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:

1. The architecture of HDFS is described in Robert Chansler et al.'s, "The Hadoop Distributed File System," which appeared in *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks* by Amy Brown and Greg Wilson (eds.).

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]

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 (see Chapter 20) 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]

2. See Konstantin V. Shvachko and Arun C. Murthy, "Scaling Hadoop to 4000 nodes at Yahoo!", September 30, 2008.

3. See Chapter 10 for a typical machine specification.

4. For an exposition of the scalability limits of HDFS, see Konstantin V. Shvachko, "HDFS Scalability: The Limits to Growth", April 2010.

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, in append-only fashion. 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, 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—128 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. (For example, a 1 MB file stored with a block size of 128 MB uses 1 MB of disk space, not 128 MB.) 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. If the block is large enough, the time it takes to transfer the data from the disk can be significantly longer than the time to seek to the start of the block. Thus, transferring 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 128 MB, although many HDFS installations use larger block sizes. 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 it 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 chunks 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 97 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:

```
% hdfs fsck / -files -blocks
```

will list the blocks that make up each file in the filesystem. (See also "Filesystem check (fsck)" on page 326.)

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 datanodes 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. (Note that it is possible to run a hot standby namenode instead of a secondary, as discussed in "HDFS High Availability" on page 48.)

See "The filesystem image and edit log" on page 318 for more details.

Block Caching

Normally a datanode reads blocks from disk, but for frequently accessed files the blocks may be explicitly cached in the datanode's memory, in an off-heap *block cache*. By default, a block is cached in only one datanode's memory, although the number is configurable on a per-file basis. Job schedulers (for MapReduce, Spark, and other frameworks) can take advantage of cached blocks by running tasks on the datanode where a block is cached, for increased read performance. A small lookup table used in a join is a good candidate for caching, for example.

Users or applications instruct the namenode which files to cache (and for how long) by adding a *cache directive* to a *cache pool*. Cache pools are an administrative grouping for managing cache permissions and resource usage.

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 294). 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.

Hadoop 2 remedied this situation by adding support for HDFS high availability (HA). In this implementation, there are 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. 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.
- The secondary namenode's role is subsumed by the standby, which takes periodic checkpoints of the active namenode's namespace.

There are two choices for the highly available shared storage: an NFS filer, or a *quorum journal manager* (QJM). The QJM is a dedicated HDFS implementation, designed for the sole purpose of providing a highly available edit log, and is the recommended choice for most HDFS installations. The QJM runs as a group of *journal nodes*, and each edit must be written to a majority of the journal nodes. Typically, there are three journal nodes, so the system can tolerate the loss of one of them. This arrangement is similar to the way ZooKeeper works, although it is important to realize that the QJM implementation does not use ZooKeeper. (Note, however, that HDFS HA *does* use ZooKeeper for electing the active namenode, as explained in the next section.)

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*. There are various failover controllers, but the default 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 QJM only allows one namenode to write to the edit log at one time; however, it is still possible for the previously active namenode to serve stale read requests to clients, so setting up an SSH fencing command that will kill the namenode's process is a good idea. Stronger fencing methods are required when using an NFS filer for the shared edit log, since it is not possible to only allow one namenode to write at a time (this is why QJM is recommended). The range of fencing mechanisms includes revoking the namenode's 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.

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.

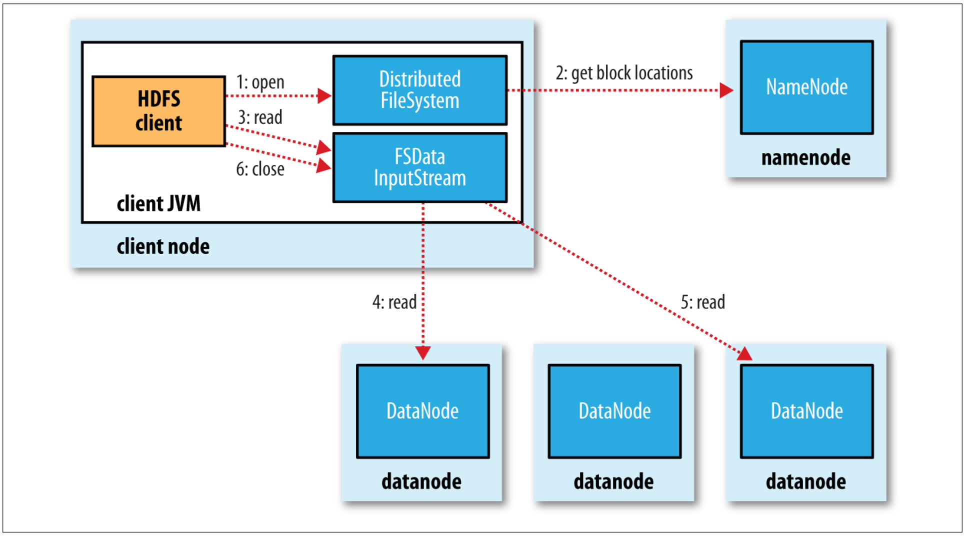

Figure 3-2. A client reading data from HDFS

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 remote procedure calls (RPCs), to determine the locations of 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 70). 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 and "Short-circuit local reads" on page 308).

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, the `DFSInputStream` attempts to read a replica of the block from another datanode; it also reports the corrupted block to the namenode.

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.)

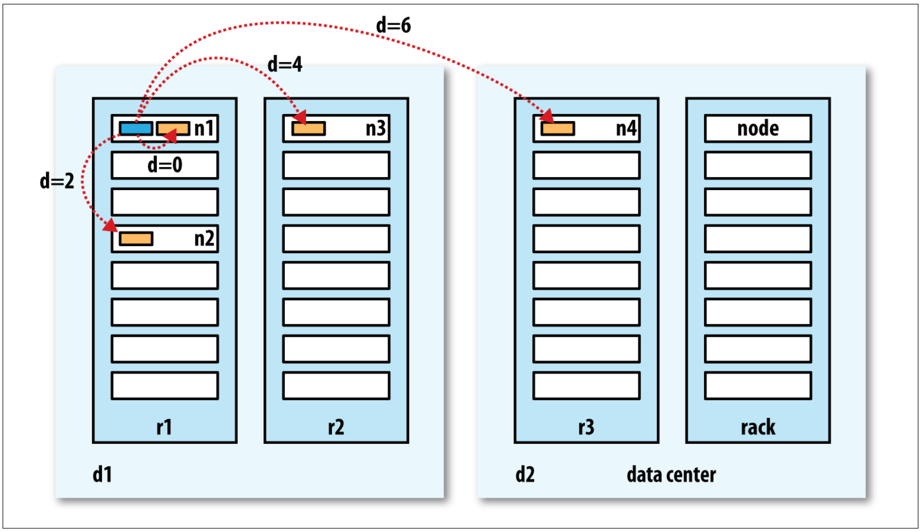

Figure 3-3. Network distance in Hadoop

Finally, it is important to realize that Hadoop cannot magically discover your network topology for you; it needs some help (we'll cover how to configure topology in "Network Topology" on page 286). 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.

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. This is illustrated in Figure 3-4.

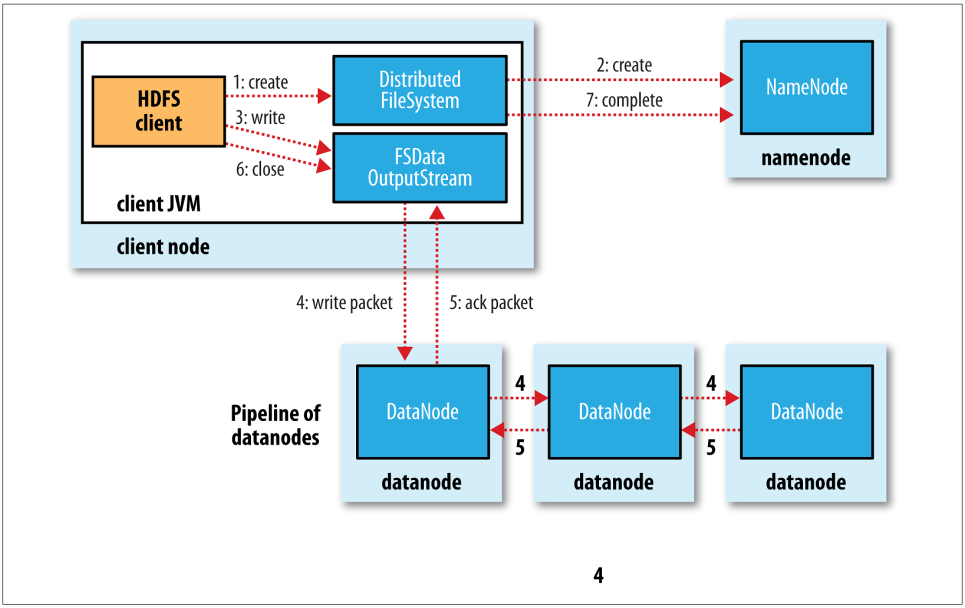

Figure 3-4. A client writing data to HDFS

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), the `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 each 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).

The `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 any 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 a new pipeline is constructed from the two good datanodes. The remainder of the block's data is written to the 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, for multiple datanodes to fail while a block is being written. As long as `dfs.namenode.replication.min` replicas (which defaults to 1) 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 3).

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 acknowledgments 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 (because `DataStreamer` asks 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 tradeoff 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 possible placement strategies.

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 in 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.

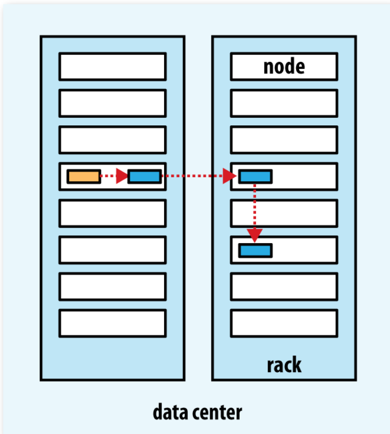

Figure 3-5. A typical replica pipeline

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));
```

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 way to force all buffers to be flushed to the datanodes via the `hflush()` method on `FSDataOutputStream`. After a successful return from `hflush()`, 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:

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

Note that `hflush()` does not guarantee that the datanodes have written the data to disk, only that it's in the datanodes' memory (so in the event of a data center power outage, for example, data could be lost). For this stronger guarantee, use `hsync()` instead.[9]

The behavior of `hsync()` is similar to that of 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 `hflush()`, 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())));
```

9. In Hadoop 1.x, `hflush()` was called `sync()`, and `hsync()` did not exist.

Consequences for application design

This coherency model has implications for the way you design applications. With no calls to `hflush()` or `hsync()`, 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 `hflush()` at suitable points, such as after writing a certain number of records or number of bytes. Though the `hflush()` operation is designed to not unduly tax HDFS, it does have some overhead (and `hsync()` has more), 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 `hflush()` (or `hsync()`) frequencies.

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 data to and from Hadoop filesystems in parallel.

One use for *distcp* is as an efficient replacement for `hadoop fs -cp`. For example, you can copy one file to another with:[10]

```
% hadoop distcp file1 file2
```

You can also copy directories:

```
% hadoop distcp dir1 dir2
```

If *dir2* does not exist, it will be created, and the contents of the *dir1* directory will be copied there. You can specify multiple source paths, and all will be copied to the destination.

If *dir2* already exists, then *dir1* will be copied under it, creating the directory structure *dir2/dir1*. If this isn't what you want, you can supply the `-overwrite` option to keep the same directory structure and force files to be overwritten. You can also update only the files that have changed using the `-update` option. This is best shown with an example. If we changed a file in the *dir1* subtree, we could synchronize the change with *dir2* by running:

```
% hadoop distcp -update dir1 dir2
```

10. Even for a single file copy, the *distcp* variant is preferred for large files since `hadoop fs -cp` copies the file via the client running the command.