Module – 3 THE HADOOP DISTRIBUTED FILE SYSTEM

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.

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, 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 to run on. It’s designed to run on clusters of commodity hardware (commonly available hardware available 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. While 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 is currently a better choice for low-latency access.

Since 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. While storing millions of files is feasible, billions is be- yond the capability of current hardware.

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.)

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, while 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.

Having ablockabstractionforadistributedfilesystembringsseveralbenefits. Thefirst 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 all 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 (since blocks are a fixed size, it is easy to calculate how many can be stored on a given disk) and eliminating metadata concerns (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 trans- parent 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. 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.

NAMENODES AND DATANODES

An HDFS cluster has two types of node operating in a master-worker pattern: a name- node (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, since this information is reconstructed from datanodes when the system starts.

A client accesses the filesystem on behalf of the user by communicating with the name- node and datanodes. The client presents a POSIX-like filesystem interface, 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 configu- ration 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, since it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged name- space 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.

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. HDFS Federation, introduced in the 0.23 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 namen- odes. 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 the ViewFileSystem, and 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 does not provide high-availability of the filesystem. The namenode is still a single point of fail- ure (SPOF), since if it did fail, all clients—including MapReduce jobs—would be un- able 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 da- tanodes 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, since unex- pected failure of the namenode is so rare, the case for planned downtime is actually more important in practice.

The 0.23 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 inter- ruption. 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 Zoo- Keeper.) 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 since the block mappings are stored in a namenode's memory, and not on disk.

Clients must be configured to handle namenode failover, which uses a mechanism that is transparent to users.

If the active namenode fails, then the standby can take over very quickly (in a few tens of seconds) since 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), since 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, since the process is a standard op- erational 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 adminstrator, in the case of routine maintenance, for example. This is known as a graceful failover, since the failover con- troller 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 com- mand), 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 graphi- cally 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 which is mapped to a pair of namenode addresses (in the configuration file), and the client library tries each namenode address until the operation succeeds.

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 direc- tories. 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.txtis copiedtothefile/user/tom/quangle.txtonthe HDFSinstancerunning 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 could also 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  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 since 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

HADOOP FILESYSTEMS

Hadoop has an abstract notion of filesystem, of which HDFS is just one implementation. The Java abstract class org.apache. hadoop .fs. File System represents a filesystem in Hadoop, and there are several concrete implementations

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 filesys- tems. 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.

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 under- lying 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 Distributed File System API.

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, while 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, while the new

WebHDFS implementation supports all filesystem operations, including Kerberos authentication. Web HDFS must be enabled by setting dfs. web hdfs. enabled to true, for you to be able to use webhdfs URIs.

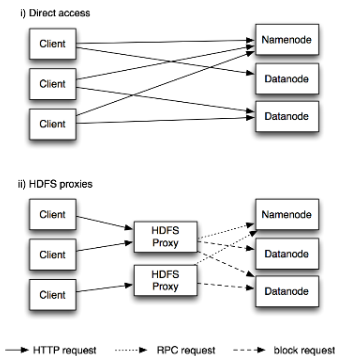

Figure 3-1. Accessing HDFS over HTTP directly, and via a bank of HDFS proxies

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 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 ac- cessed by clients using the HSFTP FileSystem implementation (hsftp URIs). From re- lease 0.23, there is a new proxy called HttpFS that has read and write capabilities, and which exposes the same HTTP interface as WebHDFS, so clients can access either using webhdfs URIs.

The HTTP REST API that WebHDFS exposes is formally defined in a specification, so it is likely 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 JAVAINTERFACE

In this section, we dig into the Hadoop’s FileSystem class: the API for interacting with one of Hadoop’s filesystems.[5] While 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, since 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 {

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

1. From release 0.21.0, 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.
2. with an instance of Fs Url Stream Handler Factory. This method can only be called 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 URL Stream Handler Factory, you won't be able to use this approach for reading data from Hadoop. The next section discusses an alternative.

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 {

3.
static {
URL.setURLStreamHandlerFactory(new    FsUrlStreamHandlerFactory());
}

4.
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.

READING DATA USING THE FILESYSTEM API

As the previous section explained, sometimes it is impossible to set a URLStreamHand lerFactory 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) throwsIOException
```

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.

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 K.

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

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);

}

}

}
```

FS Data Input Stream

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 thecurrent 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 anIOException. Unlike the skip() method of java.io.InputStream that positions the stream at

a point later than the current position, seek() can move to an arbitrary, ab- solute positionin the file.

Example 3-3 is a simple extension of Example 3-2 that writes a file to standard out twice: afterwriting 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 seekpublic

```
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:

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) 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() methodswillreadlength bytesintothebuffer(orbuffer.length bytesfor theversion

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, so they provide a convenient way to access another part of the file—metadata perhaps—while reading the main body of the file. In fact, they are just implemented using the Seekable interface using the following pattern:

```
long oldPos = getPos(); try {

seek(position);

// read data

} finally { seek(oldPos);

}
```

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 File System 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 FS Data Output Stream 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.

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 theappend()

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 a restart, for example. The append operation is optional and not implemented by all Hadoop filesystems. For example, HDFS supports append, but S3 filesystems don't.

To copy a local file to a Hadoop filesystem. We illustrate pro- gress by printing a period every time the progress() method is called by Hadoop, which is after each 64 K 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 filesystempublic 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

FS Data Output Stream

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 {

// implementation elided

}

// implementation elided

}
```

However, unlike FS Data Input Stream, FS Data Output Stream 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.

DATA FLOW

ANATOMY OF A FILE READ

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

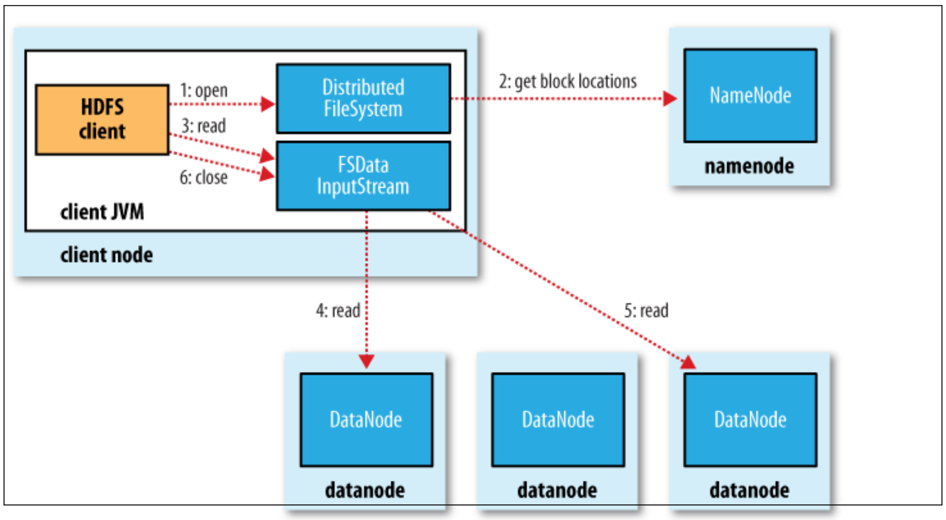

Figure 3-2. A client reading data from HDFS

The client opens the file it wishes to read by calling open() on the File System object, which for HDFS is an instance of Distributed File System Distributed File System 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 top- ology of the cluster's network; see "Network Topology and Hadoop"). If the client is itself a datanode (in the case of a MapReduce task, for instance), then it will read from the local datanode, if it hosts a copy of the block.

The DistributedFileSystem returns an FSDataInputStream (an input stream that sup- ports 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, then 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 DFSInput Stream 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, since the data traffic is spread across all the datanodes in the cluster. The namenode meanwhile 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 bot- tleneck as the number of clients grew.

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 since it clarifies HDFS's coherency model.

The case we're going to consider is the case of creating a new file, writing data to it, then closing the file.

The client creates the file by calling create() on Distributed Filesystem (step 1 in Distributed Filesystem makes an RPC call to the name node to create a new file in the filesystem's namespace, with no blocks associated with it (step 2). The name- node performsvarious 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 name node makes a record of the new file; otherwise, file creation fails and the client is thrown an IOException. The Distributed Filesystem returns an FS Data Output Stream for the client

to start writing data to. Just as in the read case, FSDataOutputStream wraps a DFSOutput Stream, which handles communication with the datanodes and namenode.

As the client writes data (step 3), DFS Output Stream splits it into packets, which it writes to an internal queue, called the data queue. The data queue is consumed by the Data Streamer, whose responsibility it is to ask the name node to allocate new blocks by picking a list of suitable datanodes to store the replicas. The list of datanodes forms a pipeline—we'll assume the replication level is three, so there are three nodes in the pipeline. The Data Streamer 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).

DFS Output Stream 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 name- node, so that the partial block on the failed datanode will be deleted if the failed.

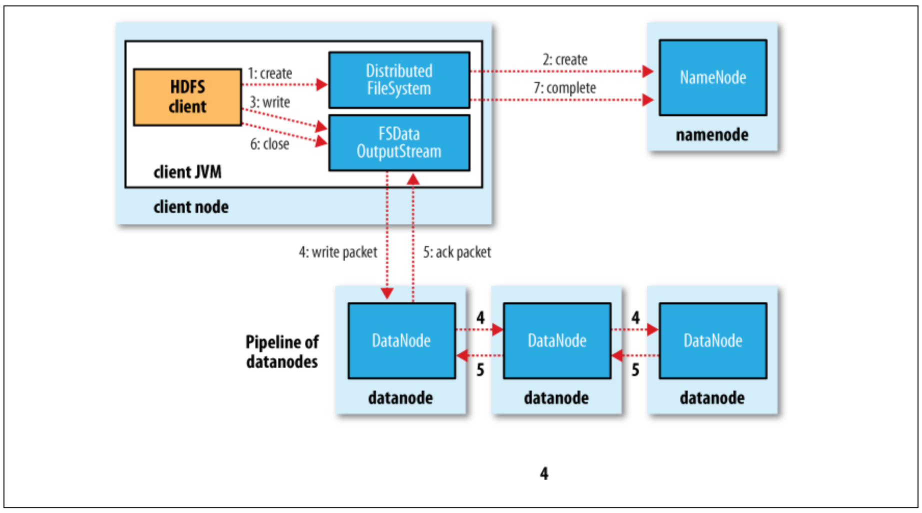

Figure 3-4. A client writing data to HDFS

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 (default one) are written, the write will succeed, and the block will be asynchronously replicated across the cluster until its target rep- lication 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 Data Streamer asking for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

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 Combine File Input

Format" 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, then consider using HDFS Federation to scale the namespace