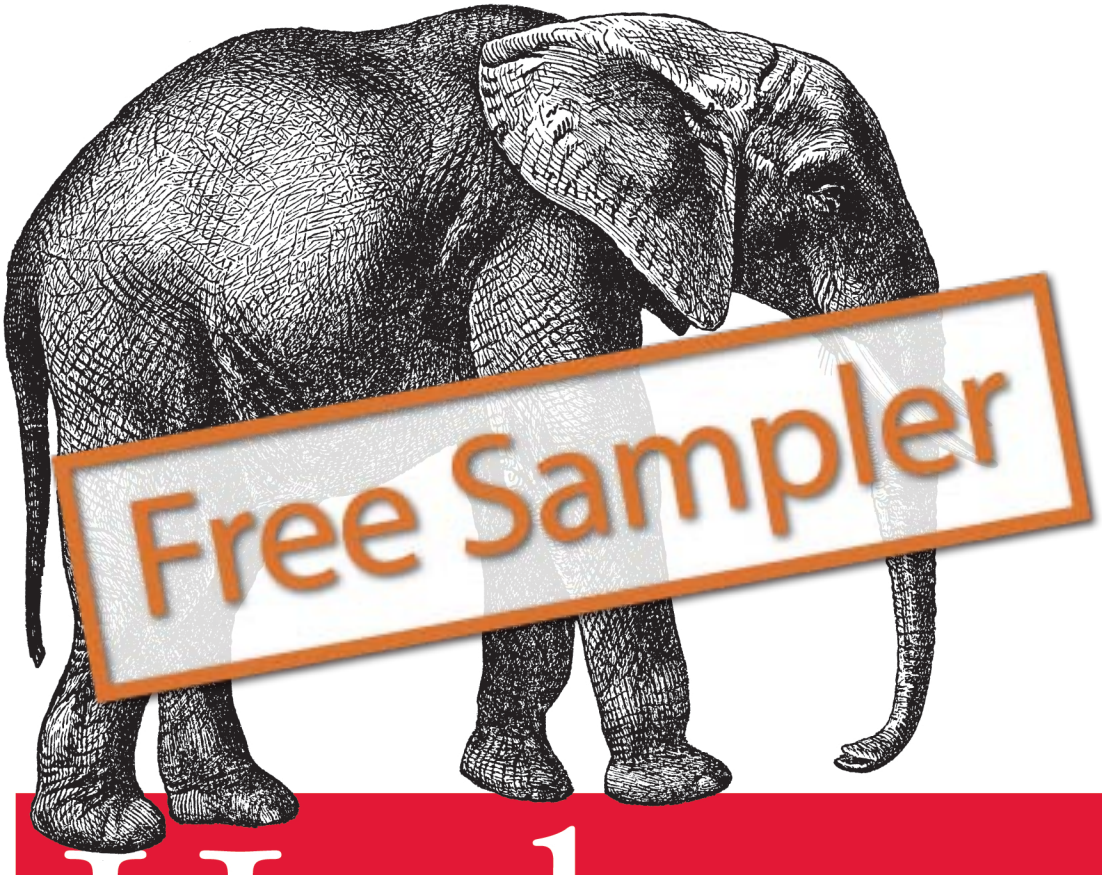
Storage and Analysis at Internet Scale

3rd Edition
Revised & Updated



Hadoop

The Definitive Guide

Chapter 3:
The Hadoop Distributed Filesystem

O'REILLY®

Tom White

Hadoop: The Definitive Guide

Ready to unlock the power of your data? With this comprehensive guide, you'll learn how to build and maintain reliable, scalable, distributed systems with Apache Hadoop. This book is ideal for programmers looking to analyze datasets of any size, and for administrators who want to set up and run Hadoop clusters.

You'll find illuminating case studies that demonstrate how Hadoop is used to solve specific problems. This third edition covers recent changes to Hadoop, including material on the new MapReduce API, as well as MapReduce 2 and its more flexible execution model (YARN).

- Store large datasets with the Hadoop Distributed File System (HDFS)
- Run distributed computations with MapReduce
- Use Hadoop's data and I/O building blocks for compression, data integrity, serialization (including Avro), and persistence
- Discover common pitfalls and advanced features for writing real-world MapReduce programs
- Design, build, and administer a dedicated Hadoop cluster—or run Hadoop in the cloud
- Load data from relational databases into HDFS, using Sqoop
- Perform large-scale data processing with the Pig query language
- Analyze datasets with Hive, Hadoop's data warehousing system
- Take advantage of HBase for structured and semi-structured data, and ZooKeeper for building distributed systems

"Now you have the opportunity to learn about Hadoop from a master—not only of the technology, but also of common sense and plain talk."

—**Doug Cutting,** *Cloudera*

Tom White, an engineer at Cloudera and member of the Apache Software Foundation, has been an Apache Hadoop committer since February 2007. He has written numerous articles for oreilly.com, java.net, and IBM's developerWorks, and speaks regularly about Hadoop at industry conferences.

cloudera

Cloudera is a leading provider of Hadoop-based software and services. Cloudera's Distribution for Hadoop (CDH) is a comprehensive Apache Hadoop-based data management platform and Cloudera Enterprise includes the tools, platform, and support necessary to use Hadoop in production.

Strata
Making Data Work

Strata is the emerging ecosystem of people, tools, and technologies that turn big data into smart decisions. Find information and resources at oreilly.com/data.

US $49.99 CAN $52.99
ISBN: 978-1-449-31152-0

Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY®
oreilly.com

THIRD EDITION

Hadoop: The Definitive Guide

Tom White

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Hadoop: The Definitive Guide, Third Edition
by Tom White

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Meghan Blanchette
Production Editor: Rachel Steely
Copyeditor: Genevieve d'Entremont
Proofreader: Kevin Broccoli

Indexer: Kevin Broccoli
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

May 2012: Third Edition.

Revision History for the Third Edition:
2012-01-27 Early release revision 1
2012-05-07 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449311520* for release details.

ISBN: 978-1-449-31152-0

[LSI]

1336503003

For Eliane, Emilia, and Lottie

CHAPTER 3

The Hadoop Distributed Filesystem

When a dataset outgrows the storage capacity of a single physical machine, it becomes necessary to partition it across a number of separate machines. Filesystems that manage the storage across a network of machines are called *distributed filesystems*. Since they are network-based, all the complications of network programming kick in, thus making distributed filesystems more complex than regular disk filesystems. For example, one of the biggest challenges is making the filesystem tolerate node failure without suffering data loss.

Hadoop comes with a distributed filesystem called HDFS, which stands for *Hadoop Distributed Filesystem*. (You may sometimes see references to "DFS"—informally or in older documentation or configurations—which is the same thing.) HDFS is Hadoop's flagship filesystem and is the focus of this chapter, but Hadoop actually has a general-purpose filesystem abstraction, so we'll see along the way how Hadoop integrates with other storage systems (such as the local filesystem and Amazon S3).

The Design of HDFS

HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.[1] Let's examine this statement in more detail:

Very large files
: "Very large" in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data.[2]

1. The architecture of HDFS is described in "The Hadoop Distributed File System" by Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler (Proceedings of MSST2010, May 2010, *http://storageconference.org/2010/Papers/MSST/Shvachko.pdf*).

2. "Scaling Hadoop to 4000 nodes at Yahoo!," *http://developer.yahoo.net/blogs/hadoop/2008/09/scaling_hadoop_to_4000_nodes_a.html*.

Streaming data access
: HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, and then various analyses are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record.

Commodity hardware
: Hadoop doesn't require expensive, highly reliable hardware. It's designed to run on clusters of commodity hardware (commonly available hardware that can be obtained from multiple vendors)[3] for which the chance of node failure across the cluster is high, at least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

It is also worth examining the applications for which using HDFS does not work so well. Although this may change in the future, these are areas where HDFS is not a good fit today:

Low-latency data access
: Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS. Remember, HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency. HBase (Chapter 13) is currently a better choice for low-latency access.

Lots of small files
: Because the namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode. As a rule of thumb, each file, directory, and block takes about 150 bytes. So, for example, if you had one million files, each taking one block, you would need at least 300 MB of memory. Although storing millions of files is feasible, billions is beyond the capability of current hardware.[4]

Multiple writers, arbitrary file modifications
: Files in HDFS may be written to by a single writer. Writes are always made at the end of the file. There is no support for multiple writers or for modifications at arbitrary offsets in the file. (These might be supported in the future, but they are likely to be relatively inefficient.)

3. See Chapter 9 for a typical machine specification.

4. For an in-depth exposition of the scalability limits of HDFS, see Konstantin V. Shvachko's "Scalability of the Hadoop Distributed File System" (*http://developer.yahoo.net/blogs/hadoop/2010/05/scalability_of_the_hadoop_dist.html*) and the companion paper, "HDFS Scalability: The limits to growth" (April 2010, pp. 6–16), *http://www.usenix.org/publications/login/2010-04/openpdfs/shvachko.pdf*), by the same author.

HDFS Concepts

Blocks

A disk has a block size, which is the minimum amount of data that it can read or write. Filesystems for a single disk build on this by dealing with data in blocks, which are an integral multiple of the disk block size. Filesystem blocks are typically a few kilobytes in size, whereas disk blocks are normally 512 bytes. This is generally transparent to the filesystem user who is simply reading or writing a file of whatever length. However, there are tools to perform filesystem maintenance, such as *df* and *fsck*, that operate on the filesystem block level.

HDFS, too, has the concept of a block, but it is a much larger unit—64 MB by default. Like in a filesystem for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage. When unqualified, the term "block" in this book refers to a block in HDFS.

Why Is a Block in HDFS So Large?

HDFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks. By making a block large enough, the time to transfer the data from the disk can be significantly longer than the time to seek to the start of the block. Thus the time to transfer a large file made of multiple blocks operates at the disk transfer rate.

A quick calculation shows that if the seek time is around 10 ms and the transfer rate is 100 MB/s, to make the seek time 1% of the transfer time, we need to make the block size around 100 MB. The default is actually 64 MB, although many HDFS installations use 128 MB blocks. This figure will continue to be revised upward as transfer speeds grow with new generations of disk drives.

This argument shouldn't be taken too far, however. Map tasks in MapReduce normally operate on one block at a time, so if you have too few tasks (fewer than nodes in the cluster), your jobs will run slower than they could otherwise.

Having a block abstraction for a distributed filesystem brings several benefits. The first benefit is the most obvious: a file can be larger than any single disk in the network. There's nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster. In fact, it would be possible, if unusual, to store a single file on an HDFS cluster whose blocks filled all the disks in the cluster.

Second, making the unit of abstraction a block rather than a file simplifies the storage subsystem. Simplicity is something to strive for in all systems, but is especially important for a distributed system in which the failure modes are so varied. The storage subsystem deals with blocks, simplifying storage management (because blocks are a fixed size, it is easy to calculate how many can be stored on a given disk) and eliminating metadata concerns (because blocks are just a chunk of data to be stored, file metadata such as permissions information does not need to be stored with the blocks, so another system can handle metadata separately).

Furthermore, blocks fit well with replication for providing fault tolerance and availability. To insure against corrupted blocks and disk and machine failure, each block is replicated to a small number of physically separate machines (typically three). If a block becomes unavailable, a copy can be read from another location in a way that is transparent to the client. A block that is no longer available due to corruption or machine failure can be replicated from its alternative locations to other live machines to bring the replication factor back to the normal level. (See "Data Integrity" on page 81 for more on guarding against corrupt data.) Similarly, some applications may choose to set a high replication factor for the blocks in a popular file to spread the read load on the cluster.

Like its disk filesystem cousin, HDFS's `fsck` command understands blocks. For example, running:

```
% hadoop fsck / -files -blocks
```

will list the blocks that make up each file in the filesystem. (See also "Filesystem check (fsck)" on page 347.)

Namenodes and Datanodes

An HDFS cluster has two types of nodes operating in a master-worker pattern: a *namenode* (the master) and a number of *datanodes* (workers). The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located; however, it does not store block locations persistently, because this information is reconstructed from datanodes when the system starts.

A *client* accesses the filesystem on behalf of the user by communicating with the namenode and datanodes. The client presents a filesystem interface similar to a Portable Operating System Interface (POSIX), so the user code does not need to know about the namenode and datanode to function.

Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing.

Without the namenode, the filesystem cannot be used. In fact, if the machine running the namenode were obliterated, all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes. For this reason, it is important to make the namenode resilient to failure, and Hadoop provides two mechanisms for this.

The first way is to back up the files that make up the persistent state of the filesystem metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic. The usual configuration choice is to write to local disk as well as a remote NFS mount.

It is also possible to run a *secondary namenode*, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine because it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing. However, the state of the secondary namenode lags that of the primary, so in the event of total failure of the primary, data loss is almost certain. The usual course of action in this case is to copy the namenode's metadata files that are on NFS to the secondary and run it as the new primary.

See "The filesystem image and edit log" on page 340 for more details.

HDFS Federation

The namenode keeps a reference to every file and block in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling (see "How Much Memory Does a Namenode Need?" on page 308). HDFS Federation, introduced in the 2.x release series, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace. For example, one namenode might manage all the files rooted under */user*, say, and a second namenode might handle files under */share*.

Under federation, each namenode manages a *namespace volume*, which is made up of the metadata for the namespace, and a *block pool* containing all the blocks for the files in the namespace. Namespace volumes are independent of each other, which means namenodes do not communicate with one another, and furthermore the failure of one namenode does not affect the availability of the namespaces managed by other namenodes. Block pool storage is *not* partitioned, however, so datanodes register with each namenode in the cluster and store blocks from multiple block pools.

To access a federated HDFS cluster, clients use client-side mount tables to map file paths to namenodes. This is managed in configuration using `ViewFileSystem` and the *viewfs://* URIs.

HDFS High-Availability

The combination of replicating namenode metadata on multiple filesystems and using the secondary namenode to create checkpoints protects against data loss, but it does not provide high-availability of the filesystem. The namenode is still a *single point of failure* (SPOF). If it did fail, all clients—including MapReduce jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping. In such an event the whole Hadoop system would effectively be out of service until a new namenode could be brought online.

To recover from a failed namenode in this situation, an administrator starts a new primary namenode with one of the filesystem metadata replicas and configures datanodes and clients to use this new namenode. The new namenode is not able to serve requests until it has i) loaded its namespace image into memory, ii) replayed its edit log, and iii) received enough block reports from the datanodes to leave safe mode. On large clusters with many files and blocks, the time it takes for a namenode to start from cold can be 30 minutes or more.

The long recovery time is a problem for routine maintenance too. In fact, because unexpected failure of the namenode is so rare, the case for planned downtime is actually more important in practice.

The 2.x release series of Hadoop remedies this situation by adding support for HDFS high-availability (HA). In this implementation there is a pair of namenodes in an active-standby configuration. In the event of the failure of the active namenode, the standby takes over its duties to continue servicing client requests without a significant interruption. A few architectural changes are needed to allow this to happen:

- The namenodes must use highly available shared storage to share the edit log. (In the initial implementation of HA this will require an NFS filer, but in future releases more options will be provided, such as a BookKeeper-based system built on ZooKeeper.) When a standby namenode comes up, it reads up to the end of the shared edit log to synchronize its state with the active namenode, and then continues to read new entries as they are written by the active namenode.
- Datanodes must send block reports to both namenodes because the block mappings are stored in a namenode's memory, and not on disk.
- Clients must be configured to handle namenode failover, using a mechanism that is transparent to users.

If the active namenode fails, the standby can take over very quickly (in a few tens of seconds) because it has the latest state available in memory: both the latest edit log entries and an up-to-date block mapping. The actual observed failover time will be longer in practice (around a minute or so), because the system needs to be conservative in deciding that the active namenode has failed.

In the unlikely event of the standby being down when the active fails, the administrator can still start the standby from cold. This is no worse than the non-HA case, and from an operational point of view it's an improvement, because the process is a standard operational procedure built into Hadoop.

Failover and fencing

The transition from the active namenode to the standby is managed by a new entity in the system called the *failover controller*. Failover controllers are pluggable, but the first implementation uses ZooKeeper to ensure that only one namenode is active. Each namenode runs a lightweight failover controller process whose job it is to monitor its namenode for failures (using a simple heartbeating mechanism) and trigger a failover should a namenode fail.

Failover may also be initiated manually by an administrator, for example, in the case of routine maintenance. This is known as a *graceful failover*, since the failover controller arranges an orderly transition for both namenodes to switch roles.

In the case of an ungraceful failover, however, it is impossible to be sure that the failed namenode has stopped running. For example, a slow network or a network partition can trigger a failover transition, even though the previously active namenode is still running and thinks it is still the active namenode. The HA implementation goes to great lengths to ensure that the previously active namenode is prevented from doing any damage and causing corruption—a method known as *fencing*. The system employs a range of fencing mechanisms, including killing the namenode's process, revoking its access to the shared storage directory (typically by using a vendor-specific NFS command), and disabling its network port via a remote management command. As a last resort, the previously active namenode can be fenced with a technique rather graphically known as *STONITH*, or "shoot the other node in the head," which uses a specialized power distribution unit to forcibly power down the host machine.

Client failover is handled transparently by the client library. The simplest implementation uses client-side configuration to control failover. The HDFS URI uses a logical hostname that is mapped to a pair of namenode addresses (in the configuration file), and the client library tries each namenode address until the operation succeeds.

The Command-Line Interface

We're going to have a look at HDFS by interacting with it from the command line. There are many other interfaces to HDFS, but the command line is one of the simplest and, to many developers, the most familiar.

We are going to run HDFS on one machine, so first follow the instructions for setting up Hadoop in pseudodistributed mode in Appendix A. Later we'll see how to run HDFS on a cluster of machines to give us scalability and fault tolerance.

There are two properties that we set in the pseudodistributed configuration that deserve further explanation. The first is `fs.default.name`, set to *hdfs://localhost/*, which is used to set a default filesystem for Hadoop. Filesystems are specified by a URI, and here we have used an `hdfs` URI to configure Hadoop to use HDFS by default. The HDFS daemons will use this property to determine the host and port for the HDFS namenode. We'll be running it on `localhost`, on the default HDFS port, 8020. And HDFS clients will use this property to work out where the namenode is running so they can connect to it.

We set the second property, `dfs.replication`, to 1 so that HDFS doesn't replicate filesystem blocks by the default factor of three. When running with a single datanode, HDFS can't replicate blocks to three datanodes, so it would perpetually warn about blocks being under-replicated. This setting solves that problem.

Basic Filesystem Operations

The filesystem is ready to be used, and we can do all of the usual filesystem operations, such as reading files, creating directories, moving files, deleting data, and listing directories. You can type `hadoop fs -help` to get detailed help on every command.

Start by copying a file from the local filesystem to HDFS:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt hdfs://localhost/user/tom/
      quangle.txt
```

This command invokes Hadoop's filesystem shell command `fs`, which supports a number of subcommands—in this case, we are running `-copyFromLocal`. The local file *quangle.txt* is copied to the file */user/tom/quangle.txt* on the HDFS instance running on localhost. In fact, we could have omitted the scheme and host of the URI and picked up the default, `hdfs://localhost`, as specified in *core-site.xml*:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt /user/tom/quangle.txt
```

We also could have used a relative path and copied the file to our home directory in HDFS, which in this case is */user/tom*:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt quangle.txt
```

Let's copy the file back to the local filesystem and check whether it's the same:

```
% hadoop fs -copyToLocal quangle.txt quangle.copy.txt
% md5 input/docs/quangle.txt quangle.copy.txt
MD5 (input/docs/quangle.txt) = a16f231da6b05e2ba7a339320e7dacd9
MD5 (quangle.copy.txt) = a16f231da6b05e2ba7a339320e7dacd9
```

The MD5 digests are the same, showing that the file survived its trip to HDFS and is back intact.

Finally, let's look at an HDFS file listing. We create a directory first just to see how it is displayed in the listing:

```
% hadoop fs -mkdir books
% hadoop fs -ls .
Found 2 items
drwxr-xr-x   - tom supergroup          0 2009-04-02 22:41 /user/tom/books
-rw-r--r--   1 tom supergroup        118 2009-04-02 22:29 /user/tom/quangle.txt
```

The information returned is very similar to the Unix command `ls -l`, with a few minor differences. The first column shows the file mode. The second column is the replication factor of the file (something a traditional Unix filesystem does not have). Remember we set the default replication factor in the site-wide configuration to be 1, which is why we see the same value here. The entry in this column is empty for directories because the concept of replication does not apply to them—directories are treated as metadata and stored by the namenode, not the datanodes. The third and fourth columns show the file owner and group. The fifth column is the size of the file in bytes, or zero for directories. The sixth and seventh columns are the last modified date and time. Finally, the eighth column is the absolute name of the file or directory.

File Permissions in HDFS

HDFS has a permissions model for files and directories that is much like POSIX.

There are three types of permission: the read permission (`r`), the write permission (`w`), and the execute permission (`x`). The read permission is required to read files or list the contents of a directory. The write permission is required to write a file, or for a directory, to create or delete files or directories in it. The execute permission is ignored for a file because you can't execute a file on HDFS (unlike POSIX), and for a directory this permission is required to access its children.

Each file and directory has an *owner*, a *group*, and a *mode*. The mode is made up of the permissions for the user who is the owner, the permissions for the users who are members of the group, and the permissions for users who are neither the owners nor members of the group.

By default, a client's identity is determined by the username and groups of the process it is running in. Because clients are remote, this makes it possible to become an arbitrary user simply by creating an account of that name on the remote system. Thus, permissions should be used only in a cooperative community of users, as a mechanism for sharing filesystem resources and for avoiding accidental data loss, and not for securing resources in a hostile environment. (Note, however, that the latest versions of Hadoop support Kerberos authentication, which removes these restrictions; see "Security" on page 325.) Despite these limitations, it is worthwhile having permissions enabled (as it is by default; see the `dfs.permissions` property), to avoid accidental modification or deletion of substantial parts of the filesystem, either by users or by automated tools or programs.

When permissions checking is enabled, the owner permissions are checked if the client's username matches the owner, and the group permissions are checked if the client is a member of the group; otherwise, the other permissions are checked.

There is a concept of a super user, which is the identity of the namenode process. Permissions checks are not performed for the super user.

Hadoop Filesystems

Hadoop has an abstract notion of filesystem, of which HDFS is just one implementation. The Java abstract class `org.apache.hadoop.fs.FileSystem` represents a filesystem in Hadoop, and there are several concrete implementations, which are described in Table 3-1.

Table 3-1. Hadoop filesystems

Filesystem	URI scheme	Java implementation (all under org.apache.hadoop)	Description
Local	*file*	`fs.LocalFileSystem`	A filesystem for a locally connected disk with client-side checksums. Use `RawLocalFileSystem` for a local filesystem with no checksums. See "LocalFileSystem" on page 82.
HDFS	*hdfs*	`hdfs.DistributedFileSystem`	Hadoop's distributed filesystem. HDFS is designed to work efficiently in conjunction with MapReduce.
HFTP	*hftp*	`hdfs.HftpFileSystem`	A filesystem providing read-only access to HDFS over HTTP. (Despite its name, HFTP has no connection with FTP.) Often used with *distcp* (see "Parallel Copying with distcp" on page 75) to copy data between HDFS clusters running different versions.
HSFTP	*hsftp*	`hdfs.HsftpFileSystem`	A filesystem providing read-only access to HDFS over HTTPS. (Again, this has no connection with FTP.)
WebHDFS	*webhdfs*	`hdfs.web.WebHdfsFileSystem`	A filesystem providing secure read-write access to HDFS over HTTP. WebHDFS is intended as a replacement for HFTP and HSFTP.
HAR	*har*	`fs.HarFileSystem`	A filesystem layered on another filesystem for archiving files. Hadoop Archives are typically used for archiving files in HDFS to reduce the namenode's memory usage. See "Hadoop Archives" on page 77.
KFS (CloudStore)	*kfs*	`fs.kfs.KosmosFileSystem`	CloudStore (formerly Kosmos filesystem) is a distributed filesystem like HDFS or Google's GFS, written in C++. Find more information about it at *http://code.google.com/p/kosmosfs/*.
FTP	*ftp*	`fs.ftp.FTPFileSystem`	A filesystem backed by an FTP server.
S3 (native)	*s3n*	`fs.s3native.NativeS3FileSystem`	A filesystem backed by Amazon S3. See *http://wiki.apache.org/hadoop/AmazonS3*.

Filesystem	URI scheme	Java implementation (all under org.apache.hadoop)	Description
S3 (block-based)	*s3*	`fs.s3.S3FileSystem`	A filesystem backed by Amazon S3, which stores files in blocks (much like HDFS) to overcome S3's 5 GB file size limit.
Distributed RAID	*hdfs*	`hdfs.DistributedRaidFileSystem`	A "RAID" version of HDFS designed for archival storage. For each file in HDFS, a (smaller) parity file is created, which allows the HDFS replication to be reduced from three to two, which reduces disk usage by 25% to 30% while keeping the probability of data loss the same. Distributed RAID requires that you run a `RaidNode` daemon on the cluster.
View	*viewfs*	`viewfs.ViewFileSystem`	A client-side mount table for other Hadoop filesystems. Commonly used to create mount points for federated namenodes (see "HDFS Federation" on page 47).

Hadoop provides many interfaces to its filesystems, and it generally uses the URI scheme to pick the correct filesystem instance to communicate with. For example, the filesystem shell that we met in the previous section operates with all Hadoop filesystems. To list the files in the root directory of the local filesystem, type:

```
% hadoop fs -ls file:///
```

Although it is possible (and sometimes very convenient) to run MapReduce programs that access any of these filesystems, when you are processing large volumes of data, you should choose a distributed filesystem that has the data locality optimization, notably HDFS (see "Scaling Out" on page 30).

Interfaces

Hadoop is written in Java, and all Hadoop filesystem interactions are mediated through the Java API. The filesystem shell, for example, is a Java application that uses the Java `FileSystem` class to provide filesystem operations. The other filesystem interfaces are discussed briefly in this section. These interfaces are most commonly used with HDFS, since the other filesystems in Hadoop typically have existing tools to access the underlying filesystem (FTP clients for FTP, S3 tools for S3, etc.), but many of them will work with any Hadoop filesystem.

HTTP

There are two ways of accessing HDFS over HTTP: directly, where the HDFS daemons serve HTTP requests to clients; and via a proxy (or proxies), which accesses HDFS on the client's behalf using the usual `DistributedFileSystem` API. The two ways are illustrated in Figure 3-1.

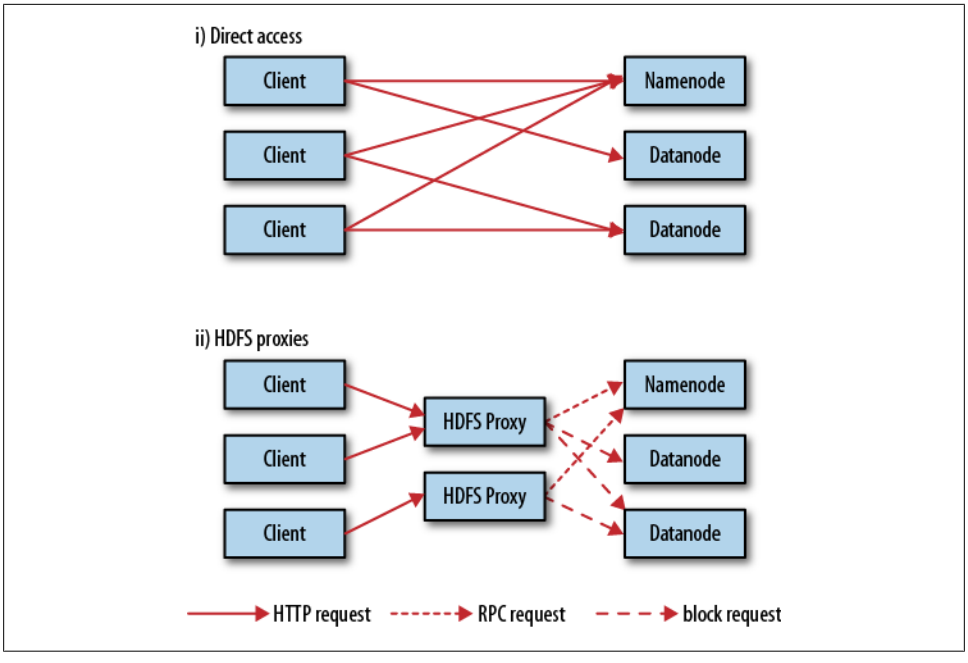


Figure 3-1. Accessing HDFS over HTTP directly and via a bank of HDFS proxies

In the first case, directory listings are served by the namenode's embedded web server (which runs on port 50070) formatted in XML or JSON, whereas file data is streamed from datanodes by their web servers (running on port 50075).

The original direct HTTP interface (HFTP and HSFTP) was read-only, but the new WebHDFS implementation supports all filesystem operations, including Kerberos authentication. WebHDFS must be enabled by setting `dfs.webhdfs.enabled` to true, which allows you to use *webhdfs* URIs.

The second way of accessing HDFS over HTTP relies on one or more standalone proxy servers. (The proxies are stateless so they can run behind a standard load balancer.) All traffic to the cluster passes through the proxy. This allows for stricter firewall and bandwidth-limiting policies to be put in place. It's common to use a proxy for transfers between Hadoop clusters located in different data centers.

The original HDFS proxy (in *src/contrib/hdfsproxy*) was read-only and could be accessed by clients using the HSFTP `FileSystem` implementation (*hsftp* URIs). From release 1.0.0, there is a new proxy called HttpFS that has read and write capabilities and exposes the same HTTP interface as WebHDFS, so clients can access both using *webhdfs* URIs.

The HTTP REST API that WebHDFS exposes is formally defined in a specification, so it is expected that over time clients in languages other than Java will be written that use it directly.

C

Hadoop provides a C library called *libhdfs* that mirrors the Java `FileSystem` interface (it was written as a C library for accessing HDFS, but despite its name it can be used to access any Hadoop filesystem). It works using the *Java Native Interface* (JNI) to call a Java filesystem client.

The C API is very similar to the Java one, but it typically lags the Java one, so newer features may not be supported. You can find the generated documentation for the C API in the *libhdfs/docs/api* directory of the Hadoop distribution.

Hadoop comes with prebuilt *libhdfs* binaries for 32-bit Linux, but for other platforms, you will need to build them yourself using the instructions at *http://wiki.apache.org/hadoop/LibHDFS*.

FUSE

Filesystem in Userspace (FUSE) allows filesystems that are implemented in user space to be integrated as a Unix filesystem. Hadoop's Fuse-DFS contrib module allows any Hadoop filesystem (but typically HDFS) to be mounted as a standard filesystem. You can then use Unix utilities (such as `ls` and `cat`) to interact with the filesystem, as well as POSIX libraries to access the filesystem from any programming language.

Fuse-DFS is implemented in C using *libhdfs* as the interface to HDFS. Documentation for compiling and running Fuse-DFS is located in the *src/contrib/fuse-dfs* directory of the Hadoop distribution.

The Java Interface

In this section, we dig into the Hadoop's `FileSystem` class: the API for interacting with one of Hadoop's filesystems.[5] Although we focus mainly on the HDFS implementation, `DistributedFileSystem`, in general you should strive to write your code against the `FileSystem` abstract class, to retain portability across filesystems. This is very useful when testing your program, for example, because you can rapidly run tests using data stored on the local filesystem.

Reading Data from a Hadoop URL

One of the simplest ways to read a file from a Hadoop filesystem is by using a `java.net.URL` object to open a stream to read the data from. The general idiom is:

```
InputStream in = null;
try {
```

5. In releases after 1.x, there is a new filesystem interface called `FileContext` with better handling of multiple filesystems (so a single `FileContext` can resolve multiple filesystem schemes, for example) and a cleaner, more consistent interface.

```
      in = new URL("hdfs://host/path").openStream();
      // process in
    } finally {
      IOUtils.closeStream(in);
    }
```

There's a little bit more work required to make Java recognize Hadoop's `hdfs` URL scheme. This is achieved by calling the `setURLStreamHandlerFactory` method on `URL` with an instance of `FsUrlStreamHandlerFactory`. This method can be called only once per JVM, so it is typically executed in a static block. This limitation means that if some other part of your program—perhaps a third-party component outside your control—sets a `URLStreamHandlerFactory`, you won't be able to use this approach for reading data from Hadoop. The next section discusses an alternative.

Example 3-1 shows a program for displaying files from Hadoop filesystems on standard output, like the Unix `cat` command.

Example 3-1. Displaying files from a Hadoop filesystem on standard output using a URLStreamHandler

```
public class URLCat {

  static {
    URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
  }

  public static void main(String[] args) throws Exception {
    InputStream in = null;
    try {
      in = new URL(args[0]).openStream();
      IOUtils.copyBytes(in, System.out, 4096, false);
    } finally {
      IOUtils.closeStream(in);
    }
  }
}
```

We make use of the handy `IOUtils` class that comes with Hadoop for closing the stream in the `finally` clause, and also for copying bytes between the input stream and the output stream (`System.out` in this case). The last two arguments to the `copyBytes` method are the buffer size used for copying and whether to close the streams when the copy is complete. We close the input stream ourselves, and `System.out` doesn't need to be closed.

Here's a sample run:[6]

```
% hadoop URLCat hdfs://localhost/user/tom/quangle.txt
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

Reading Data Using the FileSystem API

As the previous section explained, sometimes it is impossible to set a `URLStreamHandlerFactory` for your application. In this case, you will need to use the `FileSystem` API to open an input stream for a file.

A file in a Hadoop filesystem is represented by a Hadoop `Path` object (and not a `java.io.File` object, since its semantics are too closely tied to the local filesystem). You can think of a `Path` as a Hadoop filesystem URI, such as *hdfs://localhost/user/tom/quangle.txt*.

`FileSystem` is a general filesystem API, so the first step is to retrieve an instance for the filesystem we want to use—HDFS in this case. There are several static factory methods for getting a `FileSystem` instance:

```
public static FileSystem get(Configuration conf) throws IOException
public static FileSystem get(URI uri, Configuration conf) throws IOException
public static FileSystem get(URI uri, Configuration conf, String user)
  throws IOException
```

A `Configuration` object encapsulates a client or server's configuration, which is set using configuration files read from the classpath, such as *conf/core-site.xml*. The first method returns the default filesystem (as specified in the file *conf/core-site.xml*, or the default local filesystem if not specified there). The second uses the given `URI`'s scheme and authority to determine the filesystem to use, falling back to the default filesystem if no scheme is specified in the given `URI`. The third retrieves the filesystem as the given user, which is important in the context of security (see "Security" on page 325).

In some cases, you may want to retrieve a local filesystem instance, in which case you can use the convenience method, `getLocal()`:

```
public static LocalFileSystem getLocal(Configuration conf) throws IOException
```

With a `FileSystem` instance in hand, we invoke an `open()` method to get the input stream for a file:

```
public FSDataInputStream open(Path f) throws IOException
public abstract FSDataInputStream open(Path f, int bufferSize) throws IOException
```

The first method uses a default buffer size of 4 KB.

Putting this together, we can rewrite Example 3-1 as shown in Example 3-2.

6. The text is from *The Quangle Wangle's Hat* by Edward Lear.

Example 3-2. Displaying files from a Hadoop filesystem on standard output by using the FileSystem directly

```
public class FileSystemCat {

  public static void main(String[] args) throws Exception {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);
    InputStream in = null;
    try {
      in = fs.open(new Path(uri));
      IOUtils.copyBytes(in, System.out, 4096, false);
    } finally {
      IOUtils.closeStream(in);
    }
  }
}
```

The program runs as follows:

```
% hadoop FileSystemCat hdfs://localhost/user/tom/quangle.txt
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

FSDataInputStream

The `open()` method on `FileSystem` actually returns a `FSDataInputStream` rather than a standard `java.io` class. This class is a specialization of `java.io.DataInputStream` with support for random access, so you can read from any part of the stream:

```
package org.apache.hadoop.fs;

public class FSDataInputStream extends DataInputStream
    implements Seekable, PositionedReadable {
  // implementation elided
}
```

The `Seekable` interface permits seeking to a position in the file and a query method for the current offset from the start of the file (`getPos()`):

```
public interface Seekable {
  void seek(long pos) throws IOException;
  long getPos() throws IOException;
}
```

Calling `seek()` with a position that is greater than the length of the file will result in an `IOException`. Unlike the `skip()` method of `java.io.InputStream`, which positions the stream at a point later than the current position, `seek()` can move to an arbitrary, absolute position in the file.

Example 3-3 is a simple extension of Example 3-2 that writes a file to standard out twice: after writing it once, it seeks to the start of the file and streams through it once again.

Example 3-3. Displaying files from a Hadoop filesystem on standard output twice, by using seek

```
public class FileSystemDoubleCat {

  public static void main(String[] args) throws Exception {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);
    FSDataInputStream in = null;
    try {
      in = fs.open(new Path(uri));
      IOUtils.copyBytes(in, System.out, 4096, false);
      in.seek(0); // go back to the start of the file
      IOUtils.copyBytes(in, System.out, 4096, false);
    } finally {
      IOUtils.closeStream(in);
    }
  }
}
```

Here's the result of running it on a small file:

```
% hadoop FileSystemDoubleCat hdfs://localhost/user/tom/quangle.txt
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

`FSDataInputStream` also implements the `PositionedReadable` interface for reading parts of a file at a given offset:

```
public interface PositionedReadable {

  public int read(long position, byte[] buffer, int offset, int length)
    throws IOException;

  public void readFully(long position, byte[] buffer, int offset, int length)
    throws IOException;

  public void readFully(long position, byte[] buffer) throws IOException;
}
```

The `read()` method reads up to `length` bytes from the given `position` in the file into the `buffer` at the given `offset` in the buffer. The return value is the number of bytes actually read; callers should check this value, as it may be less than `length`. The `readFully()` methods will read `length` bytes into the buffer (or `buffer.length` bytes for the version

that just takes a byte array `buffer`), unless the end of the file is reached, in which case an `EOFException` is thrown.

All of these methods preserve the current offset in the file and are thread-safe (although `FSDataInputStream` is not designed for concurrent access, therefore, it's better to create multiple instances), so they provide a convenient way to access another part of the file —metadata perhaps—while reading the main body of the file.

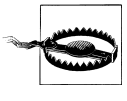
Finally, bear in mind that calling `seek()` is a relatively expensive operation and should be used sparingly. You should structure your application access patterns to rely on streaming data (by using MapReduce, for example) rather than performing a large number of seeks.

Writing Data

The `FileSystem` class has a number of methods for creating a file. The simplest is the method that takes a `Path` object for the file to be created and returns an output stream to write to:

```
public FSDataOutputStream create(Path f) throws IOException
```

There are overloaded versions of this method that allow you to specify whether to forcibly overwrite existing files, the replication factor of the file, the buffer size to use when writing the file, the block size for the file, and file permissions.



The `create()` methods create any parent directories of the file to be written that don't already exist. Though convenient, this behavior may be unexpected. If you want the write to fail when the parent directory doesn't exist, you should check for the existence of the parent directory first by calling the `exists()` method.

There's also an overloaded method for passing a callback interface, `Progressable`, so your application can be notified of the progress of the data being written to the datanodes:

```
package org.apache.hadoop.util;

public interface Progressable {
  public void progress();
}
```

As an alternative to creating a new file, you can append to an existing file using the `append()` method (there are also some other overloaded versions):

```
public FSDataOutputStream append(Path f) throws IOException
```

The append operation allows a single writer to modify an already written file by opening it and writing data from the final offset in the file. With this API, applications that produce unbounded files, such as logfiles, can write to an existing file after having

closed it. The append operation is optional and not implemented by all Hadoop filesystems. For example, HDFS supports append,[7] but S3 filesystems don't.

Example 3-4 shows how to copy a local file to a Hadoop filesystem. We illustrate progress by printing a period every time the `progress()` method is called by Hadoop, which is after each 64 KB packet of data is written to the datanode pipeline. (Note that this particular behavior is not specified by the API, so it is subject to change in later versions of Hadoop. The API merely allows you to infer that "something is happening."

Example 3-4. Copying a local file to a Hadoop filesystem

```
public class FileCopyWithProgress {
  public static void main(String[] args) throws Exception {
    String localSrc = args[0];
    String dst = args[1];

    InputStream in = new BufferedInputStream(new FileInputStream(localSrc));

    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(dst), conf);
    OutputStream out = fs.create(new Path(dst), new Progressable() {
      public void progress() {
        System.out.print(".");
      }
    });

    IOUtils.copyBytes(in, out, 4096, true);
  }
}
```

Typical usage:

```
% hadoop FileCopyWithProgress input/docs/1400-8.txt hdfs://localhost/user/tom/
      1400-8.txt
.............
```

Currently, none of the other Hadoop filesystems call `progress()` during writes. Progress is important in MapReduce applications, as you will see in later chapters.

FSDataOutputStream

The `create()` method on `FileSystem` returns an `FSDataOutputStream`, which, like `FSDataInputStream`, has a method for querying the current position in the file:

```
package org.apache.hadoop.fs;

public class FSDataOutputStream extends DataOutputStream implements Syncable {

  public long getPos() throws IOException {
```

7. There have been reliability problems with the append implementation in Hadoop 1.x, so it is generally recommended to use append only in the releases after 1.x, which contain a new, rewritten implementation.

```
      // implementation elided
    }

    // implementation elided

  }
```

However, unlike `FSDataInputStream`, `FSDataOutputStream` does not permit seeking. This is because HDFS allows only sequential writes to an open file or appends to an already written file. In other words, there is no support for writing to anywhere other than the end of the file, so there is no value in being able to seek while writing.

Directories

`FileSystem` provides a method to create a directory:

```
public boolean mkdirs(Path f) throws IOException
```

This method creates all of the necessary parent directories if they don't already exist, just like the `java.io.File`'s `mkdirs()` method. It returns `true` if the directory (and all parent directories) was (were) successfully created.

Often, you don't need to explicitly create a directory, because writing a file by calling `create()` will automatically create any parent directories.

Querying the Filesystem

File metadata: FileStatus

An important feature of any filesystem is the ability to navigate its directory structure and retrieve information about the files and directories that it stores. The `FileStatus` class encapsulates filesystem metadata for files and directories, including file length, block size, replication, modification time, ownership, and permission information.

The method `getFileStatus()` on `FileSystem` provides a way of getting a `FileStatus` object for a single file or directory. Example 3-5 shows an example of its use.

Example 3-5. Demonstrating file status information

```
public class ShowFileStatusTest {

  private MiniDFSCluster cluster; // use an in-process HDFS cluster for testing
  private FileSystem fs;

  @Before
  public void setUp() throws IOException {
    Configuration conf = new Configuration();
    if (System.getProperty("test.build.data") == null) {
      System.setProperty("test.build.data", "/tmp");
    }
    cluster = new MiniDFSCluster(conf, 1, true, null);
    fs = cluster.getFileSystem();
```

```
    OutputStream out = fs.create(new Path("/dir/file"));
    out.write("content".getBytes("UTF-8"));
    out.close();
  }

  @After
  public void tearDown() throws IOException {
    if (fs != null) { fs.close(); }
    if (cluster != null) { cluster.shutdown(); }
  }

  @Test(expected = FileNotFoundException.class)
  public void throwsFileNotFoundForNonExistentFile() throws IOException {
    fs.getFileStatus(new Path("no-such-file"));
  }

  @Test
  public void fileStatusForFile() throws IOException {
    Path file = new Path("/dir/file");
    FileStatus stat = fs.getFileStatus(file);
    assertThat(stat.getPath().toUri().getPath(), is("/dir/file"));
    assertThat(stat.isDir(), is(false));
    assertThat(stat.getLen(), is(7L));
    assertThat(stat.getModificationTime(),
        is(lessThanOrEqualTo(System.currentTimeMillis())));
    assertThat(stat.getReplication(), is((short) 1));
    assertThat(stat.getBlockSize(), is(64 * 1024 * 1024L));
    assertThat(stat.getOwner(), is("tom"));
    assertThat(stat.getGroup(), is("supergroup"));
    assertThat(stat.getPermission().toString(), is("rw-r--r--"));
  }

  @Test
  public void fileStatusForDirectory() throws IOException {
    Path dir = new Path("/dir");
    FileStatus stat = fs.getFileStatus(dir);
    assertThat(stat.getPath().toUri().getPath(), is("/dir"));
    assertThat(stat.isDir(), is(true));
    assertThat(stat.getLen(), is(0L));
    assertThat(stat.getModificationTime(),
        is(lessThanOrEqualTo(System.currentTimeMillis())));
    assertThat(stat.getReplication(), is((short) 0));
    assertThat(stat.getBlockSize(), is(0L));
    assertThat(stat.getOwner(), is("tom"));
    assertThat(stat.getGroup(), is("supergroup"));
    assertThat(stat.getPermission().toString(), is("rwxr-xr-x"));
  }

}
```

If no file or directory exists, a `FileNotFoundException` is thrown. However, if you are interested only in the existence of a file or directory, the `exists()` method on `FileSystem` is more convenient:

```
public boolean exists(Path f) throws IOException
```

Listing files

Finding information on a single file or directory is useful, but you also often need to be able to list the contents of a directory. That's what `FileSystem`'s `listStatus()` methods are for:

```
public FileStatus[] listStatus(Path f) throws IOException
public FileStatus[] listStatus(Path f, PathFilter filter) throws IOException
public FileStatus[] listStatus(Path[] files) throws IOException
public FileStatus[] listStatus(Path[] files, PathFilter filter) throws IOException
```

When the argument is a file, the simplest variant returns an array of `FileStatus` objects of length 1. When the argument is a directory, it returns zero or more `FileStatus` objects representing the files and directories contained in the directory.

Overloaded variants allow a `PathFilter` to be supplied to restrict the files and directories to match. You will see an example of this in the section "PathFilter" on page 66. Finally, if you specify an array of paths, the result is a shortcut for calling the equivalent single-path `listStatus` method for each path in turn and accumulating the `FileStatus` object arrays in a single array. This can be useful for building up lists of input files to process from distinct parts of the filesystem tree. Example 3-6 is a simple demonstration of this idea. Note the use of `stat2Paths()` in `FileUtil` for turning an array of `FileStatus` objects to an array of `Path` objects.

Example 3-6. Showing the file statuses for a collection of paths in a Hadoop filesystem

```
public class ListStatus {

  public static void main(String[] args) throws Exception {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);

    Path[] paths = new Path[args.length];
    for (int i = 0; i < paths.length; i++) {
      paths[i] = new Path(args[i]);
    }

    FileStatus[] status = fs.listStatus(paths);
    Path[] listedPaths = FileUtil.stat2Paths(status);
    for (Path p : listedPaths) {
      System.out.println(p);
    }
  }
}
```

We can use this program to find the union of directory listings for a collection of paths:

```
% hadoop ListStatus hdfs://localhost/ hdfs://localhost/user/tom
hdfs://localhost/user
hdfs://localhost/user/tom/books
hdfs://localhost/user/tom/quangle.txt
```

File patterns

It is a common requirement to process sets of files in a single operation. For example, a MapReduce job for log processing might analyze a month's worth of files contained in a number of directories. Rather than having to enumerate each file and directory to specify the input, it is convenient to use wildcard characters to match multiple files with a single expression, an operation that is known as *globbing*. Hadoop provides two `FileSystem` method for processing globs:

```
public FileStatus[] globStatus(Path pathPattern) throws IOException
public FileStatus[] globStatus(Path pathPattern, PathFilter filter) throws
  IOException
```

The `globStatus()` method returns an array of `FileStatus` objects whose paths match the supplied pattern, sorted by path. An optional `PathFilter` can be specified to restrict the matches further.

Hadoop supports the same set of glob characters as Unix *bash* (see Table 3-2).

Table 3-2. Glob characters and their meanings

Glob	Name	Matches
*	*asterisk*	Matches zero or more characters
?	*question mark*	Matches a single character
[ab]	*character class*	Matches a single character in the set {a, b}
[^ab]	*negated character class*	Matches a single character that is not in the set {a, b}
[a-b]	*character range*	Matches a single character in the (closed) range [a, b], where a is lexicographically less than or equal to b
[^a-b]	*negated character range*	Matches a single character that is not in the (closed) range [a, b], where a is lexicographically less than or equal to b
{a,b}	*alternation*	Matches either expression a or b
\c	*escaped character*	Matches character c when it is a metacharacter

Imagine that logfiles are stored in a directory structure organized hierarchically by date. So, for example, logfiles for the last day of 2007 would go in a directory named */2007/12/31*. Suppose that the full file listing is:

```
/
├── 2007/
│   └── 12/
│       ├── 30/
│       └── 31/
└── 2008/
    └── 01/
        ├── 01/
        └── 02/
```

Here are some file globs and their expansions:

Glob	Expansion
/*	*/2007 /2008*
/*/*	*/2007/12 /2008/01*
/*/12/*	*/2007/12/30 /2007/12/31*
/200?	*/2007 /2008*
/200[78]	*/2007 /2008*
/200[7-8]	*/2007 /2008*
/200[^01234569]	*/2007 /2008*
/*/*/{31,01}	*/2007/12/31 /2008/01/01*
/*/*/3{0,1}	*/2007/12/30 /2007/12/31*
/*/{12/31,01/01}	*/2007/12/31 /2008/01/01*

PathFilter

Glob patterns are not always powerful enough to describe a set of files you want to access. For example, it is not generally possible to exclude a particular file using a glob pattern. The `listStatus()` and `globStatus()` methods of `FileSystem` take an optional `PathFilter`, which allows programmatic control over matching:

```
package org.apache.hadoop.fs;

public interface PathFilter {
  boolean accept(Path path);
}
```

`PathFilter` is the equivalent of `java.io.FileFilter` for `Path` objects rather than `File` objects.

Example 3-7 shows a `PathFilter` for excluding paths that match a regular expression.

Example 3-7. A PathFilter for excluding paths that match a regular expression

```
public class RegexExcludePathFilter implements PathFilter {

  private final String regex;

  public RegexExcludePathFilter(String regex) {
    this.regex = regex;
  }

  public boolean accept(Path path) {
    return !path.toString().matches(regex);
  }
}
```

The filter passes only those files that *don't* match the regular expression. After the glob picks out an initial set of files to include, the filter is used to refine the results. For example:

```
fs.globStatus(new Path("/2007/*/*"), new RegexExcludeFilter("^.*/2007/12/31$"))
```

will expand to */2007/12/30*.

Filters can act only on a file's name, as represented by a `Path`. They can't use a file's properties, such as creation time, as the basis of the filter. Nevertheless, they can perform matching that neither glob patterns nor regular expressions can achieve. For example, if you store files in a directory structure that is laid out by date (like in the previous section), you can write a `PathFilter` to pick out files that fall in a given date range.

Deleting Data

Use the `delete()` method on `FileSystem` to permanently remove files or directories:

```
public boolean delete(Path f, boolean recursive) throws IOException
```

If `f` is a file or an empty directory, the value of `recursive` is ignored. A nonempty directory is deleted, along with its contents, only if `recursive` is `true` (otherwise, an `IOException` is thrown).

Data Flow

Anatomy of a File Read

To get an idea of how data flows between the client interacting with HDFS, the namenode, and the datanodes, consider Figure 3-2, which shows the main sequence of events when reading a file.

The client opens the file it wishes to read by calling `open()` on the `FileSystem` object, which for HDFS is an instance of `DistributedFileSystem` (step 1 in Figure 3-2). `DistributedFileSystem` calls the namenode, using RPC, to determine the locations of the blocks for the first few blocks in the file (step 2). For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Furthermore, the datanodes are sorted according to their proximity to the client (according to the topology of the cluster's network; see "Network Topology and Hadoop" on page 69). If the client is itself a datanode (in the case of a MapReduce task, for instance), the client will read from the local datanode if that datanode hosts a copy of the block (see also Figure 2-2).

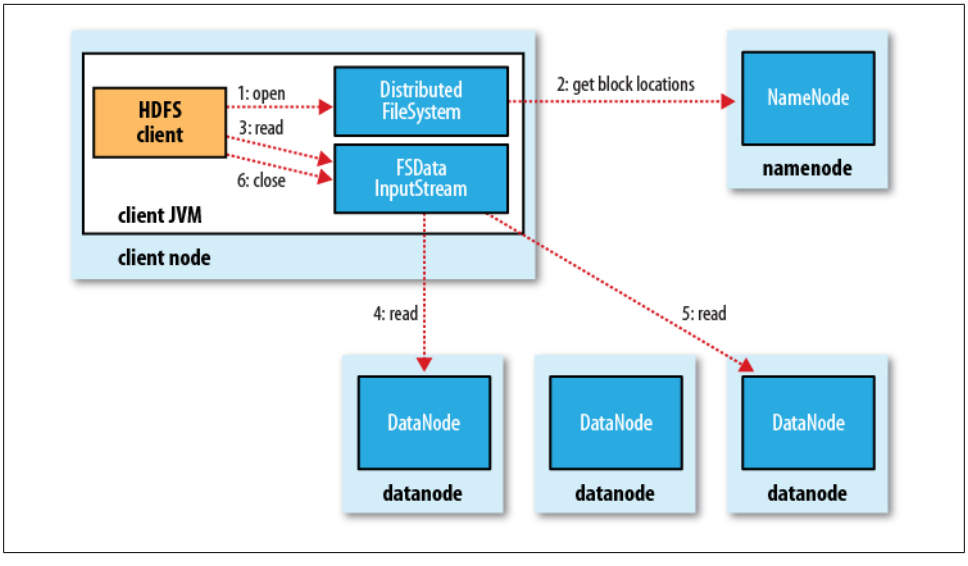


Figure 3-2. A client reading data from HDFS

The `DistributedFileSystem` returns an `FSDataInputStream` (an input stream that supports file seeks) to the client for it to read data from. `FSDataInputStream` in turn wraps a `DFSInputStream`, which manages the datanode and namenode I/O.

The client then calls `read()` on the stream (step 3). `DFSInputStream`, which has stored the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file. Data is streamed from the datanode back to the client, which calls `read()` repeatedly on the stream (step 4). When the end of the block is reached, `DFSInputStream` will close the connection to the datanode, then find the best datanode for the next block (step 5). This happens transparently to the client, which from its point of view is just reading a continuous stream.

Blocks are read in order, with the `DFSInputStream` opening new connections to datanodes as the client reads through the stream. It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed. When the client has finished reading, it calls `close()` on the `FSDataInputStream` (step 6).

During reading, if the `DFSInputStream` encounters an error while communicating with a datanode, it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesn't needlessly retry them for later blocks. The `DFSInputStream` also verifies checksums for the data transferred to it from the datanode. If a corrupted block is found, it is reported to the namenode before the `DFSInputStream` attempts to read a replica of the block from another datanode.

One important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block. This design allows HDFS to scale to a large number of concurrent clients because the data

traffic is spread across all the datanodes in the cluster. Meanwhile, the namenode merely has to service block location requests (which it stores in memory, making them very efficient) and does not, for example, serve data, which would quickly become a bottleneck as the number of clients grew.

Network Topology and Hadoop

What does it mean for two nodes in a local network to be "close" to each other? In the context of high-volume data processing, the limiting factor is the rate at which we can transfer data between nodes—bandwidth is a scarce commodity. The idea is to use the bandwidth between two nodes as a measure of distance.

Rather than measuring bandwidth between nodes, which can be difficult to do in practice (it requires a quiet cluster, and the number of pairs of nodes in a cluster grows as the square of the number of nodes), Hadoop takes a simple approach in which the network is represented as a tree and the distance between two nodes is the sum of their distances to their closest common ancestor. Levels in the tree are not predefined, but it is common to have levels that correspond to the data center, the rack, and the node that a process is running on. The idea is that the bandwidth available for each of the following scenarios becomes progressively less:

- Processes on the same node
- Different nodes on the same rack
- Nodes on different racks in the same data center
- Nodes in different data centers[8]

For example, imagine a node *n1* on rack *r1* in data center *d1*. This can be represented as */d1/r1/n1*. Using this notation, here are the distances for the four scenarios:

- *distance(/d1/r1/n1, /d1/r1/n1)* = 0 (processes on the same node)
- *distance(/d1/r1/n1, /d1/r1/n2)* = 2 (different nodes on the same rack)
- *distance(/d1/r1/n1, /d1/r2/n3)* = 4 (nodes on different racks in the same data center)
- *distance(/d1/r1/n1, /d2/r3/n4)* = 6 (nodes in different data centers)

This is illustrated schematically in Figure 3-3. (Mathematically inclined readers will notice that this is an example of a distance metric.)

Finally, it is important to realize that Hadoop cannot divine your network topology for you. It needs some help, we'll cover how to configure topology in "Network Topology" on page 299. By default, though, it assumes that the network is flat—a single-level hierarchy—or in other words, that all nodes are on a single rack in a single data center. For small clusters, this may actually be the case, and no further configuration is required.

8. At the time of this writing, Hadoop is not suited for running across data centers.

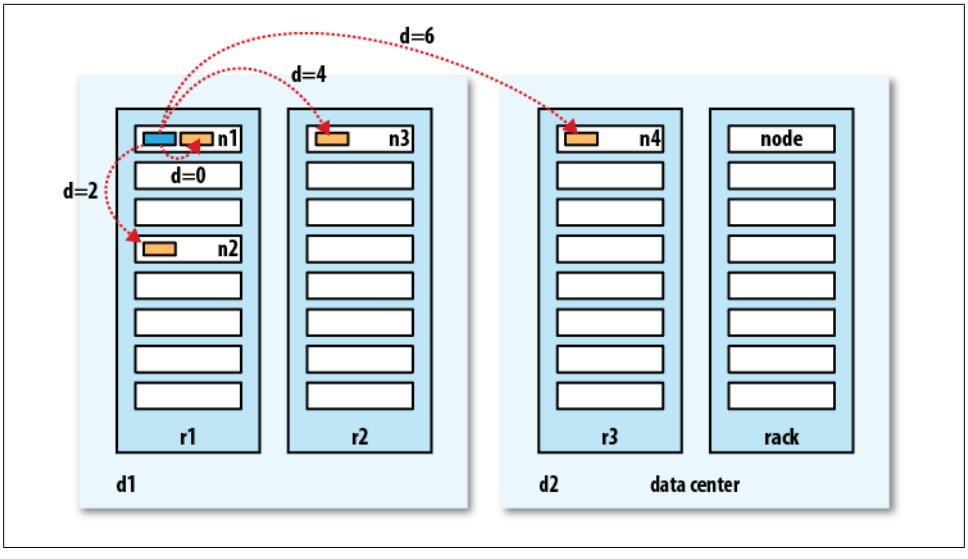


Figure 3-3. Network distance in Hadoop

Anatomy of a File Write

Next we'll look at how files are written to HDFS. Although quite detailed, it is instructive to understand the data flow because it clarifies HDFS's coherency model.

We're going to consider the case of creating a new file, writing data to it, then closing the file. See Figure 3-4.

The client creates the file by calling `create()` on `DistributedFileSystem` (step 1 in Figure 3-4). `DistributedFileSystem` makes an RPC call to the namenode to create a new file in the filesystem's namespace, with no blocks associated with it (step 2). The namenode performs various checks to make sure the file doesn't already exist and that the client has the right permissions to create the file. If these checks pass, the namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an `IOException`. The `DistributedFileSystem` returns an `FSDataOutputStream` for the client to start writing data to. Just as in the read case, `FSDataOutputStream` wraps a `DFSOutputStream`, which handles communication with the datanodes and namenode.

As the client writes data (step 3), `DFSOutputStream` splits it into packets, which it writes to an internal queue, called the *data queue*. The data queue is consumed by the `DataStreamer`, which is responsible for asking the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas. The list of datanodes forms a pipeline, and here we'll assume the replication level is three, so there are three nodes in the pipeline. The `DataStreamer` streams the packets to the first datanode in the pipeline, which stores the packet and forwards it to the second datanode in the pipeline.

Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline (step 4).

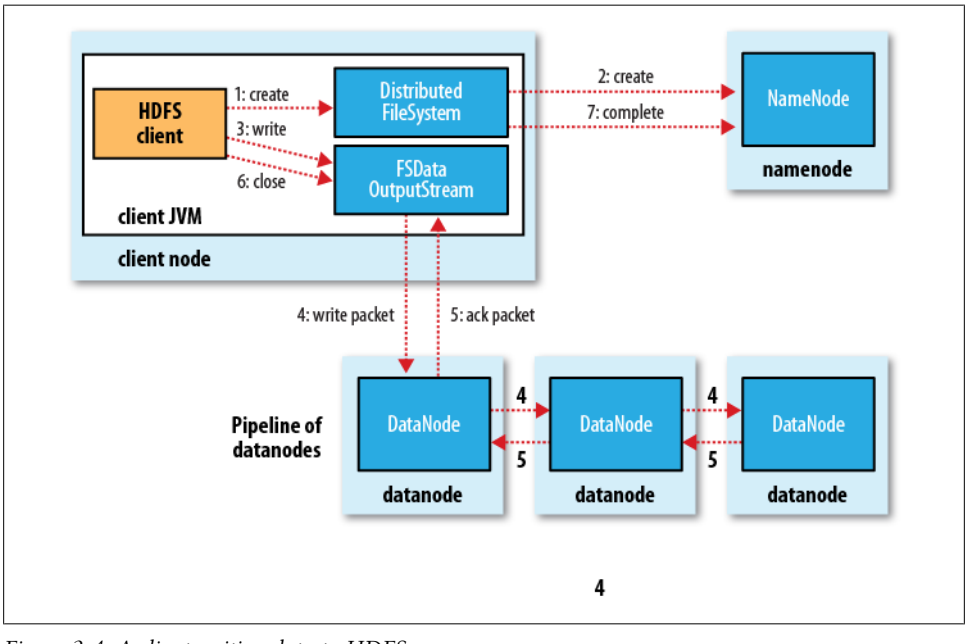


Figure 3-4. A client writing data to HDFS

`DFSOutputStream` also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the *ack queue*. A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline (step 5).

If a datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data. First, the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets. The current block on the good datanodes is given a new identity, which is communicated to the namenode, so that the partial block on the failed datanode will be deleted if the failed datanode recovers later on. The failed datanode is removed from the pipeline, and the remainder of the block's data is written to the two good datanodes in the pipeline. The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal.

It's possible, but unlikely, that multiple datanodes fail while a block is being written. As long as `dfs.replication.min` replicas (which default to one) are written, the write will succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached (`dfs.replication`, which defaults to three).

When the client has finished writing data, it calls `close()` on the stream (step 6). This action flushes all the remaining packets to the datanode pipeline and waits for ac-

knowledgments before contacting the namenode to signal that the file is complete (step 7). The namenode already knows which blocks the file is made up of (via `DataStreamer` asking for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

Replica Placement

How does the namenode choose which datanodes to store replicas on? There's a trade-off between reliability and write bandwidth and read bandwidth here. For example, placing all replicas on a single node incurs the lowest write bandwidth penalty since the replication pipeline runs on a single node, but this offers no real redundancy (if the node fails, the data for that block is lost). Also, the read bandwidth is high for off-rack reads. At the other extreme, placing replicas in different data centers may maximize redundancy, but at the cost of bandwidth. Even in the same data center (which is what all Hadoop clusters to date have run in), there are a variety of placement strategies. Indeed, Hadoop changed its placement strategy in release 0.17.0 to one that helps keep a fairly even distribution of blocks across the cluster. (See "Balancer" on page 350 for details on keeping a cluster balanced.) And in releases after 1.x, block placement policies are pluggable.

Hadoop's default strategy is to place the first replica on the same node as the client (for clients running outside the cluster, a node is chosen at random, although the system tries not to pick nodes that are too full or too busy). The second replica is placed on a different rack from the first (*off-rack*), chosen at random. The third replica is placed on the same rack as the second, but on a different node chosen at random. Further replicas are placed on random nodes on the cluster, although the system tries to avoid placing too many replicas on the same rack.

Once the replica locations have been chosen, a pipeline is built, taking network topology into account. For a replication factor of 3, the pipeline might look like Figure 3-5.

Overall, this strategy gives a good balance among reliability (blocks are stored on two racks), write bandwidth (writes only have to traverse a single network switch), read performance (there's a choice of two racks to read from), and block distribution across the cluster (clients only write a single block on the local rack).

Coherency Model

A coherency model for a filesystem describes the data visibility of reads and writes for a file. HDFS trades off some POSIX requirements for performance, so some operations may behave differently than you expect them to.

After creating a file, it is visible in the filesystem namespace, as expected:

```
Path p = new Path("p");
fs.create(p);
assertThat(fs.exists(p), is(true));
```

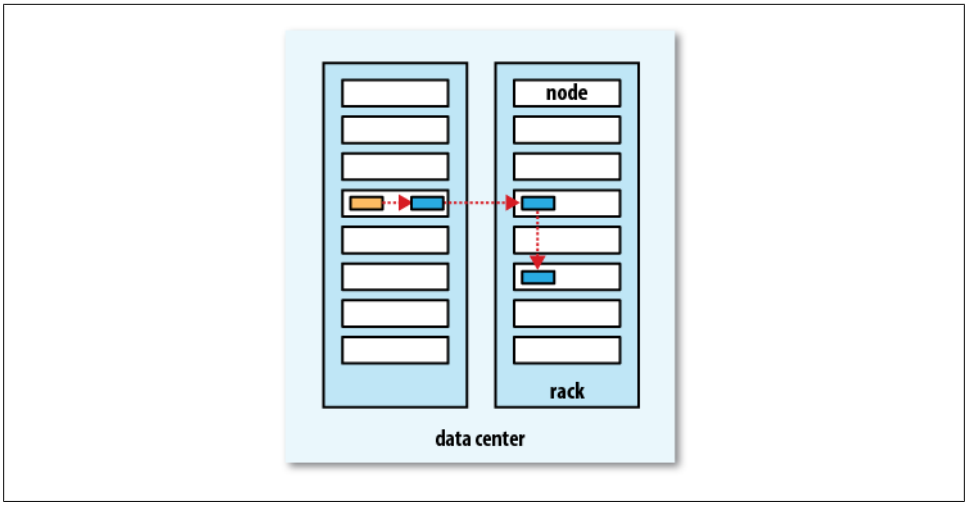


Figure 3-5. A typical replica pipeline

However, any content written to the file is not guaranteed to be visible, even if the stream is flushed. So the file appears to have a length of zero:

```
Path p = new Path("p");
OutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.flush();
assertThat(fs.getFileStatus(p).getLen(), is(0L));
```

Once more than a block's worth of data has been written, the first block will be visible to new readers. This is true of subsequent blocks, too: it is always the current block being written that is not visible to other readers.

HDFS provides a method for forcing all buffers to be synchronized to the datanodes via the `sync()` method on `FSDataOutputStream`. After a successful return from `sync()`, HDFS guarantees that the data written up to that point in the file has reached all the datanodes in the write pipeline and is visible to all new readers:[9]

```
Path p = new Path("p");
FSDataOutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.flush();
out.sync();
assertThat(fs.getFileStatus(p).getLen(), is(((long) "content".length())));
```

9. Post Hadoop 1.x, `sync()` is deprecated in favor of the equivalent `hflush()` method. Another method, `hsync()`, has also been added that makes a stronger guarantee that the operating system has flushed the data to the datanodes' disks (like POSIX `fsync`). However, at the time of this writing, this has not been implemented and merely calls `hflush()`.

This behavior is similar to the `fsync` system call in POSIX that commits buffered data for a file descriptor. For example, using the standard Java API to write a local file, we are guaranteed to see the content after flushing the stream and synchronizing:

```
FileOutputStream out = new FileOutputStream(localFile);
out.write("content".getBytes("UTF-8"));
out.flush(); // flush to operating system
out.getFD().sync(); // sync to disk
assertThat(localFile.length(), is(((long) "content".length())));
```

Closing a file in HDFS performs an implicit `sync()`, too:

```
Path p = new Path("p");
OutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.close();
assertThat(fs.getFileStatus(p).getLen(), is(((long) "content".length())));
```

Consequences for application design

This coherency model has implications for the way you design applications. With no calls to `sync()`, you should be prepared to lose up to a block of data in the event of client or system failure. For many applications, this is unacceptable, so you should call `sync()` at suitable points, such as after writing a certain number of records or number of bytes. Though the `sync()` operation is designed to not unduly tax HDFS, it does have some overhead, so there is a trade-off between data robustness and throughput. What constitutes an acceptable trade-off is application-dependent, and suitable values can be selected after measuring your application's performance with different `sync()` frequencies.

Data Ingest with Flume and Sqoop

Rather than writing an application to move data into HDFS, it's worth considering some of the existing tools for ingesting data because they cover many of the common requirements.

Apache Flume (*http://incubator.apache.org/flume/*) is a system for moving large quantities of streaming data into HDFS. A very common use case is collecting log data from one system—a bank of web servers, for example—and aggregating it in HDFS for later analysis. Flume supports a large variety of sources; some of the more commonly used ones include `tail` (which pipes data from a local file being written to into Flume, just like Unix `tail`), syslog, and Apache log4j (allowing Java applications to write events to files in HDFS via Flume).

Flume nodes can be arranged in arbitrary topologies. Typically there is a node running on each source machine (each web server, for example), with tiers of aggregating nodes that the data flows through on its way to HDFS.

Flume offers different levels of delivery reliability, from *best-effort* delivery, which doesn't tolerate any Flume node failures, to *end-to-end*, which guarantees delivery even in the event of multiple Flume node failures between the source and HDFS.

Apache Sqoop (*http://sqoop.apache.org/*), on the other hand, is designed for performing bulk imports of data into HDFS from structured data stores, such as relational databases. An example of a Sqoop use case is an organization that runs a nightly Sqoop import to load the day's data from a production database into a Hive data warehouse for analysis. Sqoop is covered in Chapter 15.

Parallel Copying with distcp

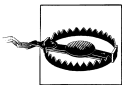
The HDFS access patterns that we have seen so far focus on single-threaded access. It's possible to act on a collection of files—by specifying file globs, for example—but for efficient parallel processing of these files, you would have to write a program yourself. Hadoop comes with a useful program called *distcp* for copying large amounts of data to and from Hadoop filesystems in parallel.

The canonical use case for *distcp* is for transferring data between two HDFS clusters. If the clusters are running identical versions of Hadoop, the *hdfs* scheme is appropriate:

```
% hadoop distcp hdfs://namenode1/foo hdfs://namenode2/bar
```

This will copy the */foo* directory (and its contents) from the first cluster to the */bar* directory on the second cluster, so the second cluster ends up with the directory structure */bar/foo*. If */bar* doesn't exist, it will be created first. You can specify multiple source paths, and all will be copied to the destination. Source paths must be absolute.

By default, *distcp* will skip files that already exist in the destination, but they can be overwritten by supplying the `-overwrite` option. You can also update only the files that have changed using the `-update` option.



Using either (or both) `-overwrite` or `-update` changes how the source and destination paths are interpreted. This is best shown with an example. If we changed a file in the */foo* subtree on the first cluster from the previous example, we could synchronize the change with the second cluster by running:

```
% hadoop distcp -update hdfs://namenode1/foo hdfs://namenode2/bar/foo
```

The extra trailing */foo* subdirectory is needed on the destination, because now the *contents* of the source directory are copied to the *contents* of the destination directory. (If you are familiar with *rsync*, you can think of the `-overwrite` or `-update` options as adding an implicit trailing slash to the source.)

If you are unsure of the effect of a *distcp* operation, it is a good idea to try it out on a small test directory tree first.

There are more options to control the behavior of *distcp*, including ones to preserve file attributes, ignore failures, and limit the number of files or total data copied. Run it with no options to see the usage instructions.

distcp is implemented as a MapReduce job where the work of copying is done by the maps that run in parallel across the cluster. There are no reducers. Each file is copied by a single map, and *distcp* tries to give each map approximately the same amount of data by bucketing files into roughly equal allocations.

The number of maps is decided as follows. Because it's a good idea to get each map to copy a reasonable amount of data to minimize overheads in task setup, each map copies at least 256 MB (unless the total size of the input is less, in which case one map handles it all). For example, 1 GB of files will be given four map tasks. When the data size is very large, it becomes necessary to limit the number of maps in order to limit bandwidth and cluster utilization. By default, the maximum number of maps is 20 per (tasktracker) cluster node. For example, copying 1,000 GB of files to a 100-node cluster will allocate 2,000 maps (20 per node), so each will copy 512 MB on average. This can be reduced by specifying the `-m` argument to *distcp*. For example, `-m 1000` would allocate 1,000 maps, each copying 1 GB on average.

When you try to use *distcp* between two HDFS clusters that are running different versions, the copy will fail if you use the *hdfs* protocol because the RPC systems are incompatible. To remedy this, you can use the read-only HTTP-based HFTP filesystem to read from the source. The job must run on the destination cluster so that the HDFS RPC versions are compatible. To repeat the previous example using HFTP:

```
% hadoop distcp hftp://namenode1:50070/foo hdfs://namenode2/bar
```

Note that you need to specify the namenode's web port in the source URI. This is determined by the `dfs.http.address` property, which defaults to 50070.

Using the newer *webhdfs* protocol (which replaces *hftp*), it is possible to use HTTP for both the source and destination clusters without hitting any wire incompatibility problems.

```
% hadoop distcp webhdfs://namenode1:50070/foo webhdfs://namenode2:50070/bar
```

Another variant is to use an HDFS HTTP proxy as the *distcp* source or destination, which has the advantage of being able to set firewall and bandwidth controls (see "HTTP" on page 53).

Keeping an HDFS Cluster Balanced

When copying data into HDFS, it's important to consider cluster balance. HDFS works best when the file blocks are evenly spread across the cluster, so you want to ensure that *distcp* doesn't disrupt this. Going back to the 1,000 GB example, by specifying `-m 1`, a single map would do the copy, which—apart from being slow and not using the cluster resources efficiently—would mean that the first replica of each block would

reside on the node running the map (until the disk filled up). The second and third replicas would be spread across the cluster, but this one node would be unbalanced. By having more maps than nodes in the cluster, this problem is avoided. For this reason, it's best to start by running *distcp* with the default of 20 maps per node.

However, it's not always possible to prevent a cluster from becoming unbalanced. Perhaps you want to limit the number of maps so that some of the nodes can be used by other jobs. In this case, you can use the *balancer* tool (see "Balancer" on page 350) to subsequently even out the block distribution across the cluster.

Hadoop Archives

HDFS stores small files inefficiently, since each file is stored in a block, and block metadata is held in memory by the namenode. Thus, a large number of small files can eat up a lot of memory on the namenode. (Note, however, that small files do not take up any more disk space than is required to store the raw contents of the file. For example, a 1 MB file stored with a block size of 128 MB uses 1 MB of disk space, not 128 MB.)

Hadoop Archives, or HAR files, are a file archiving facility that packs files into HDFS blocks more efficiently, thereby reducing namenode memory usage while still allowing transparent access to files. In particular, Hadoop Archives can be used as input to MapReduce.

Using Hadoop Archives

A Hadoop Archive is created from a collection of files using the *archive* tool. The tool runs a MapReduce job to process the input files in parallel, so to run it, you need a running MapReduce cluster to use it. Here are some files in HDFS that we would like to archive:

```
% hadoop fs -lsr /my/files
-rw-r--r--   1 tom supergroup          1 2009-04-09 19:13 /my/files/a
drwxr-xr-x   - tom supergroup          0 2009-04-09 19:13 /my/files/dir
-rw-r--r--   1 tom supergroup          1 2009-04-09 19:13 /my/files/dir/b
```

Now we can run the `archive` command:

```
% hadoop archive -archiveName files.har /my/files /my
```

The first option is the name of the archive, here *files.har*. HAR files always have a *.har* extension, which is mandatory for reasons we shall see later. Next come the files to put in the archive. Here we are archiving only one source tree, the files in */my/files* in HDFS, but the tool accepts multiple source trees. The final argument is the output directory for the HAR file. Let's see what the archive has created:

```
% hadoop fs -ls /my
Found 2 items
drwxr-xr-x   - tom supergroup          0 2009-04-09 19:13 /my/files
```

```
drwxr-xr-x   - tom supergroup          0 2009-04-09 19:13 /my/files.har
% hadoop fs -ls /my/files.har
Found 3 items
-rw-r--r--  10 tom supergroup        165 2009-04-09 19:13 /my/files.har/_index
-rw-r--r--  10 tom supergroup         23 2009-04-09 19:13 /my/files.har/_masterindex
-rw-r--r--   1 tom supergroup          2 2009-04-09 19:13 /my/files.har/part-0
```

The directory listing shows what a HAR file is made of: two index files and a collection of part files (this example has just one of the latter). The part files contain the contents of a number of the original files concatenated together, and the indexes make it possible to look up the part file that an archived file is contained in, as well as its offset and length. All these details are hidden from the application, however, which uses the *har* URI scheme to interact with HAR files, using a HAR filesystem that is layered on top of the underlying filesystem (HDFS in this case). The following command recursively lists the files in the archive:

```
% hadoop fs -lsr har:///my/files.har
drw-r--r--   - tom supergroup        0 2009-04-09 19:13 /my/files.har/my
drw-r--r--   - tom supergroup        0 2009-04-09 19:13 /my/files.har/my/files
-rw-r--r--  10 tom supergroup        1 2009-04-09 19:13 /my/files.har/my/files/a
drw-r--r--   - tom supergroup        0 2009-04-09 19:13 /my/files.har/my/files/dir
-rw-r--r--  10 tom supergroup        1 2009-04-09 19:13 /my/files.har/my/files/dir/b
```

This is quite straightforward when the filesystem that the HAR file is on is the default filesystem. On the other hand, if you want to refer to a HAR file on a different filesystem, you need to use a different form of the path URI. These two commands have the same effect, for example:

```
% hadoop fs -lsr har:///my/files.har/my/files/dir
% hadoop fs -lsr har://hdfs-localhost:8020/my/files.har/my/files/dir
```

Notice in the second form that the scheme is still *har* to signify a HAR filesystem, but the authority is *hdfs* to specify the underlying filesystem's scheme, followed by a dash and the HDFS host (localhost) and port (8020). We can now see why HAR files must have a *.har* extension. The HAR filesystem translates the *har* URI into a URI for the underlying filesystem by looking at the authority and path up to and including the component with the *.har* extension. In this case, it is *hdfs://localhost:8020/my/files.har*. The remaining part of the path is the path of the file in the archive: */my/files/dir*.

To delete a HAR file, you need to use the recursive form of delete because from the underlying filesystem's point of view, the HAR file is a directory:

```
% hadoop fs -rmr /my/files.har
```

Limitations

There are a few limitations to be aware of with HAR files. Creating an archive creates a copy of the original files, so you need as much disk space as the files you are archiving to create the archive (although you can delete the originals once you have created the archive). There is currently no support for archive compression, although the files that go into the archive can be compressed (HAR files are like *tar* files in this respect).

Archives are immutable once they have been created. To add or remove files, you must re-create the archive. In practice, this is not a problem for files that don't change after being written, since they can be archived in batches on a regular basis, such as daily or weekly.

As noted earlier, HAR files can be used as input to MapReduce. However, there is no archive-aware `InputFormat` that can pack multiple files into a single MapReduce split, so processing lots of small files, even in a HAR file, can still be inefficient. "Small files and CombineFileInputFormat" on page 239 discusses another approach to this problem.

Finally, if you are hitting namenode memory limits even after taking steps to minimize the number of small files in the system, consider using HDFS Federation to scale the namespace (see "HDFS Federation" on page 47).

About the Author

Tom White is one of the foremost experts on Hadoop. He has been an Apache Hadoop committer since February 2007, and is a Member of the Apache Software Foundation. Tom is a software engineer at Cloudera, where he has worked since its foundation, on the core distributions from Apache and Cloudera. Previously he was an independent Hadoop consultant, working with companies to set up, use, and extend Hadoop. He has written numerous articles for O'Reilly, java.net and IBM's developerWorks, and has spoken at many conferences, including ApacheCon and OSCON. Tom has a B.A. in mathematics from the University of Cambridge and an M.A. in philosophy of science from the University of Leeds, UK. He currently lives in San Francisco with his family.

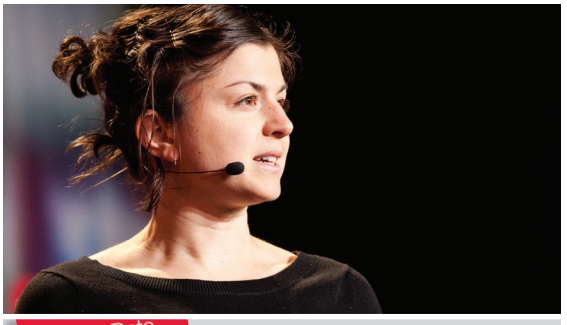
Colophon

The animal on the cover of *Hadoop: The Definitive Guide* is an African elephant. These members of the genus *Loxodonta* are the largest land animals on earth (slightly larger than their cousin, the Asian elephant) and can be identified by their ears, which have been said to look somewhat like the continent of Asia. Males stand 12 feet tall at the shoulder and weigh 12,000 pounds, but they can get as big as 15,000 pounds, whereas females stand 10 feet tall and weigh 8,000–11,000 pounds. Even young elephants are very large: at birth, they already weigh approximately 200 pounds and stand about 3 feet tall.

African elephants live throughout sub-Saharan Africa. Most of the continent's elephants live on savannas and in dry woodlands. In some regions, they can be found in desert areas; in others, they are found in mountains.

The species plays an important role in the forest and savanna ecosystems in which they live. Many plant species are dependent on passing through an elephant's digestive tract before they can germinate; it is estimated that at least a third of tree species in west African forests rely on elephants in this way. Elephants grazing on vegetation also affect the structure of habitats and influence bush fire patterns. For example, under natural conditions, elephants make gaps through the rainforest, enabling the sunlight to enter, which allows the growth of various plant species. This, in turn, facilitates more abundance and more diversity of smaller animals. As a result of the influence elephants have over many plants and animals, they are often referred to as a *keystone species* because they are vital to the long-term survival of the ecosystems in which they live.

The cover image is from the Dover Pictorial Archive. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSansMonoCondensed.



O'REILLY®

Strata

Making Data Work

O'Reilly Strata is the essential source for training and information in data science and big data—with industry news, reports, in-person and online events, and much more.

- Weekly Newsletter
- Industry News & Commentary
- Free Reports
- Webcasts
- Conferences
- Books & Videos

Dive deep into the latest in data science and big data.

strata.oreilly.com