

PART III

Hadoop Operations

Setting Up a Hadoop Cluster

This chapter explains how to set up Hadoop to run on a cluster of machines. Running HDFS, MapReduce, and YARN on a single machine is great for learning about these systems, but to do useful work, they need to run on multiple nodes.

There are a few options when it comes to getting a Hadoop cluster, from building your own, to running on rented hardware or using an offering that provides Hadoop as a hosted service in the cloud. The number of hosted options is too large to list here, but even if you choose to build a Hadoop cluster yourself, there are still a number of installation options:

Apache tarballs

The Apache Hadoop project and related projects provide binary (and source) tarballs for each release. Installation from binary tarballs gives you the most flexibility but entails the most amount of work, since you need to decide on where the installation files, configuration files, and logfiles are located on the filesystem, set their file permissions correctly, and so on.

Packages

RPM and Debian packages are available from the [Apache Bigtop project](#), as well as from all the Hadoop vendors. Packages bring a number of advantages over tarballs: they provide a consistent filesystem layout, they are tested together as a stack (so you know that the versions of Hadoop and Hive, say, will work together), and they work well with configuration management tools like Puppet.

Hadoop cluster management tools

Cloudera Manager and Apache Ambari are examples of dedicated tools for installing and managing a Hadoop cluster over its whole lifecycle. They provide a simple web UI, and are the recommended way to set up a Hadoop cluster for most users and operators. These tools encode a lot of operator knowledge about running Hadoop. For example, they use heuristics based on the hardware profile (among

other factors) to choose good defaults for Hadoop configuration settings. For more complex setups, like HA, or secure Hadoop, the management tools provide well-tested wizards for getting a working cluster in a short amount of time. Finally, they add extra features that the other installation options don't offer, such as unified monitoring and log search, and rolling upgrades (so you can upgrade the cluster without experiencing downtime).

This chapter and the next give you enough information to set up and operate your own basic cluster, but even if you are using Hadoop cluster management tools or a service in which a lot of the routine setup and maintenance are done for you, these chapters still offer valuable information about how Hadoop works from an operations point of view. For more in-depth information, I highly recommend *Hadoop Operations* by Eric Sammer (O'Reilly, 2012).

Cluster Specification

Hadoop is designed to run on commodity hardware. That means that you are not tied to expensive, proprietary offerings from a single vendor; rather, you can choose standardized, commonly available hardware from any of a large range of vendors to build your cluster.

“Commodity” does not mean “low-end.” Low-end machines often have cheap components, which have higher failure rates than more expensive (but still commodity-class) machines. When you are operating tens, hundreds, or thousands of machines, cheap components turn out to be a false economy, as the higher failure rate incurs a greater maintenance cost. On the other hand, large database-class machines are not recommended either, since they don't score well on the price/performance curve. And even though you would need fewer of them to build a cluster of comparable performance to one built of mid-range commodity hardware, when one did fail, it would have a bigger impact on the cluster because a larger proportion of the cluster hardware would be unavailable.

Hardware specifications rapidly become obsolete, but for the sake of illustration, a typical choice of machine for running an HDFS datanode and a YARN node manager in 2014 would have had the following specifications:

Processor

Two hex/octo-core 3 GHz CPUs

Memory

64–512 GB ECC RAM¹

1. ECC memory is strongly recommended, as several Hadoop users have reported seeing many checksum errors when using non-ECC memory on Hadoop clusters.

Storage

12–24 × 1–4 TB SATA disks

Network

Gigabit Ethernet with link aggregation

Although the hardware specification for your cluster will assuredly be different, Hadoop is designed to use multiple cores and disks, so it will be able to take full advantage of more powerful hardware.

Why Not Use RAID?

HDFS clusters do not benefit from using RAID (redundant array of independent disks) for datanode storage (although RAID is recommended for the namenode's disks, to protect against corruption of its metadata). The redundancy that RAID provides is not needed, since HDFS handles it by replication between nodes.

Furthermore, RAID striping (RAID 0), which is commonly used to increase performance, turns out to be *slower* than the JBOD (just a bunch of disks) configuration used by HDFS, which round-robins HDFS blocks between all disks. This is because RAID 0 read and write operations are limited by the speed of the slowest-responding disk in the RAID array. In JBOD, disk operations are independent, so the average speed of operations is greater than that of the slowest disk. Disk performance often shows considerable variation in practice, even for disks of the same model. In some **benchmarking carried out on a Yahoo! cluster**, JBOD performed 10% faster than RAID 0 in one test (Gridmix) and 30% better in another (HDFS write throughput).

Finally, if a disk fails in a JBOD configuration, HDFS can continue to operate without the failed disk, whereas with RAID, failure of a single disk causes the whole array (and hence the node) to become unavailable.

Cluster Sizing

How large should your cluster be? There isn't an exact answer to this question, but the beauty of Hadoop is that you can start with a small cluster (say, 10 nodes) and grow it as your storage and computational needs grow. In many ways, a better question is this: how fast does your cluster need to grow? You can get a good feel for this by considering storage capacity.

For example, if your data grows by 1 TB a day and you have three-way HDFS replication, you need an additional 3 TB of raw storage per day. Allow some room for intermediate files and logfiles (around 30%, say), and this is in the range of one (2014-vintage) machine per week. In practice, you wouldn't buy a new machine each week and add it to the cluster. The value of doing a back-of-the-envelope calculation like this is that it gives

you a feel for how big your cluster should be. In this example, a cluster that holds two years' worth of data needs 100 machines.

Master node scenarios

Depending on the size of the cluster, there are various configurations for running the master daemons: the namenode, secondary namenode, resource manager, and history server. For a small cluster (on the order of 10 nodes), it is usually acceptable to run the namenode and the resource manager on a single master machine (as long as at least one copy of the namenode's metadata is stored on a remote filesystem). However, as the cluster gets larger, there are good reasons to separate them.

The namenode has high memory requirements, as it holds file and block metadata for the entire namespace in memory. The secondary namenode, although idle most of the time, has a comparable memory footprint to the primary when it creates a checkpoint. (This is explained in detail in [“The filesystem image and edit log” on page 318.](#)) For filesystems with a large number of files, there may not be enough physical memory on one machine to run both the primary and secondary namenode.

Aside from simple resource requirements, the main reason to run masters on separate machines is for high availability. Both HDFS and YARN support configurations where they can run masters in active-standby pairs. If the active master fails, then the standby, running on separate hardware, takes over with little or no interruption to the service. In the case of HDFS, the standby performs the checkpointing function of the secondary namenode (so you don't need to run a standby and a secondary namenode).

Configuring and running Hadoop HA is not covered in this book. Refer to the Hadoop website or vendor documentation for details.

Network Topology

A common Hadoop cluster architecture consists of a two-level network topology, as illustrated in [Figure 10-1](#). Typically there are 30 to 40 servers per rack (only 3 are shown in the diagram), with a 10 Gb switch for the rack and an uplink to a core switch or router (at least 10 Gb or better). The salient point is that the aggregate bandwidth between nodes on the same rack is much greater than that between nodes on different racks.

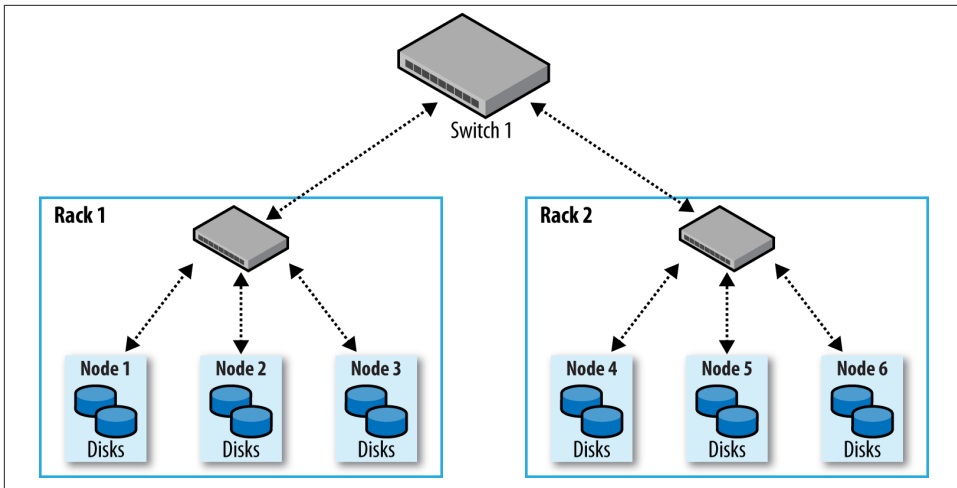


Figure 10-1. Typical two-level network architecture for a Hadoop cluster

Rack awareness

To get maximum performance out of Hadoop, it is important to configure Hadoop so that it knows the topology of your network. If your cluster runs on a single rack, then there is nothing more to do, since this is the default. However, for multirack clusters, you need to map nodes to racks. This allows Hadoop to prefer within-rack transfers (where there is more bandwidth available) to off-rack transfers when placing MapReduce tasks on nodes. HDFS will also be able to place replicas more intelligently to trade off performance and resilience.

Network locations such as nodes and racks are represented in a tree, which reflects the network “distance” between locations. The namenode uses the network location when determining where to place block replicas (see [“Network Topology and Hadoop” on page 70](#)); the MapReduce scheduler uses network location to determine where the closest replica is for input to a map task.

For the network in [Figure 10-1](#), the rack topology is described by two network locations—say, `/switch1/rack1` and `/switch1/rack2`. Because there is only one top-level switch in this cluster, the locations can be simplified to `/rack1` and `/rack2`.

The Hadoop configuration must specify a map between node addresses and network locations. The map is described by a Java interface, `DNSToSwitchMapping`, whose signature is:

```

public interface DNSToSwitchMapping {
    public List<String> resolve(List<String> names);
}

```

The `names` parameter is a list of IP addresses, and the return value is a list of corresponding network location strings. The `net.topology.node.switch.mapping.impl` configuration property defines an implementation of the `DNSToSwitchMapping` interface that the namenode and the resource manager use to resolve worker node network locations.

For the network in our example, we would map *node1*, *node2*, and *node3* to */rack1*, and *node4*, *node5*, and *node6* to */rack2*.

Most installations don't need to implement the interface themselves, however, since the default implementation is `ScriptBasedMapping`, which runs a user-defined script to determine the mapping. The script's location is controlled by the property `net.topology.script.file.name`. The script must accept a variable number of arguments that are the hostnames or IP addresses to be mapped, and it must emit the corresponding network locations to standard output, separated by whitespace. The [Hadoop wiki](#) has an example.

If no script location is specified, the default behavior is to map all nodes to a single network location, called */default-rack*.

Cluster Setup and Installation

This section describes how to install and configure a basic Hadoop cluster from scratch using the Apache Hadoop distribution on a Unix operating system. It provides background information on the things you need to think about when setting up Hadoop. For a production installation, most users and operators should consider one of the Hadoop cluster management tools listed at the beginning of this chapter.

Installing Java

Hadoop runs on both Unix and Windows operating systems, and requires Java to be installed. For a production installation, you should select a combination of operating system, Java, and Hadoop that has been certified by the vendor of the Hadoop distribution you are using. There is also a page on the [Hadoop wiki](#) that lists combinations that community members have run with success.

Creating Unix User Accounts

It's good practice to create dedicated Unix user accounts to separate the Hadoop processes from each other, and from other services running on the same machine. The HDFS, MapReduce, and YARN services are usually run as separate users, named `hdfs`, `mapred`, and `yarn`, respectively. They all belong to the same hadoop group.

Installing Hadoop

Download Hadoop from the [Apache Hadoop releases page](#), and unpack the contents of the distribution in a sensible location, such as `/usr/local` (`/opt` is another standard choice; note that Hadoop should not be installed in a user's home directory, as that may be an NFS-mounted directory):

```
% cd /usr/local
% sudo tar xzf hadoop-x.y.z.tar.gz
```

You also need to change the owner of the Hadoop files to be the `hadoop` user and group:

```
% sudo chown -R hadoop:hadoop hadoop-x.y.z
```

It's convenient to put the Hadoop binaries on the shell path too:

```
% export HADOOP_HOME=/usr/local/hadoop-x.y.z
% export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

Configuring SSH

The Hadoop control scripts (but not the daemons) rely on SSH to perform cluster-wide operations. For example, there is a script for stopping and starting all the daemons in the cluster. Note that the control scripts are optional—cluster-wide operations can be performed by other mechanisms, too, such as a distributed shell or dedicated Hadoop management applications.

To work seamlessly, SSH needs to be set up to allow passwordless login for the `hdfs` and `yarn` users from machines in the cluster.² The simplest way to achieve this is to generate a public/private key pair and place it in an NFS location that is shared across the cluster.

First, generate an RSA key pair by typing the following. You need to do this twice, once as the `hdfs` user and once as the `yarn` user:

```
% ssh-keygen -t rsa -f ~/.ssh/id_rsa
```

Even though we want passwordless logins, keys without passphrases are not considered good practice (it's OK to have an empty passphrase when running a local pseudo-distributed cluster, as described in [Appendix A](#)), so we specify a passphrase when prompted for one. We use `ssh-agent` to avoid the need to enter a password for each connection.

The private key is in the file specified by the `-f` option, `~/.ssh/id_rsa`, and the public key is stored in a file with the same name but with `.pub` appended, `~/.ssh/id_rsa.pub`.

2. The `mapred` user doesn't use SSH, as in Hadoop 2 and later, the only MapReduce daemon is the job history server.

Next, we need to make sure that the public key is in the `~/.ssh/authorized_keys` file on all the machines in the cluster that we want to connect to. If the users' home directories are stored on an NFS filesystem, the keys can be shared across the cluster by typing the following (first as `hdfs` and then as `yarn`):

```
% cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

If the home directory is not shared using NFS, the public keys will need to be shared by some other means (such as *ssh-copy-id*).

Test that you can SSH from the master to a worker machine by making sure *ssh-agent* is running,³ and then run *ssh-add* to store your passphrase. You should be able to SSH to a worker without entering the passphrase again.

Configuring Hadoop

Hadoop must have its configuration set appropriately to run in distributed mode on a cluster. The important configuration settings to achieve this are discussed in “[Hadoop Configuration](#)” on page 292.

Formatting the HDFS Filesystem

Before it can be used, a brand-new HDFS installation needs to be formatted. The formatting process creates an empty filesystem by creating the storage directories and the initial versions of the namenode's persistent data structures. Datanodes are not involved in the initial formatting process, since the namenode manages all of the filesystem's metadata, and datanodes can join or leave the cluster dynamically. For the same reason, you don't need to say how large a filesystem to create, since this is determined by the number of datanodes in the cluster, which can be increased as needed, long after the filesystem is formatted.

Formatting HDFS is a fast operation. Run the following command as the `hdfs` user:

```
% hdfs namenode -format
```

Starting and Stopping the Daemons

Hadoop comes with scripts for running commands and starting and stopping daemons across the whole cluster. To use these scripts (which can be found in the *sbin* directory), you need to tell Hadoop which machines are in the cluster. There is a file for this purpose, called *slaves*, which contains a list of the machine hostnames or IP addresses, one per line. The *slaves* file lists the machines that the datanodes and node managers should run on. It resides in Hadoop's configuration directory, although it may be placed elsewhere

3. See its man page for instructions on how to start *ssh-agent*.

(and given another name) by changing the `HADOOP_SLAVES` setting in *hadoop-env.sh*. Also, this file does not need to be distributed to worker nodes, since they are used only by the control scripts running on the namenode or resource manager.

The HDFS daemons are started by running the following command as the `hdfs` user:

```
% start-dfs.sh
```

The machine (or machines) that the namenode and secondary namenode run on is determined by interrogating the Hadoop configuration for their hostnames. For example, the script finds the namenode's hostname by executing the following:

```
% hdfs getconf -namenodes
```

By default, this finds the namenode's hostname from `fs.defaultFS`. In slightly more detail, the *start-dfs.sh* script does the following:

- Starts a namenode on each machine returned by executing `hdfs getconf -namenodes`⁴
- Starts a datanode on each machine listed in the *slaves* file
- Starts a secondary namenode on each machine returned by executing `hdfs getconf -secondarynamenodes`

The YARN daemons are started in a similar way, by running the following command as the `yarn` user on the machine hosting the resource manager:

```
% start-yarn.sh
```

In this case, the resource manager is always run on the machine from which the *start-yarn.sh* script was run. More specifically, the script:

- Starts a resource manager on the local machine
- Starts a node manager on each machine listed in the *slaves* file

Also provided are *stop-dfs.sh* and *stop-yarn.sh* scripts to stop the daemons started by the corresponding start scripts.

These scripts start and stop Hadoop daemons using the *hadoop-daemon.sh* script (or the *yarn-daemon.sh* script, in the case of YARN). If you use the aforementioned scripts, you shouldn't call *hadoop-daemon.sh* directly. But if you need to control Hadoop daemons from another system or from your own scripts, the *hadoop-daemon.sh* script is a good integration point. Likewise, *hadoop-daemons.sh* (with an "s") is handy for starting the same daemon on a set of hosts.

4. There can be more than one namenode when running HDFS HA.

Finally, there is only one MapReduce daemon—the job history server, which is started as follows, as the `mapred` user:

```
% mr-jobhistory-daemon.sh start historyserver
```

Creating User Directories

Once you have a Hadoop cluster up and running, you need to give users access to it. This involves creating a home directory for each user and setting ownership permissions on it:

```
% hadoop fs -mkdir /user/username  
% hadoop fs -chown username:username /user/username
```

This is a good time to set space limits on the directory. The following sets a 1 TB limit on the given user directory:

```
% hdfs dfsadmin -setSpaceQuota 1t /user/username
```

Hadoop Configuration

There are a handful of files for controlling the configuration of a Hadoop installation; the most important ones are listed in [Table 10-1](#).

Table 10-1. Hadoop configuration files

Filename	Format	Description
<i>hadoop-env.sh</i>	Bash script	Environment variables that are used in the scripts to run Hadoop
<i>mapred-env.sh</i>	Bash script	Environment variables that are used in the scripts to run MapReduce (overrides variables set in <i>hadoop-env.sh</i>)
<i>yarn-env.sh</i>	Bash script	Environment variables that are used in the scripts to run YARN (overrides variables set in <i>hadoop-env.sh</i>)
<i>core-site.xml</i>	Hadoop configuration XML	Configuration settings for Hadoop Core, such as I/O settings that are common to HDFS, MapReduce, and YARN
<i>hdfs-site.xml</i>	Hadoop configuration XML	Configuration settings for HDFS daemons: the namenode, the secondary namenode, and the datanodes
<i>mapred-site.xml</i>	Hadoop configuration XML	Configuration settings for MapReduce daemons: the job history server
<i>yarn-site.xml</i>	Hadoop configuration XML	Configuration settings for YARN daemons: the resource manager, the web app proxy server, and the node managers
<i>slaves</i>	Plain text	A list of machines (one per line) that each run a datanode and a node manager
<i>hadoop-metrics2.properties</i>	Java properties	Properties for controlling how metrics are published in Hadoop (see “ Metrics and JMX ” on page 331)
<i>log4j.properties</i>	Java properties	Properties for system logfiles, the namenode audit log, and the task log for the task JVM process (“ Hadoop Logs ” on page 172)

Filename	Format	Description
<i>hadoop-policy.xml</i>	Hadoop configuration XML	Configuration settings for access control lists when running Hadoop in secure mode

These files are all found in the *etc/hadoop* directory of the Hadoop distribution. The configuration directory can be relocated to another part of the filesystem (outside the Hadoop installation, which makes upgrades marginally easier) as long as daemons are started with the `--config` option (or, equivalently, with the `HADOOP_CONF_DIR` environment variable set) specifying the location of this directory on the local filesystem.

Configuration Management

Hadoop does not have a single, global location for configuration information. Instead, each Hadoop node in the cluster has its own set of configuration files, and it is up to administrators to ensure that they are kept in sync across the system. There are parallel shell tools that can help do this, such as *dsh* or *pdsh*. This is an area where Hadoop cluster management tools like Cloudera Manager and Apache Ambari really shine, since they take care of propagating changes across the cluster.

Hadoop is designed so that it is possible to have a single set of configuration files that are used for all master and worker machines. The great advantage of this is simplicity, both conceptually (since there is only one configuration to deal with) and operationally (as the Hadoop scripts are sufficient to manage a single configuration setup).

For some clusters, the one-size-fits-all configuration model breaks down. For example, if you expand the cluster with new machines that have a different hardware specification from the existing ones, you need a different configuration for the new machines to take advantage of their extra resources.

In these cases, you need to have the concept of a *class* of machine and maintain a separate configuration for each class. Hadoop doesn't provide tools to do this, but there are several excellent tools for doing precisely this type of configuration management, such as Chef, Puppet, CFEngine, and Bcfg2.

For a cluster of any size, it can be a challenge to keep all of the machines in sync. Consider what happens if the machine is unavailable when you push out an update. Who ensures it gets the update when it becomes available? This is a big problem and can lead to divergent installations, so even if you use the Hadoop control scripts for managing Hadoop, it may be a good idea to use configuration management tools for maintaining the cluster. These tools are also excellent for doing regular maintenance, such as patching security holes and updating system packages.

Environment Settings

In this section, we consider how to set the variables in *hadoop-env.sh*. There are also analogous configuration files for MapReduce and YARN (but not for HDFS), called *mapred-env.sh* and *yarn-env.sh*, where variables pertaining to those components can be set. Note that the MapReduce and YARN files override the values set in *hadoop-env.sh*.

Java

The location of the Java implementation to use is determined by the `JAVA_HOME` setting in *hadoop-env.sh* or the `JAVA_HOME` shell environment variable, if not set in *hadoop-env.sh*. It's a good idea to set the value in *hadoop-env.sh*, so that it is clearly defined in one place and to ensure that the whole cluster is using the same version of Java.

Memory heap size

By default, Hadoop allocates 1,000 MB (1 GB) of memory to each daemon it runs. This is controlled by the `HADOOP_HEAPSIZE` setting in *hadoop-env.sh*. There are also environment variables to allow you to change the heap size for a single daemon. For example, you can set `YARN_RESOURCEMANAGER_HEAPSIZE` in *yarn-env.sh* to override the heap size for the resource manager.

Surprisingly, there are no corresponding environment variables for HDFS daemons, despite it being very common to give the namenode more heap space. There is another way to set the namenode heap size, however; this is discussed in the following sidebar.

How Much Memory Does a Namenode Need?

A namenode can eat up memory, since a reference to every block of every file is maintained in memory. It's difficult to give a precise formula because memory usage depends on the number of blocks per file, the filename length, and the number of directories in the filesystem; plus, it can change from one Hadoop release to another.

The default of 1,000 MB of namenode memory is normally enough for a few million files, but as a rule of thumb for sizing purposes, you can conservatively allow 1,000 MB per million blocks of storage.

For example, a 200-node cluster with 24 TB of disk space per node, a block size of 128 MB, and a replication factor of 3 has room for about 2 million blocks (or more): $200 \times 24,000,000 \text{ MB} / (128 \text{ MB} \times 3)$. So in this case, setting the namenode memory to 12,000 MB would be a good starting point.

You can increase the namenode's memory without changing the memory allocated to other Hadoop daemons by setting `HADOOP_NAMENODE_OPTS` in *hadoop-env.sh* to include a JVM option for setting the memory size. `HADOOP_NAMENODE_OPTS` allows you to pass extra options to the namenode's JVM. So, for example, if you were using a Sun JVM,

-Xmx2000m would specify that 2,000 MB of memory should be allocated to the namenode.

If you change the namenode's memory allocation, don't forget to do the same for the secondary namenode (using the HADOOP_SECONDARYNAMENODE_OPTS variable), since its memory requirements are comparable to the primary namenode's.

In addition to the memory requirements of the daemons, the node manager allocates containers to applications, so we need to factor these into the total memory footprint of a worker machine; see “[Memory settings in YARN and MapReduce](#)” on page 301.

System logfiles

System logfiles produced by Hadoop are stored in `$HADOOP_HOME/logs` by default. This can be changed using the `HADOOP_LOG_DIR` setting in `hadoop-env.sh`. It's a good idea to change this so that logfiles are kept out of the directory that Hadoop is installed in. Changing this keeps logfiles in one place, even after the installation directory changes due to an upgrade. A common choice is `/var/log/hadoop`, set by including the following line in `hadoop-env.sh`:

```
export HADOOP_LOG_DIR=/var/log/hadoop
```

The log directory will be created if it doesn't already exist. (If it does not exist, confirm that the relevant Unix Hadoop user has permission to create it.) Each Hadoop daemon running on a machine produces two logfiles. The first is the log output written via `log4j`. This file, whose name ends in `.log`, should be the first port of call when diagnosing problems because most application log messages are written here. The standard Hadoop `log4j` configuration uses a daily rolling file appender to rotate logfiles. Old logfiles are never deleted, so you should arrange for them to be periodically deleted or archived, so as to not run out of disk space on the local node.

The second logfile is the combined standard output and standard error log. This logfile, whose name ends in `.out`, usually contains little or no output, since Hadoop uses `log4j` for logging. It is rotated only when the daemon is restarted, and only the last five logs are retained. Old logfiles are suffixed with a number between 1 and 5, with 5 being the oldest file.

Logfile names (of both types) are a combination of the name of the user running the daemon, the daemon name, and the machine hostname. For example, `hadoop-hdfs-datanode-ip-10-45-174-112.log.2014-09-20` is the name of a logfile after it has been rotated. This naming structure makes it possible to archive logs from all machines in the cluster in a single directory, if needed, since the filenames are unique.

The username in the logfile name is actually the default for the `HADOOP_IDENT_STRING` setting in `hadoop-env.sh`. If you wish to give the Hadoop instance a different identity

for the purposes of naming the logfiles, change `HADOOP_IDENT_STRING` to be the identifier you want.

SSH settings

The control scripts allow you to run commands on (remote) worker nodes from the master node using SSH. It can be useful to customize the SSH settings, for various reasons. For example, you may want to reduce the connection timeout (using the `ConnectTimeout` option) so the control scripts don't hang around waiting to see whether a dead node is going to respond. Obviously, this can be taken too far. If the timeout is too low, then busy nodes will be skipped, which is bad.

Another useful SSH setting is `StrictHostKeyChecking`, which can be set to `no` to automatically add new host keys to the known hosts files. The default, `ask`, prompts the user to confirm that the key fingerprint has been verified, which is not a suitable setting in a large cluster environment.⁵

To pass extra options to SSH, define the `HADOOP_SSH_OPTS` environment variable in *hadoop-env.sh*. See the `ssh` and `ssh_config` manual pages for more SSH settings.

Important Hadoop Daemon Properties

Hadoop has a bewildering number of configuration properties. In this section, we address the ones that you need to define (or at least understand why the default is appropriate) for any real-world working cluster. These properties are set in the Hadoop site files: *core-site.xml*, *hdfs-site.xml*, and *yarn-site.xml*. Typical instances of these files are shown in Examples 10-1, 10-2, and 10-3.⁶ You can learn more about the format of Hadoop's configuration files in “The Configuration API” on page 141.

To find the actual configuration of a running daemon, visit the */conf* page on its web server. For example, <http://resource-manager-host:8088/conf> shows the configuration that the resource manager is running with. This page shows the combined site and default configuration files that the daemon is running with, and also shows which file each property was picked up from.

Example 10-1. A typical core-site.xml configuration file

```
<?xml version="1.0"?>
<!-- core-site.xml -->
<configuration>
  <property>
```

5. For more discussion on the security implications of SSH host keys, consult the article “SSH Host Key Protection” by Brian Hatch.

6. Notice that there is no site file for MapReduce shown here. This is because the only MapReduce daemon is the job history server, and the defaults are sufficient.

```

    <name>fs.defaultFS</name>
    <value>hdfs://namenode</value>
  </property>
</configuration>

```

Example 10-2. A typical hdfs-site.xml configuration file

```

<?xml version="1.0"?>
<!-- hdfs-site.xml -->
<configuration>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>/disk1/hdfs/name,/remote/hdfs/name</value>
  </property>

  <property>
    <name>dfs.datanode.data.dir</name>
    <value>/disk1/hdfs/data,/disk2/hdfs/data</value>
  </property>

  <property>
    <name>dfs.namenode.checkpoint.dir</name>
    <value>/disk1/hdfs/namespace,/disk2/hdfs/namespace</value>
  </property>
</configuration>

```

Example 10-3. A typical yarn-site.xml configuration file

```

<?xml version="1.0"?>
<!-- yarn-site.xml -->
<configuration>
  <property>
    <name>yarn.resourcemanager.hostname</name>
    <value>resourcemanager</value>
  </property>

  <property>
    <name>yarn.nodemanager.local-dirs</name>
    <value>/disk1/nm-local-dir,/disk2/nm-local-dir</value>
  </property>

  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce.shuffle</value>
  </property>

  <property>
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>16384</value>
  </property>

  <property>
    <name>yarn.nodemanager.resource.cpu-vcores</name>

```



```
<value>16</value>
</property>
</configuration>
```

HDFS

To run HDFS, you need to designate one machine as a namenode. In this case, the property `fs.defaultFS` is an HDFS filesystem URI whose host is the namenode's host-name or IP address and whose port is the port that the namenode will listen on for RPCs. If no port is specified, the default of 8020 is used.

The `fs.defaultFS` property also doubles as specifying the default filesystem. The default filesystem is used to resolve relative paths, which are handy to use because they save typing (and avoid hardcoding knowledge of a particular namenode's address). For example, with the default filesystem defined in [Example 10-1](#), the relative URI `/a/b` is resolved to `hdfs://namenode/a/b`.



If you are running HDFS, the fact that `fs.defaultFS` is used to specify both the HDFS namenode *and* the default filesystem means HDFS has to be the default filesystem in the server configuration. Bear in mind, however, that it is possible to specify a different filesystem as the default in the client configuration, for convenience.

For example, if you use both HDFS and S3 filesystems, then you have a choice of specifying either as the default in the client configuration, which allows you to refer to the default with a relative URI and the other with an absolute URI.

There are a few other configuration properties you should set for HDFS: those that set the storage directories for the namenode and for datanodes. The property `dfs.name.node.name.dir` specifies a list of directories where the namenode stores persistent filesystem metadata (the edit log and the filesystem image). A copy of each metadata file is stored in each directory for redundancy. It's common to configure `dfs.name.node.name.dir` so that the namenode metadata is written to one or two local disks, as well as a remote disk, such as an NFS-mounted directory. Such a setup guards against failure of a local disk and failure of the entire namenode, since in both cases the files can be recovered and used to start a new namenode. (The secondary namenode takes only periodic checkpoints of the namenode, so it does not provide an up-to-date backup of the namenode.)

You should also set the `dfs.datanode.data.dir` property, which specifies a list of directories for a datanode to store its blocks in. Unlike the namenode, which uses multiple directories for redundancy, a datanode round-robins writes between its storage directories, so for performance you should specify a storage directory for each local disk. Read performance also benefits from having multiple disks for storage, because blocks

will be spread across them and concurrent reads for distinct blocks will be correspondingly spread across disks.



For maximum performance, you should mount storage disks with the `noatime` option. This setting means that last accessed time information is not written on file reads, which gives significant performance gains.

Finally, you should configure where the secondary namenode stores its checkpoints of the filesystem. The `dfs.namenode.checkpoint.dir` property specifies a list of directories where the checkpoints are kept. Like the storage directories for the namenode, which keep redundant copies of the namenode metadata, the checkpointed filesystem image is stored in each checkpoint directory for redundancy.

Table 10-2 summarizes the important configuration properties for HDFS.

Table 10-2. Important HDFS daemon properties

Property name	Type	Default value	Description
<code>fs.defaultFS</code>	URI	<code>file:///</code>	The default filesystem. The URI defines the hostname and port that the namenode's RPC server runs on. The default port is 8020. This property is set in <i>core-site.xml</i> .
<code>dfs.namenode.name.dir</code>	Comma-separated directory names	<code>file://\${hadoop.tmp.dir}/dfs/name</code>	The list of directories where the namenode stores its persistent metadata. The namenode stores a copy of the metadata in each directory in the list.
<code>dfs.datanode.data.dir</code>	Comma-separated directory names	<code>file://\${hadoop.tmp.dir}/dfs/data</code>	A list of directories where the datanode stores blocks. Each block is stored in only one of these directories.
<code>dfs.namenode.checkpoint.dir</code>	Comma-separated directory names	<code>file://\${hadoop.tmp.dir}/dfs/name/secondary</code>	A list of directories where the secondary namenode stores checkpoints. It stores a copy of the checkpoint in each directory in the list.



Note that the storage directories for HDFS are under Hadoop's temporary directory by default (this is configured via the `hadoop.tmp.dir` property, whose default is `/tmp/hadoop-${user.name}`). Therefore, it is critical that these properties are set so that data is not lost by the system when it clears out temporary directories.

YARN

To run YARN, you need to designate one machine as a resource manager. The simplest way to do this is to set the property `yarn.resourcemanager.hostname` to the hostname or IP address of the machine running the resource manager. Many of the resource manager's server addresses are derived from this property. For example, `yarn.resourcemanager.address` takes the form of a host-port pair, and the host defaults to `yarn.resourcemanager.hostname`. In a MapReduce client configuration, this property is used to connect to the resource manager over RPC.

During a MapReduce job, intermediate data and working files are written to temporary local files. Because this data includes the potentially very large output of map tasks, you need to ensure that the `yarn.nodemanager.local-dirs` property, which controls the location of local temporary storage for YARN containers, is configured to use disk partitions that are large enough. The property takes a comma-separated list of directory names, and you should use all available local disks to spread disk I/O (the directories are used in round-robin fashion). Typically, you will use the same disks and partitions (but different directories) for YARN local storage as you use for datanode block storage, as governed by the `dfs.datanode.data.dir` property, which was discussed earlier.

Unlike MapReduce 1, YARN doesn't have tasktrackers to serve map outputs to reduce tasks, so for this function it relies on shuffle handlers, which are long-running auxiliary services running in node managers. Because YARN is a general-purpose service, the MapReduce shuffle handlers need to be enabled explicitly in *yarn-site.xml* by setting the `yarn.nodemanager.aux-services` property to `mapreduce_shuffle`.

Table 10-3 summarizes the important configuration properties for YARN. The resource-related settings are covered in more detail in the next sections.

Table 10-3. Important YARN daemon properties

Property name	Type	Default value	Description
<code>yarn.resourcemanager.hostname</code>	Hostname	<code>0.0.0.0</code>	The hostname of the machine the resource manager runs on. Abbreviated <code>\${y.rm.hostname}</code> below.
<code>yarn.resourcemanager.address</code>	Hostname and port	<code>\${y.rm.hostname}:8032</code>	The hostname and port that the resource manager's RPC server runs on.

Property name	Type	Default value	Description
<code>yarn.nodemanager.local-dirs</code>	Comma-separated directory names	<code>\${hadoop.tmp.dir}/yarn-local-dir</code>	A list of directories where node managers allow containers to store intermediate data. The data is cleared out when the application ends.
<code>yarn.nodemanager.aux-services</code>	Comma-separated service names		A list of auxiliary services run by the node manager. A service is implemented by the class defined by the property <code>yarn.nodemanager.aux-services.service-name.class</code> . By default, no auxiliary services are specified.
<code>yarn.nodemanager.resource.memory-mb</code>	int	8192	The amount of physical memory (in MB) that may be allocated to containers being run by the node manager.
<code>yarn.nodemanager.vmem-pmem-ratio</code>	float	2.1	The ratio of virtual to physical memory for containers. Virtual memory usage may exceed the allocation by this amount.
<code>yarn.nodemanager.resource.cpu-vcores</code>	int	8	The number of CPU cores that may be allocated to containers being run by the node manager.

Memory settings in YARN and MapReduce

YARN treats memory in a more fine-grained manner than the slot-based model used in MapReduce 1. Rather than specifying a fixed maximum number of map and reduce slots that may run on a node at once, YARN allows applications to request an arbitrary amount of memory (within limits) for a task. In the YARN model, node managers allocate memory from a pool, so the number of tasks that are running on a particular node depends on the sum of their memory requirements, and not simply on a fixed number of slots.

The calculation for how much memory to dedicate to a node manager for running containers depends on the amount of physical memory on the machine. Each Hadoop daemon uses 1,000 MB, so for a datanode and a node manager, the total is 2,000 MB. Set aside enough for other processes that are running on the machine, and the remainder can be dedicated to the node manager's containers by setting the configuration property `yarn.nodemanager.resource.memory-mb` to the total allocation in MB. (The default is 8,192 MB, which is normally too low for most setups.)

The next step is to determine how to set memory options for individual jobs. There are two main controls: one for the size of the container allocated by YARN, and another for the heap size of the Java process run in the container.



The memory controls for MapReduce are all set by the client in the job configuration. The YARN settings are cluster settings and cannot be modified by the client.

Container sizes are determined by `mapreduce.map.memory.mb` and `mapreduce.reduce.memory.mb`; both default to 1,024 MB. These settings are used by the application master when negotiating for resources in the cluster, and also by the node manager, which runs and monitors the task containers. The heap size of the Java process is set by `mapred.child.java.opts`, and defaults to 200 MB. You can also set the Java options separately for map and reduce tasks (see [Table 10-4](#)).

Table 10-4. MapReduce job memory properties (set by the client)

Property name	Type	Default value	Description
<code>mapreduce.map.memory.mb</code>	<code>int</code>	1024	The amount of memory for map containers.
<code>mapreduce.reduce.memory.mb</code>	<code>int</code>	1024	The amount of memory for reduce containers.
<code>mapred.child.java.opts</code>	<code>String</code>	<code>-Xmx200m</code>	The JVM options used to launch the container process that runs map and reduce tasks. In addition to memory settings, this property can include JVM properties for debugging, for example.
<code>mapreduce.map.java.opts</code>	<code>String</code>	<code>-Xmx200m</code>	The JVM options used for the child process that runs map tasks.
<code>mapreduce.reduce.java.opts</code>	<code>String</code>	<code>-Xmx200m</code>	The JVM options used for the child process that runs reduce tasks.

For example, suppose `mapred.child.java.opts` is set to `-Xmx800m` and `mapreduce.map.memory.mb` is left at its default value of 1,024 MB. When a map task is run, the node manager will allocate a 1,024 MB container (decreasing the size of its pool by that amount for the duration of the task) and will launch the task JVM configured with an 800 MB maximum heap size. Note that the JVM process will have a larger memory footprint than the heap size, and the overhead will depend on such things as the native libraries that are in use, the size of the permanent generation space, and so on. The important thing is that the physical memory used by the JVM process, including any processes that it spawns, such as Streaming processes, does not exceed its allocation (1,024 MB). If a container uses more memory than it has been allocated, then it may be terminated by the node manager and marked as failed.

YARN schedulers impose a minimum or maximum on memory allocations. The default minimum is 1,024 MB (set by `yarn.scheduler.minimum-allocation-mb`), and the default maximum is 8,192 MB (set by `yarn.scheduler.maximum-allocation-mb`).

There are also virtual memory constraints that a container must meet. If a container's virtual memory usage exceeds a given multiple of the allocated physical memory, the node manager may terminate the process. The multiple is expressed by the `yarn.nodemanager.vmem-pmem-ratio` property, which defaults to 2.1. In the example used earlier, the virtual memory threshold above which the task may be terminated is 2,150 MB, which is $2.1 \times 1,024$ MB.

When configuring memory parameters it's very useful to be able to monitor a task's actual memory usage during a job run, and this is possible via MapReduce task counters. The counters `PHYSICAL_MEMORY_BYTES`, `VIRTUAL_MEMORY_BYTES`, and `COMMITTED_HEAP_BYTES` (described in [Table 9-2](#)) provide snapshot values of memory usage and are therefore suitable for observation during the course of a task attempt.

Hadoop also provides settings to control how much memory is used for MapReduce operations. These can be set on a per-job basis and are covered in [“Shuffle and Sort” on page 197](#).

CPU settings in YARN and MapReduce

In addition to memory, YARN treats CPU usage as a managed resource, and applications can request the number of cores they need. The number of cores that a node manager can allocate to containers is controlled by the `yarn.nodemanager.resource.cpu-vcores` property. It should be set to the total number of cores on the machine, minus a core for each daemon process running on the machine (datanode, node manager, and any other long-running processes).

MapReduce jobs can control the number of cores allocated to map and reduce containers by setting `mapreduce.map.cpu.vcores` and `mapreduce.reduce.cpu.vcores`. Both default to 1, an appropriate setting for normal single-threaded MapReduce tasks, which can only saturate a single core.



While the number of cores is tracked during scheduling (so a container won't be allocated on a machine where there are no spare cores, for example), the node manager will not, by default, limit actual CPU usage of running containers. This means that a container can abuse its allocation by using more CPU than it was given, possibly starving other containers running on the same host. YARN has support for enforcing CPU limits using Linux cgroups. The node manager's container executor class (`yarn.nodemanager.container-executor.class`) must be set to use the `LinuxContainerExecutor` class, which in turn must be configured to use cgroups (see the properties under `yarn.nodemanager.linux-container-executor`).

Hadoop Daemon Addresses and Ports

Hadoop daemons generally run both an RPC server for communication between daemons (Table 10-5) and an HTTP server to provide web pages for human consumption (Table 10-6). Each server is configured by setting the network address and port number to listen on. A port number of 0 instructs the server to start on a free port, but this is generally discouraged because it is incompatible with setting cluster-wide firewall policies.

In general, the properties for setting a server's RPC and HTTP addresses serve double duty: they determine the network interface that the server will bind to, and they are used by clients or other machines in the cluster to connect to the server. For example, node managers use the `yarn.resourcemanager.resource-tracker.address` property to find the address of their resource manager.

It is often desirable for servers to bind to multiple network interfaces, but setting the network address to `0.0.0.0`, which works for the server, breaks the second case, since the address is not resolvable by clients or other machines in the cluster. One solution is to have separate configurations for clients and servers, but a better way is to set the bind host for the server. By setting `yarn.resourcemanager.hostname` to the (externally resolvable) hostname or IP address and `yarn.resourcemanager.bind-host` to `0.0.0.0`, you ensure that the resource manager will bind to all addresses on the machine, while at the same time providing a resolvable address for node managers and clients.

In addition to an RPC server, datanodes run a TCP/IP server for block transfers. The server address and port are set by the `dfs.datanode.address` property, which has a default value of `0.0.0.0:50010`.

Table 10-5. RPC server properties

Property name	Default value	Description
<code>fs.defaultFS</code>	<code>file:///</code>	When set to an HDFS URI, this property determines the namenode's RPC server address and port. The default port is 8020 if not specified.
<code>dfs.namenode.rpc-bind-host</code>		The address the namenode's RPC server will bind to. If not set (the default), the bind address is determined by <code>fs.defaultFS</code> . It can be set to <code>0.0.0.0</code> to make the namenode listen on all interfaces.
<code>dfs.datanode.ipc.address</code>	<code>0.0.0.0:50020</code>	The datanode's RPC server address and port.
<code>mapreduce.jobhistory.address</code>	<code>0.0.0.0:10020</code>	The job history server's RPC server address and port. This is used by the client (typically outside the cluster) to query job history.
<code>mapreduce.jobhistory.bind-host</code>		The address the job history server's RPC and HTTP servers will bind to.
<code>yarn.resourcemanager.hostname</code>	<code>0.0.0.0</code>	The hostname of the machine the resource manager runs on. Abbreviated <code>\${y.rm.hostname}</code> below.
<code>yarn.resourcemanager.bind-host</code>		The address the resource manager's RPC and HTTP servers will bind to.
<code>yarn.resourcemanager.address</code>	<code>\${y.rm.hostname}:8032</code>	The resource manager's RPC server address and port. This is used by the client (typically outside the cluster) to communicate with the resource manager.
<code>yarn.resourcemanager.admin.address</code>	<code>\${y.rm.hostname}:8033</code>	The resource manager's admin RPC server address and port. This is used by the admin client (invoked with <code>yarn rmadmin</code> , typically run outside the cluster) to communicate with the resource manager.
<code>yarn.resourcemanager.scheduler.address</code>	<code>\${y.rm.hostname}:8030</code>	The resource manager scheduler's RPC server address and port. This is used by (in-cluster) application masters to communicate with the resource manager.
<code>yarn.resourcemanager.resource-tracker.address</code>	<code>\${y.rm.hostname}:8031</code>	The resource manager resource tracker's RPC server address and port. This is used by (in-cluster) node managers to communicate with the resource manager.
<code>yarn.nodemanager.hostname</code>	<code>0.0.0.0</code>	The hostname of the machine the node manager runs on. Abbreviated <code>\${y.nm.hostname}</code> below.
<code>yarn.nodemanager.bind-host</code>		The address the node manager's RPC and HTTP servers will bind to.

Property name	Default value	Description
yarn.nodemanager.address	<code>\${y.nm.host name}:0</code>	The node manager's RPC server address and port. This is used by (in-cluster) application masters to communicate with node managers.
yarn.nodemanager.localizer.address	<code>\${y.nm.host name}:8040</code>	The node manager localizer's RPC server address and port.

Table 10-6. HTTP server properties

Property name	Default value	Description
dfs.namenode.http-address	<code>0.0.0.0:50070</code>	The namenode's HTTP server address and port.
dfs.namenode.http-bind-host		The address the namenode's HTTP server will bind to.
dfs.namenode.secondary.http-address	<code>0.0.0.0:50090</code>	The secondary namenode's HTTP server address and port.
dfs.datanode.http.address	<code>0.0.0.0:50075</code>	The datanode's HTTP server address and port. (Note that the property name is inconsistent with the ones for the namenode.)
mapreduce.jobhistory.webapp.address	<code>0.0.0.0:19888</code>	The MapReduce job history server's address and port. This property is set in <i>mapred-site.xml</i> .
mapreduce.shuffle.port	<code>13562</code>	The shuffle handler's HTTP port number. This is used for serving map outputs, and is not a user-accessible web UI. This property is set in <i>mapred-site.xml</i> .
yarn.resourcemanager.webapp.address	<code>\${y.rm.host name}:8088</code>	The resource manager's HTTP server address and port.
yarn.nodemanager.webapp.address	<code>\${y.nm.host name}:8042</code>	The node manager's HTTP server address and port.
yarn.web-proxy.address		The web app proxy server's HTTP server address and port. If not set (the default), then the web app proxy server will run in the resource manager process.

There is also a setting for controlling which network interfaces the datanodes use as their IP addresses (for HTTP and RPC servers). The relevant property is `dfs.datanode.dns.interface`, which is set to `default` to use the default network interface. You can set this explicitly to report the address of a particular interface (`eth0`, for example).

Other Hadoop Properties

This section discusses some other properties that you might consider setting.

Cluster membership

To aid in the addition and removal of nodes in the future, you can specify a file containing a list of authorized machines that may join the cluster as datanodes or node managers. The file is specified using the `dfs.hosts` and `yarn.resourcemanager.nodes.include-path` properties (for datanodes and node managers, respectively), and the corresponding `dfs.hosts.exclude` and `yarn.resourcemanager.nodes.exclude-path` properties specify the files used for de-commissioning. See [“Commissioning and Decommissioning Nodes” on page 334](#) for further discussion.

Buffer size

Hadoop uses a buffer size of 4 KB (4,096 bytes) for its I/O operations. This is a conservative setting, and with modern hardware and operating systems, you will likely see performance benefits by increasing it; 128 KB (131,072 bytes) is a common choice. Set the value in bytes using the `io.file.buffer.size` property in *core-site.xml*.

HDFS block size

The HDFS block size is 128 MB by default, but many clusters use more (e.g., 256 MB, which is 268,435,456 bytes) to ease memory pressure on the namenode and to give mappers more data to work on. Use the `dfs.blocksize` property in *hdfs-site.xml* to specify the size in bytes.

Reserved storage space

By default, datanodes will try to use all of the space available in their storage directories. If you want to reserve some space on the storage volumes for non-HDFS use, you can set `dfs.datanode.du.reserved` to the amount, in bytes, of space to reserve.

Trash

Hadoop filesystems have a trash facility, in which deleted files are not actually deleted but rather are moved to a trash folder, where they remain for a minimum period before being permanently deleted by the system. The minimum period in minutes that a file will remain in the trash is set using the `fs.trash.interval` configuration property in *core-site.xml*. By default, the trash interval is zero, which disables trash.

Like in many operating systems, Hadoop’s trash facility is a user-level feature, meaning that only files that are deleted using the filesystem shell are put in the trash. Files deleted programmatically are deleted immediately. It is possible to use the trash

programmatically, however, by constructing a `Trash` instance, then calling its `moveToTrash()` method with the `Path` of the file intended for deletion. The method returns a value indicating success; a value of `false` means either that trash is not enabled or that the file is already in the trash.

When trash is enabled, users each have their own trash directories called *.Trash* in their home directories. File recovery is simple: you look for the file in a subdirectory of *.Trash* and move it out of the trash subtree.

HDFS will automatically delete files in trash folders, but other filesystems will not, so you have to arrange for this to be done periodically. You can *expunge* the trash, which will delete files that have been in the trash longer than their minimum period, using the filesystem shell:

```
% hadoop fs -expunge
```

The `Trash` class exposes an `expunge()` method that has the same effect.

Job scheduler

Particularly in a multiuser setting, consider updating the job scheduler queue configuration to reflect your organizational needs. For example, you can set up a queue for each group using the cluster. See [“Scheduling in YARN” on page 85](#).

Reduce slow start

By default, schedulers wait until 5% of the map tasks in a job have completed before scheduling reduce tasks for the same job. For large jobs, this can cause problems with cluster utilization, since they take up reduce containers while waiting for the map tasks to complete. Setting `mapreduce.job.reduce.slowstart.completedmaps` to a higher value, such as 0.80 (80%), can help improve throughput.

Short-circuit local reads

When reading a file from HDFS, the client contacts the datanode and the data is sent to the client via a TCP connection. If the block being read is on the same node as the client, then it is more efficient for the client to bypass the network and read the block data directly from the disk. This is termed a *short-circuit local read*, and can make applications like HBase perform better.

You can enable short-circuit local reads by setting `dfs.client.read.shortcircuit` to `true`. Short-circuit local reads are implemented using Unix domain sockets, which use a local path for client-datanode communication. The path is set using the property `dfs.domain.socket.path`, and must be a path that only the datanode user (typically `hdfs`) or root can create, such as `/var/run/hadoop-hdfs/dn_socket`.

Security

Early versions of Hadoop assumed that HDFS and MapReduce clusters would be used by a group of cooperating users within a secure environment. The measures for restricting access were designed to prevent accidental data loss, rather than to prevent unauthorized access to data. For example, the file permissions system in HDFS prevents one user from accidentally wiping out the whole filesystem because of a bug in a program, or by mistakenly typing `hadoop fs -rmr /`, but it doesn't prevent a malicious user from assuming root's identity to access or delete any data in the cluster.

In security parlance, what was missing was a secure *authentication* mechanism to assure Hadoop that the user seeking to perform an operation on the cluster is who he claims to be and therefore can be trusted. HDFS file permissions provide only a mechanism for *authorization*, which controls what a particular user can do to a particular file. For example, a file may be readable only by a certain group of users, so anyone not in that group is not authorized to read it. However, authorization is not enough by itself, because the system is still open to abuse via spoofing by a malicious user who can gain network access to the cluster.

It's common to restrict access to data that contains personally identifiable information (such as an end user's full name or IP address) to a small set of users (of the cluster) within the organization who are authorized to access such information. Less sensitive (or anonymized) data may be made available to a larger set of users. It is convenient to host a mix of datasets with different security levels on the same cluster (not least because it means the datasets with lower security levels can be shared). However, to meet regulatory requirements for data protection, secure authentication must be in place for shared clusters.

This is the situation that Yahoo! faced in 2009, which led a team of engineers there to implement secure authentication for Hadoop. In their design, Hadoop itself does not manage user credentials; instead, it relies on Kerberos, a mature open-source network authentication protocol, to authenticate the user. However, Kerberos doesn't manage permissions. Kerberos says that a user is who she says she is; it's Hadoop's job to determine whether that user has permission to perform a given action.

There's a lot to security in Hadoop, and this section only covers the highlights. For more, readers are referred to *Hadoop Security* by Ben Spivey and Joey Echeverria (O'Reilly, 2014).

Kerberos and Hadoop

At a high level, there are three steps that a client must take to access a service when using Kerberos, each of which involves a message exchange with a server:

1. *Authentication*. The client authenticates itself to the Authentication Server and receives a timestamped Ticket-Granting Ticket (TGT).
2. *Authorization*. The client uses the TGT to request a service ticket from the Ticket-Granting Server.
3. *Service request*. The client uses the service ticket to authenticate itself to the server that is providing the service the client is using. In the case of Hadoop, this might be the namenode or the resource manager.

Together, the Authentication Server and the Ticket Granting Server form the *Key Distribution Center* (KDC). The process is shown graphically in [Figure 10-2](#).

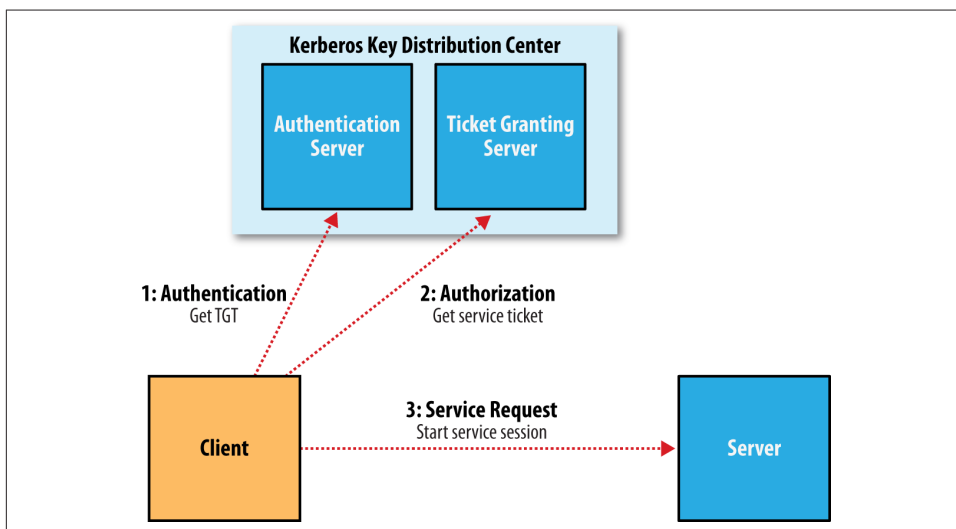


Figure 10-2. The three-step Kerberos ticket exchange protocol

The authorization and service request steps are not user-level actions; the client performs these steps on the user's behalf. The authentication step, however, is normally carried out explicitly by the user using the `kinit` command, which will prompt for a password. However, this doesn't mean you need to enter your password every time you run a job or access HDFS, since TGTs last for 10 hours by default (and can be renewed for up to a week). It's common to automate authentication at operating system login time, thereby providing *single sign-on* to Hadoop.

In cases where you don't want to be prompted for a password (for running an unattended MapReduce job, for example), you can create a Kerberos *keytab* file using the `ktutil` command. A keytab is a file that stores passwords and may be supplied to `kinit` with the `-t` option.

An example

Let's look at an example of the process in action. The first step is to enable Kerberos authentication by setting the `hadoop.security.authentication` property in *core-site.xml* to `kerberos`.⁷ The default setting is `simple`, which signifies that the old backward-compatible (but insecure) behavior of using the operating system username to determine identity should be employed.

We also need to enable service-level authorization by setting `hadoop.security.authorization` to `true` in the same file. You may configure access control lists (ACLs) in the *hadoop-policy.xml* configuration file to control which users and groups have permission to connect to each Hadoop service. Services are defined at the protocol level, so there are ones for MapReduce job submission, namenode communication, and so on. By default, all ACLs are set to `*`, which means that all users have permission to access each service; however, on a real cluster you should lock the ACLs down to only those users and groups that should have access.

The format for an ACL is a comma-separated list of usernames, followed by whitespace, followed by a comma-separated list of group names. For example, the ACL `preston,howard directors,inventors` would authorize access to users named `preston` or `howard`, or in groups `directors` or `inventors`.

With Kerberos authentication turned on, let's see what happens when we try to copy a local file to HDFS:

```
% hadoop fs -put quangle.txt .
10/07/03 15:44:58 WARN ipc.Client: Exception encountered while connecting to the
server: javax.security.sasl.SaslException: GSS initiate failed [Caused by
GSSEException: No valid credentials provided (Mechanism level: Failed to find
any Kerberos tgt)]
Bad connection to FS. command aborted. exception: Call to localhost/
127.0.0.1:8020 failed on local exception: java.io.IOException:
javax.security.sasl.SaslException: GSS initiate failed [Caused by GSSEException:
No valid credentials provided
(Mechanism level: Failed to find any Kerberos tgt)]
```

The operation fails because we don't have a Kerberos ticket. We can get one by authenticating to the KDC, using `kinit`:

```
% kinit
Password for hadoop-user@LOCALDOMAIN: password
% hadoop fs -put quangle.txt .
% hadoop fs -stat %n quangle.txt
quangle.txt
```

7. To use Kerberos authentication with Hadoop, you need to install, configure, and run a KDC (Hadoop does not come with one). Your organization may already have a KDC you can use (an Active Directory installation, for example); if not, you can set up an MIT Kerberos 5 KDC.

And we see that the file is successfully written to HDFS. Notice that even though we carried out two filesystem commands, we only needed to call `kinit` once, since the Kerberos ticket is valid for 10 hours (use the `klist` command to see the expiry time of your tickets and `kdestroy` to invalidate your tickets). After we get a ticket, everything works just as it normally would.

Delegation Tokens

In a distributed system such as HDFS or MapReduce, there are many client-server interactions, each of which must be authenticated. For example, an HDFS read operation will involve multiple calls to the namenode and calls to one or more datanodes. Instead of using the three-step Kerberos ticket exchange protocol to authenticate each call, which would present a high load on the KDC on a busy cluster, Hadoop uses *delegation tokens* to allow later authenticated access without having to contact the KDC again. Delegation tokens are created and used transparently by Hadoop on behalf of users, so there's no action you need to take as a user beyond using `kinit` to sign in, but it's useful to have a basic idea of how they are used.

A delegation token is generated by the server (the namenode, in this case) and can be thought of as a shared secret between the client and the server. On the first RPC call to the namenode, the client has no delegation token, so it uses Kerberos to authenticate. As a part of the response, it gets a delegation token from the namenode. In subsequent calls it presents the delegation token, which the namenode can verify (since it generated it using a secret key), and hence the client is authenticated to the server.

When it wants to perform operations on HDFS blocks, the client uses a special kind of delegation token, called a *block access token*, that the namenode passes to the client in response to a metadata request. The client uses the block access token to authenticate itself to datanodes. This is possible only because the namenode shares its secret key used to generate the block access token with datanodes (sending it in heartbeat messages), so that they can verify block access tokens. Thus, an HDFS block may be accessed only by a client with a valid block access token from a namenode. This closes the security hole in unsecured Hadoop where only the block ID was needed to gain access to a block. This property is enabled by setting `dfs.block.access.token.enable` to `true`.

In MapReduce, job resources and metadata (such as JAR files, input splits, and configuration files) are shared in HDFS for the application master to access, and user code runs on the node managers and accesses files on HDFS (the process is explained in [“Anatomy of a MapReduce Job Run” on page 185](#)). Delegation tokens are used by these components to access HDFS during the course of the job. When the job has finished, the delegation tokens are invalidated.

Delegation tokens are automatically obtained for the default HDFS instance, but if your job needs to access other HDFS clusters, you can load the delegation tokens for these

by setting the `mapreduce.job.hdfs-servers` job property to a comma-separated list of HDFS URIs.

Other Security Enhancements

Security has been tightened throughout the Hadoop stack to protect against unauthorized access to resources. The more notable features are listed here:

- Tasks can be run using the operating system account for the user who submitted the job, rather than the user running the node manager. This means that the operating system is used to isolate running tasks, so they can't send signals to each other (to kill another user's tasks, for example) and so local information, such as task data, is kept private via local filesystem permissions.

This feature is enabled by setting `yarn.nodemanager.container-executor.class` to `org.apache.hadoop.yarn.server.nodemanager.LinuxContainerExecutor`.⁸ In addition, administrators need to ensure that each user is given an account on every node in the cluster (typically using LDAP).

- When tasks are run as the user who submitted the job, the distributed cache (see [“Distributed Cache” on page 274](#)) is secure. Files that are world-readable are put in a shared cache (the insecure default); otherwise, they go in a private cache, readable only by the owner.
- Users can view and modify only their own jobs, not others. This is enabled by setting `mapreduce.cluster.acls.enabled` to `true`. There are two job configuration properties, `mapreduce.job.acl-view-job` and `mapreduce.job.acl-modify-job`, which may be set to a comma-separated list of users to control who may view or modify a particular job.
- The shuffle is secure, preventing a malicious user from requesting another user's map outputs.
- When appropriately configured, it's no longer possible for a malicious user to run a rogue secondary namenode, datanode, or node manager that can join the cluster and potentially compromise data stored in the cluster. This is enforced by requiring daemons to authenticate with the master node they are connecting to.

To enable this feature, you first need to configure Hadoop to use a keytab previously generated with the `ktutil` command. For a datanode, for example, you would set the `dfs.datanode.keytab.file` property to the keytab filename and `dfs.datanode.kerberos.principal` to the username to use for the datanode. Finally, the ACL for the `DataNodeProtocol` (which is used by datanodes to communicate with

8. `LinuxTaskController` uses a setuid executable called *container-executor*, found in the *bin* directory. You should ensure that this binary is owned by root and has the setuid bit set (with `chmod +s`).

the namenode) must be set in *hadoop-policy.xml*, by restricting `security.data.node.protocol.acl` to the datanode's username.

- A datanode may be run on a privileged port (one lower than 1024), so a client may be reasonably sure that it was started securely.
- A task may communicate only with its parent application master, thus preventing an attacker from obtaining MapReduce data from another user's job.
- Various parts of Hadoop can be configured to encrypt network data, including RPC (`hadoop.rpc.protection`), HDFS block transfers (`dfs.encrypt.data.transfer`), the MapReduce shuffle (`mapreduce.shuffle.ssl.enabled`), and the web UIs (`hadoopop.ssl.enabled`). Work is ongoing to encrypt data “at rest,” too, so that HDFS blocks can be stored in encrypted form, for example.

Benchmarking a Hadoop Cluster

Is the cluster set up correctly? The best way to answer this question is empirically: run some jobs and confirm that you get the expected results. Benchmarks make good tests because you also get numbers that you can compare with other clusters as a sanity check on whether your new cluster is performing roughly as expected. And you can tune a cluster using benchmark results to squeeze the best performance out of it. This is often done with monitoring systems in place (see “[Monitoring](#)” on page 330), so you can see how resources are being used across the cluster.

To get the best results, you should run benchmarks on a cluster that is not being used by others. In practice, this will be just before it is put into service and users start relying on it. Once users have scheduled periodic jobs on a cluster, it is generally impossible to find a time when the cluster is not being used (unless you arrange downtime with users), so you should run benchmarks to your satisfaction before this happens.

Experience has shown that most hardware failures for new systems are hard drive failures. By running I/O-intensive benchmarks—such as the ones described next—you can “burn in” the cluster before it goes live.

Hadoop Benchmarks

Hadoop comes with several benchmarks that you can run very easily with minimal setup cost. Benchmarks are packaged in the tests JAR file, and you can get a list of them, with descriptions, by invoking the JAR file with no arguments:

```
% hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-*-tests.jar
```

Most of the benchmarks show usage instructions when invoked with no arguments. For example:

```
% hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-tests.jar \
TestDFSIO
TestDFSIO.1.7
Missing arguments.
Usage: TestDFSIO [genericOptions] -read [-random | -backward |
-skip [-skipSize Size]] | -write | -append | -clean [-compression codecClassName]
[-nrFiles N] [-size Size[B|KB|MB|GB|TB]] [-resFile resultFileName]
[-bufferSize Bytes] [-rootDir]
```

Benchmarking MapReduce with TeraSort

Hadoop comes with a MapReduce program called *TeraSort* that does a total sort of its input.⁹ It is very useful for benchmarking HDFS and MapReduce together, as the full input dataset is transferred through the shuffle. The three steps are: generate some random data, perform the sort, then validate the results.

First, we generate some random data using *teragen* (found in the examples JAR file, not the tests one). It runs a map-only job that generates a specified number of rows of binary data. Each row is 100 bytes long, so to generate one terabyte of data using 1,000 maps, run the following (10t is short for 10 trillion):

```
% hadoop jar \
$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples.jar \
teragen -Dmapreduce.job.maps=1000 10t random-data
```

Next, run *terasort*:

```
% hadoop jar \
$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples.jar \
terasort random-data sorted-data
```

The overall execution time of the sort is the metric we are interested in, but it's instructive to watch the job's progress via the web UI (<http://resource-manager-host:8088/>), where you can get a feel for how long each phase of the job takes. Adjusting the parameters mentioned in “[Tuning a Job](#)” on page 175 is a useful exercise, too.

As a final sanity check, we validate that the data in *sorted-data* is, in fact, correctly sorted:

```
% hadoop jar \
$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples.jar \
teravalidate sorted-data report
```

This command runs a short MapReduce job that performs a series of checks on the sorted data to check whether the sort is accurate. Any errors can be found in the *report/part-r-00000* output file.

9. In 2008, TeraSort was used to break the world record for sorting 1 TB of data; see “[A Brief History of Apache Hadoop](#)” on page 12.

Other benchmarks

There are many more Hadoop benchmarks, but the following are widely used:

- *TestDFSIO* tests the I/O performance of HDFS. It does this by using a MapReduce job as a convenient way to read or write files in parallel.
- *MRBench* (invoked with `mrbench`) runs a small job a number of times. It acts as a good counterpoint to TeraSort, as it checks whether small job runs are responsive.
- *NNBench* (invoked with `nnbench`) is useful for load-testing namenode hardware.
- *Gridmix* is a suite of benchmarks designed to model a realistic cluster workload by mimicking a variety of data-access patterns seen in practice. See the documentation in the distribution for how to run Gridmix.
- *SWIM*, or the **Statistical Workload Injector for MapReduce**, is a repository of real-life MapReduce workloads that you can use to generate representative test workloads for your system.
- *TPC_x-HS* is a standardized benchmark based on TeraSort from the Transaction Processing Performance Council.

User Jobs

For tuning, it is best to include a few jobs that are representative of the jobs that your users run, so your cluster is tuned for these and not just for the standard benchmarks. If this is your first Hadoop cluster and you don't have any user jobs yet, then either Gridmix or SWIM is a good substitute.

When running your own jobs as benchmarks, you should select a dataset for your user jobs and use it each time you run the benchmarks to allow comparisons between runs. When you set up a new cluster or upgrade a cluster, you will be able to use the same dataset to compare the performance with previous runs.

Administering Hadoop

The previous chapter was devoted to setting up a Hadoop cluster. In this chapter, we look at the procedures to keep a cluster running smoothly.

HDFS

Persistent Data Structures

As an administrator, it is invaluable to have a basic understanding of how the components of HDFS—the namenode, the secondary namenode, and the datanodes—organize their persistent data on disk. Knowing which files are which can help you diagnose problems or spot that something is awry.

Namenode directory structure

A running namenode has a directory structure like this:

```

${dfs.namenode.name.dir}/
├── current
│   ├── VERSION
│   ├── edits_00000000000000000001-00000000000000000019
│   ├── edits_inprogress_0000000000000000000020
│   ├── fsimage_00000000000000000000
│   ├── fsimage_00000000000000000000.md5
│   ├── fsimage_0000000000000000000019
│   ├── fsimage_0000000000000000000019.md5
│   └── seen_txid
└── in_use.lock
  
```

Recall from [Chapter 10](#) that the `dfs.namenode.name.dir` property is a list of directories, with the same contents mirrored in each directory. This mechanism provides resilience, particularly if one of the directories is an NFS mount, as is recommended.

The *VERSION* file is a Java properties file that contains information about the version of HDFS that is running. Here are the contents of a typical file:

```
#Mon Sep 29 09:54:36 BST 2014
namespaceID=1342387246
clusterID=CID-01b5c398-959c-4ea8-aae6-1e0d9bd8b142
cTime=0
storageType=NAME_NODE
blockpoolID=BP-526805057-127.0.0.1-1411980876842
layoutVersion=-57
```

The *layoutVersion* is a negative integer that defines the version of HDFS's persistent data structures. This version number has no relation to the release number of the Hadoop distribution. Whenever the layout changes, the version number is decremented (for example, the version after *-57* is *-58*). When this happens, HDFS needs to be upgraded, since a newer namenode (or datanode) will not operate if its storage layout is an older version. Upgrading HDFS is covered in [“Upgrades” on page 337](#).

The *namespaceID* is a unique identifier for the filesystem namespace, which is created when the namenode is first formatted. The *clusterID* is a unique identifier for the HDFS cluster as a whole; this is important for HDFS federation (see [“HDFS Federation” on page 48](#)), where a cluster is made up of multiple namespaces and each namespace is managed by one namenode. The *blockpoolID* is a unique identifier for the block pool containing all the files in the namespace managed by this namenode.

The *cTime* property marks the creation time of the namenode's storage. For newly formatted storage, the value is always zero, but it is updated to a timestamp whenever the filesystem is upgraded.

The *storageType* indicates that this storage directory contains data structures for a namenode.

The *in_use.lock* file is a lock file that the namenode uses to lock the storage directory. This prevents another namenode instance from running at the same time with (and possibly corrupting) the same storage directory.

The other files in the namenode's storage directory are the *edits* and *fsimage* files, and *seen_txid*. To understand what these files are for, we need to dig into the workings of the namenode a little more.

The filesystem image and edit log

When a filesystem client performs a write operation (such as creating or moving a file), the transaction is first recorded in the edit log. The namenode also has an in-memory representation of the filesystem metadata, which it updates after the edit log has been modified. The in-memory metadata is used to serve read requests.

Conceptually the edit log is a single entity, but it is represented as a number of files on disk. Each file is called a *segment*, and has the prefix *edits* and a suffix that indicates the transaction IDs contained in it. Only one file is open for writes at any one time (*edits_inprogress_000000000000000020* in the preceding example), and it is flushed and synced after every transaction before a success code is returned to the client. For namenodes that write to multiple directories, the write must be flushed and synced to every copy before returning successfully. This ensures that no transaction is lost due to machine failure.

Each *fsimage* file is a complete persistent checkpoint of the filesystem metadata. (The suffix indicates the last transaction in the image.) However, it is *not* updated for every filesystem write operation, because writing out the *fsimage* file, which can grow to be gigabytes in size, would be very slow. This does not compromise resilience because if the namenode fails, then the latest state of its metadata can be reconstructed by loading the latest *fsimage* from disk into memory, and then applying each of the transactions from the relevant point onward in the edit log. In fact, this is precisely what the namenode does when it starts up (see “[Safe Mode](#)” on page 322).



Each *fsimage* file contains a serialized form of all the directory and file inodes in the filesystem. Each inode is an internal representation of a file or directory’s metadata and contains such information as the file’s replication level, modification and access times, access permissions, block size, and the blocks the file is made up of. For directories, the modification time, permissions, and quota metadata are stored.

An *fsimage* file does *not* record the datanodes on which the blocks are stored. Instead, the namenode keeps this mapping in memory, which it constructs by asking the datanodes for their block lists when they join the cluster and periodically afterward to ensure the namenode’s block mapping is up to date.

As described, the edit log would grow without bound (even if it was spread across several physical *edits* files). Though this state of affairs would have no impact on the system while the namenode is running, if the namenode were restarted, it would take a long time to apply each of the transactions in its (very long) edit log. During this time, the filesystem would be offline, which is generally undesirable.

The solution is to run the secondary namenode, whose purpose is to produce checkpoints of the primary's in-memory filesystem metadata.¹ The checkpointing process proceeds as follows (and is shown schematically in [Figure 11-1](#) for the edit log and image files shown earlier):

1. The secondary asks the primary to roll its in-progress *edits* file, so new edits go to a new file. The primary also updates the *seen_txid* file in all its storage directories.
2. The secondary retrieves the latest *fsimage* and *edits* files from the primary (using HTTP GET).
3. The secondary loads *fsimage* into memory, applies each transaction from *edits*, then creates a new merged *fsimage* file.
4. The secondary sends the new *fsimage* back to the primary (using HTTP PUT), and the primary saves it as a temporary *.ckpt* file.
5. The primary renames the temporary *fsimage* file to make it available.

At the end of the process, the primary has an up-to-date *fsimage* file and a short in-progress *edits* file (it is not necessarily empty, as it may have received some edits while the checkpoint was being taken). It is possible for an administrator to run this process manually while the namenode is in safe mode, using the `hdfs dfsadmin -saveNamespace` command.

This procedure makes it clear why the secondary has similar memory requirements to the primary (since it loads the *fsimage* into memory), which is the reason that the secondary needs a dedicated machine on large clusters.

The schedule for checkpointing is controlled by two configuration parameters. The secondary namenode checkpoints every hour (`dfs.namenode.checkpoint.period` in seconds), or sooner if the edit log has reached one million transactions since the last checkpoint (`dfs.namenode.checkpoint.txns`), which it checks every minute (`dfs.namenode.checkpoint.check.period` in seconds).

1. It is actually possible to start a namenode with the `-checkpoint` option so that it runs the checkpointing process against another (primary) namenode. This is functionally equivalent to running a secondary namenode, but at the time of this writing offers no advantages over the secondary namenode (and indeed, the secondary namenode is the most tried and tested option). When running in a high-availability environment (see “[HDFS High Availability](#)” on page 48), the standby node performs checkpointing.

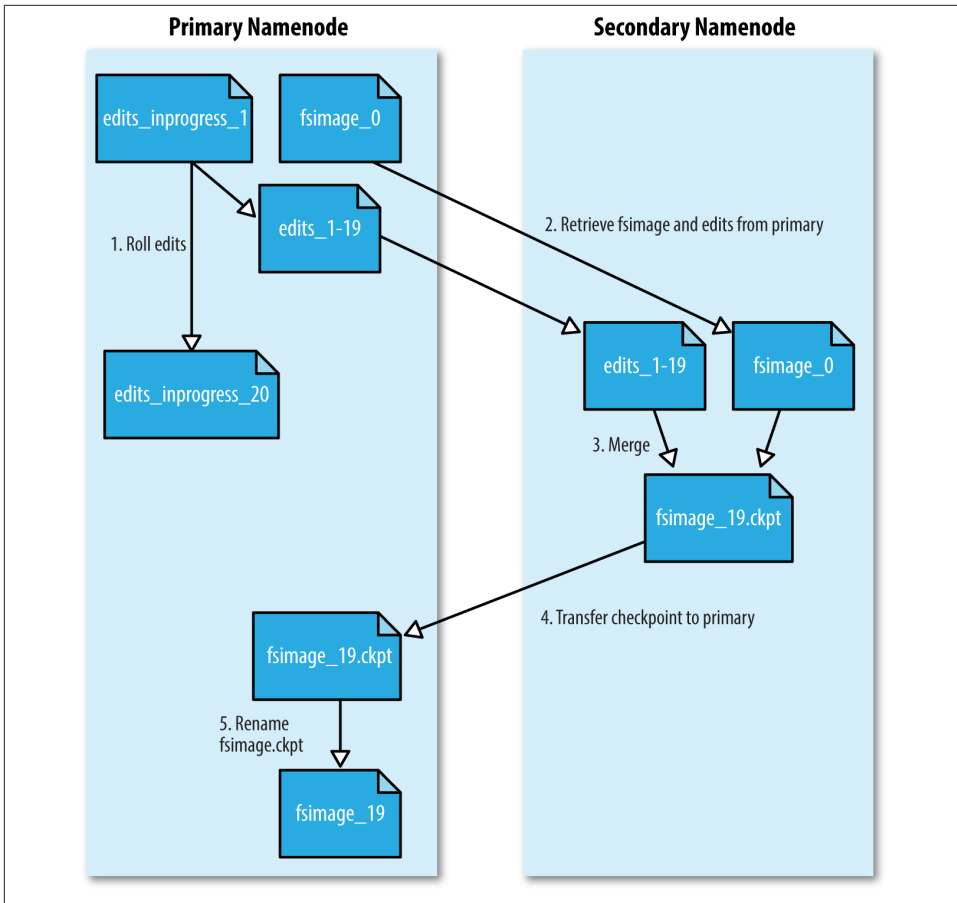


Figure 11-1. The checkpointing process

Secondary namenode directory structure

The layout of the secondary's checkpoint directory (`dfs.namenode.checkpoint.dir`) is identical to the namenode's. This is by design, since in the event of total namenode failure (when there are no recoverable backups, even from NFS), it allows recovery from a secondary namenode. This can be achieved either by copying the relevant storage directory to a new namenode or, if the secondary is taking over as the new primary namenode, by using the `-importCheckpoint` option when starting the namenode daemon. The `-importCheckpoint` option will load the namenode metadata from the latest checkpoint in the directory defined by the `dfs.namenode.checkpoint.dir` property, but only if there is no metadata in the `dfs.namenode.name.dir` directory, to ensure that there is no risk of overwriting precious metadata.

Datanode directory structure

Unlike namenodes, datanodes do not need to be explicitly formatted, because they create their storage directories automatically on startup. Here are the key files and directories:

```
{dfs.datanode.data.dir}/
├── current
│   ├── BP-526805057-127.0.0.1-1411980876842
│   │   └── current
│   │       ├── VERSION
│   │       ├── finalized
│   │       ├── blk_1073741825
│   │       ├── blk_1073741825_1001.meta
│   │       ├── blk_1073741826
│   │       └── blk_1073741826_1002.meta
│   └── rbw
├── VERSION
└── in_use.lock
```

HDFS blocks are stored in files with a *blk_* prefix; they consist of the raw bytes of a portion of the file being stored. Each block has an associated metadata file with a *.meta* suffix. It is made up of a header with version and type information, followed by a series of checksums for sections of the block.

Each block belongs to a block pool, and each block pool has its own storage directory that is formed from its ID (it's the same block pool ID from the namenode's *VERSION* file).

When the number of blocks in a directory grows to a certain size, the datanode creates a new subdirectory in which to place new blocks and their accompanying metadata. It creates a new subdirectory every time the number of blocks in a directory reaches 64 (set by the `dfs.datanode.numblocks` configuration property). The effect is to have a tree with high fan-out, so even for systems with a very large number of blocks, the directories will be only a few levels deep. By taking this measure, the datanode ensures that there is a manageable number of files per directory, which avoids the problems that most operating systems encounter when there are a large number of files (tens or hundreds of thousands) in a single directory.

If the configuration property `dfs.datanode.data.dir` specifies multiple directories on different drives, blocks are written in a round-robin fashion. Note that blocks are *not* replicated on each drive on a single datanode; instead, block replication is across distinct datanodes.

Safe Mode

When the namenode starts, the first thing it does is load its image file (*fsimage*) into memory and apply the edits from the edit log. Once it has reconstructed a consistent in-memory image of the filesystem metadata, it creates a new *fsimage* file (effectively

doing the checkpoint itself, without recourse to the secondary namenode) and an empty edit log. During this process, the namenode is running in *safe mode*, which means that it offers only a read-only view of the filesystem to clients.



Strictly speaking, in safe mode, only filesystem operations that access the filesystem metadata (such as producing a directory listing) are guaranteed to work. Reading a file will work only when the blocks are available on the current set of datanodes in the cluster, and file modifications (writes, deletes, or renames) will always fail.

Recall that the locations of blocks in the system are not persisted by the namenode; this information resides with the datanodes, in the form of a list of the blocks each one is storing. During normal operation of the system, the namenode has a map of block locations stored in memory. Safe mode is needed to give the datanodes time to check in to the namenode with their block lists, so the namenode can be informed of enough block locations to run the filesystem effectively. If the namenode didn't wait for enough datanodes to check in, it would start the process of replicating blocks to new datanodes, which would be unnecessary in most cases (because it only needed to wait for the extra datanodes to check in) and would put a great strain on the cluster's resources. Indeed, while in safe mode, the namenode does not issue any block-replication or deletion instructions to datanodes.

Safe mode is exited when the *minimal replication condition* is reached, plus an extension time of 30 seconds. The minimal replication condition is when 99.9% of the blocks in the whole filesystem meet their minimum replication level (which defaults to 1 and is set by `dfs.namenode.replication.min`; see [Table 11-1](#)).

When you are starting a newly formatted HDFS cluster, the namenode does not go into safe mode, since there are no blocks in the system.

Table 11-1. Safe mode properties

Property name	Type	Default value	Description
<code>dfs.namenode.replication.min</code>	<code>int</code>	1	The minimum number of replicas that have to be written for a write to be successful.
<code>dfs.namenode.safemode.threshold-pct</code>	<code>float</code>	0.999	The proportion of blocks in the system that must meet the minimum replication level defined by <code>dfs.namenode.replication.min</code> before the namenode will exit safe mode. Setting this value to 0 or less forces the namenode not to start in safe mode. Setting this value to more than 1 means the namenode never exits safe mode.

Property name	Type	Default value	Description
<code>dfs.namenode.safemode.extension</code>	int	30000	The time, in milliseconds, to extend safe mode after the minimum replication condition defined by <code>dfs.namenode.safemode.threshold-pct</code> has been satisfied. For small clusters (tens of nodes), it can be set to 0.

Entering and leaving safe mode

To see whether the namenode is in safe mode, you can use the `dfsadmin` command:

```
% hdfs dfsadmin -safemode get
Safe mode is ON
```

The front page of the HDFS web UI provides another indication of whether the namenode is in safe mode.

Sometimes you want to wait for the namenode to exit safe mode before carrying out a command, particularly in scripts. The `wait` option achieves this:

```
% hdfs dfsadmin -safemode wait
# command to read or write a file
```

An administrator has the ability to make the namenode enter or leave safe mode at any time. It is sometimes necessary to do this when carrying out maintenance on the cluster or after upgrading a cluster, to confirm that data is still readable. To enter safe mode, use the following command:

```
% hdfs dfsadmin -safemode enter
Safe mode is ON
```

You can use this command when the namenode is still in safe mode while starting up to ensure that it never leaves safe mode. Another way of making sure that the namenode stays in safe mode indefinitely is to set the property `dfs.namenode.safemode.threshold-pct` to a value over 1.

You can make the namenode leave safe mode by using the following:

```
% hdfs dfsadmin -safemode leave
Safe mode is OFF
```

Audit Logging

HDFS can log all filesystem access requests, a feature that some organizations require for auditing purposes. Audit logging is implemented using `log4j` logging at the `INFO` level. In the default configuration it is disabled, but it's easy to enable by adding the following line to `hadoop-env.sh`:

```
export HDFS_AUDIT_LOGGER="INFO,RFAAUDIT"
```

A log line is written to the audit log (*hdfs-audit.log*) for every HDFS event. Here's an example for a list status request on */user/tom*:

```
2014-09-30 21:35:30,484 INFO FSNamesystem.audit: allowed=true ugi=tom
(auth:SIMPLE) ip=/127.0.0.1 cmd=listStatus src=/user/tom dst=null
perm=null proto=rpc
```

Tools

dfsadmin

The *dfsadmin* tool is a multipurpose tool for finding information about the state of HDFS, as well as for performing administration operations on HDFS. It is invoked as `hdfs dfsadmin` and requires superuser privileges.

Some of the available commands to *dfsadmin* are described in [Table 11-2](#). Use the `-help` command to get more information.

Table 11-2. *dfsadmin* commands

Command	Description
<code>-help</code>	Shows help for a given command, or all commands if no command is specified.
<code>-report</code>	Shows filesystem statistics (similar to those shown in the web UI) and information on connected datanodes.
<code>-metasave</code>	Dumps information to a file in Hadoop's log directory about blocks that are being replicated or deleted, as well as a list of connected datanodes.
<code>-safemode</code>	Changes or queries the state of safe mode. See “Safe Mode” on page 322 .
<code>-saveNamespace</code>	Saves the current in-memory filesystem image to a new <i>fsimage</i> file and resets the <i>edits</i> file. This operation may be performed only in safe mode.
<code>-fetchImage</code>	Retrieves the latest <i>fsimage</i> from the namenode and saves it in a local file.
<code>-refreshNodes</code>	Updates the set of datanodes that are permitted to connect to the namenode. See “Commissioning and Decommissioning Nodes” on page 334 .
<code>-upgradeProgress</code>	Gets information on the progress of an HDFS upgrade or forces an upgrade to proceed. See “Upgrades” on page 337 .
<code>-finalizeUpgrade</code>	Removes the previous version of the namenode and datanode storage directories. Used after an upgrade has been applied and the cluster is running successfully on the new version. See “Upgrades” on page 337 .
<code>-setQuota</code>	Sets directory quotas. Directory quotas set a limit on the number of names (files or directories) in the directory tree. Directory quotas are useful for preventing users from creating large numbers of small files, a measure that helps preserve the namenode's memory (recall that accounting information for every file, directory, and block in the filesystem is stored in memory).
<code>-clrQuota</code>	Clears specified directory quotas.
<code>-setSpaceQuota</code>	Sets space quotas on directories. Space quotas set a limit on the size of files that may be stored in a directory tree. They are useful for giving users a limited amount of storage.
<code>-clrSpaceQuota</code>	Clears specified space quotas.

Command	Description
-refreshServiceAcl	Refreshes the namenode's service-level authorization policy file.
-allowSnapshot	Allows snapshot creation for the specified directory.
-disallowSnapshot	Disallows snapshot creation for the specified directory.

Filesystem check (fsck)

Hadoop provides an *fsck* utility for checking the health of files in HDFS. The tool looks for blocks that are missing from all datanodes, as well as under- or over-replicated blocks. Here is an example of checking the whole filesystem for a small cluster:

```
% hdfs fsck /
.....Status: HEALTHY
Total size: 511799225 B
Total dirs: 10
Total files: 22
Total blocks (validated): 22 (avg. block size 23263601 B)
Minimally replicated blocks: 22 (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: 4
Number of racks: 1
```

The filesystem under path '/' is HEALTHY

fsck recursively walks the filesystem namespace, starting at the given path (here the filesystem root), and checks the files it finds. It prints a dot for every file it checks. To check a file, *fsck* retrieves the metadata for the file's blocks and looks for problems or inconsistencies. Note that *fsck* retrieves all of its information from the namenode; it does not communicate with any datanodes to actually retrieve any block data.

Most of the output from *fsck* is self-explanatory, but here are some of the conditions it looks for:

Over-replicated blocks

These are blocks that exceed their target replication for the file they belong to. Normally, over-replication is not a problem, and HDFS will automatically delete excess replicas.

Under-replicated blocks

These are blocks that do not meet their target replication for the file they belong to. HDFS will automatically create new replicas of under-replicated blocks until they

meet the target replication. You can get information about the blocks being replicated (or waiting to be replicated) using `hdfs dfsadmin -metasave`.

Misreplicated blocks

These are blocks that do not satisfy the block replica placement policy (see “[Replica Placement](#)” on page 73). For example, for a replication level of three in a multirack cluster, if all three replicas of a block are on the same rack, then the block is misreplicated because the replicas should be spread across at least two racks for resilience. HDFS will automatically re-replicate misreplicated blocks so that they satisfy the rack placement policy.

Corrupt blocks

These are blocks whose replicas are all corrupt. Blocks with at least one noncorrupt replica are not reported as corrupt; the namenode will replicate the noncorrupt replica until the target replication is met.

Missing replicas

These are blocks with no replicas anywhere in the cluster.

Corrupt or missing blocks are the biggest cause for concern, as they mean data has been lost. By default, *fsck* leaves files with corrupt or missing blocks, but you can tell it to perform one of the following actions on them:

- *Move* the affected files to the */lost+found* directory in HDFS, using the *-move* option. Files are broken into chains of contiguous blocks to aid any salvaging efforts you may attempt.
- *Delete* the affected files, using the *-delete* option. Files cannot be recovered after being deleted.

Finding the blocks for a file. The *fsck* tool provides an easy way to find out which blocks are in any particular file. For example:

```
% hdfs fsck /user/tom/part-00007 -files -blocks -racks
/user/tom/part-00007 25582428 bytes, 1 block(s): OK
0. blk_-3724870485760122836_1035 len=25582428 repl=3 [/default-rack/10.251.43.2:
50010,/default-rack/10.251.27.178:50010, /default-rack/10.251.123.163:50010]
```

This says that the file */user/tom/part-00007* is made up of one block and shows the datanodes where the block is located. The *fsck* options used are as follows:

- The *-files* option shows the line with the filename, size, number of blocks, and its health (whether there are any missing blocks).
- The *-blocks* option shows information about each block in the file, one line per block.

- The `-racks` option displays the rack location and the datanode addresses for each block.

Running `hdfs fsck` without any arguments displays full usage instructions.

Datanode block scanner

Every datanode runs a *block scanner*, which periodically verifies all the blocks stored on the datanode. This allows bad blocks to be detected and fixed before they are read by clients. The scanner maintains a list of blocks to verify and scans them one by one for checksum errors. It employs a throttling mechanism to preserve disk bandwidth on the datanode.

Blocks are verified every three weeks to guard against disk errors over time (this period is controlled by the `dfs.datanode.scan.period.hours` property, which defaults to 504 hours). Corrupt blocks are reported to the namenode to be fixed.

You can get a block verification report for a datanode by visiting the datanode's web interface at <http://datanode:50075/blockScannerReport>. Here's an example of a report, which should be self-explanatory:

```
Total Blocks           : 21131
Verified in last hour   :    70
Verified in last day    :   1767
Verified in last week   :   7360
Verified in last four weeks : 20057
Verified in SCAN_PERIOD : 20057
Not yet verified        :   1074
Verified since restart   :  35912
Scans since restart     :   6541
Scan errors since restart :    0
Transient scan errors    :    0
Current scan rate limit KBps :  1024
Progress this period    :   109%
Time left in cur period  :  53.08%
```

If you specify the `listblocks` parameter, <http://datanode:50075/blockScannerReport?listblocks>, the report is preceded by a list of all the blocks on the datanode along with their latest verification status. Here is a snippet of the block list (lines are split to fit the page):

```
blk_6035596358209321442 : status : ok      type : none   scan time :
0                          not yet verified
blk_3065580480714947643 : status : ok      type : remote scan time :
1215755306400            2008-07-11 05:48:26,400
blk_8729669677359108508 : status : ok      type : local  scan time :
1215755727345            2008-07-11 05:55:27,345
```

The first column is the block ID, followed by some key-value pairs. The status can be one of `failed` or `ok`, according to whether the last scan of the block detected a checksum

error. The type of scan is `local` if it was performed by the background thread, `remote` if it was performed by a client or a remote datanode, or `none` if a scan of this block has yet to be made. The last piece of information is the scan time, which is displayed as the number of milliseconds since midnight on January 1, 1970, and also as a more readable value.

Balancer

Over time, the distribution of blocks across datanodes can become unbalanced. An unbalanced cluster can affect locality for MapReduce, and it puts a greater strain on the highly utilized datanodes, so it's best avoided.

The *balancer* program is a Hadoop daemon that redistributes blocks by moving them from overutilized datanodes to underutilized datanodes, while adhering to the block replica placement policy that makes data loss unlikely by placing block replicas on different racks (see [“Replica Placement” on page 73](#)). It moves blocks until the cluster is deemed to be balanced, which means that the utilization of every datanode (ratio of used space on the node to total capacity of the node) differs from the utilization of the cluster (ratio of used space on the cluster to total capacity of the cluster) by no more than a given threshold percentage. You can start the balancer with:

```
% start-balancer.sh
```

The `-threshold` argument specifies the threshold percentage that defines what it means for the cluster to be balanced. The flag is optional; if omitted, the threshold is 10%. At any one time, only one balancer may be running on the cluster.

The balancer runs until the cluster is balanced, it cannot move any more blocks, or it loses contact with the namenode. It produces a logfile in the standard log directory, where it writes a line for every iteration of redistribution that it carries out. Here is the output from a short run on a small cluster (slightly reformatted to fit the page):

Time Stamp	Iteration#	Bytes Already Moved	...Left To Move	...Being Moved
Mar 18, 2009 5:23:42 PM	0	0 KB	219.21 MB	150.29 MB
Mar 18, 2009 5:27:14 PM	1	195.24 MB	22.45 MB	150.29 MB

The cluster is balanced. Exiting...
Balancing took 6.072933333333333 minutes

The balancer is designed to run in the background without unduly taxing the cluster or interfering with other clients using the cluster. It limits the bandwidth that it uses to copy a block from one node to another. The default is a modest 1 MB/s, but this can be changed by setting the `dfs.datanode.balance.bandwidthPerSec` property in *hdfs-site.xml*, specified in bytes.

Monitoring

Monitoring is an important part of system administration. In this section, we look at the monitoring facilities in Hadoop and how they can hook into external monitoring systems.

The purpose of monitoring is to detect when the cluster is not providing the expected level of service. The master daemons are the most important to monitor: the namenodes (primary and secondary) and the resource manager. Failure of datanodes and node managers is to be expected, particularly on larger clusters, so you should provide extra capacity so that the cluster can tolerate having a small percentage of dead nodes at any time.

In addition to the facilities described next, some administrators run test jobs on a periodic basis as a test of the cluster's health.

Logging

All Hadoop daemons produce logfiles that can be very useful for finding out what is happening in the system. “System logfiles” on page 295 explains how to configure these files.

Setting log levels

When debugging a problem, it is very convenient to be able to change the log level temporarily for a particular component in the system.

Hadoop daemons have a web page for changing the log level for any log4j log name, which can be found at */logLevel* in the daemon's web UI. By convention, log names in Hadoop correspond to the names of the classes doing the logging, although there are exceptions to this rule, so you should consult the source code to find log names.

It's also possible to enable logging for all packages that start with a given prefix. For example, to enable debug logging for all classes related to the resource manager, we would visit the its web UI at *http://resource-manager-host:8088/logLevel* and set the log name `org.apache.hadoop.yarn.server.resourcemanager` to level `DEBUG`.

The same thing can be achieved from the command line as follows:

```
% hadoop daemonlog -setlevel resource-manager-host:8088 \  
org.apache.hadoop.yarn.server.resourcemanager DEBUG
```

Log levels changed in this way are reset when the daemon restarts, which is usually what you want. However, to make a persistent change to a log level, you can simply change the *log4j.properties* file in the configuration directory. In this case, the line to add is:

```
log4j.logger.org.apache.hadoop.yarn.server.resourcemanager=DEBUG
```

Getting stack traces

Hadoop daemons expose a web page (`/stacks` in the web UI) that produces a thread dump for all running threads in the daemon's JVM. For example, you can get a thread dump for a resource manager from `http://resource-manager-host:8088/stacks`.

Metrics and JMX

The Hadoop daemons collect information about events and measurements that are collectively known as *metrics*. For example, datanodes collect the following metrics (and many more): the number of bytes written, the number of blocks replicated, and the number of read requests from clients (both local and remote).



The metrics system in Hadoop 2 and later is sometimes referred to as *metrics2* to distinguish it from the older (now deprecated) metrics system in earlier versions of Hadoop.

Metrics belong to a *context*; “dfs,” “mapred,” “yarn,” and “rpc” are examples of different contexts. Hadoop daemons usually collect metrics under several contexts. For example, datanodes collect metrics for the “dfs” and “rpc” contexts.

How Do Metrics Differ from Counters?

The main difference is their scope: metrics are collected by Hadoop daemons, whereas counters (see “[Counters](#)” on page 247) are collected for MapReduce tasks and aggregated for the whole job. They have different audiences, too: broadly speaking, metrics are for administrators, and counters are for MapReduce users.

The way they are collected and aggregated is also different. Counters are a MapReduce feature, and the MapReduce system ensures that counter values are propagated from the task JVMs where they are produced back to the application master, and finally back to the client running the MapReduce job. (Counters are propagated via RPC heartbeats; see “[Progress and Status Updates](#)” on page 190.) Both the task process and the application master perform aggregation.

The collection mechanism for metrics is decoupled from the component that receives the updates, and there are various pluggable outputs, including local files, Ganglia, and JMX. The daemon collecting the metrics performs aggregation on them before they are sent to the output.

All Hadoop metrics are published to JMX (Java Management Extensions), so you can use standard JMX tools like JConsole (which comes with the JDK) to view them. For

remote monitoring, you must set the JMX system property `com.sun.management.jmxremote.port` (and others for security) to allow access. To do this for the namenode, say, you would set the following in *hadoop-env.sh*:

```
HADOOP_NAMENODE_OPTS="-Dcom.sun.management.jmxremote.port=8004"
```

You can also view JMX metrics (in JSON format) gathered by a particular Hadoop daemon by connecting to its */jmx* web page. This is handy for debugging. For example, you can view namenode metrics at *http://namenode-host:50070/jmx*.

Hadoop comes with a number of metrics sinks for publishing metrics to external systems, such as local files or the Ganglia monitoring system. Sinks are configured in the *hadoop-metrics2.properties* file; see that file for sample configuration settings.

Maintenance

Routine Administration Procedures

Metadata backups

If the namenode's persistent metadata is lost or damaged, the entire filesystem is rendered unusable, so it is critical that backups are made of these files. You should keep multiple copies of different ages (one hour, one day, one week, and one month, say) to protect against corruption, either in the copies themselves or in the live files running on the namenode.

A straightforward way to make backups is to use the `dfsadmin` command to download a copy of the namenode's most recent *fsimage*:

```
% hdfs dfsadmin -fetchImage fsimage.backup
```

You can write a script to run this command from an offsite location to store archive copies of the *fsimage*. The script should additionally test the integrity of the copy. This can be done by starting a local namenode daemon and verifying that it has successfully read the *fsimage* and *edits* files into memory (by scanning the namenode log for the appropriate success message, for example).²

Data backups

Although HDFS is designed to store data reliably, data loss can occur, just like in any storage system; thus, a backup strategy is essential. With the large data volumes that

2. Hadoop comes with an Offline Image Viewer and an Offline Edits Viewer, which can be used to check the integrity of the *fsimage* and *edits* files. Note that both viewers support older formats of these files, so you can use them to diagnose problems in these files generated by previous releases of Hadoop. Type `hdfs oiv` and `hdfs oev` to invoke these tools.

Hadoop can store, deciding what data to back up and where to store it is a challenge. The key here is to prioritize your data. The highest priority is the data that cannot be regenerated and that is critical to the business; however, data that is either straightforward to regenerate or essentially disposable because it is of limited business value is the lowest priority, and you may choose not to make backups of this low-priority data.



Do not make the mistake of thinking that HDFS replication is a substitute for making backups. Bugs in HDFS can cause replicas to be lost, and so can hardware failures. Although Hadoop is expressly designed so that hardware failure is very unlikely to result in data loss, the possibility can never be completely ruled out, particularly when combined with software bugs or human error.

When it comes to backups, think of HDFS in the same way as you would RAID. Although the data will survive the loss of an individual RAID disk, it may not survive if the RAID controller fails or is buggy (perhaps overwriting some data), or the entire array is damaged.

It's common to have a policy for user directories in HDFS. For example, they may have space quotas and be backed up nightly. Whatever the policy, make sure your users know what it is, so they know what to expect.

The *distcp* tool is ideal for making backups to other HDFS clusters (preferably running on a different version of the software, to guard against loss due to bugs in HDFS) or other Hadoop filesystems (such as S3) because it can copy files in parallel. Alternatively, you can employ an entirely different storage system for backups, using one of the methods for exporting data from HDFS described in “[Hadoop Filesystems](#)” on page 53.

HDFS allows administrators and users to take *snapshots* of the filesystem. A snapshot is a read-only copy of a filesystem subtree at a given point in time. Snapshots are very efficient since they do not copy data; they simply record each file's metadata and block list, which is sufficient to reconstruct the filesystem contents at the time the snapshot was taken.

Snapshots are not a replacement for data backups, but they are a useful tool for point-in-time data recovery for files that were mistakenly deleted by users. You might have a policy of taking periodic snapshots and keeping them for a specific period of time according to age. For example, you might keep hourly snapshots for the previous day and daily snapshots for the previous month.

Filesystem check (fsck)

It is advisable to run HDFS's *fsck* tool regularly (i.e., daily) on the whole filesystem to proactively look for missing or corrupt blocks. See [“Filesystem check \(fsck\)” on page 326](#).

Filesystem balancer

Run the balancer tool (see [“Balancer” on page 329](#)) regularly to keep the filesystem datanodes evenly balanced.

Commissioning and Decommissioning Nodes

As an administrator of a Hadoop cluster, you will need to add or remove nodes from time to time. For example, to grow the storage available to a cluster, you commission new nodes. Conversely, sometimes you may wish to shrink a cluster, and to do so, you decommission nodes. Sometimes it is necessary to decommission a node if it is misbehaving, perhaps because it is failing more often than it should or its performance is noticeably slow.

Nodes normally run both a datanode and a node manager, and both are typically commissioned or decommissioned in tandem.

Commissioning new nodes

Although commissioning a new node can be as simple as configuring the *hdfs-site.xml* file to point to the namenode, configuring the *yarn-site.xml* file to point to the resource manager, and starting the datanode and resource manager daemons, it is generally best to have a list of authorized nodes.

It is a potential security risk to allow any machine to connect to the namenode and act as a datanode, because the machine may gain access to data that it is not authorized to see. Furthermore, because such a machine is not a real datanode, it is not under your control and may stop at any time, potentially causing data loss. (Imagine what would happen if a number of such nodes were connected and a block of data was present only on the “alien” nodes.) This scenario is a risk even inside a firewall, due to the possibility of misconfiguration, so datanodes (and node managers) should be explicitly managed on all production clusters.

Datanodes that are permitted to connect to the namenode are specified in a file whose name is specified by the `dfs.hosts` property. The file resides on the namenode's local filesystem, and it contains a line for each datanode, specified by network address (as reported by the datanode; you can see what this is by looking at the namenode's web UI). If you need to specify multiple network addresses for a datanode, put them on one line, separated by whitespace.

Similarly, node managers that may connect to the resource manager are specified in a file whose name is specified by the `yarn.resourcemanager.nodes.include-path` property. In most cases, there is one shared file, referred to as the *include file*, that both `dfs.hosts` and `yarn.resourcemanager.nodes.include-path` refer to, since nodes in the cluster run both datanode and node manager daemons.



The file (or files) specified by the `dfs.hosts` and `yarn.resourcemanager.nodes.include-path` properties is different from the *slaves* file. The former is used by the namenode and resource manager to determine which worker nodes may connect. The *slaves* file is used by the Hadoop control scripts to perform cluster-wide operations, such as cluster restarts. It is never used by the Hadoop daemons.

To add new nodes to the cluster:

1. Add the network addresses of the new nodes to the include file.
2. Update the namenode with the new set of permitted datanodes using this command:

```
% hdfs dfsadmin -refreshNodes
```
3. Update the resource manager with the new set of permitted node managers using:

```
% yarn rmadmin -refreshNodes
```
4. Update the *slaves* file with the new nodes, so that they are included in future operations performed by the Hadoop control scripts.
5. Start the new datanodes and node managers.
6. Check that the new datanodes and node managers appear in the web UI.

HDFS will not move blocks from old datanodes to new datanodes to balance the cluster. To do this, you should run the balancer described in [“Balancer” on page 329](#).

Decommissioning old nodes

Although HDFS is designed to tolerate datanode failures, this does not mean you can just terminate datanodes en masse with no ill effect. With a replication level of three, for example, the chances are very high that you will lose data by simultaneously shutting down three datanodes if they are on different racks. The way to decommission datanodes is to inform the namenode of the nodes that you wish to take out of circulation, so that it can replicate the blocks to other datanodes before the datanodes are shut down.

With node managers, Hadoop is more forgiving. If you shut down a node manager that is running MapReduce tasks, the application master will notice the failure and reschedule the tasks on other nodes.

The decommissioning process is controlled by an *exclude file*, which is set for HDFS by the `dfs.hosts.exclude` property and for YARN by the `yarn.resourcemanager.nodes.exclude-path` property. It is often the case that these properties refer to the same file. The exclude file lists the nodes that are not permitted to connect to the cluster.

The rules for whether a node manager may connect to the resource manager are simple: a node manager may connect only if it appears in the include file and does *not* appear in the exclude file. An unspecified or empty include file is taken to mean that all nodes are in the include file.

For HDFS, the rules are slightly different. If a datanode appears in both the include and the exclude file, then it may connect, but only to be decommissioned. [Table 11-3](#) summarizes the different combinations for datanodes. As for node managers, an unspecified or empty include file means all nodes are included.

Table 11-3. HDFS include and exclude file precedence

Node appears in include file	Node appears in exclude file	Interpretation
No	No	Node may not connect.
No	Yes	Node may not connect.
Yes	No	Node may connect.
Yes	Yes	Node may connect and will be decommissioned.

To remove nodes from the cluster:

1. Add the network addresses of the nodes to be decommissioned to the exclude file. Do not update the include file at this point.
2. Update the namenode with the new set of permitted datanodes, using this command:

```
% hdfs dfsadmin -refreshNodes
```
3. Update the resource manager with the new set of permitted node managers using:

```
% yarn rmadmin -refreshNodes
```
4. Go to the web UI and check whether the admin state has changed to “Decommission In Progress” for the datanodes being decommissioned. They will start copying their blocks to other datanodes in the cluster.
5. When all the datanodes report their state as “Decommissioned,” all the blocks have been replicated. Shut down the decommissioned nodes.
6. Remove the nodes from the include file, and run:

```
% hdfs dfsadmin -refreshNodes
```

```
% yarn rmadmin -refreshNodes
```

7. Remove the nodes from the *slaves* file.

Upgrades

Upgrading a Hadoop cluster requires careful planning. The most important consideration is the HDFS upgrade. If the layout version of the filesystem has changed, then the upgrade will automatically migrate the filesystem data and metadata to a format that is compatible with the new version. As with any procedure that involves data migration, there is a risk of data loss, so you should be sure that both your data and the metadata are backed up (see “[Routine Administration Procedures](#)” on page 332).

Part of the planning process should include a trial run on a small test cluster with a copy of data that you can afford to lose. A trial run will allow you to familiarize yourself with the process, customize it to your particular cluster configuration and toolset, and iron out any snags before running the upgrade procedure on a production cluster. A test cluster also has the benefit of being available to test client upgrades on. You can read about general compatibility concerns for clients in the following sidebar.

Compatibility

When moving from one release to another, you need to think about the upgrade steps that are needed. There are several aspects to consider: API compatibility, data compatibility, and wire compatibility.

API compatibility concerns the contract between user code and the published Hadoop APIs, such as the Java MapReduce APIs. Major releases (e.g., from 1.x.y to 2.0.0) are allowed to break API compatibility, so user programs may need to be modified and recompiled. Minor releases (e.g., from 1.0.x to 1.1.0) and point releases (e.g., from 1.0.1 to 1.0.2) should not break compatibility.



Hadoop uses a classification scheme for API elements to denote their stability. The preceding rules for API compatibility cover those elements that are marked `InterfaceStability.Stable`. Some elements of the public Hadoop APIs, however, are marked with the `InterfaceStability.Evolving` or `InterfaceStability.Unstable` annotations (all these annotations are in the `org.apache.hadoop.classification` package), which mean they are allowed to break compatibility on minor and point releases, respectively.

Data compatibility concerns persistent data and metadata formats, such as the format in which the HDFS namenode stores its persistent data. The formats can change across minor or major releases, but the change is transparent to users because the upgrade will

automatically migrate the data. There may be some restrictions about upgrade paths, and these are covered in the release notes. For example, it may be necessary to upgrade via an intermediate release rather than upgrading directly to the later final release in one step.

Wire compatibility concerns the interoperability between clients and servers via wire protocols such as RPC and HTTP. The rule for wire compatibility is that the client must have the same major release number as the server, but may differ in its minor or point release number (e.g., client version 2.0.2 will work with server 2.0.1 or 2.1.0, but not necessarily with server 3.0.0).



This rule for wire compatibility differs from earlier versions of Hadoop, where internal clients (like datanodes) had to be upgraded in lockstep with servers. The fact that internal client and server versions can be mixed allows Hadoop 2 to support rolling upgrades.

The full set of compatibility rules that Hadoop adheres to are documented at the [Apache Software Foundation's website](#).

Upgrading a cluster when the filesystem layout has not changed is fairly straightforward: install the new version of Hadoop on the cluster (and on clients at the same time), shut down the old daemons, update the configuration files, and then start up the new daemons and switch clients to use the new libraries. This process is reversible, so rolling back an upgrade is also straightforward.

After every successful upgrade, you should perform a couple of final cleanup steps:

1. Remove the old installation and configuration files from the cluster.
2. Fix any deprecation warnings in your code and configuration.

Upgrades are where Hadoop cluster management tools like Cloudera Manager and Apache Ambari come into their own. They simplify the upgrade process and also make it easy to do rolling upgrades, where nodes are upgraded in batches (or one at a time for master nodes), so that clients don't experience service interruptions.

HDFS data and metadata upgrades

If you use the procedure just described to upgrade to a new version of HDFS and it expects a different layout version, then the namenode will refuse to run. A message like the following will appear in its log:

```
File system image contains an old layout version -16.  
An upgrade to version -18 is required.  
Please restart NameNode with -upgrade option.
```

The most reliable way of finding out whether you need to upgrade the filesystem is by performing a trial on a test cluster.

An upgrade of HDFS makes a copy of the previous version's metadata and data. Doing an upgrade does not double the storage requirements of the cluster, as the datanodes use hard links to keep two references (for the current and previous version) to the same block of data. This design makes it straightforward to roll back to the previous version of the filesystem, if you need to. You should understand that any changes made to the data on the upgraded system will be lost after the rollback completes, however.

You can keep only the previous version of the filesystem, which means you can't roll back several versions. Therefore, to carry out another upgrade to HDFS data and metadata, you will need to delete the previous version, a process called *finalizing the upgrade*. Once an upgrade is finalized, there is no procedure for rolling back to a previous version.

In general, you can skip releases when upgrading, but in some cases, you may have to go through intermediate releases. The release notes make it clear when this is required.

You should only attempt to upgrade a healthy filesystem. Before running the upgrade, do a full *fsck* (see “**Filesystem check (fsck)**” on page 326). As an extra precaution, you can keep a copy of the *fsck* output that lists all the files and blocks in the system, so you can compare it with the output of running *fsck* after the upgrade.

It's also worth clearing out temporary files before doing the upgrade—both local temporary files and those in the MapReduce system directory on HDFS.

With these preliminaries out of the way, here is the high-level procedure for upgrading a cluster when the filesystem layout needs to be migrated:

1. Ensure that any previous upgrade is finalized before proceeding with another upgrade.
2. Shut down the YARN and MapReduce daemons.
3. Shut down HDFS, and back up the namenode directories.
4. Install the new version of Hadoop on the cluster and on clients.
5. Start HDFS with the `-upgrade` option.
6. Wait until the upgrade is complete.
7. Perform some sanity checks on HDFS.
8. Start the YARN and MapReduce daemons.
9. Roll back or finalize the upgrade (optional).

While running the upgrade procedure, it is a good idea to remove the Hadoop scripts from your `PATH` environment variable. This forces you to be explicit about which version

of the scripts you are running. It can be convenient to define two environment variables for the new installation directories; in the following instructions, we have defined `OLD_HADOOP_HOME` and `NEW_HADOOP_HOME`.

Start the upgrade. To perform the upgrade, run the following command (this is step 5 in the high-level upgrade procedure):

```
% $NEW_HADOOP_HOME/bin/start-dfs.sh -upgrade
```

This causes the namenode to upgrade its metadata, placing the previous version in a new directory called *previous* under `dfs.namenode.name.dir`. Similarly, datanodes upgrade their storage directories, preserving the old copy in a directory called *previous*.

Wait until the upgrade is complete. The upgrade process is not instantaneous, but you can check the progress of an upgrade using *dfsadmin* (step 6; upgrade events also appear in the daemons' logfiles):

```
% $NEW_HADOOP_HOME/bin/hdfs dfsadmin -upgradeProgress status
Upgrade for version -18 has been completed.
Upgrade is not finalized.
```

Check the upgrade. This shows that the upgrade is complete. At this stage, you should run some sanity checks (step 7) on the filesystem (e.g., check files and blocks using *fsck*, test basic file operations). You might choose to put HDFS into safe mode while you are running some of these checks (the ones that are read-only) to prevent others from making changes; see [“Safe Mode” on page 322](#).

Roll back the upgrade (optional). If you find that the new version is not working correctly, you may choose to roll back to the previous version (step 9). This is possible only if you have not finalized the upgrade.



A rollback reverts the filesystem state to before the upgrade was performed, so any changes made in the meantime will be lost. In other words, it rolls back to the previous state of the filesystem, rather than downgrading the current state of the filesystem to a former version.

First, shut down the new daemons:

```
% $NEW_HADOOP_HOME/bin/stop-dfs.sh
```

Then start up the old version of HDFS with the `-rollback` option:

```
% $OLD_HADOOP_HOME/bin/start-dfs.sh -rollback
```

This command gets the namenode and datanodes to replace their current storage directories with their previous copies. The filesystem will be returned to its previous state.

Finalize the upgrade (optional). When you are happy with the new version of HDFS, you can finalize the upgrade (step 9) to remove the previous storage directories.



After an upgrade has been finalized, there is no way to roll back to the previous version.

This step is required before performing another upgrade:

```
% $NEW_HADOOP_HOME/bin/hdfs dfsadmin -finalizeUpgrade
% $NEW_HADOOP_HOME/bin/hdfs dfsadmin -upgradeProgress status
There are no upgrades in progress.
```

HDFS is now fully upgraded to the new version.