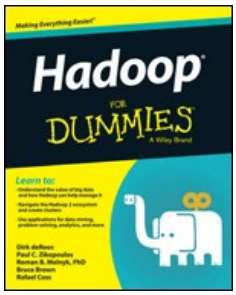

Hadoop for Dummies

by Dirk deRoos et al.

Chapter 5: Reading and Writing Data

In This Chapter

- Compressing data
- Managing files with the Hadoop file system commands
- Ingesting log data with Flume

This chapter tells you all about getting data in and out of Hadoop, which are basic operations along the path of big data discovery.

We begin by describing the importance of data compression for optimizing the performance of your Hadoop installation, and we briefly outline some of the available compression utilities that are supported by Hadoop. We also give you an overview of the Hadoop file system (FS) shell (a command-line interface), which includes a number of shell-like commands that you can use to directly interact with the Hadoop Distributed File System (HDFS) and other file systems that Hadoop supports. Finally, we describe how you can use Apache Flume — the Hadoop community technology for collecting large volumes of log files and storing them in Hadoop — to efficiently ingest huge volumes of log data.

TIP We use the word "ingest" all over this chapter and this book. In short, ingesting data simply means to accept data from an outside source and store it in Hadoop. With Hadoop's scalable, reliable, and inexpensive storage, we think you'll understand why people are so keen on this.

Compressing Data

The huge data volumes that are realities in a typical Hadoop deployment make compression a necessity. Data compression definitely saves you a great deal of storage space and is sure to speed up the movement of that data throughout your cluster. Not surprisingly, a number of available compression schemes, called codecs, are out there for you to consider.

REMEMBER In a Hadoop deployment, you're dealing (potentially) with quite a large number of individual slave nodes, each of which has a number of large disk drives. It's not uncommon for an individual slave node to have upwards of 45TB of raw storage space available for HDFS. Even though Hadoop slave nodes are designed to be inexpensive, they're not free, and with large volumes of data that have a tendency to grow at increasing rates, compression is an obvious tool to control extreme data volumes.

First, some basic terms: A *codec*, which is a shortened form of *co*mpressor/*dec*ompressor, is technology (software or hardware, or both) for compressing and decompressing data; it's the implementation of a compression/decompression algorithm. You need to know that some codecs support something called splittable compression and that codecs differ in both the speed with which they can compress and decompress data and the degree to which they can compress it.

Splittable compression is an important concept in a Hadoop context. The way Hadoop works is that files are split if they're larger than the file's block size setting, and individual file splits can be processed in parallel by different mappers. With most codecs, text file splits cannot be decompressed independently of other splits from the same file, so those codecs are said to be non-splittable, so MapReduce processing is limited to a single mapper. Because the file can be decompressed only as a whole, and not as individual parts based on splits, there can be no parallel processing of such a file, and performance might take a huge hit as a job waits for a single mapper to process multiple data blocks that can't be decompressed independently. (For more on how MapReduce processing works, see Chapter 6.)

REMEMBER Splittable compression is only a factor for text files. For binary files, Hadoop compression codecs compress data within a binary-encoded container, depending on the file type (for example, a SequenceFile, Avro, or ProtocolBuffer).

Speaking of performance, there's a cost (in terms of processing resources and time) associated with compressing the data that is being written to your Hadoop cluster. With computers, as with life, nothing is free. When compressing data, you're exchanging processing cycles for disk space. And when that data is being read, there's a cost associated with decompressing the data as well. Be sure to weigh the advantages of storage savings against the additional performance overhead.

If the input file to a MapReduce job contains compressed data, the time that is needed to read that data from HDFS is reduced and job performance is enhanced. The input data is decompressed automatically when it is being read by MapReduce. The input filename extension determines which supported codec is used to automatically decompress the data. For example, a `.gz` extension identifies the file as a gzip-compressed file.

It can also be useful to compress the intermediate output of the map phase in the MapReduce processing flow. Because map function output is written to disk and shipped across the network to the reduce tasks, compressing the output can result in significant performance improvements. And if you want to store the MapReduce output as history files for future use, compressing this data can significantly reduce the amount of needed space in HDFS.

There are many different compression algorithms and tools, and their characteristics and strengths vary. The most common trade-off is between compression ratios (the degree to which a file is compressed) and compress/decompress speeds. The Hadoop framework supports several codecs. The framework transparently compresses and decompresses most input and output file formats.

The following list identifies some common codecs that are supported by the Hadoop framework. Be sure to choose the codec that most closely matches the demands of your particular use case (for example, with workloads where the speed of processing is important, choose a codec with high decompression speeds):

- **Gzip**: A compression utility that was adopted by the GNU project, Gzip (short for GNU zip) generates compressed files that have a `.gz` extension. You can use the `gunzip` command to decompress files that were created by a number of compression utilities, including Gzip.
- **Bzip2**: From a usability standpoint, Bzip2 and Gzip are similar. Bzip2 generates a better compression ratio than does Gzip, but it's much slower. In fact, of all the available compression codecs in Hadoop, Bzip2 is by far the slowest. If you're setting up an archive that you'll rarely need to query and space is at a high premium, then maybe would Bzip2 be worth considering. (The *B* in Bzip comes from its use of the Burrows-Wheeler algorithm, in case you're curious.)
- **Snappy**: The Snappy codec from Google provides modest compression ratios, but fast compression and decompression speeds. (In fact, it has the fastest decompression speeds, which makes it highly desirable for data sets that are likely to be queried often.) The Snappy codec is integrated into Hadoop Common, a set of common utilities that supports other Hadoop subprojects. You can use Snappy as an add-on for more recent versions of Hadoop that do not yet provide Snappy codec support.
- **LZO**: Similar to Snappy, LZO (short for Lempel-Ziv-Oberhumer, the trio of computer scientists who came up with the algorithm) provides modest compression ratios, but fast compression and decompression speeds. LZO is licensed under the GNU Public License (GPL). This license is incompatible with the Apache license, and as a result, LZO has been removed from some distributions. (Some distributions, such as IBM's BigInsights, have made an end run around this restriction by releasing GPL-free versions of LZO.)

  LZO supports splittable compression, which, as we mention earlier in this chapter, enables the parallel processing of compressed text file splits by your MapReduce jobs. LZO needs to create an index when it compresses a file, because with variable-length compression blocks, an index is required to tell the mapper where it can safely split the compressed file. LZO is only really desirable if you need to compress text files. For binary files, which are not impacted by non-splittable codecs, Snappy is your best option.

Table 5-1 summarizes the common characteristics of some of the codecs that are supported by the Hadoop framework.

Table 5-1: Hadoop Codecs

Codec	*File Extension*	*Splittable?*	*Degree of Compression*	*Compression Speed*
Gzip	`.gz`	No	Medium	Medium
Bzip2	`.bz2`	Yes	High	Slow
Snappy	`.snappy`	No	Medium	Fast
LZO	`.lzo`	No, unless indexed	Medium	Fast

All compression algorithms must make trade-offs between the degree of compression and the speed of compression that they can achieve. The codecs that are listed in Table 5-1 provide you with some control over what the balance between the compression ratio and speed should be at compression time. For example, Gzip lets you regulate the speed of compression by specifying a negative integer (or keyword), where `-1` (or `--fast`) indicates the fastest compression level, and `-9` (or `--best`) indicates the slowest compression level. The default compression level is –6.

Managing Files with the Hadoop File System Commands

HDFS is one of the two main components of the Hadoop framework; the other is the computational paradigm known as MapReduce. A *distributed file system* is a file system that manages storage across a networked cluster of machines.

HDFS stores data in *blocks*, units whose default size is 64MB. Files that you want stored in HDFS need to be broken into block-size chunks that are then stored independently throughout the cluster. You can use the `fsck` line command to list the blocks that make up each file in HDFS, as follows:

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

REMEMBER Because Hadoop is written in Java, all interactions with HDFS are managed via the Java API. Keep in mind, though, that you don't need to be a Java guru to work with files in HDFS. Several Hadoop interfaces built on top of the Java API are now in common use (and hide Java), but the simplest one is the command-line interface; we use the command line to interact with HDFS in the examples we provide in this chapter.

You access the Hadoop file system shell by running one form of the `hadoop` command. (We tell you more about that topic later.) All `hadoop` commands are invoked by the `bin/hadoop` script. (To retrieve a description of all `hadoop` commands, run the hadoop script without specifying any arguments.) The `hadoop` command has the syntax

```
hadoop [--config confdir] [COMMAND] [GENERIC_OPTIONS]
          [COMMAND_OPTIONS]
```

The `--config confdir option` overwrites the default configuration directory (`$HADOOP_HOME/conf`), so you can easily customize your Hadoop environment configuration. The generic options and command options are a common set of options that are supported by several commands.

Hadoop file system shell commands (for command line interfaces) take uniform resource identifiers (URIs) as arguments. A *URI* is a string of characters that's used to identify a name or a web resource. The string can include a *scheme name* — a qualifier for the nature of the data source. For HDFS, the scheme name is `hdfs`, and for the local file system, the scheme name is `file`. If you don't specify a scheme name, the default is the scheme name that's specified in the configuration file. A file or directory in HDFS can be specified in a fully qualified way,

such as in this example:

```
hdfs://namenodehost/parent/child
```

Or it can simply be `/parent/child` if the configuration file points to `hdfs://namenodehost`.

The Hadoop file system shell commands, which are similar to Linux file commands, have the following general syntax:

```
hadoop hdfs -file_cmd
```

TECHNICAL STUFF Readers with some prior Hadoop experience might ask, "But what about the `hadoop fs` command?" The `fs` command is deprecated in the Hadoop 0.2 release series, but it does still work in Hadoop 2. We recommend that you use `hdfs dfs` instead.

As you might expect, you use the `mkdir` command to create a directory in HDFS, just as you would do on Linux or on Unix-based operating systems. Though HDFS has a default working directory, `/user/$USER`, where `$USER` is your login username, you need to create it yourself by using the syntax

```
$ hadoop hdfs dfs -mkdir /user/login_user_name
```

For example, to create a directory named "`joanna`", run this `mkdir` command:

```
$ hadoop hdfs dfs -mkdir /user/joanna
```

Use the Hadoop `put` command to copy a file from your local file system to HDFS:

```
$ hadoop hdfs dfs -put file_name /user/login_user_name
```

For example, to copy a file named `data.txt` to this new directory, run the following `put` command:

```
$ hadoop hdfs dfs -put data.txt /user/joanna
```

Run the `ls` command to get an HDFS file listing:

```
$ hadoop hdfs dfs -ls .
Found 2 items
drwxr-xr-x - joanna supergroup 0 2013-06-30 12:25 /user/joanna
-rw-r--r-- 1 joanna supergroup 118 2013-06-30 12:15 /user/joanna/data.txt
```

The file listing itself breaks down as described in this list:

- **Column 1 shows the file mode ("d" for directory and "-" for normal file, followed by the permissions)**. The three permission types — read (r), write (w), and execute (x) — are the same as you find on Linux- and Unix-based systems. The execute permission for a file is ignored because you cannot execute a file on HDFS. The permissions are grouped by owner, group, and public (everyone else).
- **Column 2 shows the replication factor for files. (The concept of replication doesn't apply to directories.)** The blocks that make up a file in HDFS are replicated to ensure fault tolerance. The *replication factor*, or the number of replicas that are kept for a specific file, is configurable. You can specify the replication factor when the file is created or later, via your application.
- **Columns 3 and 4 show the file *owner and group. Supergroup*** is the name of the group of superusers, and a *superuser* is the user with the same identity as the NameNode process. If you start the NameNode, you're the superuser for now. This is a special group – regular users will have their userids belong to a group without special characteristics — a group that's simply defined by a Hadoop administrator.
- **Column 5 shows the size of the file, in bytes, or 0 if it's a directory**.
- **Columns 6 and 7 show the date and time of the last modification, respectively**.
- **Column 8 shows the unqualified name (meaning that the scheme name isn't specified) of the file or directory**.

Use the Hadoop `get` command to copy a file from HDFS to your local file system:

```
$ hadoop hdfs dfs -get file_name /user/login_user_name
```

Use the Hadoop `rm` command to delete a file or an empty directory:

```
$ hadoop hdfs dfs -rm file_name /user/login_user_name
```

TIP Use the `hadoop hdfs dfs -help` command to get detailed help for every option.

Table 5-2 summarizes the Hadoop file system shell commands.

Table 5-2: File System Shell Commands

Command	*What It Does*	*Usage*	*Examples*
`dcat`	Copies source paths to `stdout`.	`hdfs dfs -cat URI [URI …]`	`hdfs dfs -cat hdfs://<path>/file1; hdfs dfs -c file:///file2 /user/hadoop/file3`

chgrp	Changes the group association of files. With -R, makes the change recursively by way of the directory structure. The user must be the file owner or the superuser.	hdfs dfs -chgrp [-R] GROUP URI [URI …]	hdfs dfs -chgrp analysts test/data1.txt
chmod	Changes the permissions of files. With -R, makes the change recursively by way of the directory structure. The user must be the file owner or the superuser.	hdfs dfs -chmod [-R] <MODE[,MODE]… | OCTALMODE> URI [URI …]	hdfs dfs -chmod 777 test/data1.txt
chown	Changes the owner of files. With -R, makes the change recursively by way of the directory structure. The user must be the superuser.	hdfs dfs -chown [-R] [OWNER][:[GROUP]] URI [URI]	hdfs dfs -chown -R hduser2 /opt/hadoop/logs
copyFromLocal	Works similarly to the put command, except that the source is restricted to a local file reference.	hdfs dfs -copyFromLocal <localsrc> URI	hdfs dfs -copyFromLocal input/docs/data2.txt hdfs://localhost/user/rosemary/data2.txt
copyToLocal	Works similarly to the get command, except that the destination is restricted to a local file reference.	hdfs dfs -copyToLocal [-ignorecrc] [-crc] URI <localdst>	hdfs dfs -copyToLocal data2.txt data2.copy.txt
count	Counts the number of directories, files, and bytes under the paths that match the specified file pattern.	hdfs dfs -count [-q] <paths>	hdfs dfs -count hdfs://nn1.example.com/file1 hdfs://nn2.example.com/file2
cp	Copies one or more files from a specified source to a specified destination. If you specify multiple sources, the specified destination must be a directory.	hdfs dfs -cp URI [URI …] <dest>	hdfs dfs -cp /user/hadoop/file1 /user/hadoop/file2 /user
du	Displays the size of the specified file, or the sizes of files and directories that are contained in the specified directory. If you specify the -s option, displays an aggregate summary of file sizes rather than individual file sizes. If you specify the -h option, formats the file sizes in a "human-readable" way.	hdfs dfs -du [-s] [-h] URI [URI …]	hdfs dfs -du /user/hadoop/dir1 /user/hadoop/fi
expunge	Empties the trash. When you delete a file, it isn't removed immediately from HDFS, but is renamed to a file in the /trash directory. As long as the file remains there, you can undelete it if you change your mind, though only the latest copy of the deleted file can be restored.	hdfs dfs -expunge	hdfs dfs -expunge
get	Copies files to the local file system. Files that fail a cyclic redundancy check (CRC) can still be copied if you specify the -ignorecrc option. The CRC is a common technique for detecting data transmission errors. CRC checksum files have the .crc extension and are used to verify the data integrity of another file. These files are copied if you specify the -crc option.	hdfs dfs -get [-ignorecrc] [-crc] <src> <localdst>	hdfs dfs -get /user/hadoop/file3 localfile
getmerge	Concatenates the files in src and writes the result to the specified local destination file. To add a newline character at the end of each file, specify the addnl option.	hdfs dfs -getmerge <src> <localdst> [addnl]	hdfs dfs -getmerge/user/hadoop/mydir/~/result_
ls	Returns statistics for the specified files or directories.	hdfs dfs -ls <args>	hdfs dfs -ls /user/hadoop/file1

lsr	Serves as the recursive version of ls; similar to the Unix command ls -R.	hdfs dfs -lsr <args>	hdfs dfs -lsr /user/hadoop
mkdir	Creates directories on one or more specified paths. Its behavior is similar to the Unix mkdir -p command, which creates all directories that lead up to the specified directory if they don't exist already.	hdfs dfs -mkdir <paths>	hdfs dfs -mkdir /user/hadoop/dir5/temp
moveFromLocal	Works similarly to the put command, except that the source is deleted after it is copied.	hdfs dfs -moveFromLocal <localsrc> <dest>	hdfs dfs -moveFromLocal localfile1 localfile2 /user/hadoop/hadoopdir
mv	Moves one or more files from a specified source to a specified destination. If you specify multiple sources, the specified destination must be a directory. Moving files across file systems isn't permitted.	hdfs dfs -mv URI [URI …] <dest>	hdfs dfs -mv /user/hadoop/file1 /user/hadoop/f
put	Copies files from the local file system to the destination file system. This command can also read input from stdin and write to the destination file system.	hdfs dfs -put <localsrc> … <dest>	hdfs dfs -put localfile1 localfile2 /user/hadoop/hadoopdir; hdfs dfs -p /user/hadoop/hadoopdir (reads input from stdin)
rm	Deletes one or more specified files. This command doesn't delete empty directories or files. To bypass the trash (if it's enabled) and delete the specified files immediately, specify the -skipTrash option.	hdfs dfs -rm [-skipTrash] URI [URI …]	hdfs dfs -rm hdfs://nn.example.com/file9
rmr	Serves as the recursive version of -rm.	hdfs dfs -rmr [-skipTrash] URI [URI …]	hdfs dfs -rmr /user/hadoop/dir
setrep	Changes the replication factor for a specified file or directory. With -R, makes the change recursively by way of the directory structure.	hdfs dfs -setrep <rep> [-R] <path>	hdfs dfs -setrep 3 -R /user/hadoop/dir1
stat	Displays information about the specified path.	hdfs dfs -stat URI [URI …]	hdfs dfs -stat /user/hadoop/dir1
tail	Displays the last kilobyte of a specified file to stdout. The syntax supports the Unix -f option, which enables the specified file to be monitored. As new lines are added to the file by another process, tail updates the display.	hdfs dfs -tail [-f] URI	hdfs dfs -tail /user/hadoop/dir1
test	Returns attributes of the specified file or directory. Specifies -e to determine whether the file or directory exists; -z to determine whether the file or directory is empty; and -d to determine whether the URI is a directory.	hdfs dfs -test -[ezd] URI	hdfs dfs -test /user/hadoop/dir1
text	Outputs a specified source file in text format. Valid input file formats are zip and TextRecordInputStream.	hdfs dfs -text <src>	hdfs dfs -text /user/hadoop/file8.zip
touchz	Creates a new, empty file of size 0 in the specified path.	hdfs dfs -touchz <path>	hdfs dfs -touchz /user/hadoop/file12

Ingesting Log Data with Flume

Some of the data that ends up in HDFS might land there via database load operations or other types of batch processes, but what if you want to capture the data that's flowing in high-throughput data streams, such as application log data? Apache Flume is the current standard way to do that easily, efficiently, and safely.

Apache Flume, another top-level project from the Apache Software Foundation, is a distributed system for aggregating and moving large amounts of streaming data from different sources to a centralized data store. Put another way, Flume is designed for the continuous ingestion of data into HDFS. The data can be any kind of data, but Flume is particularly well-suited to handling log data, such as the log data from web servers. Units of the data that Flume processes are called *events*; an example of an event is a log record.

To understand how Flume works within a Hadoop cluster, you need to know that Flume runs as one or more agents, and that each agent has three pluggable components: sources, channels, and sinks, as shown in Figure 5-1 and described in this list:

- *Sources* retrieve data and send it to channels.
- *Channels* hold data queues and serve as conduits between sources and sinks, which is useful when the incoming flow rate exceeds the outgoing flow rate.
- *Sinks* process data that was taken from channels and deliver it to a destination, such as HDFS.

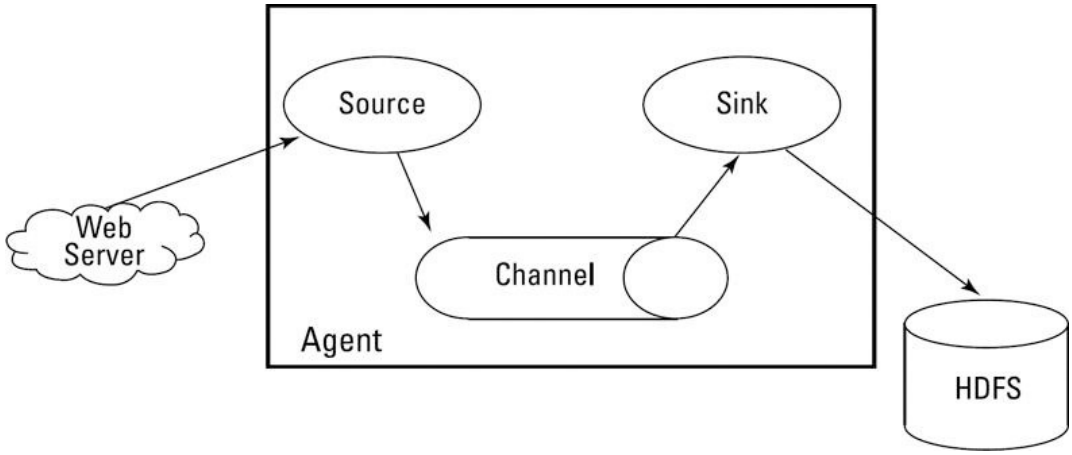

Figure 5-1: The Flume data flow model

An agent must have at least one of each component to run, and each agent is contained within its own instance of the Java Virtual Machine (JVM).

REMEMBER An event that is written to a channel by a source isn't removed from that channel until a sink removes it by way of a transaction. If a network failure occurs, channels keep their events queued until the sinks can write them to the cluster. An in-memory channel can process events quickly, but it is volatile and cannot be recovered, whereas a file-based channel offers persistence and can be recovered in the event of failure.

Each agent can have several sources, channels, and sinks, and although a source can write to many channels, a sink can take data from only one channel.

An agent is just a JVM that's running Flume, and the sinks for each *agent node* in the Hadoop cluster send data to *collector nodes*, which aggregate the data from many agents before writing it to HDFS, where it can be analyzed by other Hadoop tools.

Agents can be chained together so that the sink from one agent sends data to the source from another agent. Avro, Apache's remote call-and-serialization framework, is the usual way of sending data across a network with Flume, because it serves as a useful tool for the efficient serialization or transformation of data into a compact binary format. In the context of Flume, compatibility is important: An Avro event requires an Avro source, for example, and a sink must deliver events that are appropriate to the destination.

What makes this great chain of sources, channels, and sinks work is the Flume agent configuration, which is stored in a local text file that's structured like a Java properties file. You can configure multiple agents in the same file. Let's look at an sample file, which we name `flume-agent.conf` — it's set to configure an agent we named `shaman`:

```
# Identify the components on agent shaman:
shaman.sources = netcat_s1
shaman.sinks = hdfs_w1
shaman.channels = in-mem_c1

# Configure the source:
shaman.sources.netcat_s1.type = netcat
shaman.sources.netcat_s1.bind = localhost
shaman.sources.netcat_s1.port = 44444

# Describe the sink:
shaman.sinks.hdfs_w1.type = hdfs
shaman.sinks.hdfs_w1.hdfs.path = hdfs://<path>
```

```
shaman.sinks.hdfs_w1.hdfs.writeFormat = Text
shaman.sinks.hdfs_w1.hdfs.fileType = DataStream

# Configure a channel that buffers events in memory:
shaman.channels.in-mem_c1.type = memory
shaman.channels.in-mem_c1.capacity = 20000
shaman.channels.in-mem_c1.transactionCapacity = 100

# Bind the source and sink to the channel:
shaman.sources.netcat_s1.channels = in-mem_c1
shaman.sinks.hdfs_w1.channels = in-mem_c1
```

The configuration file includes properties for each source, channel, and sink in the agent and specifies how they're connected. In this example, agent `shaman` has a source that listens for data (messages to `netcat`) on port 44444, a channel that buffers event data in memory, and a sink that logs event data to the console. This configuration file could have been used to define several agents; we're configuring only one to keep things simple.

To start the agent, use a shell script called `flume-ng`, which is located in the `bin` directory of the Flume distribution. From the command line, issue the `agent` command, specifying the path to the configuration file and the agent name.

The following sample command starts the Flume agent that we showed you how to configure:

```
flume-ng agent -f /<path to flume-agent.conf> -n shaman
```

The Flume agent's log should have entries verifying that the source, channel, and sink started successfully.

To further test the configuration, you can telnet to port 44444 from another terminal and send Flume an event by entering an arbitrary text string. If all goes well, the original Flume terminal will output the event in a log message that you should be able to see in the agent's log.