PART I
Hadoop Fundamentals

CHAPTER 1
Meet Hadoop

> In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers.
>
> —Grace Hopper

Data!

We live in the data age. It's not easy to measure the total volume of data stored electronically, but an IDC estimate put the size of the "digital universe" at 4.4 zettabytes in 2013 and is forecasting a tenfold growth by 2020 to 44 zettabytes.[1] A zettabyte is 10^{21} bytes, or equivalently one thousand exabytes, one million petabytes, or one billion terabytes. That's more than one disk drive for every person in the world.

This flood of data is coming from many sources. Consider the following:[2]

- The New York Stock Exchange generates about 4–5 terabytes of data per day.
- Facebook hosts more than 240 billion photos, growing at 7 petabytes per month.
- Ancestry.com, the genealogy site, stores around 10 petabytes of data.
- The Internet Archive stores around 18.5 petabytes of data.

1. These statistics were reported in a study entitled "The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things."
2. All figures are from 2013 or 2014. For more information, see Tom Groenfeldt, "At NYSE, The Data Deluge Overwhelms Traditional Databases"; Rich Miller, "Facebook Builds Exabyte Data Centers for Cold Storage"; Ancestry.com's "Company Facts"; Archive.org's "Petabox"; and the Worldwide LHC Computing Grid project's welcome page.

- The Large Hadron Collider near Geneva, Switzerland, produces about 30 petabytes of data per year.

So there's a lot of data out there. But you are probably wondering how it affects you. Most of the data is locked up in the largest web properties (like search engines) or in scientific or financial institutions, isn't it? Does the advent of big data affect smaller organizations or individuals?

I argue that it does. Take photos, for example. My wife's grandfather was an avid photographer and took photographs throughout his adult life. His entire corpus of medium-format, slide, and 35mm film, when scanned in at high resolution, occupies around 10 gigabytes. Compare this to the digital photos my family took in 2008, which take up about 5 gigabytes of space. My family is producing photographic data at 35 times the rate my wife's grandfather's did, and the rate is increasing every year as it becomes easier to take more and more photos.

More generally, the digital streams that individuals are producing are growing apace. Microsoft Research's MyLifeBits project gives a glimpse of the archiving of personal information that may become commonplace in the near future. MyLifeBits was an experiment where an individual's interactions—phone calls, emails, documents—were captured electronically and stored for later access. The data gathered included a photo taken every minute, which resulted in an overall data volume of 1 gigabyte per month. When storage costs come down enough to make it feasible to store continuous audio and video, the data volume for a future MyLifeBits service will be many times that.

The trend is for every individual's data footprint to grow, but perhaps more significantly, the amount of data generated by machines as a part of the Internet of Things will be even greater than that generated by people. Machine logs, RFID readers, sensor networks, vehicle GPS traces, retail transactions—all of these contribute to the growing mountain of data.

The volume of data being made publicly available increases every year, too. Organizations no longer have to merely manage their own data; success in the future will be dictated to a large extent by their ability to extract value from other organizations' data.

Initiatives such as Public Data Sets on Amazon Web Services and Infochimps.org exist to foster the "information commons," where data can be freely (or for a modest price) shared for anyone to download and analyze. Mashups between different information sources make for unexpected and hitherto unimaginable applications.

Take, for example, the Astrometry.net project, which watches the Astrometry group on Flickr for new photos of the night sky. It analyzes each image and identifies which part of the sky it is from, as well as any interesting celestial bodies, such as stars or galaxies. This project shows the kinds of things that are possible when data (in this case, tagged photographic images) is made available and used for something (image analysis) that was not anticipated by the creator.

It has been said that "more data usually beats better algorithms," which is to say that for some problems (such as recommending movies or music based on past preferences), however fiendish your algorithms, often they can be beaten simply by having more data (and a less sophisticated algorithm).[3]

The good news is that big data is here. The bad news is that we are struggling to store and analyze it.

Data Storage and Analysis

The problem is simple: although the storage capacities of hard drives have increased massively over the years, access speeds—the rate at which data can be read from drives—have not kept up. One typical drive from 1990 could store 1,370 MB of data and had a transfer speed of 4.4 MB/s,[4] so you could read all the data from a full drive in around five minutes. Over 20 years later, 1-terabyte drives are the norm, but the transfer speed is around 100 MB/s, so it takes more than two and a half hours to read all the data off the disk.

This is a long time to read all data on a single drive—and writing is even slower. The obvious way to reduce the time is to read from multiple disks at once. Imagine if we had 100 drives, each holding one hundredth of the data. Working in parallel, we could read the data in under two minutes.

Using only one hundredth of a disk may seem wasteful. But we can store 100 datasets, each of which is 1 terabyte, and provide shared access to them. We can imagine that the users of such a system would be happy to share access in return for shorter analysis times, and statistically, that their analysis jobs would be likely to be spread over time, so they wouldn't interfere with each other too much.

There's more to being able to read and write data in parallel to or from multiple disks, though.

The first problem to solve is hardware failure: as soon as you start using many pieces of hardware, the chance that one will fail is fairly high. A common way of avoiding data loss is through replication: redundant copies of the data are kept by the system so that in the event of failure, there is another copy available. This is how RAID works, for instance, although Hadoop's filesystem, the Hadoop Distributed Filesystem (HDFS), takes a slightly different approach, as you shall see later.

3. The quote is from Anand Rajaraman's blog post "More data usually beats better algorithms," in which he writes about the Netflix Challenge. Alon Halevy, Peter Norvig, and Fernando Pereira make the same point in "The Unreasonable Effectiveness of Data," *IEEE Intelligent Systems*, March/April 2009.

4. These specifications are for the Seagate ST-41600n.

The second problem is that most analysis tasks need to be able to combine the data in some way, and data read from one disk may need to be combined with data from any of the other 99 disks. Various distributed systems allow data to be combined from multiple sources, but doing this correctly is notoriously challenging. MapReduce provides a programming model that abstracts the problem from disk reads and writes, transforming it into a computation over sets of keys and values. We look at the details of this model in later chapters, but the important point for the present discussion is that there are two parts to the computation—the map and the reduce—and it's the interface between the two where the "mixing" occurs. Like HDFS, MapReduce has built-in reliability.

In a nutshell, this is what Hadoop provides: a reliable, scalable platform for storage and analysis. What's more, because it runs on commodity hardware and is open source, Hadoop is affordable.

Querying All Your Data

The approach taken by MapReduce may seem like a brute-force approach. The premise is that the entire dataset—or at least a good portion of it—can be processed for each query. But this is its power. MapReduce is a *batch* query processor, and the ability to run an ad hoc query against your whole dataset and get the results in a reasonable time is transformative. It changes the way you think about data and unlocks data that was previously archived on tape or disk. It gives people the opportunity to innovate with data. Questions that took too long to get answered before can now be answered, which in turn leads to new questions and new insights.

For example, Mailtrust, Rackspace's mail division, used Hadoop for processing email logs. One ad hoc query they wrote was to find the geographic distribution of their users. In their words:

> This data was so useful that we've scheduled the MapReduce job to run monthly and we will be using this data to help us decide which Rackspace data centers to place new mail servers in as we grow.

By bringing several hundred gigabytes of data together and having the tools to analyze it, the Rackspace engineers were able to gain an understanding of the data that they otherwise would never have had, and furthermore, they were able to use what they had learned to improve the service for their customers.

Beyond Batch

For all its strengths, MapReduce is fundamentally a batch processing system, and is not suitable for interactive analysis. You can't run a query and get results back in a few seconds or less. Queries typically take minutes or more, so it's best for offline use, where there isn't a human sitting in the processing loop waiting for results.

However, since its original incarnation, Hadoop has evolved beyond batch processing. Indeed, the term "Hadoop" is sometimes used to refer to a larger ecosystem of projects, not just HDFS and MapReduce, that fall under the umbrella of infrastructure for distributed computing and large-scale data processing. Many of these are hosted by the Apache Software Foundation, which provides support for a community of open source software projects, including the original HTTP Server from which it gets its name.

The first component to provide online access was HBase, a key-value store that uses HDFS for its underlying storage. HBase provides both online read/write access of individual rows and batch operations for reading and writing data in bulk, making it a good solution for building applications on.

The real enabler for new processing models in Hadoop was the introduction of YARN (which stands for *Yet Another Resource Negotiator*) in Hadoop 2. YARN is a cluster resource management system, which allows any distributed program (not just MapReduce) to run on data in a Hadoop cluster.

In the last few years, there has been a flowering of different processing patterns that work with Hadoop. Here is a sample:

Interactive SQL
: By dispensing with MapReduce and using a distributed query engine that uses dedicated "always on" daemons (like Impala) or container reuse (like Hive on Tez), it's possible to achieve low-latency responses for SQL queries on Hadoop while still scaling up to large dataset sizes.

Iterative processing
: Many algorithms—such as those in machine learning—are iterative in nature, so it's much more efficient to hold each intermediate working set in memory, compared to loading from disk on each iteration. The architecture of MapReduce does not allow this, but it's straightforward with Spark, for example, and it enables a highly exploratory style of working with datasets.

Stream processing
: Streaming systems like Storm, Spark Streaming, or Samza make it possible to run real-time, distributed computations on unbounded streams of data and emit results to Hadoop storage or external systems.

Search
: The Solr search platform can run on a Hadoop cluster, indexing documents as they are added to HDFS, and serving search queries from indexes stored in HDFS.

Despite the emergence of different processing frameworks on Hadoop, MapReduce still has a place for batch processing, and it is useful to understand how it works since it introduces several concepts that apply more generally (like the idea of input formats, or how a dataset is split into pieces).

Comparison with Other Systems

Hadoop isn't the first distributed system for data storage and analysis, but it has some unique properties that set it apart from other systems that may seem similar. Here we look at some of them.

Relational Database Management Systems

Why can't we use databases with lots of disks to do large-scale analysis? Why is Hadoop needed?

The answer to these questions comes from another trend in disk drives: seek time is improving more slowly than transfer rate. Seeking is the process of moving the disk's head to a particular place on the disk to read or write data. It characterizes the latency of a disk operation, whereas the transfer rate corresponds to a disk's bandwidth.

If the data access pattern is dominated by seeks, it will take longer to read or write large portions of the dataset than streaming through it, which operates at the transfer rate. On the other hand, for updating a small proportion of records in a database, a traditional B-Tree (the data structure used in relational databases, which is limited by the rate at which it can perform seeks) works well. For updating the majority of a database, a B-Tree is less efficient than MapReduce, which uses Sort/Merge to rebuild the database.

In many ways, MapReduce can be seen as a complement to a Relational Database Management System (RDBMS). (The differences between the two systems are shown in Table 1-1.) MapReduce is a good fit for problems that need to analyze the whole dataset in a batch fashion, particularly for ad hoc analysis. An RDBMS is good for point queries or updates, where the dataset has been indexed to deliver low-latency retrieval and update times of a relatively small amount of data. MapReduce suits applications where the data is written once and read many times, whereas a relational database is good for datasets that are continually updated.[5]

Table 1-1. RDBMS compared to MapReduce

	Traditional RDBMS	MapReduce
Data size	Gigabytes	Petabytes
Access	Interactive and batch	Batch
Updates	Read and write many times	Write once, read many times
Transactions	ACID	None

5. In January 2007, David J. DeWitt and Michael Stonebraker caused a stir by publishing "MapReduce: A major step backwards," in which they criticized MapReduce for being a poor substitute for relational databases. Many commentators argued that it was a false comparison (see, for example, Mark C. Chu-Carroll's "Databases are hammers; MapReduce is a screwdriver"), and DeWitt and Stonebraker followed up with "MapReduce II," where they addressed the main topics brought up by others.

	Traditional RDBMS	MapReduce
Structure	Schema-on-write	Schema-on-read
Integrity	High	Low
Scaling	Nonlinear	Linear

However, the differences between relational databases and Hadoop systems are blurring. Relational databases have started incorporating some of the ideas from Hadoop, and from the other direction, Hadoop systems such as Hive are becoming more interactive (by moving away from MapReduce) and adding features like indexes and transactions that make them look more and more like traditional RDBMSs.

Another difference between Hadoop and an RDBMS is the amount of structure in the datasets on which they operate. *Structured data* is organized into entities that have a defined format, such as XML documents or database tables that conform to a particular predefined schema. This is the realm of the RDBMS. *Semi-structured data*, on the other hand, is looser, and though there may be a schema, it is often ignored, so it may be used only as a guide to the structure of the data: for example, a spreadsheet, in which the structure is the grid of cells, although the cells themselves may hold any form of data. *Unstructured data* does not have any particular internal structure: for example, plain text or image data. Hadoop works well on unstructured or semi-structured data because it is designed to interpret the data at processing time (so called *schema-on-read*). This provides flexibility and avoids the costly data loading phase of an RDBMS, since in Hadoop it is just a file copy.

Relational data is often *normalized* to retain its integrity and remove redundancy. Normalization poses problems for Hadoop processing because it makes reading a record a nonlocal operation, and one of the central assumptions that Hadoop makes is that it is possible to perform (high-speed) streaming reads and writes.

A web server log is a good example of a set of records that is *not* normalized (for example, the client hostnames are specified in full each time, even though the same client may appear many times), and this is one reason that logfiles of all kinds are particularly well suited to analysis with Hadoop. Note that Hadoop can perform joins; it's just that they are not used as much as in the relational world.

MapReduce—and the other processing models in Hadoop—scales linearly with the size of the data. Data is partitioned, and the functional primitives (like map and reduce) can work in parallel on separate partitions. This means that if you double the size of the input data, a job will run twice as slowly. But if you also double the size of the cluster, a job will run as fast as the original one. This is not generally true of SQL queries.

Grid Computing

The high-performance computing (HPC) and grid computing communities have been doing large-scale data processing for years, using such application program interfaces (APIs) as the Message Passing Interface (MPI). Broadly, the approach in HPC is to distribute the work across a cluster of machines, which access a shared filesystem, hosted by a storage area network (SAN). This works well for predominantly compute-intensive jobs, but it becomes a problem when nodes need to access larger data volumes (hundreds of gigabytes, the point at which Hadoop really starts to shine), since the network bandwidth is the bottleneck and compute nodes become idle.

Hadoop tries to co-locate the data with the compute nodes, so data access is fast because it is local.[6] This feature, known as *data locality*, is at the heart of data processing in Hadoop and is the reason for its good performance. Recognizing that network bandwidth is the most precious resource in a data center environment (it is easy to saturate network links by copying data around), Hadoop goes to great lengths to conserve it by explicitly modeling network topology. Notice that this arrangement does not preclude high-CPU analyses in Hadoop.

MPI gives great control to programmers, but it requires that they explicitly handle the mechanics of the data flow, exposed via low-level C routines and constructs such as sockets, as well as the higher-level algorithms for the analyses. Processing in Hadoop operates only at the higher level: the programmer thinks in terms of the data model (such as key-value pairs for MapReduce), while the data flow remains implicit.

Coordinating the processes in a large-scale distributed computation is a challenge. The hardest aspect is gracefully handling partial failure—when you don't know whether or not a remote process has failed—and still making progress with the overall computation. Distributed processing frameworks like MapReduce spare the programmer from having to think about failure, since the implementation detects failed tasks and reschedules replacements on machines that are healthy. MapReduce is able to do this because it is a *shared-nothing* architecture, meaning that tasks have no dependence on one other. (This is a slight oversimplification, since the output from mappers is fed to the reducers, but this is under the control of the MapReduce system; in this case, it needs to take more care rerunning a failed reducer than rerunning a failed map, because it has to make sure it can retrieve the necessary map outputs and, if not, regenerate them by running the relevant maps again.) So from the programmer's point of view, the order in which the tasks run doesn't matter. By contrast, MPI programs have to explicitly manage their own checkpointing and recovery, which gives more control to the programmer but makes them more difficult to write.

6. Jim Gray was an early advocate of putting the computation near the data. See "Distributed Computing Economics," March 2003.

Volunteer Computing

When people first hear about Hadoop and MapReduce they often ask, "How is it different from SETI@home?" SETI, the Search for Extra-Terrestrial Intelligence, runs a project called SETI@home in which volunteers donate CPU time from their otherwise idle computers to analyze radio telescope data for signs of intelligent life outside Earth. SETI@home is the most well known of many *volunteer computing* projects; others include the Great Internet Mersenne Prime Search (to search for large prime numbers) and Folding@home (to understand protein folding and how it relates to disease).

Volunteer computing projects work by breaking the problems they are trying to solve into chunks called *work units*, which are sent to computers around the world to be analyzed. For example, a SETI@home work unit is about 0.35 MB of radio telescope data, and takes hours or days to analyze on a typical home computer. When the analysis is completed, the results are sent back to the server, and the client gets another work unit. As a precaution to combat cheating, each work unit is sent to three different machines and needs at least two results to agree to be accepted.

Although SETI@home may be superficially similar to MapReduce (breaking a problem into independent pieces to be worked on in parallel), there are some significant differences. The SETI@home problem is very CPU-intensive, which makes it suitable for running on hundreds of thousands of computers across the world[7] because the time to transfer the work unit is dwarfed by the time to run the computation on it. Volunteers are donating CPU cycles, not bandwidth.

7. In January 2008, SETI@home was reported to be processing 300 gigabytes a day, using 320,000 computers (most of which are not dedicated to SETI@home; they are used for other things, too).

MapReduce is designed to run jobs that last minutes or hours on trusted, dedicated hardware running in a single data center with very high aggregate bandwidth interconnects. By contrast, SETI@home runs a perpetual computation on untrusted machines on the Internet with highly variable connection speeds and no data locality.

A Brief History of Apache Hadoop

Hadoop was created by Doug Cutting, the creator of Apache Lucene, the widely used text search library. Hadoop has its origins in Apache Nutch, an open source web search engine, itself a part of the Lucene project.

The Origin of the Name "Hadoop"

The name Hadoop is not an acronym; it's a made-up name. The project's creator, Doug Cutting, explains how the name came about:

> The name my kid gave a stuffed yellow elephant. Short, relatively easy to spell and pronounce, meaningless, and not used elsewhere: those are my naming criteria. Kids are good at generating such. Googol is a kid's term.

Projects in the Hadoop ecosystem also tend to have names that are unrelated to their function, often with an elephant or other animal theme ("Pig," for example). Smaller components are given more descriptive (and therefore more mundane) names. This is a good principle, as it means you can generally work out what something does from its name. For example, the namenode[8] manages the filesystem namespace.

Building a web search engine from scratch was an ambitious goal, for not only is the software required to crawl and index websites complex to write, but it is also a challenge to run without a dedicated operations team, since there are so many moving parts. It's expensive, too: Mike Cafarella and Doug Cutting estimated a system supporting a one-billion-page index would cost around $500,000 in hardware, with a monthly running cost of $30,000.[9] Nevertheless, they believed it was a worthy goal, as it would open up and ultimately democratize search engine algorithms.

Nutch was started in 2002, and a working crawler and search system quickly emerged. However, its creators realized that their architecture wouldn't scale to the billions of pages on the Web. Help was at hand with the publication of a paper in 2003 that described the architecture of Google's distributed filesystem, called GFS, which was being used in

8. In this book, we use the lowercase form, "namenode," to denote the entity when it's being referred to generally, and the CamelCase form `NameNode` to denote the Java class that implements it.

9. See Mike Cafarella and Doug Cutting, "Building Nutch: Open Source Search," *ACM Queue*, April 2004.

production at Google.[10] GFS, or something like it, would solve their storage needs for the very large files generated as a part of the web crawl and indexing process. In particular, GFS would free up time being spent on administrative tasks such as managing storage nodes. In 2004, Nutch's developers set about writing an open source implementation, the Nutch Distributed Filesystem (NDFS).

In 2004, Google published the paper that introduced MapReduce to the world.[11] Early in 2005, the Nutch developers had a working MapReduce implementation in Nutch, and by the middle of that year all the major Nutch algorithms had been ported to run using MapReduce and NDFS.

NDFS and the MapReduce implementation in Nutch were applicable beyond the realm of search, and in February 2006 they moved out of Nutch to form an independent subproject of Lucene called Hadoop. At around the same time, Doug Cutting joined Yahoo!, which provided a dedicated team and the resources to turn Hadoop into a system that ran at web scale (see the following sidebar). This was demonstrated in February 2008 when Yahoo! announced that its production search index was being generated by a 10,000-core Hadoop cluster.[12]

Hadoop at Yahoo!

Building Internet-scale search engines requires huge amounts of data and therefore large numbers of machines to process it. Yahoo! Search consists of four primary components: the *Crawler*, which downloads pages from web servers; the *WebMap*, which builds a graph of the known Web; the *Indexer*, which builds a reverse index to the best pages; and the *Runtime*, which answers users' queries. The WebMap is a graph that consists of roughly 1 trillion (10^{12}) edges, each representing a web link, and 100 billion (10^{11}) nodes, each representing distinct URLs. Creating and analyzing such a large graph requires a large number of computers running for many days. In early 2005, the infrastructure for the WebMap, named *Dreadnaught*, needed to be redesigned to scale up to more nodes. Dreadnaught had successfully scaled from 20 to 600 nodes, but required a complete redesign to scale out further. Dreadnaught is similar to MapReduce in many ways, but provides more flexibility and less structure. In particular, each fragment in a Dreadnaught job could send output to each of the fragments in the next stage of the job, but the sort was all done in library code. In practice, most of the WebMap phases were pairs that corresponded to MapReduce. Therefore, the WebMap applications would not require extensive refactoring to fit into MapReduce.

10. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google File System," October 2003.
11. Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," December 2004.
12. "Yahoo! Launches World's Largest Hadoop Production Application," February 19, 2008.

Eric Baldeschwieler (aka Eric14) created a small team, and we started designing and prototyping a new framework, written in C++ modeled and after GFS and MapReduce, to replace Dreadnaught. Although the immediate need was for a new framework for WebMap, it was clear that standardization of the batch platform across Yahoo! Search was critical and that by making the framework general enough to support other users, we could better leverage investment in the new platform.

At the same time, we were watching Hadoop, which was part of Nutch, and its progress. In January 2006, Yahoo! hired Doug Cutting, and a month later we decided to abandon our prototype and adopt Hadoop. The advantage of Hadoop over our prototype and design was that it was already working with a real application (Nutch) on 20 nodes. That allowed us to bring up a research cluster two months later and start helping real customers use the new framework much sooner than we could have otherwise. Another advantage, of course, was that since Hadoop was already open source, it was easier (although far from easy!) to get permission from Yahoo!'s legal department to work in open source. So, we set up a 200-node cluster for the researchers in early 2006 and put the WebMap conversion plans on hold while we supported and improved Hadoop for the research users.

—Owen O'Malley, 2009

In January 2008, Hadoop was made its own top-level project at Apache, confirming its success and its diverse, active community. By this time, Hadoop was being used by many other companies besides Yahoo!, such as Last.fm, Facebook, and the *New York Times*.

In one well-publicized feat, the *New York Times* used Amazon's EC2 compute cloud to crunch through 4 terabytes of scanned archives from the paper, converting them to PDFs for the Web.[13] The processing took less than 24 hours to run using 100 machines, and the project probably wouldn't have been embarked upon without the combination of Amazon's pay-by-the-hour model (which allowed the *NYT* to access a large number of machines for a short period) and Hadoop's easy-to-use parallel programming model.

In April 2008, Hadoop broke a world record to become the fastest system to sort an entire terabyte of data. Running on a 910-node cluster, Hadoop sorted 1 terabyte in 209 seconds (just under 3.5 minutes), beating the previous year's winner of 297 seconds.[14] In November of the same year, Google reported that its MapReduce implementation sorted 1 terabyte in 68 seconds.[15] Then, in April 2009, it was announced that a team at Yahoo! had used Hadoop to sort 1 terabyte in 62 seconds.[16]

13. Derek Gottfrid, "Self-Service, Prorated Super Computing Fun!" November 1, 2007.
14. Owen O'Malley, "TeraByte Sort on Apache Hadoop," May 2008.
15. Grzegorz Czajkowski, "Sorting 1PB with MapReduce," November 21, 2008.
16. Owen O'Malley and Arun C. Murthy, "Winning a 60 Second Dash with a Yellow Elephant," April 2009.

The trend since then has been to sort even larger volumes of data at ever faster rates. In the 2014 competition, a team from Databricks were joint winners of the Gray Sort benchmark. They used a 207-node Spark cluster to sort 100 terabytes of data in 1,406 seconds, a rate of 4.27 terabytes per minute.[17]

Today, Hadoop is widely used in mainstream enterprises. Hadoop's role as a general-purpose storage and analysis platform for big data has been recognized by the industry, and this fact is reflected in the number of products that use or incorporate Hadoop in some way. Commercial Hadoop support is available from large, established enterprise vendors, including EMC, IBM, Microsoft, and Oracle, as well as from specialist Hadoop companies such as Cloudera, Hortonworks, and MapR.

What's in This Book?

The book is divided into five main parts: Parts I to III are about core Hadoop, Part IV covers related projects in the Hadoop ecosystem, and Part V contains Hadoop case studies. You can read the book from cover to cover, but there are alternative pathways through the book that allow you to skip chapters that aren't needed to read later ones. See Figure 1-1.

Part I is made up of five chapters that cover the fundamental components in Hadoop and should be read before tackling later chapters. Chapter 1 (this chapter) is a high-level introduction to Hadoop. Chapter 2 provides an introduction to MapReduce. Chapter 3 looks at Hadoop filesystems, and in particular HDFS, in depth. Chapter 4 discusses YARN, Hadoop's cluster resource management system. Chapter 5 covers the I/O building blocks in Hadoop: data integrity, compression, serialization, and file-based data structures.

Part II has four chapters that cover MapReduce in depth. They provide useful understanding for later chapters (such as the data processing chapters in Part IV), but could be skipped on a first reading. Chapter 6 goes through the practical steps needed to develop a MapReduce application. Chapter 7 looks at how MapReduce is implemented in Hadoop, from the point of view of a user. Chapter 8 is about the MapReduce programming model and the various data formats that MapReduce can work with. Chapter 9 is on advanced MapReduce topics, including sorting and joining data.

Part III concerns the administration of Hadoop: Chapters 10 and 11 describe how to set up and maintain a Hadoop cluster running HDFS and MapReduce on YARN.

Part IV of the book is dedicated to projects that build on Hadoop or are closely related to it. Each chapter covers one project and is largely independent of the other chapters in this part, so they can be read in any order.

17. Reynold Xin et al., "GraySort on Apache Spark by Databricks," November 2014.

The first two chapters in this part are about data formats. Chapter 12 looks at Avro, a cross-language data serialization library for Hadoop, and Chapter 13 covers Parquet, an efficient columnar storage format for nested data.

The next two chapters look at data ingestion, or how to get your data into Hadoop. Chapter 14 is about Flume, for high-volume ingestion of streaming data. Chapter 15 is about Sqoop, for efficient bulk transfer of data between structured data stores (like relational databases) and HDFS.

The common theme of the next four chapters is data processing, and in particular using higher-level abstractions than MapReduce. Pig (Chapter 16) is a data flow language for exploring very large datasets. Hive (Chapter 17) is a data warehouse for managing data stored in HDFS and provides a query language based on SQL. Crunch (Chapter 18) is a high-level Java API for writing data processing pipelines that can run on MapReduce or Spark. Spark (Chapter 19) is a cluster computing framework for large-scale data processing; it provides a *directed acyclic graph* (DAG) engine, and APIs in Scala, Java, and Python.

Chapter 20 is an introduction to HBase, a distributed column-oriented real-time database that uses HDFS for its underlying storage. And Chapter 21 is about ZooKeeper, a distributed, highly available coordination service that provides useful primitives for building distributed applications.

Finally, Part V is a collection of case studies contributed by people using Hadoop in interesting ways.

Supplementary information about Hadoop, such as how to install it on your machine, can be found in the appendixes.

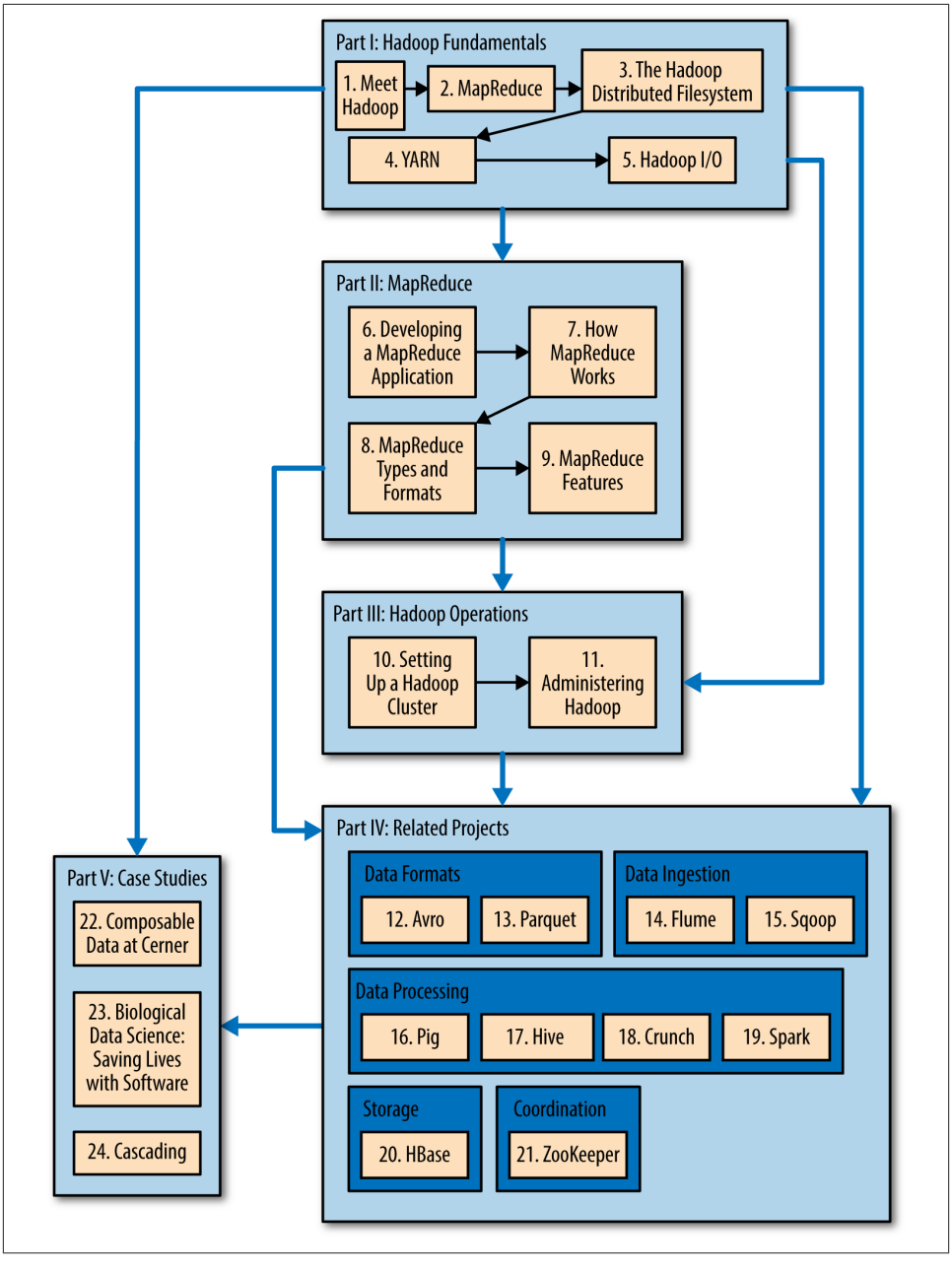

Figure 1-1. Structure of the book: there are various pathways through the content

CHAPTER 2
MapReduce

MapReduce is a programming model for data processing. The model is simple, yet not too simple to express useful programs in. Hadoop can run MapReduce programs written in various languages; in this chapter, we look at the same program expressed in Java, Ruby, and Python. Most importantly, MapReduce programs are inherently parallel, thus putting very large-scale data analysis into the hands of anyone with enough machines at their disposal. MapReduce comes into its own for large datasets, so let's start by looking at one.

A Weather Dataset

For our example, we will write a program that mines weather data. Weather sensors collect data every hour at many locations across the globe and gather a large volume of log data, which is a good candidate for analysis with MapReduce because we want to process all the data, and the data is semi-structured and record-oriented.

Data Format

The data we will use is from the National Climatic Data Center, or NCDC. The data is stored using a line-oriented ASCII format, in which each line is a record. The format supports a rich set of meteorological elements, many of which are optional or with variable data lengths. For simplicity, we focus on the basic elements, such as temperature, which are always present and are of fixed width.

Example 2-1 shows a sample line with some of the salient fields annotated. The line has been split into multiple lines to show each field; in the real file, fields are packed into one line with no delimiters.

Example 2-1. Format of a National Climatic Data Center record

```
0057
332130   # USAF weather station identifier
99999    # WBAN weather station identifier
19500101 # observation date
0300     # observation time
4
+51317   # latitude (degrees x 1000)
+028783  # longitude (degrees x 1000)
FM-12
+0171    # elevation (meters)
99999
V020
320      # wind direction (degrees)
1        # quality code
N
0072
1
00450    # sky ceiling height (meters)
1        # quality code
C
N
010000   # visibility distance (meters)
1        # quality code
N
9
-0128    # air temperature (degrees Celsius x 10)
1        # quality code
-0139    # dew point temperature (degrees Celsius x 10)
1        # quality code
10268    # atmospheric pressure (hectopascals x 10)
1        # quality code
```

Datafiles are organized by date and weather station. There is a directory for each year from 1901 to 2001, each containing a gzipped file for each weather station with its readings for that year. For example, here are the first entries for 1990:

```
% ls raw/1990 | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
010040-99999-1990.gz
010080-99999-1990.gz
010100-99999-1990.gz
010150-99999-1990.gz
```

There are tens of thousands of weather stations, so the whole dataset is made up of a large number of relatively small files. It's generally easier and more efficient to process

a smaller number of relatively large files, so the data was preprocessed so that each year's readings were concatenated into a single file. (The means by which this was carried out is described in Appendix C.)

Analyzing the Data with Unix Tools

What's the highest recorded global temperature for each year in the dataset? We will answer this first without using Hadoop, as this information will provide a performance baseline and a useful means to check our results.

The classic tool for processing line-oriented data is *awk*. Example 2-2 is a small script to calculate the maximum temperature for each year.

Example 2-2. A program for finding the maximum recorded temperature by year from NCDC weather records

```
#!/usr/bin/env bash
for year in all/*
do
  echo -ne `basename $year .gz`"\t"
  gunzip -c $year | \
    awk '{ temp = substr($0, 88, 5) + 0;
           q = substr($0, 93, 1);
           if (temp !=9999 && q ~ /[01459]/ && temp > max) max = temp }
         END { print max }'
done
```

The script loops through the compressed year files, first printing the year, and then processing each file using *awk*. The *awk* script extracts two fields from the data: the air temperature and the quality code. The air temperature value is turned into an integer by adding 0. Next, a test is applied to see whether the temperature is valid (the value 9999 signifies a missing value in the NCDC dataset) and whether the quality code indicates that the reading is not suspect or erroneous. If the reading is OK, the value is compared with the maximum value seen so far, which is updated if a new maximum is found. The `END` block is executed after all the lines in the file have been processed, and it prints the maximum value.

Here is the beginning of a run:

```
% ./max_temperature.sh
1901 317
1902 244
1903 289
1904 256
1905 283
...
```

The temperature values in the source file are scaled by a factor of 10, so this works out as a maximum temperature of 31.7°C for 1901 (there were very few readings at the

beginning of the century, so this is plausible). The complete run for the century took 42 minutes in one run on a single EC2 High-CPU Extra Large instance.

To speed up the processing, we need to run parts of the program in parallel. In theory, this is straightforward: we could process different years in different processes, using all the available hardware threads on a machine. There are a few problems with this, however.

First, dividing the work into equal-size pieces isn't always easy or obvious. In this case, the file size for different years varies widely, so some processes will finish much earlier than others. Even if they pick up further work, the whole run is dominated by the longest file. A better approach, although one that requires more work, is to split the input into fixed-size chunks and assign each chunk to a process.

Second, combining the results from independent processes may require further processing. In this case, the result for each year is independent of other years, and they may be combined by concatenating all the results and sorting by year. If using the fixed-size chunk approach, the combination is more delicate. For this example, data for a particular year will typically be split into several chunks, each processed independently. We'll end up with the maximum temperature for each chunk, so the final step is to look for the highest of these maximums for each year.

Third, you are still limited by the processing capacity of a single machine. If the best time you can achieve is 20 minutes with the number of processors you have, then that's it. You can't make it go faster. Also, some datasets grow beyond the capacity of a single machine. When we start using multiple machines, a whole host of other factors come into play, mainly falling into the categories of coordination and reliability. Who runs the overall job? How do we deal with failed processes?

So, although it's feasible to parallelize the processing, in practice it's messy. Using a framework like Hadoop to take care of these issues is a great help.

Analyzing the Data with Hadoop

To take advantage of the parallel processing that Hadoop provides, we need to express our query as a MapReduce job. After some local, small-scale testing, we will be able to run it on a cluster of machines.

Map and Reduce

MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer. The programmer also specifies two functions: the map function and the reduce function.

The input to our map phase is the raw NCDC data. We choose a text input format that gives us each line in the dataset as a text value. The key is the offset of the beginning of the line from the beginning of the file, but as we have no need for this, we ignore it.

Our map function is simple. We pull out the year and the air temperature, because these are the only fields we are interested in. In this case, the map function is just a data preparation phase, setting up the data in such a way that the reduce function can do its work on it: finding the maximum temperature for each year. The map function is also a good place to drop bad records: here we filter out temperatures that are missing, suspect, or erroneous.

To visualize the way the map works, consider the following sample lines of input data (some unused columns have been dropped to fit the page, indicated by ellipses):

```
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
```

These lines are presented to the map function as the key-value pairs:

```
(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

The keys are the line offsets within the file, which we ignore in our map function. The map function merely extracts the year and the air temperature (indicated in bold text), and emits them as its output (the temperature values have been interpreted as integers):

```
(1950, 0)
(1950, 22)
(1950, −11)
(1949, 111)
(1949, 78)
```

The output from the map function is processed by the MapReduce framework before being sent to the reduce function. This processing sorts and groups the key-value pairs by key. So, continuing the example, our reduce function sees the following input:

```
(1949, [111, 78])
(1950, [0, 22, −11])
```

Each year appears with a list of all its air temperature readings. All the reduce function has to do now is iterate through the list and pick up the maximum reading:

```
(1949, 111)
(1950, 22)
```

This is the final output: the maximum global temperature recorded in each year.

The whole data flow is illustrated in Figure 2-1. At the bottom of the diagram is a Unix pipeline, which mimics the whole MapReduce flow and which we will see again later in this chapter when we look at Hadoop Streaming.

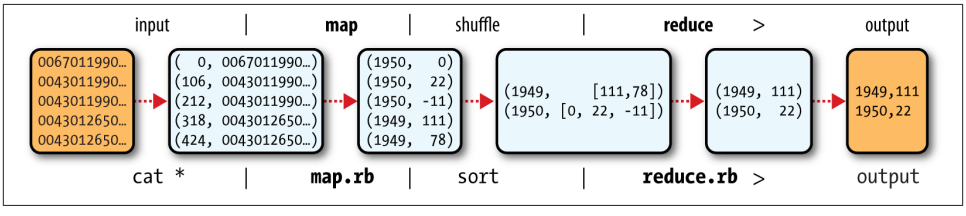

Figure 2-1. MapReduce logical data flow

Java MapReduce

Having run through how the MapReduce program works, the next step is to express it in code. We need three things: a map function, a reduce function, and some code to run the job. The map function is represented by the `Mapper` class, which declares an abstract `map()` method. Example 2-3 shows the implementation of our map function.

Example 2-3. Mapper for the maximum temperature example

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

  private static final int MISSING = 9999;

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    String line = value.toString();
    String year = line.substring(15, 19);
    int airTemperature;
    if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
      airTemperature = Integer.parseInt(line.substring(88, 92));
    } else {
      airTemperature = Integer.parseInt(line.substring(87, 92));
    }
    String quality = line.substring(92, 93);
```

```
    if (airTemperature != MISSING && quality.matches("[01459]")) {
      context.write(new Text(year), new IntWritable(airTemperature));
    }
  }
}
```

The `Mapper` class is a generic type, with four formal type parameters that specify the input key, input value, output key, and output value types of the map function. For the present example, the input key is a long integer offset, the input value is a line of text, the output key is a year, and the output value is an air temperature (an integer). Rather than using built-in Java types, Hadoop provides its own set of basic types that are optimized for network serialization. These are found in the `org.apache.hadoop.io` package. Here we use `LongWritable`, which corresponds to a Java `Long`, `Text` (like Java `String`), and `IntWritable` (like Java `Integer`).

The `map()` method is passed a key and a value. We convert the `Text` value containing the line of input into a Java `String`, then use its `substring()` method to extract the columns we are interested in.

The `map()` method also provides an instance of `Context` to write the output to. In this case, we write the year as a `Text` object (since we are just using it as a key), and the temperature is wrapped in an `IntWritable`. We write an output record only if the temperature is present and the quality code indicates the temperature reading is OK.

The reduce function is similarly defined using a `Reducer`, as illustrated in Example 2-4.

Example 2-4. Reducer for the maximum temperature example

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MaxTemperatureReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

  @Override
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {

    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(key, new IntWritable(maxValue));
  }
}
```

Again, four formal type parameters are used to specify the input and output types, this time for the reduce function. The input types of the reduce function must match the output types of the map function: `Text` and `IntWritable`. And in this case, the output types of the reduce function are `Text` and `IntWritable`, for a year and its maximum temperature, which we find by iterating through the temperatures and comparing each with a record of the highest found so far.

The third piece of code runs the MapReduce job (see Example 2-5).

Example 2-5. Application to find the maximum temperature in the weather dataset

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MaxTemperature {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperature <input path> <output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperature.class);
    job.setJobName("Max temperature");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

A `Job` object forms the specification of the job and gives you control over how the job is run. When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute around the cluster). Rather than explicitly specifying the name of the JAR file, we can pass a class in the `Job`'s `setJarByClass()` method, which Hadoop will use to locate the relevant JAR file by looking for the JAR file containing this class.

Having constructed a `Job` object, we specify the input and output paths. An input path is specified by calling the static `addInputPath()` method on `FileInputFormat`, and it can be a single file, a directory (in which case, the input forms all the files in that directory), or a file pattern. As the name suggests, `addInputPath()` can be called more than once to use input from multiple paths.

The output path (of which there is only one) is specified by the static `setOutputPath()` method on `FileOutputFormat`. It specifies a directory where the output files from the reduce function are written. The directory shouldn't exist before running the job because Hadoop will complain and not run the job. This precaution is to prevent data loss (it can be very annoying to accidentally overwrite the output of a long job with that of another).

Next, we specify the map and reduce types to use via the `setMapperClass()` and `setReducerClass()` methods.

The `setOutputKeyClass()` and `setOutputValueClass()` methods control the output types for the reduce function, and must match what the `Reduce` class produces. The map output types default to the same types, so they do not need to be set if the mapper produces the same types as the reducer (as it does in our case). However, if they are different, the map output types must be set using the `setMapOutputKeyClass()` and `setMapOutputValueClass()` methods.

The input types are controlled via the input format, which we have not explicitly set because we are using the default `TextInputFormat`.

After setting the classes that define the map and reduce functions, we are ready to run the job. The `waitForCompletion()` method on `Job` submits the job and waits for it to finish. The single argument to the method is a flag indicating whether verbose output is generated. When `true`, the job writes information about its progress to the console.

The return value of the `waitForCompletion()` method is a Boolean indicating success (`true`) or failure (`false`), which we translate into the program's exit code of `0` or `1`.

The Java MapReduce API used in this section, and throughout the book, is called the "new API"; it replaces the older, functionally equivalent API. The differences between the two APIs are explained in Appendix D, along with tips on how to convert between the two APIs. You can also find the old API equivalent of the maximum temperature application there.

A test run

After writing a MapReduce job, it's normal to try it out on a small dataset to flush out any immediate problems with the code. First, install Hadoop in standalone mode (there are instructions for how to do this in Appendix A). This is the mode in which Hadoop

runs using the local filesystem with a local job runner. Then, install and compile the examples using the instructions on the book's website.

Let's test it on the five-line sample discussed earlier (the output has been slightly reformatted to fit the page, and some lines have been removed):

```
% export HADOOP_CLASSPATH=hadoop-examples.jar
% hadoop MaxTemperature input/ncdc/sample.txt output
14/09/16 09:48:39 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
14/09/16 09:48:40 WARN mapreduce.JobSubmitter: Hadoop command-line option
parsing not performed. Implement the Tool interface and execute your application
with ToolRunner to remedy this.
14/09/16 09:48:40 INFO input.FileInputFormat: Total input paths to process : 1
14/09/16 09:48:40 INFO mapreduce.JobSubmitter: number of splits:1
14/09/16 09:48:40 INFO mapreduce.JobSubmitter: Submitting tokens for job:
job_local26392882_0001
14/09/16 09:48:40 INFO mapreduce.Job: The url to track the job:
http://localhost:8080/
14/09/16 09:48:40 INFO mapreduce.Job: Running job: job_local26392882_0001
14/09/16 09:48:40 INFO mapred.LocalJobRunner: OutputCommitter set in config null
14/09/16 09:48:40 INFO mapred.LocalJobRunner: OutputCommitter is
org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Waiting for map tasks
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Starting task:
attempt_local26392882_0001_m_000000_0
14/09/16 09:48:40 INFO mapred.Task:  Using ResourceCalculatorProcessTree : null
14/09/16 09:48:40 INFO mapred.LocalJobRunner:
14/09/16 09:48:40 INFO mapred.Task: Task:attempt_local26392882_0001_m_000000_0
is done. And is in the process of committing
14/09/16 09:48:40 INFO mapred.LocalJobRunner: map
14/09/16 09:48:40 INFO mapred.Task: Task 'attempt_local26392882_0001_m_000000_0'
 done.
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Finishing task:
attempt_local26392882_0001_m_000000_0
14/09/16 09:48:40 INFO mapred.LocalJobRunner: map task executor complete.
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Waiting for reduce tasks
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Starting task:
attempt_local26392882_0001_r_000000_0
14/09/16 09:48:40 INFO mapred.Task:  Using ResourceCalculatorProcessTree : null
14/09/16 09:48:40 INFO mapred.LocalJobRunner: 1 / 1 copied.
14/09/16 09:48:40 INFO mapred.Merger: Merging 1 sorted segments
14/09/16 09:48:40 INFO mapred.Merger: Down to the last merge-pass, with 1
segments left of total size: 50 bytes
14/09/16 09:48:40 INFO mapred.Merger: Merging 1 sorted segments
14/09/16 09:48:40 INFO mapred.Merger: Down to the last merge-pass, with 1
segments left of total size: 50 bytes
14/09/16 09:48:40 INFO mapred.LocalJobRunner: 1 / 1 copied.
14/09/16 09:48:40 INFO mapred.Task: Task:attempt_local26392882_0001_r_000000_0
is done. And is in the process of committing
14/09/16 09:48:40 INFO mapred.LocalJobRunner: 1 / 1 copied.
14/09/16 09:48:40 INFO mapred.Task: Task attempt_local26392882_0001_r_000000_0
```

```
is allowed to commit now
14/09/16 09:48:40 INFO output.FileOutputCommitter: Saved output of task
'attempt...local26392882_0001_r_000000_0' to file:/Users/tom/book-workspace/
hadoop-book/output/_temporary/0/task_local26392882_0001_r_000000
14/09/16 09:48:40 INFO mapred.LocalJobRunner: reduce > reduce
14/09/16 09:48:40 INFO mapred.Task: Task 'attempt_local26392882_0001_r_000000_0'
 done.
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Finishing task:
attempt_local26392882_0001_r_000000_0
14/09/16 09:48:40 INFO mapred.LocalJobRunner: reduce task executor complete.
14/09/16 09:48:41 INFO mapreduce.Job: Job job_local26392882_0001 running in uber
 mode : false
14/09/16 09:48:41 INFO mapreduce.Job:  map 100% reduce 100%
14/09/16 09:48:41 INFO mapreduce.Job: Job job_local26392882_0001 completed
successfully
14/09/16 09:48:41 INFO mapreduce.Job: Counters: 30
    File System Counters
        FILE: Number of bytes read=377168
        FILE: Number of bytes written=828464
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
    Map-Reduce Framework
        Map input records=5
        Map output records=5
        Map output bytes=45
        Map output materialized bytes=61
        Input split bytes=129
        Combine input records=0
        Combine output records=0
        Reduce input groups=2
        Reduce shuffle bytes=61
        Reduce input records=5
        Reduce output records=2
        Spilled Records=10
        Shuffled Maps =1
        Failed Shuffles=0
        Merged Map outputs=1
        GC time elapsed (ms)=39
        Total committed heap usage (bytes)=226754560
    File Input Format Counters
        Bytes Read=529
    File Output Format Counters
        Bytes Written=29
```

When the `hadoop` command is invoked with a classname as the first argument, it launches a Java virtual machine (JVM) to run the class. The `hadoop` command adds the Hadoop libraries (and their dependencies) to the classpath and picks up the Hadoop configuration, too. To add the application classes to the classpath, we've defined an environment variable called `HADOOP_CLASSPATH`, which the *hadoop* script picks up.

When running in local (standalone) mode, the programs in this book all assume that you have set the `HADOOP_CLASSPATH` in this way. The commands should be run from the directory that the example code is installed in.

The output from running the job provides some useful information. For example, we can see that the job was given an ID of `job_local26392882_0001`, and it ran one map task and one reduce task (with the following IDs: `attempt_local26392882_0001_m_000000_0` and `attempt_local26392882_0001_r_000000_0`). Knowing the job and task IDs can be very useful when debugging MapReduce jobs.

The last section of the output, titled "Counters," shows the statistics that Hadoop generates for each job it runs. These are very useful for checking whether the amount of data processed is what you expected. For example, we can follow the number of records that went through the system: five map input records produced five map output records (since the mapper emitted one output record for each valid input record), then five reduce input records in two groups (one for each unique key) produced two reduce output records.

The output was written to the *output* directory, which contains one output file per reducer. The job had a single reducer, so we find a single file, named *part-r-00000*:

```
% cat output/part-r-00000
1949 111
1950 22
```

This result is the same as when we went through it by hand earlier. We interpret this as saying that the maximum temperature recorded in 1949 was 11.1°C, and in 1950 it was 2.2°C.

Scaling Out

You've seen how MapReduce works for small inputs; now it's time to take a bird's-eye view of the system and look at the data flow for large inputs. For simplicity, the examples so far have used files on the local filesystem. However, to scale out, we need to store the data in a distributed filesystem (typically HDFS, which you'll learn about in the next chapter). This allows Hadoop to move the MapReduce computation to each machine hosting a part of the data, using Hadoop's resource management system, called YARN (see Chapter 4). Let's see how this works.

Data Flow

First, some terminology. A MapReduce *job* is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration

information. Hadoop runs the job by dividing it into *tasks*, of which there are two types: *map tasks* and *reduce tasks*. The tasks are scheduled using YARN and run on nodes in the cluster. If a task fails, it will be automatically rescheduled to run on a different node.

Hadoop divides the input to a MapReduce job into fixed-size pieces called *input splits*, or just *splits*. Hadoop creates one map task for each split, which runs the user-defined map function for each *record* in the split.

Having many splits means the time taken to process each split is small compared to the time to process the whole input. So if we are processing the splits in parallel, the processing is better load balanced when the splits are small, since a faster machine will be able to process proportionally more splits over the course of the job than a slower machine. Even if the machines are identical, failed processes or other jobs running concurrently make load balancing desirable, and the quality of the load balancing increases as the splits become more fine grained.

On the other hand, if splits are too small, the overhead of managing the splits and map task creation begins to dominate the total job execution time. For most jobs, a good split size tends to be the size of an HDFS block, which is 128 MB by default, although this can be changed for the cluster (for all newly created files) or specified when each file is created.

Hadoop does its best to run the map task on a node where the input data resides in HDFS, because it doesn't use valuable cluster bandwidth. This is called the *data locality optimization*. Sometimes, however, all the nodes hosting the HDFS block replicas for a map task's input split are running other map tasks, so the job scheduler will look for a free map slot on a node in the same rack as one of the blocks. Very occasionally even this is not possible, so an off-rack node is used, which results in an inter-rack network transfer. The three possibilities are illustrated in Figure 2-2.

It should now be clear why the optimal split size is the same as the block size: it is the largest size of input that can be guaranteed to be stored on a single node. If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks, so some of the split would have to be transferred across the network to the node running the map task, which is clearly less efficient than running the whole map task using local data.

Map tasks write their output to the local disk, not to HDFS. Why is this? Map output is intermediate output: it's processed by reduce tasks to produce the final output, and once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to re-create the map output.

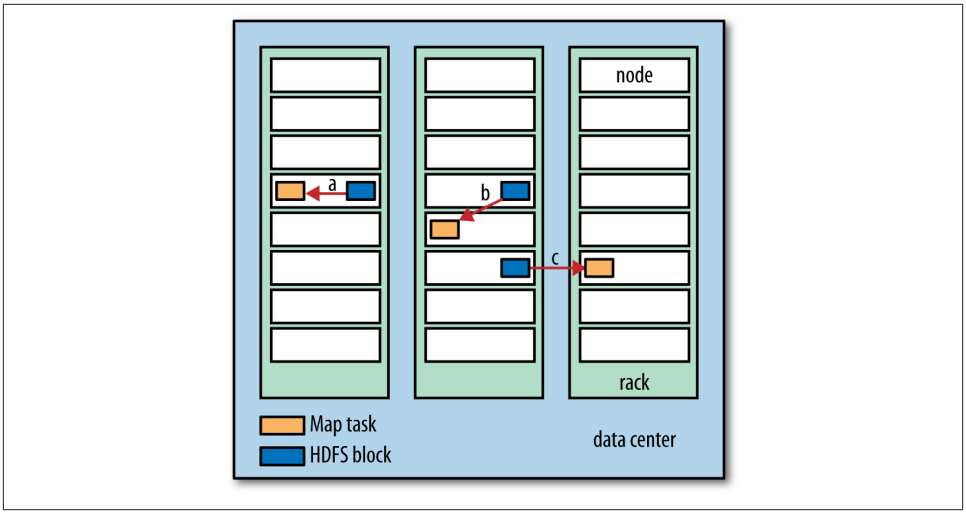

Figure 2-2. Data-local (a), rack-local (b), and off-rack (c) map tasks

Reduce tasks don't have the advantage of data locality; the input to a single reduce task is normally the output from all mappers. In the present example, we have a single reduce task that is fed by all of the map tasks. Therefore, the sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function. The output of the reduce is normally stored in HDFS for reliability. As explained in Chapter 3, for each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes for reliability. Thus, writing the reduce output does consume network bandwidth, but only as much as a normal HDFS write pipeline consumes.

The whole data flow with a single reduce task is illustrated in Figure 2-3. The dotted boxes indicate nodes, the dotted arrows show data transfers on a node, and the solid arrows show data transfers between nodes.

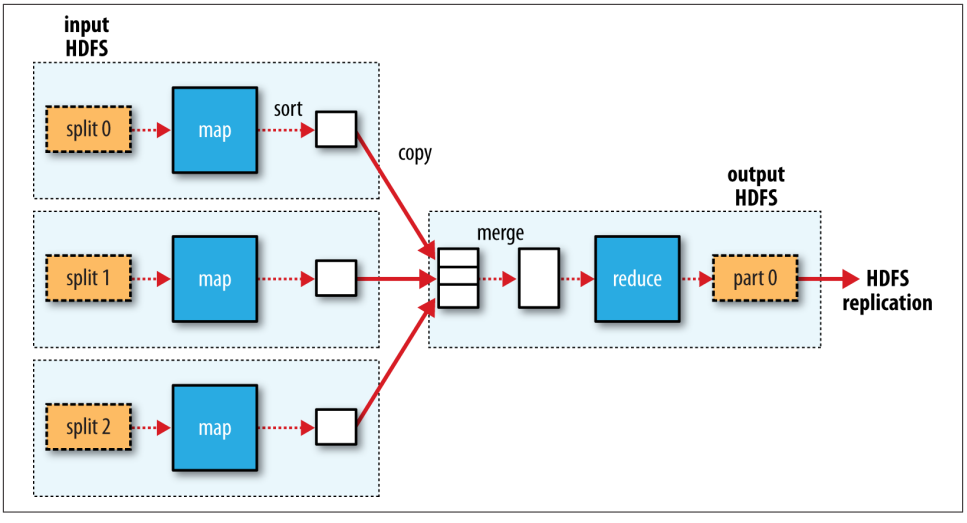

Figure 2-3. MapReduce data flow with a single reduce task

The number of reduce tasks is not governed by the size of the input, but instead is specified independently. In "The Default MapReduce Job" on page 214, you will see how to choose the number of reduce tasks for a given job.

When there are multiple reducers, the map tasks *partition* their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well.

The data flow for the general case of multiple reduce tasks is illustrated in Figure 2-4. This diagram makes it clear why the data flow between map and reduce tasks is colloquially known as "the shuffle," as each reduce task is fed by many map tasks. The shuffle is more complicated than this diagram suggests, and tuning it can have a big impact on job execution time, as you will see in "Shuffle and Sort" on page 197.

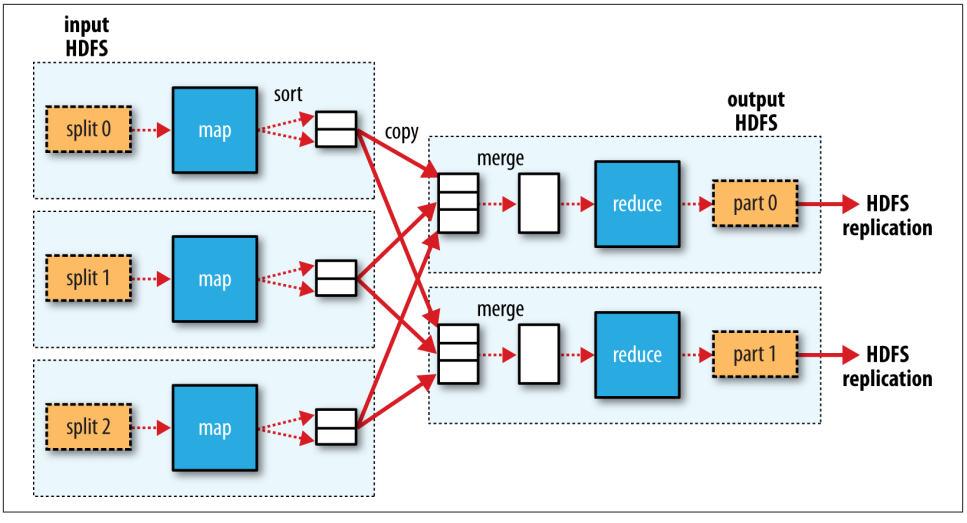

Figure 2-4. MapReduce data flow with multiple reduce tasks

Finally, it's also possible to have zero reduce tasks. This can be appropriate when you don't need the shuffle because the processing can be carried out entirely in parallel (a few examples are discussed in "NLineInputFormat" on page 234). In this case, the only off-node data transfer is when the map tasks write to HDFS (see Figure 2-5).

Combiner Functions

Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Hadoop allows the user to specify a *combiner function* to be run on the map output, and the combiner function's output forms the input to the reduce function. Because the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record, if at all. In other words, calling the combiner function zero, one, or many times should produce the same output from the reducer.

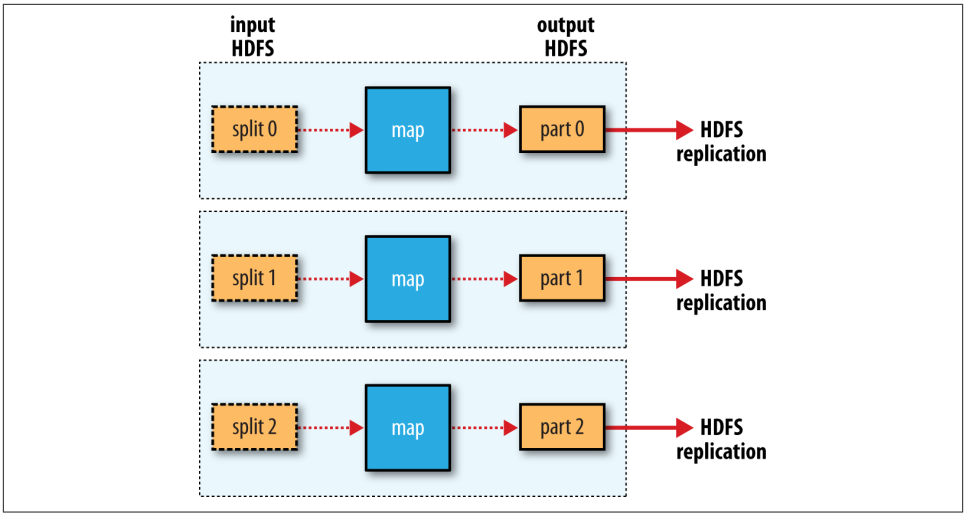

Figure 2-5. MapReduce data flow with no reduce tasks

The contract for the combiner function constrains the type of function that may be used. This is best illustrated with an example. Suppose that for the maximum temperature example, readings for the year 1950 were processed by two maps (because they were in different splits). Imagine the first map produced the output:

```
(1950, 0)
(1950, 20)
(1950, 10)
```

and the second produced:

```
(1950, 25)
(1950, 15)
```

The reduce function would be called with a list of all the values:

```
(1950, [0, 20, 10, 25, 15])
```

with output:

```
(1950, 25)
```

since 25 is the maximum value in the list. We could use a combiner function that, just like the reduce function, finds the maximum temperature for each map output. The reduce function would then be called with:

```
(1950, [20, 25])
```

and would produce the same output as before. More succinctly, we may express the function calls on the temperature values in this case as follows:

```
max(0, 20, 10, 25, 15) = max(max(0, 20, 10), max(25, 15)) = max(20, 25) = 25
```

Not all functions possess this property.[1] For example, if we were calculating mean temperatures, we couldn't use the mean as our combiner function, because:

```
mean(0, 20, 10, 25, 15) = 14
```

but:

```
mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15
```

The combiner function doesn't replace the reduce function. (How could it? The reduce function is still needed to process records with the same key from different maps.) But it can help cut down the amount of data shuffled between the mappers and the reducers, and for this reason alone it is always worth considering whether you can use a combiner function in your MapReduce job.

Specifying a combiner function

Going back to the Java MapReduce program, the combiner function is defined using the `Reducer` class, and for this application, it is the same implementation as the reduce function in `MaxTemperatureReducer`. The only change we need to make is to set the combiner class on the `Job` (see Example 2-6).

Example 2-6. Application to find the maximum temperature, using a combiner function for efficiency

```
public class MaxTemperatureWithCombiner {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperatureWithCombiner <input path> " +
          "<output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperatureWithCombiner.class);
    job.setJobName("Max temperature");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    job.setOutputKeyClass(Text.class);
```

1. Functions with this property are called *commutative* and *associative*. They are also sometimes referred to as *distributive*, such as by Jim Gray et al.'s "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals," February1995.

```
    job.setOutputValueClass(IntWritable.class);

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Running a Distributed MapReduce Job

The same program will run, without alteration, on a full dataset. This is the point of MapReduce: it scales to the size of your data and the size of your hardware. Here's one data point: on a 10-node EC2 cluster running High-CPU Extra Large instances, the program took six minutes to run.[2]

We'll go through the mechanics of running programs on a cluster in Chapter 6.

Hadoop Streaming

Hadoop provides an API to MapReduce that allows you to write your map and reduce functions in languages other than Java. *Hadoop Streaming* uses Unix standard streams as the interface between Hadoop and your program, so you can use any language that can read standard input and write to standard output to write your MapReduce program.[3]

Streaming is naturally suited for text processing. Map input data is passed over standard input to your map function, which processes it line by line and writes lines to standard output. A map output key-value pair is written as a single tab-delimited line. Input to the reduce function is in the same format—a tab-separated key-value pair—passed over standard input. The reduce function reads lines from standard input, which the framework guarantees are sorted by key, and writes its results to standard output.

Let's illustrate this by rewriting our MapReduce program for finding maximum temperatures by year in Streaming.

Ruby

The map function can be expressed in Ruby as shown in Example 2-7.

2. This is a factor of seven faster than the serial run on one machine using *awk*. The main reason it wasn't proportionately faster is because the input data wasn't evenly partitioned. For convenience, the input files were gzipped by year, resulting in large files for later years in the dataset, when the number of weather records was much higher.
3. Hadoop Pipes is an alternative to Streaming for C++ programmers. It uses sockets to communicate with the process running the C++ map or reduce function.

Example 2-7. Map function for maximum temperature in Ruby

```
#!/usr/bin/env ruby

STDIN.each_line do |line|
  val = line
  year, temp, q = val[15,4], val[87,5], val[92,1]
  puts "#{year}\t#{temp}" if (temp != "+9999" && q =~ /[01459]/)
end
```

The program iterates over lines from standard input by executing a block for each line from `STDIN` (a global constant of type `IO`). The block pulls out the relevant fields from each input line and, if the temperature is valid, writes the year and the temperature separated by a tab character, `\t`, to standard output (using `puts`).

It's worth drawing out a design difference between Streaming and the Java MapReduce API. The Java API is geared toward processing your map function one record at a time. The framework calls the `map()` method on your `Mapper` for each record in the input, whereas with Streaming the map program can decide how to process the input—for example, it could easily read and process multiple lines at a time since it's in control of the reading. The user's Java map implementation is "pushed" records, but it's still possible to consider multiple lines at a time by accumulating previous lines in an instance variable in the `Mapper`.[4] In this case, you need to implement the `cleanup()` method so that you know when the last record has been read, so you can finish processing the last group of lines.

Because the script just operates on standard input and output, it's trivial to test the script without using Hadoop, simply by using Unix pipes:

```
% cat input/ncdc/sample.txt | ch02-mr-intro/src/main/ruby/max_temperature_map.rb
1950    +0000
1950    +0022
1950    -0011
1949    +0111
1949    +0078
```

The reduce function shown in Example 2-8 is a little more complex.

Example 2-8. Reduce function for maximum temperature in Ruby

```
#!/usr/bin/env ruby

last_key, max_val = nil, -1000000
STDIN.each_line do |line|
  key, val = line.split("\t")
```

4. Alternatively, you could use "pull"-style processing in the new MapReduce API; see Appendix D.

```
  if last_key && last_key != key
    puts "#{last_key}\t#{max_val}"
    last_key, max_val = key, val.to_i
  else
    last_key, max_val = key, [max_val, val.to_i].max
  end
end
puts "#{last_key}\t#{max_val}" if last_key
```

Again, the program iterates over lines from standard input, but this time we have to store some state as we process each key group. In this case, the keys are the years, and we store the last key seen and the maximum temperature seen so far for that key. The MapReduce framework ensures that the keys are ordered, so we know that if a key is different from the previous one, we have moved into a new key group. In contrast to the Java API, where you are provided an iterator over each key group, in Streaming you have to find key group boundaries in your program.

For each line, we pull out the key and value. Then, if we've just finished a group (`last_key && last_key != key`), we write the key and the maximum temperature for that group, separated by a tab character, before resetting the maximum temperature for the new key. If we haven't just finished a group, we just update the maximum temperature for the current key.

The last line of the program ensures that a line is written for the last key group in the input.

We can now simulate the whole MapReduce pipeline with a Unix pipeline (which is equivalent to the Unix pipeline shown in Figure 2-1):

```
% cat input/ncdc/sample.txt | \
  ch02-mr-intro/src/main/ruby/max_temperature_map.rb | \
  sort | ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb
1949 111
1950 22
```

The output is the same as that of the Java program, so the next step is to run it using Hadoop itself.

The `hadoop` command doesn't support a Streaming option; instead, you specify the Streaming JAR file along with the `jar` option. Options to the Streaming program specify the input and output paths and the map and reduce scripts. This is what it looks like:

```
% hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
  -input input/ncdc/sample.txt \
  -output output \
  -mapper ch02-mr-intro/src/main/ruby/max_temperature_map.rb \
  -reducer ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb
```

When running on a large dataset on a cluster, we should use the `-combiner` option to set the combiner:

```
% hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
  -files ch02-mr-intro/src/main/ruby/max_temperature_map.rb,\
ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb \
  -input input/ncdc/all \
  -output output \
  -mapper ch02-mr-intro/src/main/ruby/max_temperature_map.rb \
  -combiner ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb \
  -reducer ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb
```

Note also the use of `-files`, which we use when running Streaming programs on the cluster to ship the scripts to the cluster.

Python

Streaming supports any programming language that can read from standard input and write to standard output, so for readers more familiar with Python, here's the same example again.[5] The map script is in Example 2-9, and the reduce script is in Example 2-10.

Example 2-9. Map function for maximum temperature in Python

```
#!/usr/bin/env python

import re
import sys

for line in sys.stdin:
  val = line.strip()
  (year, temp, q) = (val[15:19], val[87:92], val[92:93])
  if (temp != "+9999" and re.match("[01459]", q)):
    print "%s\t%s" % (year, temp)
```

Example 2-10. Reduce function for maximum temperature in Python

```
#!/usr/bin/env python

import sys

(last_key, max_val) = (None, -sys.maxint)
for line in sys.stdin:
  (key, val) = line.strip().split("\t")
  if last_key and last_key != key:
    print "%s\t%s" % (last_key, max_val)
    (last_key, max_val) = (key, int(val))
  else:
    (last_key, max_val) = (key, max(max_val, int(val)))
```

5. As an alternative to Streaming, Python programmers should consider Dumbo, which makes the Streaming MapReduce interface more Pythonic and easier to use.

```
if last_key:
  print "%s\t%s" % (last_key, max_val)
```

We can test the programs and run the job in the same way we did in Ruby. For example, to run a test:

```
% cat input/ncdc/sample.txt | \
  ch02-mr-intro/src/main/python/max_temperature_map.py | \
  sort | ch02-mr-intro/src/main/python/max_temperature_reduce.py
1949    111
1950    22
```

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.

There are two properties that we set in the pseudodistributed configuration that deserve further explanation. The first is `fs.defaultFS`, set to `hdfs://localhost/`, which is used to set a default filesystem for Hadoop.[5] Filesystems are specified by a URI, and here we have used an `hdfs` URI to configure Hadoop to use HDFS by default. The HDFS daemons will use this property to determine the host and port for the HDFS namenode. We'll be running it on localhost, on the default HDFS port, 8020. And HDFS clients will use this property to work out where the namenode is running so they can connect to it.

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

Basic Filesystem Operations

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

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

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

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

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

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

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

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

```
% hadoop fs -copyToLocal quangle.txt quangle.copy.txt
% md5 input/docs/quangle.txt quangle.copy.txt
MD5 (input/docs/quangle.txt) = e7891a2627cf263a079fb0f18256ffb2
MD5 (quangle.copy.txt) = e7891a2627cf263a079fb0f18256ffb2
```

5. In Hadoop 1, the name for this property was `fs.default.name`. Hadoop 2 introduced many new property names, and deprecated the old ones (see "Which Properties Can I Set?" on page 150). This book uses the new property names.

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

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

```
% hadoop fs -mkdir books
% hadoop fs -ls .
Found 2 items
drwxr-xr-x   - tom supergroup          0 2014-10-04 13:22 books
-rw-r--r--   1 tom supergroup        119 2014-10-04 13:21 quangle.txt
```

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

File Permissions in HDFS

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

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

By default, Hadoop runs with security disabled, which means that a client's identity is not authenticated. Because clients are remote, it is possible for a client to become an arbitrary user simply by creating an account of that name on the remote system. This is not possible if security is turned on; see "Security" on page 309. Either way, it is worthwhile having permissions enabled (as they are by default; see the `dfs.permissions.enabled` property) to avoid accidental modification or deletion of substantial parts of the filesystem, either by users or by automated tools or programs.

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

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

Hadoop Filesystems

Hadoop has an abstract notion of filesystems, of which HDFS is just one implementation. The Java abstract class `org.apache.hadoop.fs.FileSystem` represents the client interface to a filesystem in Hadoop, and there are several concrete implementations. The main ones that ship with Hadoop are described in Table 3-1.

Table 3-1. Hadoop filesystems

Filesystem	URI scheme	Java implementation (all under org.apache.hadoop)	Description
Local	`file`	`fs.LocalFileSystem`	A filesystem for a locally connected disk with client-side checksums. Use `RawLocalFileSystem` for a local filesystem with no checksums. See "LocalFileSystem" on page 99.
HDFS	`hdfs`	`hdfs.DistributedFileSystem`	Hadoop's distributed filesystem. HDFS is designed to work efficiently in conjunction with MapReduce.
WebHDFS	`webhdfs`	`hdfs.web.WebHdfsFileSystem`	A filesystem providing authenticated read/write access to HDFS over HTTP. See "HTTP" on page 54.
Secure WebHDFS	`swebhdfs`	`hdfs.web.SWebHdfsFileSystem`	The HTTPS version of WebHDFS.
HAR	`har`	`fs.HarFileSystem`	A filesystem layered on another filesystem for archiving files. Hadoop Archives are used for packing lots of files in HDFS into a single archive file to reduce the namenode's memory usage. Use the `hadoop archive` command to create HAR files.
View	`viewfs`	`viewfs.ViewFileSystem`	A client-side mount table for other Hadoop filesystems. Commonly used to create mount points for federated namenodes (see "HDFS Federation" on page 48).
FTP	`ftp`	`fs.ftp.FTPFileSystem`	A filesystem backed by an FTP server.
S3	`s3a`	`fs.s3a.S3AFileSystem`	A filesystem backed by Amazon S3. Replaces the older `s3n` (S3 native) implementation.

Filesystem	URI scheme	Java implementation (all under org.apache.hadoop)	Description
Azure	wasb	fs.azure.NativeAzureFileSystem	A filesystem backed by Microsoft Azure.
Swift	swift	fs.swift.snative.SwiftNativeFileSystem	A filesystem backed by OpenStack Swift.

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

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

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

Interfaces

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

HTTP

By exposing its filesystem interface as a Java API, Hadoop makes it awkward for non-Java applications to access HDFS. The HTTP REST API exposed by the WebHDFS protocol makes it easier for other languages to interact with HDFS. Note that the HTTP interface is slower than the native Java client, so should be avoided for very large data transfers if possible.

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

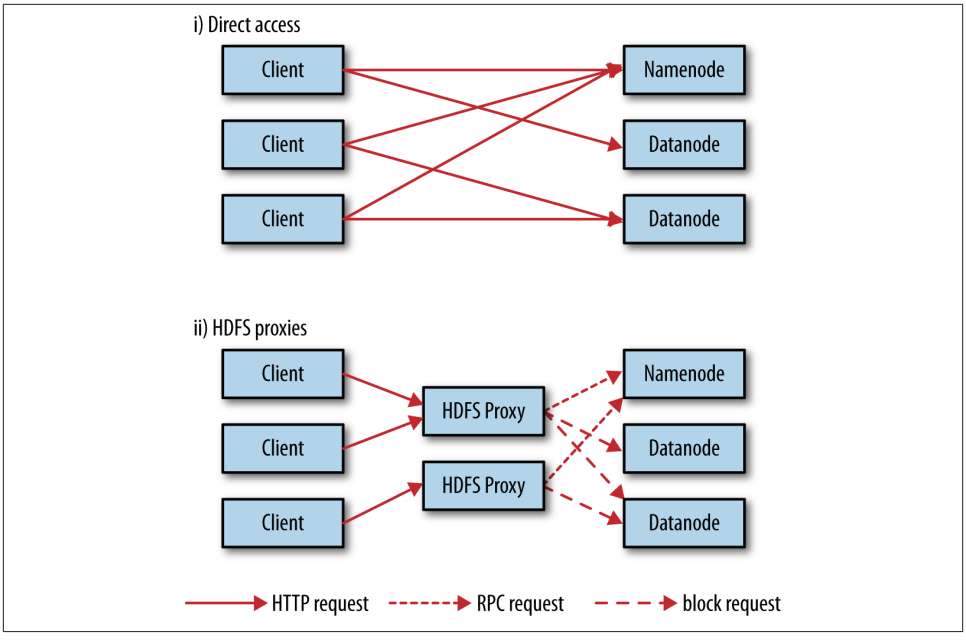

Figure 3-1. Accessing HDFS over HTTP directly and via a bank of HDFS proxies

In the first case, the embedded web servers in the namenode and datanodes act as WebHDFS endpoints. (WebHDFS is enabled by default, since `dfs.webhdfs.enabled` is set to `true`.) File metadata operations are handled by the namenode, while file read (and write) operations are sent first to the namenode, which sends an HTTP redirect to the client indicating the datanode to stream file data from (or to).

The second way of accessing HDFS over HTTP relies on one or more standalone proxy servers. (The proxies are stateless, so they can run behind a standard load balancer.) All traffic to the cluster passes through the proxy, so the client never accesses the namenode or datanode directly. This allows for stricter firewall and bandwidth-limiting policies to be put in place. It's common to use a proxy for transfers between Hadoop clusters located in different data centers, or when accessing a Hadoop cluster running in the cloud from an external network.

The HttpFS proxy exposes the same HTTP (and HTTPS) interface as WebHDFS, so clients can access both using `webhdfs` (or `swebhdfs`) URIs. The HttpFS proxy is started independently of the namenode and datanode daemons, using the *httpfs.sh* script, and by default listens on a different port number (14000).

C

Hadoop provides a C library called *libhdfs* that mirrors the Java `FileSystem` interface (it was written as a C library for accessing HDFS, but despite its name it can be used to

access any Hadoop filesystem). It works using the *Java Native Interface* (JNI) to call a Java filesystem client. There is also a *libwebhdfs* library that uses the WebHDFS interface described in the previous section.

The C API is very similar to the Java one, but it typically lags the Java one, so some newer features may not be supported. You can find the header file, *hdfs.h*, in the *include* directory of the Apache Hadoop binary tarball distribution.

The Apache Hadoop binary tarball comes with prebuilt *libhdfs* binaries for 64-bit Linux, but for other platforms you will need to build them yourself by following the *BUILD ING.txt* instructions at the top level of the source tree.

NFS

It is possible to mount HDFS on a local client's filesystem using Hadoop's NFSv3 gateway. You can then use Unix utilities (such as `ls` and `cat`) to interact with the filesystem, upload files, and in general use POSIX libraries to access the filesystem from any programming language. Appending to a file works, but random modifications of a file do not, since HDFS can only write to the end of a file.

Consult the Hadoop documentation for how to configure and run the NFS gateway and connect to it from a client.

FUSE

Filesystem in Userspace (FUSE) allows filesystems that are implemented in user space to be integrated as Unix filesystems. Hadoop's Fuse-DFS contrib module allows HDFS (or any Hadoop filesystem) to be mounted as a standard local filesystem. Fuse-DFS is implemented in C using *libhdfs* as the interface to HDFS. At the time of writing, the Hadoop NFS gateway is the more robust solution to mounting HDFS, so should be preferred over Fuse-DFS.

The Java Interface

In this section, we dig into the Hadoop `FileSystem` class: the API for interacting with one of Hadoop's filesystems.[6] Although we focus mainly on the HDFS implementation, `DistributedFileSystem`, in general you should strive to write your code against the `FileSystem` abstract class, to retain portability across filesystems. This is very useful when testing your program, for example, because you can rapidly run tests using data stored on the local filesystem.

6. In Hadoop 2 and later, there is a new filesystem interface called `FileContext` with better handling of multiple filesystems (so a single `FileContext` can resolve multiple filesystem schemes, for example) and a cleaner, more consistent interface. `FileSystem` is still more widely used, however.

Reading Data from a Hadoop URL

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

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

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

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

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

```
public class URLCat {

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

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

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

Here's a sample run:[7]

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

Reading Data Using the FileSystem API

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

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

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

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

A `Configuration` object encapsulates a client or server's configuration, which is set using configuration files read from the classpath, such as *etc/hadoop/core-site.xml*. The first method returns the default filesystem (as specified in *core-site.xml*, or the default local filesystem if not specified there). The second uses the given `URI`'s scheme and authority to determine the filesystem to use, falling back to the default filesystem if no scheme is specified in the given `URI`. The third retrieves the filesystem as the given user, which is important in the context of security (see "Security" on page 309).

In some cases, you may want to retrieve a local filesystem instance. For this, you can use the convenience method `getLocal()`:

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

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

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

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

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

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

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

```
public class FileSystemCat {

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

The program runs as follows:

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

FSDataInputStream

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

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

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

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

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

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

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

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

```
public class FileSystemDoubleCat {

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

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

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

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

```
public interface PositionedReadable {

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

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

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

The `read()` method reads up to `length` bytes from the given `position` in the file into the `buffer` at the given `offset` in the buffer. The return value is the number of bytes actually read; callers should check this value, as it may be less than `length`. The `readFully()` methods will read `length` bytes into the buffer (or `buffer.length` bytes for the version that just takes a byte array `buffer`), unless the end of the file is reached, in which case an `EOFException` is thrown.

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

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

Writing Data

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

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

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

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

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

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

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

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

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

The append operation allows a single writer to modify an already written file by opening it and writing data from the final offset in the file. With this API, applications that produce unbounded files, such as logfiles, can write to an existing file after having closed it. The append operation is optional and not implemented by all Hadoop filesystems. For example, HDFS supports append, but S3 filesystems don't.

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

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

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

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

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

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

Typical usage:

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

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

FSDataOutputStream

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

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

public class FSDataOutputStream extends DataOutputStream implements Syncable {

  public long getPos() throws IOException {
    // implementation elided
  }

  // implementation elided

}
```

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

Directories

`FileSystem` provides a method to create a directory:

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

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

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

Querying the Filesystem

File metadata: FileStatus

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

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

Example 3-5. Demonstrating file status information

```
public class ShowFileStatusTest {

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

  @Before
  public void setUp() throws IOException {
    Configuration conf = new Configuration();
    if (System.getProperty("test.build.data") == null) {
      System.setProperty("test.build.data", "/tmp");
    }
    cluster = new MiniDFSCluster.Builder(conf).build();
    fs = cluster.getFileSystem();
    OutputStream out = fs.create(new Path("/dir/file"));
    out.write("content".getBytes("UTF-8"));
    out.close();
  }

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

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

  @Test
  public void fileStatusForFile() throws IOException {
    Path file = new Path("/dir/file");
    FileStatus stat = fs.getFileStatus(file);
    assertThat(stat.getPath().toUri().getPath(), is("/dir/file"));
    assertThat(stat.isDirectory(), is(false));
    assertThat(stat.getLen(), is(7L));
    assertThat(stat.getModificationTime(),
        is(lessThanOrEqualTo(System.currentTimeMillis())));
    assertThat(stat.getReplication(), is((short) 1));
    assertThat(stat.getBlockSize(), is(128 * 1024 * 1024L));
    assertThat(stat.getOwner(), is(System.getProperty("user.name")));
    assertThat(stat.getGroup(), is("supergroup"));
    assertThat(stat.getPermission().toString(), is("rw-r--r--"));
  }

  @Test
  public void fileStatusForDirectory() throws IOException {
    Path dir = new Path("/dir");
    FileStatus stat = fs.getFileStatus(dir);
    assertThat(stat.getPath().toUri().getPath(), is("/dir"));
    assertThat(stat.isDirectory(), is(true));
```

```
    assertThat(stat.getLen(), is(0L));
    assertThat(stat.getModificationTime(),
        is(lessThanOrEqualTo(System.currentTimeMillis())));
    assertThat(stat.getReplication(), is((short) 0));
    assertThat(stat.getBlockSize(), is(0L));
    assertThat(stat.getOwner(), is(System.getProperty("user.name")));
    assertThat(stat.getGroup(), is("supergroup"));
    assertThat(stat.getPermission().toString(), is("rwxr-xr-x"));
  }

}
```

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

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

Listing files

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

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

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

Overloaded variants allow a `PathFilter` to be supplied to restrict the files and directories to match. You will see an example of this in the section "PathFilter" on page 67. Finally, if you specify an array of paths, the result is a shortcut for calling the equivalent single-path `listStatus()` method for each path in turn and accumulating the `FileStatus` object arrays in a single array. This can be useful for building up lists of input files to process from distinct parts of the filesystem tree. Example 3-6 is a simple demonstration of this idea. Note the use of `stat2Paths()` in Hadoop's `FileUtil` for turning an array of `FileStatus` objects into an array of `Path` objects.

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

```
public class ListStatus {

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

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

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

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

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

File patterns

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

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

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

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

Table 3-2. Glob characters and their meanings

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

Glob	Name	Matches
`[^a-b]`	*negated character range*	Matches a single character that is not in the (closed) range `[a, b]`, where `a` is lexicographically less than or equal to `b`
`{a,b}`	*alternation*	Matches either expression `a` or `b`
`\c`	*escaped character*	Matches character `c` when it is a metacharacter

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

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

Here are some file globs and their expansions:

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

PathFilter

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

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

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

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

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

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

```
public class RegexExcludePathFilter implements PathFilter {

  private final String regex;

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

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

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

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

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

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

Deleting Data

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

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

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

Data Flow

Anatomy of a File Read

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

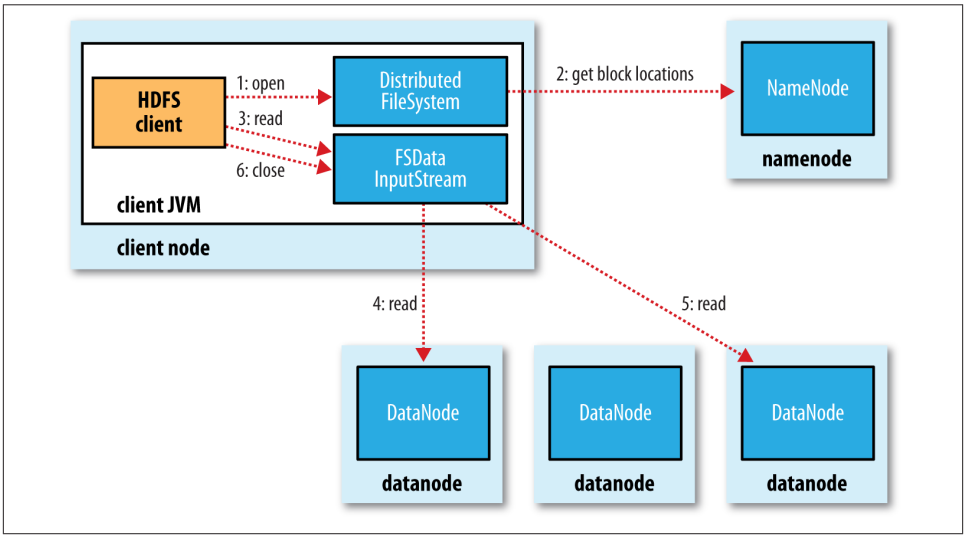

Figure 3-2. A client reading data from HDFS

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.)

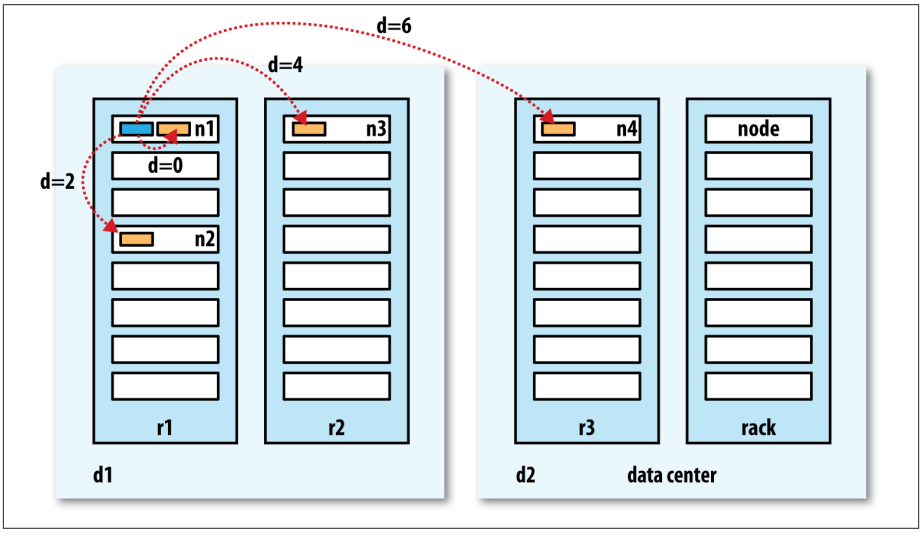

Figure 3-3. Network distance in Hadoop

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.

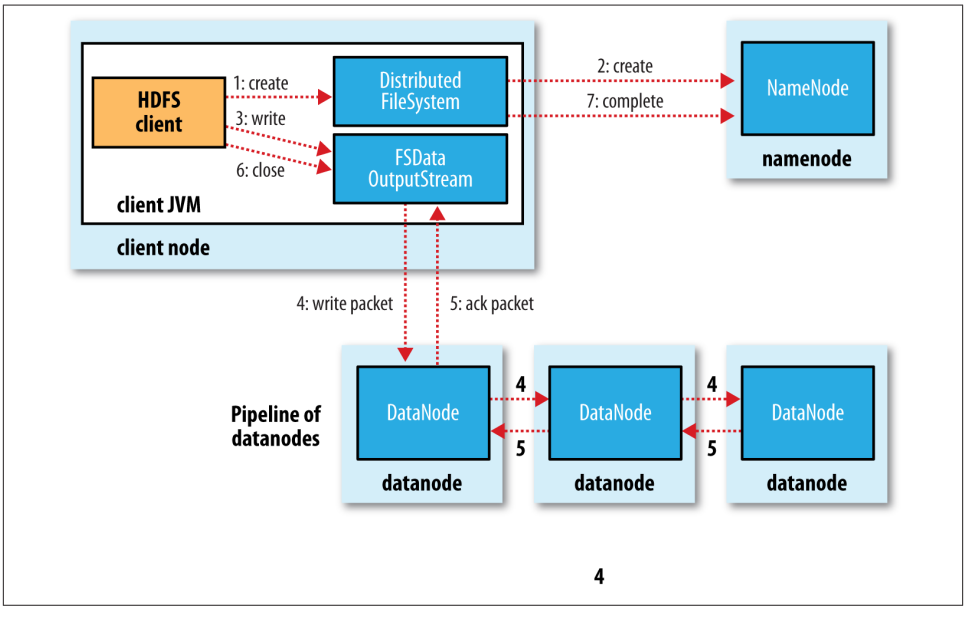

Figure 3-4. A client writing data to HDFS

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 trade-off between reliability and write bandwidth and read bandwidth here. For example, placing all replicas on a single node incurs the lowest write bandwidth penalty (since the replication pipeline runs on a single node), but this offers no real redundancy (if the node fails, the data for that block is lost). Also, the read bandwidth is high for off-rack reads. At the other extreme, placing replicas in different data centers may maximize redundancy, but at the cost of bandwidth. Even in the same data center (which is what all Hadoop clusters to date have run in), there are a variety of 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.

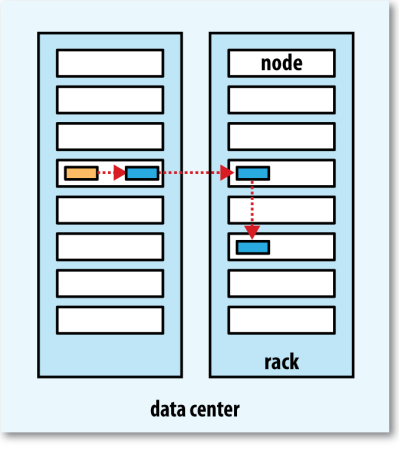

Figure 3-5. A typical replica pipeline

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.

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

distcp is implemented as a MapReduce job where the work of copying is done by the maps that run in parallel across the cluster. There are no reducers. Each file is copied by a single map, and *distcp* tries to give each map approximately the same amount of data by bucketing files into roughly equal allocations. By default, up to 20 maps are used, but this can be changed by specifying the `-m` argument to *distcp*.

A very common use case for *distcp* is for transferring data between two HDFS clusters. For example, the following creates a backup of the first cluster's */foo* directory on the second:

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

The `-delete` flag causes *distcp* to delete any files or directories from the destination that are not present in the source, and `-p` means that file status attributes like permissions, block size, and replication are preserved. You can run *distcp* with no arguments to see precise usage instructions.

If the two clusters are running incompatible versions of HDFS, then you can use the `webhdfs` protocol to *distcp* between them:

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

Another variant is to use an HttpFs proxy as the *distcp* source or destination (again using the `webhdfs` protocol), which has the advantage of being able to set firewall and bandwidth controls (see "HTTP" on page 54).

Keeping an HDFS Cluster Balanced

When copying data into HDFS, it's important to consider cluster balance. HDFS works best when the file blocks are evenly spread across the cluster, so you want to ensure that *distcp* doesn't disrupt this. For example, if you specified `-m 1`, a single map would do the copy, which—apart from being slow and not using the cluster resources efficiently—would mean that the first replica of each block would reside on the node running the map (until the disk filled up). The second and third replicas would be spread across the cluster, but this one node would be unbalanced. By having more maps than nodes in the cluster, this problem is avoided. For this reason, it's best to start by running *distcp* with the default of 20 maps per node.

However, it's not always possible to prevent a cluster from becoming unbalanced. Perhaps you want to limit the number of maps so that some of the nodes can be used by other jobs. In this case, you can use the *balancer* tool (see "Balancer" on page 329) to subsequently even out the block distribution across the cluster.

CHAPTER 4
YARN

Apache YARN (Yet Another Resource Negotiator) is Hadoop's cluster resource management system. YARN was introduced in Hadoop 2 to improve the MapReduce implementation, but it is general enough to support other distributed computing paradigms as well.

YARN provides APIs for requesting and working with cluster resources, but these APIs are not typically used directly by user code. Instead, users write to higher-level APIs provided by distributed computing frameworks, which themselves are built on YARN and hide the resource management details from the user. The situation is illustrated in Figure 4-1, which shows some distributed computing frameworks (MapReduce, Spark, and so on) running as *YARN applications* on the cluster compute layer (YARN) and the cluster storage layer (HDFS and HBase).

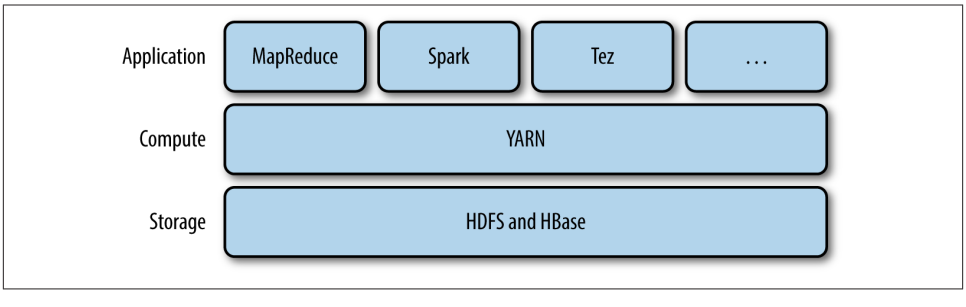

Figure 4-1. YARN applications

There is also a layer of applications that build on the frameworks shown in Figure 4-1. Pig, Hive, and Crunch are all examples of processing frameworks that run on MapReduce, Spark, or Tez (or on all three), and don't interact with YARN directly.

This chapter walks through the features in YARN and provides a basis for understanding later chapters in Part IV that cover Hadoop's distributed processing frameworks.

Anatomy of a YARN Application Run

YARN provides its core services via two types of long-running daemon: a *resource manager* (one per cluster) to manage the use of resources across the cluster, and *node managers* running on all the nodes in the cluster to launch and monitor *containers*. A container executes an application-specific process with a constrained set of resources (memory, CPU, and so on). Depending on how YARN is configured (see "YARN" on page 300), a container may be a Unix process or a Linux cgroup. Figure 4-2 illustrates how YARN runs an application.

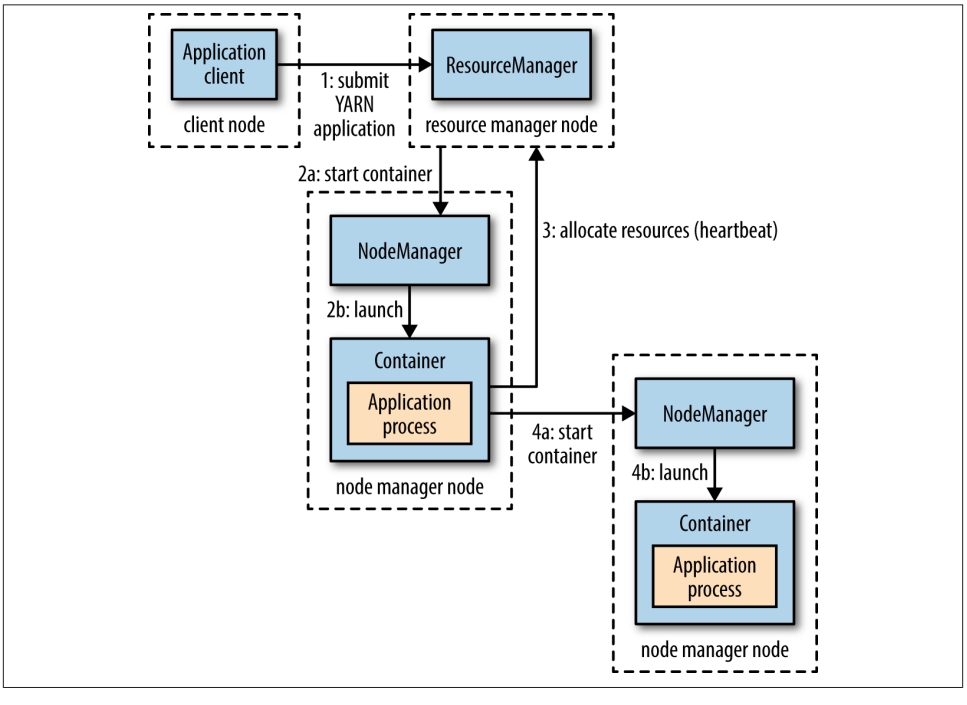

Figure 4-2. How YARN runs an application

To run an application on YARN, a client contacts the resource manager and asks it to run an *application master* process (step 1 in Figure 4-2). The resource manager then finds a node manager that can launch the application master in a container (steps 2a

and 2b).[1] Precisely what the application master does once it is running depends on the application. It could simply run a computation in the container it is running in and return the result to the client. Or it could request more containers from the resource managers (step 3), and use them to run a distributed computation (steps 4a and 4b). The latter is what the MapReduce YARN application does, which we'll look at in more detail in "Anatomy of a MapReduce Job Run" on page 185.

Notice from Figure 4-2 that YARN itself does not provide any way for the parts of the application (client, master, process) to communicate with one another. Most nontrivial YARN applications use some form of remote communication (such as Hadoop's RPC layer) to pass status updates and results back to the client, but these are specific to the application.

Resource Requests

YARN has a flexible model for making resource requests. A request for a set of containers can express the amount of computer resources required for each container (memory and CPU), as well as locality constraints for the containers in that request.

Locality is critical in ensuring that distributed data processing algorithms use the cluster bandwidth efficiently,[2] so YARN allows an application to specify locality constraints for the containers it is requesting. Locality constraints can be used to request a container on a specific node or rack, or anywhere on the cluster (off-rack).

Sometimes the locality constraint cannot be met, in which case either no allocation is made or, optionally, the constraint can be loosened. For example, if a specific node was requested but it is not possible to start a container on it (because other containers are running on it), then YARN will try to start a container on a node in the same rack, or, if that's not possible, on any node in the cluster.

In the common case of launching a container to process an HDFS block (to run a map task in MapReduce, say), the application will request a container on one of the nodes hosting the block's three replicas, or on a node in one of the racks hosting the replicas, or, failing that, on any node in the cluster.

A YARN application can make resource requests at any time while it is running. For example, an application can make all of its requests up front, or it can take a more dynamic approach whereby it requests more resources dynamically to meet the changing needs of the application.

1. It's also possible for the client to start the application master, possibly outside the cluster, or in the same JVM as the client. This is called an *unmanaged application master*.
2. For more on this topic see "Scaling Out" on page 30 and "Network Topology and Hadoop" on page 70.

Spark takes the first approach, starting a fixed number of executors on the cluster (see "Spark on YARN" on page 571). MapReduce, on the other hand, has two phases: the map task containers are requested up front, but the reduce task containers are not started until later. Also, if any tasks fail, additional containers will be requested so the failed tasks can be rerun.

Application Lifespan

The lifespan of a YARN application can vary dramatically: from a short-lived application of a few seconds to a long-running application that runs for days or even months. Rather than look at how long the application runs for, it's useful to categorize applications in terms of how they map to the jobs that users run. The simplest case is one application per user job, which is the approach that MapReduce takes.

The second model is to run one application per workflow or user session of (possibly unrelated) jobs. This approach can be more efficient than the first, since containers can be reused between jobs, and there is also the potential to cache intermediate data between jobs. Spark is an example that uses this model.

The third model is a long-running application that is shared by different users. Such an application often acts in some kind of coordination role. For example, Apache Slider has a long-running application master for launching other applications on the cluster. This approach is also used by Impala (see "SQL-on-Hadoop Alternatives" on page 484) to provide a proxy application that the Impala daemons communicate with to request cluster resources. The "always on" application master means that users have very low-latency responses to their queries since the overhead of starting a new application master is avoided.[3]

Building YARN Applications

Writing a YARN application from scratch is fairly involved, but in many cases is not necessary, as it is often possible to use an existing application that fits the bill. For example, if you are interested in running a directed acyclic graph (DAG) of jobs, then Spark or Tez is appropriate; or for stream processing, Spark, Samza, or Storm works.[4]

There are a couple of projects that simplify the process of building a YARN application. Apache Slider, mentioned earlier, makes it possible to run existing distributed applications on YARN. Users can run their own instances of an application (such as HBase) on a cluster, independently of other users, which means that different users can run different versions of the same application. Slider provides controls to change the number

3. The low-latency application master code lives in the Llama project.

4. All of these projects are Apache Software Foundation projects.

of nodes an application is running on, and to suspend then resume a running application.

Apache Twill is similar to Slider, but in addition provides a simple programming model for developing distributed applications on YARN. Twill allows you to define cluster processes as an extension of a Java `Runnable`, then runs them in YARN containers on the cluster. Twill also provides support for, among other things, real-time logging (log events from runnables are streamed back to the client) and command messages (sent from the client to runnables).

In cases where none of these options are sufficient—such as an application that has complex scheduling requirements—then the *distributed shell* application that is a part of the YARN project itself serves as an example of how to write a YARN application. It demonstrates how to use YARN's client APIs to handle communication between the client or application master and the YARN daemons.

YARN Compared to MapReduce 1

The distributed implementation of MapReduce in the original version of Hadoop (version 1 and earlier) is sometimes referred to as "MapReduce 1" to distinguish it from MapReduce 2, the implementation that uses YARN (in Hadoop 2 and later).

It's important to realize that the old and new MapReduce APIs are not the same thing as the MapReduce 1 and MapReduce 2 implementations. The APIs are user-facing client-side features and determine how you write MapReduce programs (see Appendix D), whereas the implementations are just different ways of running MapReduce programs. All four combinations are supported: both the old and new MapReduce APIs run on both MapReduce 1 and 2.

In MapReduce 1, there are two types of daemon that control the job execution process: a *jobtracker* and one or more *tasktrackers*. The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a task fails, the jobtracker can reschedule it on a different tasktracker.

In MapReduce 1, the jobtracker takes care of both job scheduling (matching tasks with tasktrackers) and task progress monitoring (keeping track of tasks, restarting failed or slow tasks, and doing task bookkeeping, such as maintaining counter totals). By contrast, in YARN these responsibilities are handled by separate entities: the resource manager and an application master (one for each MapReduce job). The jobtracker is also responsible for storing job history for completed jobs, although it is possible to run a

job history server as a separate daemon to take the load off the jobtracker. In YARN, the equivalent role is the timeline server, which stores application history.[5]

The YARN equivalent of a tasktracker is a node manager. The mapping is summarized in Table 4-1.

Table 4-1. A comparison of MapReduce 1 and YARN components

MapReduce 1	YARN
Jobtracker	Resource manager, application master, timeline server
Tasktracker	Node manager
Slot	Container

YARN was designed to address many of the limitations in MapReduce 1. The benefits to using YARN include the following:

Scalability

YARN can run on larger clusters than MapReduce 1. MapReduce 1 hits scalability bottlenecks in the region of 4,000 nodes and 40,000 tasks,[6] stemming from the fact that the jobtracker has to manage both jobs *and* tasks. YARN overcomes these limitations by virtue of its split resource manager/application master architecture: it is designed to scale up to 10,000 nodes and 100,000 tasks.

In contrast to the jobtracker, each instance of an application—here, a MapReduce job—has a dedicated application master, which runs for the duration of the application. This model is actually closer to the original Google MapReduce paper, which describes how a master process is started to coordinate map and reduce tasks running on a set of workers.

Availability

High availability (HA) is usually achieved by replicating the state needed for another daemon to take over the work needed to provide the service, in the event of the service daemon failing. However, the large amount of rapidly changing complex state in the jobtracker's memory (each task status is updated every few seconds, for example) makes it very difficult to retrofit HA into the jobtracker service.

With the jobtracker's responsibilities split between the resource manager and application master in YARN, making the service highly available became a divide-and-conquer problem: provide HA for the resource manager, then for YARN applications (on a per-application basis). And indeed, Hadoop 2 supports HA both

5. As of Hadoop 2.5.1, the YARN timeline server does not yet store MapReduce job history, so a MapReduce job history server daemon is still needed (see "Cluster Setup and Installation" on page 288).

6. Arun C. Murthy, "The Next Generation of Apache Hadoop MapReduce," February 14, 2011.

for the resource manager and for the application master for MapReduce jobs. Failure recovery in YARN is discussed in more detail in "Failures" on page 193.

Utilization
: In MapReduce 1, each tasktracker is configured with a static allocation of fixed-size "slots," which are divided into map slots and reduce slots at configuration time. A map slot can only be used to run a map task, and a reduce slot can only be used for a reduce task.

  In YARN, a node manager manages a pool of resources, rather than a fixed number of designated slots. MapReduce running on YARN will not hit the situation where a reduce task has to wait because only map slots are available on the cluster, which can happen in MapReduce 1. If the resources to run the task are available, then the application will be eligible for them.

  Furthermore, resources in YARN are fine grained, so an application can make a request for what it needs, rather than for an indivisible slot, which may be too big (which is wasteful of resources) or too small (which may cause a failure) for the particular task.

Multitenancy
: In some ways, the biggest benefit of YARN is that it opens up Hadoop to other types of distributed application beyond MapReduce. MapReduce is just one YARN application among many.

  It is even possible for users to run different versions of MapReduce on the same YARN cluster, which makes the process of upgrading MapReduce more manageable. (Note, however, that some parts of MapReduce, such as the job history server and the shuffle handler, as well as YARN itself, still need to be upgraded across the cluster.)

Since Hadoop 2 is widely used and is the latest stable version, in the rest of this book the term "MapReduce" refers to MapReduce 2 unless otherwise stated. Chapter 7 looks in detail at how MapReduce running on YARN works.

Scheduling in YARN

In an ideal world, the requests that a YARN application makes would be granted immediately. In the real world, however, resources are limited, and on a busy cluster, an application will often need to wait to have some of its requests fulfilled. It is the job of the YARN scheduler to allocate resources to applications according to some defined policy. Scheduling in general is a difficult problem and there is no one "best" policy, which is why YARN provides a choice of schedulers and configurable policies. We look at these next.

Scheduler Options

Three schedulers are available in YARN: the FIFO, Capacity, and Fair Schedulers. The FIFO Scheduler places applications in a queue and runs them in the order of submission (first in, first out). Requests for the first application in the queue are allocated first; once its requests have been satisfied, the next application in the queue is served, and so on.

The FIFO Scheduler has the merit of being simple to understand and not needing any configuration, but it's not suitable for shared clusters. Large applications will use all the resources in a cluster, so each application has to wait its turn. On a shared cluster it is better to use the Capacity Scheduler or the Fair Scheduler. Both of these allow long-running jobs to complete in a timely manner, while still allowing users who are running concurrent smaller ad hoc queries to get results back in a reasonable time.

The difference between schedulers is illustrated in Figure 4-3, which shows that under the FIFO Scheduler (i) the small job is blocked until the large job completes.

With the Capacity Scheduler (ii in Figure 4-3), a separate dedicated queue allows the small job to start as soon as it is submitted, although this is at the cost of overall cluster utilization since the queue capacity is reserved for jobs in that queue. This means that the large job finishes later than when using the FIFO Scheduler.

With the Fair Scheduler (iii in Figure 4-3), there is no need to reserve a set amount of capacity, since it will dynamically balance resources between all running jobs. Just after the first (large) job starts, it is the only job running, so it gets all the resources in the cluster. When the second (small) job starts, it is allocated half of the cluster resources so that each job is using its fair share of resources.

Note that there is a lag between the time the second job starts and when it receives its fair share, since it has to wait for resources to free up as containers used by the first job complete. After the small job completes and no longer requires resources, the large job goes back to using the full cluster capacity again. The overall effect is both high cluster utilization and timely small job completion.

Figure 4-3 contrasts the basic operation of the three schedulers. In the next two sections, we examine some of the more advanced configuration options for the Capacity and Fair Schedulers.

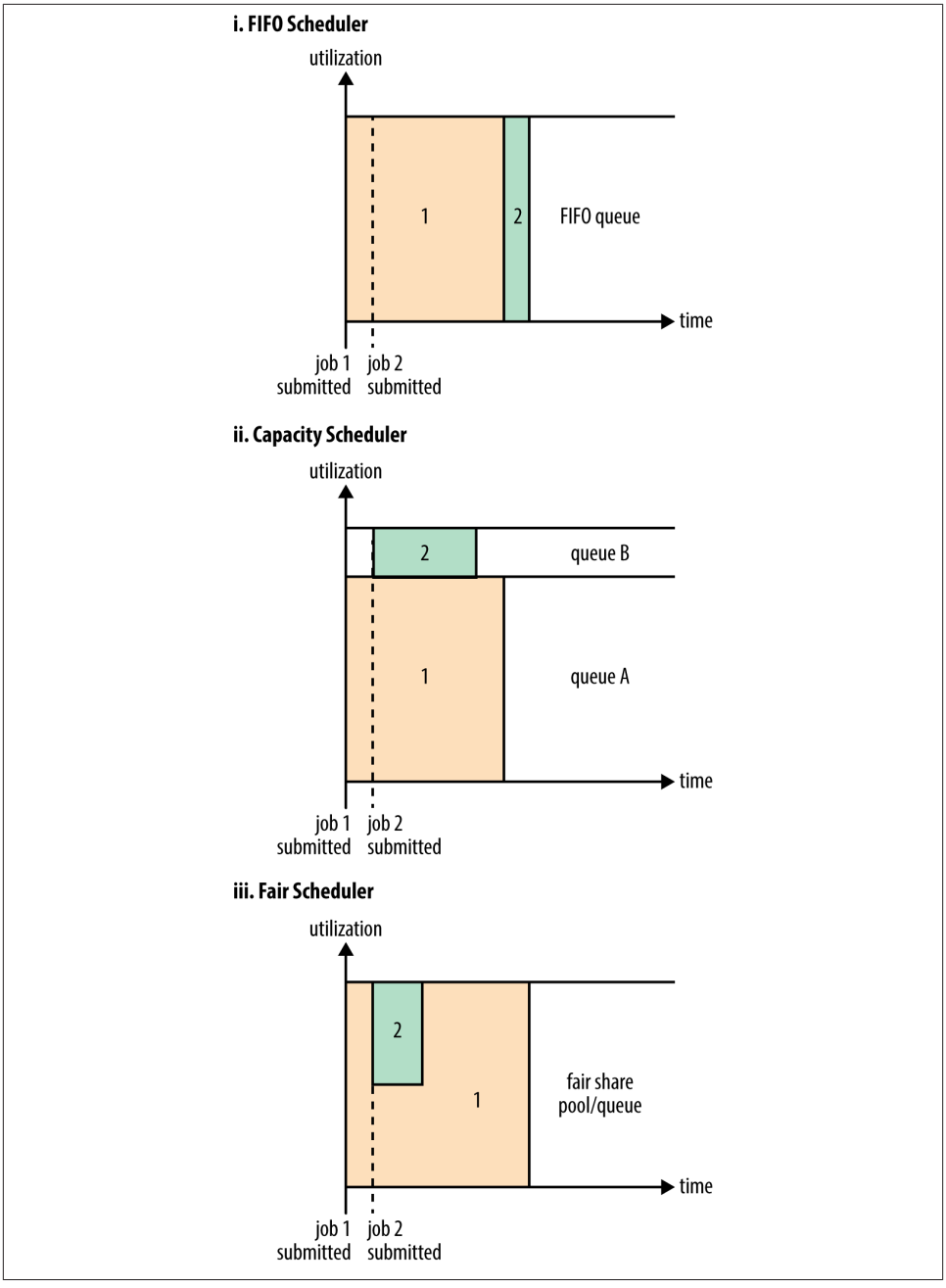

Figure 4-3. Cluster utilization over time when running a large job and a small job under the FIFO Scheduler (i), Capacity Scheduler (ii), and Fair Scheduler (iii)

Capacity Scheduler Configuration

The Capacity Scheduler allows sharing of a Hadoop cluster along organizational lines, whereby each organization is allocated a certain capacity of the overall cluster. Each organization is set up with a dedicated queue that is configured to use a given fraction of the cluster capacity. Queues may be further divided in hierarchical fashion, allowing each organization to share its cluster allowance between different groups of users within the organization. Within a queue, applications are scheduled using FIFO scheduling.

As we saw in Figure 4-3, a single job does not use more resources than its queue's capacity. However, if there is more than one job in the queue and there are idle resources available, then the Capacity Scheduler may allocate the spare resources to jobs in the queue, even if that causes the queue's capacity to be exceeded.[7] This behavior is known as *queue elasticity*.

In normal operation, the Capacity Scheduler does not preempt containers by forcibly killing them,[8] so if a queue is under capacity due to lack of demand, and then demand increases, the queue will only return to capacity as resources are released from other queues as containers complete. It is possible to mitigate this by configuring queues with a maximum capacity so that they don't eat into other queues' capacities too much. This is at the cost of queue elasticity, of course, so a reasonable trade-off should be found by trial and error.

Imagine a queue hierarchy that looks like this:

```
root
├── prod
└── dev
    ├── eng
    └── science
```

The listing in Example 4-1 shows a sample Capacity Scheduler configuration file, called *capacity-scheduler.xml*, for this hierarchy. It defines two queues under the `root` queue, `prod` and `dev`, which have 40% and 60% of the capacity, respectively. Notice that a particular queue is configured by setting configuration properties of the form `yarn.scheduler.capacity.<queue-path>.<sub-property>`, where `<queue-path>` is the hierarchical (dotted) path of the queue, such as `root.prod`.

7. If the property `yarn.scheduler.capacity.<queue-path>.user-limit-factor` is set to a value larger than 1 (the default), then a single job is allowed to use more than its queue's capacity.

8. However, the Capacity Scheduler can perform work-preserving preemption, where the resource manager asks applications to return containers to balance capacity.

Example 4-1. A basic configuration file for the Capacity Scheduler

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>yarn.scheduler.capacity.root.queues</name>
    <value>prod,dev</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.queues</name>
    <value>eng,science</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.prod.capacity</name>
    <value>40</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.capacity</name>
    <value>60</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.maximum-capacity</name>
    <value>75</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.eng.capacity</name>
    <value>50</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.science.capacity</name>
    <value>50</value>
  </property>
</configuration>
```

As you can see, the `dev` queue is further divided into `eng` and `science` queues of equal capacity. So that the `dev` queue does not use up all the cluster resources when the `prod` queue is idle, it has its maximum capacity set to 75%. In other words, the `prod` queue always has 25% of the cluster available for immediate use. Since no maximum capacities have been set for other queues, it's possible for jobs in the `eng` or `science` queues to use all of the `dev` queue's capacity (up to 75% of the cluster), or indeed for the `prod` queue to use the entire cluster.

Beyond configuring queue hierarchies and capacities, there are settings to control the maximum number of resources a single user or application can be allocated, how many applications can be running at any one time, and ACLs on queues. See the reference page for details.

Queue placement

The way that you specify which queue an application is placed in is specific to the application. For example, in MapReduce, you set the property `mapreduce.job.queuename` to the name of the queue you want to use. If the queue does not exist, then you'll get an error at submission time. If no queue is specified, applications will be placed in a queue called `default`.

For the Capacity Scheduler, the queue name should be the last part of the hierarchical name since the full hierarchical name is not recognized. So, for the preceding example configuration, `prod` and `eng` are OK, but `root.dev.eng` and `dev.eng` do not work.

Fair Scheduler Configuration

The Fair Scheduler attempts to allocate resources so that all running applications get the same share of resources. Figure 4-3 showed how fair sharing works for applications in the same queue; however, fair sharing actually works *between* queues, too, as we'll see next.

The terms *queue* and *pool* are used interchangeably in the context of the Fair Scheduler.

To understand how resources are shared between queues, imagine two users *A* and *B*, each with their own queue (Figure 4-4). *A* starts a job, and it is allocated all the resources available since there is no demand from *B*. Then *B* starts a job while *A*'s job is still running, and after a while each job is using half of the resources, in the way we saw earlier. Now if *B* starts a second job while the other jobs are still running, it will share its resources with *B*'s other job, so each of *B*'s jobs will have one-fourth of the resources, while *A*'s will continue to have half. The result is that resources are shared fairly between users.

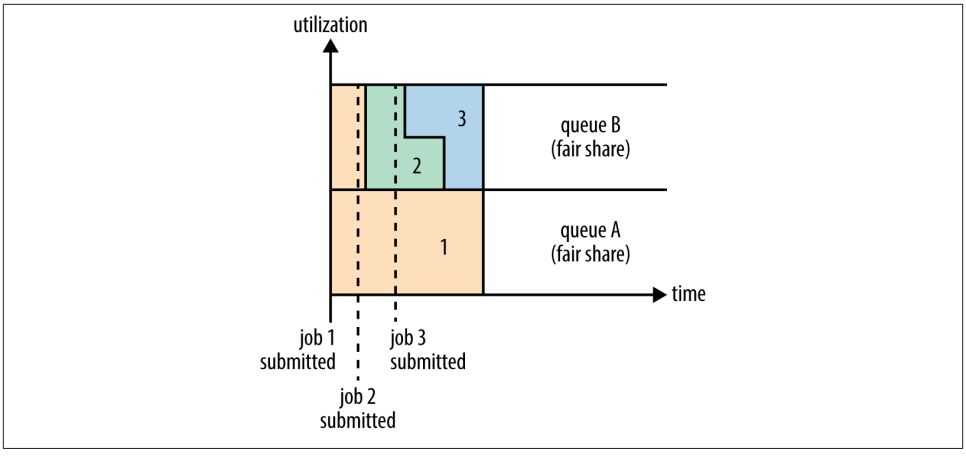

Figure 4-4. Fair sharing between user queues

Enabling the Fair Scheduler

The scheduler in use is determined by the setting of `yarn.resourcemanager.scheduler.class`. The Capacity Scheduler is used by default (although the Fair Scheduler is the default in some Hadoop distributions, such as CDH), but this can be changed by setting `yarn.resourcemanager.scheduler.class` in *yarn-site.xml* to the fully qualified classname of the scheduler, `org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.FairScheduler`.

Queue configuration

The Fair Scheduler is configured using an allocation file named *fair-scheduler.xml* that is loaded from the classpath. (The name can be changed by setting the property `yarn.scheduler.fair.allocation.file`.) In the absence of an allocation file, the Fair Scheduler operates as described earlier: each application is placed in a queue named after the user and queues are created dynamically when users submit their first applications.

Per-queue configuration is specified in the allocation file. This allows configuration of hierarchical queues like those supported by the Capacity Scheduler. For example, we can define `prod` and `dev` queues like we did for the Capacity Scheduler using the allocation file in Example 4-2.

Example 4-2. An allocation file for the Fair Scheduler

```
<?xml version="1.0"?>
<allocations>
  <defaultQueueSchedulingPolicy>fair</defaultQueueSchedulingPolicy>

  <queue name="prod">
```

```
    <weight>40</weight>
    <schedulingPolicy>fifo</schedulingPolicy>
  </queue>

  <queue name="dev">
    <weight>60</weight>
    <queue name="eng" />
    <queue name="science" />
  </queue>

  <queuePlacementPolicy>
    <rule name="specified" create="false" />
    <rule name="primaryGroup" create="false" />
    <rule name="default" queue="dev.eng" />
  </queuePlacementPolicy>
</allocations>
```

The queue hierarchy is defined using nested `queue` elements. All queues are children of the `root` queue, even if not actually nested in a `root` `queue` element. Here we subdivide the `dev` queue into a queue called `eng` and another called `science`.

Queues can have weights, which are used in the fair share calculation. In this example, the cluster allocation is considered fair when it is divided into a 40:60 proportion between `prod` and `dev`. The `eng` and `science` queues do not have weights specified, so they are divided evenly. Weights are not quite the same as percentages, even though the example uses numbers that add up to 100 for the sake of simplicity. We could have specified weights of 2 and 3 for the `prod` and `dev` queues to achieve the same queue weighting.

When setting weights, remember to consider the default queue and dynamically created queues (such as queues named after users). These are not specified in the allocation file, but still have weight 1.

Queues can have different scheduling policies. The default policy for queues can be set in the top-level `defaultQueueSchedulingPolicy` element; if it is omitted, fair scheduling is used. Despite its name, the Fair Scheduler also supports a FIFO (`fifo`) policy on queues, as well as Dominant Resource Fairness (`drf`), described later in the chapter.

The policy for a particular queue can be overridden using the `schedulingPolicy` element for that queue. In this case, the `prod` queue uses FIFO scheduling since we want each production job to run serially and complete in the shortest possible amount of time. Note that fair sharing is still used to divide resources between the `prod` and `dev` queues, as well as between (and within) the `eng` and `science` queues.

Although not shown in this allocation file, queues can be configured with minimum and maximum resources, and a maximum number of running applications. (See the reference page for details.) The minimum resources setting is not a hard limit, but rather is used by the scheduler to prioritize resource allocations. If two queues are below their fair share, then the one that is furthest below its minimum is allocated resources first. The minimum resource setting is also used for preemption, discussed momentarily.

Queue placement

The Fair Scheduler uses a rules-based system to determine which queue an application is placed in. In Example 4-2, the `queuePlacementPolicy` element contains a list of rules, each of which is tried in turn until a match occurs. The first rule, `specified`, places an application in the queue it specified; if none is specified, or if the specified queue doesn't exist, then the rule doesn't match and the next rule is tried. The `primaryGroup` rule tries to place an application in a queue with the name of the user's primary Unix group; if there is no such queue, rather than creating it, the next rule is tried. The `default` rule is a catch-all and always places the application in the `dev.eng` queue.

The `queuePlacementPolicy` can be omitted entirely, in which case the default behavior is as if it had been specified with the following:

```
<queuePlacementPolicy>
  <rule name="specified" />
  <rule name="user" />
</queuePlacementPolicy>
```

In other words, unless the queue is explicitly specified, the user's name is used for the queue, creating it if necessary.

Another simple queue placement policy is one where all applications are placed in the same (default) queue. This allows resources to be shared fairly between applications, rather than users. The definition is equivalent to this:

```
<queuePlacementPolicy>
  <rule name="default" />
</queuePlacementPolicy>
```

It's also possible to set this policy without using an allocation file, by setting `yarn.scheduler.fair.user-as-default-queue` to `false` so that applications will be placed in the default queue rather than a per-user queue. In addition, `yarn.scheduler.fair.allow-undeclared-pools` should be set to `false` so that users can't create queues on the fly.

Preemption

When a job is submitted to an empty queue on a busy cluster, the job cannot start until resources free up from jobs that are already running on the cluster. To make the time taken for a job to start more predictable, the Fair Scheduler supports *preemption*.

Preemption allows the scheduler to kill containers for queues that are running with more than their fair share of resources so that the resources can be allocated to a queue that is under its fair share. Note that preemption reduces overall cluster efficiency, since the terminated containers need to be reexecuted.

Preemption is enabled globally by setting `yarn.scheduler.fair.preemption` to `true`. There are two relevant preemption timeout settings: one for minimum share and one for fair share, both specified in seconds. By default, the timeouts are not set, so you need to set at least one to allow containers to be preempted.

If a queue waits for as long as its *minimum share preemption timeout* without receiving its minimum guaranteed share, then the scheduler may preempt other containers. The default timeout is set for all queues via the `defaultMinSharePreemptionTimeout` top-level element in the allocation file, and on a per-queue basis by setting the `minSharePreemptionTimeout` element for a queue.

Likewise, if a queue remains below *half* of its fair share for as long as the *fair share preemption timeout*, then the scheduler may preempt other containers. The default timeout is set for all queues via the `defaultFairSharePreemptionTimeout` top-level element in the allocation file, and on a per-queue basis by setting `fairSharePreemptionTimeout` on a queue. The threshold may also be changed from its default of 0.5 by setting `defaultFairSharePreemptionThreshold` and `fairSharePreemptionThreshold` (per-queue).

Delay Scheduling

All the YARN schedulers try to honor locality requests. On a busy cluster, if an application requests a particular node, there is a good chance that other containers are running on it at the time of the request. The obvious course of action is to immediately loosen the locality requirement and allocate a container on the same rack. However, it has been observed in practice that waiting a short time (no more than a few seconds) can dramatically increase the chances of being allocated a container on the requested node, and therefore increase the efficiency of the cluster. This feature is called *delay scheduling*, and it is supported by both the Capacity Scheduler and the Fair Scheduler.

Every node manager in a YARN cluster periodically sends a heartbeat request to the resource manager—by default, one per second. Heartbeats carry information about the node manager's running containers and the resources available for new containers, so each heartbeat is a potential *scheduling opportunity* for an application to run a container.

When using delay scheduling, the scheduler doesn't simply use the first scheduling opportunity it receives, but waits for up to a given maximum number of scheduling opportunities to occur before loosening the locality constraint and taking the next scheduling opportunity.

For the Capacity Scheduler, delay scheduling is configured by setting `yarn.scheduler.capacity.node-locality-delay` to a positive integer representing the number of scheduling opportunities that it is prepared to miss before loosening the node constraint to match any node in the same rack.

The Fair Scheduler also uses the number of scheduling opportunities to determine the delay, although it is expressed as a proportion of the cluster size. For example, setting `yarn.scheduler.fair.locality.threshold.node` to 0.5 means that the scheduler should wait until half of the nodes in the cluster have presented scheduling opportunities before accepting another node in the same rack. There is a corresponding property, `yarn.scheduler.fair.locality.threshold.rack`, for setting the threshold before another rack is accepted instead of the one requested.

Dominant Resource Fairness

When there is only a single resource type being scheduled, such as memory, then the concept of capacity or fairness is easy to determine. If two users are running applications, you can measure the amount of memory that each is using to compare the two applications. However, when there are multiple resource types in play, things get more complicated. If one user's application requires lots of CPU but little memory and the other's requires little CPU and lots of memory, how are these two applications compared?

The way that the schedulers in YARN address this problem is to look at each user's dominant resource and use it as a measure of the cluster usage. This approach is called *Dominant Resource Fairness*, or DRF for short.[9] The idea is best illustrated with a simple example.

Imagine a cluster with a total of 100 CPUs and 10 TB of memory. Application *A* requests containers of (2 CPUs, 300 GB), and application *B* requests containers of (6 CPUs, 100 GB). *A*'s request is (2%, 3%) of the cluster, so memory is dominant since its proportion (3%) is larger than CPU's (2%). *B*'s request is (6%, 1%), so CPU is dominant. Since *B*'s container requests are twice as big in the dominant resource (6% versus 3%), it will be allocated half as many containers under fair sharing.

By default DRF is not used, so during resource calculations, only memory is considered and CPU is ignored. The Capacity Scheduler can be configured to use DRF by setting `yarn.scheduler.capacity.resource-calculator` to `org.apache.hadoop.yarn.util.resource.DominantResourceCalculator` in *capacity-scheduler.xml*.

For the Fair Scheduler, DRF can be enabled by setting the top-level element `defaultQueueSchedulingPolicy` in the allocation file to `drf`.

9. DRF was introduced in Ghodsi et al.'s "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types," March 2011.

Further Reading

This chapter has given a short overview of YARN. For more detail, see *Apache Hadoop YARN* by Arun C. Murthy et al. (Addison-Wesley, 2014).

CHAPTER 5
Hadoop I/O

Hadoop comes with a set of primitives for data I/O. Some of these are techniques that are more general than Hadoop, such as data integrity and compression, but deserve special consideration when dealing with multiterabyte datasets. Others are Hadoop tools or APIs that form the building blocks for developing distributed systems, such as serialization frameworks and on-disk data structures.

Data Integrity

Users of Hadoop rightly expect that no data will be lost or corrupted during storage or processing. However, because every I/O operation on the disk or network carries with it a small chance of introducing errors into the data that it is reading or writing, when the volumes of data flowing through the system are as large as the ones Hadoop is capable of handling, the chance of data corruption occurring is high.

The usual way of detecting corrupted data is by computing a *checksum* for the data when it first enters the system, and again whenever it is transmitted across a channel that is unreliable and hence capable of corrupting the data. The data is deemed to be corrupt if the newly generated checksum doesn't exactly match the original. This technique doesn't offer any way to fix the data—it is merely error detection. (And this is a reason for not using low-end hardware; in particular, be sure to use ECC memory.) Note that it is possible that it's the checksum that is corrupt, not the data, but this is very unlikely, because the checksum is much smaller than the data.

A commonly used error-detecting code is CRC-32 (32-bit cyclic redundancy check), which computes a 32-bit integer checksum for input of any size. CRC-32 is used for checksumming in Hadoop's `ChecksumFileSystem`, while HDFS uses a more efficient variant called CRC-32C.

Data Integrity in HDFS

HDFS transparently checksums all data written to it and by default verifies checksums when reading data. A separate checksum is created for every `dfs.bytes-per-checksum` bytes of data. The default is 512 bytes, and because a CRC-32C checksum is 4 bytes long, the storage overhead is less than 1%.

Datanodes are responsible for verifying the data they receive before storing the data and its checksum. This applies to data that they receive from clients and from other datanodes during replication. A client writing data sends it to a pipeline of datanodes (as explained in Chapter 3), and the last datanode in the pipeline verifies the checksum. If the datanode detects an error, the client receives a subclass of `IOException`, which it should handle in an application-specific manner (for example, by retrying the operation).

When clients read data from datanodes, they verify checksums as well, comparing them with the ones stored at the datanodes. Each datanode keeps a persistent log of checksum verifications, so it knows the last time each of its blocks was verified. When a client successfully verifies a block, it tells the datanode, which updates its log. Keeping statistics such as these is valuable in detecting bad disks.

In addition to block verification on client reads, each datanode runs a `DataBlockScanner` in a background thread that periodically verifies all the blocks stored on the datanode. This is to guard against corruption due to "bit rot" in the physical storage media. See "Datanode block scanner" on page 328 for details on how to access the scanner reports.

Because HDFS stores replicas of blocks, it can "heal" corrupted blocks by copying one of the good replicas to produce a new, uncorrupt replica. The way this works is that if a client detects an error when reading a block, it reports the bad block and the datanode it was trying to read from to the namenode before throwing a `ChecksumException`. The namenode marks the block replica as corrupt so it doesn't direct any more clients to it or try to copy this replica to another datanode. It then schedules a copy of the block to be replicated on another datanode, so its replication factor is back at the expected level. Once this has happened, the corrupt replica is deleted.

It is possible to disable verification of checksums by passing `false` to the `setVerifyChecksum()` method on `FileSystem` before using the `open()` method to read a file. The same effect is possible from the shell by using the `-ignoreCrc` option with the `-get` or the equivalent `-copyToLocal` command. This feature is useful if you have a corrupt file that you want to inspect so you can decide what to do with it. For example, you might want to see whether it can be salvaged before you delete it.

You can find a file's checksum with `hadoop fs -checksum`. This is useful to check whether two files in HDFS have the same contents—something that *distcp* does, for example (see "Parallel Copying with distcp" on page 76).

LocalFileSystem

The Hadoop `LocalFileSystem` performs client-side checksumming. This means that when you write a file called *filename*, the filesystem client transparently creates a hidden file, *.filename.crc*, in the same directory containing the checksums for each chunk of the file. The chunk size is controlled by the `file.bytes-per-checksum` property, which defaults to 512 bytes. The chunk size is stored as metadata in the *.crc* file, so the file can be read back correctly even if the setting for the chunk size has changed. Checksums are verified when the file is read, and if an error is detected, `LocalFileSystem` throws a `ChecksumException`.

Checksums are fairly cheap to compute (in Java, they are implemented in native code), typically adding a few percent overhead to the time to read or write a file. For most applications, this is an acceptable price to pay for data integrity. It is, however, possible to disable checksums, which is typically done when the underlying filesystem supports checksums natively. This is accomplished by using `RawLocalFileSystem` in place of `LocalFileSystem`. To do this globally in an application, it suffices to remap the implementation for `file` URIs by setting the property `fs.file.impl` to the value `org.apache.hadoop.fs.RawLocalFileSystem`. Alternatively, you can directly create a `RawLocalFileSystem` instance, which may be useful if you want to disable checksum verification for only some reads, for example:

```
Configuration conf = ...
FileSystem fs = new RawLocalFileSystem();
fs.initialize(null, conf);
```

ChecksumFileSystem

`LocalFileSystem` uses `ChecksumFileSystem` to do its work, and this class makes it easy to add checksumming to other (noncheckssummed) filesystems, as `ChecksumFileSystem` is just a wrapper around `FileSystem`. The general idiom is as follows:

```
FileSystem rawFs = ...
FileSystem checksummedFs = new ChecksumFileSystem(rawFs);
```

The underlying filesystem is called the *raw* filesystem, and may be retrieved using the `getRawFileSystem()` method on `ChecksumFileSystem`. `ChecksumFileSystem` has a few more useful methods for working with checksums, such as `getChecksumFile()` for getting the path of a checksum file for any file. Check the documentation for the others.

If an error is detected by `ChecksumFileSystem` when reading a file, it will call its `reportChecksumFailure()` method. The default implementation does nothing, but `LocalFileSystem` moves the offending file and its checksum to a side directory on the same device called *bad_files*. Administrators should periodically check for these bad files and take action on them.

Compression

File compression brings two major benefits: it reduces the space needed to store files, and it speeds up data transfer across the network or to or from disk. When dealing with large volumes of data, both of these savings can be significant, so it pays to carefully consider how to use compression in Hadoop.

There are many different compression formats, tools, and algorithms, each with different characteristics. Table 5-1 lists some of the more common ones that can be used with Hadoop.

Table 5-1. A summary of compression formats

Compression format	Tool	Algorithm	Filename extension	Splittable?
DEFLATE[a]	N/A	DEFLATE	*.deflate*	No
gzip	*gzip*	DEFLATE	*.gz*	No
bzip2	*bzip2*	bzip2	*.bz2*	Yes
LZO	*lzop*	LZO	*.lzo*	No[b]
LZ4	N/A	LZ4	*.lz4*	No
Snappy	N/A	Snappy	*.snappy*	No

[a] DEFLATE is a compression algorithm whose standard implementation is zlib. There is no commonly available command-line tool for producing files in DEFLATE format, as gzip is normally used. (Note that the gzip file format is DEFLATE with extra headers and a footer.) The *.deflate* filename extension is a Hadoop convention.

[b] However, LZO files are splittable if they have been indexed in a preprocessing step. See "Compression and Input Splits" on page 105.

All compression algorithms exhibit a space/time trade-off: faster compression and decompression speeds usually come at the expense of smaller space savings. The tools listed in Table 5-1 typically give some control over this trade-off at compression time by offering nine different options: `-1` means optimize for speed, and `-9` means optimize for space. For example, the following command creates a compressed file *file.gz* using the fastest compression method:

```
% gzip -1 file
```

The different tools have very different compression characteristics. gzip is a general-purpose compressor and sits in the middle of the space/time trade-off. bzip2 compresses more effectively than gzip, but is slower. bzip2's decompression speed is faster than its compression speed, but it is still slower than the other formats. LZO, LZ4, and Snappy, on the other hand, all optimize for speed and are around an order of magnitude faster

than gzip, but compress less effectively. Snappy and LZ4 are also significantly faster than LZO for decompression.[1]

The "Splittable" column in Table 5-1 indicates whether the compression format supports splitting (that is, whether you can seek to any point in the stream and start reading from some point further on). Splittable compression formats are especially suitable for MapReduce; see "Compression and Input Splits" on page 105 for further discussion.

Codecs

A *codec* is the implementation of a compression-decompression algorithm. In Hadoop, a codec is represented by an implementation of the `CompressionCodec` interface. So, for example, `GzipCodec` encapsulates the compression and decompression algorithm for gzip. Table 5-2 lists the codecs that are available for Hadoop.

Table 5-2. Hadoop compression codecs

Compression format	Hadoop CompressionCodec
DEFLATE	`org.apache.hadoop.io.compress.DefaultCodec`
gzip	`org.apache.hadoop.io.compress.GzipCodec`
bzip2	`org.apache.hadoop.io.compress.BZip2Codec`
LZO	`com.hadoop.compression.lzo.LzopCodec`
LZ4	`org.apache.hadoop.io.compress.Lz4Codec`
Snappy	`org.apache.hadoop.io.compress.SnappyCodec`

The LZO libraries are GPL licensed and may not be included in Apache distributions, so for this reason the Hadoop codecs must be downloaded separately from Google (or GitHub, which includes bug fixes and more tools). The `LzopCodec`, which is compatible with the *lzop* tool, is essentially the LZO format with extra headers, and is the one you normally want. There is also an `LzoCodec` for the pure LZO format, which uses the *.lzo_deflate* filename extension (by analogy with DEFLATE, which is gzip without the headers).

Compressing and decompressing streams with CompressionCodec

`CompressionCodec` has two methods that allow you to easily compress or decompress data. To compress data being written to an output stream, use the `createOutputStream(OutputStream out)` method to create a `CompressionOutputStream` to which you write your uncompressed data to have it written in compressed form to the underlying stream. Conversely, to decompress data being read from an input stream,

1. For a comprehensive set of compression benchmarks, *jvm-compressor-benchmark* is a good reference for JVM-compatible libraries (including some native libraries).

call `createInputStream(InputStream in)` to obtain a `CompressionInputStream`, which allows you to read uncompressed data from the underlying stream.

`CompressionOutputStream` and `CompressionInputStream` are similar to `java.util.zip.DeflaterOutputStream` and `java.util.zip.DeflaterInputStream`, except that both of the former provide the ability to reset their underlying compressor or decompressor. This is important for applications that compress sections of the data stream as separate blocks, such as in a `SequenceFile`, described in "SequenceFile" on page 127.

Example 5-1 illustrates how to use the API to compress data read from standard input and write it to standard output.

Example 5-1. A program to compress data read from standard input and write it to standard output

```
public class StreamCompressor {

  public static void main(String[] args) throws Exception {
    String codecClassname = args[0];
    Class<?> codecClass = Class.forName(codecClassname);
    Configuration conf = new Configuration();
    CompressionCodec codec = (CompressionCodec)
      ReflectionUtils.newInstance(codecClass, conf);

    CompressionOutputStream out = codec.createOutputStream(System.out);
    IOUtils.copyBytes(System.in, out, 4096, false);
    out.finish();
  }
}
```

The application expects the fully qualified name of the `CompressionCodec` implementation as the first command-line argument. We use `ReflectionUtils` to construct a new instance of the codec, then obtain a compression wrapper around `System.out`. Then we call the utility method `copyBytes()` on `IOUtils` to copy the input to the output, which is compressed by the `CompressionOutputStream`. Finally, we call `finish()` on `CompressionOutputStream`, which tells the compressor to finish writing to the compressed stream, but doesn't close the stream. We can try it out with the following command line, which compresses the string "Text" using the `StreamCompressor` program with the `GzipCodec`, then decompresses it from standard input using *gunzip*:

```
% echo "Text" | hadoop StreamCompressor org.apache.hadoop.io.compress.GzipCodec \
  | gunzip -
Text
```

Inferring CompressionCodecs using CompressionCodecFactory

If you are reading a compressed file, normally you can infer which codec to use by looking at its filename extension. A file ending in *.gz* can be read with `GzipCodec`, and so on. The extensions for each compression format are listed in Table 5-1.

`CompressionCodecFactory` provides a way of mapping a filename extension to a `CompressionCodec` using its `getCodec()` method, which takes a `Path` object for the file in question. Example 5-2 shows an application that uses this feature to decompress files.

Example 5-2. A program to decompress a compressed file using a codec inferred from the file's extension

```
public class FileDecompressor {

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

    Path inputPath = new Path(uri);
    CompressionCodecFactory factory = new CompressionCodecFactory(conf);
    CompressionCodec codec = factory.getCodec(inputPath);
    if (codec == null) {
      System.err.println("No codec found for " + uri);
      System.exit(1);
    }

    String outputUri =
        CompressionCodecFactory.removeSuffix(uri, codec.getDefaultExtension());

    InputStream in = null;
    OutputStream out = null;
    try {
      in = codec.createInputStream(fs.open(inputPath));
      out = fs.create(new Path(outputUri));
      IOUtils.copyBytes(in, out, conf);
    } finally {
      IOUtils.closeStream(in);
      IOUtils.closeStream(out);
    }
  }
}
```

Once the codec has been found, it is used to strip off the file suffix to form the output filename (via the `removeSuffix()` static method of `CompressionCodecFactory`). In this way, a file named *file.gz* is decompressed to *file* by invoking the program as follows:

```
% hadoop FileDecompressor file.gz
```

`CompressionCodecFactory` loads all the codecs in Table 5-2, except LZO, as well as any listed in the `io.compression.codecs` configuration property (Table 5-3). By default, the property is empty; you would need to alter it only if you have a custom codec that you wish to register (such as the externally hosted LZO codecs). Each codec knows its default filename extension, thus permitting `CompressionCodecFactory` to search through the registered codecs to find a match for the given extension (if any).

Table 5-3. Compression codec properties

Property name	Type	Default value	Description
`io.compression.codecs`	Comma-separated `Class` names		A list of additional `CompressionCodec` classes for compression/decompression

Native libraries

For performance, it is preferable to use a native library for compression and decompression. For example, in one test, using the native gzip libraries reduced decompression times by up to 50% and compression times by around 10% (compared to the built-in Java implementation). Table 5-4 shows the availability of Java and native implementations for each compression format. All formats have native implementations, but not all have a Java implementation (LZO, for example).

Table 5-4. Compression library implementations

Compression format	Java implementation?	Native implementation?
DEFLATE	Yes	Yes
gzip	Yes	Yes
bzip2	Yes	Yes
LZO	No	Yes
LZ4	No	Yes
Snappy	No	Yes

The Apache Hadoop binary tarball comes with prebuilt native compression binaries for 64-bit Linux, called *libhadoop.so*. For other platforms, you will need to compile the libraries yourself, following the *BUILDING.txt* instructions at the top level of the source tree.

The native libraries are picked up using the Java system property `java.library.path`. The *hadoop* script in the *etc/hadoop* directory sets this property for you, but if you don't use this script, you will need to set the property in your application.

By default, Hadoop looks for native libraries for the platform it is running on, and loads them automatically if they are found. This means you don't have to change any configuration settings to use the native libraries. In some circumstances, however, you may wish to disable use of native libraries, such as when you are debugging a compression-related problem. You can do this by setting the property `io.native.lib.available` to `false`, which ensures that the built-in Java equivalents will be used (if they are available).

CodecPool. If you are using a native library and you are doing a lot of compression or decompression in your application, consider using `CodecPool`, which allows you to

reuse compressors and decompressors, thereby amortizing the cost of creating these objects.

The code in Example 5-3 shows the API, although in this program, which creates only a single `Compressor`, there is really no need to use a pool.

Example 5-3. A program to compress data read from standard input and write it to standard output using a pooled compressor

```
public class PooledStreamCompressor {

  public static void main(String[] args) throws Exception {
    String codecClassname = args[0];
    Class<?> codecClass = Class.forName(codecClassname);
    Configuration conf = new Configuration();
    CompressionCodec codec = (CompressionCodec)
        ReflectionUtils.newInstance(codecClass, conf);
    Compressor compressor = null;
    try {
      compressor = CodecPool.getCompressor(codec);
      CompressionOutputStream out =
          codec.createOutputStream(System.out, compressor);
      IOUtils.copyBytes(System.in, out, 4096, false);
      out.finish();
    } finally {
      CodecPool.returnCompressor(compressor);
    }
  }
}
```

We retrieve a `Compressor` instance from the pool for a given `CompressionCodec`, which we use in the codec's overloaded `createOutputStream()` method. By using a `finally` block, we ensure that the compressor is returned to the pool even if there is an `IOException` while copying the bytes between the streams.

Compression and Input Splits

When considering how to compress data that will be processed by MapReduce, it is important to understand whether the compression format supports splitting. Consider an uncompressed file stored in HDFS whose size is 1 GB. With an HDFS block size of 128 MB, the file will be stored as eight blocks, and a MapReduce job using this file as input will create eight input splits, each processed independently as input to a separate map task.

Imagine now that the file is a gzip-compressed file whose compressed size is 1 GB. As before, HDFS will store the file as eight blocks. However, creating a split for each block won't work, because it is impossible to start reading at an arbitrary point in the gzip stream and therefore impossible for a map task to read its split independently of the

others. The gzip format uses DEFLATE to store the compressed data, and DEFLATE stores data as a series of compressed blocks. The problem is that the start of each block is not distinguished in any way that would allow a reader positioned at an arbitrary point in the stream to advance to the beginning of the next block, thereby synchronizing itself with the stream. For this reason, gzip does not support splitting.

In this case, MapReduce will do the right thing and not try to split the gzipped file, since it knows that the input is gzip-compressed (by looking at the filename extension) and that gzip does not support splitting. This will work, but at the expense of locality: a single map will process the eight HDFS blocks, most of which will not be local to the map. Also, with fewer maps, the job is less granular and so may take longer to run.

If the file in our hypothetical example were an LZO file, we would have the same problem because the underlying compression format does not provide a way for a reader to synchronize itself with the stream. However, it is possible to preprocess LZO files using an indexer tool that comes with the Hadoop LZO libraries, which you can obtain from the Google and GitHub sites listed in "Codecs" on page 101. The tool builds an index of split points, effectively making them splittable when the appropriate MapReduce input format is used.

A bzip2 file, on the other hand, does provide a synchronization marker between blocks (a 48-bit approximation of pi), so it does support splitting. (Table 5-1 lists whether each compression format supports splitting.)

Which Compression Format Should I Use?

Hadoop applications process large datasets, so you should strive to take advantage of compression. Which compression format you use depends on such considerations as file size, format, and the tools you are using for processing. Here are some suggestions, arranged roughly in order of most to least effective:

- Use a container file format such as sequence files (see the section on page 127), Avro datafiles (see the section on page 352), ORCFiles (see the section on page 136), or Parquet files (see the section on page 370), all of which support both compression and splitting. A fast compressor such as LZO, LZ4, or Snappy is generally a good choice.
- Use a compression format that supports splitting, such as bzip2 (although bzip2 is fairly slow), or one that can be indexed to support splitting, such as LZO.
- Split the file into chunks in the application, and compress each chunk separately using any supported compression format (it doesn't matter whether it is splittable). In this case, you should choose the chunk size so that the compressed chunks are approximately the size of an HDFS block.

- Store the files uncompressed.

For large files, you should *not* use a compression format that does not support splitting on the whole file, because you lose locality and make MapReduce applications very inefficient.

Using Compression in MapReduce

As described in "Inferring CompressionCodecs using CompressionCodecFactory" on page 102, if your input files are compressed, they will be decompressed automatically as they are read by MapReduce, using the filename extension to determine which codec to use.

In order to compress the output of a MapReduce job, in the job configuration, set the `mapreduce.output.fileoutputformat.compress` property to `true` and set the `mapreduce.output.fileoutputformat.compress.codec` property to the classname of the compression codec you want to use. Alternatively, you can use the static convenience methods on `FileOutputFormat` to set these properties, as shown in Example 5-4.

Example 5-4. Application to run the maximum temperature job producing compressed output

```
public class MaxTemperatureWithCompression {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperatureWithCompression <input path> " +
          "<output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperature.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    FileOutputFormat.setCompressOutput(job, true);
    FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    System.exit(job.waitForCompletion(true) ? 0 : 1);
```

```
  }
}
```

We run the program over compressed input (which doesn't have to use the same compression format as the output, although it does in this example) as follows:

```
% hadoop MaxTemperatureWithCompression input/ncdc/sample.txt.gz output
```

Each part of the final output is compressed; in this case, there is a single part:

```
% gunzip -c output/part-r-00000.gz
1949    111
1950    22
```

If you are emitting sequence files for your output, you can set the `mapreduce.output.fileoutputformat.compress.type` property to control the type of compression to use. The default is `RECORD`, which compresses individual records. Changing this to `BLOCK`, which compresses groups of records, is recommended because it compresses better (see "The SequenceFile format" on page 133).

There is also a static convenience method on `SequenceFileOutputFormat` called `setOutputCompressionType()` to set this property.

The configuration properties to set compression for MapReduce job outputs are summarized in Table 5-5. If your MapReduce driver uses the `Tool` interface (described in "GenericOptionsParser, Tool, and ToolRunner" on page 148), you can pass any of these properties to the program on the command line, which may be more convenient than modifying your program to hardcode the compression properties.

Table 5-5. MapReduce compression properties

Property name	Type	Default value	Description
`mapreduce.output.fileoutputformat.compress`	`boolean`	`false`	Whether to compress outputs
`mapreduce.output.fileoutputformat.compress.codec`	`Class name`	`org.apache.hadoop.io.compress.DefaultCodec`	The compression codec to use for outputs
`mapreduce.output.fileoutputformat.compress.type`	`String`	`RECORD`	The type of compression to use for sequence file outputs: NONE, RECORD, or BLOCK

Compressing map output

Even if your MapReduce application reads and writes uncompressed data, it may benefit from compressing the intermediate output of the map phase. The map output is written to disk and transferred across the network to the reducer nodes, so by using a fast

compressor such as LZO, LZ4, or Snappy, you can get performance gains simply because the volume of data to transfer is reduced. The configuration properties to enable compression for map outputs and to set the compression format are shown in Table 5-6.

Table 5-6. Map output compression properties

Property name	Type	Default value	Description
`mapreduce.map.output.compress`	`boolean`	`false`	Whether to compress map outputs
`mapreduce.map.output.compress.codec`	`Class`	`org.apache.hadoop.io.compress.DefaultCodec`	The compression codec to use for map outputs

Here are the lines to add to enable gzip map output compression in your job (using the new API):

```
Configuration conf = new Configuration();
conf.setBoolean(Job.MAP_OUTPUT_COMPRESS, true);
conf.setClass(Job.MAP_OUTPUT_COMPRESS_CODEC, GzipCodec.class,
    CompressionCodec.class);
Job job = new Job(conf);
```

In the old API (see Appendix D), there are convenience methods on the `JobConf` object for doing the same thing:

```
conf.setCompressMapOutput(true);
conf.setMapOutputCompressorClass(GzipCodec.class);
```

Serialization

Serialization is the process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage. *Deserialization* is the reverse process of turning a byte stream back into a series of structured objects.

Serialization is used in two quite distinct areas of distributed data processing: for interprocess communication and for persistent storage.

In Hadoop, interprocess communication between nodes in the system is implemented using *remote procedure calls* (RPCs). The RPC protocol uses serialization to render the message into a binary stream to be sent to the remote node, which then deserializes the binary stream into the original message. In general, it is desirable that an RPC serialization format is:

Compact
: A compact format makes the best use of network bandwidth, which is the most scarce resource in a data center.

Fast
: Interprocess communication forms the backbone for a distributed system, so it is essential that there is as little performance overhead as possible for the serialization and deserialization process.

Extensible
: Protocols change over time to meet new requirements, so it should be straightforward to evolve the protocol in a controlled manner for clients and servers. For example, it should be possible to add a new argument to a method call and have the new servers accept messages in the old format (without the new argument) from old clients.

Interoperable
: For some systems, it is desirable to be able to support clients that are written in different languages to the server, so the format needs to be designed to make this possible.

On the face of it, the data format chosen for persistent storage would have different requirements from a serialization framework. After all, the lifespan of an RPC is less than a second, whereas persistent data may be read years after it was written. But it turns out, the four desirable properties of an RPC's serialization format are also crucial for a persistent storage format. We want the storage format to be compact (to make efficient use of storage space), fast (so the overhead in reading or writing terabytes of data is minimal), extensible (so we can transparently read data written in an older format), and interoperable (so we can read or write persistent data using different languages).

Hadoop uses its own serialization format, Writables, which is certainly compact and fast, but not so easy to extend or use from languages other than Java. Because Writables are central to Hadoop (most MapReduce programs use them for their key and value types), we look at them in some depth in the next three sections, before looking at some of the other serialization frameworks supported in Hadoop. Avro (a serialization system that was designed to overcome some of the limitations of Writables) is covered in Chapter 12.

The Writable Interface

The `Writable` interface defines two methods—one for writing its state to a `DataOutput` binary stream and one for reading its state from a `DataInput` binary stream:

```
package org.apache.hadoop.io;

import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

public interface Writable {
  void write(DataOutput out) throws IOException;
```

```
  void readFields(DataInput in) throws IOException;
}
```

Let's look at a particular `Writable` to see what we can do with it. We will use `IntWritable`, a wrapper for a Java `int`. We can create one and set its value using the `set()` method:

```
IntWritable writable = new IntWritable();
writable.set(163);
```

Equivalently, we can use the constructor that takes the integer value:

```
IntWritable writable = new IntWritable(163);
```

To examine the serialized form of the `IntWritable`, we write a small helper method that wraps a `java.io.ByteArrayOutputStream` in a `java.io.DataOutputStream` (an implementation of `java.io.DataOutput`) to capture the bytes in the serialized stream:

```
public static byte[] serialize(Writable writable) throws IOException {
  ByteArrayOutputStream out = new ByteArrayOutputStream();
  DataOutputStream dataOut = new DataOutputStream(out);
  writable.write(dataOut);
  dataOut.close();
  return out.toByteArray();
}
```

An integer is written using four bytes (as we see using JUnit 4 assertions):

```
byte[] bytes = serialize(writable);
assertThat(bytes.length, is(4));
```

The bytes are written in big-endian order (so the most significant byte is written to the stream first, which is dictated by the `java.io.DataOutput` interface), and we can see their hexadecimal representation by using a method on Hadoop's `StringUtils`:

```
assertThat(StringUtils.byteToHexString(bytes), is("000000a3"));
```

Let's try deserialization. Again, we create a helper method to read a `Writable` object from a byte array:

```
public static byte[] deserialize(Writable writable, byte[] bytes)
    throws IOException {
  ByteArrayInputStream in = new ByteArrayInputStream(bytes);
  DataInputStream dataIn = new DataInputStream(in);
  writable.readFields(dataIn);
  dataIn.close();
  return bytes;
}
```

We construct a new, value-less `IntWritable`, and then call `deserialize()` to read from the output data that we just wrote. Then we check that its value, retrieved using the `get()` method, is the original value, 163:

```
IntWritable newWritable = new IntWritable();
deserialize(newWritable, bytes);
assertThat(newWritable.get(), is(163));
```

WritableComparable and comparators

`IntWritable` implements the `WritableComparable` interface, which is just a subinterface of the `Writable` and `java.lang.Comparable` interfaces:

```
package org.apache.hadoop.io;

public interface WritableComparable<T> extends Writable, Comparable<T> {
}
```

Comparison of types is crucial for MapReduce, where there is a sorting phase during which keys are compared with one another. One optimization that Hadoop provides is the `RawComparator` extension of Java's `Comparator`:

```
package org.apache.hadoop.io;

import java.util.Comparator;

public interface RawComparator<T> extends Comparator<T> {

  public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2);

}
```

This interface permits implementors to compare records read from a stream without deserializing them into objects, thereby avoiding any overhead of object creation. For example, the comparator for `IntWritables` implements the raw `compare()` method by reading an integer from each of the byte arrays `b1` and `b2` and comparing them directly from the given start positions (`s1` and `s2`) and lengths (`l1` and `l2`).

`WritableComparator` is a general-purpose implementation of `RawComparator` for `WritableComparable` classes. It provides two main functions. First, it provides a default implementation of the raw `compare()` method that deserializes the objects to be compared from the stream and invokes the object `compare()` method. Second, it acts as a factory for `RawComparator` instances (that `Writable` implementations have registered). For example, to obtain a comparator for `IntWritable`, we just use:

```
RawComparator<IntWritable> comparator =
    WritableComparator.get(IntWritable.class);
```

The comparator can be used to compare two `IntWritable` objects:

```
IntWritable w1 = new IntWritable(163);
IntWritable w2 = new IntWritable(67);
assertThat(comparator.compare(w1, w2), greaterThan(0));
```

or their serialized representations:

```
byte[] b1 = serialize(w1);
byte[] b2 = serialize(w2);
assertThat(comparator.compare(b1, 0, b1.length, b2, 0, b2.length),
    greaterThan(0));
```

Writable Classes

Hadoop comes with a large selection of `Writable` classes, which are available in the `org.apache.hadoop.io` package. They form the class hierarchy shown in Figure 5-1.

Writable wrappers for Java primitives

There are `Writable` wrappers for all the Java primitive types (see Table 5-7) except `char` (which can be stored in an `IntWritable`). All have a `get()` and `set()` method for retrieving and storing the wrapped value.

Table 5-7. Writable wrapper classes for Java primitives

Java primitive	Writable implementation	Serialized size (bytes)
boolean	BooleanWritable	1
byte	ByteWritable	1
short	ShortWritable	2
int	IntWritable	4
	VIntWritable	1–5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1–9
double	DoubleWritable	8

When it comes to encoding integers, there is a choice between the fixed-length formats (`IntWritable` and `LongWritable`) and the variable-length formats (`VIntWritable` and `VLongWritable`). The variable-length formats use only a single byte to encode the value if it is small enough (between –112 and 127, inclusive); otherwise, they use the first byte to indicate whether the value is positive or negative, and how many bytes follow. For example, 163 requires two bytes:

```
byte[] data = serialize(new VIntWritable(163));
assertThat(StringUtils.byteToHexString(data), is("8fa3"));
```

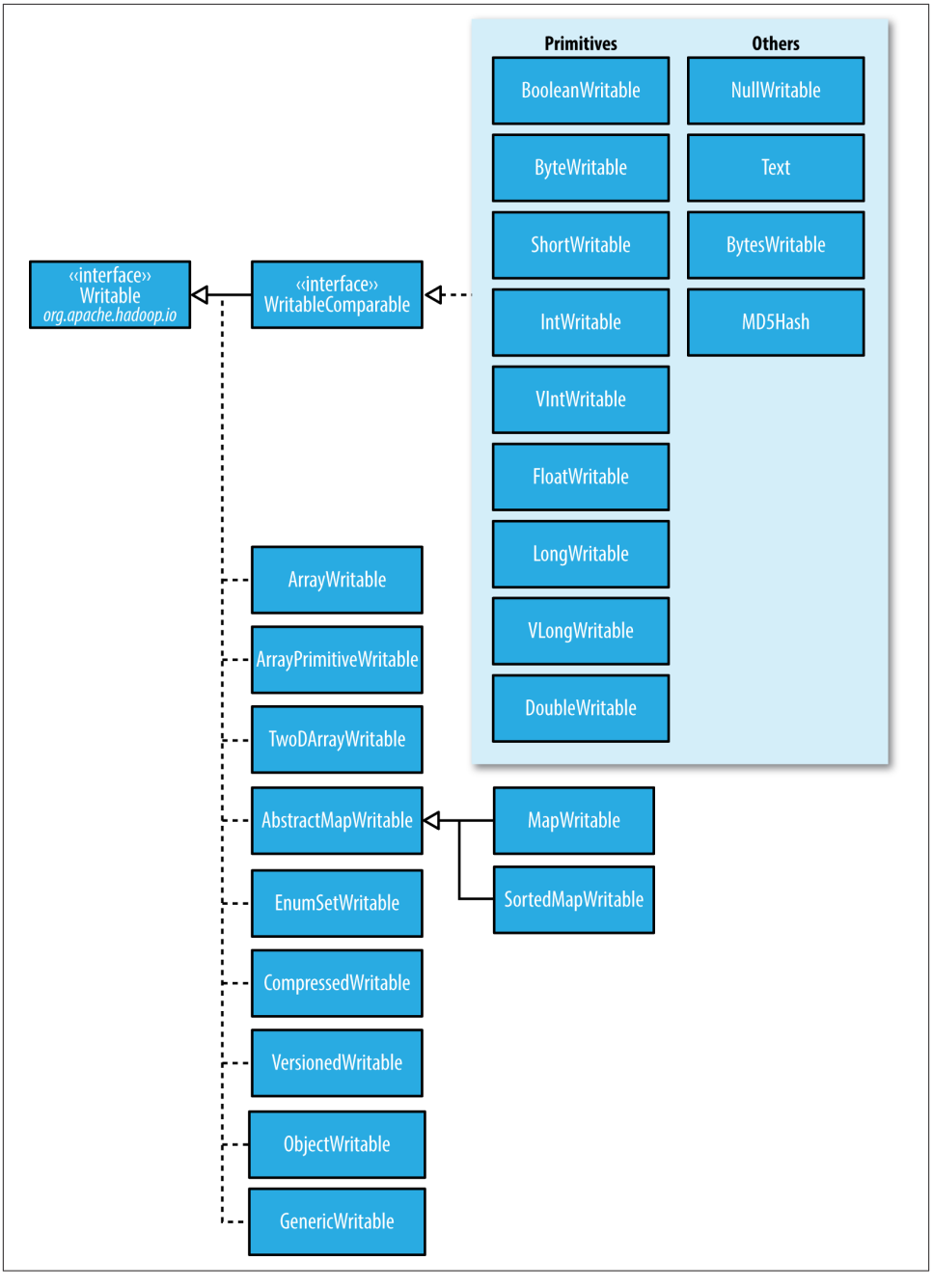

Figure 5-1. Writable class hierarchy

How do you choose between a fixed-length and a variable-length encoding? Fixed-length encodings are good when the distribution of values is fairly uniform across the whole value space, such as when using a (well-designed) hash function. Most numeric variables tend to have nonuniform distributions, though, and on average, the variable-length encoding will save space. Another advantage of variable-length encodings is that you can switch from `VIntWritable` to `VLongWritable`, because their encodings are actually the same. So, by choosing a variable-length representation, you have room to grow without committing to an 8-byte `long` representation from the beginning.

Text

`Text` is a Writable for UTF-8 sequences. It can be thought of as the Writable equivalent of `java.lang.String`.

The `Text` class uses an `int` (with a variable-length encoding) to store the number of bytes in the string encoding, so the maximum value is 2 GB. Furthermore, `Text` uses standard UTF-8, which makes it potentially easier to interoperate with other tools that understand UTF-8.

Indexing. Because of its emphasis on using standard UTF-8, there are some differences between `Text` and the Java `String` class. Indexing for the `Text` class is in terms of position in the encoded byte sequence, not the Unicode character in the string or the Java `char` code unit (as it is for `String`). For ASCII strings, these three concepts of index position coincide. Here is an example to demonstrate the use of the `charAt()` method:

```
Text t = new Text("hadoop");
assertThat(t.getLength(), is(6));
assertThat(t.getBytes().length, is(6));

assertThat(t.charAt(2), is((int) 'd'));
assertThat("Out of bounds", t.charAt(100), is(-1));
```

Notice that `charAt()` returns an `int` representing a Unicode code point, unlike the `String` variant that returns a `char`. `Text` also has a `find()` method, which is analogous to `String`'s `indexOf()`:

```
Text t = new Text("hadoop");
assertThat("Find a substring", t.find("do"), is(2));
assertThat("Finds first 'o'", t.find("o"), is(3));
assertThat("Finds 'o' from position 4 or later", t.find("o", 4), is(4));
assertThat("No match", t.find("pig"), is(-1));
```

Unicode. When we start using characters that are encoded with more than a single byte, the differences between `Text` and `String` become clear. Consider the Unicode characters shown in Table 5-8.[2]

Table 5-8. Unicode characters

Unicode code point	U+0041	U+00DF	U+6771	U+10400
Name	LATIN CAPITAL LETTER A	LATIN SMALL LETTER SHARP S	N/A (a unified Han ideograph)	DESERET CAPITAL LETTER LONG I
UTF-8 code units	`41`	`c3 9f`	`e6 9d b1`	`f0 90 90 80`
Java representation	`\u0041`	`\u00DF`	`\u6771`	`\uD801\uDC00`

All but the last character in the table, U+10400, can be expressed using a single Java `char`. U+10400 is a supplementary character and is represented by two Java `chars`, known as a *surrogate pair*. The tests in Example 5-5 show the differences between `String` and `Text` when processing a string of the four characters from Table 5-8.

Example 5-5. Tests showing the differences between the String and Text classes

```
public class StringTextComparisonTest {

  @Test
  public void string() throws UnsupportedEncodingException {

    String s = "\u0041\u00DF\u6771\uD801\uDC00";
    assertThat(s.length(), is(5));
    assertThat(s.getBytes("UTF-8").length, is(10));

    assertThat(s.indexOf("\u0041"), is(0));
    assertThat(s.indexOf("\u00DF"), is(1));
    assertThat(s.indexOf("\u6771"), is(2));
    assertThat(s.indexOf("\uD801\uDC00"), is(3));

    assertThat(s.charAt(0), is('\u0041'));
    assertThat(s.charAt(1), is('\u00DF'));
    assertThat(s.charAt(2), is('\u6771'));
    assertThat(s.charAt(3), is('\uD801'));
    assertThat(s.charAt(4), is('\uDC00'));

    assertThat(s.codePointAt(0), is(0x0041));
    assertThat(s.codePointAt(1), is(0x00DF));
    assertThat(s.codePointAt(2), is(0x6771));
    assertThat(s.codePointAt(3), is(0x10400));
  }
```

2. This example is based on one from Norbert Lindenberg and Masayoshi Okutsu's "Supplementary Characters in the Java Platform," May 2004.

```
  @Test
  public void text() {

    Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00");
    assertThat(t.getLength(), is(10));

    assertThat(t.find("\u0041"), is(0));
    assertThat(t.find("\u00DF"), is(1));
    assertThat(t.find("\u6771"), is(3));
    assertThat(t.find("\uD801\uDC00"), is(6));

    assertThat(t.charAt(0), is(0x0041));
    assertThat(t.charAt(1), is(0x00DF));
    assertThat(t.charAt(3), is(0x6771));
    assertThat(t.charAt(6), is(0x10400));
  }
}
```

The test confirms that the length of a `String` is the number of `char` code units it contains (five, made up of one from each of the first three characters in the string and a surrogate pair from the last), whereas the length of a `Text` object is the number of bytes in its UTF-8 encoding (10 = 1+2+3+4). Similarly, the `indexOf()` method in `String` returns an index in `char` code units, and `find()` for `Text` returns a byte offset.

The `charAt()` method in `String` returns the `char` code unit for the given index, which in the case of a surrogate pair will not represent a whole Unicode character. The `codePointAt()` method, indexed by `char` code unit, is needed to retrieve a single Unicode character represented as an `int`. In fact, the `charAt()` method in `Text` is more like the `codePointAt()` method than its namesake in `String`. The only difference is that it is indexed by byte offset.

Iteration. Iterating over the Unicode characters in `Text` is complicated by the use of byte offsets for indexing, since you can't just increment the index. The idiom for iteration is a little obscure (see Example 5-6): turn the `Text` object into a `java.nio.ByteBuffer`, then repeatedly call the `bytesToCodePoint()` static method on `Text` with the buffer. This method extracts the next code point as an `int` and updates the position in the buffer. The end of the string is detected when `bytesToCodePoint()` returns –1.

Example 5-6. Iterating over the characters in a Text object

```
public class TextIterator {

  public static void main(String[] args) {
    Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00");

    ByteBuffer buf = ByteBuffer.wrap(t.getBytes(), 0, t.getLength());
    int cp;
    while (buf.hasRemaining() && (cp = Text.bytesToCodePoint(buf)) != -1) {
```

```
      System.out.println(Integer.toHexString(cp));
    }
  }
}
```

Running the program prints the code points for the four characters in the string:

```
% hadoop TextIterator
41
df
6771
10400
```

Mutability. Another difference from `String` is that `Text` is mutable (like all `Writable` implementations in Hadoop, except `NullWritable`, which is a singleton). You can reuse a `Text` instance by calling one of the `set()` methods on it. For example:

```
Text t = new Text("hadoop");
t.set("pig");
assertThat(t.getLength(), is(3));
assertThat(t.getBytes().length, is(3));
```

In some situations, the byte array returned by the `getBytes()` method may be longer than the length returned by `getLength()`:

```
Text t = new Text("hadoop");
t.set(new Text("pig"));
assertThat(t.getLength(), is(3));
assertThat("Byte length not shortened", t.getBytes().length,
    is(6));
```

This shows why it is imperative that you always call `getLength()` when calling `getBytes()`, so you know how much of the byte array is valid data.

Resorting to String. `Text` doesn't have as rich an API for manipulating strings as `java.lang.String`, so in many cases, you need to convert the `Text` object to a `String`. This is done in the usual way, using the `toString()` method:

```
assertThat(new Text("hadoop").toString(), is("hadoop"));
```

BytesWritable

`BytesWritable` is a wrapper for an array of binary data. Its serialized format is a 4-byte integer field that specifies the number of bytes to follow, followed by the bytes themselves. For example, the byte array of length 2 with values 3 and 5 is serialized as a 4-byte integer (`00000002`) followed by the two bytes from the array (`03` and `05`):

```
BytesWritable b = new BytesWritable(new byte[] { 3, 5 });
byte[] bytes = serialize(b);
assertThat(StringUtils.byteToHexString(bytes), is("000000020305"));
```

`BytesWritable` is mutable, and its value may be changed by calling its `set()` method. As with `Text`, the size of the byte array returned from the `getBytes()` method for `BytesWritable`—the capacity—may not reflect the actual size of the data stored in the `BytesWritable`. You can determine the size of the `BytesWritable` by calling `getLength()`. To demonstrate:

```
b.setCapacity(11);
assertThat(b.getLength(), is(2));
assertThat(b.getBytes().length, is(11));
```

NullWritable

`NullWritable` is a special type of Writable, as it has a zero-length serialization. No bytes are written to or read from the stream. It is used as a placeholder; for example, in MapReduce, a key or a value can be declared as a `NullWritable` when you don't need to use that position, effectively storing a constant empty value. `NullWritable` can also be useful as a key in a `SequenceFile` when you want to store a list of values, as opposed to key-value pairs. It is an immutable singleton, and the instance can be retrieved by calling `NullWritable.get()`.

ObjectWritable and GenericWritable

`ObjectWritable` is a general-purpose wrapper for the following: Java primitives, `String`, `enum`, `Writable`, `null`, or arrays of any of these types. It is used in Hadoop RPC to marshal and unmarshal method arguments and return types.

`ObjectWritable` is useful when a field can be of more than one type. For example, if the values in a `SequenceFile` have multiple types, you can declare the value type as an `ObjectWritable` and wrap each type in an `ObjectWritable`. Being a general-purpose mechanism, it wastes a fair amount of space because it writes the classname of the wrapped type every time it is serialized. In cases where the number of types is small and known ahead of time, this can be improved by having a static array of types and using the index into the array as the serialized reference to the type. This is the approach that `GenericWritable` takes, and you have to subclass it to specify which types to support.

Writable collections

The `org.apache.hadoop.io` package includes six `Writable` collection types: `ArrayWritable`, `ArrayPrimitiveWritable`, `TwoDArrayWritable`, `MapWritable`, `SortedMapWritable`, and `EnumSetWritable`.

`ArrayWritable` and `TwoDArrayWritable` are `Writable` implementations for arrays and two-dimensional arrays (array of arrays) of `Writable` instances. All the elements of an

`ArrayWritable` or a `TwoDArrayWritable` must be instances of the same class, which is specified at construction as follows:

```
ArrayWritable writable = new ArrayWritable(Text.class);
```

In contexts where the `Writable` is defined by type, such as in `SequenceFile` keys or values or as input to MapReduce in general, you need to subclass `ArrayWritable` (or `TwoDArrayWritable`, as appropriate) to set the type statically. For example:

```
public class TextArrayWritable extends ArrayWritable {
  public TextArrayWritable() {
    super(Text.class);
  }
}
```

`ArrayWritable` and `TwoDArrayWritable` both have `get()` and `set()` methods, as well as a `toArray()` method, which creates a shallow copy of the array (or 2D array).

`ArrayPrimitiveWritable` is a wrapper for arrays of Java primitives. The component type is detected when you call `set()`, so there is no need to subclass to set the type.

`MapWritable` is an implementation of `java.util.Map<Writable, Writable>`, and `SortedMapWritable` is an implementation of `java.util.SortedMap<WritableComparable, Writable>`. The type of each key and value field is a part of the serialization format for that field. The type is stored as a single byte that acts as an index into an array of types. The array is populated with the standard types in the `org.apache.hadoop.io` package, but custom `Writable` types are accommodated, too, by writing a header that encodes the type array for nonstandard types. As they are implemented, `MapWritable` and `SortedMapWritable` use positive `byte` values for custom types, so a maximum of 127 distinct nonstandard `Writable` classes can be used in any particular `MapWritable` or `SortedMapWritable` instance. Here's a demonstration of using a `MapWritable` with different types for keys and values:

```
MapWritable src = new MapWritable();
src.put(new IntWritable(1), new Text("cat"));
src.put(new VIntWritable(2), new LongWritable(163));

MapWritable dest = new MapWritable();
WritableUtils.cloneInto(dest, src);
assertThat((Text) dest.get(new IntWritable(1)), is(new Text("cat")));
assertThat((LongWritable) dest.get(new VIntWritable(2)),
    is(new LongWritable(163)));
```

Conspicuous by their absence are `Writable` collection implementations for sets and lists. A general set can be emulated by using a `MapWritable` (or a `SortedMapWritable` for a sorted set) with `NullWritable` values. There is also `EnumSetWritable` for sets of enum types. For lists of a single type of `Writable`, `ArrayWritable` is adequate, but to store different types of `Writable` in a single list, you can use `GenericWritable` to wrap

the elements in an `ArrayWritable`. Alternatively, you could write a general `ListWritable` using the ideas from `MapWritable`.

Implementing a Custom Writable

Hadoop comes with a useful set of `Writable` implementations that serve most purposes; however, on occasion, you may need to write your own custom implementation. With a custom `Writable`, you have full control over the binary representation and the sort order. Because `Writables` are at the heart of the MapReduce data path, tuning the binary representation can have a significant effect on performance. The stock `Writable` implementations that come with Hadoop are well tuned, but for more elaborate structures, it is often better to create a new `Writable` type rather than composing the stock types.

If you are considering writing a custom `Writable`, it may be worth trying another serialization framework, like Avro, that allows you to define custom types declaratively. See "Serialization Frameworks" on page 126 and Chapter 12.

To demonstrate how to create a custom `Writable`, we shall write an implementation that represents a pair of strings, called `TextPair`. The basic implementation is shown in Example 5-7.

Example 5-7. A Writable implementation that stores a pair of Text objects

```
import java.io.*;

import org.apache.hadoop.io.*;

public class TextPair implements WritableComparable<TextPair> {

  private Text first;
  private Text second;

  public TextPair() {
    set(new Text(), new Text());
  }

  public TextPair(String first, String second) {
    set(new Text(first), new Text(second));
  }

  public TextPair(Text first, Text second) {
    set(first, second);
  }

  public void set(Text first, Text second) {
```

```
    this.first = first;
    this.second = second;
  }

  public Text getFirst() {
    return first;
  }

  public Text getSecond() {
    return second;
  }

  @Override
  public void write(DataOutput out) throws IOException {
    first.write(out);
    second.write(out);
  }

  @Override
  public void readFields(DataInput in) throws IOException {
    first.readFields(in);
    second.readFields(in);
  }

  @Override
  public int hashCode() {
    return first.hashCode() * 163 + second.hashCode();
  }

  @Override
  public boolean equals(Object o) {
    if (o instanceof TextPair) {
      TextPair tp = (TextPair) o;
      return first.equals(tp.first) && second.equals(tp.second);
    }
    return false;
  }

  @Override
  public String toString() {
    return first + "\t" + second;
  }

  @Override
  public int compareTo(TextPair tp) {
    int cmp = first.compareTo(tp.first);
    if (cmp != 0) {
      return cmp;
    }
    return second.compareTo(tp.second);
  }
}
```

The first part of the implementation is straightforward: there are two `Text` instance variables, `first` and `second`, and associated constructors, getters, and setters. All `Writable` implementations must have a default constructor so that the MapReduce framework can instantiate them, then populate their fields by calling `readFields()`. `Writable` instances are mutable and often reused, so you should take care to avoid allocating objects in the `write()` or `readFields()` methods.

`TextPair`'s `write()` method serializes each `Text` object in turn to the output stream by delegating to the `Text` objects themselves. Similarly, `readFields()` deserializes the bytes from the input stream by delegating to each `Text` object. The `DataOutput` and `DataInput` interfaces have a rich set of methods for serializing and deserializing Java primitives, so, in general, you have complete control over the wire format of your `Writable` object.

Just as you would for any value object you write in Java, you should override the `hashCode()`, `equals()`, and `toString()` methods from `java.lang.Object`. The `hashCode()` method is used by the `HashPartitioner` (the default partitioner in MapReduce) to choose a reduce partition, so you should make sure that you write a good hash function that mixes well to ensure reduce partitions are of a similar size.

If you plan to use your custom `Writable` with `TextOutputFormat`, you must implement its `toString()` method. `TextOutputFormat` calls `toString()` on keys and values for their output representation. For `TextPair`, we write the underlying `Text` objects as strings separated by a tab character.

`TextPair` is an implementation of `WritableComparable`, so it provides an implementation of the `compareTo()` method that imposes the ordering you would expect: it sorts by the first string followed by the second. Notice that, apart from the number of `Text` objects it can store, `TextPair` differs from `TextArrayWritable` (which we discussed in the previous section), since `TextArrayWritable` is only a `Writable`, not a `WritableComparable`.

Implementing a RawComparator for speed

The code for `TextPair` in Example 5-7 will work as it stands; however, there is a further optimization we can make. As explained in "WritableComparable and comparators" on page 112, when `TextPair` is being used as a key in MapReduce, it will have to be deserialized into an object for the `compareTo()` method to be invoked. What if it were possible to compare two `TextPair` objects just by looking at their serialized representations?

It turns out that we can do this because `TextPair` is the concatenation of two `Text` objects, and the binary representation of a `Text` object is a variable-length integer containing the number of bytes in the UTF-8 representation of the string, followed by the

UTF-8 bytes themselves. The trick is to read the initial length so we know how long the first `Text` object's byte representation is; then we can delegate to `Text`'s `RawComparator` and invoke it with the appropriate offsets for the first or second string. Example 5-8 gives the details (note that this code is nested in the `TextPair` class).

Example 5-8. A RawComparator for comparing TextPair byte representations

```
public static class Comparator extends WritableComparator {

  private static final Text.Comparator TEXT_COMPARATOR = new Text.Comparator();

  public Comparator() {
    super(TextPair.class);
  }

  @Override
  public int compare(byte[] b1, int s1, int l1,
                     byte[] b2, int s2, int l2) {

    try {
      int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) + readVInt(b1, s1);
      int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) + readVInt(b2, s2);
      int cmp = TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2, firstL2);
      if (cmp != 0) {
        return cmp;
      }
      return TEXT_COMPARATOR.compare(b1, s1 + firstL1, l1 - firstL1,
                                     b2, s2 + firstL2, l2 - firstL2);
    } catch (IOException e) {
      throw new IllegalArgumentException(e);
    }
  }
}

static {
  WritableComparator.define(TextPair.class, new Comparator());
}
```

We actually subclass `WritableComparator` rather than implementing `RawComparator` directly, since it provides some convenience methods and default implementations. The subtle part of this code is calculating `firstL1` and `firstL2`, the lengths of the first `Text` field in each byte stream. Each is made up of the length of the variable-length integer (returned by `decodeVIntSize()` on `WritableUtils`) and the value it is encoding (returned by `readVInt()`).

The static block registers the raw comparator so that whenever MapReduce sees the `TextPair` class, it knows to use the raw comparator as its default comparator.

Custom comparators

As you can see with `TextPair`, writing raw comparators takes some care because you have to deal with details at the byte level. It is worth looking at some of the implementations of `Writable` in the `org.apache.hadoop.io` package for further ideas if you need to write your own. The utility methods on `WritableUtils` are very handy, too.

Custom comparators should also be written to be `RawComparators`, if possible. These are comparators that implement a different sort order from the natural sort order defined by the default comparator. Example 5-9 shows a comparator for `TextPair`, called `FirstComparator`, that considers only the first string of the pair. Note that we override the `compare()` method that takes objects so both `compare()` methods have the same semantics.

We will make use of this comparator in Chapter 9, when we look at joins and secondary sorting in MapReduce (see "Joins" on page 268).

Example 5-9. A custom RawComparator for comparing the first field of TextPair byte representations

```
public static class FirstComparator extends WritableComparator {

  private static final Text.Comparator TEXT_COMPARATOR = new Text.Comparator();

  public FirstComparator() {
    super(TextPair.class);
  }

  @Override
  public int compare(byte[] b1, int s1, int l1,
                     byte[] b2, int s2, int l2) {

    try {
      int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) + readVInt(b1, s1);
      int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) + readVInt(b2, s2);
      return TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2, firstL2);
    } catch (IOException e) {
      throw new IllegalArgumentException(e);
    }
  }

  @Override
  public int compare(WritableComparable a, WritableComparable b) {
    if (a instanceof TextPair && b instanceof TextPair) {
      return ((TextPair) a).first.compareTo(((TextPair) b).first);
    }
    return super.compare(a, b);
  }
}
```

Serialization Frameworks

Although most MapReduce programs use `Writable` key and value types, this isn't mandated by the MapReduce API. In fact, any type can be used; the only requirement is a mechanism that translates to and from a binary representation of each type.

To support this, Hadoop has an API for pluggable serialization frameworks. A serialization framework is represented by an implementation of `Serialization` (in the `org.apache.hadoop.io.serializer` package). `WritableSerialization`, for example, is the implementation of `Serialization` for `Writable` types.

A `Serialization` defines a mapping from types to `Serializer` instances (for turning an object into a byte stream) and `Deserializer` instances (for turning a byte stream into an object).

Set the `io.serializations` property to a comma-separated list of classnames in order to register `Serialization` implementations. Its default value includes `org.apache.hadoop.io.serializer.WritableSerialization` and the Avro Specific and Reflect serializations (see "Avro Data Types and Schemas" on page 346), which means that only `Writable` or Avro objects can be serialized or deserialized out of the box.

Hadoop includes a class called `JavaSerialization` that uses Java Object Serialization. Although it makes it convenient to be able to use standard Java types such as `Integer` or `String` in MapReduce programs, Java Object Serialization is not as efficient as Writables, so it's not worth making this trade-off (see the following sidebar).

Why Not Use Java Object Serialization?

Java comes with its own serialization mechanism, called Java Object Serialization (often referred to simply as "Java Serialization"), that is tightly integrated with the language, so it's natural to ask why this wasn't used in Hadoop. Here's what Doug Cutting said in response to that question:

> Why didn't I use Serialization when we first started Hadoop? Because it looked big and hairy and I thought we needed something lean and mean, where we had precise control over exactly how objects are written and read, since that is central to Hadoop. With Serialization you can get some control, but you have to fight for it.
>
> The logic for not using RMI [Remote Method Invocation] was similar. Effective, high-performance inter-process communications are critical to Hadoop. I felt like we'd need to precisely control how things like connections, timeouts and buffers are handled, and RMI gives you little control over those.

The problem is that Java Serialization doesn't meet the criteria for a serialization format listed earlier: compact, fast, extensible, and interoperable.

Serialization IDL

There are a number of other serialization frameworks that approach the problem in a different way: rather than defining types through code, you define them in a language-neutral, declarative fashion, using an *interface description language* (IDL). The system can then generate types for different languages, which is good for interoperability. They also typically define versioning schemes that make type evolution straightforward.

Apache Thrift and Google Protocol Buffers are both popular serialization frameworks, and both are commonly used as a format for persistent binary data. There is limited support for these as MapReduce formats;[3] however, they are used internally in parts of Hadoop for RPC and data exchange.

Avro is an IDL-based serialization framework designed to work well with large-scale data processing in Hadoop. It is covered in Chapter 12.

File-Based Data Structures

For some applications, you need a specialized data structure to hold your data. For doing MapReduce-based processing, putting each blob of binary data into its own file doesn't scale, so Hadoop developed a number of higher-level containers for these situations.

SequenceFile

Imagine a logfile where each log record is a new line of text. If you want to log binary types, plain text isn't a suitable format. Hadoop's `SequenceFile` class fits the bill in this situation, providing a persistent data structure for binary key-value pairs. To use it as a logfile format, you would choose a key, such as timestamp represented by a `LongWritable`, and the value would be a `Writable` that represents the quantity being logged.

`SequenceFile`s also work well as containers for smaller files. HDFS and MapReduce are optimized for large files, so packing files into a `SequenceFile` makes storing and processing the smaller files more efficient ("Processing a whole file as a record" on page 228 contains a program to pack files into a `SequenceFile`).[4]

Writing a SequenceFile

To create a `SequenceFile`, use one of its `createWriter()` static methods, which return a `SequenceFile.Writer` instance. There are several overloaded versions, but they all require you to specify a stream to write to (either an `FSDataOutputStream` or a

3. Twitter's Elephant Bird project includes tools for working with Thrift and Protocol Buffers in Hadoop.
4. In a similar vein, the blog post "A Million Little Files" by Stuart Sierra includes code for converting a tar file into a `SequenceFile`.

FileSystem and Path pairing), a Configuration object, and the key and value types. Optional arguments include the compression type and codec, a Progressable callback to be informed of write progress, and a Metadata instance to be stored in the SequenceFile header.

The keys and values stored in a SequenceFile do not necessarily need to be Writables. Any types that can be serialized and deserialized by a Serialization may be used.

Once you have a SequenceFile.Writer, you then write key-value pairs using the append() method. When you've finished, you call the close() method (SequenceFile.Writer implements java.io.Closeable).

Example 5-10 shows a short program to write some key-value pairs to a SequenceFile using the API just described.

Example 5-10. Writing a SequenceFile

```
public class SequenceFileWriteDemo {

  private static final String[] DATA = {
    "One, two, buckle my shoe",
    "Three, four, shut the door",
    "Five, six, pick up sticks",
    "Seven, eight, lay them straight",
    "Nine, ten, a big fat hen"
  };

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

    IntWritable key = new IntWritable();
    Text value = new Text();
    SequenceFile.Writer writer = null;
    try {
      writer = SequenceFile.createWriter(fs, conf, path,
          key.getClass(), value.getClass());

      for (int i = 0; i < 100; i++) {
        key.set(100 - i);
        value.set(DATA[i % DATA.length]);
        System.out.printf("[%s]\t%s\t%s\n", writer.getLength(), key, value);
        writer.append(key, value);
      }
    } finally {
      IOUtils.closeStream(writer);
    }
  }
}
```

The keys in the sequence file are integers counting down from 100 to 1, represented as `IntWritable` objects. The values are `Text` objects. Before each record is appended to the `SequenceFile.Writer`, we call the `getLength()` method to discover the current position in the file. (We will use this information about record boundaries in the next section, when we read the file nonsequentially.) We write the position out to the console, along with the key and value pairs. The result of running it is shown here:

```
% hadoop SequenceFileWriteDemo numbers.seq
[128]   100     One, two, buckle my shoe
[173]   99      Three, four, shut the door
[220]   98      Five, six, pick up sticks
[264]   97      Seven, eight, lay them straight
[314]   96      Nine, ten, a big fat hen
[359]   95      One, two, buckle my shoe
[404]   94      Three, four, shut the door
[451]   93      Five, six, pick up sticks
[495]   92      Seven, eight, lay them straight
[545]   91      Nine, ten, a big fat hen
...
[1976]  60      One, two, buckle my shoe
[2021]  59      Three, four, shut the door
[2088]  58      Five, six, pick up sticks
[2132]  57      Seven, eight, lay them straight
[2182]  56      Nine, ten, a big fat hen
...
[4557]  5       One, two, buckle my shoe
[4602]  4       Three, four, shut the door
[4649]  3       Five, six, pick up sticks
[4693]  2       Seven, eight, lay them straight
[4743]  1       Nine, ten, a big fat hen
```

Reading a SequenceFile

Reading sequence files from beginning to end is a matter of creating an instance of `SequenceFile.Reader` and iterating over records by repeatedly invoking one of the `next()` methods. Which one you use depends on the serialization framework you are using. If you are using `Writable` types, you can use the `next()` method that takes a key and a value argument and reads the next key and value in the stream into these variables:

```
public boolean next(Writable key, Writable val)
```

The return value is `true` if a key-value pair was read and `false` if the end of the file has been reached.

For other, non-Writable serialization frameworks (such as Apache Thrift), you should use these two methods:

```
public Object next(Object key) throws IOException
public Object getCurrentValue(Object val) throws IOException
```

In this case, you need to make sure that the serialization you want to use has been set in the `io.serializations` property; see "Serialization Frameworks" on page 126.

If the `next()` method returns a non-`null` object, a key-value pair was read from the stream, and the value can be retrieved using the `getCurrentValue()` method. Otherwise, if `next()` returns `null`, the end of the file has been reached.

The program in Example 5-11 demonstrates how to read a sequence file that has `Writable` keys and values. Note how the types are discovered from the `SequenceFile.Reader` via calls to `getKeyClass()` and `getValueClass()`, and then `ReflectionUtils` is used to create an instance for the key and an instance for the value. This technique allows the program to be used with any sequence file that has `Writable` keys and values.

Example 5-11. Reading a SequenceFile

```
public class SequenceFileReadDemo {

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

    SequenceFile.Reader reader = null;
    try {
      reader = new SequenceFile.Reader(fs, path, conf);
      Writable key = (Writable)
          ReflectionUtils.newInstance(reader.getKeyClass(), conf);
      Writable value = (Writable)
          ReflectionUtils.newInstance(reader.getValueClass(), conf);
      long position = reader.getPosition();
      while (reader.next(key, value)) {
        String syncSeen = reader.syncSeen() ? "*" : "";
        System.out.printf("[%s%s]\t%s\t%s\n", position, syncSeen, key, value);
        position = reader.getPosition(); // beginning of next record
      }
    } finally {
      IOUtils.closeStream(reader);
    }
  }
}
```

Another feature of the program is that it displays the positions of the *sync points* in the sequence file. A sync point is a point in the stream that can be used to resynchronize with a record boundary if the reader is "lost"—for example, after seeking to an arbitrary position in the stream. Sync points are recorded by `SequenceFile.Writer`, which inserts a special entry to mark the sync point every few records as a sequence file is being

written. Such entries are small enough to incur only a modest storage overhead—less than 1%. Sync points always align with record boundaries.

Running the program in Example 5-11 shows the sync points in the sequence file as asterisks. The first one occurs at position 2021 (the second one occurs at position 4075, but is not shown in the output):

```
% hadoop SequenceFileReadDemo numbers.seq
[128]   100     One, two, buckle my shoe
[173]   99      Three, four, shut the door
[220]   98      Five, six, pick up sticks
[264]   97      Seven, eight, lay them straight
[314]   96      Nine, ten, a big fat hen
[359]   95      One, two, buckle my shoe
[404]   94      Three, four, shut the door
[451]   93      Five, six, pick up sticks
[495]   92      Seven, eight, lay them straight
[545]   91      Nine, ten, a big fat hen
[590]   90      One, two, buckle my shoe
...
[1976]  60      One, two, buckle my shoe
[2021*] 59      Three, four, shut the door
[2088]  58      Five, six, pick up sticks
[2132]  57      Seven, eight, lay them straight
[2182]  56      Nine, ten, a big fat hen
...
[4557]  5       One, two, buckle my shoe
[4602]  4       Three, four, shut the door
[4649]  3       Five, six, pick up sticks
[4693]  2       Seven, eight, lay them straight
[4743]  1       Nine, ten, a big fat hen
```

There are two ways to seek to a given position in a sequence file. The first is the `seek()` method, which positions the reader at the given point in the file. For example, seeking to a record boundary works as expected:

```
reader.seek(359);
assertThat(reader.next(key, value), is(true));
assertThat(((IntWritable) key).get(), is(95));
```

But if the position in the file is not at a record boundary, the reader fails when the `next()` method is called:

```
reader.seek(360);
reader.next(key, value); // fails with IOException
```

The second way to find a record boundary makes use of sync points. The `sync(long position)` method on `SequenceFile.Reader` positions the reader at the next sync point after `position`. (If there are no sync points in the file after this position, then the reader will be positioned at the end of the file.) Thus, we can call `sync()` with any position in

the stream—not necessarily a record boundary—and the reader will reestablish itself at the next sync point so reading can continue:

```
reader.sync(360);
assertThat(reader.getPosition(), is(2021L));
assertThat(reader.next(key, value), is(true));
assertThat(((IntWritable) key).get(), is(59));
```

`SequenceFile.Writer` has a method called `sync()` for inserting a sync point at the current position in the stream. This is not to be confused with the `hsync()` method defined by the `Syncable` interface for synchronizing buffers to the underlying device (see "Coherency Model" on page 74).

Sync points come into their own when using sequence files as input to MapReduce, since they permit the files to be split and different portions to be processed independently by separate map tasks (see "SequenceFileInputFormat" on page 236).

Displaying a SequenceFile with the command-line interface

The `hadoop fs` command has a `-text` option to display sequence files in textual form. It looks at a file's magic number so that it can attempt to detect the type of the file and appropriately convert it to text. It can recognize gzipped files, sequence files, and Avro datafiles; otherwise, it assumes the input is plain text.

For sequence files, this command is really useful only if the keys and values have meaningful string representations (as defined by the `toString()` method). Also, if you have your own key or value classes, you will need to make sure they are on Hadoop's classpath.

Running it on the sequence file we created in the previous section gives the following output:

```
% hadoop fs -text numbers.seq | head
100     One, two, buckle my shoe
99      Three, four, shut the door
98      Five, six, pick up sticks
97      Seven, eight, lay them straight
96      Nine, ten, a big fat hen
95      One, two, buckle my shoe
94      Three, four, shut the door
93      Five, six, pick up sticks
92      Seven, eight, lay them straight
91      Nine, ten, a big fat hen
```

Sorting and merging SequenceFiles

The most powerful way of sorting (and merging) one or more sequence files is to use MapReduce. MapReduce is inherently parallel and will let you specify the number of

reducers to use, which determines the number of output partitions. For example, by specifying one reducer, you get a single output file. We can use the sort example that comes with Hadoop by specifying that the input and output are sequence files and by setting the key and value types:

```
% hadoop jar \
  $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar \
  sort -r 1 \
  -inFormat org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat \
  -outFormat org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat \
  -outKey org.apache.hadoop.io.IntWritable \
  -outValue org.apache.hadoop.io.Text \
  numbers.seq sorted
% hadoop fs -text sorted/part-r-00000 | head
1       Nine, ten, a big fat hen
2       Seven, eight, lay them straight
3       Five, six, pick up sticks
4       Three, four, shut the door
5       One, two, buckle my shoe
6       Nine, ten, a big fat hen
7       Seven, eight, lay them straight
8       Five, six, pick up sticks
9       Three, four, shut the door
10      One, two, buckle my shoe
```

Sorting is covered in more detail in "Sorting" on page 255.

An alternative to using MapReduce for sort/merge is the `SequenceFile.Sorter` class, which has a number of `sort()` and `merge()` methods. These functions predate MapReduce and are lower-level functions than MapReduce (for example, to get parallelism, you need to partition your data manually), so in general MapReduce is the preferred approach to sort and merge sequence files.

The SequenceFile format

A sequence file consists of a header followed by one or more records (see Figure 5-2). The first three bytes of a sequence file are the bytes `SEQ`, which act as a magic number; these are followed by a single byte representing the version number. The header contains other fields, including the names of the key and value classes, compression details, user-defined metadata, and the sync marker.[5] Recall that the sync marker is used to allow a reader to synchronize to a record boundary from any position in the file. Each file has a randomly generated sync marker, whose value is stored in the header. Sync markers appear between records in the sequence file. They are designed to incur less than a 1% storage overhead, so they don't necessarily appear between every pair of records (such is the case for short records).

5. Full details of the format of these fields may be found in `SequenceFile`'s documentation and source code.

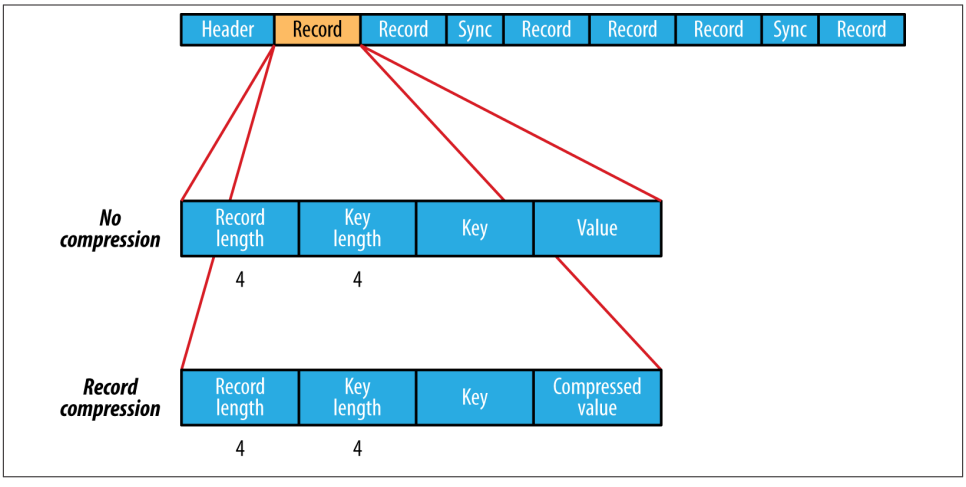

Figure 5-2. The internal structure of a sequence file with no compression and with record compression

The internal format of the records depends on whether compression is enabled, and if it is, whether it is record compression or block compression.

If no compression is enabled (the default), each record is made up of the record length (in bytes), the key length, the key, and then the value. The length fields are written as 4-byte integers adhering to the contract of the `writeInt()` method of `java.io.DataOutput`. Keys and values are serialized using the `Serialization` defined for the class being written to the sequence file.

The format for record compression is almost identical to that for no compression, except the value bytes are compressed using the codec defined in the header. Note that keys are not compressed.

Block compression (Figure 5-3) compresses multiple records at once; it is therefore more compact than and should generally be preferred over record compression because it has the opportunity to take advantage of similarities between records. Records are added to a block until it reaches a minimum size in bytes, defined by the `io.seqfile.compress.blocksize` property; the default is one million bytes. A sync marker is written before the start of every block. The format of a block is a field indicating the number of records in the block, followed by four compressed fields: the key lengths, the keys, the value lengths, and the values.

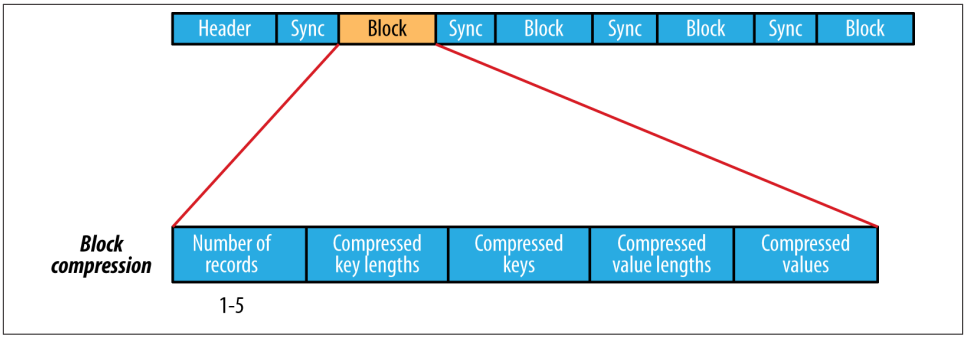

Figure 5-3. The internal structure of a sequence file with block compression

MapFile

A `MapFile` is a sorted `SequenceFile` with an index to permit lookups by key. The index is itself a `SequenceFile` that contains a fraction of the keys in the map (every 128th key, by default). The idea is that the index can be loaded into memory to provide fast lookups from the main data file, which is another `SequenceFile` containing all the map entries in sorted key order.

`MapFile` offers a very similar interface to `SequenceFile` for reading and writing—the main thing to be aware of is that when writing using `MapFile.Writer`, map entries must be added in order, otherwise an `IOException` will be thrown.

MapFile variants

Hadoop comes with a few variants on the general key-value `MapFile` interface:

- `SetFile` is a specialization of `MapFile` for storing a set of `Writable` keys. The keys must be added in sorted order.
- `ArrayFile` is a `MapFile` where the key is an integer representing the index of the element in the array and the value is a `Writable` value.
- `BloomMapFile` is a `MapFile` that offers a fast version of the `get()` method, especially for sparsely populated files. The implementation uses a dynamic Bloom filter for testing whether a given key is in the map. The test is very fast because it is in-memory, and it has a nonzero probability of false positives. Only if the test passes (the key is present) is the regular `get()` method called.

Other File Formats and Column-Oriented Formats

While sequence files and map files are the oldest binary file formats in Hadoop, they are not the only ones, and in fact there are better alternatives that should be considered for new projects.

Avro datafiles (covered in "Avro Datafiles" on page 352) are like sequence files in that they are designed for large-scale data processing—they are compact and splittable—but they are portable across different programming languages. Objects stored in Avro datafiles are described by a schema, rather than in the Java code of the implementation of a `Writable` object (as is the case for sequence files), making them very Java-centric. Avro datafiles are widely supported across components in the Hadoop ecosystem, so they are a good default choice for a binary format.

Sequence files, map files, and Avro datafiles are all row-oriented file formats, which means that the values for each row are stored contiguously in the file. In a column-oriented format, the rows in a file (or, equivalently, a table in Hive) are broken up into row splits, then each split is stored in column-oriented fashion: the values for each row in the first column are stored first, followed by the values for each row in the second column, and so on. This is shown diagrammatically in Figure 5-4.

A column-oriented layout permits columns that are not accessed in a query to be skipped. Consider a query of the table in Figure 5-4 that processes only column 2. With row-oriented storage, like a sequence file, the whole row (stored in a sequence file record) is loaded into memory, even though only the second column is actually read. Lazy deserialization saves some processing cycles by deserializing only the column fields that are accessed, but it can't avoid the cost of reading each row's bytes from disk.

With column-oriented storage, only the column 2 parts of the file (highlighted in the figure) need to be read into memory. In general, column-oriented formats work well when queries access only a small number of columns in the table. Conversely, row-oriented formats are appropriate when a large number of columns of a single row are needed for processing at the same time.

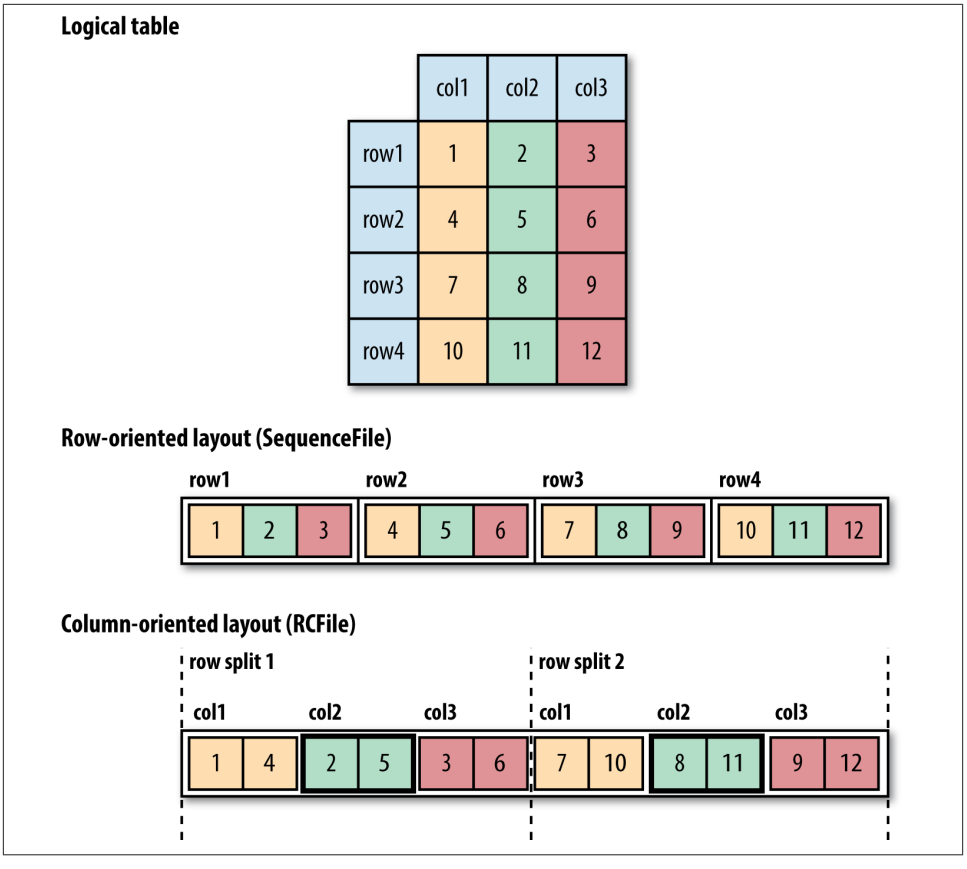

Figure 5-4. Row-oriented versus column-oriented storage

Column-oriented formats need more memory for reading and writing, since they have to buffer a row split in memory, rather than just a single row. Also, it's not usually possible to control when writes occur (via flush or sync operations), so column-oriented formats are not suited to streaming writes, as the current file cannot be recovered if the writer process fails. On the other hand, row-oriented formats like sequence files and Avro datafiles can be read up to the last sync point after a writer failure. It is for this reason that Flume (see Chapter 14) uses row-oriented formats.

The first column-oriented file format in Hadoop was Hive's *RCFile*, short for *Record Columnar File*. It has since been superseded by Hive's *ORCFile* (*Optimized Record Columnar File*), and *Parquet* (covered in Chapter 13). Parquet is a general-purpose column-oriented file format based on Google's Dremel, and has wide support across Hadoop components. Avro also has a column-oriented format called *Trevni*.