

O'REILLY®

4th Edition  
Revised & Updated



# Hadoop

## The Definitive Guide

---

STORAGE AND ANALYSIS AT INTERNET SCALE

Tom White

# 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.

“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

- 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

**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



5 4 9 9 9



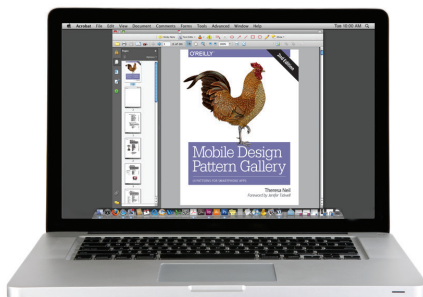
Twitter: @oreillymedia  
facebook.com/oreilly

# O'Reilly ebooks.

Your bookshelf on your devices.



**PDF**



**Mobi**



**ePub**



**DAISY**

When you buy an ebook through [oreilly.com](http://oreilly.com) you get lifetime access to the book, and whenever possible we provide it to you in four DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, and DAISY—that you can use on the devices of your choice. Our ebook files are fully searchable, and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

**Learn more at [ebooks.oreilly.com](http://ebooks.oreilly.com)**

You can also purchase O'Reilly ebooks through the iBookstore, the [Android Marketplace](http://AndroidMarketplace), and [Amazon.com](http://Amazon.com).

**O'REILLY®**

## **Hadoop: The Definitive Guide, Fourth Edition**

by Tom White

Copyright © 2015 Tom White. All rights reserved.

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](mailto: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

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

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Hadoop: The Definitive Guide*, the cover image of an African elephant, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

ISBN: 978-1-491-90163-2

[M]

---

# Table of Contents

<b>Foreword.....</b>	<b>xvii</b>
<b>Preface.....</b>	<b>xix</b>

---

## **Part I. Hadoop Fundamentals**

<b>1. Meet Hadoop.....</b>	<b>3</b>
Data!	3
Data Storage and Analysis	5
Querying All Your Data	6
Beyond Batch	6
Comparison with Other Systems	8
Relational Database Management Systems	8
Grid Computing	10
Volunteer Computing	11
A Brief History of Apache Hadoop	12
What's in This Book?	15
<b>2. MapReduce.....</b>	<b>19</b>
A Weather Dataset	19
Data Format	19
Analyzing the Data with Unix Tools	21
Analyzing the Data with Hadoop	22
Map and Reduce	22
Java MapReduce	24
Scaling Out	30
Data Flow	30
Combiner Functions	34
Running a Distributed MapReduce Job	37
Hadoop Streaming	37

---

Ruby	37
Python	40
<b>3. The Hadoop Distributed Filesystem.....</b>	<b>43</b>
The Design of HDFS	43
HDFS Concepts	45
Blocks	45
Namenodes and Datanodes	46
Block Caching	47
HDFS Federation	48
HDFS High Availability	48
The Command-Line Interface	50
Basic Filesystem Operations	51
Hadoop Filesystems	53
Interfaces	54
The Java Interface	56
Reading Data from a Hadoop URL	57
Reading Data Using the FileSystem API	58
Writing Data	61
Directories	63
Querying the Filesystem	63
Deleting Data	68
Data Flow	69
Anatomy of a File Read	69
Anatomy of a File Write	72
Coherency Model	74
Parallel Copying with distcp	76
Keeping an HDFS Cluster Balanced	77
<b>4. YARN.....</b>	<b>79</b>
Anatomy of a YARN Application Run	80
Resource Requests	81
Application Lifespan	82
Building YARN Applications	82
YARN Compared to MapReduce 1	83
Scheduling in YARN	85
Scheduler Options	86
Capacity Scheduler Configuration	88
Fair Scheduler Configuration	90
Delay Scheduling	94
Dominant Resource Fairness	95
Further Reading	96

<b>5. Hadoop I/O.....</b>	<b>97</b>
Data Integrity	97
Data Integrity in HDFS	98
LocalFileSystem	99
ChecksumFileSystem	99
Compression	100
Codecs	101
Compression and Input Splits	105
Using Compression in MapReduce	107
Serialization	109
The Writable Interface	110
Writable Classes	113
Implementing a Custom Writable	121
Serialization Frameworks	126
File-Based Data Structures	127
SequenceFile	127
MapFile	135
Other File Formats and Column-Oriented Formats	136

---

## Part II. MapReduce

<b>6. Developing a MapReduce Application.....</b>	<b>141</b>
The Configuration API	141
Combining Resources	143
Variable Expansion	143
Setting Up the Development Environment	144
Managing Configuration	146
GenericOptionsParser, Tool, and ToolRunner	148
Writing a Unit Test with MRUnit	152
Mapper	153
Reducer	156
Running Locally on Test Data	156
Running a Job in a Local Job Runner	157
Testing the Driver	158
Running on a Cluster	160
Packaging a Job	160
Launching a Job	162
The MapReduce Web UI	165
Retrieving the Results	167
Debugging a Job	168
Hadoop Logs	172

Remote Debugging	174
Tuning a Job	175
Profiling Tasks	175
MapReduce Workflows	177
Decomposing a Problem into MapReduce Jobs	177
JobControl	178
Apache Oozie	179
<b>7. How MapReduce Works. ....</b>	<b>185</b>
Anatomy of a MapReduce Job Run	185
Job Submission	186
Job Initialization	187
Task Assignment	188
Task Execution	189
Progress and Status Updates	190
Job Completion	192
Failures	193
Task Failure	193
Application Master Failure	194
Node Manager Failure	195
Resource Manager Failure	196
Shuffle and Sort	197
The Map Side	197
The Reduce Side	198
Configuration Tuning	201
Task Execution	203
The Task Execution Environment	203
Speculative Execution	204
Output Committers	206
<b>8. MapReduce Types and Formats. ....</b>	<b>209</b>
MapReduce Types	209
The Default MapReduce Job	214
Input Formats	220
Input Splits and Records	220
Text Input	232
Binary Input	236
Multiple Inputs	237
Database Input (and Output)	238
Output Formats	238
Text Output	239
Binary Output	239



Multiple Outputs	240
Lazy Output	245
Database Output	245
<b>9. MapReduce Features.....</b>	<b>247</b>
Counters	247
Built-in Counters	247
User-Defined Java Counters	251
User-Defined Streaming Counters	255
Sorting	255
Preparation	256
Partial Sort	257
Total Sort	259
Secondary Sort	262
Joins	268
Map-Side Joins	269
Reduce-Side Joins	270
Side Data Distribution	273
Using the Job Configuration	273
Distributed Cache	274
MapReduce Library Classes	279

---

## Part III. Hadoop Operations

<b>10. Setting Up a Hadoop Cluster.....</b>	<b>283</b>
Cluster Specification	284
Cluster Sizing	285
Network Topology	286
Cluster Setup and Installation	288
Installing Java	288
Creating Unix User Accounts	288
Installing Hadoop	289
Configuring SSH	289
Configuring Hadoop	290
Formatting the HDFS Filesystem	290
Starting and Stopping the Daemons	290
Creating User Directories	292
Hadoop Configuration	292
Configuration Management	293
Environment Settings	294
Important Hadoop Daemon Properties	296

Hadoop Daemon Addresses and Ports	304
Other Hadoop Properties	307
Security	309
Kerberos and Hadoop	309
Delegation Tokens	312
Other Security Enhancements	313
Benchmarking a Hadoop Cluster	314
Hadoop Benchmarks	314
User Jobs	316
<b>11. Administering Hadoop.....</b>	<b>317</b>
HDFS	317
Persistent Data Structures	317
Safe Mode	322
Audit Logging	324
Tools	325
Monitoring	330
Logging	330
Metrics and JMX	331
Maintenance	332
Routine Administration Procedures	332
Commissioning and Decommissioning Nodes	334
Upgrades	337
<hr/>	
<b>Part IV. Related Projects</b>	
<b>12. Avro.....</b>	<b>345</b>
Avro Data Types and Schemas	346
In-Memory Serialization and Deserialization	349
The Specific API	351
Avro Datafiles	352
Interoperability	354
Python API	354
Avro Tools	355
Schema Resolution	355
Sort Order	358
Avro MapReduce	359
Sorting Using Avro MapReduce	363
Avro in Other Languages	365

<b>13. Parquet.....</b>	<b>367</b>
Data Model	368
Nested Encoding	370
Parquet File Format	370
Parquet Configuration	372
Writing and Reading Parquet Files	373
Avro, Protocol Buffers, and Thrift	375
Parquet MapReduce	377
 <b>14. Flume.....</b>	 <b>381</b>
Installing Flume	381
An Example	382
Transactions and Reliability	384
Batching	385
The HDFS Sink	385
Partitioning and Interceptors	387
File Formats	387
Fan Out	388
Delivery Guarantees	389
Replicating and Multiplexing Selectors	390
Distribution: Agent Tiers	390
Delivery Guarantees	393
Sink Groups	395
Integrating Flume with Applications	398
Component Catalog	399
Further Reading	400
 <b>15. Sqoop.....</b>	 <b>401</b>
Getting Sqoop	401
Sqoop Connectors	403
A Sample Import	403
Text and Binary File Formats	406
Generated Code	407
Additional Serialization Systems	407
Imports: A Deeper Look	408
Controlling the Import	410
Imports and Consistency	411
Incremental Imports	411
Direct-Mode Imports	411
Working with Imported Data	412
Imported Data and Hive	413
Importing Large Objects	415

Performing an Export	417
Exports: A Deeper Look	419
Exports and Transactionality	420
Exports and SequenceFiles	421
Further Reading	422
<b>16. Pig.....</b>	<b>423</b>
Installing and Running Pig	424
Execution Types	424
Running Pig Programs	426
Grunt	426
Pig Latin Editors	427
An Example	427
Generating Examples	429
Comparison with Databases	430
Pig Latin	432
Structure	432
Statements	433
Expressions	438
Types	439
Schemas	441
Functions	445
Macros	447
User-Defined Functions	448
A Filter UDF	448
An Eval UDF	452
A Load UDF	453
Data Processing Operators	456
Loading and Storing Data	456
Filtering Data	457
Grouping and Joining Data	459
Sorting Data	465
Combining and Splitting Data	466
Pig in Practice	466
Parallelism	467
Anonymous Relations	467
Parameter Substitution	467
Further Reading	469
<b>17. Hive.....</b>	<b>471</b>
Installing Hive	472
The Hive Shell	473

An Example	474
Running Hive	475
Configuring Hive	475
Hive Services	478
The Metastore	480
Comparison with Traditional Databases	482
Schema on Read Versus Schema on Write	482
Updates, Transactions, and Indexes	483
SQL-on-Hadoop Alternatives	484
HiveQL	485
Data Types	486
Operators and Functions	488
Tables	489
Managed Tables and External Tables	490
Partitions and Buckets	491
Storage Formats	496
Importing Data	500
Altering Tables	502
Dropping Tables	502
Querying Data	503
Sorting and Aggregating	503
MapReduce Scripts	503
Joins	505
Subqueries	508
Views	509
User-Defined Functions	510
Writing a UDF	511
Writing a UDAF	513
Further Reading	518
<b>18. Crunch.....</b>	<b>519</b>
An Example	520
The Core Crunch API	523
Primitive Operations	523
Types	528
Sources and Targets	531
Functions	533
Materialization	535
Pipeline Execution	538
Running a Pipeline	538
Stopping a Pipeline	539
Inspecting a Crunch Plan	540

Iterative Algorithms	543
Checkpointing a Pipeline	545
Crunch Libraries	545
Further Reading	548
<b>19. Spark.....</b>	<b>549</b>
Installing Spark	550
An Example	550
Spark Applications, Jobs, Stages, and Tasks	552
A Scala Standalone Application	552
A Java Example	554
A Python Example	555
Resilient Distributed Datasets	556
Creation	556
Transformations and Actions	557
Persistence	560
Serialization	562
Shared Variables	564
Broadcast Variables	564
Accumulators	564
Anatomy of a Spark Job Run	565
Job Submission	565
DAG Construction	566
Task Scheduling	569
Task Execution	570
Executors and Cluster Managers	570
Spark on YARN	571
Further Reading	574
<b>20. HBase.....</b>	<b>575</b>
HBasics	575
Backdrop	576
Concepts	576
Whirlwind Tour of the Data Model	576
Implementation	578
Installation	581
Test Drive	582
Clients	584
Java	584
MapReduce	587
REST and Thrift	589
Building an Online Query Application	589

Schema Design	590
Loading Data	591
Online Queries	594
HBase Versus RDBMS	597
Successful Service	598
HBase	599
Praxis	600
HDFS	600
UI	601
Metrics	601
Counters	601
Further Reading	601
<b>21. ZooKeeper.....</b>	<b>603</b>
Installing and Running ZooKeeper	604
An Example	606
Group Membership in ZooKeeper	606
Creating the Group	607
Joining a Group	609
Listing Members in a Group	610
Deleting a Group	612
The ZooKeeper Service	613
Data Model	614
Operations	616
Implementation	620
Consistency	621
Sessions	623
States	625
Building Applications with ZooKeeper	627
A Configuration Service	627
The Resilient ZooKeeper Application	630
A Lock Service	634
More Distributed Data Structures and Protocols	636
ZooKeeper in Production	637
Resilience and Performance	637
Configuration	639
Further Reading	640

---

## Part V. Case Studies

<b>22. Composable Data at Cerner.....</b>	<b>643</b>
From CPUs to Semantic Integration	643
Enter Apache Crunch	644
Building a Complete Picture	644
Integrating Healthcare Data	647
Composability over Frameworks	650
Moving Forward	651
<b>23. Biological Data Science: Saving Lives with Software.....</b>	<b>653</b>
The Structure of DNA	655
The Genetic Code: Turning DNA Letters into Proteins	656
Thinking of DNA as Source Code	657
The Human Genome Project and Reference Genomes	659
Sequencing and Aligning DNA	660
ADAM, A Scalable Genome Analysis Platform	661
Literate programming with the Avro interface description language (IDL)	662
Column-oriented access with Parquet	663
A simple example: <i>k</i> -mer counting using Spark and ADAM	665
From Personalized Ads to Personalized Medicine	667
Join In	668
<b>24. Cascading.....</b>	<b>669</b>
Fields, Tuples, and Pipes	670
Operations	673
Taps, Schemes, and Flows	675
Cascading in Practice	676
Flexibility	679
Hadoop and Cascading at ShareThis	680
Summary	684
<b>A. Installing Apache Hadoop.....</b>	<b>685</b>
<b>B. Cloudera's Distribution Including Apache Hadoop.....</b>	<b>691</b>
<b>C. Preparing the NCDC Weather Data.....</b>	<b>693</b>
<b>D. The Old and New Java MapReduce APIs.....</b>	<b>697</b>
<b>Index.....</b>	<b>701</b>



---

## 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.<sup>1</sup> 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:<sup>2</sup>

- 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).<sup>3</sup>

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,<sup>4</sup> 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.<sup>5</sup>

Table 1-1. RDBMS compared to MapReduce

	Traditional RDBMS	MapReduce
<b>Data size</b>	Gigabytes	Petabytes
<b>Access</b>	Interactive and batch	Batch
<b>Updates</b>	Read and write many times	Write once, read many times
<b>Transactions</b>	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
<b>Structure</b>	Schema-on-write	Schema-on-read
<b>Integrity</b>	High	Low
<b>Scaling</b>	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.<sup>6</sup> 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<sup>7</sup> 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<sup>8</sup> 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.<sup>9</sup> 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.<sup>10</sup> 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.<sup>11</sup> 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.<sup>12</sup>

## 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.<sup>13</sup> 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.<sup>14</sup> In November of the same year, Google reported that its MapReduce implementation sorted 1 terabyte in 68 seconds.<sup>15</sup> Then, in April 2009, it was announced that a team at Yahoo! had used Hadoop to sort 1 terabyte in 62 seconds.<sup>16</sup>

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.<sup>17</sup>

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.

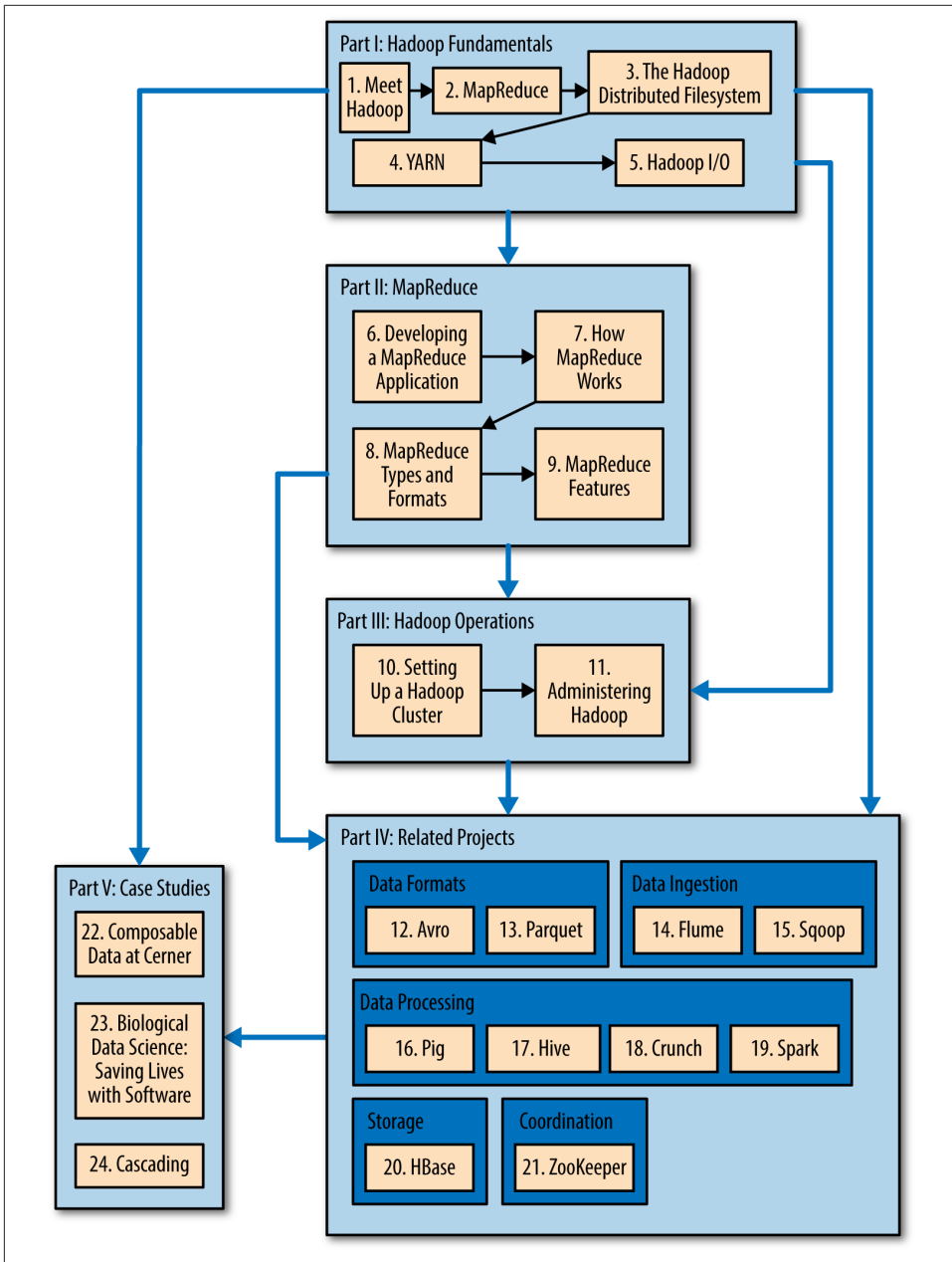


Figure 1-1. Structure of the book: there are various pathways through the content

# Want to read more?

You can [buy this book](#) at oreilly.com in print and ebook format.

**Buy 2 books, get the 3rd FREE!**

Use discount code OPC10

All orders over \$29.95 qualify for **free shipping** within the US.

It's also available at your favorite book retailer, including the iBookstore, the [Android Marketplace](#), and [Amazon.com](#).



**O'REILLY®**