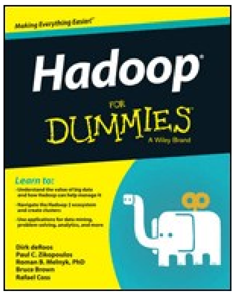

Hadoop for Dummies

by Dirk deRoos et al.

Chapter 4: Storing Data in Hadoop—The Hadoop Distributed File System

In This Chapter

- Seeing how HDFS stores files in blocks
- Looking at HDFS components and architecture
- Scaling out HDFS
- Working with checkpoints
- Federating your NameNode
- Putting HDFS to the availability test

When it comes to the core Hadoop infrastructure, you have two components: storage and processing. The Hadoop Distributed File System (HDFS) is the storage component. In short, HDFS provides a distributed architecture for extremely large scale storage, which can easily be extended by scaling out.

Let us remind you why this is a big deal. In the late 1990s, after the Internet established itself as a fixture in society, Google was facing the major challenge of having to be able to store and process not only all the pages on the Internet but also Google users' web log data. Google's major claim to fame, then and now, was its expansive and current index of the Internet's many pages, and its ability to return highly relevant search results to its users. The key to its success was being able to process and analyze both the Internet data *and* its user data. At the time, Google was using a *scale-up* architecture model — a model where you increase system capacity by adding CPU cores, RAM, and disk to an existing server — and it had two major problems:

- **Expense**: Scaling up the hardware by using increasingly bigger servers with more storage was becoming incredibly expensive. As computer systems increased in their size, their cost increased at an even higher rate. In addition, Google needed a *highly available environment* — one that would ensure its mission critical workloads could continue running in the event of a failure — so a failover system was also needed, doubling the IT expense.
- **Structural limitations**: Google engineers were reaching the limits of what a scale-up architecture could sustain. For example, with the increasing data volumes Google was seeing, it was taking much longer for data sets to be transferred from SANs to the CPUs for processing. And all the while, the Internet's growth and usage showed no sign of slowing down.

Rather than scale *up*, Google engineers decided to scale *out* by using a cluster of smaller servers they could continually add to if they needed more power or capacity. To enable a scale-out model, they developed the Google File System (GFS), which was the inspiration for the engineers who first developed HDFS. The early use cases, for both the Google and HDFS engineers, were solely based on the batch processing of large data sets. This concept is reflected in the design of HDFS, which is optimized for large-scale batch processing workloads. Since Hadoop came on the scene in 2005, it has emerged as the premier platform for large-scale data storage and processing. There's a growing demand for the optimization of interactive workloads as well, which involve queries that involve small subsets of the data. Though today's HDFS still works best for batch workloads, features are being added to improve the performance of interactive workloads.

Data Storage in HDFS

Just to be clear, storing data in HDFS is not entirely the same as saving files on your personal computer. In fact, quite a number of differences exist — most having to do with optimizations that make HDFS able to scale out easily across thousands of slave nodes and perform well with batch workloads.

The most noticeable difference initially is the size of files. Hadoop is designed to work best with a modest number of extremely large files. Average file sizes that are larger than 500MB are the norm.

Here's an additional bit of background information on how data is stored: HDFS has a Write Once, Read Often model of data access. That means the contents of individual files cannot be modified, other than appending new data to the end of the file.

Don't worry, though: There's still lots you can do with HDFS files, including

- Create a new file
- Append content to the end of a file
- Delete a file
- Rename a file
- Modify file attributes like owner

Taking a Closer Look at Data Blocks

When you store a file in HDFS, the system breaks it down into a set of individual blocks and stores these blocks in various slave nodes in the

Hadoop cluster, as shown in Figure 4-1. This is an entirely normal thing to do, as all file systems break files down into blocks before storing them to disk. HDFS has no idea (and doesn't care) what's stored inside the file, so raw files are not split in accordance with rules that we humans would understand. Humans, for example, would want *record boundaries* — the lines showing where a record begins and ends — to be respected. HDFS is often blissfully unaware that the final record in one block may be only a partial record, with the rest of its content shunted off to the following block. HDFS only wants to make sure that files are split into evenly sized blocks that match the predefined block size for the Hadoop instance (unless a custom value was entered for the file being stored). In Figure 4-1, that block size is 128MB.

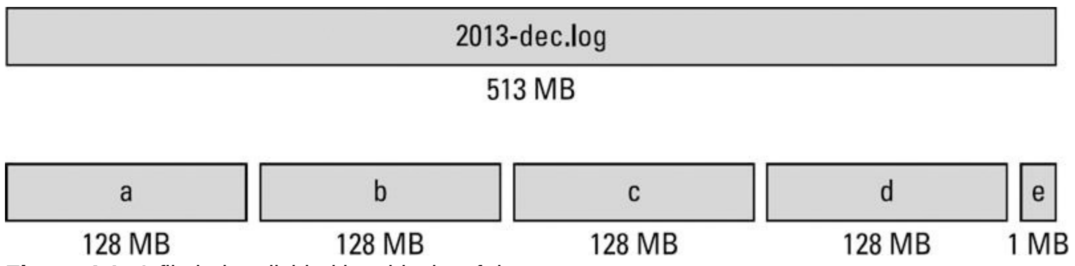

Figure 4-1: A file being divided into blocks of data

REMEMBER Not every file you need to store is an exact multiple of your system's block size, so the final data block for a file uses only as much space as is needed. In the case of Figure 4-1, the final block of data is 1MB.

The concept of storing a file as a collection of blocks is entirely consistent with how file systems normally work. But what's different about HDFS is the scale. A typical block size that you'd see in a file system under Linux is 4KB, whereas a typical block size in Hadoop is 128MB. This value is configurable, and it can be customized, as both a new system default and a custom value for individual files.

Hadoop was designed to store data at the petabyte scale, where any potential limitations to scaling out are minimized. The high block size is a direct consequence of this need to store data on a massive scale. First of all, every data block stored in HDFS has its own metadata and needs to be tracked by a central server so that applications needing to access a specific file can be directed to wherever all the file's blocks are stored. If the block size were in the kilobyte range, even modest volumes of data in the terabyte scale would overwhelm the metadata server with too many blocks to track. Second, HDFS is designed to enable high throughput so that the parallel processing of these large data sets happens as quickly as possible. The key to Hadoop's scalability on the data processing side is, and always will be, *parallelism* — the ability to process the individual blocks of these large files in parallel. To enable efficient processing, a balance needs to be struck. On one hand, the block size needs to be large enough to warrant the resources dedicated to an individual unit of data processing (for instance, a map or reduce task, which we look at in Chapter 6). On the other hand, the block size can't be so large that the system is waiting a very long time for one last unit of data processing to finish its work. These two considerations obviously depend on the kinds of work being done on the data blocks.

Replicating Data Blocks

HDFS is designed to store data on inexpensive, and more unreliable, hardware. (We say more on that topic later in this chapter.) *Inexpensive* has an attractive ring to it, but it does raise concerns about the reliability of the system as a whole, especially for ensuring the high availability of the data. Planning ahead for disaster, the brains behind HDFS made the decision to set up the system so that it would store three (count 'em — three) copies of every data block.

HDFS assumes that every disk drive and every slave node is inherently unreliable, so, clearly, care must be taken in choosing where the three copies of the data blocks are stored. Figure 4-2 shows how data blocks from the earlier file are *striped* across the Hadoop cluster — meaning they are evenly distributed between the slave nodes so that a copy of the block will still be available regardless of disk, node, or rack failures.

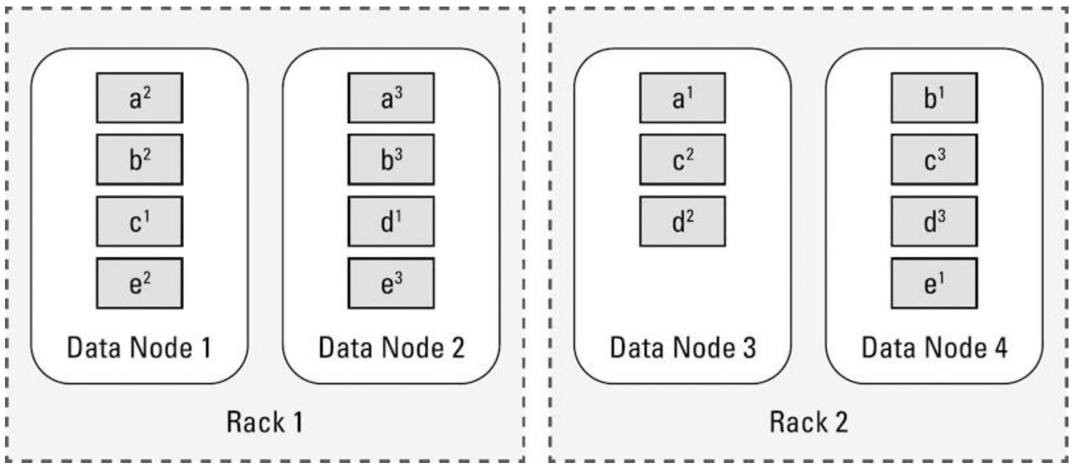

Figure 4-2: Replication patterns of data blocks in HDFS

The file shown in Figure 4-2 has five data blocks, labeled a, b, c, d, and e. If you take a closer look, you can see this particular cluster is made up of two racks with two nodes apiece, and that the three copies of each data block have been spread out across the various slave nodes.

Every component in the Hadoop cluster is seen as a potential failure point, so when HDFS stores the replicas of the original blocks across the Hadoop cluster, it tries to ensure that the block replicas are stored in different failure points. For example, take a look at Block A. At the time it needed to be stored, Slave Node 3 was chosen, and the first copy of Block A was stored there. For multiple rack systems, HDFS then determines that the remaining two copies of block A need to be stored in a different rack. So the second copy of block A is stored on Slave Node 1. The final copy can be stored on the same rack as the second copy, but not on the same slave node, so it gets stored on Slave Node 2.

Slave Node and Disk Failures

Like death and taxes, disk failures (and given enough time, even node or rack failures), are inevitable. Given the example in Figure 4-2, even if one rack were to fail, the cluster could continue functioning. Performance would suffer because you've lost half your processing resources, but the system is still online and all data is still available.

In a scenario where a disk drive or a slave node fails, the central metadata server for HDFS (called the NameNode) eventually finds out that the file blocks stored on the failed resource are no longer available. For example, if Slave Node 3 in Figure 4-2 fails, it would mean that Blocks A, C, and D are *underreplicated*. In other words, too few copies of these blocks are available in HDFS. When HDFS senses that a block is underreplicated, it orders a new copy.

To continue the example, let's say that Slave Node 3 comes back online after a few hours. Meanwhile, HDFS has ensured that there are three copies of all the file blocks. So now, Blocks A, C, and D have four copies apiece and are *overreplicated*. As with underreplicated blocks, the HDFS central metadata server will find out about this as well, and will order one copy of every file to be deleted.

One nice result of the availability of data is that when disk failures do occur, there's no need to immediately replace failed hard drives. This can more effectively be done at regularly scheduled intervals.

Sketching Out the HDFS Architecture

The core concept of HDFS is that it can be made up of dozens, hundreds, or even thousands of individual computers, where the system's files are stored in directly attached disk drives. Each of these individual computers is a self-contained server with its own memory, CPU, disk storage, and installed operating system (typically Linux, though Windows is also supported). Technically speaking, HDFS is a *user-space-level file system* because it lives on top of the file systems that are installed on all individual computers that make up the Hadoop cluster. Figure 4-3 illustrates this concept.

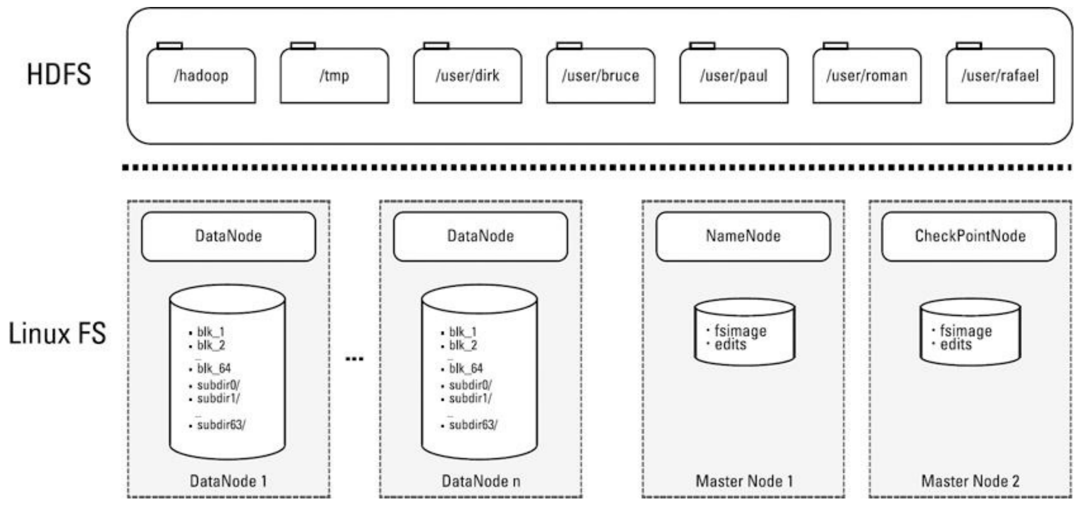

Figure 4-3: HDFS as a user-space-level file system

Figure 4-3 shows that a Hadoop cluster is made up of two classes of servers: *slave nodes*, where the data is stored and processed, and *master nodes*, which govern the management of the Hadoop cluster. On each of the master nodes and slave nodes, HDFS runs special services and stores raw data to capture the state of the file system. In the case of the slave nodes, the raw data consists of the blocks stored on the node, and with the master nodes, the raw data consists of metadata that maps data blocks to the files stored in HDFS.

Looking at Slave Nodes

In a Hadoop cluster, each data node (also known as a *slave node*) runs a background process named DataNode. This background process (also known as a *daemon*) keeps track of the slices of data that the system stores on its computer. It regularly talks to the master server for HDFS (known as the NameNode) to report on the health and status of the locally stored data.

REMEMBER Data blocks are stored as raw files in the local file system. From the perspective of a Hadoop user, you have no idea which of the slave nodes has the pieces of the file you need to process. From within Hadoop, you don't see data blocks or how they're distributed across the cluster — all you see is a listing of files in HDFS. The complexity of how the file blocks are distributed across the cluster is hidden from you — you don't know how complicated it all is, and you don't *need* to know. Actually, the slave nodes themselves don't even know what's inside the data blocks they're storing. It's the NameNode server that knows the mappings of which data blocks compose the files stored in HDFS.

Better Living Through Redundancy

One core design principle of HDFS is the concept of minimizing the cost of the individual slave nodes by using commodity hardware components. For massively scalable systems, this idea is a sensible one because costs escalate quickly when you need hundreds or thousands of slave nodes. Using lower-cost hardware has a consequence, though, in that individual components aren't as reliable as more expensive hardware.

When you're choosing storage options, consider the impact of using commodity drives rather than more expensive enterprise-quality drives. Imagine that you have a 750-node cluster, where each node has 12 hard disk drives dedicated to HDFS storage. Based on an annual failure rate (AFR) of 4 percent for commodity disk drives (a given hard disk drive has a 4 percent likelihood of failing in a given year, in other words), your cluster will likely experience a hard disk failure every day of the year.

Because there can be so many slave nodes, their failure is also a common occurrence in larger clusters with hundreds or more nodes. With this information in mind, HDFS has been engineered on the assumption that *all* hardware components, even at the slave node level, are unreliable. HDFS overcomes the unreliability of individual hardware components by way of redundancy: That's the idea behind those three copies of every file stored in HDFS, distributed throughout the system. More specifically, each file block stored in HDFS has a total of three replicas. If one system breaks with a specific file block that you need, you can turn to the other two.

Sketching Out Slave Node Server Design

To balance such important factors as total cost of ownership, storage capacity, and performance, you need to carefully plan the design of your slave nodes. Chapter 16 covers this topic in greater detail, but we want to take a quick look in this section at what a typical slave node looks like.

We commonly see slave nodes now where each node typically has between 12 and 16 locally attached 3TB hard disk drives. Slave nodes use moderately fast dual-socket CPUs with six to eight cores each — no speed demons, in other words. This is accompanied by 48GB of RAM. In short, this server is optimized for dense storage.

Because HDFS is a user-space-level file system, it's important to optimize the local file system on the slave nodes to work with HDFS. In this regard, one high-impact decision when setting up your servers is choosing a file system for the Linux installation on the slave nodes. Ext3 is the most commonly deployed file system because it has been the most stable option for a number of years. Take a look at Ext4, however. It's the next version of Ext3, and it has been available long enough to be widely considered stable and reliable. More importantly for our purposes, it has a number of optimizations for handling large files, which makes it an ideal choice for HDFS slave node servers.

TIP Don't use the Linux Logical Volume Manager (LVM) — it represents an additional layer between the Linux file system and HDFS, which prevents Hadoop from optimizing its performance. Specifically, LVM aggregates disks, which hampers the resource management that HDFS and YARN do, based on how files are distributed on the physical drives.

Keeping Track of Data Blocks with NameNode

When a user stores a file in HDFS, the file is divided into data blocks, and three copies of these data blocks are stored in slave nodes throughout the Hadoop cluster. That's a lot of data blocks to keep track of. The NameNode acts as the address book for HDFS because it knows not only which blocks make up individual files but also where each of these blocks and their replicas are stored. As you might expect, knowing where the bodies are buried makes the NameNode a critically important component in a Hadoop cluster. If the NameNode is unavailable, applications cannot access any data stored in HDFS.

If you take another look at Figure 4-3, you can see the NameNode daemon running on a master node server. All mapping information dealing with the data blocks and their corresponding files is stored in a file named `fsimage`. HDFS is a journaling file system, which means that any data changes are logged in an edit journal that tracks events since the last *checkpoint* — the last time when the edit log was merged with `fsimage`. In HDFS, the edit journal is maintained in a file named `edits` that's stored on the NameNode.

NameNode Startup and Operation

To understand how the NameNode works, it's helpful to take a look at how it starts up. Because the purpose of the NameNode is to inform applications of how many data blocks they need to process and to keep track of the exact location where they're stored, it needs all the block locations and block-to-file mappings that are available in RAM. These are the steps the NameNode takes. To load all the information that the NameNode needs after it starts up, the following happens:

1. The NameNode loads the `fsimage` file into memory.
2. The NameNode loads the `edits` file and re-plays the journaled changes to update the block metadata that's already in memory.
3. The DataNode daemons send the NameNode block reports.

   For each slave node, there's a block report that lists all the data blocks stored there and describes the health of each one.

After the startup process is completed, the NameNode has a complete picture of all the data stored in HDFS, and it's ready to receive application requests from Hadoop clients. As data files are added and removed based on client requests, the changes are written to the slave node's disk volumes, journal updates are made to the `edits` file, and the changes are reflected in the block locations and metadata stored in the NameNode's memory (see Figure 4-4).

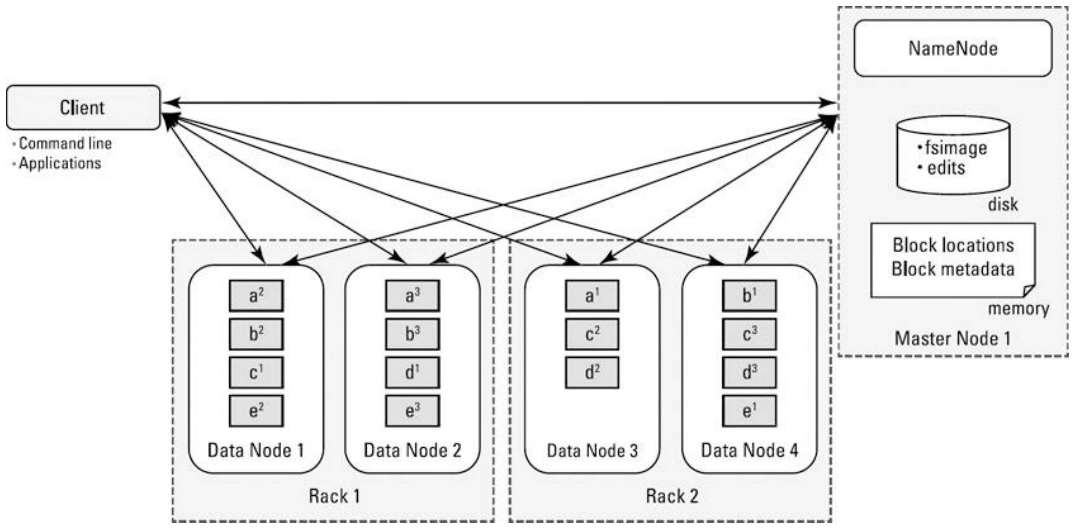

Figure 4-4: Interaction between HDFS components

Throughout the life of the cluster, the DataNode daemons send the NameNode heartbeats (a quick signal) every three seconds, indicating they're active. (This default value is configurable.) Every six hours (again, a configurable default), the DataNodes send the NameNode a block report outlining which file blocks are on their nodes. This way, the NameNode always has a current view of the available resources in the cluster.

Writing Data

To create new files in HDFS, the following process would have to take place (refer to Figure 4-4 to see the components involved):

1. The client sends a request to the NameNode to create a new file.

   The NameNode determines how many blocks are needed, and the client is granted a *lease* for creating these new file blocks in the cluster. As part of this lease, the client has a time limit to complete the creation task. (This time limit ensures that storage space isn't taken up by failed client applications.)

2. The client then writes the first copies of the file blocks to the slave nodes using the lease assigned by the NameNode.

   The NameNode handles write requests and determines where the file blocks and their replicas need to be written, balancing availability and performance. The first copy of a file block is written in one rack, and the second and third copies are written on a different rack than the first copy, but in different slave nodes in the same rack. This arrangement minimizes network traffic while ensuring that no data blocks are on the same failure point.

3. As each block is written to HDFS, a special process writes the remaining replicas to the other slave nodes identified by the NameNode.

4. After the DataNode daemons acknowledge the file block replicas have been created, the client application closes the file and notifies the NameNode, which then closes the open lease.

Reading Data

To read files from HDFS, the following process would have to take place (again, refer to Figure 4-4 for the components involved):

1. The client sends a request to the NameNode for a file.

   The NameNode determines which blocks are involved and chooses, based on overall proximity of the blocks to one another and to the client, the most efficient access path.

2. The client then accesses the blocks using the addresses given by the NameNode.

Balancing Data in the Hadoop Cluster

Over time, with combinations of uneven data-ingestion patterns (where some slave nodes might have more data written to them) or node failures, data is likely to become unevenly distributed across the racks and slave nodes in your Hadoop cluster. This uneven distribution can have a detrimental impact on performance because the demand on individual slave nodes will become unbalanced; nodes with little data won't be fully used; and nodes with many blocks will be overused. (***Note***: The overuse and underuse are based on disk activity, not on CPU or RAM.) HDFS includes a balancer utility to redistribute blocks from overused slave nodes to underused ones while maintaining the policy of putting blocks on different slave nodes and racks. Hadoop administrators should regularly check HDFS health, and if data becomes unevenly distributed, they should invoke the balancer utility.

NameNode Master Server Design

Because of its mission-critical nature, the master server running the NameNode daemon needs markedly different hardware requirements than the ones for a slave node. Most significantly, enterprise-level components need to be used to minimize the probability of an outage. Also, you'll need enough RAM to load into memory all the metadata and location data about all the data blocks stored in HDFS. See Chapter 16 for a full discussion on this topic.

Checkpointing Updates

Earlier in this chapter, we say that HDFS is a journaled file system, where new changes to files in HDFS are captured in an edit log that's stored on the NameNode in a file named `edits`. Periodically, when the `edits` file reaches a certain threshold or after a certain period has elapsed, the journaled entries need to be committed to the master `fsimage` file. The NameNode itself doesn't do this, because it's designed to answer application requests as quickly as possible. More importantly, considerable risk is involved in having this metadata update operation managed by a single master server.

WARNING! If the metadata describing the mappings between the data blocks and their corresponding files becomes corrupted, the original data is as good as lost.

Checkpointing services for a Hadoop cluster are handled by one of four possible daemons, which need to run on their own dedicated master node alongside the NameNode daemon's master node:

- **Secondary NameNode**: Prior to Hadoop 2, this was the only checkpointing daemon, performing the checkpointing process described in this section. The Secondary NameNode has a notoriously inaccurate name because it is in no way "secondary" or a "standby" for the NameNode.
- **Checkpoint Node**: The Checkpoint Node is the replacement for the Secondary NameNode. It performs checkpointing and nothing more.
- **Backup Node**: Provides checkpointing service, but also maintains a backup of the `fsimage` and edits file.
- **Standby NameNode**: Performs checkpointing service and, unlike the old Secondary NameNode, the Standby NameNode is a true standby server, enabling a hot-swap of the NameNode process to avoid any downtime.

The Checkpointing Process

The following steps, depicted in Figure 4-5, describe the checkpointing process as it's carried out by the NameNode and the checkpointing service (note that four possible daemons can be used for checkpointing — see above):

1. When it's time to perform the checkpoint, the NameNode creates a new file to accept the journaled file system changes.

   It names the new file `edits.new`.
2. As a result, the `edits` file accepts no further changes and is copied to the checkpointing service, along with the `fsimage` file.
3. The checkpointing service merges these two files, creating a file named `fsimage.ckpt`.
4. The checkpointing service copies the `fsimage.ckpt` file to the NameNode.
5. The NameNode overwrites the file `fsimage` with `fsimage.ckpt`.
6. The NameNode renames the `edits.new` file to `edits`.

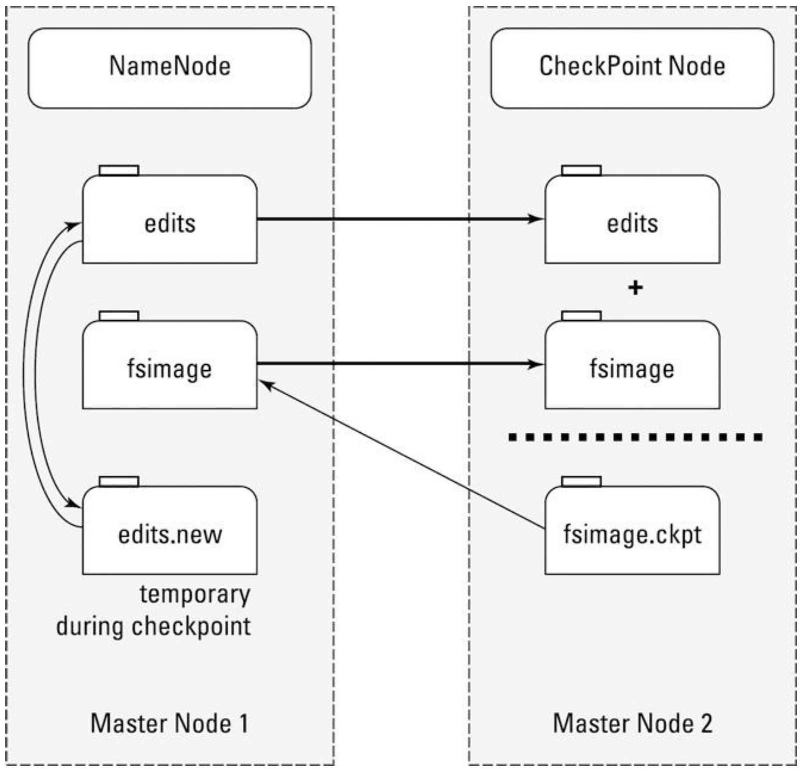

Figure 4-5: Check-pointing the HDFS edit journal

Backup Node Considerations

In addition to providing checkpointing functionality, the Backup Node maintains the current state of all the HDFS block metadata in memory, just like the NameNode. In this sense, it maintains a real-time backup of the NameNode's state. As a result of keeping the block metadata in memory, the Backup Node is far more efficient than the Checkpoint Node at performing the checkpointing task, because the `fsimage` and `edits` files don't need to be transferred and then merged. These changes are already merged in memory.

REMEMBER Another benefit of using the Backup Node is that the NameNode can be configured to delegate the Backup Node so that it persists journal data to disk.

If you're using the Backup Node, you can't run the Checkpoint Node. There's no need to do so, because the checkpointing process is already being taken care of.

Standby NameNode Considerations

The Standby NameNode is the designated hot standby master server for the NameNode. While serving as standby, it also performs the checkpointing process. As such, you can't run the Backup Node or Standby Node.

Secondary NameNode, Checkpoint Node, Backup Node, and Standby NameNode Master Server Design

The master server running the Secondary NameNode, Checkpoint Node, Backup Node, or Standby NameNode daemons have the same hardware requirements as the ones deployed for the NameNode master server. The reason is that these servers also load into memory all the metadata and location data about all the data blocks stored in HDFS. See Chapter 16 for a full discussion on this topic.

HDFS Federation

Before Hadoop 2 entered the scene, Hadoop clusters had to live with the fact that NameNode placed limits on the degree to which they could scale. Few clusters were able to scale beyond 3,000 or 4,000 nodes. NameNode's need to maintain records for every block of data stored in the cluster turned out to be the most significant factor restricting greater cluster growth. When you have too many blocks, it becomes increasingly difficult for the NameNode to scale up as the Hadoop cluster scales out.

The solution to expanding Hadoop clusters indefinitely is to *federate* the NameNode. Specifically, you must set it up so that you have multiple NameNode instances running on their own, dedicated master nodes and then making each NameNode responsible only for the file blocks in its own name space. In Figure 4-6, you can see a Hadoop cluster with two NameNodes serving a single cluster. The slave nodes all contain blocks from both name spaces.

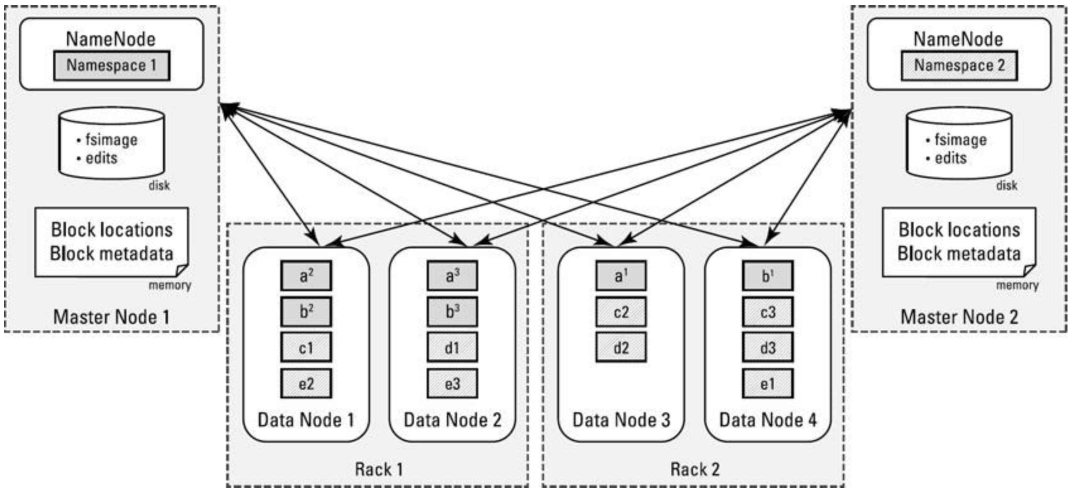

Figure 4-6: Replication patterns of data blocks in HDFS

HDFS High Availability

Often in Hadoop's infancy, a great amount of discussion was centered on the NameNode's representation of a single point of failure. Hadoop, overall, has always had a robust and failure-tolerant architecture, with the exception of this key area. As we mention earlier in this chapter, without the NameNode, there's no Hadoop cluster.

Using Hadoop 2, you can configure HDFS so that there's an Active NameNode and a Standby NameNode (see Figure 4-7). The Standby NameNode needs to be on a dedicated master node that's configured identically to the master node used by the Active NameNode (refer to Figure 4-7).

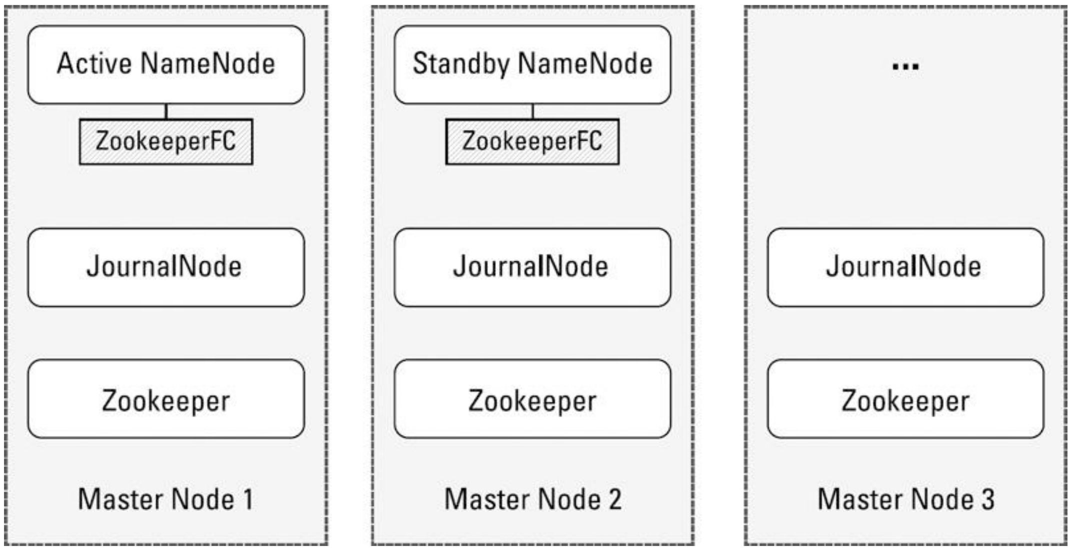

Figure 4-7: High availability of the NameNode

The Standby NameNode isn't sitting idly by while the NameNode handles all the block address requests. The Standby NameNode, charged with the task of keeping the state of the block locations and block metadata in memory, handles the HDFS checkpointing responsibilities. The Active NameNode writes journal entries on file changes to the majority of the JournalNode services, which run on the master nodes. (***Note***: The HDFS high availability solution requires at least three master nodes, and if there are more, there can be only an odd number.) If a failure occurs, the Standby Node first reads all completed journal entries (where a majority of Journal Nodes have an entry, in other words), to ensure that the new Active NameNode is fully consistent with the state of the cluster.

Zookeeper is used to monitor the Active NameNode and to handle the failover logistics if the Active NameNode becomes unavailable. Both the Active and Standby NameNodes have dedicated Zookeeper Failover Controllers (ZFC) that perform the monitoring and failover tasks. In the event of a failure, the ZFC informs the Zookeeper instances on the cluster, which then elect a new Active NameNode.

REMEMBER Apache Zookeeper provides coordination and configuration services for distributed systems, so it's no wonder we see it used all over the place in Hadoop. See Chapter 12 for more information about Zookeeper.