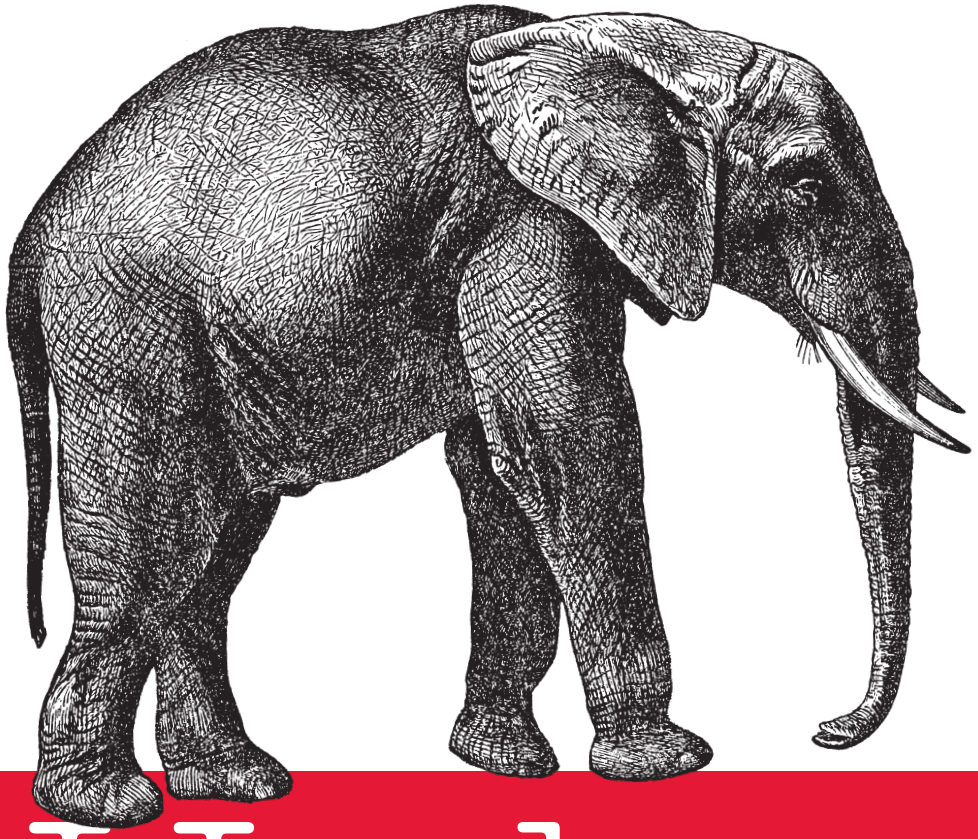
O'REILLY®

4th Edition
Revised & Updated



Hadoop

The Definitive Guide

STORAGE AND ANALYSIS AT INTERNET SCALE

Tom White

O'REILLY®

Hadoop: The Definitive Guide

Get ready to unlock the power of your data. With the fourth edition of this comprehensive guide, you'll learn how to build and maintain reliable, scalable, distributed systems with Apache Hadoop. This book is ideal for programmers looking to analyze datasets of any size, and for administrators who want to set up and run Hadoop clusters.

Using Hadoop 2 exclusively, author Tom White presents new chapters on YARN and several Hadoop-related projects such as Parquet, Flume, Crunch, and Spark. You'll learn about recent changes to Hadoop, and explore new case studies on Hadoop's role in healthcare systems and genomics data processing.

- Learn fundamental components such as MapReduce, HDFS, and YARN
- Explore MapReduce in depth, including steps for developing applications with it
- Set up and maintain a Hadoop cluster running HDFS and MapReduce on YARN
- Learn two data formats: Avro for data serialization and Parquet for nested data
- Use data ingestion tools such as Flume (for streaming data) and Sqoop (for bulk data transfer)
- Understand how high-level data processing tools like Pig, Hive, Crunch, and Spark work with Hadoop
- Learn the HBase distributed database and the ZooKeeper distributed configuration service

"Now you have the opportunity to learn about Hadoop from a master—not only of the technology, but also of common sense and plain talk."

—**Doug Cutting**
Cloudera

Tom White, an engineer at Cloudera and member of the Apache Software Foundation, has been an Apache Hadoop committer since 2007. He has written numerous articles for *oreilly.com*, *java.net*, and IBM's developerWorks, and speaks regularly about Hadoop at industry conferences.

PROGRAMMING LANGUAGES/HADOOP

US $49.99 **CAN $57.99**

ISBN: 978-1-491-90163-2

Twitter: @oreillymedia
facebook.com/oreilly

FOURTH EDITION

Hadoop: The Definitive Guide

Tom White

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Hadoop: The Definitive Guide, Fourth Edition

by Tom White

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Meghan Blanchette
Production Editor: Matthew Hacker
Copyeditor: Jasmine Kwityn
Proofreader: Rachel Head
Indexer: Lucie Haskins
Cover Designer: Ellie Volckhausen
Interior Designer: David Futato
Illustrator: Rebecca Demarest

June 2009: First Edition

October 2010: Second Edition

May 2012: Third Edition

April 2015: Fourth Edition

Revision History for the Fourth Edition:

2015-03-19: First release

2015-04-17: Second release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491901632* for release details.

ISBN: 978-1-491-90163-2

[LSI]

For Eliane, Emilia, and Lottie

Table of Contents

Part V. Case Studies

Foreword

Hadoop got its start in Nutch. A few of us were attempting to build an open source web search engine and having trouble managing computations running on even a handful of computers. Once Google published its GFS and MapReduce papers, the route became clear. They'd devised systems to solve precisely the problems we were having with Nutch. So we started, two of us, half-time, to try to re-create these systems as a part of Nutch.

We managed to get Nutch limping along on 20 machines, but it soon became clear that to handle the Web's massive scale, we'd need to run it on thousands of machines, and moreover, that the job was bigger than two half-time developers could handle.

Around that time, Yahoo! got interested, and quickly put together a team that I joined. We split off the distributed computing part of Nutch, naming it Hadoop. With the help of Yahoo!, Hadoop soon grew into a technology that could truly scale to the Web.

In 2006, Tom White started contributing to Hadoop. I already knew Tom through an excellent article he'd written about Nutch, so I knew he could present complex ideas in clear prose. I soon learned that he could also develop software that was as pleasant to read as his prose.

From the beginning, Tom's contributions to Hadoop showed his concern for users and for the project. Unlike most open source contributors, Tom is not primarily interested in tweaking the system to better meet his own needs, but rather in making it easier for anyone to use.

Initially, Tom specialized in making Hadoop run well on Amazon's EC2 and S3 services. Then he moved on to tackle a wide variety of problems, including improving the MapReduce APIs, enhancing the website, and devising an object serialization framework. In all cases, Tom presented his ideas precisely. In short order, Tom earned the role of Hadoop committer and soon thereafter became a member of the Hadoop Project Management Committee.

Tom is now a respected senior member of the Hadoop developer community. Though he's an expert in many technical corners of the project, his specialty is making Hadoop easier to use and understand.

Given this, I was very pleased when I learned that Tom intended to write a book about Hadoop. Who could be better qualified? Now you have the opportunity to learn about Hadoop from a master—not only of the technology, but also of common sense and plain talk.

—Doug Cutting, April 2009
Shed in the Yard, California

Preface

Martin Gardner, the mathematics and science writer, once said in an interview:

> Beyond calculus, I am lost. That was the secret of my column's success. It took me so long to understand what I was writing about that I knew how to write in a way most readers would understand.[1]

In many ways, this is how I feel about Hadoop. Its inner workings are complex, resting as they do on a mixture of distributed systems theory, practical engineering, and common sense. And to the uninitiated, Hadoop can appear alien.

But it doesn't need to be like this. Stripped to its core, the tools that Hadoop provides for working with big data are simple. If there's a common theme, it is about raising the level of abstraction—to create building blocks for programmers who have lots of data to store and analyze, and who don't have the time, the skill, or the inclination to become distributed systems experts to build the infrastructure to handle it.

With such a simple and generally applicable feature set, it seemed obvious to me when I started using it that Hadoop deserved to be widely used. However, at the time (in early 2006), setting up, configuring, and writing programs to use Hadoop was an art. Things have certainly improved since then: there is more documentation, there are more examples, and there are thriving mailing lists to go to when you have questions. And yet the biggest hurdle for newcomers is understanding what this technology is capable of, where it excels, and how to use it. That is why I wrote this book.

The Apache Hadoop community has come a long way. Since the publication of the first edition of this book, the Hadoop project has blossomed. "Big data" has become a household term.[2] In this time, the software has made great leaps in adoption, performance, reliability, scalability, and manageability. The number of things being built and run on the Hadoop platform has grown enormously. In fact, it's difficult for one person to keep

1. Alex Bellos, "The science of fun," *The Guardian*, May 31, 2008.
2. It was added to the *Oxford English Dictionary* in 2013.

track. To gain even wider adoption, I believe we need to make Hadoop even easier to use. This will involve writing more tools; integrating with even more systems; and writing new, improved APIs. I'm looking forward to being a part of this, and I hope this book will encourage and enable others to do so, too.

Administrative Notes

During discussion of a particular Java class in the text, I often omit its package name to reduce clutter. If you need to know which package a class is in, you can easily look it up in the Java API documentation for Hadoop (linked to from the Apache Hadoop home page), or the relevant project. Or if you're using an integrated development environment (IDE), its auto-complete mechanism can help find what you're looking for.

Similarly, although it deviates from usual style guidelines, program listings that import multiple classes from the same package may use the asterisk wildcard character to save space (for example, `import org.apache.hadoop.io.*`).

The sample programs in this book are available for download from the book's website. You will also find instructions there for obtaining the datasets that are used in examples throughout the book, as well as further notes for running the programs in the book and links to updates, additional resources, and my blog.

What's New in the Fourth Edition?

The fourth edition covers Hadoop 2 exclusively. The Hadoop 2 release series is the current active release series and contains the most stable versions of Hadoop.

There are new chapters covering YARN (Chapter 4), Parquet (Chapter 13), Flume (Chapter 14), Crunch (Chapter 18), and Spark (Chapter 19). There's also a new section to help readers navigate different pathways through the book ("What's in This Book?" on page 15).

This edition includes two new case studies (Chapters 22 and 23): one on how Hadoop is used in healthcare systems, and another on using Hadoop technologies for genomics data processing. Case studies from the previous editions can now be found online.

Many corrections, updates, and improvements have been made to existing chapters to bring them up to date with the latest releases of Hadoop and its related projects.

What's New in the Third Edition?

The third edition covers the 1.x (formerly 0.20) release series of Apache Hadoop, as well as the newer 0.22 and 2.x (formerly 0.23) series. With a few exceptions, which are noted in the text, all the examples in this book run against these versions.

This edition uses the new MapReduce API for most of the examples. Because the old API is still in widespread use, it continues to be discussed in the text alongside the new API, and the equivalent code using the old API can be found on the book's website.

The major change in Hadoop 2.0 is the new MapReduce runtime, MapReduce 2, which is built on a new distributed resource management system called YARN. This edition includes new sections covering MapReduce on YARN: how it works (Chapter 7) and how to run it (Chapter 10).

There is more MapReduce material, too, including development practices such as packaging MapReduce jobs with Maven, setting the user's Java classpath, and writing tests with MRUnit (all in Chapter 6). In addition, there is more depth on features such as output committers and the distributed cache (both in Chapter 9), as well as task memory monitoring (Chapter 10). There is a new section on writing MapReduce jobs to process Avro data (Chapter 12), and one on running a simple MapReduce workflow in Oozie (Chapter 6).

The chapter on HDFS (Chapter 3) now has introductions to high availability, federation, and the new WebHDFS and HttpFS filesystems.

The chapters on Pig, Hive, Sqoop, and ZooKeeper have all been expanded to cover the new features and changes in their latest releases.

In addition, numerous corrections and improvements have been made throughout the book.

What's New in the Second Edition?

The second edition has two new chapters on Sqoop and Hive (Chapters 15 and 17, respectively), a new section covering Avro (in Chapter 12), an introduction to the new security features in Hadoop (in Chapter 10), and a new case study on analyzing massive network graphs using Hadoop.

This edition continues to describe the 0.20 release series of Apache Hadoop, because this was the latest stable release at the time of writing. New features from later releases are occasionally mentioned in the text, however, with reference to the version that they were introduced in.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to commands and command-line options and to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a general note.

This icon signifies a tip or suggestion.

This icon indicates a warning or caution.

Using Code Examples

Supplemental material (code, examples, exercise, etc.) is available for download at this book's website and on GitHub.

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Hadoop: The Definitive Guide*, Fourth Edition, by Tom White (O'Reilly). Copyright 2015 Tom White, 978-1-491-90163-2."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/hadoop_tdg_4e*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I have relied on many people, both directly and indirectly, in writing this book. I would like to thank the Hadoop community, from whom I have learned, and continue to learn, a great deal.

In particular, I would like to thank Michael Stack and Jonathan Gray for writing the chapter on HBase. Thanks also go to Adrian Woodhead, Marc de Palol, Joydeep Sen Sarma, Ashish Thusoo, Andrzej Białecki, Stu Hood, Chris K. Wensel, and Owen O'Malley for contributing case studies.

I would like to thank the following reviewers who contributed many helpful suggestions and improvements to my drafts: Raghu Angadi, Matt Biddulph, Christophe Bisciglia, Ryan Cox, Devaraj Das, Alex Dorman, Chris Douglas, Alan Gates, Lars George, Patrick Hunt, Aaron Kimball, Peter Krey, Hairong Kuang, Simon Maxen, Olga Natkovich, Benjamin Reed, Konstantin Shvachko, Allen Wittenauer, Matei Zaharia, and Philip Zeyliger. Ajay Anand kept the review process flowing smoothly. Philip ("flip") Kromer kindly helped me with the NCDC weather dataset featured in the examples in this book. Special thanks to Owen O'Malley and Arun C. Murthy for explaining the intricacies of the MapReduce shuffle to me. Any errors that remain are, of course, to be laid at my door.

For the second edition, I owe a debt of gratitude for the detailed reviews and feedback from Jeff Bean, Doug Cutting, Glynn Durham, Alan Gates, Jeff Hammerbacher, Alex Kozlov, Ken Krugler, Jimmy Lin, Todd Lipcon, Sarah Sproehnle, Vinithra Varadharajan, and Ian Wrigley, as well as all the readers who submitted errata for the first edition. I would also like to thank Aaron Kimball for contributing the chapter on Sqoop, and Philip ("flip") Kromer for the case study on graph processing.

For the third edition, thanks go to Alejandro Abdelnur, Eva Andreasson, Eli Collins, Doug Cutting, Patrick Hunt, Aaron Kimball, Aaron T. Myers, Brock Noland, Arvind Prabhakar, Ahmed Radwan, and Tom Wheeler for their feedback and suggestions. Rob Weltman kindly gave very detailed feedback for the whole book, which greatly improved the final manuscript. Thanks also go to all the readers who submitted errata for the second edition.

For the fourth edition, I would like to thank Jodok Batlogg, Meghan Blanchette, Ryan Blue, Jarek Jarcec Cecho, Jules Damji, Dennis Dawson, Matthew Gast, Karthik Kambatla, Julien Le Dem, Brock Noland, Sandy Ryza, Akshai Sarma, Ben Spivey, Michael Stack, Kate Ting, Josh Walter, Josh Wills, and Adrian Woodhead for all of their invaluable review feedback. Ryan Brush, Micah Whitacre, and Matt Massie kindly contributed new case studies for this edition. Thanks again to all the readers who submitted errata.

I am particularly grateful to Doug Cutting for his encouragement, support, and friendship, and for contributing the Foreword.

Thanks also go to the many others with whom I have had conversations or email discussions over the course of writing the book.

Halfway through writing the first edition of this book, I joined Cloudera, and I want to thank my colleagues for being incredibly supportive in allowing me the time to write and to get it finished promptly.

I am grateful to my editors, Mike Loukides and Meghan Blanchette, and their colleagues at O'Reilly for their help in the preparation of this book. Mike and Meghan have been there throughout to answer my questions, to read my first drafts, and to keep me on schedule.

Finally, the writing of this book has been a great deal of work, and I couldn't have done it without the constant support of my family. My wife, Eliane, not only kept the home going, but also stepped in to help review, edit, and chase case studies. My daughters, Emilia and Lottie, have been very understanding, and I'm looking forward to spending lots more time with all of them.

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.

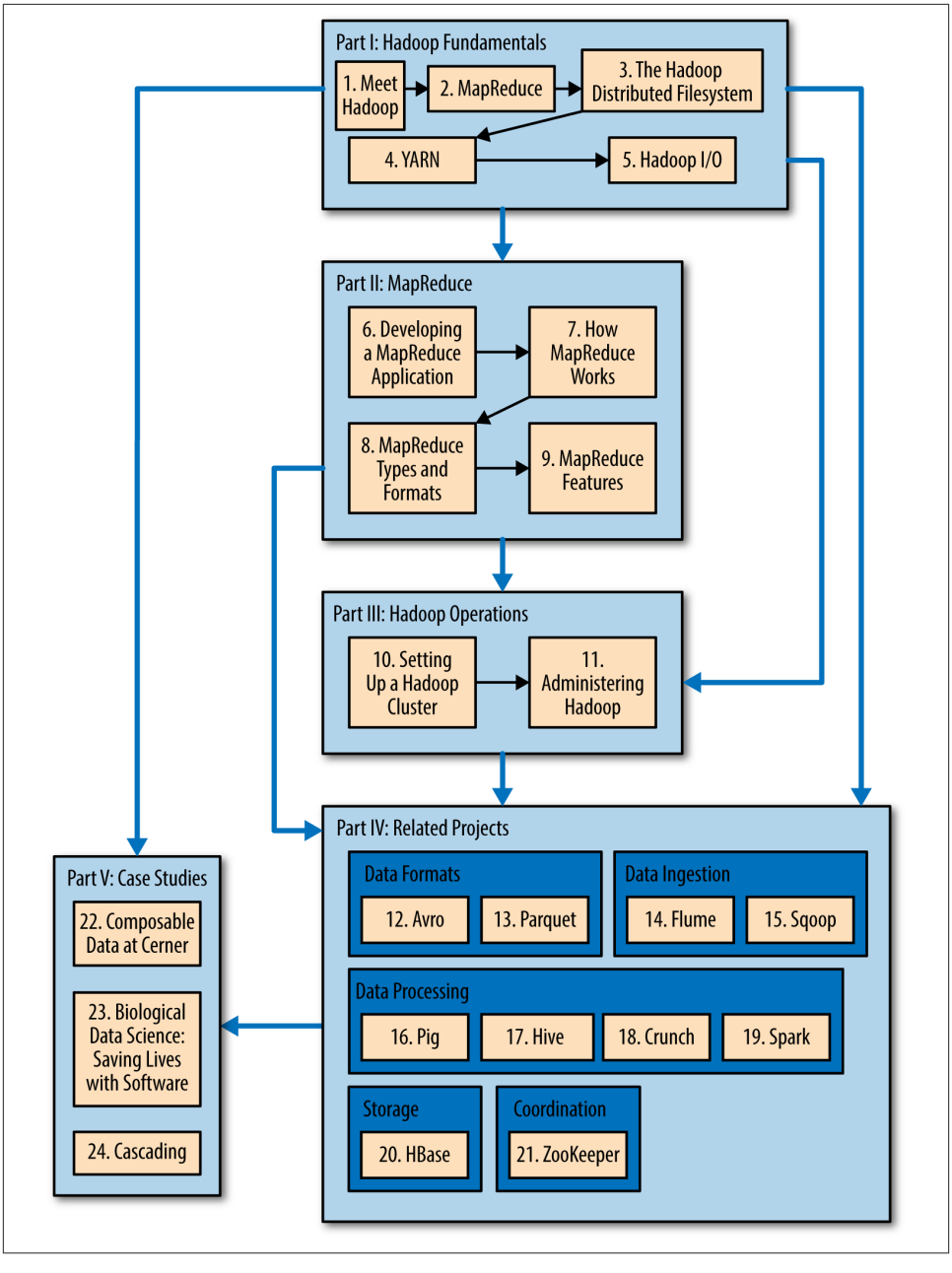


Figure 1-1. Structure of the book: there are various pathways through the content

CHAPTER 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.

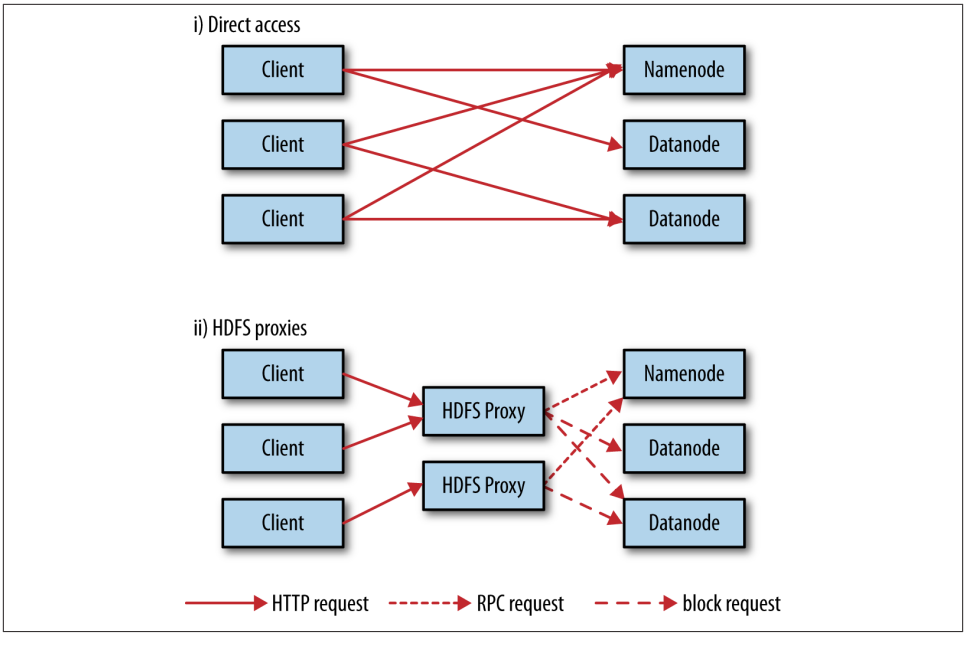


Figure 3-1. Accessing HDFS over HTTP directly and via a bank of HDFS proxies

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.

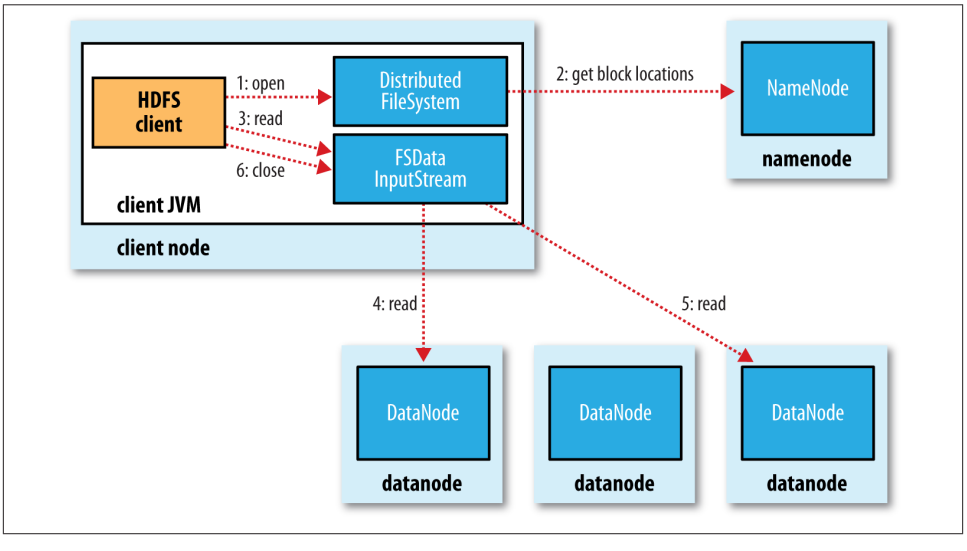


Figure 3-2. A client reading data from HDFS

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.)

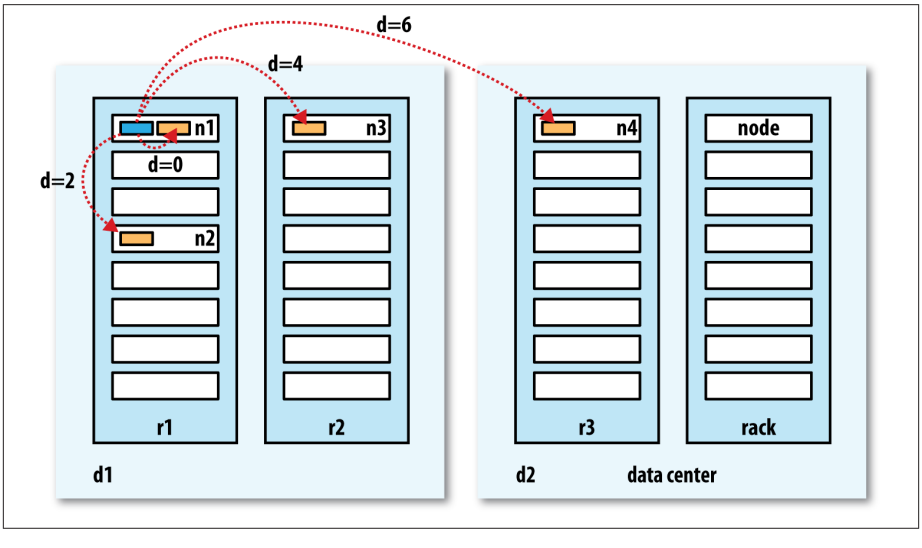


Figure 3-3. Network distance in Hadoop

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.

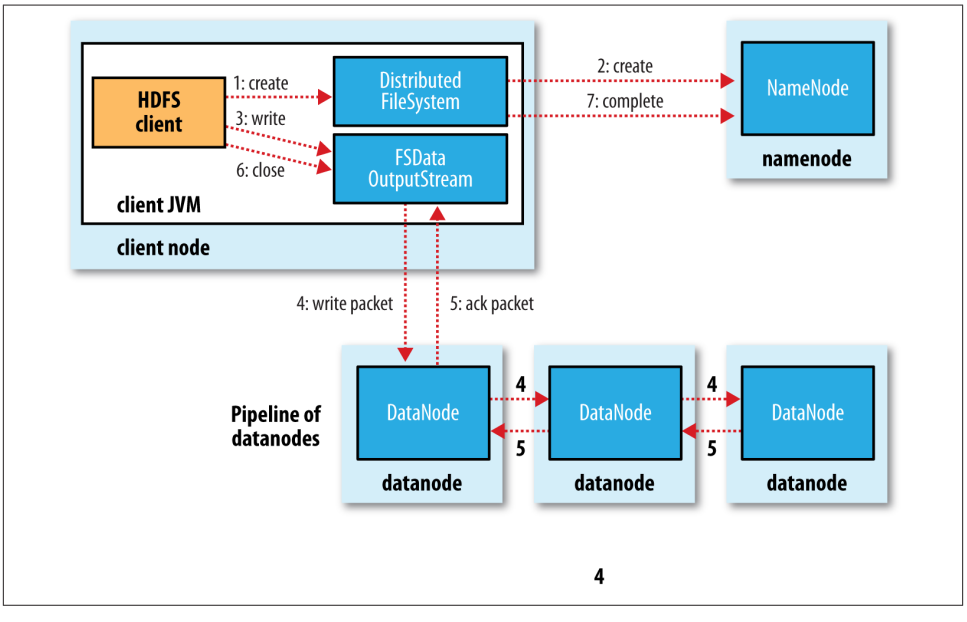


Figure 3-4. A client writing data to HDFS

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.

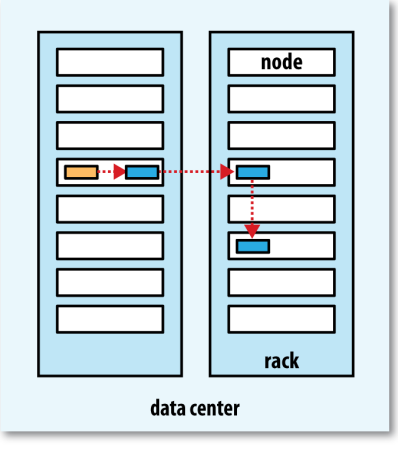


Figure 3-5. A typical replica pipeline

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.