   ```

   Ensure that NN1 is running correctly.

5. Format NN2 and copy the latest checkpoint (FSImage) from NN1 to NN2 by executing the following command:

   ```
   su -l hdfs -c "namenode -bootstrapStandby -force"
   ```

   This command connects with HH1 to get the namespace metadata and the checkpointed fsimage. This command also ensures that NN2 receives sufficient editlogs from the JournalNodes (corresponding to the fsimage). This command fails if JournalNodes are not correctly initialized and cannot provide the required editlogs.

6. Start NN2. Execute the following command on the NN2 host machine:

   ```
   su -l hdfs -c "/usr/lib/hadoop/sbin/hadoop-daemon.sh --config /etc/hadoop/
   conf start namenode"
   ```

   Ensure that NN2 is running correctly.

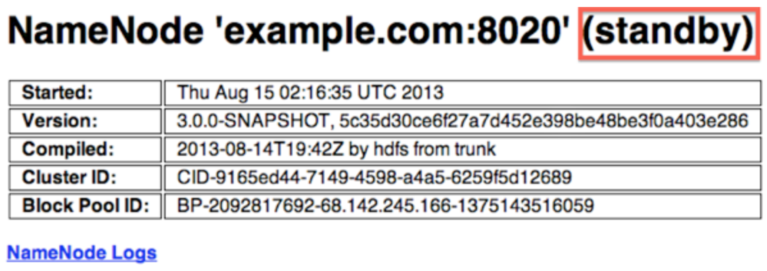
7. **Start DataNodes.** Execute the following command on all the DataNodes:

```
su -l hdfs -c "/usr/lib/hadoop/sbin/hadoop-daemon.sh --config /etc/hadoop/
conf start datanode"
```

8. Validate the HA configuration.

   Go to the NameNodes' web pages separately by browsing to their configured HTTP addresses.

   Under the configured address label, you should see that HA state of the NameNode. The NameNode can be either in "standby" or "active" state.

   NameNode 'example.com:8020' (standby)

Started:	Thu Aug 15 02:16:35 UTC 2013
Version:	3.0.0-SNAPSHOT, 5c35d30ce6f27a7d452e398be48be3f0a403e286
Compiled:	2013-08-14T19:42Z by hdfs from trunk
Cluster ID:	CID-9165ed44-7149-4598-a4a5-6259f5d12689
Block Pool ID:	BP-2092817692-68.142.245.166-1375143516059

   NameNode Logs

   ### Note

   The HA NameNode is initially in the Standby state after it is bootstrapped.

   You can also use either JMX (`tag.HAState`) to query the HA state of a NameNode.

   The following command can also be used query HA state for NameNode:

   ```
   hdfs haadmin -getServiceState
   ```

9. Transition one of the HA NameNode to Active state.

   Initially, both NN1 and NN2 are in Standby state. Therefore you must transition one of the NameNode to Active state. This transition can be performed using one of the following options:

   - **Option I - Using CLI**

     Use the command line interface (CLI) to transition one of the NameNode to Active State. Execute the following command on that NameNode host machine:

     ```
     hdfs haadmin -failover --forcefence --forceactive <serviceId> <namenodeId>
     ```

     For more information on the `haadmin` command, see Appendix - Administrative Commands section in this document.

   - **Option II - Deploying Automatic Failover**

     You can configure and deploy automatic failover using the instructions provided here.

8.3.3. Deploy Hue with an HA Cluster

If you are going to use Hue with an HA Cluster, make the following changes to `/etc/hue/conf/hue.ini`:

1. Install the Hadoop HttpFS component on the Hue server.

   For RHEL/CentOS/Oracle Linux:

   ```
    yum install hadoop-httpfs
   ```

   For SLES:

   ```
   zypper install hadoop-httpfs
   ```

2. Modify `/etc/hadoop-httpfs/conf/httpfs-site.xml` to configure HttpFS to talk to the cluster by confirming the following properties are correct:

   ```
   <property>
     <name>httpfs.proxyuser.hue.hosts</name>
     <value>*</value>
   </property>
   <property>
     <name>httpfs.proxyuser.hue.groups</name>
     <value>*</value>
   </property>
   ```

3. Start the HttpFS service.

   ```
   service hadoop-httpfs start
   ```

4. Modify the core-site.xml file. On the NameNodes and all the DataNodes, add the following properties to the $HADOOP_CONF_DIR /core-site.xml file. Where $HADOOP_CONF_DIR is the directory for storing the Hadoop configuration files. For example, /etc/hadoop/conf.

   ```
   <property>
     <name>hadoop.proxyuser.httpfs.groups</name>
     <value>*</value>
   </property>
   <property>
     <name>hadoop.proxyuser.httpfs.hosts</name>
     <value>*</value>
   </property>
   ```

5. In the `hue.ini` file, under the `[hadoop][[hdfs_clusters]][[[default]]]` sub-section, use the following variables to configure the cluster:

   Table 8.1. Hue Configuration Properties

Property	Description	Example
fs_defaultfs	NameNode URL using the logical name for the new name service. For reference, this is the dfs.nameservices property in hdfs-site.xml in your Hadoop configuration.	hdfs://mycluster

Property	Description	Example
webhdfs_url	URL to the HttpFS server.	http://c6401.apache.org:14000/webhdfs/v1/

6. Restart Hue for the changes to take effect.

```
service hue restart
```

8.3.4. Deploy Oozie with HA Cluster

You can configure multiple Oozie servers against the same database to provide High Availability (HA) of the Oozie service. You need the following prerequisites:

- A database that supports multiple concurrent connections. In order to have full HA, the database should also have HA support, or it becomes a single point of failure.

  Note

  The default derby database does not support this.

- A ZooKeeper ensemble. Apache ZooKeeper is a distributed, open-source coordination service for distributed applications; the Oozie servers use it for coordinating access to the database and communicating with each other. In order to have full HA, there should be at least 3 ZooKeeper servers. Find more information about Zookeeper here.

- Multiple Oozie servers.

  Important

  While not strictly required, you should configure all ZooKeeper servers to have identical properties.

- A Loadbalancer, Virtual IP, or Round-Robin DNS. This is used to provide a single entry-point for users and for callbacks from the JobTracker. The load balancer should be configured for round-robin between the Oozie servers to distribute the requests. Users (using either the Oozie client, a web browser, or the REST API) should connect through the load balancer. In order to have full HA, the load balancer should also have HA support, or it becomes a single point of failure.

  For information about how to set up your Oozie servers to handle failover, see Configuring Oozie Failover.

8.4. Operating a NameNode HA cluster

- While operating an HA cluster, the Active NameNode cannot commit a transaction if it cannot write successfully to a quorum of the JournalNodes.
- When restarting an HA cluster, the steps for initializing JournalNodes and NN2 can be skipped.
- Start the services in the following order:

1. JournalNodes
2. NameNodes

   Note

   Verify that the ZKFailoverController (ZKFC) process on each node is running so that one of the NameNodes can be converted to active state.

3. DataNodes

8.5. Configure and Deploy NameNode Automatic Failover

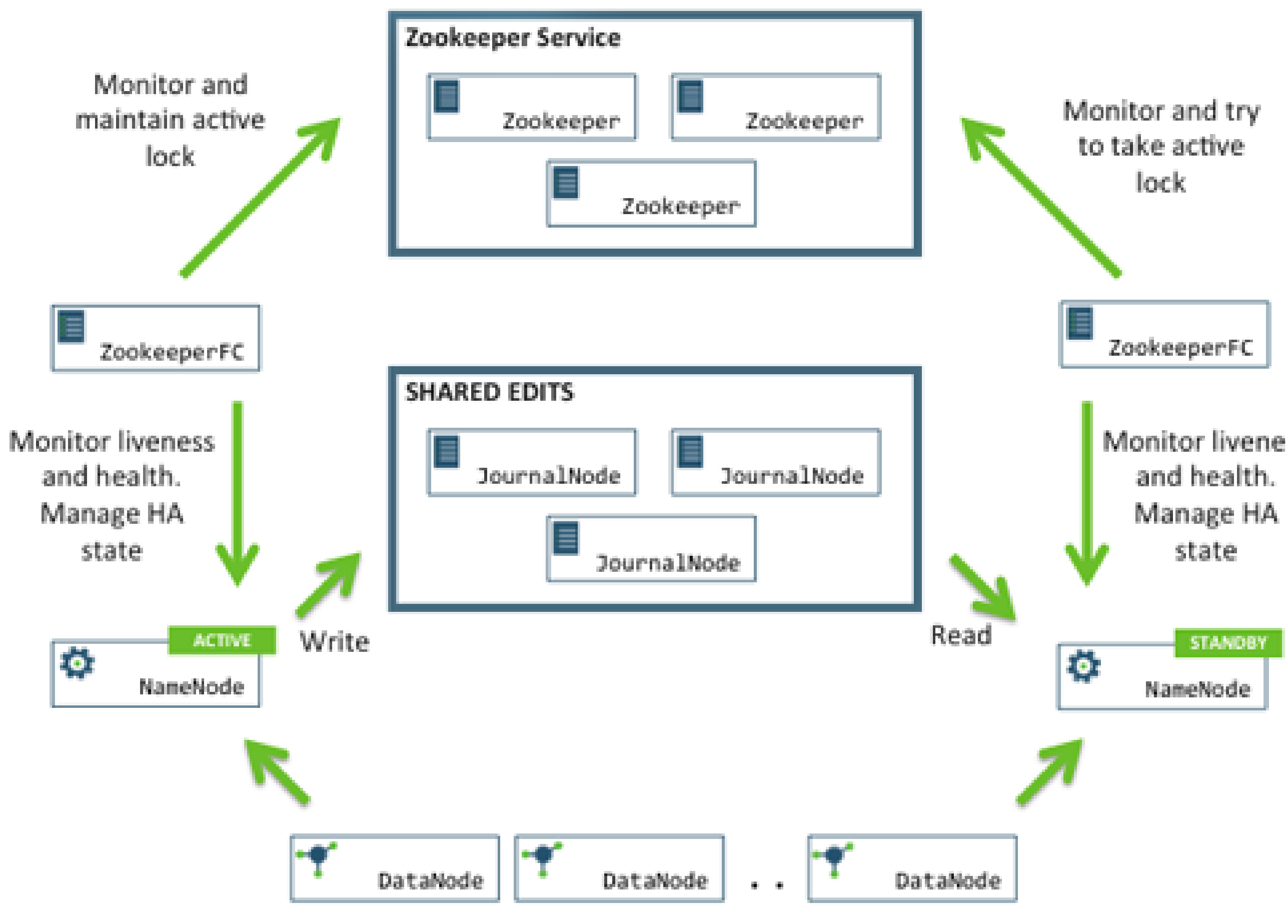
The preceding sections describe how to configure manual failover. In that mode, the system will not automatically trigger a failover from the active to the standby NameNode, even if the active node has failed. This section describes how to configure and deploy automatic failover.

Automatic failover adds following components to an HDFS deployment:

- ZooKeeper quorum
- ZKFailoverController process (abbreviated as ZKFC).

The ZKFailoverController (ZKFC) is a ZooKeeper client that monitors and manages the state of the NameNode. Each of the machines which run NameNode service also runs a ZKFC. ZKFC is responsible for:

- **Health monitoring:** ZKFC periodically pings its local NameNode with a health-check command.
- **ZooKeeper session management:** When the local NameNode is healthy, the ZKFC holds a session open in ZooKeeper. If the local NameNode is active, it also holds a special "lock" znode. This lock uses ZooKeeper's support for "ephemeral" nodes; if the session expires, the lock node will be automatically deleted.
- **ZooKeeper-based election:** If the local NameNode is healthy and no other node currently holds the lock znode, ZKFC will try to acquire the lock. If ZKFC succeeds, then it has "won the election" and will be responsible for running a failover to make its local NameNode active. The failover process is similar to the manual failover described above: first, the previous active is fenced if necessary and then the local NameNode transitions to active state.



Use the following instructions to configure and deploy automatic failover:

- Prerequisites
- Instructions

8.5.1. Prerequisites

Complete the following prerequisites:

- Ensure you have a working Zookeeper service. If you had an Ambari deployed HDP cluser with Zookeeper, you can use that. If not, deploy ZooKeeper using the instructions provided here.

Note

In a typical deployment, ZooKeeper daemons are configured to run on three or five nodes. It is however acceptable to co-locate the ZooKeeper nodes on the same hardware as the HDFS NameNode and Standby Node. Many operators choose to deploy the third ZooKeeper process on the same node as the YARN ResourceManager. To achieve performance and improve isolation, Hortonworks recommends configuring the ZooKeeper nodes such that the ZooKeeper data and HDFS metadata is stored on separate disk drives.

- Shut down your HA cluster (configured for manual failover) using the instructions provided here.

  Currently, you cannot transition from a manual failover setup to an automatic failover setup while the cluster is running.

8.5.2. Instructions

Complete the following instructions:

1. Configure automatic failover.

   a. Set up your cluster for automatic failover.

      Add the following property to the the `hdfs-site.xml` file for both the NameNode machines:

      ```
      <property>
         <name>dfs.ha.automatic-failover.enabled</name>
         <value>true</value>
       </property>
      ```

   b. List the host-port pairs running the ZooKeeper service.

      Add the following property to the the `core-site.xml` file for both the NameNode machines:

      ```
      <property>
         <name>ha.zookeeper.quorum</name>
         <value>zk1.example.com:2181,zk2.example.com:2181,zk3.example.
      com:2181</value>
       </property>
      ```

      Note

      Suffix the configuration key with the nameservice ID to configure the above settings on a per-nameservice basis. For example, in a cluster with federation enabled, you can explicitly enable automatic failover for only one of the nameservices by setting `dfs.ha.automatic-failover.enabled.`*`$my-nameservice-id`*.

2. Initialize HA state in ZooKeeper.

   Execute the following command on NN1:

   ```
   hdfs zkfc -formatZK -force
   ```

   This command creates a `znode` in ZooKeeper. The automatic failover system stores uses this znode for data storage.

3. Check to see if Zookeeper is running. If not, start Zookeeper by executing the following command on the ZooKeeper host machine(s).

```
su - zookeeper -c "export  ZOOCFGDIR=/etc/zookeeper/conf ; export ZOOCFG=
zoo.cfg ; source /etc/zookeeper/conf/zookeeper-env.sh ; /usr/lib/zookeeper/
bin/zkServer.sh start"
```

4. Start the JournalNodes, NameNodes, and DataNodes using the instructions provided here.

5. Start ZKFC.

   Manually start the `zkfc` daemon on each of the NameNode host machines using the following command:

   ```
   /usr/lib/hadoop/sbin/hadoop-daemon.sh start zkfc
   ```

   The sequence of starting ZKFC determines which NameNode will become Active. For example, if ZKFC is started on NN1 first, it will cause NN1 to become Active.

   Note

   To convert a non-HA cluster to an HA cluster, Hortonworks recommends that you run the `bootstrapStandby` command (this command is used to initialize NN2) **before** you start ZKFC on any of the NameNode machines.

6. Verify automatic failover.

   a. Locate the Active NameNode.

      Use the NameNode web UI to check the status for each NameNode host machine.

   b. Cause a failure on the Active NameNode host machine.

      For example, you can use the following command to simulate a JVM crash:

      ```
      kill -9 $PID_of_Active_NameNode
      ```

      Or, you could power cycle the machine or unplug its network interface to simulate outage.

   c. The Standby NameNode should now automatically become Active within several seconds.

      Note

      The amount of time required to detect a failure and trigger a failover depends on the configuration of `ha.zookeeper.session-timeout.ms` property (default value is 5 seconds).

   d. If the test fails, your HA settings might be incorrectly configured.

      Check the logs for the zkfc daemons and the NameNode daemons to diagnose the issue.

8.5.3. Configuring Oozie Failover

1. Set up your database for High Availability (see the database documentation for details).

   Oozie database configuration properties may need special configuration (see the JDBC driver documentation for details).

2. Configure Oozie identically on two or more servers.

3. Set the `OOZIE_HTTP_HOSTNAME` variable in `oozie-env.sh` to the Load Balancer or Virtual IP address.

4. Start all Oozie servers.

5. Use either a Virtual IP Address or Load Balancer to direct traffic to Oozie servers.

6. Access Oozie via the Virtual IP or Load Balancer address.

8.6. Appendix: Administrative Commands

The subcommands of `hdfs haadmin` are extensively used for administering an HA cluster.

Running the `hdfs haadmin` command without any additional arguments will display the following usage information:

```
Usage: DFSHAAdmin [-ns <nameserviceId>]
    [-transitionToActive <serviceId>]
    [-transitionToStandby <serviceId>]
    [-failover [--forcefence] [--forceactive] <serviceId> <serviceId>]
    [-getServiceState <serviceId>]
    [-checkHealth <serviceId>]
    [-help <command>]
```

This section provides high-level uses of each of these subcommands.

- **transitionToActive** and **transitionToStandby**: Transition the state of the given NameNode to Active or Standby.

  These subcommands cause a given NameNode to transition to the Active or Standby state, respectively. These commands do not attempt to perform any fencing, and thus should be used **rarely**. Instead, Hortonworks recommends using the following subcommand:

  ```
  hdfs haadmin -failover
  ```

- **failover**: Initiate a failover between two NameNodes.

  This subcommand causes a failover from the first provided NameNode to the second.

  - If the first NameNode is in the Standby state, this command transitions the second to the Active state without error.

  - If the first NameNode is in the Active state, an attempt will be made to gracefully transition it to the Standby state. If this fails, the fencing methods (as configured by

dfs.ha.fencing.methods) will be attempted in order until one succeeds. Only after this process will the second NameNode be transitioned to the Active state. If the fencing methods fail, the second NameNode is not transitioned to Active state and an error is returned.

- **getServiceState**: Determine whether the given NameNode is Active or Standby.

  This subcommand connects to the provided NameNode, determines its current state, and prints either "standby" or "active" to STDOUT appropriately. This subcommand might be used by cron jobs or monitoring scripts.

- **checkHealth**: Check the health of the given NameNode.

  This subcommand connects to the NameNode to check its health. The NameNode is capable of performing some diagnostics that include checking if internal services are running as expected. This command will return 0 if the NameNode is healthy else it will return a non-zero code.

  Note

  This subcommand is in implementation phase and currently always returns success unless the given NameNode is down.

9. ResourceManager High Availability for Hadoop

This guide provides instructions on setting up the ResourceManager (RM) High Availability (HA) feature in a HDFS cluster. The Active and Standby ResourceManagers embed the Zookeeper-based ActiveStandbyElector to determine which ResourceManager should be active.

Note

ResourceManager HA is currently only supported for non-Ambari clusters. This guide assumes that an existing HDP cluster has been manually installed and deployed. It provides instructions on how to manually enable ResourceManager HA on top of the existing cluster.

The ResourceManager was a single point of failure (SPOF) in an HDFS cluster. Each cluster had a single ResourceManager, and if that machine or process became unavailable, the entire cluster would be unavailable until the ResourceManager was either restarted or started on a separate machine. This situation impacted the total availability of the HDFS cluster in two major ways:

- In the case of an unplanned event such as a machine crash, the cluster would be unavailable until an operator restarted the ResourceManager.
- Planned maintenance events such as software or hardware upgrades on the ResourceManager machine would result in windows of cluster downtime.

The ResourceManager HA feature addresses these problems. This feature enables you to run redundant ResourceManagers in the same cluster in an Active/Passive configuration with a hot standby. This mechanism thus facilitates either a fast failover to the standby ResourceManager during machine crash, or a graceful administrator-initiated failover during planned maintenance.

In this document:

9.1. Hardware Resources

Ensure that you prepare the following hardware resources:

- **ResourceManager machines**: The machines where you run Active and Standby ResourceManagers should have exactly the same hardware. For recommended hardware for Hadoop, see Hardware recommendations for Apache Hadoop.
- **Zookeeper machines**: For automated failover functionality, there must be an existing Zookeeper cluster available. The Zookeeper service nodes can be co-located with other Hadoop daemons.

9.2. Deploy ResourceManager HA Cluster

HA configuration is backward-compatible and works with your existing single ResourceManager configuration.

Note

These instructions describe how to manually configure and deploy ResourceManager HA on a cluster. Currently, Ambari cannot be used to set up ResourceManager HA.

First, configure manual or automatic ResourceManager failover. Then, deploy the ResourceManager HA cluster.

- Configure Manual or Automatic ResourceManager Failover
- Deploy the ResourceManager HA Cluster
- Minimum Settings for ResourceManager Configuration

9.2.1. Configure Manual or Automatic ResourceManager Failover

- Prerequisites
- Set Common ResourceManager HA Properties
- Configure Manual ResourceManager Failover
- Configure Automatic ResourceManager Failover

9.2.1.1. Prerequisites

Complete the following prerequisites:

- Ensure that you have a working Zookeeper service. If you had an Ambari deployed HDP cluster with Zookeeper, you can use that Zookeeper service. If not, deploy ZooKeeper using the instructions provided here.

  Note

  In a typical deployment, ZooKeeper daemons are configured to run on three or five nodes. It is however acceptable to collocate the ZooKeeper nodes on the same hardware as the HDFS NameNode and Standby Node. Many operators choose to deploy the third ZooKeeper process on the same node as the YARN ResourceManager. To achieve performance and improve isolation, Hortonworks recommends configuring the ZooKeeper nodes such that the ZooKeeper data and HDFS metadata is stored on separate disk drives.

- Shut down the cluster using the instructions provided here.

9.2.1.2. Set Common ResourceManager HA Properties

The following properties are required for both manual and automatic ResourceManager HA. Add these properties to the `etc/hadoop/conf/yarn-site.xml` file:

Property Name	Recommended Value
yarn.resourcemanager.ha.enabled	true
yarn.resourcemanager.ha.rm-ids	Cluster-specific, e.g., rm1,rm2
yarn.resourcemanager.hostname.<rm-id>	Cluster-specific
yarn.resourcemanager.recovery.enabled	true
yarn.resourcemanager.store.class	org.apache.hadoop.yarn.server. resourcemanager.recovery.ZKRM
yarn.resourcemanager.zk-address	Cluster- specific
yarn.client.failover-proxy-provider	org.apache.hadoop.yarn.client. ConfiguredRMFailoverProxyProvid

Note

You can also set values for each of the following properties in `yarn-site.xml`:

- `yarn.resourcemanager.address.<rm#id>`
- `yarn.resourcemanager.scheduler.address.<rm#id>`
- `yarn.resourcemanager.admin.address.<rm#id>`
- `yarn.resourcemanager.resource#tracker.address.<rm#id>`
- `yarn.resourcemanager.webapp.address.<rm#id>`

If these addresses are not explicitly set, each of these properties will use `yarn.resourcemana` `id>:default_port`, such as DEFAULT_RM_PORT, DEFAULT_RM_SCHEDULER_PORT, etc.

The following is a sample `yarn-site.xml` file with these common ResourceManager HA properties configured:

```
<!-- RM HA Configurations-->

 <property>
    <name>yarn.resourcemanager.ha.enabled</name>
    <value>true</value>
 </property>

 <property>
    <name>yarn.resourcemanager.ha.rm-ids</name>
    <value>rm1,rm2</value>
 </property>

 <property>
    <name>yarn.resourcemanager.hostname.rm1</name>
    <value>${rm1 address}</value>
 </property>
 <property>
    <name>yarn.resourcemanager.hostname.rm2</name>
    <value>${rm2 address}</value>
 </property>

 <property>
    <name>yarn.resourcemanager.webapp.address.rm1</name>
    <value>rm1_web_address:port_num</value>
    <description>We can set rm1_web_address separately. If not, it will use
    ${yarn.resourcemanager.hostname.rm1}:DEFAULT_RM_WEBAPP_PORT</description>
 </property>

 <property>
    <name>yarn.resourcemanager.webapp.address.rm2</name>
    <value>rm2_web_address:port_num</value>
 </property>

 <property>
    <name>yarn.resourcemanager.recovery.enabled</name>
    <value>true</value>
 </property>

 <property>
    <name>yarn.resourcemanager.store.class</name>
    <value>org.apache.hadoop.yarn.server.resourcemanager.recovery.
ZKRMStateStore</value>
 </property>

 <property>
    <name>yarn.resourcemanager.zk-address</name>
 <value>${zk1.address,zk2.address}</value>
 </property>

 <property>
   <name>yarn.client.failover-proxy-provider</name>
   <value>org.apache.hadoop.yarn.client.ConfiguredRMFailoverProxyProvider</
value>
 </property>
```

9.2.1.3. Configure Manual ResourceManager Failover

Automatic ResourceManager failover is enabled by default, so it must be disabled for manual failover.

To configure manual failover for ResourceManager HA, add the `yarn.resourcemanager.ha.automatic-failover.enabled` configuration property to the `etc/hadoop/conf/yarn-site.xml` file, and set its value to "false":

```
<property>
        <name>yarn.resourcemanager.ha.automatic-failover.enabled</name>
        <value>false</value>
 </property>
```

9.2.1.4. Configure Automatic ResourceManager Failover

The preceding section described how to configure manual failover. In that mode, the system will not automatically trigger a failover from the active to the standby

ResourceManager, even if the active node has failed. This section describes how to configure automatic failover.

1. Add the following configuration options to the `yarn-site.xml` file:

Property Name	Recommended Value	Description
yarn.resourcemanager.ha.automatic-failover.zk-base-path	/yarn-leader-election	The base znode pat when using ZooKee configuration. The c
yarn.resourcemanager.cluster-id	yarn-cluster	The name of the clu participates in leade not affect other clus

   Example:

```
<property>
  <name>yarn.resourcemanager.ha.automatic-failover.zk-base-path</name>
  <value>/yarn-leader-election</value>
<description>Optional setting. The default value is /yarn-leader-election</
description>
</property>

<property>
   <name>yarn.resourcemanager.cluster-id</name>
   <value>yarn-cluster</value>
</property>
```

2. Automatic ResourceManager failover is enabled by default.

   If you previously configured manual ResourceManager failover by setting the value of `yarn.resourcemanager.ha.automatic-failover.enabled` to "false", you must delete this property to return automatic failover to its default enabled state.

9.2.2. Deploy the ResourceManager HA Cluster

1. Copy the `etc/hadoop/conf/yarn-site.xml` file from the primary ResourceManager host to the standby ResourceManager host.

2. Make sure that the `clientPort` value set in `etc/zookeeper/conf/zoo.cfg` matches the port set in the following `yarn-site.xml` property:

```
<property>
        <name>yarn.resourcemanager.zk-state-store.address</name>
        <value>localhost:2181</value>
    </property>
```

3. Start Zookeeper. Execute this command on the ZooKeeper host machine(s).

```
su - zookeeper -c "export  ZOOCFGDIR=/etc/zookeeper/conf ; export ZOOCFG=
zoo.cfg ; source /etc/zookeeper/conf/zookeeper-env.sh ; /usr/lib/zookeeper/
bin/zkServer.sh start"
```

4. Start HDFS

   a. Execute this command on the NameNode host machine:

```
su -l hdfs -c "/usr/lib/hadoop/sbin/hadoop-daemon.sh --config /etc/
hadoop/conf start namenode"
```

   b. Execute this command on the Secondary NameNode host machine:

   ```
   su -l hdfs -c "/usr/lib/hadoop/sbin/hadoop-daemon.sh --config /etc/
   hadoop/conf start secondarynamenode”
   ```

   c. Execute this command on all DataNodes:

   ```
   su -l hdfs -c "/usr/lib/hadoop/sbin/hadoop-daemon.sh --config /etc/
   hadoop/conf start datanode"
   ```

5. Start YARN

   a. Execute this command to start the primary ResourceManager:

   ```
   su - yarn -c "export HADOOP_LIBEXEC_DIR=/usr/lib/hadoop/libexec && /
   usr/lib/hadoop-yarn/sbin/yarn-daemon.sh --config /etc/hadoop/conf start
    resourcemanager"
   ```

   b. Execute this command to start the standby ResourceManager:

   ```
   su - yarn -c "export HADOOP_LIBEXEC_DIR=/usr/lib/hadoop/libexec && /usr/
   lib/hadoop-yarn/sbin/yarn-daemon.sh --config /etc/hadoop/conf2 start
    resourcemanager"
   ```

   c. Execute this command on the History Server host machine:

   ```
   su - mapred -c "export HADOOP_LIBEXEC_DIR=/usr/lib/hadoop/libexec &&
    /usr/lib/hadoop-mapreduce/sbin/mr-jobhistory-daemon.sh --config /etc/
   hadoop/conf start historyserver"
   ```

   d. Execute this command on all NodeManagers:

   ```
   su - yarn -c "export HADOOP_LIBEXEC_DIR=/usr/lib/hadoop/libexec && /
   usr/lib/hadoop-yarn/sbin/yarn-daemon.sh --config /etc/hadoop/conf start
    nodemanager"
   ```

   At this point, you should be able to see messages in the console that the NodeManager is trying to connect to the active ResourceManager.

6. Set the active ResourceManager:

   MANUAL FAILOVER ONLY: If you configured manual ResourceManager failover, you must transition one of the ResourceManagers to Active mode. Execute the following CLI command to transition ResourceManager "rm1" to Active:

   ```
   yarn rmadmin -transitionToActive rm1
   ```

   You can use the following CLI command to transition ResourceManager "rm1" to Standby mode:

   ```
   yarn rmadmin -transitionToStandby rm1
   ```

   AUTOMATIC FAILOVER: If you configured automatic ResourceManager failover, no action is required – the Active ResourceManager will be chosen automatically.

7. Start all remaining unstarted cluster services using the instructions provided here.

9.2.3. Minimum Settings for Automatic ResourceManager HA Configuration

The minimum `yarn-site.xml` configuration settings for ResourceManager HA with automatic failover are as follows:

```
   <property>
        <name>yarn.resourcemanager.ha.enabled</name>
        <value>true</value>
    </property>

    <property>
        <name>yarn.resourcemanager.ha.rm-ids</name>
        <value>rm1,rm2</value>
    </property>

    <property>
        <name>yarn.resourcemanager.hostname.rm1</name>
        <value>192.168.1.9</value>
    </property>

    <property>
        <name>yarn.resourcemanager.hostname.rm2</name>
        <value>192.168.1.10</value>
    </property>

    <property>
        <name>yarn.resourcemanager.recovery.enabled</name>
        <value>true</value>
    </property>

    <property>
        <name>yarn.resourcemanager.store.class</name>
        <value>org.apache.hadoop.yarn.server.resourcemanager.recovery.
ZKRMStateStore</value>
    </property>

    <property>
        <name>yarn.resourcemanager.zk-address</name>
        <value>192.168.1.9:2181,192.168.1.10:2181</value>
        <description>For multiple zk services, separate them with comma</
description>
    </property>

    <property>
          <name>yarn.resourcemanager.cluster-id</name>
          <value>yarn-cluster</value>
        </property>
```

Testing ResourceManager HA on a Single Node

If you would like to test ResourceManager HA on a single node (launch more than one ResourceManager on a single node), you need to add the following settings in `yarn-site.xml`:

To enable ResourceManager "rm1" to launch:

```
<property>
```

```
        <name>yarn.resourcemanager.ha.id</name>
        <value>rm1</value>
        <description>If we want to launch more than one RM in single node, we
 need this configuration</description>
    </property>
```

To enable ResourceManager "rm2" to launch:

```
<property>
        <name>yarn.resourcemanager.ha.id</name>
        <value>rm2</value>
        <description>If we want to launch more than one RM in single node, we
 need this configuration</description>
    </property>
```

You should also explicitly set values specific to each ResourceManager for the following properties:

- `yarn.resourcemanager.address.<rm#id>`
- `yarn.resourcemanager.scheduler.address.<rm#id>`
- `yarn.resourcemanager.admin.address.<rm#id>`
- `yarn.resourcemanager.resource#tracker.address.<rm#id>`
- `yarn.resourcemanager.webapp.address.<rm#id> separately in yarn# site.xml`

For example:

```
    <!-- RM1 Configs -->
    <property>
        <name>yarn.resourcemanager.address.rm1</name>
        <value>localhost:23140</value>
    </property>
    <property>
        <name>yarn.resourcemanager.scheduler.address.rm1</name>
        <value>localhost:23130</value>
    </property>
    <property>
        <name>yarn.resourcemanager.webapp.address.rm1</name>
        <value>localhost:23188</value>
    </property>
    <property>
        <name>yarn.resourcemanager.resource-tracker.address.rm1</name>
        <value>localhost:23125</value>
    </property>
    <property>
        <name>yarn.resourcemanager.admin.address.rm1</name>
        <value>localhost:23141</value>
    </property>


    <!-- RM2 configs -->
    <property>
        <name>yarn.resourcemanager.address.rm2</name>
        <value>localhost:33140</value>
    </property>
    <property>
        <name>yarn.resourcemanager.scheduler.address.rm2</name>
        <value>localhost:33130</value>
    </property>
    <property>
        <name>yarn.resourcemanager.webapp.address.rm2</name>
        <value>localhost:33188</value>
    </property>
    <property>
        <name>yarn.resourcemanager.resource-tracker.address.rm2</name>
        <value>localhost:33125</value>
    </property>
    <property>
        <name>yarn.resourcemanager.admin.address.rm2</name>
        <value>localhost:33141</value>
    </property>
```

10. 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 underutilized 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.

In this section:

10.1. 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.

10.2. 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 multi-part files, which are indexed in order to retain the original separation of data. Moreover, the part files 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 user must de-archive (expand) the archive 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 it.

  For example, if we have the following command to copy a 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.

The Hadoop archiving tool is discussed in further detail in the next section.

10.3. Creating a Hadoop Archive

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

```
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 will 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, you would 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
```

10.4. 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, you would use the following command to list the files in the archive:

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

The output would be:

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

As you may recall, these archives were created with the following command:

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

If we were to 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 output would be:

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

Notice that with the modified parent argument, the files have been archived relative to `/user/` rather than `/user/hadoop`.

10.5. Hadoop Archives and MapReduce

To use Hadoop Archives with MapReduce, you need to 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 would need to 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.

11. High Availability for Hive Metastore

This document is intended for system administrators who need to configure the Hive Metastore service for High Availability.

11.1. Use Cases and Fail Over Scenarios

This section provides information on the use cases and fail over scenarios for high availability (HA) in the Hive metastore.

Use Cases

The metastore HA solution is designed to handle metastore service failures. Whenever a deployed metastore service goes down, metastore service can remain unavailable for a considerable time until service is brought back up. To avoid such outages, deploy the metastore service in HA mode.

Deployment Scenarios

We recommend deploying the metastore service on multiple boxes concurrently. Each Hive metastore client will read the configuration property `hive.metastore.uris` to get a list of metastore servers with which it can try to communicate.

```
<property>
 <name> hive.metastore.uris </name>
 <value> thrift://$Hive_Metastore_Server_Host_Machine_FQDN </value>
 <description> A comma separated list of metastore uris on which metastore
 service is running </description>
 </property>
```

These metastore servers store their state in a MySQL HA cluster, which should be set up as recommended in the whitepaper *"MySQL Replication for Failover Protection."*

In the case of a secure cluster, each of the metastore servers will additionally need to have the following configuration property in its `hive-site.xml` file.

```
<property>
 <name> hive.cluster.delegation.token.store.class </name>
 <value> org.apache.hadoop.hive.thrift.DBTokenStore </value>
 </property>
```

Fail Over Scenario

A Hive metastore client always uses the first URI to connect with the metastore server. In case the metastore server becomes unreachable, the client will randomly pick up a URI from the list and try connecting with that.

11.2. Software Configuration

Complete the following tasks to configure Hive HA solution:

- Install HDP
- Update the Hive Metastore
- Validate configuration

11.2.1. Install HDP

Use the following instructions to install HDP on your cluster hardware. Ensure that you specify the virtual machine (configured in the previous section) as your NameNode.

1. Download Apache Ambari using the instructions provided here.

   Note

   Do not start the Ambari server until you have configured the relevant templates as outlined in the following steps.

2. Edit the `<master-install-machine-for-Hive-Metastore>/etc/hive/conf.server/hive-site.xml` configuration file to add the following properties:

   a. Provide the URI for the client to contact Metastore server. The following property can have a comma separated list when your cluster has multiple Hive Metastore servers.

```
<property>
 <name> hive.metastore.uris </name>
 <value> thrift://$Hive_Metastore_Server_Host_Machine_FQDN </value>
 <description> URI for client to contact metastore server </description>
</property>
```

   b. Configure Hive cluster delegation token storage class.

```
<property>
 <name> hive.cluster.delegation.token.store.class </name>
 <value> org.apache.hadoop.hive.thrift.DBTokenStore </value>
 </property>
```

3. Complete HDP installation.

   - Continue the Ambari installation process using the instructions provided here.
   - Complete the Ambari installation. Ensure that the installation was successful.

11.2.2. Update the Hive Metastore

HDP components configured for HA must use a NameService rather than a NameNode. Use the following instructions to update the Hive Metastore to reference the NameService rather than a Name Node.

Note

Hadoop administrators also often use the following procedure to update the Hive metastore with the new URI for a node in a Hadoop cluster. For example, administrators sometimes rename an existing node as their cluster grows.

1. Open a command prompt on the machine hosting the Hive metastore.
2. Execute the following command to retrieve a list of URIs for the filesystem roots, including the location of the NameService:

   ```
   hive --service metatool -listFSRoot
   ```

3. Execute the following command with the `-dryRun` option to test your configuration change before implementing it:

   ```
   hive --service metatool -updateLocation <nameservice-uri> <namenode-uri> -
   dryRun
   ```

4. Execute the command again, this time without the `-dryRun` option:

   ```
   hive --service metatool -updateLocation <nameservice-uri> <namenode-uri>
   ```

11.2.3. Validate configuration

Test various fail over scenarios to validate your configuration.

12. Highly Available Reads with HBase

Apache HBase 0.98.0 enables HBase administrators to configure their HBase clusters with read-only High Availability, or HA. This feature greatly benefits HBase applications that require low latency queries but can tolerate stale data, such as remote sensors, distributed messaging, object stores, and user profile management. HBase provides read-only HA on a per-table basis by replicating table regions with one or more secondary region servers. See Understanding Regions and RegionServers for more information.

HA for HBase features the following functionality:

- Data safely protected in HDFS
- Failed nodes are automatically recovered
- No single point of failure

However, HBase administrators should carefully consider the the following costs associated with using secondary region servers:

- Double or triple memstore usage
- Increased block cache usage
- Increased network traffic for log replication
- Extra backup RPCs for replicas

Important

This release of HA for HBase is not compatible with region splits and merges. Do not execute region merges on tables with region replicas. Rather, HBase administrators must split tables before enabling HA for HBase and disable region splits with the `DisabledRegionSplitPolicy`. This can be done with both the HBase API and with the `hbase.regionserver.region.split.policy` property in the region server's `hbase-site.xml` configuration file. This default value can be overridden for individual HBase tables.

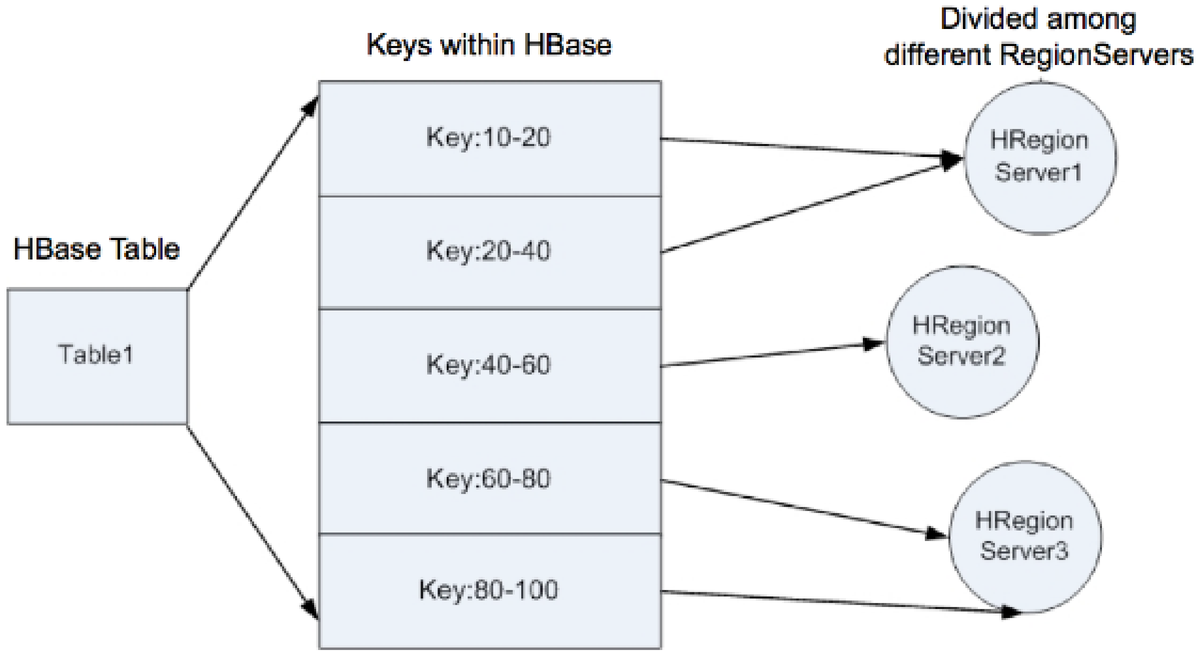
12.1. Understanding HBase Concepts

HBase administrators should understand the following concepts before configuring HA for HBase:

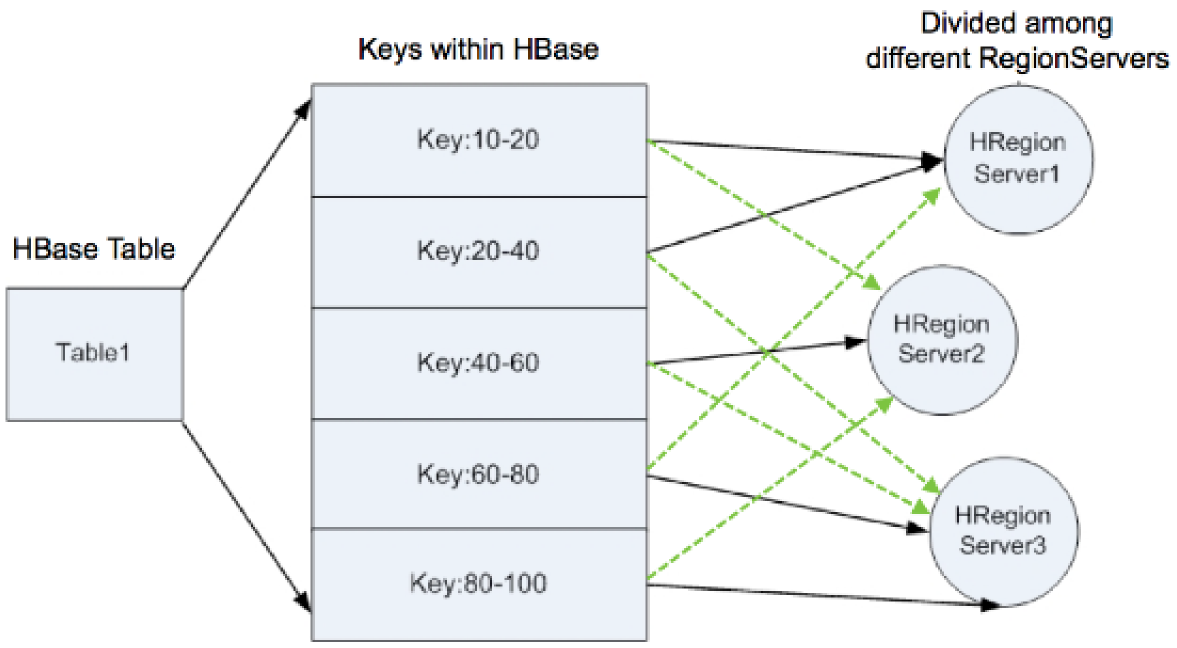
- RegionServers and Secondary Mode
- TIMELINE and STRONG Data Consistency

12.1.1. Region Servers and Secondary Mode

HBase is a key-value store, which allows fast table scanning and data retrieval at petabyte scale. Table keys are assigned to one or more Region Servers. In the following image, for example, Region Server 1 is responsible for responding to queries and scans for keys 10 through 40. If Region Server 1 crashes, keys 10-40 are unavailable.



HA provides a way to access keys 10-40 even if Region Server 1 is not available by assigning other Region Servers as backups. In the following image, Region Server 2 is the secondary region server for keys 10-20 and Region Server 3 is the secondary region server for keys 20-40. Region Server 2 is also the secondary region server for keys 80-100. Secondary Region Servers are not separate processes. Rather, they are Region Servers acting in *secondary mode*. When Region Server 2 services queries and scans for keys 10-20, it acts in secondary mode.



Note

Region Servers acting in secondary mode are also known as *Secondary Region Servers*. However, there is no second server process.

A Region Server in secondary mode can read but not write data. In addition, the data it returns may be stale, as described in the following section.

12.1.2. Timeline and Strong Data Consistency

HBase guarantees *timeline consistency* for all data served from Region Servers in secondary mode, meaning all HBase clients see the same data in the same order, but that data may be slightly stale. Only the primary Region Server is guaranteed to have the latest data. Timeline consistency simplifies the programming logic for complex HBase queries and provides lower latency than quorum-based consistency. By contrast, *strong* data consistency means that the latest data is always served. However, strong data consistency can greatly increase latency since only the primary region server is guarnateed to have the latest data. The HBase API allows application developers to specify which data consistency is required for a query. See Querying Replicas for more information.

Note

The HBase API contains a new method,`Result.isStale()`, to indicate whether data returned in secondary mode does not contain the latest write operation to the primary region server.

12.2. Enabling HA Reads for HBase

Use the following procedure to enable HA for HBase:

1. Add the following configuration properties in the `hbase-site.xml` configuration file to enable HA for HBase:

Table 12.1.

Configuration Property	Description
`hbase.regionserver.storefile.refresh.period`	The period, in milliseconds, for refreshing the store files for the Secondary Region Servers. The default value of `0` indicates that the feature is disabled. Secondary Region Servers receive new store files from the Primary Region Server after the Secondary Region Server refreshes the list of files in the region. However, too frequent refreshes can place an additional load on the NameNode. Read requests are rejected if these store files cannot be refreshed for longer than the HFILE TTL period specifed with the `hbase.master.loadbalancer.class` configuration property.
`hbase.master.loadbalancer.class`	The Java class used for balancing the load of all HBase clients. The default value is `org.apache.hadoop.hbase.masgter.balancer.StochasticLoadBalancer`, the only load balancer that supports reading data from Region Servers in

Configuration Property	Description
	secondary mode.
`hbase.client.allowsInterrupt`	Whether to enable interruption of RPC threads at the client. The default value of `true` is required to enable Primary Region Servers to access other Region Servers in secondary mode.
`hbase.client.primaryCallTimeout.get`	The timeout period, in milliseconds, before an HBase client's RPCs, or remote procedure calls, are submitted to a Secondary Region Server with timeline data consistency. The default value is `10`.
`hbase.client.primaryCallTimeout.multiget`	The timeout period, in milliseconds, before an HBase client's multi-get request, such as `HTable.get(List<GET>))`, is submitted to a Secondary Region Server with timeline data consistency. The default value is `10`. Lower values increase the number of RPCs but will also lower latency.

2. Restart the HBase Master and Region Servers.

12.3. Creating Highly-Available HBase Tables

HBase tables are not highly available by default. Rather, administrators and developers designate a table as HA during table creation.

12.3.1. Creating HA Tables with the HBase Java API

HBase application developers create highly-available HBase tables programmaticaly using the Java API, as shown in the following example:

```
HTableDescriptor htd = new HTableDesscriptor(TableName.valueOf("test_table"));
htd.setRegionReplication(2);
...
admin.createTable(htd);
```

This example creates a table named `test_table` that is replicated to one secondary region server. To replicate `test_table` to two secondary region servers, pass 3 as a parameter to the `setRegionReplication()` method.

12.3.2. Creating HA Tables with the HBase Shell

Create HA tables using the HBase shell using the `REGION_REPLICATION` keyword, as shown in the following example:

```
CREATE 't1', 'f1', {REGION_REPLICATION => 2}
```

This example creates a table named `t1` that is replicated to one secondary region server. To replicate `t1` to two secondary region servers, specify `{REGION_REPLICATION => 3}`.

12.4. Querying Secondary Region Servers

This section describes how to query HA-enabled HBase tables.

12.4.1. Querying HBase with the Java API

The HBase Java API allows application developers to specify the desired data consistency for a query using the `setConsistency()` method, as shown in the following example. A new enum, `CONSISTENCY`, specifies two levels of data consistency: TIMELINE and STRONG.

```
Get get = new Get(row);
get.setConsistency(CONSISTENCY.TIMELINE);
...
Result result = table.get(get);
```

HBase application developers can also pass multiple gets:

```
Get get1 = new Get(row);
get1.setConsistency(Consistency.TIMELINE);
...
ArrayList<Get> gets = new ArrayList<Get>();
...
Result[] results = table.get(gets);
```

The `setConsistency()` method is also available for `Scan` objects:

```
Scan scan = new Scan();
```

```
scan.setConsistency(CONSISTENCY.TIMELINE);
...
ResultScanner scanner = table.getScanner(scan);
```

In addition, use the `Result.isStale()` method to determine whether the query results arrived from the primary or a secondary region server:

```
Result result = table.get(get);
if (result.isStale()) {
   ...
}
```

12.4.2. Querying HBase Interactively

The HBase shell allows administrators and developers to specify the desired data consistency for each query:

```
hbase(main):001:0> get 't1', 'r6', {CONSISTENCY => "TIMELINE"}
```

Interactive scans also accept this syntax:

```
hbase(main):001:0> scan 't1', {CONSISTENCY => 'TIMELINE'}
```

Note

This release of HBase does not provide a mechanism to determine if the results from an interactive query arrived from the primary or a secondary region server.

12.5. Monitoring Secondary Region Servers

HBase provides highly-available tables by replicating table regions. All replicated regions have a unique replica ID. The replica ID for a primary region server is always `0`. The HBase web-based interface displays the replica IDs for all defined table regions. In the following example, the table `t1` has two regions. The secondary region server is identified by a replica ID of `1`.

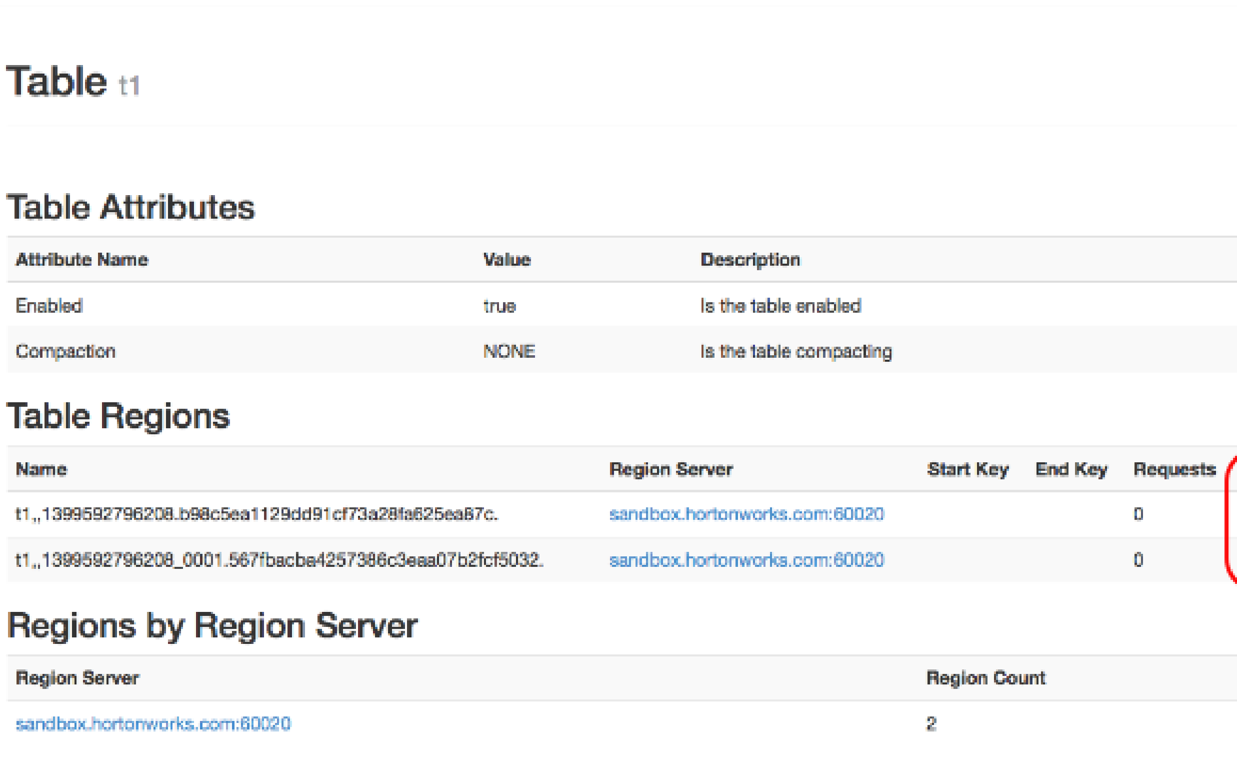


Table t1

Table Attributes

Attribute Name	Value	Description
Enabled	true	Is the table enabled
Compaction	NONE	Is the table compacting

Table Regions

Name	Region Server	Start Key	End Key	Requests
t1,,1399592796208.b98c5ea1129dd91cf73a28fa625ea87c.	sandbox.hortonworks.com:60020			0
t1,,1399592796208_0001.567fbacbe4257386c3eaa07b2fcf5032.	sandbox.hortonworks.com:60020			0

Regions by Region Server

Region Server	Region Count
sandbox.hortonworks.com:60020	2

Point your browser to port 60010 to access the HBase Master Server user interface.

13. HBase Cluster Capacity and Region Sizing

This chapter discusses the topics of planning the capacity of an HBase cluster and the size of its region servers. There are several considerations when planning the capacity of an Apache HBase cluster and performing the initial configuration:

- Node Count and JVM Configuration
- Region Count and Size
- Initial Configuration and Tuning

This chapter requires an understanding of the following HBase concepts:

Table 13.1. HBase Concepts

HBase Concept	Description
Region	A group of contiguous HBase table rows. Tables start with one region and additional regions are dynamically added as the table grows. Regions can be spread across multiple hosts to provide load balancing and quick recovery from failure. There are two types of region: primary and secondary. A secondary region is a replicated primary region located on a different region server.
region server	Serves data requests for one or more regions. A single region is serviced by only one region server, but a region server may

HBase Concept	Description
	serve multiple regions.
Column family	A group of semantically related columns stored together.
Memstore	In-memory storage for a region server. region servers write files to HDFS after the memstore reaches a configurable maximum value specified with the `hbase.hregion.memstore.flush.size` property in the `hbase-site.xml` configuration file.
Write Ahead Log (WAL)	In-memory log where operations are recorded before they are stored in the memstore.
Compaction storm	When the operations stored in the memstore are flushed to disk, HBase consolidates and merges many smaller files into fewer large files. This consolidation is called *compaction*, and it is usually very fast. However, if many region servers hit the data limit specified by the memstore at the same time, HBase performance may degrade from the large number of simultaneous major compactions.

HBase Concept	Description
	Administrators can avoid this by manually splitting tables over time.

13.1. Node Count and JVM Configuration

The number of nodes in an HBase cluster is typically driven by the following considerations:

- Physical size of the data
- Read/Write Throughput

Physical Size of the Data

The physical size of data on disk is affected by the following factors:

Table 13.2. Factors Affecting Physical Size of Data

Factor Affecting Size of Physical Data	Description
HBase Overhead	The default amount of disk space required for a single HBase table cell. Smaller table cells require less overhead. The minimum cell size is 24 bytes and the default maximum is 10485760 bytes. Administrators can customize the maximum cell size with the `hbase.client.keyvalue.maxsize` property in the `hbase-site.xml` configuration file. HBase table cells are aggregated into blocks, and the block size is also configurable for each column family

Factor Affecting Size of Physical Data	Description
	with the `hbase.mapreduce.hfileoutputformat.blocksize` property. The default value is 65536 bytes. Administrators may reduce this value for tables with highly random data access patterns to improve query latency.
Compression	Choose a data compression tool that makes sense for your data to reduce the physical size of data on disk. Unfortunately, HBase cannot ship with LZO due to licensing issues. However, HBase administrators may install LZO after installing HBase. GZIP provides better compression than LZO but is slower. HBase also supports Snappy.
HDFS Replication	HBase uses HDFS for storage, so replicating HBase data stored in HDFS affects the total physical size of data. A typical replication factor of 3 for all HBase tables in a cluster would triple the physical size of the stored data.
region server Write	The size of the Write Ahead Log, or WAL,

Factor Affecting Size of Physical Data	Description
Ahead Log (WAL)	for each region server has minimal impact on the physical size of data. The size of the WAL is typically fixed at less than half of the memory for the region server. Although this factor is included here for completeness, the impact of the WAL on data size is negligible and its size is usually not configured.

Read/Write Throughput

The number of nodes in an HBase cluster may also be driven by required throughput for disk reads and/or writes. The throughput per node greatly depends on table cell size, data request patterns, as well as node and cluster configuration. Use YCSB tools to test the throughput of a single node or a cluster to determine if read/write throughput should drive your the number of nodes in your HBase cluster. A typical throughput for write operations for one region server is 5-15 Mb/s. Unfortunately, there is no good estimate for read throughput; it varies greatly depending on physical data size, request patterns, and hit rate for the block cache.

13.2. Region Count and Size

In general, an HBase cluster runs smoother with fewer regions. However, administrators cannot directly configure the number of regions for a region server. However, administrators can indirectly increase the number of regions in the following ways: In addition, administrators can indirectly affect the number of regions for a region server in the following ways:

- Increase the size of the memstore for a region server
- Increase the size of a region

In addition, administrators can increase the number of regions for a region server by pre-splitting large regions to spread data and the request load across the cluster. HBase allows administrators to individually configure each HBase table, which is useful when tables have different workloads and use cases. Most region settings can be set on a per-table basis with HTableDescriptor class, as well as the HBase CLI. These methods override the properties in the `hbase-site.xml` configuration file.

Note

The HDFS replication factor affects only disk usage and should not be considered when planning the size of regions. The other factors described in the table above are applicable.

13.2.1. Increase Memstore Size for region server

Usage of the region server's memstore largely determines the maximum number of regions for the region server. Each region has its own memstores, one for each column family, which grow to a configurable size, usually between 128 and 256 Mb. Administrators specify this size with the `hbase.hregion.memstore.flush.size` property in the `hbase-site.xml` configuration file. The region server dedicates some fraction of total memory to region memstores based on the value of the `hbase.regionserver.global.memstore.size` configuration property. If usage exceeds this configurable size, HBase may become unresponsive or compaction storms might occur.

Use the following formula to estimate the number of Regions for a region server:

```
(regionserver_memory_size) * (memstore_fraction) / ((memstore_size) *
 (num_column_families))
```

For example, assume the following configuration:

- region server with 16 Gb RAM (or 16384 Mb)
- Memstore fraction of .4
- Memstore with 128 Mb RAM
- 1 column family in table

The formula for this configuration would look as follows:

```
(16384 Mb * .4) / ((128 Mb * 1) = approximately 51 regions
```

The easiest way to decrease the number of regions for this example region server is increase the RAM of the memstore to 256 Mb. The reconfigured region server would then have approximately 25 regions, and the HBase cluster will run more smoothly if the reconfiguration is applied to all region servers in the cluster. The formula can be used for multiple tables with the same configuration by using the total number of column families in all the tables.

Note

The formula assumes all regions are filled at approximately the same rate. If a fraction of the cluster's regions are written to, divide the result by this fraction.

Tip

If the data request pattern is dominated by write operations rather than read operations, increase the memstore fraction. However, this increase negatively impacts the block cache.

13.2.2. Increase Size of the Region

The other way to indirectly increase the number of regions for a region server is to increase the size of the region with the `hbase.hregion.max.filesize` property in the `hbase-site.xml` configuration file. Administrators increase the number of regions for a region server by increasing the specified size at which new regions are dynamically allocated. Maximum region size is primarily limited by compactions. Very large compactions can degrade cluster performance. The recommended maximum region size is 10 - 20 Gb. For HBase clusters running version 0.90.x, the maximum recommended region size is 4 Gb and the default is 256 Mb. If you are unable to estimate the size of your tables, retain the default value. Increase the region size only if your table cells tend to be 100 Kb or larger.

Note

HBase 0.98 introduces stripe compactions as an experimental feature that also allows administrators to increase the size of regions. See Experimental: Stripe Compactions at the Apache HBase site for more information.

13.3. Initial Configuration and Tuning

HBase administrators typically use the following methods to initially configure the cluster:

- Increase the request handler thread count
- Configure the size and number of WAL files
- Configure compactions
- Pre-split tables
- Tune JVM garbage collection

Increase the Request Handler Thread Count

Administrators who expect their HBase cluster to experience a high volume request pattern should increase the number of listeners generated by the region servers. Use the `hbase.regionserver.handler.count` property in the `hbase-site.xml` configuration file to set the number higher than the default value of `30`.

Configure the Size and Number of WAL Files

HBase uses the Write Ahead Log, or WAL, to recover memstore data not yet flushed to disk if a region server crashes. Administrators should configure these WAL files to be slightly smaller than the HDFS block size. By default, an HDFS block is 64 Mb and a WAL is approximately 60 Mb. Hortonworks recommends that administrators ensure that enough WAL files are allocated to contain the total capacity of the memstores. Use the following formula to determine the number of WAL files needed:

```
(regionserver_heap_size * memstore fraction) / (default_WAL_size)
```

For example, assume the following HBase cluster configuration:

- 16 GB RegionServer heap

- 0.4 memstore fraction
- 60 MB default WAL size

The formula for this configuration looks as follows:

```
(16384 MB * 0.4 / 60 MB = approximately 109 WAL files
```

Use the following properties in the `hbase-site.xml` configuration file to configure the size and number of WAL files:

Table 13.3. Authentication Schemes in TCP Transport Mode

Configuration Property	Description	Default
`hbase.regionserver.maxlogs`	Sets the maximum number of WAL files.	32
`hbase.regionserver.logroll.multiplier`	Multiplier of HDFS block size.	0.95
`hbase.regionserver.hlog.blocksize`	Optional override of HDFS block size.	Value assigned to actual HDFS block size.

Tip

If recovery from failure takes longer than expected, try reducing the number of WAL files to improve performance.

Configure Compactions

Administrators who expect their HBase clusters to host large amounts of data should consider the affect that compactions have on write throughput. For write-intensive data request patterns, administrators should consider less frequent compactions and more store files per region. Use the `hbase.hstore.compaction.min` property in the `hbase-site.xml` configuration file to increase the minimum number of files required to trigger a compaction. Administrators opting to increase this value should also increase the value assigned to the `hbase.hstore.blockingStoreFiles` property since more files will accumulate.

Pre-split Tables

Administrators can pre-split tables during table creation based on the target number of regions per region server to avoid costly dynamic splitting as the table starts to fill up. In addition, it ensures that the regions in the pre-split table are distributed across many host machines. Pre-splitting a table avoids the cost of compactions required to rewrite the data into separate physical files during automatic splitting. If a table is expected to grow very large, administrators should create at least one region per region server. However, do not immediately split the table into the total number of desired regions. Rather, choose a low to intermediate value. For multiple tables, do not create more than one region per region server, especially if you are uncertain how large the table will grow. Creating too many regions for a table that will never exceed 100 Mb in size isn't useful; a single region can adequately services a table of this size.

Configure the JVM Garbage Collector

A region server cannot utilize a very large heap due to the cost of garbage collection. Administrators should specify no more than 24 GB for one region server.

14. 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. So-called "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.

14.1. Prerequisites

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

14.2. 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.

Table 14.1. 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.
`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`.
`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

Property Name	Property Value	Description
		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#getFileVBlockStorageLocations API.
`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 lots of 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>
```

15. Timeline Server

This guide describes how to configure and run the Timeline Server, which enables you to collect generic and per-framework information about YARN applications.

In this section:

15.1. Introduction

The Timeline Server maintains historical state and provides metrics visibility for YARN applications, similar to the functionality the Job History Server provides for MapReduce.

The Timeline Server provides the following information:

- Generic Information about Completed Applications

  Generic information includes application-level data such as queue name, user information, information about application attempts, a list of Containers that were run under each application attempt, and information about each Container. Generic data is stored by the ResourceManager in a history store (the default implementation on a file system), and is used by the web UI to display information about completed applications.

- Per-Framework Information for Running and Completed Applications

  Per-framework information is specific to an application or framework. For example, the Hadoop MapReduce framework can include pieces of information such as the number of map tasks, reduce tasks, counters, etc. Application developers can publish this information to the Timeline Server via the TimelineClient (from within a client), the ApplicationMaster, or the application's Containers. This information can then be queried via REST APIs that enable rendering by application/framework-specific UIs.

The Timeline Server is a stand-alone server daemon that is deployed to a cluster node. It may or may not be co-located with the ResourceManager.

15.2. Configuring the Timeline Server

Required Properties

Only one property needs to be specified in the `etc/hadoop/conf/yarn-site.xml` file in order to enable the Timeline Server:

- `yarn.timeline-service.hostname`

  The host name of the Timeline Server web application.

  Example:

```
<property>
     <description>The hostname of the timeline server web application.</
description>
     <name>yarn.timeline-service.hostname</name>
     <value>0.0.0.0</value>
</property>
```

Advanced Properties

In addition to the host name, administrators can also configure the ports of the RPC and the web interfaces, as well as the number of RPC handler threads.

- `yarn.timeline-service.address`

  The default address for the Timeline Server to start the RPC server.

  Example:

```
<property>
     <description>This is default address for the timeline server to start
 the RPC server.</description>
     <name>yarn.timeline-service.address</name>
     <value>${yarn.timeline-service.hostname}:10200</value>
</property>
```

- `yarn.timeline-service.webapp.address`

  The HTTP address of the Timeline Server web application.

  Example:

```
<property>
     <description>The http address of the timeline server web application.</
description>
     <name>yarn.timeline-service.webapp.address</name>
     <value>${yarn.timeline-service.hostname}:8188</value>
</property>
```

- `yarn.timeline-service.webapp.https.address`

  The HTTPS address of the Timeline Server web application.

  Example:

```
<property>
     <description>The https adddress of the timeline server web application.
</description>
     <name>yarn.timeline-service.webapp.https.address</name>
     <value>${yarn.timeline-service.hostname}:8190</value>
```

```
  </property>
```

- `yarn.timeline-service.handler-thread-count`

  The handler thread count to serve the client RPC requests.

  Example:

```
<property>
     <description>Handler thread count to serve the client RPC requests.</
description>
     <name>yarn.timeline-service.handler-thread-count</name>
     <value>10</value>
</property>
```

15.3. Configuring Generic Data Collection

Enabling the Collection of Generic Data

- `yarn.timeline-service.generic-application-history.enabled`

  This property indicates to the ResourceManager, as well as to clients, whether or not the Generic History Service (GHS) is enabled. If the GHS is enabled, the ResourceManager begins recording historical data that the GHS can consume, and clients can redirect to the GHS when applications finish running.

  Example:

```
<property>
     <description>Enable or disable the GHS</description>
     <name>yarn.timeline-service.generic-application-history.enabled</name>
     <value>true</value>
</property>
```

Configuring the Store for Generic Data

- `yarn.timeline-service.generic-application-history.store-class`

  The store class name for the history store. Defaults to the file system store.

  Example:

```
<property>
     <description>Store class name for history store, defaulting to file
 system store</description>
     <name>yarn.timeline-service.generic-application-history.store-class</
name>
     <value>org.apache.hadoop.yarn.server.applicationhistoryservice.
FileSystemApplicationHistoryStore</value>
</property>
```

- `yarn.timeline-service.generic-application-history.fs-history-store.uri`

  The URI pointing to the FileSystem path location where the
  history will be persisted. This must be supplied when using

`org.apache.hadoop.yarn.server.applicationhistoryservice.FileSystemApplicatio` as the value for `yarn.timeline-service.generic-application-history.store-class`.

Example:

```
<property>
     <description>URI pointing to the location of the FileSystem path where
 the history will be persisted.</description>
     <name>yarn.timeline-service.generic-application-history.fs-history-
store.uri</name>
     <value>${hadoop.log.dir}/yarn/system/history</value>
</property>
```

- `yarn.timeline-service.generic-application-history.fs-history-store.compression-type`

  The T-file compression types used to compress history data. The available values are:

  - "none" – No compression
  - "lzo" – LZO compression
  - "gz" – GZIP compression

  Example:

```
<property>
     <description>T-file compression types used to compress history data.</
description>
     <name>yarn.timeline-service.generic-application-history.fs-history-
store.compression-type</name>
     <value>none</value>
</property>
```

15.4. Configuring Per-Framework Data Collection

Configuration for Per-framework Data

- `yarn.timeline-service.enabled`

  Indicates to clients whether or not the Timeline Server is enabled. If it is enabled, the TimelineClient library used by end-users will post entities and events to the Timeline Server.

  Example:

```
<property>
     <description>Enable or disable the Timeline Server.</description>
     <name>yarn.timeline-service.enabled</name>
     <value>true</value>
</property>
```

- `yarn.timeline-service.store-class`

  The class name for the Timeline store.

Example:

```
<property>
     <description>Store class name for timeline store</description>
     <name>yarn.timeline-service.store-class</name>
     <value>org.apache.hadoop.yarn.server.timeline.LeveldbTimelineStore</
value>
</property>
```

- `yarn.timeline-service.leveldb-timeline-store.path`

  The store file path and name for the Timeline Server LevelDB store (if the LevelDB store is used).

  Example:

```
<property>
     <description>Store file name for leveldb timeline store</description>
     <name>yarn.timeline-service.leveldb-timeline-store.path</name>
     <value>${yarn.log.dir}/timeline</value>
</property>
```

- `yarn.timeline-service.ttl-enable`

  Enable age-off of timeline store data.

  Example:

```
<property>
  <description>Enable age off of timeline store data.</description>
  <name>yarn.timeline-service.ttl-enable</name>
  <value>true</value>
</property>
```

- `yarn.timeline-service.ttl-ms`

  The Time-to-live for timeline store data (in milliseconds).

  Example:

```
<property>
  <description>Time to live for timeline store data in milliseconds.</
description>
  <name>yarn.timeline-service.ttl-ms</name>
  <value>604800000</value>
</property>
```

15.5. Running the Timeline Server

To start the Timeline Server, run the following command:

```
$ yarn historyserver
```

To start the Timeline Server as a daemon, run the following command:

```
$ sbin/yarn-daemon.sh start historyserver
```

15.6. Accessing Generic Data from the Command-Line

You can use the following commands to access application generic history data from the command-line. Note that these same commands can be used to obtain corresponding information about running applications.

```
$ yarn application -status <Application ID>
$ yarn applicationattempt -list <Application ID>
$ yarn applicationattempt -status <Application Attempt ID>
$ yarn container -list <Application Attempt ID>
$ yarn container -status <Container ID>
```

15.7. Publishing Per-Framework Data in Applications

Developers can define the information they would like to record for their applications by composing `TimelineEntity` and `TimelineEvent` objects, and then putting the entities and events to the Timeline server via `TimelineClient`. For example:

```
// Create and start the Timeline client
TimelineClient client = TimelineClient.createTimelineClient();
client.init(conf);
client.start();

TimelineEntity entity = null;
// Compose the entity
try {
  TimelinePutResponse response = client.putEntities(entity);
} catch (IOException e) {
  // Handle the exception
} catch (YarnException e) {
  // Handle the exception
}

// Stop the Timeline client
client.stop();
```

16. WebHDFS Administrator Guide

Use the following instructions to set upo 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>
   ```

2. [Optional] - If running a secure cluster, follow the steps listed below.

   a. 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

   b. Create keytab files for the HTTP principals.

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

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

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

   d. 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

3. Restart the NameNode and DataNode services using the instructions provided here.