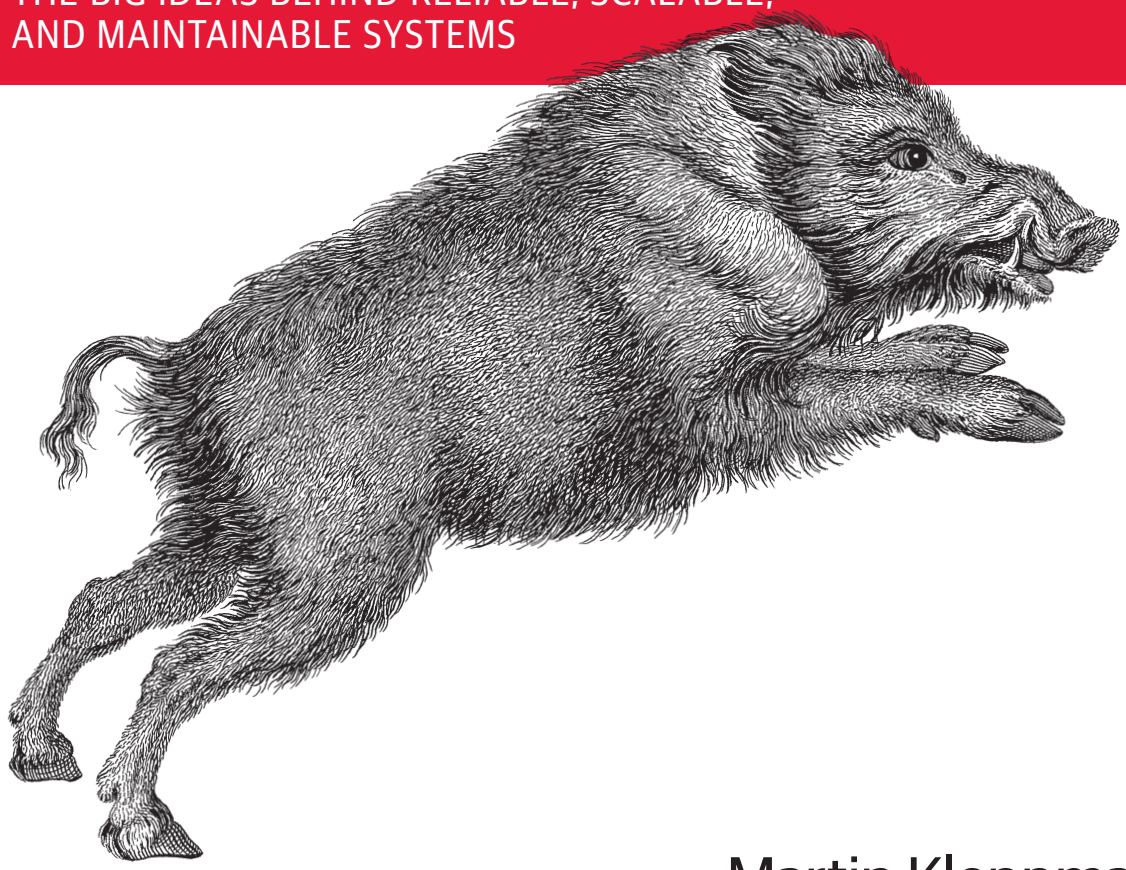
O'REILLY®

Designing Data-Intensive Applications

THE BIG IDEAS BEHIND RELIABLE, SCALABLE, AND MAINTAINABLE SYSTEMS



Martin Kleppmann

O'REILLY®

Designing Data-Intensive Applications

Data is at the center of many challenges in system design today. Difficult issues need to be figured out, such as scalability, consistency, reliability, efficiency, and maintainability. In addition, we have an overwhelming variety of tools, including relational databases, NoSQL datastores, stream or batch processors, and message brokers. What are the right choices for your application? How do you make sense of all these buzzwords?

In this practical and comprehensive guide, author Martin Kleppmann helps you navigate this diverse landscape by examining the pros and cons of various technologies for processing and storing data. Software keeps changing, but the fundamental principles remain the same. With this book, software engineers and architects will learn how to apply those ideas in practice, and how to make full use of data in modern applications.

- Peer under the hood of the systems you already use, and learn how to use and operate them more effectively
- Make informed decisions by identifying the strengths and weaknesses of different tools
- Navigate the trade-offs around consistency, scalability, fault tolerance, and complexity
- Understand the distributed systems research upon which modern databases are built
- Peek behind the scenes of major online services, and learn from their architectures

Martin Kleppmann is a researcher in distributed systems at the University of Cambridge, UK. Previously he was a software engineer and entrepreneur at internet companies including LinkedIn and Rapportive, where he worked on large-scale data infrastructure. Martin is a regular conference speaker, blogger, and open source contributor.

"This book is awesome. It bridges the huge gap between distributed systems theory and practical engineering. I wish it had existed a decade ago, so I could have read it then and saved myself all the mistakes along the way."

—Jay Kreps
Creator of Apache Kafka
and CEO of Confluent

"This book should be required reading for software engineers. *Designing Data-Intensive Applications* is a rare resource that connects theory and practice to help developers make smart decisions as they design and implement data infrastructure and systems."

—Kevin Scott
Chief Technology Officer at Microsoft

DATA | HADOOP

US $44.99 **CAN $59.99**
ISBN: 978-1-449-37332-0

Twitter: @oreillymedia
facebook.com/oreilly

Designing Data-Intensive Applications

The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

Martin Kleppmann

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Designing Data-Intensive Applications
by Martin Kleppmann

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://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Ann Spencer and Marie Beaugureau
Production Editor: Kristen Brown
Copyeditor: Rachel Head
Proofreader: Amanda Kersey
Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2017: First Edition

Revision History for the First Edition
2017-03-01: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449373320* for release details.

978-1-449-37332-0

[LSI]

Technology is a powerful force in our society. Data, software, and communication can be used for bad: to entrench unfair power structures, to undermine human rights, and to protect vested interests. But they can also be used for good: to make underrepresented people's voices heard, to create opportunities for everyone, and to avert disasters. This book is dedicated to everyone working toward the good.

Computing is pop culture. […] Pop culture holds a disdain for history. Pop culture is all about identity and feeling like you're participating. It has nothing to do with cooperation, the past or the future—it's living in the present. I think the same is true of most people who write code for money. They have no idea where [their culture came from].

—Alan Kay, in interview with *Dr Dobb's Journal* (2012)

Table of Contents

Part II. Distributed Data

Part III. Derived Data

Preface

If you have worked in software engineering in recent years, especially in server-side and backend systems, you have probably been bombarded with a plethora of buzzwords relating to storage and processing of data. NoSQL! Big Data! Web-scale! Sharding! Eventual consistency! ACID! CAP theorem! Cloud services! MapReduce! Real-time!

In the last decade we have seen many interesting developments in databases, in distributed systems, and in the ways we build applications on top of them. There are various driving forces for these developments:

- Internet companies such as Google, Yahoo!, Amazon, Facebook, LinkedIn, Microsoft, and Twitter are handling huge volumes of data and traffic, forcing them to create new tools that enable them to efficiently handle such scale.
- Businesses need to be agile, test hypotheses cheaply, and respond quickly to new market insights by keeping development cycles short and data models flexible.
- Free and open source software has become very successful and is now preferred to commercial or bespoke in-house software in many environments.
- CPU clock speeds are barely increasing, but multi-core processors are standard, and networks are getting faster. This means parallelism is only going to increase.
- Even if you work on a small team, you can now build systems that are distributed across many machines and even multiple geographic regions, thanks to infrastructure as a service (IaaS) such as Amazon Web Services.
- Many services are now expected to be highly available; extended downtime due to outages or maintenance is becoming increasingly unacceptable.

Data-intensive applications are pushing the boundaries of what is possible by making use of these technological developments. We call an application *data-intensive* if data is its primary challenge—the quantity of data, the complexity of data, or the speed at

which it is changing—as opposed to *compute-intensive*, where CPU cycles are the bottleneck.

The tools and technologies that help data-intensive applications store and process data have been rapidly adapting to these changes. New types of database systems ("NoSQL") have been getting lots of attention, but message queues, caches, search indexes, frameworks for batch and stream processing, and related technologies are very important too. Many applications use some combination of these.

The buzzwords that fill this space are a sign of enthusiasm for the new possibilities, which is a great thing. However, as software engineers and architects, we also need to have a technically accurate and precise understanding of the various technologies and their trade-offs if we want to build good applications. For that understanding, we have to dig deeper than buzzwords.

Fortunately, behind the rapid changes in technology, there are enduring principles that remain true, no matter which version of a particular tool you are using. If you understand those principles, you're in a position to see where each tool fits in, how to make good use of it, and how to avoid its pitfalls. That's where this book comes in.

The goal of this book is to help you navigate the diverse and fast-changing landscape of technologies for processing and storing data. This book is not a tutorial for one particular tool, nor is it a textbook full of dry theory. Instead, we will look at examples of successful data systems: technologies that form the foundation of many popular applications and that have to meet scalability, performance, and reliability requirements in production every day.

We will dig into the internals of those systems, tease apart their key algorithms, discuss their principles and the trade-offs they have to make. On this journey, we will try to find useful ways of *thinking about* data systems—not just *how* they work, but also *why* they work that way, and what questions we need to ask.

After reading this book, you will be in a great position to decide which kind of technology is appropriate for which purpose, and understand how tools can be combined to form the foundation of a good application architecture. You won't be ready to build your own database storage engine from scratch, but fortunately that is rarely necessary. You will, however, develop a good intuition for what your systems are doing under the hood so that you can reason about their behavior, make good design decisions, and track down any problems that may arise.

Who Should Read This Book?

If you develop applications that have some kind of server/backend for storing or processing data, and your applications use the internet (e.g., web applications, mobile apps, or internet-connected sensors), then this book is for you.

This book is for software engineers, software architects, and technical managers who love to code. It is especially relevant if you need to make decisions about the architecture of the systems you work on—for example, if you need to choose tools for solving a given problem and figure out how best to apply them. But even if you have no choice over your tools, this book will help you better understand their strengths and weaknesses.

You should have some experience building web-based applications or network services, and you should be familiar with relational databases and SQL. Any nonrelational databases and other data-related tools you know are a bonus, but not required. A general understanding of common network protocols like TCP and HTTP is helpful. Your choice of programming language or framework makes no difference for this book.

If any of the following are true for you, you'll find this book valuable:

- You want to learn how to make data systems scalable, for example, to support web or mobile apps with millions of users.
- You need to make applications highly available (minimizing downtime) and operationally robust.
- You are looking for ways of making systems easier to maintain in the long run, even as they grow and as requirements and technologies change.
- You have a natural curiosity for the way things work and want to know what goes on inside major websites and online services. This book breaks down the internals of various databases and data processing systems, and it's great fun to explore the bright thinking that went into their design.

Sometimes, when discussing scalable data systems, people make comments along the lines of, "You're not Google or Amazon. Stop worrying about scale and just use a relational database." There is truth in that statement: building for scale that you don't need is wasted effort and may lock you into an inflexible design. In effect, it is a form of premature optimization. However, it's also important to choose the right tool for the job, and different technologies each have their own strengths and weaknesses. As we shall see, relational databases are important but not the final word on dealing with data.

Scope of This Book

This book does not attempt to give detailed instructions on how to install or use specific software packages or APIs, since there is already plenty of documentation for those things. Instead we discuss the various principles and trade-offs that are fundamental to data systems, and we explore the different design decisions taken by different products.

In the ebook editions we have included links to the full text of online resources. All links were verified at the time of publication, but unfortunately links tend to break frequently due to the nature of the web. If you come across a broken link, or if you are reading a print copy of this book, you can look up references using a search engine. For academic papers, you can search for the title in Google Scholar to find open-access PDF files. Alternatively, you can find all of the references at *https://github.com/ept/ddia-references*, where we maintain up-to-date links.

We look primarily at the *architecture* of data systems and the ways they are integrated into data-intensive applications. This book doesn't have space to cover deployment, operations, security, management, and other areas—those are complex and important topics, and we wouldn't do them justice by making them superficial side notes in this book. They deserve books of their own.

Many of the technologies described in this book fall within the realm of the *Big Data* buzzword. However, the term "Big Data" is so overused and underdefined that it is not useful in a serious engineering discussion. This book uses less ambiguous terms, such as single-node versus distributed systems, or online/interactive versus offline/batch processing systems.

This book has a bias toward free and open source software (FOSS), because reading, modifying, and executing source code is a great way to understand how something works in detail. Open platforms also reduce the risk of vendor lock-in. However, where appropriate, we also discuss proprietary software (closed-source software, software as a service, or companies' in-house software that is only described in literature but not released publicly).

Outline of This Book

This book is arranged into three parts:

1. In Part I, we discuss the fundamental ideas that underpin the design of data-intensive applications. We start in Chapter 1 by discussing what we're actually trying to achieve: reliability, scalability, and maintainability; how we need to think about them; and how we can achieve them. In Chapter 2 we compare several different data models and query languages, and see how they are appropriate to different situations. In Chapter 3 we talk about storage engines: how databases arrange data on disk so that we can find it again efficiently. Chapter 4 turns to formats for data encoding (serialization) and evolution of schemas over time.
2. In Part II, we move from data stored on one machine to data that is distributed across multiple machines. This is often necessary for scalability, but brings with it a variety of unique challenges. We first discuss replication (Chapter 5), partitioning/sharding (Chapter 6), and transactions (Chapter 7). We then go into

more detail on the problems with distributed systems (Chapter 8) and what it means to achieve consistency and consensus in a distributed system (Chapter 9).

3. In Part III, we discuss systems that derive some datasets from other datasets. Derived data often occurs in heterogeneous systems: when there is no one database that can do everything well, applications need to integrate several different databases, caches, indexes, and so on. In Chapter 10 we start with a batch processing approach to derived data, and we build upon it with stream processing in Chapter 11. Finally, in Chapter 12 we put everything together and discuss approaches for building reliable, scalable, and maintainable applications in the future.

References and Further Reading

Most of what we discuss in this book has already been said elsewhere in some form or another—in conference presentations, research papers, blog posts, code, bug trackers, mailing lists, and engineering folklore. This book summarizes the most important ideas from many different sources, and it includes pointers to the original literature throughout the text. The references at the end of each chapter are a great resource if you want to explore an area in more depth, and most of them are freely available online.

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, 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, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

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/designing-data-intensive-apps*.

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

This book is an amalgamation and systematization of a large number of other people's ideas and knowledge, combining experience from both academic research and industrial practice. In computing we tend to be attracted to things that are new and shiny, but I think we have a huge amount to learn from things that have been done before. This book has over 800 references to articles, blog posts, talks, documentation, and more, and they have been an invaluable learning resource for me. I am very grateful to the authors of this material for sharing their knowledge.

I have also learned a lot from personal conversations, thanks to a large number of people who have taken the time to discuss ideas or patiently explain things to me. In particular, I would like to thank Joe Adler, Ross Anderson, Peter Bailis, Márton Balassi, Alastair Beresford, Mark Callaghan, Mat Clayton, Patrick Collison, Sean Cribbs, Shirshanka Das, Niklas Ekström, Stephan Ewen, Alan Fekete, Gyula Fóra, Camille Fournier, Andres Freund, John Garbutt, Seth Gilbert, Tom Haggett, Pat Helland, Joe Hellerstein, Jakob Homan, Heidi Howard, John Hugg, Julian Hyde, Conrad Irwin, Evan Jones, Flavio Junqueira, Jessica Kerr, Kyle Kingsbury, Jay Kreps, Carl Lerche, Nicolas Liochon, Steve Loughran, Lee Mallabone, Nathan Marz, Caitie

McCaffrey, Josie McLellan, Christopher Meiklejohn, Ian Meyers, Neha Narkhede, Neha Narula, Cathy O'Neil, Onora O'Neill, Ludovic Orban, Zoran Perkov, Julia Powles, Chris Riccomini, Henry Robinson, David Rosenthal, Jennifer Rullmann, Matthew Sackman, Martin Scholl, Amit Sela, Gwen Shapira, Greg Spurrier, Sam Stokes, Ben Stopford, Tom Stuart, Diana Vasile, Rahul Vohra, Pete Warden, and Brett Wooldridge.

Several more people have been invaluable to the writing of this book by reviewing drafts and providing feedback. For these contributions I am particularly indebted to Raul Agepati, Tyler Akidau, Mattias Andersson, Sasha Baranov, Veena Basavaraj, David Beyer, Jim Brikman, Paul Carey, Raul Castro Fernandez, Joseph Chow, Derek Elkins, Sam Elliott, Alexander Gallego, Mark Grover, Stu Halloway, Heidi Howard, Nicola Kleppmann, Stefan Kruppa, Bjorn Madsen, Sander Mak, Stefan Podkowinski, Phil Potter, Hamid Ramazani, Sam Stokes, and Ben Summers. Of course, I take all responsibility for any remaining errors or unpalatable opinions in this book.

For helping this book become real, and for their patience with my slow writing and unusual requests, I am grateful to my editors Marie Beaugureau, Mike Loukides, Ann Spencer, and all the team at O'Reilly. For helping find the right words, I thank Rachel Head. For giving me the time and freedom to write in spite of other work commitments, I thank Alastair Beresford, Susan Goodhue, Neha Narkhede, and Kevin Scott.

Very special thanks are due to Shabbir Diwan and Edie Freedman, who illustrated with great care the maps that accompany the chapters. It's wonderful that they took on the unconventional idea of creating maps, and made them so beautiful and compelling.

Finally, my love goes to my family and friends, without whom I would not have been able to get through this writing process that has taken almost four years. You're the best.

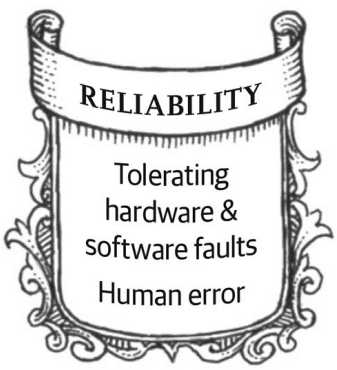
PART I
Foundations of Data Systems

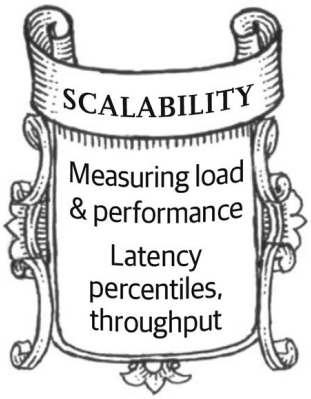
The first four chapters go through the fundamental ideas that apply to all data systems, whether running on a single machine or distributed across a cluster of machines:

1. Chapter 1 introduces the terminology and approach that we're going to use throughout this book. It examines what we actually mean by words like *reliability*, *scalability*, and *maintainability*, and how we can try to achieve these goals.
2. Chapter 2 compares several different data models and query languages—the most visible distinguishing factor between databases from a developer's point of view. We will see how different models are appropriate to different situations.
3. Chapter 3 turns to the internals of storage engines and looks at how databases lay out data on disk. Different storage engines are optimized for different workloads, and choosing the right one can have a huge effect on performance.
4. Chapter 4 compares various formats for data encoding (serialization) and especially examines how they fare in an environment where application requirements change and schemas need to adapt over time.

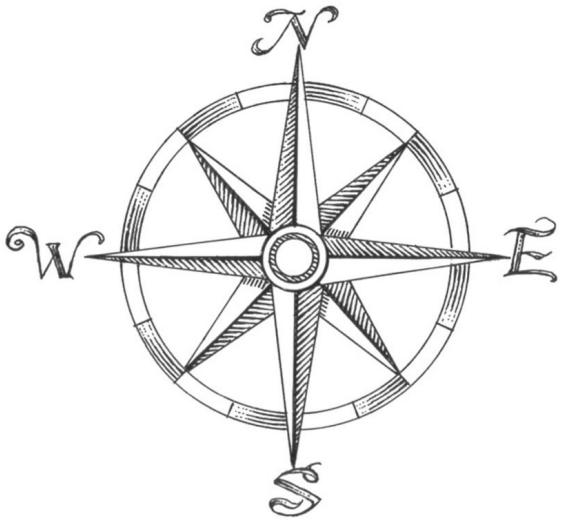
Later, Part II will turn to the particular issues of distributed data systems.


DESIGNING
Data-Intensive Applications
The big ideas behind reliable, scalable & maintainable systems
RELIABILITY
SCALABILITY
MAINTAINABILITY


RELIABILITY
Tolerating hardware & software faults
Human error


SCALABILITY
Measuring load & performance
Latency percentiles, throughput


MAINTAINABILITY
Operability, simplicity & evolvability


N
W
E
S

CHAPTER 1
Reliable, Scalable, and Maintainable Applications

> *The Internet was done so well that most people think of it as a natural resource like the Pacific Ocean, rather than something that was man-made. When was the last time a technology with a scale like that was so error-free?*
>
> —Alan Kay, in interview with *Dr Dobb's Journal* (2012)

Many applications today are *data-intensive*, as opposed to *compute-intensive*. Raw CPU power is rarely a limiting factor for these applications—bigger problems are usually the amount of data, the complexity of data, and the speed at which it is changing.

A data-intensive application is typically built from standard building blocks that provide commonly needed functionality. For example, many applications need to:

- Store data so that they, or another application, can find it again later (*databases*)
- Remember the result of an expensive operation, to speed up reads (*caches*)
- Allow users to search data by keyword or filter it in various ways (*search indexes*)
- Send a message to another process, to be handled asynchronously (*stream processing*)
- Periodically crunch a large amount of accumulated data (*batch processing*)

If that sounds painfully obvious, that's just because these *data systems* are such a successful abstraction: we use them all the time without thinking too much. When building an application, most engineers wouldn't dream of writing a new data storage engine from scratch, because databases are a perfectly good tool for the job.

But reality is not that simple. There are many database systems with different characteristics, because different applications have different requirements. There are various approaches to caching, several ways of building search indexes, and so on. When building an application, we still need to figure out which tools and which approaches are the most appropriate for the task at hand. And it can be hard to combine tools when you need to do something that a single tool cannot do alone.

This book is a journey through both the principles and the practicalities of data systems, and how you can use them to build data-intensive applications. We will explore what different tools have in common, what distinguishes them, and how they achieve their characteristics.

In this chapter, we will start by exploring the fundamentals of what we are trying to achieve: reliable, scalable, and maintainable data systems. We'll clarify what those things mean, outline some ways of thinking about them, and go over the basics that we will need for later chapters. In the following chapters we will continue layer by layer, looking at different design decisions that need to be considered when working on a data-intensive application.

Thinking About Data Systems

We typically think of databases, queues, caches, etc. as being very different categories of tools. Although a database and a message queue have some superficial similarity—both store data for some time—they have very different access patterns, which means different performance characteristics, and thus very different implementations.

So why should we lump them all together under an umbrella term like *data systems*?

Many new tools for data storage and processing have emerged in recent years. They are optimized for a variety of different use cases, and they no longer neatly fit into traditional categories [1]. For example, there are datastores that are also used as message queues (Redis), and there are message queues with database-like durability guarantees (Apache Kafka). The boundaries between the categories are becoming blurred.

Secondly, increasingly many applications now have such demanding or wide-ranging requirements that a single tool can no longer meet all of its data processing and storage needs. Instead, the work is broken down into tasks that *can* be performed efficiently on a single tool, and those different tools are stitched together using application code.

For example, if you have an application-managed caching layer (using Memcached or similar), or a full-text search server (such as Elasticsearch or Solr) separate from your main database, it is normally the application code's responsibility to keep those caches and indexes in sync with the main database. Figure 1-1 gives a glimpse of what this may look like (we will go into detail in later chapters).

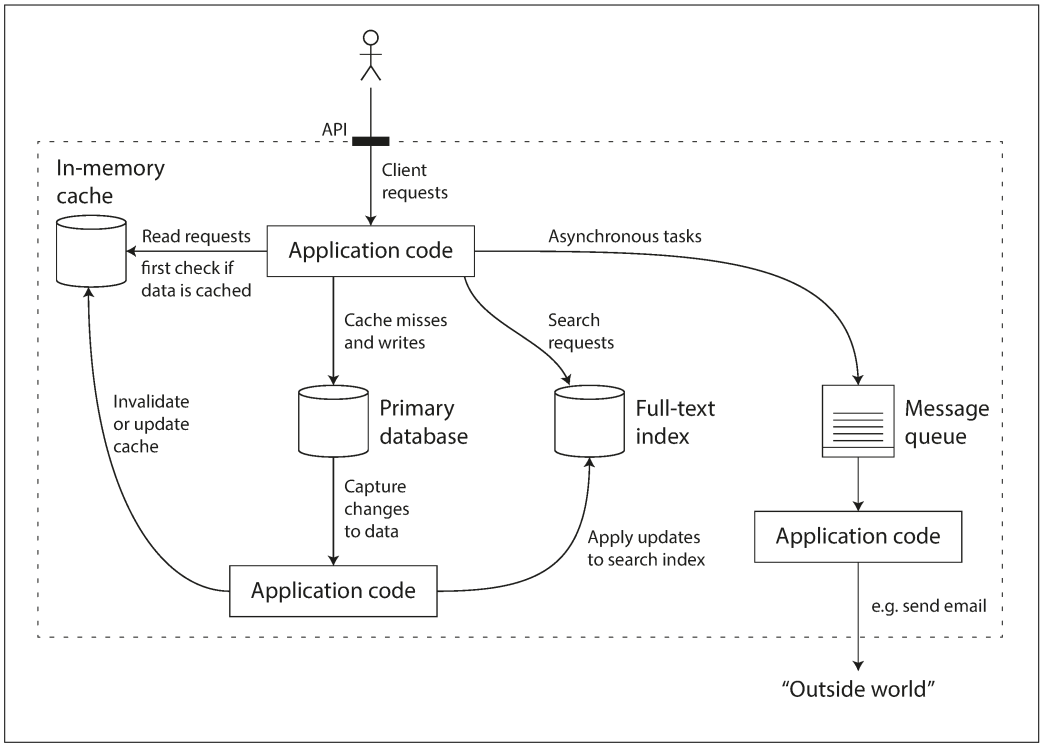


Figure 1-1. One possible architecture for a data system that combines several components.

When you combine several tools in order to provide a service, the service's interface or application programming interface (API) usually hides those implementation details from clients. Now you have essentially created a new, special-purpose data system from smaller, general-purpose components. Your composite data system may provide certain guarantees: e.g., that the cache will be correctly invalidated or updated on writes so that outside clients see consistent results. You are now not only an application developer, but also a data system designer.

If you are designing a data system or service, a lot of tricky questions arise. How do you ensure that the data remains correct and complete, even when things go wrong internally? How do you provide consistently good performance to clients, even when parts of your system are degraded? How do you scale to handle an increase in load? What does a good API for the service look like?

There are many factors that may influence the design of a data system, including the skills and experience of the people involved, legacy system dependencies, the timescale for delivery, your organization's tolerance of different kinds of risk, regulatory constraints, etc. Those factors depend very much on the situation.

In this book, we focus on three concerns that are important in most software systems:

Reliability
: The system should continue to work *correctly* (performing the correct function at the desired level of performance) even in the face of *adversity* (hardware or software faults, and even human error). See "Reliability" on page 6.

Scalability
: As the system *grows* (in data volume, traffic volume, or complexity), there should be reasonable ways of dealing with that growth. See "Scalability" on page 10.

Maintainability
: Over time, many different people will work on the system (engineering and operations, both maintaining current behavior and adapting the system to new use cases), and they should all be able to work on it *productively*. See "Maintainability" on page 18.

These words are often cast around without a clear understanding of what they mean. In the interest of thoughtful engineering, we will spend the rest of this chapter exploring ways of thinking about reliability, scalability, and maintainability. Then, in the following chapters, we will look at various techniques, architectures, and algorithms that are used in order to achieve those goals.

Reliability

Everybody has an intuitive idea of what it means for something to be reliable or unreliable. For software, typical expectations include:

- The application performs the function that the user expected.
- It can tolerate the user making mistakes or using the software in unexpected ways.
- Its performance is good enough for the required use case, under the expected load and data volume.
- The system prevents any unauthorized access and abuse.

If all those things together mean "working correctly," then we can understand *reliability* as meaning, roughly, "continuing to work correctly, even when things go wrong."

The things that can go wrong are called *faults*, and systems that anticipate faults and can cope with them are called *fault-tolerant* or *resilient*. The former term is slightly misleading: it suggests that we could make a system tolerant of every possible kind of fault, which in reality is not feasible. If the entire planet Earth (and all servers on it) were swallowed by a black hole, tolerance of that fault would require web hosting in

space—good luck getting that budget item approved. So it only makes sense to talk about tolerating *certain types* of faults.

Note that a fault is not the same as a failure [2]. A fault is usually defined as one component of the system deviating from its spec, whereas a *failure* is when the system as a whole stops providing the required service to the user. It is impossible to reduce the probability of a fault to zero; therefore it is usually best to design fault-tolerance mechanisms that prevent faults from causing failures. In this book we cover several techniques for building reliable systems from unreliable parts.

Counterintuitively, in such fault-tolerant systems, it can make sense to *increase* the rate of faults by triggering them deliberately—for example, by randomly killing individual processes without warning. Many critical bugs are actually due to poor error handling [3]; by deliberately inducing faults, you ensure that the fault-tolerance machinery is continually exercised and tested, which can increase your confidence that faults will be handled correctly when they occur naturally. The Netflix *Chaos Monkey* [4] is an example of this approach.

Although we generally prefer tolerating faults over preventing faults, there are cases where prevention is better than cure (e.g., because no cure exists). This is the case with security matters, for example: if an attacker has compromised a system and gained access to sensitive data, that event cannot be undone. However, this book mostly deals with the kinds of faults that can be cured, as described in the following sections.

Hardware Faults

When we think of causes of system failure, hardware faults quickly come to mind. Hard disks crash, RAM becomes faulty, the power grid has a blackout, someone unplugs the wrong network cable. Anyone who has worked with large datacenters can tell you that these things happen *all the time* when you have a lot of machines.

Hard disks are reported as having a mean time to failure (MTTF) of about 10 to 50 years [5, 6]. Thus, on a storage cluster with 10,000 disks, we should expect on average one disk to die per day.

Our first response is usually to add redundancy to the individual hardware components in order to reduce the failure rate of the system. Disks may be set up in a RAID configuration, servers may have dual power supplies and hot-swappable CPUs, and datacenters may have batteries and diesel generators for backup power. When one component dies, the redundant component can take its place while the broken component is replaced. This approach cannot completely prevent hardware problems from causing failures, but it is well understood and can often keep a machine running uninterrupted for years.

Until recently, redundancy of hardware components was sufficient for most applications, since it makes total failure of a single machine fairly rare. As long as you can restore a backup onto a new machine fairly quickly, the downtime in case of failure is not catastrophic in most applications. Thus, multi-machine redundancy was only required by a small number of applications for which high availability was absolutely essential.

However, as data volumes and applications' computing demands have increased, more applications have begun using larger numbers of machines, which proportionally increases the rate of hardware faults. Moreover, in some cloud platforms such as Amazon Web Services (AWS) it is fairly common for virtual machine instances to become unavailable without warning [7], as the platforms are designed to prioritize flexibility and elasticity[i] over single-machine reliability.

Hence there is a move toward systems that can tolerate the loss of entire machines, by using software fault-tolerance techniques in preference or in addition to hardware redundancy. Such systems also have operational advantages: a single-server system requires planned downtime if you need to reboot the machine (to apply operating system security patches, for example), whereas a system that can tolerate machine failure can be patched one node at a time, without downtime of the entire system (a *rolling upgrade*; see Chapter 4).

Software Errors

We usually think of hardware faults as being random and independent from each other: one machine's disk failing does not imply that another machine's disk is going to fail. There may be weak correlations (for example due to a common cause, such as the temperature in the server rack), but otherwise it is unlikely that a large number of hardware components will fail at the same time.

Another class of fault is a systematic error within the system [8]. Such faults are harder to anticipate, and because they are correlated across nodes, they tend to cause many more system failures than uncorrelated hardware faults [5]. Examples include:

- A software bug that causes every instance of an application server to crash when given a particular bad input. For example, consider the leap second on June 30, 2012, that caused many applications to hang simultaneously due to a bug in the Linux kernel [9].
- A runaway process that uses up some shared resource—CPU time, memory, disk space, or network bandwidth.

i. Defined in "Approaches for Coping with Load" on page 17.

- A service that the system depends on that slows down, becomes unresponsive, or starts returning corrupted responses.
- Cascading failures, where a small fault in one component triggers a fault in another component, which in turn triggers further faults [10].

The bugs that cause these kinds of software faults often lie dormant for a long time until they are triggered by an unusual set of circumstances. In those circumstances, it is revealed that the software is making some kind of assumption about its environment—and while that assumption is usually true, it eventually stops being true for some reason [11].

There is no quick solution to the problem of systematic faults in software. Lots of small things can help: carefully thinking about assumptions and interactions in the system; thorough testing; process isolation; allowing processes to crash and restart; measuring, monitoring, and analyzing system behavior in production. If a system is expected to provide some guarantee (for example, in a message queue, that the number of incoming messages equals the number of outgoing messages), it can constantly check itself while it is running and raise an alert if a discrepancy is found [12].

Human Errors

Humans design and build software systems, and the operators who keep the systems running are also human. Even when they have the best intentions, humans are known to be unreliable. For example, one study of large internet services found that configuration errors by operators were the leading cause of outages, whereas hardware faults (servers or network) played a role in only 10–25% of outages [13].

How do we make our systems reliable, in spite of unreliable humans? The best systems combine several approaches:

- Design systems in a way that minimizes opportunities for error. For example, well-designed abstractions, APIs, and admin interfaces make it easy to do "the right thing" and discourage "the wrong thing." However, if the interfaces are too restrictive people will work around them, negating their benefit, so this is a tricky balance to get right.
- Decouple the places where people make the most mistakes from the places where they can cause failures. In particular, provide fully featured non-production *sandbox* environments where people can explore and experiment safely, using real data, without affecting real users.
- Test thoroughly at all levels, from unit tests to whole-system integration tests and manual tests [3]. Automated testing is widely used, well understood, and especially valuable for covering corner cases that rarely arise in normal operation.

- Allow quick and easy recovery from human errors, to minimize the impact in the case of a failure. For example, make it fast to roll back configuration changes, roll out new code gradually (so that any unexpected bugs affect only a small subset of users), and provide tools to recompute data (in case it turns out that the old computation was incorrect).
- Set up detailed and clear monitoring, such as performance metrics and error rates. In other engineering disciplines this is referred to as *telemetry*. (Once a rocket has left the ground, telemetry is essential for tracking what is happening, and for understanding failures [14].) Monitoring can show us early warning signals and allow us to check whether any assumptions or constraints are being violated. When a problem occurs, metrics can be invaluable in diagnosing the issue.
- Implement good management practices and training—a complex and important aspect, and beyond the scope of this book.

How Important Is Reliability?

Reliability is not just for nuclear power stations and air traffic control software—more mundane applications are also expected to work reliably. Bugs in business applications cause lost productivity (and legal risks if figures are reported incorrectly), and outages of ecommerce sites can have huge costs in terms of lost revenue and damage to reputation.

Even in "noncritical" applications we have a responsibility to our users. Consider a parent who stores all their pictures and videos of their children in your photo application [15]. How would they feel if that database was suddenly corrupted? Would they know how to restore it from a backup?

There are situations in which we may choose to sacrifice reliability in order to reduce development cost (e.g., when developing a prototype product for an unproven market) or operational cost (e.g., for a service with a very narrow profit margin)—but we should be very conscious of when we are cutting corners.

Scalability

Even if a system is working reliably today, that doesn't mean it will necessarily work reliably in the future. One common reason for degradation is increased load: perhaps the system has grown from 10,000 concurrent users to 100,000 concurrent users, or from 1 million to 10 million. Perhaps it is processing much larger volumes of data than it did before.

Scalability is the term we use to describe a system's ability to cope with increased load. Note, however, that it is not a one-dimensional label that we can attach to a system: it is meaningless to say "X is scalable" or "Y doesn't scale." Rather, discussing

scalability means considering questions like "If the system grows in a particular way, what are our options for coping with the growth?" and "How can we add computing resources to handle the additional load?"

Describing Load

First, we need to succinctly describe the current load on the system; only then can we discuss growth questions (what happens if our load doubles?). Load can be described with a few numbers which we call *load parameters*. The best choice of parameters depends on the architecture of your system: it may be requests per second to a web server, the ratio of reads to writes in a database, the number of simultaneously active users in a chat room, the hit rate on a cache, or something else. Perhaps the average case is what matters for you, or perhaps your bottleneck is dominated by a small number of extreme cases.

To make this idea more concrete, let's consider Twitter as an example, using data published in November 2012 [16]. Two of Twitter's main operations are:

Post tweet
: A user can publish a new message to their followers (4.6k requests/sec on average, over 12k requests/sec at peak).

Home timeline
: A user can view tweets posted by the people they follow (300k requests/sec).

Simply handling 12,000 writes per second (the peak rate for posting tweets) would be fairly easy. However, Twitter's scaling challenge is not primarily due to tweet volume, but due to *fan-out*[ii]—each user follows many people, and each user is followed by many people. There are broadly two ways of implementing these two operations:

1. Posting a tweet simply inserts the new tweet into a global collection of tweets. When a user requests their home timeline, look up all the people they follow, find all the tweets for each of those users, and merge them (sorted by time). In a relational database like in Figure 1-2, you could write a query such as:

```
SELECT tweets.*, users.* FROM tweets
  JOIN users   ON tweets.sender_id    = users.id
  JOIN follows ON follows.followee_id = users.id
  WHERE follows.follower_id = current_user
```

ii. A term borrowed from electronic engineering, where it describes the number of logic gate inputs that are attached to another gate's output. The output needs to supply enough current to drive all the attached inputs. In transaction processing systems, we use it to describe the number of requests to other services that we need to make in order to serve one incoming request.

2. Maintain a cache for each user's home timeline—like a mailbox of tweets for each recipient user (see Figure 1-3). When a user *posts a tweet*, look up all the people who follow that user, and insert the new tweet into each of their home timeline caches. The request to read the home timeline is then cheap, because its result has been computed ahead of time.

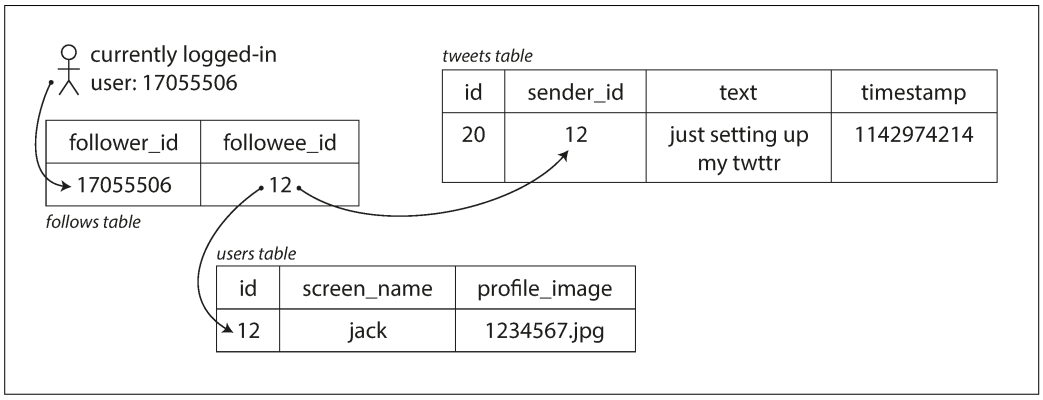


Figure 1-2. Simple relational schema for implementing a Twitter home timeline.

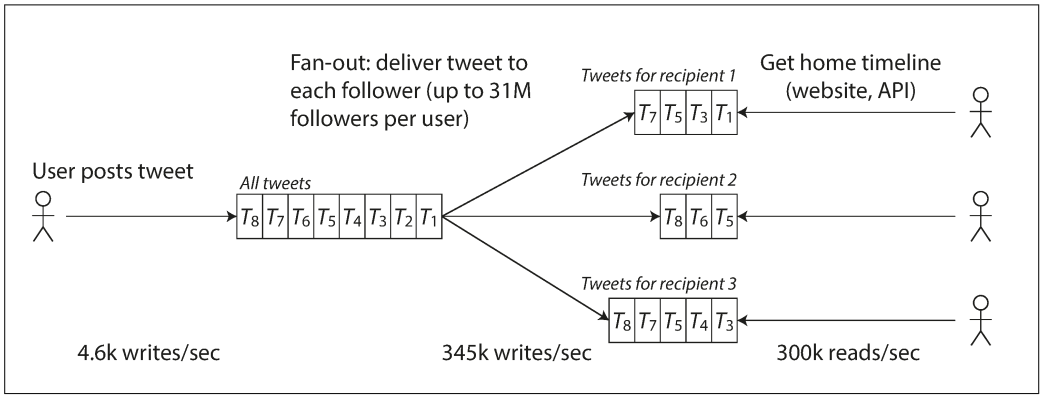


Figure 1-3. Twitter's data pipeline for delivering tweets to followers, with load parameters as of November 2012 [16].

The first version of Twitter used approach 1, but the systems struggled to keep up with the load of home timeline queries, so the company switched to approach 2. This works better because the average rate of published tweets is almost two orders of magnitude lower than the rate of home timeline reads, and so in this case it's preferable to do more work at write time and less at read time.

However, the downside of approach 2 is that posting a tweet now requires a lot of extra work. On average, a tweet is delivered to about 75 followers, so 4.6k tweets per second become 345k writes per second to the home timeline caches. But this average hides the fact that the number of followers per user varies wildly, and some users

have over 30 million followers. This means that a single tweet may result in over 30 million writes to home timelines! Doing this in a timely manner—Twitter tries to deliver tweets to followers within five seconds—is a significant challenge.

In the example of Twitter, the distribution of followers per user (maybe weighted by how often those users tweet) is a key load parameter for discussing scalability, since it determines the fan-out load. Your application may have very different characteristics, but you can apply similar principles to reasoning about its load.

The final twist of the Twitter anecdote: now that approach 2 is robustly implemented, Twitter is moving to a hybrid of both approaches. Most users' tweets continue to be fanned out to home timelines at the time when they are posted, but a small number of users with a very large number of followers (i.e., celebrities) are excepted from this fan-out. Tweets from any celebrities that a user may follow are fetched separately and merged with that user's home timeline when it is read, like in approach 1. This hybrid approach is able to deliver consistently good performance. We will revisit this example in Chapter 12 after we have covered some more technical ground.

Describing Performance

Once you have described the load on your system, you can investigate what happens when the load increases. You can look at it in two ways:

- When you increase a load parameter and keep the system resources (CPU, memory, network bandwidth, etc.) unchanged, how is the performance of your system affected?
- When you increase a load parameter, how much do you need to increase the resources if you want to keep performance unchanged?

Both questions require performance numbers, so let's look briefly at describing the performance of a system.

In a batch processing system such as Hadoop, we usually care about *throughput*—the number of records we can process per second, or the total time it takes to run a job on a dataset of a certain size.[iii] In online systems, what's usually more important is the service's *response time*—that is, the time between a client sending a request and receiving a response.

iii. In an ideal world, the running time of a batch job is the size of the dataset divided by the throughput. In practice, the running time is often longer, due to skew (data not being spread evenly across worker processes) and needing to wait for the slowest task to complete.

Latency and response time

Latency and *response time* are often used synonymously, but they are not the same. The response time is what the client sees: besides the actual time to process the request (the *service time*), it includes network delays and queueing delays. Latency is the duration that a request is waiting to be handled—during which it is *latent*, awaiting service [17].

Even if you only make the same request over and over again, you'll get a slightly different response time on every try. In practice, in a system handling a variety of requests, the response time can vary a lot. We therefore need to think of response time not as a single number, but as a *distribution* of values that you can measure.

In Figure 1-4, each gray bar represents a request to a service, and its height shows how long that request took. Most requests are reasonably fast, but there are occasional *outliers* that take much longer. Perhaps the slow requests are intrinsically more expensive, e.g., because they process more data. But even in a scenario where you'd think all requests should take the same time, you get variation: random additional latency could be introduced by a context switch to a background process, the loss of a network packet and TCP retransmission, a garbage collection pause, a page fault forcing a read from disk, mechanical vibrations in the server rack [18], or many other causes.

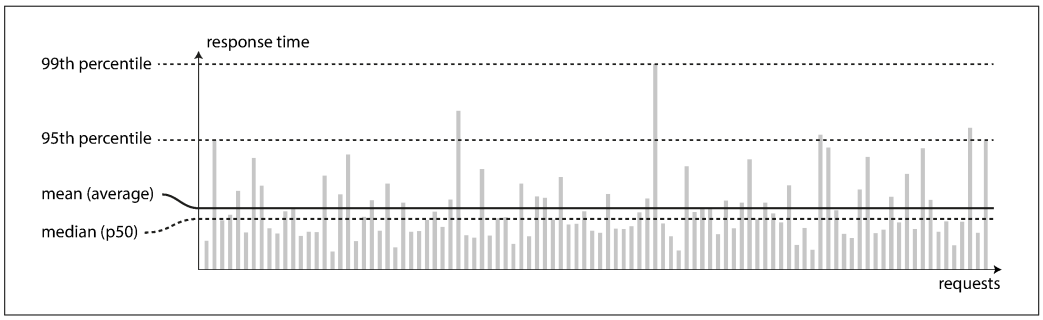


Figure 1-4. Illustrating mean and percentiles: response times for a sample of 100 requests to a service.

It's common to see the *average* response time of a service reported. (Strictly speaking, the term "average" doesn't refer to any particular formula, but in practice it is usually understood as the *arithmetic mean*: given n values, add up all the values, and divide by n.) However, the mean is not a very good metric if you want to know your "typical" response time, because it doesn't tell you how many users actually experienced that delay.

Usually it is better to use *percentiles*. If you take your list of response times and sort it from fastest to slowest, then the *median* is the halfway point: for example, if your

median response time is 200 ms, that means half your requests return in less than 200 ms, and half your requests take longer than that.

This makes the median a good metric if you want to know how long users typically have to wait: half of user requests are served in less than the median response time, and the other half take longer than the median. The median is also known as the *50th percentile*, and sometimes abbreviated as *p50*. Note that the median refers to a single request; if the user makes several requests (over the course of a session, or because several resources are included in a single page), the probability that at least one of them is slower than the median is much greater than 50%.

In order to figure out how bad your outliers are, you can look at higher percentiles: the *95th*, *99th*, and *99.9th* percentiles are common (abbreviated *p95*, *p99*, and *p999*). They are the response time thresholds at which 95%, 99%, or 99.9% of requests are faster than that particular threshold. For example, if the 95th percentile response time is 1.5 seconds, that means 95 out of 100 requests take less than 1.5 seconds, and 5 out of 100 requests take 1.5 seconds or more. This is illustrated in Figure 1-4.

High percentiles of response times, also known as *tail latencies*, are important because they directly affect users' experience of the service. For example, Amazon describes response time requirements for internal services in terms of the 99.9th percentile, even though it only affects 1 in 1,000 requests. This is because the customers with the slowest requests are often those who have the most data on their accounts because they have made many purchases—that is, they're the most valuable customers [19]. It's important to keep those customers happy by ensuring the website is fast for them: Amazon has also observed that a 100 ms increase in response time reduces sales by 1% [20], and others report that a 1-second slowdown reduces a customer satisfaction metric by 16% [21, 22].

On the other hand, optimizing the 99.99th percentile (the slowest 1 in 10,000 requests) was deemed too expensive and to not yield enough benefit for Amazon's purposes. Reducing response times at very high percentiles is difficult because they are easily affected by random events outside of your control, and the benefits are diminishing.

For example, percentiles are often used in *service level objectives* (SLOs) and *service level agreements* (SLAs), contracts that define the expected performance and availability of a service. An SLA may state that the service is considered to be up if it has a median response time of less than 200 ms and a 99th percentile under 1 s (if the response time is longer, it might as well be down), and the service may be required to be up at least 99.9% of the time. These metrics set expectations for clients of the service and allow customers to demand a refund if the SLA is not met.

Queueing delays often account for a large part of the response time at high percentiles. As a server can only process a small number of things in parallel (limited, for

example, by its number of CPU cores), it only takes a small number of slow requests to hold up the processing of subsequent requests—an effect sometimes known as *head-of-line blocking*. Even if those subsequent requests are fast to process on the server, the client will see a slow overall response time due to the time waiting for the prior request to complete. Due to this effect, it is important to measure response times on the client side.

When generating load artificially in order to test the scalability of a system, the load-generating client needs to keep sending requests independently of the response time. If the client waits for the previous request to complete before sending the next one, that behavior has the effect of artificially keeping the queues shorter in the test than they would be in reality, which skews the measurements [23].

Percentiles in Practice

High percentiles become especially important in backend services that are called multiple times as part of serving a single end-user request. Even if you make the calls in parallel, the end-user request still needs to wait for the slowest of the parallel calls to complete. It takes just one slow call to make the entire end-user request slow, as illustrated in Figure 1-5. Even if only a small percentage of backend calls are slow, the chance of getting a slow call increases if an end-user request requires multiple backend calls, and so a higher proportion of end-user requests end up being slow (an effect known as *tail latency amplification* [24]).

If you want to add response time percentiles to the monitoring dashboards for your services, you need to efficiently calculate them on an ongoing basis. For example, you may want to keep a rolling window of response times of requests in the last 10 minutes. Every minute, you calculate the median and various percentiles over the values in that window and plot those metrics on a graph.

The naïve implementation is to keep a list of response times for all requests within the time window and to sort that list every minute. If that is too inefficient for you, there are algorithms that can calculate a good approximation of percentiles at minimal CPU and memory cost, such as forward decay [25], t-digest [26], or HdrHistogram [27]. Beware that averaging percentiles, e.g., to reduce the time resolution or to combine data from several machines, is mathematically meaningless—the right way of aggregating response time data is to add the histograms [28].

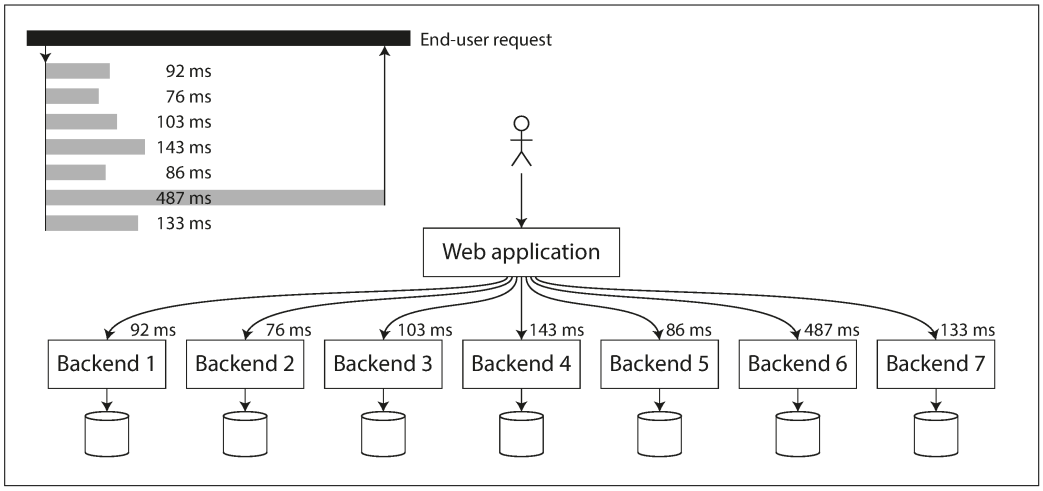


Figure 1-5. When several backend calls are needed to serve a request, it takes just a single slow backend request to slow down the entire end-user request.

Approaches for Coping with Load

Now that we have discussed the parameters for describing load and metrics for measuring performance, we can start discussing scalability in earnest: how do we maintain good performance even when our load parameters increase by some amount?

An architecture that is appropriate for one level of load is unlikely to cope with 10 times that load. If you are working on a fast-growing service, it is therefore likely that you will need to rethink your architecture on every order of magnitude load increase —or perhaps even more often than that.

People often talk of a dichotomy between *scaling up* (*vertical scaling*, moving to a more powerful machine) and *scaling out* (*horizontal scaling*, distributing the load across multiple smaller machines). Distributing load across multiple machines is also known as a *shared-nothing* architecture. A system that can run on a single machine is often simpler, but high-end machines can become very expensive, so very intensive workloads often can't avoid scaling out. In reality, good architectures usually involve a pragmatic mixture of approaches: for example, using several fairly powerful machines can still be simpler and cheaper than a large number of small virtual machines.

Some systems are *elastic*, meaning that they can automatically add computing resources when they detect a load increase, whereas other systems are scaled manually (a human analyzes the capacity and decides to add more machines to the system). An elastic system can be useful if load is highly unpredictable, but manually scaled systems are simpler and may have fewer operational surprises (see "Rebalancing Partitions" on page 209).

While distributing stateless services across multiple machines is fairly straightforward, taking stateful data systems from a single node to a distributed setup can introduce a lot of additional complexity. For this reason, common wisdom until recently was to keep your database on a single node (scale up) until scaling cost or high-availability requirements forced you to make it distributed.

As the tools and abstractions for distributed systems get better, this common wisdom may change, at least for some kinds of applications. It is conceivable that distributed data systems will become the default in the future, even for use cases that don't handle large volumes of data or traffic. Over the course of the rest of this book we will cover many kinds of distributed data systems, and discuss how they fare not just in terms of scalability, but also ease of use and maintainability.

The architecture of systems that operate at large scale is usually highly specific to the application—there is no such thing as a generic, one-size-fits-all scalable architecture (informally known as *magic scaling sauce*). The problem may be the volume of reads, the volume of writes, the volume of data to store, the complexity of the data, the response time requirements, the access patterns, or (usually) some mixture of all of these plus many more issues.

For example, a system that is designed to handle 100,000 requests per second, each 1 kB in size, looks very different from a system that is designed for 3 requests per minute, each 2 GB in size—even though the two systems have the same data throughput.

An architecture that scales well for a particular application is built around assumptions of which operations will be common and which will be rare—the load parameters. If those assumptions turn out to be wrong, the engineering effort for scaling is at best wasted, and at worst counterproductive. In an early-stage startup or an unproven product it's usually more important to be able to iterate quickly on product features than it is to scale to some hypothetical future load.

Even though they are specific to a particular application, scalable architectures are nevertheless usually built from general-purpose building blocks, arranged in familiar patterns. In this book we discuss those building blocks and patterns.

Maintainability

It is well known that the majority of the cost of software is not in its initial development, but in its ongoing maintenance—fixing bugs, keeping its systems operational, investigating failures, adapting it to new platforms, modifying it for new use cases, repaying technical debt, and adding new features.

Yet, unfortunately, many people working on software systems dislike maintenance of so-called *legacy* systems—perhaps it involves fixing other people's mistakes, or work-

ing with platforms that are now outdated, or systems that were forced to do things they were never intended for. Every legacy system is unpleasant in its own way, and so it is difficult to give general recommendations for dealing with them.

However, we can and should design software in such a way that it will hopefully minimize pain during maintenance, and thus avoid creating legacy software ourselves. To this end, we will pay particular attention to three design principles for software systems:

Operability
: Make it easy for operations teams to keep the system running smoothly.

Simplicity
: Make it easy for new engineers to understand the system, by removing as much complexity as possible from the system. (Note this is not the same as simplicity of the user interface.)

Evolvability
: Make it easy for engineers to make changes to the system in the future, adapting it for unanticipated use cases as requirements change. Also known as *extensibility*, *modifiability*, or *plasticity*.

As previously with reliability and scalability, there are no easy solutions for achieving these goals. Rather, we will try to think about systems with operability, simplicity, and evolvability in mind.

Operability: Making Life Easy for Operations

It has been suggested that "good operations can often work around the limitations of bad (or incomplete) software, but good software cannot run reliably with bad operations" [12]. While some aspects of operations can and should be automated, it is still up to humans to set up that automation in the first place and to make sure it's working correctly.

Operations teams are vital to keeping a software system running smoothly. A good operations team typically is responsible for the following, and more [29]:

- Monitoring the health of the system and quickly restoring service if it goes into a bad state
- Tracking down the cause of problems, such as system failures or degraded performance
- Keeping software and platforms up to date, including security patches
- Keeping tabs on how different systems affect each other, so that a problematic change can be avoided before it causes damage

- Anticipating future problems and solving them before they occur (e.g., capacity planning)
- Establishing good practices and tools for deployment, configuration management, and more
- Performing complex maintenance tasks, such as moving an application from one platform to another
- Maintaining the security of the system as configuration changes are made
- Defining processes that make operations predictable and help keep the production environment stable
- Preserving the organization's knowledge about the system, even as individual people come and go

Good operability means making routine tasks easy, allowing the operations team to focus their efforts on high-value activities. Data systems can do various things to make routine tasks easy, including:

- Providing visibility into the runtime behavior and internals of the system, with good monitoring
- Providing good support for automation and integration with standard tools
- Avoiding dependency on individual machines (allowing machines to be taken down for maintenance while the system as a whole continues running uninterrupted)
- Providing good documentation and an easy-to-understand operational model ("If I do X, Y will happen")
- Providing good default behavior, but also giving administrators the freedom to override defaults when needed
- Self-healing where appropriate, but also giving administrators manual control over the system state when needed
- Exhibiting predictable behavior, minimizing surprises

Simplicity: Managing Complexity

Small software projects can have delightfully simple and expressive code, but as projects get larger, they often become very complex and difficult to understand. This complexity slows down everyone who needs to work on the system, further increasing the cost of maintenance. A software project mired in complexity is sometimes described as a *big ball of mud* [30].

There are various possible symptoms of complexity: explosion of the state space, tight coupling of modules, tangled dependencies, inconsistent naming and terminology, hacks aimed at solving performance problems, special-casing to work around issues elsewhere, and many more. Much has been said on this topic already [31, 32, 33].

When complexity makes maintenance hard, budgets and schedules are often overrun. In complex software, there is also a greater risk of introducing bugs when making a change: when the system is harder for developers to understand and reason about, hidden assumptions, unintended consequences, and unexpected interactions are more easily overlooked. Conversely, reducing complexity greatly improves the maintainability of software, and thus simplicity should be a key goal for the systems we build.

Making a system simpler does not necessarily mean reducing its functionality; it can also mean removing *accidental* complexity. Moseley and Marks [32] define complexity as accidental if it is not inherent in the problem that the software solves (as seen by the users) but arises only from the implementation.

One of the best tools we have for removing accidental complexity is *abstraction*. A good abstraction can hide a great deal of implementation detail behind a clean, simple-to-understand façade. A good abstraction can also be used for a wide range of different applications. Not only is this reuse more efficient than reimplementing a similar thing multiple times, but it also leads to higher-quality software, as quality improvements in the abstracted component benefit all applications that use it.

For example, high-level programming languages are abstractions that hide machine code, CPU registers, and syscalls. SQL is an abstraction that hides complex on-disk and in-memory data structures, concurrent requests from other clients, and inconsistencies after crashes. Of course, when programming in a high-level language, we are still using machine code; we are just not using it *directly*, because the programming language abstraction saves us from having to think about it.

However, finding good abstractions is very hard. In the field of distributed systems, although there are many good algorithms, it is much less clear how we should be packaging them into abstractions that help us keep the complexity of the system at a manageable level.

Throughout this book, we will keep our eyes open for good abstractions that allow us to extract parts of a large system into well-defined, reusable components.

Evolvability: Making Change Easy

It's extremely unlikely that your system's requirements will remain unchanged forever. They are much more likely to be in constant flux: you learn new facts, previously unanticipated use cases emerge, business priorities change, users request new

features, new platforms replace old platforms, legal or regulatory requirements change, growth of the system forces architectural changes, etc.

In terms of organizational processes, *Agile* working patterns provide a framework for adapting to change. The Agile community has also developed technical tools and patterns that are helpful when developing software in a frequently changing environment, such as test-driven development (TDD) and refactoring.

Most discussions of these Agile techniques focus on a fairly small, local scale (a couple of source code files within the same application). In this book, we search for ways of increasing agility on the level of a larger data system, perhaps consisting of several different applications or services with different characteristics. For example, how would you "refactor" Twitter's architecture for assembling home timelines ("Describing Load" on page 11) from approach 1 to approach 2?

The ease with which you can modify a data system, and adapt it to changing requirements, is closely linked to its simplicity and its abstractions: simple and easy-to-understand systems are usually easier to modify than complex ones. But since this is such an important idea, we will use a different word to refer to agility on a data system level: *evolvability* [34].

Summary

In this chapter, we have explored some fundamental ways of thinking about data-intensive applications. These principles will guide us through the rest of the book, where we dive into deep technical detail.

An application has to meet various requirements in order to be useful. There are *functional requirements* (what it should do, such as allowing data to be stored, retrieved, searched, and processed in various ways), and some *nonfunctional requirements* (general properties like security, reliability, compliance, scalability, compatibility, and maintainability). In this chapter we discussed reliability, scalability, and maintainability in detail.

Reliability means making systems work correctly, even when faults occur. Faults can be in hardware (typically random and uncorrelated), software (bugs are typically systematic and hard to deal with), and humans (who inevitably make mistakes from time to time). Fault-tolerance techniques can hide certain types of faults from the end user.

Scalability means having strategies for keeping performance good, even when load increases. In order to discuss scalability, we first need ways of describing load and performance quantitatively. We briefly looked at Twitter's home timelines as an example of describing load, and response time percentiles as a way of measuring per-

formance. In a scalable system, you can add processing capacity in order to remain reliable under high load.

Maintainability has many facets, but in essence it's about making life better for the engineering and operations teams who need to work with the system. Good abstractions can help reduce complexity and make the system easier to modify and adapt for new use cases. Good operability means having good visibility into the system's health, and having effective ways of managing it.

There is unfortunately no easy fix for making applications reliable, scalable, or maintainable. However, there are certain patterns and techniques that keep reappearing in different kinds of applications. In the next few chapters we will take a look at some examples of data systems and analyze how they work toward those goals.

Later in the book, in Part III, we will look at patterns for systems that consist of several components working together, such as the one in Figure 1-1.

References

[1] Michael Stonebraker and Uğur Çetintemel: "'One Size Fits All': An Idea Whose Time Has Come and Gone," at *21st International Conference on Data Engineering* (ICDE), April 2005.

[2] Walter L. Heimerdinger and Charles B. Weinstock: "A Conceptual Framework for System Fault Tolerance," Technical Report CMU/SEI-92-TR-033, Software Engineering Institute, Carnegie Mellon University, October 1992.

[3] Ding Yuan, Yu Luo, Xin Zhuang, et al.: "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems," at *11th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2014.

[4] Yury Izrailevsky and Ariel Tseitlin: "The Netflix Simian Army," *techblog.netflix.com*, July 19, 2011.

[5] Daniel Ford, François Labelle, Florentina I. Popovici, et al.: "Availability in Globally Distributed Storage Systems," at *9th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2010.

[6] Brian Beach: "Hard Drive Reliability Update – Sep 2014," *backblaze.com*, September 23, 2014.

[7] Laurie Voss: "AWS: The Good, the Bad and the Ugly," *blog.awe.sm*, December 18, 2012.

[8] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, et al.: "What Bugs Live in the Cloud?," at *5th ACM Symposium on Cloud Computing* (SoCC), November 2014. doi:10.1145/2670979.2670986

[9] Nelson Minar: "Leap Second Crashes Half the Internet," *somebits.com*, July 3, 2012.

[10] Amazon Web Services: "Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region," *aws.amazon.com*, April 29, 2011.

[11] Richard I. Cook: "How Complex Systems Fail," Cognitive Technologies Laboratory, April 2000.

[12] Jay Kreps: "Getting Real About Distributed System Reliability," *blog.empathybox.com*, March 19, 2012.

[13] David Oppenheimer, Archana Ganapathi, and David A. Patterson: "Why Do Internet Services Fail, and What Can Be Done About It?," at *4th USENIX Symposium on Internet Technologies and Systems* (USITS), March 2003.

[14] Nathan Marz: "Principles of Software Engineering, Part 1," *nathanmarz.com*, April 2, 2013.

[15] Michael Jurewitz: "The Human Impact of Bugs," *jury.me*, March 15, 2013.

[16] Raffi Krikorian: "Timelines at Scale," at *QCon San Francisco*, November 2012.

[17] Martin Fowler: *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002. ISBN: 978-0-321-12742-6

[18] Kelly Sommers: "After all that run around, what caused 500ms disk latency even when we replaced physical server?" *twitter.com*, November 13, 2014.

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al.: "Dynamo: Amazon's Highly Available Key-Value Store," at *21st ACM Symposium on Operating Systems Principles* (SOSP), October 2007.

[20] Greg Linden: "Make Data Useful," slides from presentation at Stanford University Data Mining class (CS345), December 2006.

[21] Tammy Everts: "The Real Cost of Slow Time vs Downtime," *webperformancetoday.com*, November 12, 2014.

[22] Jake Brutlag: "Speed Matters for Google Web Search," *googleresearch.blogspot.co.uk*, June 22, 2009.

[23] Tyler Treat: "Everything You Know About Latency Is Wrong," *bravenewgeek.com*, December 12, 2015.

[24] Jeffrey Dean and Luiz André Barroso: "The Tail at Scale," *Communications of the ACM*, volume 56, number 2, pages 74–80, February 2013. doi:10.1145/2408776.2408794

[25] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu: "Forward Decay: A Practical Time Decay Model for Streaming Systems," at *25th IEEE International Conference on Data Engineering* (ICDE), March 2009.

[26] Ted Dunning and Otmar Ertl: "Computing Extremely Accurate Quantiles Using t-Digests," *github.com*, March 2014.

[27] Gil Tene: "HdrHistogram," *hdrhistogram.org*.

[28] Baron Schwartz: "Why Percentiles Don't Work the Way You Think," *vividcortex.com*, December 7, 2015.

[29] James Hamilton: "On Designing and Deploying Internet-Scale Services," at *21st Large Installation System Administration Conference* (LISA), November 2007.

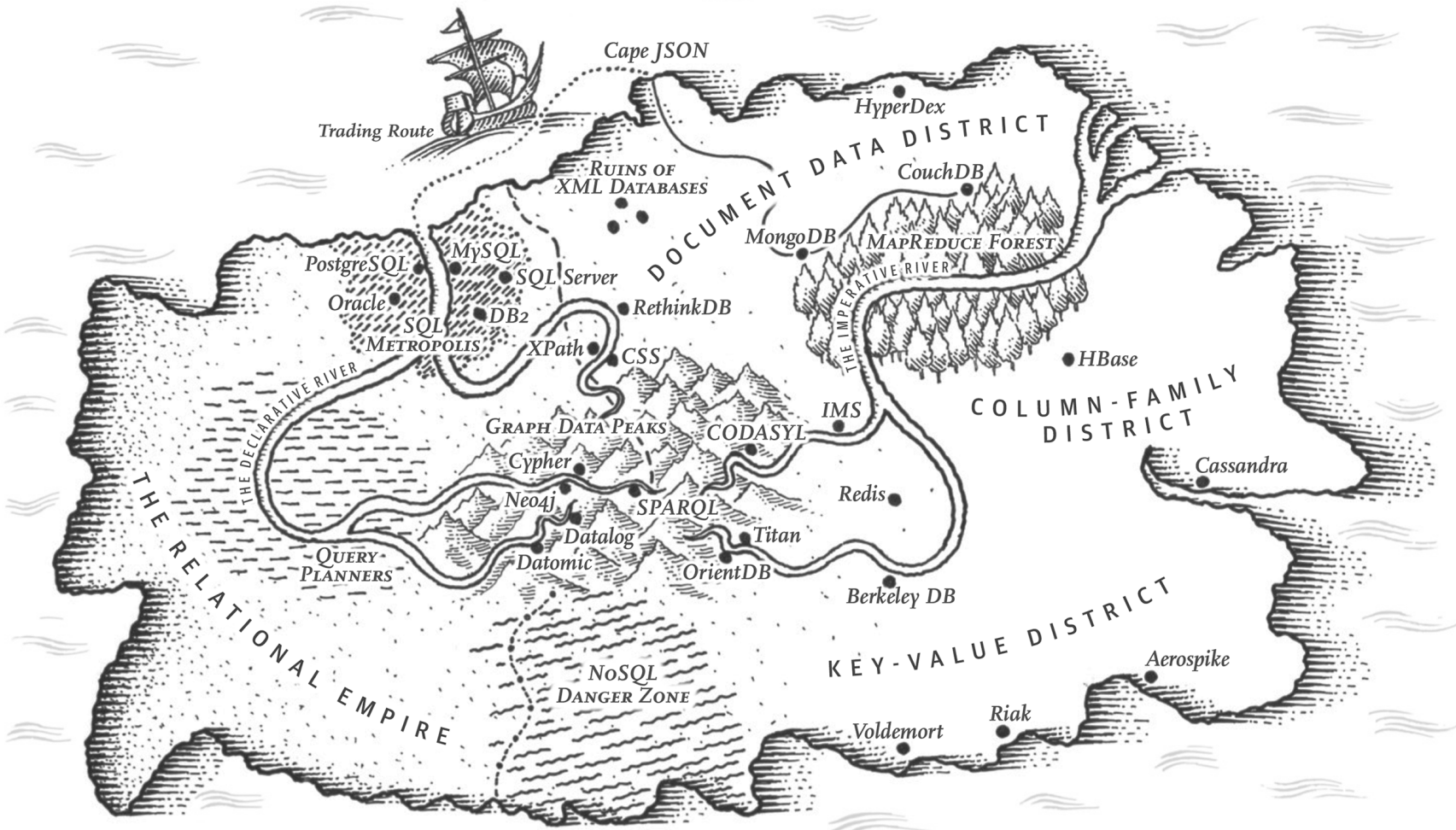
[30] Brian Foote and Joseph Yoder: "Big Ball of Mud," at *4th Conference on Pattern Languages of Programs* (PLoP), September 1997.

[31] Frederick P Brooks: "No Silver Bullet – Essence and Accident in Software Engineering," in *The Mythical Man-Month*, Anniversary edition, Addison-Wesley, 1995. ISBN: 978-0-201-83595-3

[32] Ben Moseley and Peter Marks: "Out of the Tar Pit," at *BCS Software Practice Advancement* (SPA), 2006.

[33] Rich Hickey: "Simple Made Easy," at *Strange Loop*, September 2011.

[34] Hongyu Pei Breivold, Ivica Crnkovic, and Peter J. Eriksson: "Analyzing Software Evolvability," at *32nd Annual IEEE International Computer Software and Applications Conference* (COMPSAC), July 2008. doi:10.1109/COMPSAC.2008.50


Cape JSON
Trading Route
HyperDex
DOCUMENT DATA DISTRICT
Ruins of XML Databases
CouchDB
MongoDB
MapReduce Forest
THE IMPERATIVE RIVER
PostgreSQL
MySQL
SQL Server
Oracle
DB2
RethinkDB
SQL Metropolis
XPath
CSS
THE DECLARATIVE RIVER
HBase
IMS
COLUMN-FAMILY DISTRICT
Graph Data Peaks
CODASYL
Cypher
Cassandra
Redis
Neo4j
SPARQL
Datalog
Titan
Query Planners
Datomic
OrientDB
Berkeley DB
THE RELATIONAL EMPIRE
KEY-VALUE DISTRICT
NoSQL Danger Zone
Aerospike
Riak
Voldemort

CHAPTER 2
Data Models and Query Languages

> *The limits of my language mean the limits of my world.*
>
> —Ludwig Wittgenstein, *Tractatus Logico-Philosophicus* (1922)

Data models are perhaps the most important part of developing software, because they have such a profound effect: not only on how the software is written, but also on how we *think about the problem* that we are solving.

Most applications are built by layering one data model on top of another. For each layer, the key question is: how is it *represented* in terms of the next-lower layer? For example:

1. As an application developer, you look at the real world (in which there are people, organizations, goods, actions, money flows, sensors, etc.) and model it in terms of objects or data structures, and APIs that manipulate those data structures. Those structures are often specific to your application.
2. When you want to store those data structures, you express them in terms of a general-purpose data model, such as JSON or XML documents, tables in a relational database, or a graph model.
3. The engineers who built your database software decided on a way of representing that JSON/XML/relational/graph data in terms of bytes in memory, on disk, or on a network. The representation may allow the data to be queried, searched, manipulated, and processed in various ways.
4. On yet lower levels, hardware engineers have figured out how to represent bytes in terms of electrical currents, pulses of light, magnetic fields, and more.

In a complex application there may be more intermediary levels, such as APIs built upon APIs, but the basic idea is still the same: each layer hides the complexity of the layers below it by providing a clean data model. These abstractions allow different

groups of people—for example, the engineers at the database vendor and the application developers using their database—to work together effectively.

There are many different kinds of data models, and every data model embodies assumptions about how it is going to be used. Some kinds of usage are easy and some are not supported; some operations are fast and some perform badly; some data transformations feel natural and some are awkward.

It can take a lot of effort to master just one data model (think how many books there are on relational data modeling). Building software is hard enough, even when working with just one data model and without worrying about its inner workings. But since the data model has such a profound effect on what the software above it can and can't do, it's important to choose one that is appropriate to the application.

In this chapter we will look at a range of general-purpose data models for data storage and querying (point 2 in the preceding list). In particular, we will compare the relational model, the document model, and a few graph-based data models. We will also look at various query languages and compare their use cases. In Chapter 3 we will discuss how storage engines work; that is, how these data models are actually implemented (point 3 in the list).

Relational Model Versus Document Model

The best-known data model today is probably that of SQL, based on the relational model proposed by Edgar Codd in 1970 [1]: data is organized into *relations* (called *tables* in SQL), where each relation is an unordered collection of *tuples* (*rows* in SQL).

The relational model was a theoretical proposal, and many people at the time doubted whether it could be implemented efficiently. However, by the mid-1980s, relational database management systems (RDBMSes) and SQL had become the tools of choice for most people who needed to store and query data with some kind of regular structure. The dominance of relational databases has lasted around 25–30 years—an eternity in computing history.

The roots of relational databases lie in *business data processing*, which was performed on mainframe computers in the 1960s and '70s. The use cases appear mundane from today's perspective: typically *transaction processing* (entering sales or banking transactions, airline reservations, stock-keeping in warehouses) and *batch processing* (customer invoicing, payroll, reporting).

Other databases at that time forced application developers to think a lot about the internal representation of the data in the database. The goal of the relational model was to hide that implementation detail behind a cleaner interface.

Over the years, there have been many competing approaches to data storage and querying. In the 1970s and early 1980s, the *network model* and the *hierarchical model*

were the main alternatives, but the relational model came to dominate them. Object databases came and went again in the late 1980s and early 1990s. XML databases appeared in the early 2000s, but have only seen niche adoption. Each competitor to the relational model generated a lot of hype in its time, but it never lasted [2].

As computers became vastly more powerful and networked, they started being used for increasingly diverse purposes. And remarkably, relational databases turned out to generalize very well, beyond their original scope of business data processing, to a broad variety of use cases. Much of what you see on the web today is still powered by relational databases, be it online publishing, discussion, social networking, ecommerce, games, software-as-a-service productivity applications, or much more.

The Birth of NoSQL

Now, in the 2010s, *NoSQL* is the latest attempt to overthrow the relational model's dominance. The name "NoSQL" is unfortunate, since it doesn't actually refer to any particular technology—it was originally intended simply as a catchy Twitter hashtag for a meetup on open source, distributed, nonrelational databases in 2009 [3]. Nevertheless, the term struck a nerve and quickly spread through the web startup community and beyond. A number of interesting database systems are now associated with the #NoSQL hashtag, and it has been retroactively reinterpreted as *Not Only SQL* [4].

There are several driving forces behind the adoption of NoSQL databases, including:

- A need for greater scalability than relational databases can easily achieve, including very large datasets or very high write throughput
- A widespread preference for free and open source software over commercial database products
- Specialized query operations that are not well supported by the relational model
- Frustration with the restrictiveness of relational schemas, and a desire for a more dynamic and expressive data model [5]

Different applications have different requirements, and the best choice of technology for one use case may well be different from the best choice for another use case. It therefore seems likely that in the foreseeable future, relational databases will continue to be used alongside a broad variety of nonrelational datastores—an idea that is sometimes called *polyglot persistence* [3].

The Object-Relational Mismatch

Most application development today is done in object-oriented programming languages, which leads to a common criticism of the SQL data model: if data is stored in relational tables, an awkward translation layer is required between the objects in the

application code and the database model of tables, rows, and columns. The disconnect between the models is sometimes called an *impedance mismatch*.[i]

Object-relational mapping (ORM) frameworks like ActiveRecord and Hibernate reduce the amount of boilerplate code required for this translation layer, but they can't completely hide the differences between the two models.

For example, Figure 2-1 illustrates how a résumé (a LinkedIn profile) could be expressed in a relational schema. The profile as a whole can be identified by a unique identifier, `user_id`. Fields like `first_name` and `last_name` appear exactly once per user, so they can be modeled as columns on the `users` table. However, most people have had more than one job in their career (positions), and people may have varying numbers of periods of education and any number of pieces of contact information. There is a one-to-many relationship from the user to these items, which can be represented in various ways:

- In the traditional SQL model (prior to SQL:1999), the most common normalized representation is to put positions, education, and contact information in separate tables, with a foreign key reference to the `users` table, as in Figure 2-1.
- Later versions of the SQL standard added support for structured datatypes and XML data; this allowed multi-valued data to be stored within a single row, with support for querying and indexing inside those documents. These features are supported to varying degrees by Oracle, IBM DB2, MS SQL Server, and PostgreSQL [6, 7]. A JSON datatype is also supported by several databases, including IBM DB2, MySQL, and PostgreSQL [8].
- A third option is to encode jobs, education, and contact info as a JSON or XML document, store it on a text column in the database, and let the application interpret its structure and content. In this setup, you typically cannot use the database to query for values inside that encoded column.

i. A term borrowed from electronics. Every electric circuit has a certain impedance (resistance to alternating current) on its inputs and outputs. When you connect one circuit's output to another one's input, the power transfer across the connection is maximized if the output and input impedances of the two circuits match. An impedance mismatch can lead to signal reflections and other troubles.

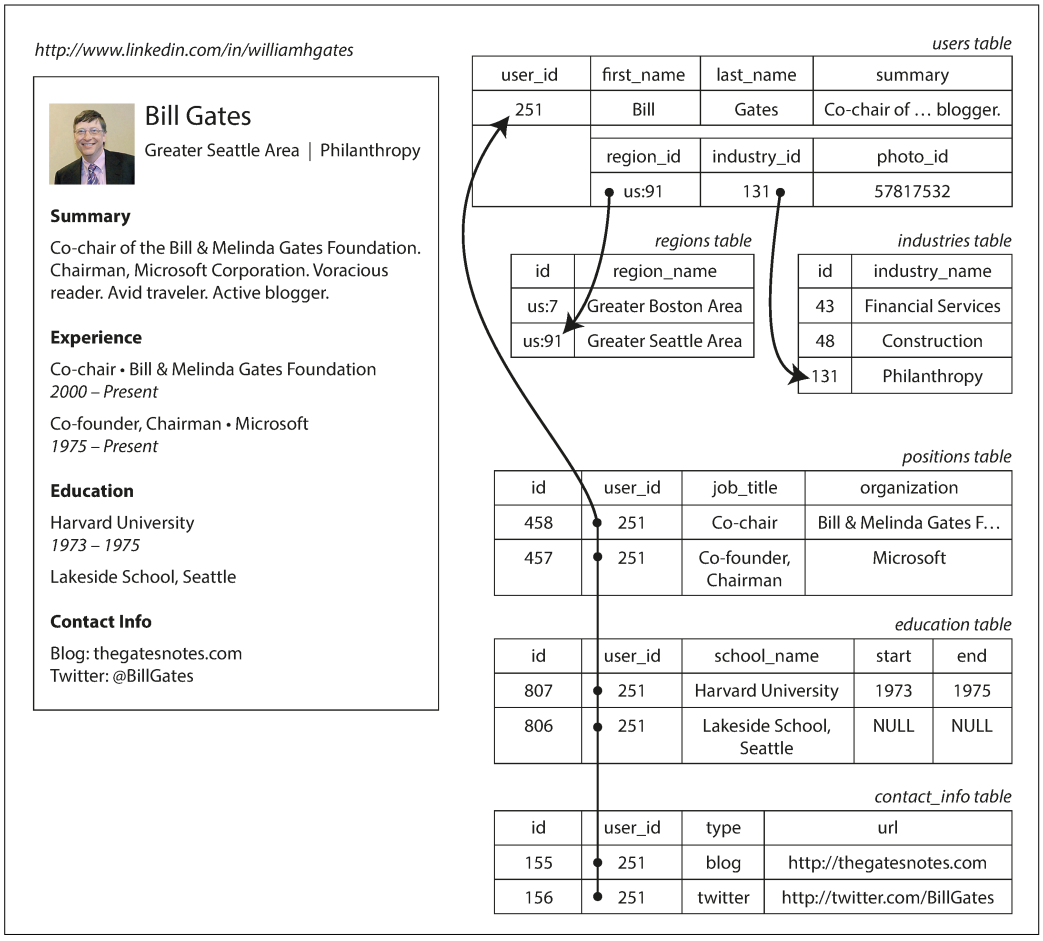


Figure 2-1. Representing a LinkedIn profile using a relational schema. Photo of Bill Gates courtesy of Wikimedia Commons, Ricardo Stuckert, Agência Brasil.

For a data structure like a résumé, which is mostly a self-contained *document*, a JSON representation can be quite appropriate: see Example 2-1. JSON has the appeal of being much simpler than XML. Document-oriented databases like MongoDB [9], RethinkDB [10], CouchDB [11], and Espresso [12] support this data model.

Example 2-1. Representing a LinkedIn profile as a JSON document

```
{
  "user_id":     251,
  "first_name":  "Bill",
  "last_name":   "Gates",
  "summary":     "Co-chair of the Bill & Melinda Gates... Active blogger.",
  "region_id":   "us:91",
  "industry_id": 131,
  "photo_url":   "/p/7/000/253/05b/308dd6e.jpg",
```

```
  "positions": [
    {"job_title": "Co-chair", "organization": "Bill & Melinda Gates Foundation"},
    {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}
  ],
  "education": [
    {"school_name": "Harvard University",       "start": 1973, "end": 1975},
    {"school_name": "Lakeside School, Seattle", "start": null, "end": null}
  ],
  "contact_info": {
    "blog":    "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}
```

Some developers feel that the JSON model reduces the impedance mismatch between the application code and the storage layer. However, as we shall see in Chapter 4, there are also problems with JSON as a data encoding format. The lack of a schema is often cited as an advantage; we will discuss this in "Schema flexibility in the document model" on page 39.

The JSON representation has better *locality* than the multi-table schema in Figure 2-1. If you want to fetch a profile in the relational example, you need to either perform multiple queries (query each table by `user_id`) or perform a messy multi-way join between the `users` table and its subordinate tables. In the JSON representation, all the relevant information is in one place, and one query is sufficient.

The one-to-many relationships from the user profile to the user's positions, educational history, and contact information imply a tree structure in the data, and the JSON representation makes this tree structure explicit (see Figure 2-2).

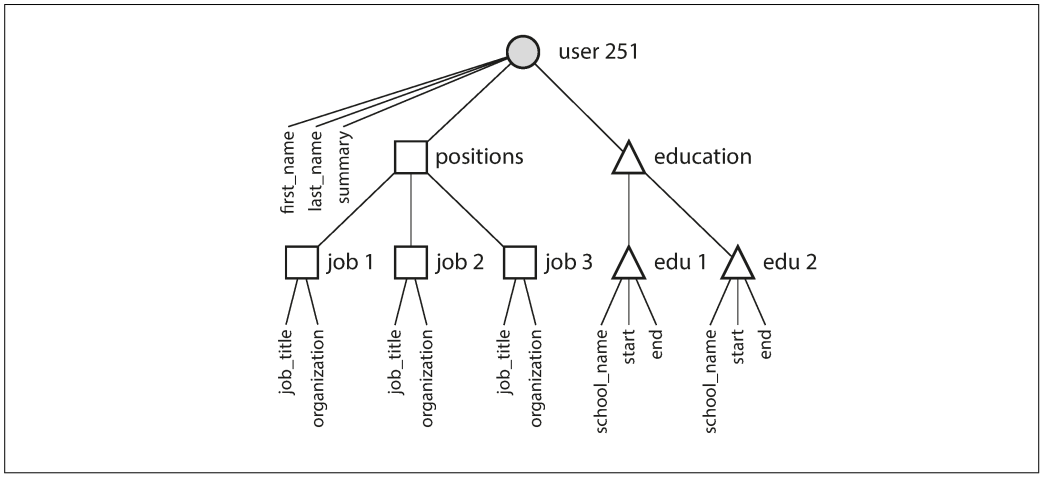


Figure 2-2. One-to-many relationships forming a tree structure.

Many-to-One and Many-to-Many Relationships

In Example 2-1 in the preceding section, `region_id` and `industry_id` are given as IDs, not as plain-text strings `"Greater Seattle Area"` and `"Philanthropy"`. Why?

If the user interface has free-text fields for entering the region and the industry, it makes sense to store them as plain-text strings. But there are advantages to having standardized lists of geographic regions and industries, and letting users choose from a drop-down list or autocompleter:

- Consistent style and spelling across profiles
- Avoiding ambiguity (e.g., if there are several cities with the same name)
- Ease of updating—the name is stored in only one place, so it is easy to update across the board if it ever needs to be changed (e.g., change of a city name due to political events)
- Localization support—when the site is translated into other languages, the standardized lists can be localized, so the region and industry can be displayed in the viewer's language
- Better search—e.g., a search for philanthropists in the state of Washington can match this profile, because the list of regions can encode the fact that Seattle is in Washington (which is not apparent from the string `"Greater Seattle Area"`)

Whether you store an ID or a text string is a question of duplication. When you use an ID, the information that is meaningful to humans (such as the word *Philanthropy*) is stored in only one place, and everything that refers to it uses an ID (which only has meaning within the database). When you store the text directly, you are duplicating the human-meaningful information in every record that uses it.

The advantage of using an ID is that because it has no meaning to humans, it never needs to change: the ID can remain the same, even if the information it identifies changes. Anything that is meaningful to humans may need to change sometime in the future—and if that information is duplicated, all the redundant copies need to be updated. That incurs write overheads, and risks inconsistencies (where some copies of the information are updated but others aren't). Removing such duplication is the key idea behind *normalization* in databases.[ii]

ii. Literature on the relational model distinguishes several different normal forms, but the distinctions are of little practical interest. As a rule of thumb, if you're duplicating values that could be stored in just one place, the schema is not normalized.

Database administrators and developers love to argue about normalization and denormalization, but we will suspend judgment for now. In Part III of this book we will return to this topic and explore systematic ways of dealing with caching, denormalization, and derived data.

Unfortunately, normalizing this data requires *many-to-one* relationships (many people live in one particular region, many people work in one particular industry), which don't fit nicely into the document model. In relational databases, it's normal to refer to rows in other tables by ID, because joins are easy. In document databases, joins are not needed for one-to-many tree structures, and support for joins is often weak.[iii]

If the database itself does not support joins, you have to emulate a join in application code by making multiple queries to the database. (In this case, the lists of regions and industries are probably small and slow-changing enough that the application can simply keep them in memory. But nevertheless, the work of making the join is shifted from the database to the application code.)

Moreover, even if the initial version of an application fits well in a join-free document model, data has a tendency of becoming more interconnected as features are added to applications. For example, consider some changes we could make to the résumé example:

Organizations and schools as entities
: In the previous description, `organization` (the company where the user worked) and `school_name` (where they studied) are just strings. Perhaps they should be references to entities instead? Then each organization, school, or university could have its own web page (with logo, news feed, etc.); each résumé could link to the organizations and schools that it mentions, and include their logos and other information (see Figure 2-3 for an example from LinkedIn).

Recommendations
: Say you want to add a new feature: one user can write a recommendation for another user. The recommendation is shown on the résumé of the user who was recommended, together with the name and photo of the user making the recommendation. If the recommender updates their photo, any recommendations they have written need to reflect the new photo. Therefore, the recommendation should have a reference to the author's profile.

iii. At the time of writing, joins are supported in RethinkDB, not supported in MongoDB, and only supported in predeclared views in CouchDB.

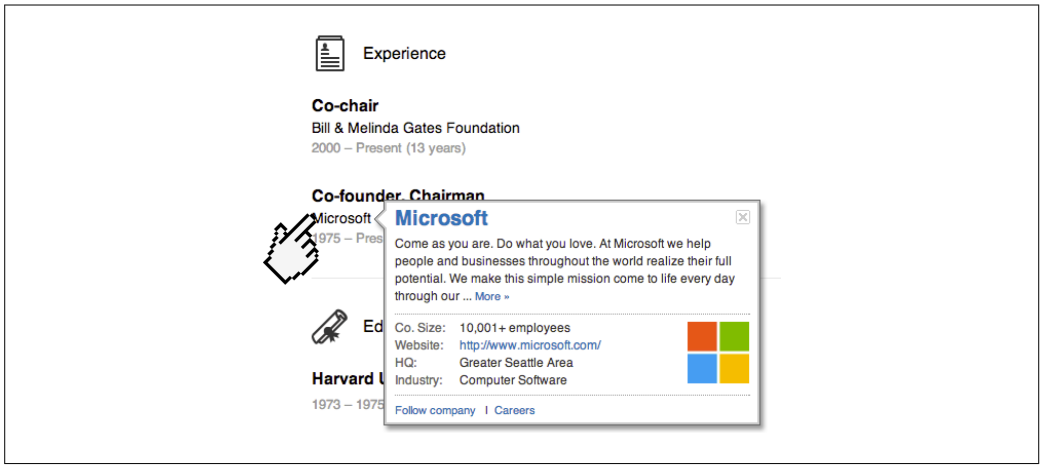


Figure 2-3. The company name is not just a string, but a link to a company entity. Screenshot of linkedin.com.

Figure 2-4 illustrates how these new features require many-to-many relationships. The data within each dotted rectangle can be grouped into one document, but the references to organizations, schools, and other users need to be represented as references, and require joins when queried.

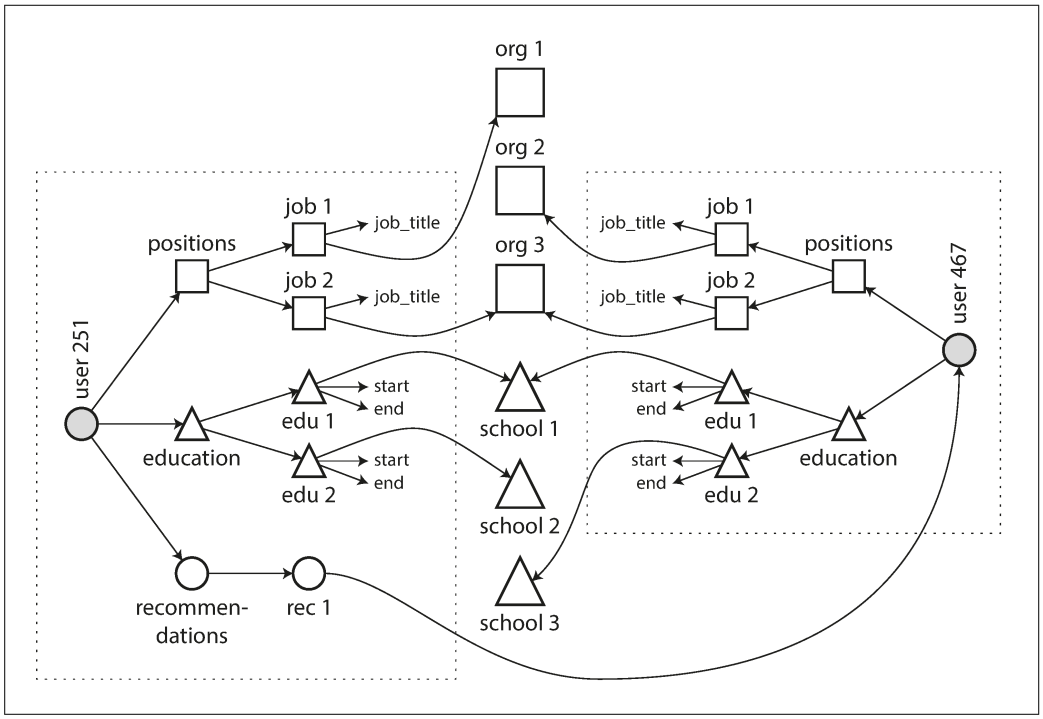


Figure 2-4. Extending résumés with many-to-many relationships.

Are Document Databases Repeating History?

While many-to-many relationships and joins are routinely used in relational databases, document databases and NoSQL reopened the debate on how best to represent such relationships in a database. This debate is much older than NoSQL—in fact, it goes back to the very earliest computerized database systems.

The most popular database for business data processing in the 1970s was IBM's *Information Management System* (IMS), originally developed for stock-keeping in the Apollo space program and first commercially released in 1968 [13]. It is still in use and maintained today, running on OS/390 on IBM mainframes [14].

The design of IMS used a fairly simple data model called the *hierarchical model*, which has some remarkable similarities to the JSON model used by document databases [2]. It represented all data as a tree of records nested within records, much like the JSON structure of Figure 2-2.

Like document databases, IMS worked well for one-to-many relationships, but it made many-to-many relationships difficult, and it didn't support joins. Developers had to decide whether to duplicate (denormalize) data or to manually resolve references from one record to another. These problems of the 1960s and '70s were very much like the problems that developers are running into with document databases today [15].

Various solutions were proposed to solve the limitations of the hierarchical model. The two most prominent were the *relational model* (which became SQL, and took over the world) and the *network model* (which initially had a large following but eventually faded into obscurity). The "great debate" between these two camps lasted for much of the 1970s [2].

Since the problem that the two models were solving is still so relevant today, it's worth briefly revisiting this debate in today's light.

The network model

The network model was standardized by a committee called the Conference on Data Systems Languages (CODASYL) and implemented by several different database vendors; it is also known as the *CODASYL model* [16].

The CODASYL model was a generalization of the hierarchical model. In the tree structure of the hierarchical model, every record has exactly one parent; in the network model, a record could have multiple parents. For example, there could be one record for the `"Greater Seattle Area"` region, and every user who lived in that region could be linked to it. This allowed many-to-one and many-to-many relationships to be modeled.

The links between records in the network model were not foreign keys, but more like pointers in a programming language (while still being stored on disk). The only way of accessing a record was to follow a path from a root record along these chains of links. This was called an *access path*.

In the simplest case, an access path could be like the traversal of a linked list: start at the head of the list, and look at one record at a time until you find the one you want. But in a world of many-to-many relationships, several different paths can lead to the same record, and a programmer working with the network model had to keep track of these different access paths in their head.

A query in CODASYL was performed by moving a cursor through the database by iterating over lists of records and following access paths. If a record had multiple parents (i.e., multiple incoming pointers from other records), the application code had to keep track of all the various relationships. Even CODASYL committee members admitted that this was like navigating around an *n*-dimensional data space [17].

Although manual access path selection was able to make the most efficient use of the very limited hardware capabilities in the 1970s (such as tape drives, whose seeks are extremely slow), the problem was that they made the code for querying and updating the database complicated and inflexible. With both the hierarchical and the network model, if you didn't have a path to the data you wanted, you were in a difficult situation. You could change the access paths, but then you had to go through a lot of handwritten database query code and rewrite it to handle the new access paths. It was difficult to make changes to an application's data model.

The relational model

What the relational model did, by contrast, was to lay out all the data in the open: a relation (table) is simply a collection of tuples (rows), and that's it. There are no labyrinthine nested structures, no complicated access paths to follow if you want to look at the data. You can read any or all of the rows in a table, selecting those that match an arbitrary condition. You can read a particular row by designating some columns as a key and matching on those. You can insert a new row into any table without worrying about foreign key relationships to and from other tables.[iv]

In a relational database, the query optimizer automatically decides which parts of the query to execute in which order, and which indexes to use. Those choices are effectively the "access path," but the big difference is that they are made automatically by

iv. Foreign key constraints allow you to restrict modifications, but such constraints are not required by the relational model. Even with constraints, joins on foreign keys are performed at query time, whereas in CODASYL, the join was effectively done at insert time.

the query optimizer, not by the application developer, so we rarely need to think about them.

If you want to query your data in new ways, you can just declare a new index, and queries will automatically use whichever indexes are most appropriate. You don't need to change your queries to take advantage of a new index. (See also "Query Languages for Data" on page 42.) The relational model thus made it much easier to add new features to applications.

Query optimizers for relational databases are complicated beasts, and they have consumed many years of research and development effort [18]. But a key insight of the relational model was this: you only need to build a query optimizer once, and then all applications that use the database can benefit from it. If you don't have a query optimizer, it's easier to handcode the access paths for a particular query than to write a general-purpose optimizer—but the general-purpose solution wins in the long run.

Comparison to document databases

Document databases reverted back to the hierarchical model in one aspect: storing nested records (one-to-many relationships, like `positions`, `education`, and `contact_info` in Figure 2-1) within their parent record rather than in a separate table.

However, when it comes to representing many-to-one and many-to-many relationships, relational and document databases are not fundamentally different: in both cases, the related item is referenced by a unique identifier, which is called a *foreign key* in the relational model and a *document reference* in the document model [9]. That identifier is resolved at read time by using a join or follow-up queries. To date, document databases have not followed the path of CODASYL.

Relational Versus Document Databases Today

There are many differences to consider when comparing relational databases to document databases, including their fault-tolerance properties (see Chapter 5) and handling of concurrency (see Chapter 7). In this chapter, we will concentrate only on the differences in the data model.

The main arguments in favor of the document data model are schema flexibility, better performance due to locality, and that for some applications it is closer to the data structures used by the application. The relational model counters by providing better support for joins, and many-to-one and many-to-many relationships.

Which data model leads to simpler application code?

If the data in your application has a document-like structure (i.e., a tree of one-to-many relationships, where typically the entire tree is loaded at once), then it's proba-

bly a good idea to use a document model. The relational technique of *shredding*—splitting a document-like structure into multiple tables (like `positions`, `education`, and `contact_info` in Figure 2-1)—can lead to cumbersome schemas and unnecessarily complicated application code.

The document model has limitations: for example, you cannot refer directly to a nested item within a document, but instead you need to say something like "the second item in the list of positions for user 251" (much like an access path in the hierarchical model). However, as long as documents are not too deeply nested, that is not usually a problem.

The poor support for joins in document databases may or may not be a problem, depending on the application. For example, many-to-many relationships may never be needed in an analytics application that uses a document database to record which events occurred at which time [19].

However, if your application does use many-to-many relationships, the document model becomes less appealing. It's possible to reduce the need for joins by denormalizing, but then the application code needs to do additional work to keep the denormalized data consistent. Joins can be emulated in application code by making multiple requests to the database, but that also moves complexity into the application and is usually slower than a join performed by specialized code inside the database. In such cases, using a document model can lead to significantly more complex application code and worse performance [15].

It's not possible to say in general which data model leads to simpler application code; it depends on the kinds of relationships that exist between data items. For highly interconnected data, the document model is awkward, the relational model is acceptable, and graph models (see "Graph-Like Data Models" on page 49) are the most natural.

Schema flexibility in the document model

Most document databases, and the JSON support in relational databases, do not enforce any schema on the data in documents. XML support in relational databases usually comes with optional schema validation. No schema means that arbitrary keys and values can be added to a document, and when reading, clients have no guarantees as to what fields the documents may contain.

Document databases are sometimes called *schemaless*, but that's misleading, as the code that reads the data usually assumes some kind of structure—i.e., there is an implicit schema, but it is not enforced by the database [20]. A more accurate term is *schema-on-read* (the structure of the data is implicit, and only interpreted when the data is read), in contrast with *schema-on-write* (the traditional approach of relational

databases, where the schema is explicit and the database ensures all written data conforms to it) [21].

Schema-on-read is similar to dynamic (runtime) type checking in programming languages, whereas schema-on-write is similar to static (compile-time) type checking. Just as the advocates of static and dynamic type checking have big debates about their relative merits [22], enforcement of schemas in database is a contentious topic, and in general there's no right or wrong answer.

The difference between the approaches is particularly noticeable in situations where an application wants to change the format of its data. For example, say you are currently storing each user's full name in one field, and you instead want to store the first name and last name separately [23]. In a document database, you would just start writing new documents with the new fields and have code in the application that handles the case when old documents are read. For example:

```
if (user && user.name && !user.first_name) {
    // Documents written before Dec 8, 2013 don't have first_name
    user.first_name = user.name.split(" ")[0];
}
```

On the other hand, in a "statically typed" database schema, you would typically perform a *migration* along the lines of:

```
ALTER TABLE users ADD COLUMN first_name text;
UPDATE users SET first_name = split_part(name, ' ', 1);      -- PostgreSQL
UPDATE users SET first_name = substring_index(name, ' ', 1);      -- MySQL
```

Schema changes have a bad reputation of being slow and requiring downtime. This reputation is not entirely deserved: most relational database systems execute the `ALTER TABLE` statement in a few milliseconds. MySQL is a notable exception—it copies the entire table on `ALTER TABLE`, which can mean minutes or even hours of downtime when altering a large table—although various tools exist to work around this limitation [24, 25, 26].

Running the `UPDATE` statement on a large table is likely to be slow on any database, since every row needs to be rewritten. If that is not acceptable, the application can leave `first_name` set to its default of `NULL` and fill it in at read time, like it would with a document database.

The schema-on-read approach is advantageous if the items in the collection don't all have the same structure for some reason (i.e., the data is heterogeneous)—for example, because:

- There are many different types of objects, and it is not practical to put each type of object in its own table.

- The structure of the data is determined by external systems over which you have no control and which may change at any time.

In situations like these, a schema may hurt more than it helps, and schemaless documents can be a much more natural data model. But in cases where all records are expected to have the same structure, schemas are a useful mechanism for documenting and enforcing that structure. We will discuss schemas and schema evolution in more detail in Chapter 4.

Data locality for queries

A document is usually stored as a single continuous string, encoded as JSON, XML, or a binary variant thereof (such as MongoDB's BSON). If your application often needs to access the entire document (for example, to render it on a web page), there is a performance advantage to this *storage locality*. If data is split across multiple tables, like in Figure 2-1, multiple index lookups are required to retrieve it all, which may require more disk seeks and take more time.

The locality advantage only applies if you need large parts of the document at the same time. The database typically needs to load the entire document, even if you access only a small portion of it, which can be wasteful on large documents. On updates to a document, the entire document usually needs to be rewritten—only modifications that don't change the encoded size of a document can easily be performed in place [19]. For these reasons, it is generally recommended that you keep documents fairly small and avoid writes that increase the size of a document [9]. These performance limitations significantly reduce the set of situations in which document databases are useful.

It's worth pointing out that the idea of grouping related data together for locality is not limited to the document model. For example, Google's Spanner database offers the same locality properties in a relational data model, by allowing the schema to declare that a table's rows should be interleaved (nested) within a parent table [27]. Oracle allows the same, using a feature called *multi-table index cluster tables* [28]. The *column-family* concept in the Bigtable data model (used in Cassandra and HBase) has a similar purpose of managing locality [29].

We will also see more on locality in Chapter 3.

Convergence of document and relational databases

Most relational database systems (other than MySQL) have supported XML since the mid-2000s. This includes functions to make local modifications to XML documents and the ability to index and query inside XML documents, which allows applications to use data models very similar to what they would do when using a document database.

PostgreSQL since version 9.3 [8], MySQL since version 5.7, and IBM DB2 since version 10.5 [30] also have a similar level of support for JSON documents. Given the popularity of JSON for web APIs, it is likely that other relational databases will follow in their footsteps and add JSON support.

On the document database side, RethinkDB supports relational-like joins in its query language, and some MongoDB drivers automatically resolve database references (effectively performing a client-side join, although this is likely to be slower than a join performed in the database since it requires additional network round-trips and is less optimized).

It seems that relational and document databases are becoming more similar over time, and that is a good thing: the data models complement each other.[v] If a database is able to handle document-like data and also perform relational queries on it, applications can use the combination of features that best fits their needs.

A hybrid of the relational and document models is a good route for databases to take in the future.

Query Languages for Data

When the relational model was introduced, it included a new way of querying data: SQL is a *declarative* query language, whereas IMS and CODASYL queried the database using *imperative* code. What does that mean?

Many commonly used programming languages are imperative. For example, if you have a list of animal species, you might write something like this to return only the sharks in the list:

```
function getSharks() {
    var sharks = [];
    for (var i = 0; i < animals.length; i++) {
        if (animals[i].family === "Sharks") {
            sharks.push(animals[i]);
        }
    }
    return sharks;
}
```

In the relational algebra, you would instead write:

$$\text{sharks} = \sigma_{\text{family} = \text{“Sharks”}}(\text{animals})$$

v. Codd's original description of the relational model [1] actually allowed something quite similar to JSON documents within a relational schema. He called it *nonsimple domains*. The idea was that a value in a row doesn't have to just be a primitive datatype like a number or a string, but could also be a nested relation (table)—so you can have an arbitrarily nested tree structure as a value, much like the JSON or XML support that was added to SQL over 30 years later.

where σ (the Greek letter sigma) is the selection operator, returning only those animals that match the condition *family* = *"Sharks"*.

When SQL was defined, it followed the structure of the relational algebra fairly closely:

```
SELECT * FROM animals WHERE family = 'Sharks';
```

An imperative language tells the computer to perform certain operations in a certain order. You can imagine stepping through the code line by line, evaluating conditions, updating variables, and deciding whether to go around the loop one more time.

In a declarative query language, like SQL or relational algebra, you just specify the pattern of the data you want—what conditions the results must meet, and how you want the data to be transformed (e.g., sorted, grouped, and aggregated)—but not *how* to achieve that goal. It is up to the database system's query optimizer to decide which indexes and which join methods to use, and in which order to execute various parts of the query.

A declarative query language is attractive because it is typically more concise and easier to work with than an imperative API. But more importantly, it also hides implementation details of the database engine, which makes it possible for the database system to introduce performance improvements without requiring any changes to queries.

For example, in the imperative code shown at the beginning of this section, the list of animals appears in a particular order. If the database wants to reclaim unused disk space behind the scenes, it might need to move records around, changing the order in which the animals appear. Can the database do that safely, without breaking queries?

The SQL example doesn't guarantee any particular ordering, and so it doesn't mind if the order changes. But if the query is written as imperative code, the database can never be sure whether the code is relying on the ordering or not. The fact that SQL is more limited in functionality gives the database much more room for automatic optimizations.

Finally, declarative languages often lend themselves to parallel execution. Today, CPUs are getting faster by adding more cores, not by running at significantly higher clock speeds than before [31]. Imperative code is very hard to parallelize across multiple cores and multiple machines, because it specifies instructions that must be performed in a particular order. Declarative languages have a better chance of getting faster in parallel execution because they specify only the pattern of the results, not the algorithm that is used to determine the results. The database is free to use a parallel implementation of the query language, if appropriate [32].

Declarative Queries on the Web

The advantages of declarative query languages are not limited to just databases. To illustrate the point, let's compare declarative and imperative approaches in a completely different environment: a web browser.

Say you have a website about animals in the ocean. The user is currently viewing the page on sharks, so you mark the navigation item "Sharks" as currently selected, like this:

```
<ul>
    <li class="selected"> ❶
        <p>Sharks</p> ❷
        <ul>
            <li>Great White Shark</li>
            <li>Tiger Shark</li>
            <li>Hammerhead Shark</li>
        </ul>
    </li>
    <li>
        <p>Whales</p>
        <ul>
            <li>Blue Whale</li>
            <li>Humpback Whale</li>
            <li>Fin Whale</li>
        </ul>
    </li>
</ul>
```

❶ The selected item is marked with the CSS class `"selected"`.

❷ `<p>Sharks</p>` is the title of the currently selected page.

Now say you want the title of the currently selected page to have a blue background, so that it is visually highlighted. This is easy, using CSS:

```
li.selected > p {
    background-color: blue;
}
```

Here the CSS selector `li.selected > p` declares the pattern of elements to which we want to apply the blue style: namely, all `<p>` elements whose direct parent is an `<li>` element with a CSS class of `selected`. The element `<p>Sharks</p>` in the example matches this pattern, but `<p>Whales</p>` does not match because its `<li>` parent lacks `class="selected"`.

If you were using XSL instead of CSS, you could do something similar:

```
<xsl:template match="li[@class='selected']/p">
    <fo:block background-color="blue">
        <xsl:apply-templates/>
    </fo:block>
</xsl:template>
```

Here, the XPath expression `li[@class='selected']/p` is equivalent to the CSS selector `li.selected > p` in the previous example. What CSS and XSL have in common is that they are both *declarative* languages for specifying the styling of a document.

Imagine what life would be like if you had to use an imperative approach. In JavaScript, using the core Document Object Model (DOM) API, the result might look something like this:

```
var liElements = document.getElementsByTagName("li");
for (var i = 0; i < liElements.length; i++) {
    if (liElements[i].className === "selected") {
        var children = liElements[i].childNodes;
        for (var j = 0; j < children.length; j++) {
            var child = children[j];
            if (child.nodeType === Node.ELEMENT_NODE && child.tagName === "P") {
                child.setAttribute("style", "background-color: blue");
            }
        }
    }
}
```

This JavaScript imperatively sets the element `<p>Sharks</p>` to have a blue background, but the code is awful. Not only is it much longer and harder to understand than the CSS and XSL equivalents, but it also has some serious problems:

- If the `selected` class is removed (e.g., because the user clicks a different page), the blue color won't be removed, even if the code is rerun—and so the item will remain highlighted until the entire page is reloaded. With CSS, the browser automatically detects when the `li.selected > p` rule no longer applies and removes the blue background as soon as the `selected` class is removed.
- If you want to take advantage of a new API, such as `document.getElementsByClassName("selected")` or even `document.evaluate()`—which may improve performance—you have to rewrite the code. On the other hand, browser vendors can improve the performance of CSS and XPath without breaking compatibility.

In a web browser, using declarative CSS styling is much better than manipulating styles imperatively in JavaScript. Similarly, in databases, declarative query languages like SQL turned out to be much better than imperative query APIs.[vi]

MapReduce Querying

MapReduce is a programming model for processing large amounts of data in bulk across many machines, popularized by Google [33]. A limited form of MapReduce is supported by some NoSQL datastores, including MongoDB and CouchDB, as a mechanism for performing read-only queries across many documents.

MapReduce in general is described in more detail in Chapter 10. For now, we'll just briefly discuss MongoDB's use of the model.

MapReduce is neither a declarative query language nor a fully imperative query API, but somewhere in between: the logic of the query is expressed with snippets of code, which are called repeatedly by the processing framework. It is based on the `map` (also known as `collect`) and `reduce` (also known as `fold` or `inject`) functions that exist in many functional programming languages.

To give an example, imagine you are a marine biologist, and you add an observation record to your database every time you see animals in the ocean. Now you want to generate a report saying how many sharks you have sighted per month.

In PostgreSQL you might express that query like this:

```
SELECT date_trunc('month', observation_timestamp) AS observation_month, ❶
       sum(num_animals) AS total_animals
FROM observations
WHERE family = 'Sharks'
GROUP BY observation_month;
```

❶ The `date_trunc('month', timestamp)` function determines the calendar month containing `timestamp`, and returns another timestamp representing the beginning of that month. In other words, it rounds a timestamp down to the nearest month.

This query first filters the observations to only show species in the `Sharks` family, then groups the observations by the calendar month in which they occurred, and finally adds up the number of animals seen in all observations in that month.

The same can be expressed with MongoDB's MapReduce feature as follows:

vi. IMS and CODASYL both used imperative query APIs. Applications typically used COBOL code to iterate over records in the database, one record at a time [2, 16].

```
db.observations.mapReduce(
    function map() { ❷
        var year  = this.observationTimestamp.getFullYear();
        var month = this.observationTimestamp.getMonth() + 1;
        emit(year + "-" + month, this.numAnimals); ❸
    },
    function reduce(key, values) { ❹
        return Array.sum(values); ❺
    },
    {
        query: { family: "Sharks" }, ❶
        out: "monthlySharkReport" ❻
    }
);
```

❶ The filter to consider only shark species can be specified declaratively (this is a MongoDB-specific extension to MapReduce).

❷ The JavaScript function `map` is called once for every document that matches `query`, with `this` set to the document object.

❸ The `map` function emits a key (a string consisting of year and month, such as `"2013-12"` or `"2014-1"`) and a value (the number of animals in that observation).

❹ The key-value pairs emitted by `map` are grouped by key. For all key-value pairs with the same key (i.e., the same month and year), the `reduce` function is called once.

❺ The `reduce` function adds up the number of animals from all observations in a particular month.

❻ The final output is written to the collection `monthlySharkReport`.

For example, say the `observations` collection contains these two documents:

```
{
    observationTimestamp: Date.parse("Mon, 25 Dec 1995 12:34:56 GMT"),
    family:     "Sharks",
    species:    "Carcharodon carcharias",
    numAnimals: 3
}
{
    observationTimestamp: Date.parse("Tue, 12 Dec 1995 16:17:18 GMT"),
    family:     "Sharks",
    species:    "Carcharias taurus",
    numAnimals: 4
}
```

The `map` function would be called once for each document, resulting in `emit("1995-12", 3)` and `emit("1995-12", 4)`. Subsequently, the `reduce` function would be called with `reduce("1995-12", [3, 4])`, returning `7`.

The `map` and `reduce` functions are somewhat restricted in what they are allowed to do. They must be *pure* functions, which means they only use the data that is passed to them as input, they cannot perform additional database queries, and they must not have any side effects. These restrictions allow the database to run the functions anywhere, in any order, and rerun them on failure. However, they are nevertheless powerful: they can parse strings, call library functions, perform calculations, and more.

MapReduce is a fairly low-level programming model for distributed execution on a cluster of machines. Higher-level query languages like SQL can be implemented as a pipeline of MapReduce operations (see Chapter 10), but there are also many distributed implementations of SQL that don't use MapReduce. Note there is nothing in SQL that constrains it to running on a single machine, and MapReduce doesn't have a monopoly on distributed query execution.

Being able to use JavaScript code in the middle of a query is a great feature for advanced queries, but it's not limited to MapReduce—some SQL databases can be extended with JavaScript functions too [34].

A usability problem with MapReduce is that you have to write two carefully coordinated JavaScript functions, which is often harder than writing a single query. Moreover, a declarative query language offers more opportunities for a query optimizer to improve the performance of a query. For these reasons, MongoDB 2.2 added support for a declarative query language called the *aggregation pipeline* [9]. In this language, the same shark-counting query looks like this:

```
db.observations.aggregate([
    { $match: { family: "Sharks" } },
    { $group: {
        _id: {
            year:  { $year:  "$observationTimestamp" },
            month: { $month: "$observationTimestamp" }
        },
        totalAnimals: { $sum: "$numAnimals" }
    } }
]);
```

The aggregation pipeline language is similar in expressiveness to a subset of SQL, but it uses a JSON-based syntax rather than SQL's English-sentence-style syntax; the difference is perhaps a matter of taste. The moral of the story is that a NoSQL system may find itself accidentally reinventing SQL, albeit in disguise.

Graph-Like Data Models

We saw earlier that many-to-many relationships are an important distinguishing feature between different data models. If your application has mostly one-to-many relationships (tree-structured data) or no relationships between records, the document model is appropriate.

But what if many-to-many relationships are very common in your data? The relational model can handle simple cases of many-to-many relationships, but as the connections within your data become more complex, it becomes more natural to start modeling your data as a graph.

A graph consists of two kinds of objects: *vertices* (also known as *nodes* or *entities*) and *edges* (also known as *relationships* or *arcs*). Many kinds of data can be modeled as a graph. Typical examples include:

Social graphs
: Vertices are people, and edges indicate which people know each other.

The web graph
: Vertices are web pages, and edges indicate HTML links to other pages.

Road or rail networks
: Vertices are junctions, and edges represent the roads or railway lines between them.

Well-known algorithms can operate on these graphs: for example, car navigation systems search for the shortest path between two points in a road network, and PageRank can be used on the web graph to determine the popularity of a web page and thus its ranking in search results.

In the examples just given, all the vertices in a graph represent the same kind of thing (people, web pages, or road junctions, respectively). However, graphs are not limited to such *homogeneous* data: an equally powerful use of graphs is to provide a consistent way of storing completely different types of objects in a single datastore. For example, Facebook maintains a single graph with many different types of vertices and edges: vertices represent people, locations, events, checkins, and comments made by users; edges indicate which people are friends with each other, which checkin happened in which location, who commented on which post, who attended which event, and so on [35].

In this section we will use the example shown in Figure 2-5. It could be taken from a social network or a genealogical database: it shows two people, Lucy from Idaho and Alain from Beaune, France. They are married and living in London.

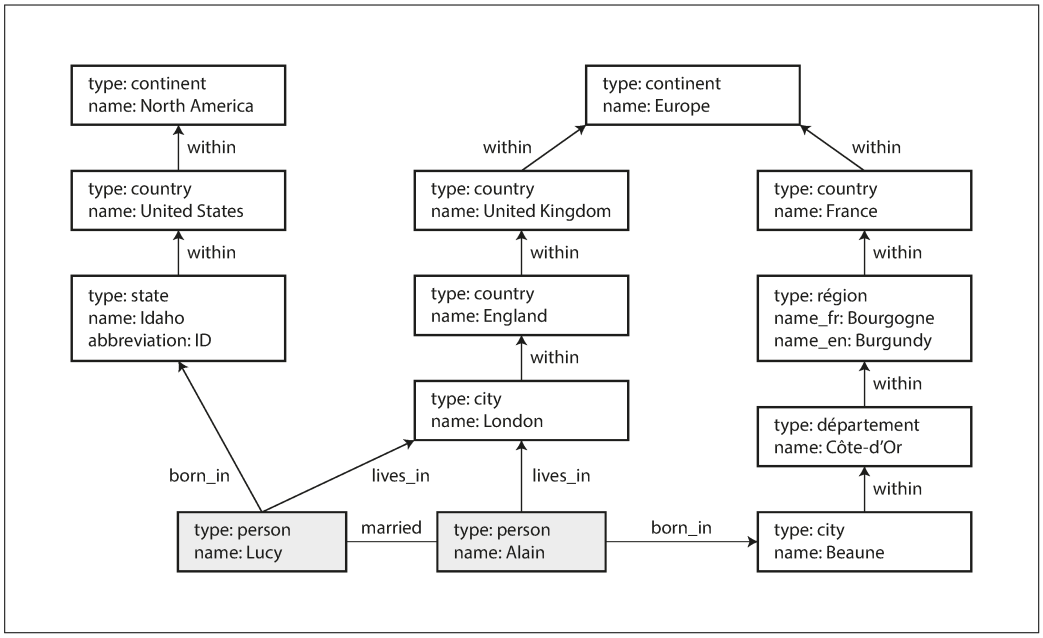


Figure 2-5. Example of graph-structured data (boxes represent vertices, arrows represent edges).

There are several different, but related, ways of structuring and querying data in graphs. In this section we will discuss the *property graph* model (implemented by Neo4j, Titan, and InfiniteGraph) and the *triple-store* model (implemented by Datomic, AllegroGraph, and others). We will look at three declarative query languages for graphs: Cypher, SPARQL, and Datalog. Besides these, there are also imperative graph query languages such as Gremlin [36] and graph processing frameworks like Pregel (see Chapter 10).

Property Graphs

In the property graph model, each vertex consists of:

- A unique identifier
- A set of outgoing edges
- A set of incoming edges
- A collection of properties (key-value pairs)

Each edge consists of:

- A unique identifier
- The vertex at which the edge starts (the *tail vertex*)

- The vertex at which the edge ends (the *head vertex*)
- A label to describe the kind of relationship between the two vertices
- A collection of properties (key-value pairs)

You can think of a graph store as consisting of two relational tables, one for vertices and one for edges, as shown in Example 2-2 (this schema uses the PostgreSQL `json` datatype to store the properties of each vertex or edge). The head and tail vertex are stored for each edge; if you want the set of incoming or outgoing edges for a vertex, you can query the `edges` table by `head_vertex` or `tail_vertex`, respectively.

Example 2-2. Representing a property graph using a relational schema

```
CREATE TABLE vertices (
    vertex_id   integer PRIMARY KEY,
    properties  json
);

CREATE TABLE edges (
    edge_id     integer PRIMARY KEY,
    tail_vertex integer REFERENCES vertices (vertex_id),
    head_vertex integer REFERENCES vertices (vertex_id),
    label       text,
    properties  json
);

CREATE INDEX edges_tails ON edges (tail_vertex);
CREATE INDEX edges_heads ON edges (head_vertex);
```

Some important aspects of this model are:

1. Any vertex can have an edge connecting it with any other vertex. There is no schema that restricts which kinds of things can or cannot be associated.
2. Given any vertex, you can efficiently find both its incoming and its outgoing edges, and thus *traverse* the graph—i.e., follow a path through a chain of vertices—both forward and backward. (That's why Example 2-2 has indexes on both the `tail_vertex` and `head_vertex` columns.)
3. By using different labels for different kinds of relationships, you can store several different kinds of information in a single graph, while still maintaining a clean data model.

Those features give graphs a great deal of flexibility for data modeling, as illustrated in Figure 2-5. The figure shows a few things that would be difficult to express in a traditional relational schema, such as different kinds of regional structures in different countries (France has *départements* and *régions*, whereas the US has *counties* and *states*), quirks of history such as a country within a country (ignoring for now the

intricacies of sovereign states and nations), and varying granularity of data (Lucy's current residence is specified as a city, whereas her place of birth is specified only at the level of a state).

You could imagine extending the graph to also include many other facts about Lucy and Alain, or other people. For instance, you could use it to indicate any food allergies they have (by introducing a vertex for each allergen, and an edge between a person and an allergen to indicate an allergy), and link the allergens with a set of vertices that show which foods contain which substances. Then you could write a query to find out what is safe for each person to eat. Graphs are good for evolvability: as you add features to your application, a graph can easily be extended to accommodate changes in your application's data structures.

The Cypher Query Language

Cypher is a declarative query language for property graphs, created for the Neo4j graph database [37]. (It is named after a character in the movie *The Matrix* and is not related to ciphers in cryptography [38].)

Example 2-3 shows the Cypher query to insert the lefthand portion of Figure 2-5 into a graph database. The rest of the graph can be added similarly and is omitted for readability. Each vertex is given a symbolic name like `USA` or `Idaho`, and other parts of the query can use those names to create edges between the vertices, using an arrow notation: `(Idaho) -[:WITHIN]-> (USA)` creates an edge labeled `WITHIN`, with `Idaho` as the tail node and `USA` as the head node.

Example 2-3. A subset of the data in Figure 2-5, represented as a Cypher query

```
CREATE
  (NAmerica:Location {name:'North America', type:'continent'}),
  (USA:Location      {name:'United States', type:'country'  }),
  (Idaho:Location    {name:'Idaho',         type:'state'    }),
  (Lucy:Person       {name:'Lucy' }),
  (Idaho) -[:WITHIN]->  (USA)  -[:WITHIN]-> (NAmerica),
  (Lucy)  -[:BORN_IN]-> (Idaho)
```

When all the vertices and edges of Figure 2-5 are added to the database, we can start asking interesting questions: for example, *find the names of all the people who emigrated from the United States to Europe*. To be more precise, here we want to find all the vertices that have a `BORN_IN` edge to a location within the US, and also a `LIVING_IN` edge to a location within Europe, and return the `name` property of each of those vertices.

Example 2-4 shows how to express that query in Cypher. The same arrow notation is used in a `MATCH` clause to find patterns in the graph: `(person) -[:BORN_IN]-> ()`

matches any two vertices that are related by an edge labeled `BORN_IN`. The tail vertex of that edge is bound to the variable `person`, and the head vertex is left unnamed.

Example 2-4. Cypher query to find people who emigrated from the US to Europe

```
MATCH
  (person) -[:BORN_IN]->  () -[:WITHIN*0..]-> (us:Location {name:'United States'}),
  (person) -[:LIVES_IN]-> () -[:WITHIN*0..]-> (eu:Location {name:'Europe'})
RETURN person.name
```

The query can be read as follows:

> Find any vertex (call it `person`) that meets *both* of the following conditions:
>
> 1. `person` has an outgoing `BORN_IN` edge to some vertex. From that vertex, you can follow a chain of outgoing `WITHIN` edges until eventually you reach a vertex of type `Location`, whose `name` property is equal to `"United States"`.
> 2. That same `person` vertex also has an outgoing `LIVES_IN` edge. Following that edge, and then a chain of outgoing `WITHIN` edges, you eventually reach a vertex of type `Location`, whose `name` property is equal to `"Europe"`.
>
> For each such `person` vertex, return the `name` property.

There are several possible ways of executing the query. The description given here suggests that you start by scanning all the people in the database, examine each person's birthplace and residence, and return only those people who meet the criteria.

But equivalently, you could start with the two `Location` vertices and work backward. If there is an index on the `name` property, you can probably efficiently find the two vertices representing the US and Europe. Then you can proceed to find all locations (states, regions, cities, etc.) in the US and Europe respectively by following all incoming `WITHIN` edges. Finally, you can look for people who can be found through an incoming `BORN_IN` or `LIVES_IN` edge at one of the location vertices.

As is typical for a declarative query language, you don't need to specify such execution details when writing the query: the query optimizer automatically chooses the strategy that is predicted to be the most efficient, so you can get on with writing the rest of your application.

Graph Queries in SQL

Example 2-2 suggested that graph data can be represented in a relational database. But if we put graph data in a relational structure, can we also query it using SQL?

The answer is yes, but with some difficulty. In a relational database, you usually know in advance which joins you need in your query. In a graph query, you may need to

traverse a variable number of edges before you find the vertex you're looking for—that is, the number of joins is not fixed in advance.

In our example, that happens in the `() -[:WITHIN*0..]-> ()` rule in the Cypher query. A person's `LIVES_IN` edge may point at any kind of location: a street, a city, a district, a region, a state, etc. A city may be `WITHIN` a region, a region `WITHIN` a state, a state `WITHIN` a country, etc. The `LIVES_IN` edge may point directly at the location vertex you're looking for, or it may be several levels removed in the location hierarchy.

In Cypher, `:WITHIN*0..` expresses that fact very concisely: it means "follow a `WITHIN` edge, zero or more times." It is like the `*` operator in a regular expression.

Since SQL:1999, this idea of variable-length traversal paths in a query can be expressed using something called *recursive common table expressions* (the `WITH RECURSIVE` syntax). Example 2-5 shows the same query—finding the names of people who emigrated from the US to Europe—expressed in SQL using this technique (supported in PostgreSQL, IBM DB2, Oracle, and SQL Server). However, the syntax is very clumsy in comparison to Cypher.

Example 2-5. The same query as Example 2-4, expressed in SQL using recursive common table expressions

```
WITH RECURSIVE

  -- in_usa is the set of vertex IDs of all locations within the United States
  in_usa(vertex_id) AS (
      SELECT vertex_id FROM vertices WHERE properties->>'name' = 'United States' ❶
    UNION
      SELECT edges.tail_vertex FROM edges ❷
        JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
        WHERE edges.label = 'within'
  ),

  -- in_europe is the set of vertex IDs of all locations within Europe
  in_europe(vertex_id) AS (
      SELECT vertex_id FROM vertices WHERE properties->>'name' = 'Europe' ❸
    UNION
      SELECT edges.tail_vertex FROM edges
        JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
        WHERE edges.label = 'within'
  ),

  -- born_in_usa is the set of vertex IDs of all people born in the US
  born_in_usa(vertex_id) AS ( ❹
    SELECT edges.tail_vertex FROM edges
      JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
      WHERE edges.label = 'born_in'
  ),
```

```
  -- lives_in_europe is the set of vertex IDs of all people living in Europe
  lives_in_europe(vertex_id) AS ( ❺
    SELECT edges.tail_vertex FROM edges
      JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
      WHERE edges.label = 'lives_in'
  )

SELECT vertices.properties->>'name'
FROM vertices
-- join to find those people who were both born in the US *and* live in Europe
JOIN born_in_usa     ON vertices.vertex_id = born_in_usa.vertex_id ❻
JOIN lives_in_europe ON vertices.vertex_id = lives_in_europe.vertex_id;
```

❶ First find the vertex whose `name` property has the value `"United States"`, and make it the first element of the set of vertices `in_usa`.

❷ Follow all incoming `within` edges from vertices in the set `in_usa`, and add them to the same set, until all incoming `within` edges have been visited.

❸ Do the same starting with the vertex whose `name` property has the value `"Europe"`, and build up the set of vertices `in_europe`.

❹ For each of the vertices in the set `in_usa`, follow incoming `born_in` edges to find people who were born in some place within the United States.

❺ Similarly, for each of the vertices in the set `in_europe`, follow incoming `lives_in` edges to find people who live in Europe.

❻ Finally, intersect the set of people born in the USA with the set of people living in Europe, by joining them.

If the same query can be written in 4 lines in one query language but requires 29 lines in another, that just shows that different data models are designed to satisfy different use cases. It's important to pick a data model that is suitable for your application.

Triple-Stores and SPARQL

The triple-store model is mostly equivalent to the property graph model, using different words to describe the same ideas. It is nevertheless worth discussing, because there are various tools and languages for triple-stores that can be valuable additions to your toolbox for building applications.

In a triple-store, all information is stored in the form of very simple three-part statements: (*subject*, *predicate*, *object*). For example, in the triple (*Jim*, *likes*, *bananas*), *Jim* is the subject, *likes* is the predicate (verb), and *bananas* is the object.

The subject of a triple is equivalent to a vertex in a graph. The object is one of two things:

1. A value in a primitive datatype, such as a string or a number. In that case, the predicate and object of the triple are equivalent to the key and value of a property on the subject vertex. For example, (*lucy*, *age*, *33*) is like a vertex `lucy` with properties `{"age":33}`.
2. Another vertex in the graph. In that case, the predicate is an edge in the graph, the subject is the tail vertex, and the object is the head vertex. For example, in (*lucy*, *marriedTo*, *alain*) the subject and object *lucy* and *alain* are both vertices, and the predicate *marriedTo* is the label of the edge that connects them.

Example 2-6 shows the same data as in Example 2-3, written as triples in a format called *Turtle*, a subset of *Notation3* (*N3*) [39].

Example 2-6. A subset of the data in Figure 2-5, represented as Turtle triples

```
@prefix : <urn:example:>.
_:lucy     a       :Person.
_:lucy     :name   "Lucy".
_:lucy     :bornIn _:idaho.
_:idaho    a       :Location.
_:idaho    :name   "Idaho".
_:idaho    :type   "state".
_:idaho    :within _:usa.
_:usa      a       :Location.
_:usa      :name   "United States".
_:usa      :type   "country".
_:usa      :within _:namerica.
_:namerica a       :Location.
_:namerica :name   "North America".
_:namerica :type   "continent".
```

In this example, vertices of the graph are written as `_:someName`. The name doesn't mean anything outside of this file; it exists only because we otherwise wouldn't know which triples refer to the same vertex. When the predicate represents an edge, the object is a vertex, as in `_:idaho :within _:usa`. When the predicate is a property, the object is a string literal, as in `_:usa :name "United States"`.

It's quite repetitive to repeat the same subject over and over again, but fortunately you can use semicolons to say multiple things about the same subject. This makes the Turtle format quite nice and readable: see Example 2-7.

Example 2-7. A more concise way of writing the data in Example 2-6

```
@prefix : <urn:example:>.
_:lucy     a :Person;   :name "Lucy";          :bornIn _:idaho.
_:idaho    a :Location; :name "Idaho";         :type "state";   :within _:usa.
_:usa      a :Location; :name "United States"; :type "country"; :within _:namerica.
_:namerica a :Location; :name "North America"; :type "continent".
```

The semantic web

If you read more about triple-stores, you may get sucked into a maelstrom of articles written about the *semantic web*. The triple-store data model is completely independent of the semantic web—for example, Datomic [40] is a triple-store that does not claim to have anything to do with it.[vii] But since the two are so closely linked in many people's minds, we should discuss them briefly.

The semantic web is fundamentally a simple and reasonable idea: websites already publish information as text and pictures for humans to read, so why don't they also publish information as machine-readable data for computers to read? The *Resource Description Framework* (RDF) [41] was intended as a mechanism for different websites to publish data in a consistent format, allowing data from different websites to be automatically combined into a *web of data*—a kind of internet-wide "database of everything."

Unfortunately, the semantic web was overhyped in the early 2000s but so far hasn't shown any sign of being realized in practice, which has made many people cynical about it. It has also suffered from a dizzying plethora of acronyms, overly complex standards proposals, and hubris.

However, if you look past those failings, there is also a lot of good work that has come out of the semantic web project. Triples can be a good internal data model for applications, even if you have no interest in publishing RDF data on the semantic web.

The RDF data model

The Turtle language we used in Example 2-7 is a human-readable format for RDF data. Sometimes RDF is also written in an XML format, which does the same thing much more verbosely—see Example 2-8. Turtle/N3 is preferable as it is much easier on the eyes, and tools like Apache Jena [42] can automatically convert between different RDF formats if necessary.

vii. Technically, Datomic uses 5-tuples rather than triples; the two additional fields are metadata for versioning.

Example 2-8. The data of Example 2-7, expressed using RDF/XML syntax

```
<rdf:RDF xmlns="urn:example:"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

  <Location rdf:nodeID="idaho">
    <name>Idaho</name>
    <type>state</type>
    <within>
      <Location rdf:nodeID="usa">
        <name>United States</name>
        <type>country</type>
        <within>
          <Location rdf:nodeID="namerica">
            <name>North America</name>
            <type>continent</type>
          </Location>
        </within>
      </Location>
    </within>
  </Location>

  <Person rdf:nodeID="lucy">
    <name>Lucy</name>
    <bornIn rdf:nodeID="idaho"/>
  </Person>
</rdf:RDF>
```

RDF has a few quirks due to the fact that it is designed for internet-wide data exchange. The subject, predicate, and object of a triple are often URIs. For example, a predicate might be an URI such as `<http://my-company.com/namespace#within>` or `<http://my-company.com/namespace#lives_in>`, rather than just `WITHIN` or `LIVES_IN`. The reasoning behind this design is that you should be able to combine your data with someone else's data, and if they attach a different meaning to the word `within` or `lives_in`, you won't get a conflict because their predicates are actually `<http://other.org/foo#within>` and `<http://other.org/foo#lives_in>`.

The URL `<http://my-company.com/namespace>` doesn't necessarily need to resolve to anything—from RDF's point of view, it is simply a namespace. To avoid potential confusion with `http://` URLs, the examples in this section use non-resolvable URIs such as `urn:example:within`. Fortunately, you can just specify this prefix once at the top of the file, and then forget about it.

The SPARQL query language

SPARQL is a query language for triple-stores using the RDF data model [43]. (It is an acronym for *SPARQL Protocol and RDF Query Language*, pronounced "sparkle.") It predates Cypher, and since Cypher's pattern matching is borrowed from SPARQL, they look quite similar [37].

The same query as before—finding people who have moved from the US to Europe—is even more concise in SPARQL than it is in Cypher (see Example 2-9).

Example 2-9. The same query as Example 2-4, expressed in SPARQL

```
PREFIX : <urn:example:>

SELECT ?personName WHERE {
  ?person :name ?personName.
  ?person :bornIn  / :within* / :name "United States".
  ?person :livesIn / :within* / :name "Europe".
}
```

The structure is very similar. The following two expressions are equivalent (variables start with a question mark in SPARQL):

```
(person) -[:BORN_IN]-> () -[:WITHIN*0..]-> (location)   # Cypher

?person :bornIn / :within* ?location.                   # SPARQL
```

Because RDF doesn't distinguish between properties and edges but just uses predicates for both, you can use the same syntax for matching properties. In the following expression, the variable `usa` is bound to any vertex that has a `name` property whose value is the string `"United States"`:

```
(usa {name:'United States'})   # Cypher

?usa :name "United States".    # SPARQL
```

SPARQL is a nice query language—even if the semantic web never happens, it can be a powerful tool for applications to use internally.

Graph Databases Compared to the Network Model

In "Are Document Databases Repeating History?" on page 36 we discussed how CODASYL and the relational model competed to solve the problem of many-to-many relationships in IMS. At first glance, CODASYL's network model looks similar to the graph model. Are graph databases the second coming of CODASYL in disguise?

No. They differ in several important ways:

- In CODASYL, a database had a schema that specified which record type could be nested within which other record type. In a graph database, there is no such restriction: any vertex can have an edge to any other vertex. This gives much greater flexibility for applications to adapt to changing requirements.
- In CODASYL, the only way to reach a particular record was to traverse one of the access paths to it. In a graph database, you can refer directly to any vertex by its unique ID, or you can use an index to find vertices with a particular value.
- In CODASYL, the children of a record were an ordered set, so the database had to maintain that ordering (which had consequences for the storage layout) and applications that inserted new records into the database had to worry about the positions of the new records in these sets. In a graph database, vertices and edges are not ordered (you can only sort the results when making a query).
- In CODASYL, all queries were imperative, difficult to write and easily broken by changes in the schema. In a graph database, you can write your traversal in imperative code if you want to, but most graph databases also support high-level, declarative query languages such as Cypher or SPARQL.

The Foundation: Datalog

Datalog is a much older language than SPARQL or Cypher, having been studied extensively by academics in the 1980s [44, 45, 46]. It is less well known among software engineers, but it is nevertheless important, because it provides the foundation that later query languages build upon.

In practice, Datalog is used in a few data systems: for example, it is the query language of Datomic [40], and Cascalog [47] is a Datalog implementation for querying large datasets in Hadoop.[viii]

viii. Datomic and Cascalog use a Clojure S-expression syntax for Datalog. In the following examples we use a Prolog syntax, which is a little easier to read, but this makes no functional difference.

Datalog's data model is similar to the triple-store model, generalized a bit. Instead of writing a triple as (*subject*, *predicate*, *object*), we write it as *predicate*(*subject*, *object*). Example 2-10 shows how to write the data from our example in Datalog.

Example 2-10. A subset of the data in Figure 2-5, represented as Datalog facts

```
name(namerica, 'North America').
type(namerica, continent).

name(usa, 'United States').
type(usa, country).
within(usa, namerica).

name(idaho, 'Idaho').
type(idaho, state).
within(idaho, usa).

name(lucy, 'Lucy').
born_in(lucy, idaho).
```

Now that we have defined the data, we can write the same query as before, as shown in Example 2-11. It looks a bit different from the equivalent in Cypher or SPARQL, but don't let that put you off. Datalog is a subset of Prolog, which you might have seen before if you've studied computer science.

Example 2-11. The same query as Example 2-4, expressed in Datalog

```
within_recursive(Location, Name) :- name(Location, Name).     /* Rule 1 */

within_recursive(Location, Name) :- within(Location, Via),    /* Rule 2 */
                                    within_recursive(Via, Name).

migrated(Name, BornIn, LivingIn) :- name(Person, Name),       /* Rule 3 */
                                    born_in(Person, BornLoc),
                                    within_recursive(BornLoc, BornIn),
                                    lives_in(Person, LivingLoc),
                                    within_recursive(LivingLoc, LivingIn).

?- migrated(Who, 'United States', 'Europe').
/* Who = 'Lucy'. */
```

Cypher and SPARQL jump in right away with `SELECT`, but Datalog takes a small step at a time. We define *rules* that tell the database about new predicates: here, we define two new predicates, `within_recursive` and `migrated`. These predicates aren't triples stored in the database, but instead they are derived from data or from other rules. Rules can refer to other rules, just like functions can call other functions or recursively call themselves. Like this, complex queries can be built up a small piece at a time.

In rules, words that start with an uppercase letter are variables, and predicates are matched like in Cypher and SPARQL. For example, `name(Location, Name)` matches the triple `name(namerica, 'North America')` with variable bindings `Location = namerica` and `Name = 'North America'`.

A rule applies if the system can find a match for *all* predicates on the righthand side of the `:-` operator. When the rule applies, it's as though the lefthand side of the `:-` was added to the database (with variables replaced by the values they matched).

One possible way of applying the rules is thus:

1. `name(namerica, 'North America')` exists in the database, so rule 1 applies. It generates `within_recursive(namerica, 'North America')`.
2. `within(usa, namerica)` exists in the database and the previous step generated `within_recursive(namerica, 'North America')`, so rule 2 applies. It generates `within_recursive(usa, 'North America')`.
3. `within(idaho, usa)` exists in the database and the previous step generated `within_recursive(usa, 'North America')`, so rule 2 applies. It generates `within_recursive(idaho, 'North America')`.

By repeated application of rules 1 and 2, the `within_recursive` predicate can tell us all the locations in North America (or any other location name) contained in our database. This process is illustrated in Figure 2-6.

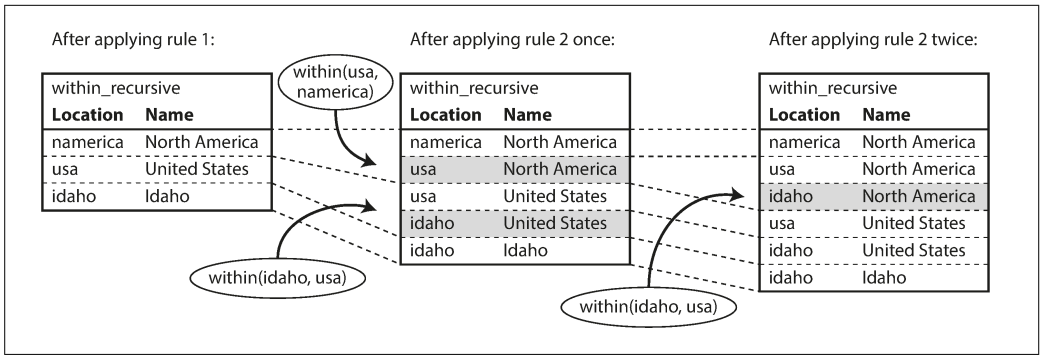


Figure 2-6. Determining that Idaho is in North America, using the Datalog rules from Example 2-11.

Now rule 3 can find people who were born in some location `BornIn` and live in some location `LivingIn`. By querying with `BornIn = 'United States'` and `LivingIn = 'Europe'`, and leaving the person as a variable `Who`, we ask the Datalog system to find out which values can appear for the variable `Who`. So, finally we get the same answer as in the earlier Cypher and SPARQL queries.

The Datalog approach requires a different kind of thinking to the other query languages discussed in this chapter, but it's a very powerful approach, because rules can be combined and reused in different queries. It's less convenient for simple one-off queries, but it can cope better if your data is complex.

Summary

Data models are a huge subject, and in this chapter we have taken a quick look at a broad variety of different models. We didn't have space to go into all the details of each model, but hopefully the overview has been enough to whet your appetite to find out more about the model that best fits your application's requirements.

Historically, data started out being represented as one big tree (the hierarchical model), but that wasn't good for representing many-to-many relationships, so the relational model was invented to solve that problem. More recently, developers found that some applications don't fit well in the relational model either. New nonrelational "NoSQL" datastores have diverged in two main directions:

1. *Document databases* target use cases where data comes in self-contained documents and relationships between one document and another are rare.
2. *Graph databases* go in the opposite direction, targeting use cases where anything is potentially related to everything.

All three models (document, relational, and graph) are widely used today, and each is good in its respective domain. One model can be emulated in terms of another model—for example, graph data can be represented in a relational database—but the result is often awkward. That's why we have different systems for different purposes, not a single one-size-fits-all solution.

One thing that document and graph databases have in common is that they typically don't enforce a schema for the data they store, which can make it easier to adapt applications to changing requirements. However, your application most likely still assumes that data has a certain structure; it's just a question of whether the schema is explicit (enforced on write) or implicit (handled on read).

Each data model comes with its own query language or framework, and we discussed several examples: SQL, MapReduce, MongoDB's aggregation pipeline, Cypher, SPARQL, and Datalog. We also touched on CSS and XSL/XPath, which aren't database query languages but have interesting parallels.

Although we have covered a lot of ground, there are still many data models left unmentioned. To give just a few brief examples:

- Researchers working with genome data often need to perform *sequence-similarity searches*, which means taking one very long string (representing a

DNA molecule) and matching it against a large database of strings that are similar, but not identical. None of the databases described here can handle this kind of usage, which is why researchers have written specialized genome database software like GenBank [48].

- Particle physicists have been doing Big Data–style large-scale data analysis for decades, and projects like the Large Hadron Collider (LHC) now work with hundreds of petabytes! At such a scale custom solutions are required to stop the hardware cost from spiraling out of control [49].
- *Full-text search* is arguably a kind of data model that is frequently used alongside databases. Information retrieval is a large specialist subject that we won't cover in great detail in this book, but we'll touch on search indexes in Chapter 3 and Part III.

We have to leave it there for now. In the next chapter we will discuss some of the trade-offs that come into play when *implementing* the data models described in this chapter.

References

[1] Edgar F. Codd: "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, volume 13, number 6, pages 377–387, June 1970. doi: 10.1145/362384.362685

[2] Michael Stonebraker and Joseph M. Hellerstein: "What Goes Around Comes Around," in *Readings in Database Systems*, 4th edition, MIT Press, pages 2–41, 2005. ISBN: 978-0-262-69314-1

[3] Pramod J. Sadalage and Martin Fowler: *NoSQL Distilled*. Addison-Wesley, August 2012. ISBN: 978-0-321-82662-6

[4] Eric Evans: "NoSQL: What's in a Name?," *blog.sym-link.com*, October 30, 2009.

[5] James Phillips: "Surprises in Our NoSQL Adoption Survey," *blog.couchbase.com*, February 8, 2012.

[6] Michael Wagner: *SQL/XML:2006 – Evaluierung der Standardkonformität ausgewählter Datenbanksysteme*. Diplomica Verlag, Hamburg, 2010. ISBN: 978-3-836-64609-3

[7] "XML Data in SQL Server," SQL Server 2012 documentation, *technet.microsoft.com*, 2013.

[8] "PostgreSQL 9.3.1 Documentation," The PostgreSQL Global Development Group, 2013.

[9] "The MongoDB 2.4 Manual," MongoDB, Inc., 2013.

[10] "RethinkDB 1.11 Documentation," *rethinkdb.com*, 2013.

[11] "Apache CouchDB 1.6 Documentation," *docs.couchdb.org*, 2014.

[12] Lin Qiao, Kapil Surlaker, Shirshanka Das, et al.: "On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform," at *ACM International Conference on Management of Data* (SIGMOD), June 2013.

[13] Rick Long, Mark Harrington, Robert Hain, and Geoff Nicholls: *IMS Primer*. IBM Redbook SG24-5352-00, IBM International Technical Support Organization, January 2000.

[14] Stephen D. Bartlett: "IBM's IMS—Myths, Realities, and Opportunities," The Clipper Group Navigator, TCG2013015LI, July 2013.

[15] Sarah Mei: "Why You Should Never Use MongoDB," *sarahmei.com*, November 11, 2013.

[16] J. S. Knowles and D. M. R. Bell: "The CODASYL Model," in *Databases—Role and Structure: An Advanced Course*, edited by P. M. Stocker, P. M. D. Gray, and M. P. Atkinson, pages 19–56, Cambridge University Press, 1984. ISBN: 978-0-521-25430-4

[17] Charles W. Bachman: "The Programmer as Navigator," *Communications of the ACM*, volume 16, number 11, pages 653–658, November 1973. doi: 10.1145/355611.362534

[18] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton: "Architecture of a Database System," *Foundations and Trends in Databases*, volume 1, number 2, pages 141–259, November 2007. doi:10.1561/1900000002

[19] Sandeep Parikh and Kelly Stirman: "Schema Design for Time Series Data in MongoDB," *blog.mongodb.org*, October 30, 2013.

[20] Martin Fowler: "Schemaless Data Structures," *martinfowler.com*, January 7, 2013.

[21] Amr Awadallah: "Schema-on-Read vs. Schema-on-Write," at *Berkeley EECS RAD Lab Retreat*, Santa Cruz, CA, May 2009.

[22] Martin Odersky: "The Trouble with Types," at *Strange Loop*, September 2013.

[23] Conrad Irwin: "MongoDB—Confessions of a PostgreSQL Lover," at *HTML5DevConf*, October 2013.

[24] "Percona Toolkit Documentation: pt-online-schema-change," Percona Ireland Ltd., 2013.

[25] Rany Keddo, Tobias Bielohlawek, and Tobias Schmidt: "Large Hadron Migrator," SoundCloud, 2013.

[26] Shlomi Noach: "gh-ost: GitHub's Online Schema Migration Tool for MySQL," *githubengineering.com*, August 1, 2016.

[27] James C. Corbett, Jeffrey Dean, Michael Epstein, et al.: "Spanner: Google's Globally-Distributed Database," at *10th USENIX Symposium on Operating System Design and Implementation* (OSDI), October 2012.

[28] Donald K. Burleson: "Reduce I/O with Oracle Cluster Tables," *dba-oracle.com*.

[29] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al.: "Bigtable: A Distributed Storage System for Structured Data," at *7th USENIX Symposium on Operating System Design and Implementation* (OSDI), November 2006.

[30] Bobbie J. Cochrane and Kathy A. McKnight: "DB2 JSON Capabilities, Part 1: Introduction to DB2 JSON," IBM developerWorks, June 20, 2013.

[31] Herb Sutter: "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, volume 30, number 3, pages 202-210, March 2005.

[32] Joseph M. Hellerstein: "The Declarative Imperative: Experiences and Conjectures in Distributed Logic," Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech report UCB/EECS-2010-90, June 2010.

[33] Jeffrey Dean and Sanjay Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters," at *6th USENIX Symposium on Operating System Design and Implementation* (OSDI), December 2004.

[34] Craig Kerstiens: "JavaScript in Your Postgres," *blog.heroku.com*, June 5, 2013.

[35] Nathan Bronson, Zach Amsden, George Cabrera, et al.: "TAO: Facebook's Distributed Data Store for the Social Graph," at *USENIX Annual Technical Conference* (USENIX ATC), June 2013.

[36] "Apache TinkerPop3.2.3 Documentation," *tinkerpop.apache.org*, October 2016.

[37] "The Neo4j Manual v2.0.0," Neo Technology, 2013.

[38] Emil Eifrem: Twitter correspondence, January 3, 2014.

[39] David Beckett and Tim Berners-Lee: "Turtle – Terse RDF Triple Language," W3C Team Submission, March 28, 2011.

[40] "Datomic Development Resources," Metadata Partners, LLC, 2013.

[41] W3C RDF Working Group: "Resource Description Framework (RDF)," *w3.org*, 10 February 2004.

[42] "Apache Jena," Apache Software Foundation.

[43] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux: "SPARQL 1.1 Query Language," W3C Recommendation, March 2013.

[44] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou: "Datalog and Recursive Query Processing," *Foundations and Trends in Databases*, volume 5, number 2, pages 105–195, November 2013. doi:10.1561/1900000017

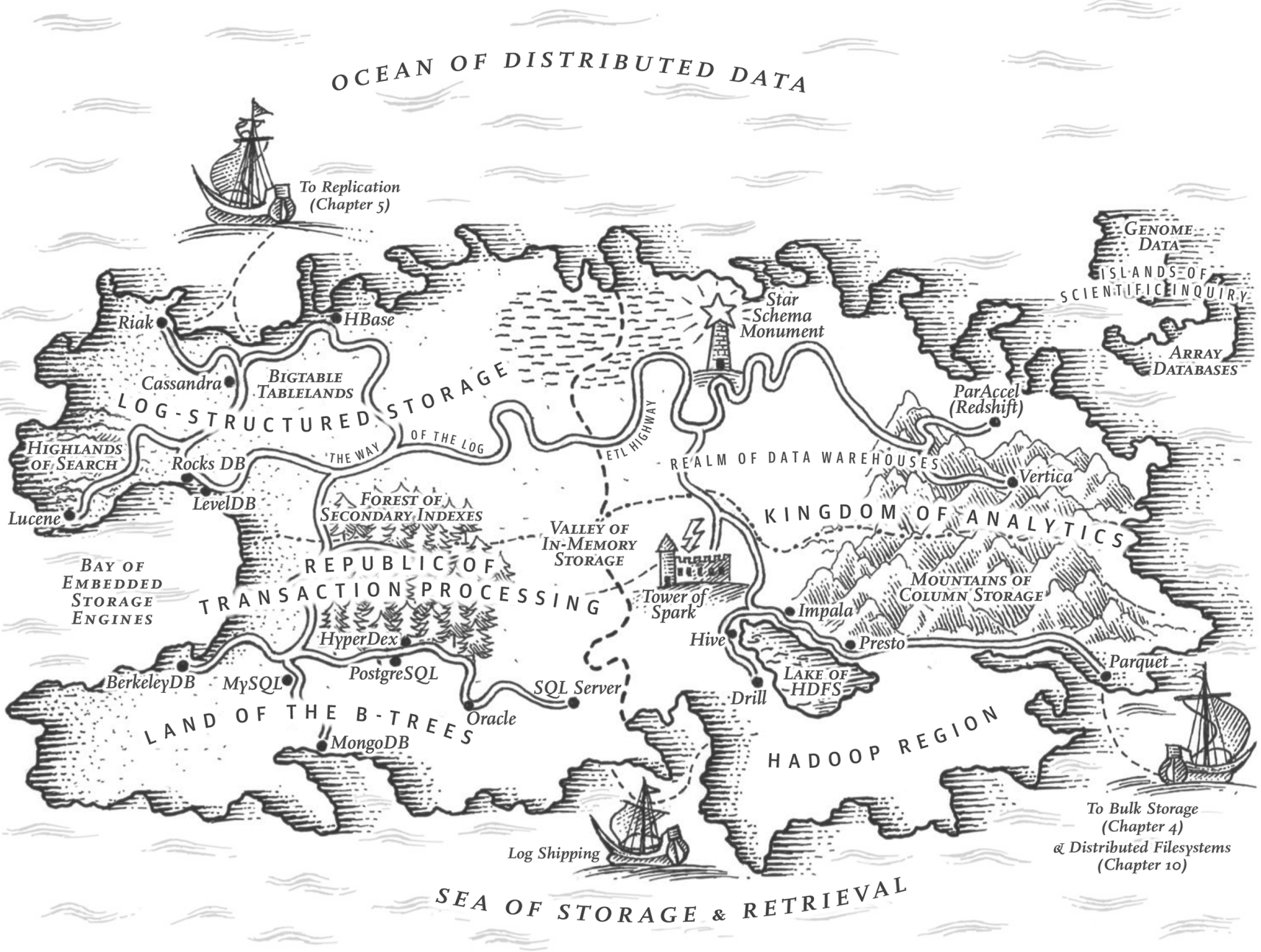
[45] Stefano Ceri, Georg Gottlob, and Letizia Tanca: "What You Always Wanted to Know About Datalog (And Never Dared to Ask)," *IEEE Transactions on Knowledge and Data Engineering*, volume 1, number 1, pages 146–166, March 1989. doi:10.1109/69.43410

[46] Serge Abiteboul, Richard Hull, and Victor Vianu: *Foundations of Databases*. Addison-Wesley, 1995. ISBN: 978-0-201-53771-0, available online at *webdam.inria.fr/Alice*

[47] Nathan Marz: "Cascalog," *cascalog.org*.

[48] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, et al.: "GenBank," *Nucleic Acids Research*, volume 36, Database issue, pages D25–D30, December 2007. doi:10.1093/nar/gkm929

[49] Fons Rademakers: "ROOT for Big Data Analysis," at *Workshop on the Future of Big Data Management*, London, UK, June 2013.


OCEAN OF DISTRIBUTED DATA
To Replication (Chapter 5)
Riak
HBase
Cassandra
Bigtable Tablelands
LOG-STRUCTURED STORAGE
Highlands of Search
Rocks DB
LevelDB
Lucene
THE WAY OF THE LOG
ETL HIGHWAY
Star Schema Monument
Genome Data
ISLANDS OF SCIENTIFIC INQUIRY
Array Databases
ParAccel (Redshift)
REALM OF DATA WAREHOUSES
Vertica
KINGDOM OF ANALYTICS
Forest of Secondary Indexes
Valley of In-Memory Storage
Bay of Embedded Storage Engines
REPUBLIC OF TRANSACTION PROCESSING
Tower of Spark
Mountains of Column Storage
Impala
Hive
Presto
Parquet
HyperDex
PostgreSQL
BerkeleyDB
MySQL
SQL Server
Oracle
Drill
Lake of HDFS
LAND OF THE B-TREES
MongoDB
HADOOP REGION
Log Shipping
To Bulk Storage (Chapter 4) & Distributed Filesystems (Chapter 10)
SEA OF STORAGE & RETRIEVAL

CHAPTER 3
Storage and Retrieval

Wer Ordnung hält, ist nur zu faul zum Suchen.
(If you keep things tidily ordered, you're just too lazy to go searching.)
—German proverb

On the most fundamental level, a database needs to do two things: when you give it some data, it should store the data, and when you ask it again later, it should give the data back to you.

In Chapter 2 we discussed data models and query languages—i.e., the format in which you (the application developer) give the database your data, and the mechanism by which you can ask for it again later. In this chapter we discuss the same from the database's point of view: how we can store the data that we're given, and how we can find it again when we're asked for it.

Why should you, as an application developer, care how the database handles storage and retrieval internally? You're probably not going to implement your own storage engine from scratch, but you *do* need to select a storage engine that is appropriate for your application, from the many that are available. In order to tune a storage engine to perform well on your kind of workload, you need to have a rough idea of what the storage engine is doing under the hood.

In particular, there is a big difference between storage engines that are optimized for transactional workloads and those that are optimized for analytics. We will explore that distinction later in "Transaction Processing or Analytics?" on page 90, and in "Column-Oriented Storage" on page 95 we'll discuss a family of storage engines that is optimized for analytics.

However, first we'll start this chapter by talking about storage engines that are used in the kinds of databases that you're probably familiar with: traditional relational databases, and also most so-called NoSQL databases. We will examine two families of

storage engines: *log-structured* storage engines, and *page-oriented* storage engines such as B-trees.

Data Structures That Power Your Database

Consider the world's simplest database, implemented as two Bash functions:

```
#!/bin/bash

db_set () {
    echo "$1,$2" >> database
}

db_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

These two functions implement a key-value store. You can call `db_set key value`, which will store `key` and `value` in the database. The key and value can be (almost) anything you like—for example, the value could be a JSON document. You can then call `db_get key`, which looks up the most recent value associated with that particular key and returns it.

And it works:

```
$ db_set 123456 '{"name":"London","attractions":["Big Ben","London Eye"]}'

$ db_set 42 '{"name":"San Francisco","attractions":["Golden Gate Bridge"]}'

$ db_get 42
{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
```

The underlying storage format is very simple: a text file where each line contains a key-value pair, separated by a comma (roughly like a CSV file, ignoring escaping issues). Every call to `db_set` appends to the end of the file, so if you update a key several times, the old versions of the value are not overwritten—you need to look at the last occurrence of a key in a file to find the latest value (hence the `tail -n 1` in `db_get`):

```
$ db_set 42 '{"name":"San Francisco","attractions":["Exploratorium"]}'

$ db_get 42
{"name":"San Francisco","attractions":["Exploratorium"]}

$ cat database
123456,{"name":"London","attractions":["Big Ben","London Eye"]}
42,{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
42,{"name":"San Francisco","attractions":["Exploratorium"]}
```

Our `db_set` function actually has pretty good performance for something that is so simple, because appending to a file is generally very efficient. Similarly to what `db_set` does, many databases internally use a *log*, which is an append-only data file. Real databases have more issues to deal with (such as concurrency control, reclaiming disk space so that the log doesn't grow forever, and handling errors and partially written records), but the basic principle is the same. Logs are incredibly useful, and we will encounter them several times in the rest of this book.

The word *log* is often used to refer to application logs, where an application outputs text that describes what's happening. In this book, *log* is used in the more general sense: an append-only sequence of records. It doesn't have to be human-readable; it might be binary and intended only for other programs to read.

On the other hand, our `db_get` function has terrible performance if you have a large number of records in your database. Every time you want to look up a key, `db_get` has to scan the entire database file from beginning to end, looking for occurrences of the key. In algorithmic terms, the cost of a lookup is $O(n)$: if you double the number of records n in your database, a lookup takes twice as long. That's not good.

In order to efficiently find the value for a particular key in the database, we need a different data structure: an *index*. In this chapter we will look at a range of indexing structures and see how they compare; the general idea behind them is to keep some additional metadata on the side, which acts as a signpost and helps you to locate the data you want. If you want to search the same data in several different ways, you may need several different indexes on different parts of the data.

An index is an *additional* structure that is derived from the primary data. Many databases allow you to add and remove indexes, and this doesn't affect the contents of the database; it only affects the performance of queries. Maintaining additional structures incurs overhead, especially on writes. For writes, it's hard to beat the performance of simply appending to a file, because that's the simplest possible write operation. Any kind of index usually slows down writes, because the index also needs to be updated every time data is written.

This is an important trade-off in storage systems: well-chosen indexes speed up read queries, but every index slows down writes. For this reason, databases don't usually index everything by default, but require you—the application developer or database administrator—to choose indexes manually, using your knowledge of the application's typical query patterns. You can then choose the indexes that give your application the greatest benefit, without introducing more overhead than necessary.

Hash Indexes

Let's start with indexes for key-value data. This is not the only kind of data you can index, but it's very common, and it's a useful building block for more complex indexes.

Key-value stores are quite similar to the *dictionary* type that you can find in most programming languages, and which is usually implemented as a hash map (hash table). Hash maps are described in many algorithms textbooks [1, 2], so we won't go into detail of how they work here. Since we already have hash maps for our in-memory data structures, why not use them to index our data on disk?

Let's say our data storage consists only of appending to a file, as in the preceding example. Then the simplest possible indexing strategy is this: keep an in-memory hash map where every key is mapped to a byte offset in the data file—the location at which the value can be found, as illustrated in Figure 3-1. Whenever you append a new key-value pair to the file, you also update the hash map to reflect the offset of the data you just wrote (this works both for inserting new keys and for updating existing keys). When you want to look up a value, use the hash map to find the offset in the data file, seek to that location, and read the value.

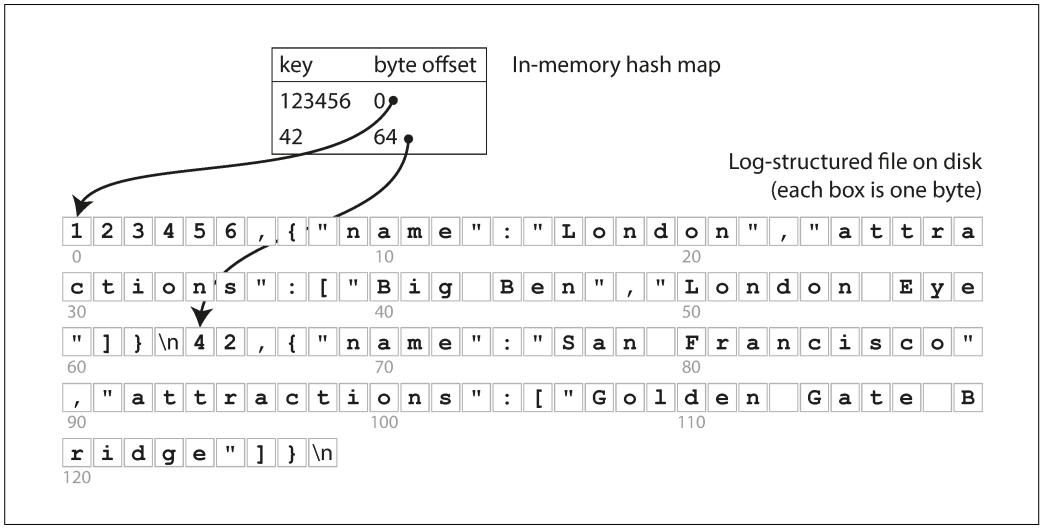


Figure 3-1. Storing a log of key-value pairs in a CSV-like format, indexed with an in-memory hash map.

This may sound simplistic, but it is a viable approach. In fact, this is essentially what Bitcask (the default storage engine in Riak) does [3]. Bitcask offers high-performance reads and writes, subject to the requirement that all the keys fit in the available RAM, since the hash map is kept completely in memory. The values can use more space than there is available memory, since they can be loaded from disk with just one disk

seek. If that part of the data file is already in the filesystem cache, a read doesn't require any disk I/O at all.

A storage engine like Bitcask is well suited to situations where the value for each key is updated frequently. For example, the key might be the URL of a cat video, and the value might be the number of times it has been played (incremented every time someone hits the play button). In this kind of workload, there are a lot of writes, but there are not too many distinct keys—you have a large number of writes per key, but it's feasible to keep all keys in memory.

As described so far, we only ever append to a file—so how do we avoid eventually running out of disk space? A good solution is to break the log into segments of a certain size by closing a segment file when it reaches a certain size, and making subsequent writes to a new segment file. We can then perform *compaction* on these segments, as illustrated in Figure 3-2. Compaction means throwing away duplicate keys in the log, and keeping only the most recent update for each key.

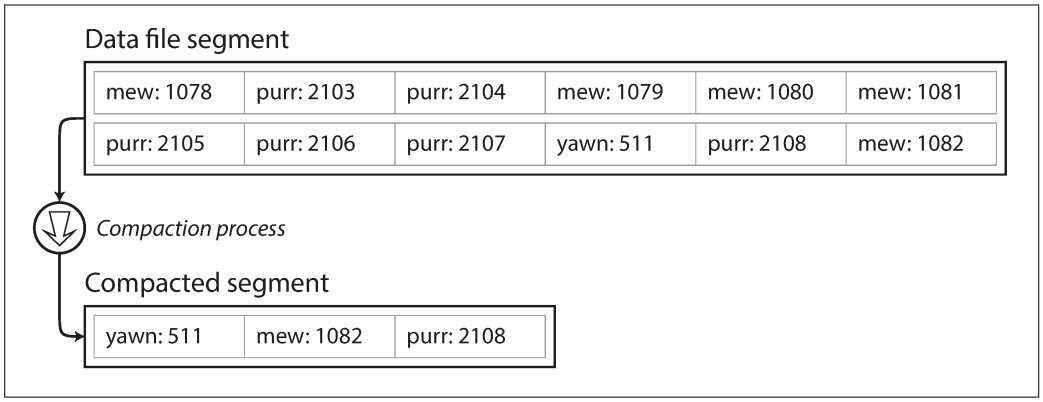


Figure 3-2. Compaction of a key-value update log (counting the number of times each cat video was played), retaining only the most recent value for each key.

Moreover, since compaction often makes segments much smaller (assuming that a key is overwritten several times on average within one segment), we can also merge several segments together at the same time as performing the compaction, as shown in Figure 3-3. Segments are never modified after they have been written, so the merged segment is written to a new file. The merging and compaction of frozen segments can be done in a background thread, and while it is going on, we can still continue to serve read and write requests as normal, using the old segment files. After the merging process is complete, we switch read requests to using the new merged segment instead of the old segments—and then the old segment files can simply be deleted.

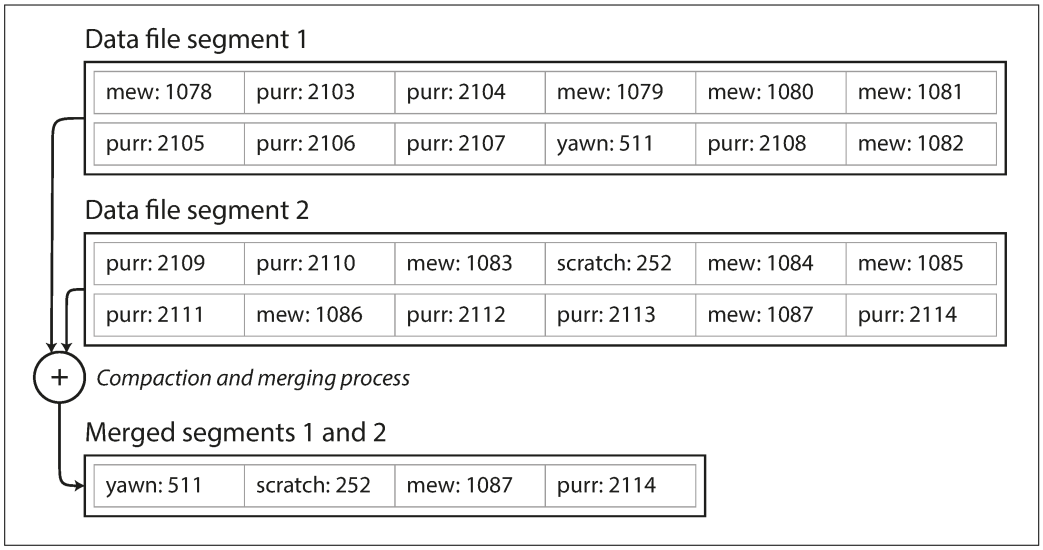


Figure 3-3. Performing compaction and segment merging simultaneously.

Each segment now has its own in-memory hash table, mapping keys to file offsets. In order to find the value for a key, we first check the most recent segment's hash map; if the key is not present we check the second-most-recent segment, and so on. The merging process keeps the number of segments small, so lookups don't need to check many hash maps.

Lots of detail goes into making this simple idea work in practice. Briefly, some of the issues that are important in a real implementation are:

File format
: CSV is not the best format for a log. It's faster and simpler to use a binary format that first encodes the length of a string in bytes, followed by the raw string (without need for escaping).

Deleting records
: If you want to delete a key and its associated value, you have to append a special deletion record to the data file (sometimes called a *tombstone*). When log segments are merged, the tombstone tells the merging process to discard any previous values for the deleted key.

Crash recovery
: If the database is restarted, the in-memory hash maps are lost. In principle, you can restore each segment's hash map by reading the entire segment file from beginning to end and noting the offset of the most recent value for every key as you go along. However, that might take a long time if the segment files are large, which would make server restarts painful. Bitcask speeds up recovery by storing

a snapshot of each segment's hash map on disk, which can be loaded into memory more quickly.

Partially written records
: The database may crash at any time, including halfway through appending a record to the log. Bitcask files include checksums, allowing such corrupted parts of the log to be detected and ignored.

Concurrency control
: As writes are appended to the log in a strictly sequential order, a common implementation choice is to have only one writer thread. Data file segments are append-only and otherwise immutable, so they can be read concurrently by multiple threads.

An append-only log seems wasteful at first glance: why don't you update the file in place, overwriting the old value with the new value? But an append-only design turns out to be good for several reasons:

- Appending and segment merging are sequential write operations, which are generally much faster than random writes, especially on magnetic spinning-disk hard drives. To some extent sequential writes are also preferable on flash-based *solid state drives* (SSDs) [4]. We will discuss this issue further in "Comparing B-Trees and LSM-Trees" on page 83.
- Concurrency and crash recovery are much simpler if segment files are append-only or immutable. For example, you don't have to worry about the case where a crash happened while a value was being overwritten, leaving you with a file containing part of the old and part of the new value spliced together.
- Merging old segments avoids the problem of data files getting fragmented over time.

However, the hash table index also has limitations:

- The hash table must fit in memory, so if you have a very large number of keys, you're out of luck. In principle, you could maintain a hash map on disk, but unfortunately it is difficult to make an on-disk hash map perform well. It requires a lot of random access I/O, it is expensive to grow when it becomes full, and hash collisions require fiddly logic [5].
- Range queries are not efficient. For example, you cannot easily scan over all keys between `kitty00000` and `kitty99999`—you'd have to look up each key individually in the hash maps.

In the next section we will look at an indexing structure that doesn't have those limitations.

SSTables and LSM-Trees

In Figure 3-3, each log-structured storage segment is a sequence of key-value pairs. These pairs appear in the order that they were written, and values later in the log take precedence over values for the same key earlier in the log. Apart from that, the order of key-value pairs in the file does not matter.

Now we can make a simple change to the format of our segment files: we require that the sequence of key-value pairs is *sorted by key*. At first glance, that requirement seems to break our ability to use sequential writes, but we'll get to that in a moment.

We call this format *Sorted String Table*, or *SSTable* for short. We also require that each key only appears once within each merged segment file (the compaction process already ensures that). SSTables have several big advantages over log segments with hash indexes:

1. Merging segments is simple and efficient, even if the files are bigger than the available memory. The approach is like the one used in the *mergesort* algorithm and is illustrated in Figure 3-4: you start reading the input files side by side, look at the first key in each file, copy the lowest key (according to the sort order) to the output file, and repeat. This produces a new merged segment file, also sorted by key.

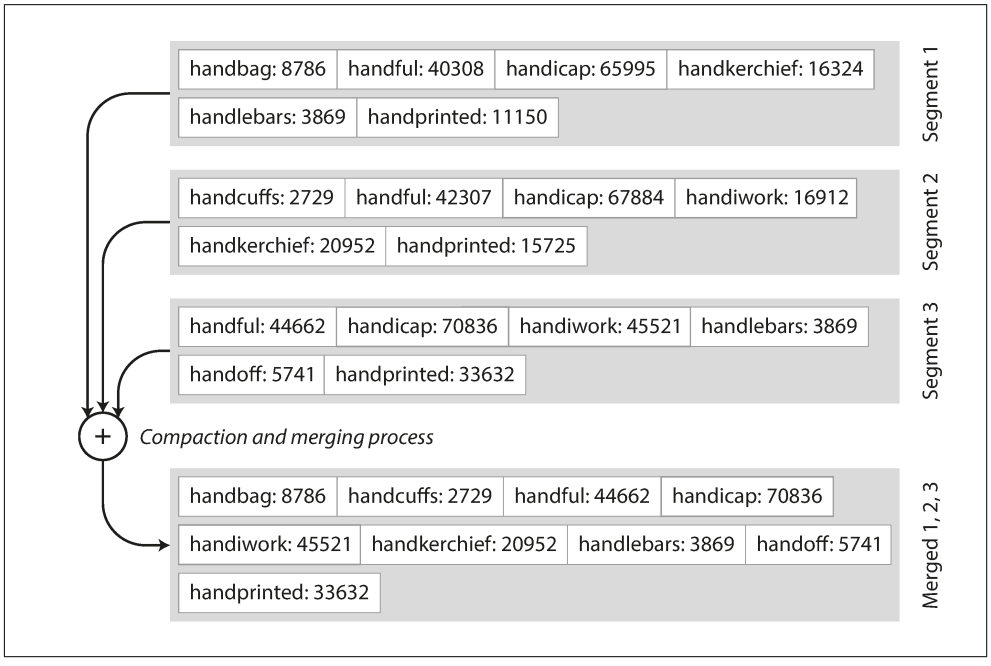


Figure 3-4. Merging several SSTable segments, retaining only the most recent value for each key.

What if the same key appears in several input segments? Remember that each segment contains all the values written to the database during some period of time. This means that all the values in one input segment must be more recent than all the values in the other segment (assuming that we always merge adjacent segments). When multiple segments contain the same key, we can keep the value from the most recent segment and discard the values in older segments.

2. In order to find a particular key in the file, you no longer need to keep an index of all the keys in memory. See Figure 3-5 for an example: say you're looking for the key `handiwork`, but you don't know the exact offset of that key in the segment file. However, you do know the offsets for the keys *handbag* and *handsome*, and because of the sorting you know that *handiwork* must appear between those two. This means you can jump to the offset for *handbag* and scan from there until you find *handiwork* (or not, if the key is not present in the file).

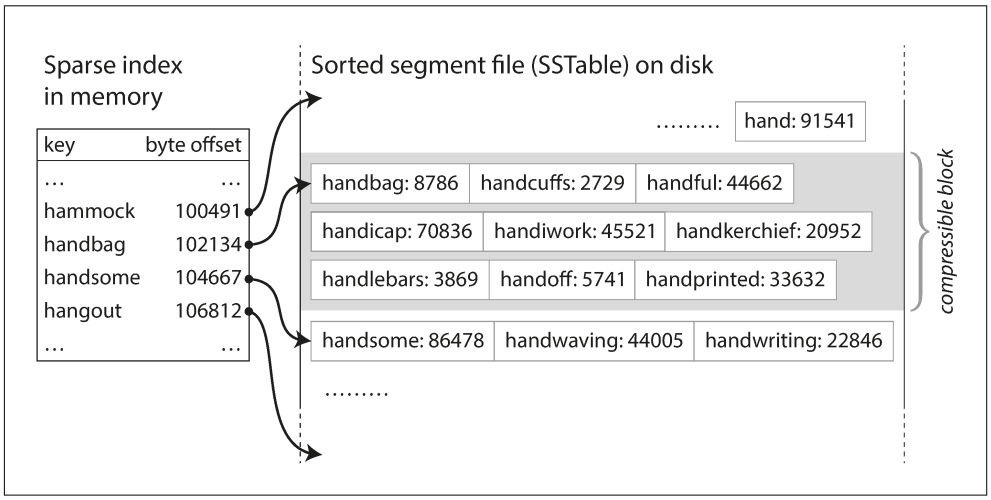


Figure 3-5. An SSTable with an in-memory index.

You still need an in-memory index to tell you the offsets for some of the keys, but it can be sparse: one key for every few kilobytes of segment file is sufficient, because a few kilobytes can be scanned very quickly.[i]

3. Since read requests need to scan over several key-value pairs in the requested range anyway, it is possible to group those records into a block and compress it before writing it to disk (indicated by the shaded area in Figure 3-5). Each entry of the sparse in-memory index then points at the start of a compressed block. Besides saving disk space, compression also reduces the I/O bandwidth use.

i. If all keys and values had a fixed size, you could use binary search on a segment file and avoid the in-memory index entirely. However, they are usually variable-length in practice, which makes it difficult to tell where one record ends and the next one starts if you don't have an index.

Constructing and maintaining SSTables

Fine so far—but how do you get your data to be sorted by key in the first place? Our incoming writes can occur in any order.

Maintaining a sorted structure on disk is possible (see "B-Trees" on page 79), but maintaining it in memory is much easier. There are plenty of well-known tree data structures that you can use, such as red-black trees or AVL trees [2]. With these data structures, you can insert keys in any order and read them back in sorted order.

We can now make our storage engine work as follows:

- When a write comes in, add it to an in-memory balanced tree data structure (for example, a red-black tree). This in-memory tree is sometimes called a *memtable*.
- When the memtable gets bigger than some threshold—typically a few megabytes—write it out to disk as an SSTable file. This can be done efficiently because the tree already maintains the key-value pairs sorted by key. The new SSTable file becomes the most recent segment of the database. While the SSTable is being written out to disk, writes can continue to a new memtable instance.
- In order to serve a read request, first try to find the key in the memtable, then in the most recent on-disk segment, then in the next-older segment, etc.
- From time to time, run a merging and compaction process in the background to combine segment files and to discard overwritten or deleted values.

This scheme works very well. It only suffers from one problem: if the database crashes, the most recent writes (which are in the memtable but not yet written out to disk) are lost. In order to avoid that problem, we can keep a separate log on disk to which every write is immediately appended, just like in the previous section. That log is not in sorted order, but that doesn't matter, because its only purpose is to restore the memtable after a crash. Every time the memtable is written out to an SSTable, the corresponding log can be discarded.

Making an LSM-tree out of SSTables

The algorithm described here is essentially what is used in LevelDB [6] and RocksDB [7], key-value storage engine libraries that are designed to be embedded into other applications. Among other things, LevelDB can be used in Riak as an alternative to Bitcask. Similar storage engines are used in Cassandra and HBase [8], both of which were inspired by Google's Bigtable paper [9] (which introduced the terms *SSTable* and *memtable*).

Originally this indexing structure was described by Patrick O'Neil et al. under the name *Log-Structured Merge-Tree* (or LSM-Tree) [10], building on earlier work on

log-structured filesystems [11]. Storage engines that are based on this principle of merging and compacting sorted files are often called LSM storage engines.

Lucene, an indexing engine for full-text search used by Elasticsearch and Solr, uses a similar method for storing its *term dictionary* [12, 13]. A full-text index is much more complex than a key-value index but is based on a similar idea: given a word in a search query, find all the documents (web pages, product descriptions, etc.) that mention the word. This is implemented with a key-value structure where the key is a word (a *term*) and the value is the list of IDs of all the documents that contain the word (the *postings list*). In Lucene, this mapping from term to postings list is kept in SSTable-like sorted files, which are merged in the background as needed [14].

Performance optimizations

As always, a lot of detail goes into making a storage engine perform well in practice. For example, the LSM-tree algorithm can be slow when looking up keys that do not exist in the database: you have to check the memtable, then the segments all the way back to the oldest (possibly having to read from disk for each one) before you can be sure that the key does not exist. In order to optimize this kind of access, storage engines often use additional *Bloom filters* [15]. (A Bloom filter is a memory-efficient data structure for approximating the contents of a set. It can tell you if a key does not appear in the database, and thus saves many unnecessary disk reads for nonexistent keys.)

There are also different strategies to determine the order and timing of how SSTables are compacted and merged. The most common options are *size-tiered* and *leveled* compaction. LevelDB and RocksDB use leveled compaction (hence the name of LevelDB), HBase uses size-tiered, and Cassandra supports both [16]. In size-tiered compaction, newer and smaller SSTables are successively merged into older and larger SSTables. In leveled compaction, the key range is split up into smaller SSTables and older data is moved into separate "levels," which allows the compaction to proceed more incrementally and use less disk space.

Even though there are many subtleties, the basic idea of LSM-trees—keeping a cascade of SSTables that are merged in the background—is simple and effective. Even when the dataset is much bigger than the available memory it continues to work well. Since data is stored in sorted order, you can efficiently perform range queries (scanning all keys above some minimum and up to some maximum), and because the disk writes are sequential the LSM-tree can support remarkably high write throughput.

B-Trees

The log-structured indexes we have discussed so far are gaining acceptance, but they are not the most common type of index. The most widely used indexing structure is quite different: the *B-tree*.

Introduced in 1970 [17] and called "ubiquitous" less than 10 years later [18], B-trees have stood the test of time very well. They remain the standard index implementation in almost all relational databases, and many nonrelational databases use them too.

Like SSTables, B-trees keep key-value pairs sorted by key, which allows efficient key-value lookups and range queries. But that's where the similarity ends: B-trees have a very different design philosophy.

The log-structured indexes we saw earlier break the database down into variable-size *segments*, typically several megabytes or more in size, and always write a segment sequentially. By contrast, B-trees break the database down into fixed-size *blocks* or *pages*, traditionally 4 KB in size (sometimes bigger), and read or write one page at a time. This design corresponds more closely to the underlying hardware, as disks are also arranged in fixed-size blocks.

Each page can be identified using an address or location, which allows one page to refer to another—similar to a pointer, but on disk instead of in memory. We can use these page references to construct a tree of pages, as illustrated in Figure 3-6.

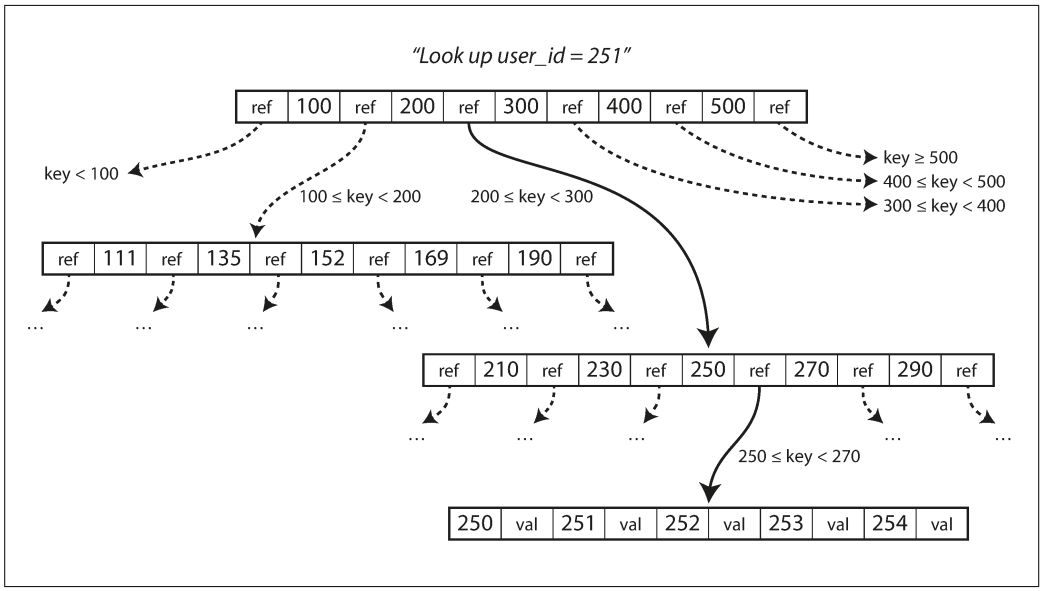


Figure 3-6. Looking up a key using a B-tree index.

One page is designated as the *root* of the B-tree; whenever you want to look up a key in the index, you start here. The page contains several keys and references to child pages. Each child is responsible for a continuous range of keys, and the keys between the references indicate where the boundaries between those ranges lie.

In the example in Figure 3-6, we are looking for the key 251, so we know that we need to follow the page reference between the boundaries 200 and 300. That takes us to a similar-looking page that further breaks down the 200–300 range into subranges.

Eventually we get down to a page containing individual keys (a *leaf page*), which either contains the value for each key inline or contains references to the pages where the values can be found.

The number of references to child pages in one page of the B-tree is called the *branching factor*. For example, in Figure 3-6 the branching factor is six. In practice, the branching factor depends on the amount of space required to store the page references and the range boundaries, but typically it is several hundred.

If you want to update the value for an existing key in a B-tree, you search for the leaf page containing that key, change the value in that page, and write the page back to disk (any references to that page remain valid). If you want to add a new key, you need to find the page whose range encompasses the new key and add it to that page. If there isn't enough free space in the page to accommodate the new key, it is split into two half-full pages, and the parent page is updated to account for the new subdivision of key ranges—see Figure 3-7.[ii]

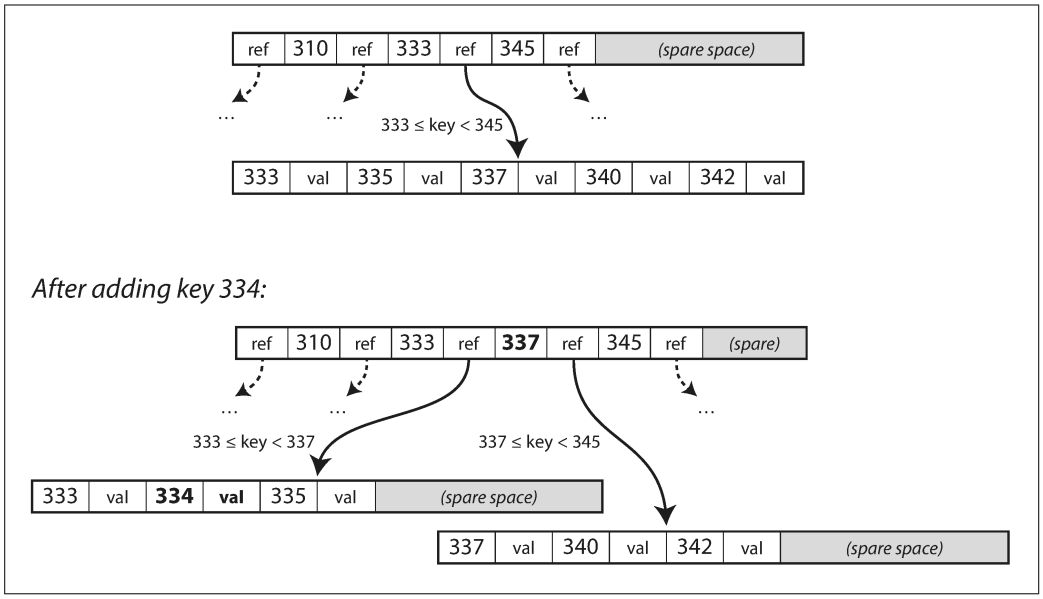


Figure 3-7. Growing a B-tree by splitting a page.

This algorithm ensures that the tree remains *balanced*: a B-tree with n keys always has a depth of $O(\log n)$. Most databases can fit into a B-tree that is three or four levels deep, so you don't need to follow many page references to find the page you are looking for. (A four-level tree of 4 KB pages with a branching factor of 500 can store up to 256 TB.)

ii. Inserting a new key into a B-tree is reasonably intuitive, but deleting one (while keeping the tree balanced) is somewhat more involved [2].

Making B-trees reliable

The basic underlying write operation of a B-tree is to overwrite a page on disk with new data. It is assumed that the overwrite does not change the location of the page; i.e., all references to that page remain intact when the page is overwritten. This is in stark contrast to log-structured indexes such as LSM-trees, which only append to files (and eventually delete obsolete files) but never modify files in place.

You can think of overwriting a page on disk as an actual hardware operation. On a magnetic hard drive, this means moving the disk head to the right place, waiting for the right position on the spinning platter to come around, and then overwriting the appropriate sector with new data. On SSDs, what happens is somewhat more complicated, due to the fact that an SSD must erase and rewrite fairly large blocks of a storage chip at a time [19].

Moreover, some operations require several different pages to be overwritten. For example, if you split a page because an insertion caused it to be overfull, you need to write the two pages that were split, and also overwrite their parent page to update the references to the two child pages. This is a dangerous operation, because if the database crashes after only some of the pages have been written, you end up with a corrupted index (e.g., there may be an *orphan* page that is not a child of any parent).

In order to make the database resilient to crashes, it is common for B-tree implementations to include an additional data structure on disk: a *write-ahead log* (WAL, also known as a *redo log*). This is an append-only file to which every B-tree modification must be written before it can be applied to the pages of the tree itself. When the database comes back up after a crash, this log is used to restore the B-tree back to a consistent state [5, 20].

An additional complication of updating pages in place is that careful concurrency control is required if multiple threads are going to access the B-tree at the same time —otherwise a thread may see the tree in an inconsistent state. This is typically done by protecting the tree's data structures with *latches* (lightweight locks). Log-structured approaches are simpler in this regard, because they do all the merging in the background without interfering with incoming queries and atomically swap old segments for new segments from time to time.

B-tree optimizations

As B-trees have been around for so long, it's not surprising that many optimizations have been developed over the years. To mention just a few:

- Instead of overwriting pages and maintaining a WAL for crash recovery, some databases (like LMDB) use a copy-on-write scheme [21]. A modified page is written to a different location, and a new version of the parent pages in the tree is created, pointing at the new location. This approach is also useful for concur-

rency control, as we shall see in "Snapshot Isolation and Repeatable Read" on page 237.

- We can save space in pages by not storing the entire key, but abbreviating it. Especially in pages on the interior of the tree, keys only need to provide enough information to act as boundaries between key ranges. Packing more keys into a page allows the tree to have a higher branching factor, and thus fewer levels.[iii]
- In general, pages can be positioned anywhere on disk; there is nothing requiring pages with nearby key ranges to be nearby on disk. If a query needs to scan over a large part of the key range in sorted order, that page-by-page layout can be inefficient, because a disk seek may be required for every page that is read. Many B-tree implementations therefore try to lay out the tree so that leaf pages appear in sequential order on disk. However, it's difficult to maintain that order as the tree grows. By contrast, since LSM-trees rewrite large segments of the storage in one go during merging, it's easier for them to keep sequential keys close to each other on disk.
- Additional pointers have been added to the tree. For example, each leaf page may have references to its sibling pages to the left and right, which allows scanning keys in order without jumping back to parent pages.
- B-tree variants such as *fractal trees* [22] borrow some log-structured ideas to reduce disk seeks (and they have nothing to do with fractals).

Comparing B-Trees and LSM-Trees

Even though B-tree implementations are generally more mature than LSM-tree implementations, LSM-trees are also interesting due to their performance characteristics. As a rule of thumb, LSM-trees are typically faster for writes, whereas B-trees are thought to be faster for reads [23]. Reads are typically slower on LSM-trees because they have to check several different data structures and SSTables at different stages of compaction.

However, benchmarks are often inconclusive and sensitive to details of the workload. You need to test systems with your particular workload in order to make a valid comparison. In this section we will briefly discuss a few things that are worth considering when measuring the performance of a storage engine.

iii. This variant is sometimes known as a B+ tree, although the optimization is so common that it often isn't distinguished from other B-tree variants.

Advantages of LSM-trees

A B-tree index must write every piece of data at least twice: once to the write-ahead log, and once to the tree page itself (and perhaps again as pages are split). There is also overhead from having to write an entire page at a time, even if only a few bytes in that page changed. Some storage engines even overwrite the same page twice in order to avoid ending up with a partially updated page in the event of a power failure [24, 25].

Log-structured indexes also rewrite data multiple times due to repeated compaction and merging of SSTables. This effect—one write to the database resulting in multiple writes to the disk over the course of the database's lifetime—is known as *write amplification*. It is of particular concern on SSDs, which can only overwrite blocks a limited number of times before wearing out.

In write-heavy applications, the performance bottleneck might be the rate at which the database can write to disk. In this case, write amplification has a direct performance cost: the more that a storage engine writes to disk, the fewer writes per second it can handle within the available disk bandwidth.

Moreover, LSM-trees are typically able to sustain higher write throughput than B-trees, partly because they sometimes have lower write amplification (although this depends on the storage engine configuration and workload), and partly because they sequentially write compact SSTable files rather than having to overwrite several pages in the tree [26]. This difference is particularly important on magnetic hard drives, where sequential writes are much faster than random writes.

LSM-trees can be compressed better, and thus often produce smaller files on disk than B-trees. B-tree storage engines leave some disk space unused due to fragmentation: when a page is split or when a row cannot fit into an existing page, some space in a page remains unused. Since LSM-trees are not page-oriented and periodically rewrite SSTables to remove fragmentation, they have lower storage overheads, especially when using leveled compaction [27].

On many SSDs, the firmware internally uses a log-structured algorithm to turn random writes into sequential writes on the underlying storage chips, so the impact of the storage engine's write pattern is less pronounced [19]. However, lower write amplification and reduced fragmentation are still advantageous on SSDs: representing data more compactly allows more read and write requests within the available I/O bandwidth.

Downsides of LSM-trees

A downside of log-structured storage is that the compaction process can sometimes interfere with the performance of ongoing reads and writes. Even though storage engines try to perform compaction incrementally and without affecting concurrent

access, disks have limited resources, so it can easily happen that a request needs to wait while the disk finishes an expensive compaction operation. The impact on throughput and average response time is usually small, but at higher percentiles (see "Describing Performance" on page 13) the response time of queries to log-structured storage engines can sometimes be quite high, and B-trees can be more predictable [28].

Another issue with compaction arises at high write throughput: the disk's finite write bandwidth needs to be shared between the initial write (logging and flushing a memtable to disk) and the compaction threads running in the background. When writing to an empty database, the full disk bandwidth can be used for the initial write, but the bigger the database gets, the more disk bandwidth is required for compaction.

If write throughput is high and compaction is not configured carefully, it can happen that compaction cannot keep up with the rate of incoming writes. In this case, the number of unmerged segments on disk keeps growing until you run out of disk space, and reads also slow down because they need to check more segment files. Typically, SSTable-based storage engines do not throttle the rate of incoming writes, even if compaction cannot keep up, so you need explicit monitoring to detect this situation [29, 30].

An advantage of B-trees is that each key exists in exactly one place in the index, whereas a log-structured storage engine may have multiple copies of the same key in different segments. This aspect makes B-trees attractive in databases that want to offer strong transactional semantics: in many relational databases, transaction isolation is implemented using locks on ranges of keys, and in a B-tree index, those locks can be directly attached to the tree [5]. In Chapter 7 we will discuss this point in more detail.

B-trees are very ingrained in the architecture of databases and provide consistently good performance for many workloads, so it's unlikely that they will go away anytime soon. In new datastores, log-structured indexes are becoming increasingly popular. There is no quick and easy rule for determining which type of storage engine is better for your use case, so it is worth testing empirically.

Other Indexing Structures

So far we have only discussed key-value indexes, which are like a *primary key* index in the relational model. A primary key uniquely identifies one row in a relational table, or one document in a document database, or one vertex in a graph database. Other records in the database can refer to that row/document/vertex by its primary key (or ID), and the index is used to resolve such references.

It is also very common to have *secondary indexes*. In relational databases, you can create several secondary indexes on the same table using the `CREATE INDEX` com-

mand, and they are often crucial for performing joins efficiently. For example, in Figure 2-1 in Chapter 2 you would most likely have a secondary index on the `user_id` columns so that you can find all the rows belonging to the same user in each of the tables.

A secondary index can easily be constructed from a key-value index. The main difference is that keys are not unique; i.e., there might be many rows (documents, vertices) with the same key. This can be solved in two ways: either by making each value in the index a list of matching row identifiers (like a postings list in a full-text index) or by making each key unique by appending a row identifier to it. Either way, both B-trees and log-structured indexes can be used as secondary indexes.

Storing values within the index

The key in an index is the thing that queries search for, but the value can be one of two things: it could be the actual row (document, vertex) in question, or it could be a reference to the row stored elsewhere. In the latter case, the place where rows are stored is known as a *heap file*, and it stores data in no particular order (it may be append-only, or it may keep track of deleted rows in order to overwrite them with new data later). The heap file approach is common because it avoids duplicating data when multiple secondary indexes are present: each index just references a location in the heap file, and the actual data is kept in one place.

When updating a value without changing the key, the heap file approach can be quite efficient: the record can be overwritten in place, provided that the new value is not larger than the old value. The situation is more complicated if the new value is larger, as it probably needs to be moved to a new location in the heap where there is enough space. In that case, either all indexes need to be updated to point at the new heap location of the record, or a forwarding pointer is left behind in the old heap location [5].

In some situations, the extra hop from the index to the heap file is too much of a performance penalty for reads, so it can be desirable to store the indexed row directly within an index. This is known as a *clustered index*. For example, in MySQL's InnoDB storage engine, the primary key of a table is always a clustered index, and secondary indexes refer to the primary key (rather than a heap file location) [31]. In SQL Server, you can specify one clustered index per table [32].

A compromise between a clustered index (storing all row data within the index) and a nonclustered index (storing only references to the data within the index) is known as a *covering index* or *index with included columns*, which stores *some* of a table's columns within the index [33]. This allows some queries to be answered by using the index alone (in which case, the index is said to *cover* the query) [32].

As with any kind of duplication of data, clustered and covering indexes can speed up reads, but they require additional storage and can add overhead on writes. Databases also need to go to additional effort to enforce transactional guarantees, because applications should not see inconsistencies due to the duplication.

Multi-column indexes

The indexes discussed so far only map a single key to a value. That is not sufficient if we need to query multiple columns of a table (or multiple fields in a document) simultaneously.

The most common type of multi-column index is called a *concatenated index*, which simply combines several fields into one key by appending one column to another (the index definition specifies in which order the fields are concatenated). This is like an old-fashioned paper phone book, which provides an index from (*lastname*, *firstname*) to phone number. Due to the sort order, the index can be used to find all the people with a particular last name, or all the people with a particular *lastname-firstname* combination. However, the index is useless if you want to find all the people with a particular first name.

Multi-dimensional indexes are a more general way of querying several columns at once, which is particularly important for geospatial data. For example, a restaurant-search website may have a database containing the latitude and longitude of each restaurant. When a user is looking at the restaurants on a map, the website needs to search for all the restaurants within the rectangular map area that the user is currently viewing. This requires a two-dimensional range query like the following:

```
SELECT * FROM restaurants WHERE latitude  > 51.4946 AND latitude  < 51.5079
                            AND longitude > -0.1162 AND longitude < -0.1004;
```

A standard B-tree or LSM-tree index is not able to answer that kind of query efficiently: it can give you either all the restaurants in a range of latitudes (but at any longitude), or all the restaurants in a range of longitudes (but anywhere between the North and South poles), but not both simultaneously.

One option is to translate a two-dimensional location into a single number using a space-filling curve, and then to use a regular B-tree index [34]. More commonly, specialized spatial indexes such as R-trees are used. For example, PostGIS implements geospatial indexes as R-trees using PostgreSQL's Generalized Search Tree indexing facility [35]. We don't have space to describe R-trees in detail here, but there is plenty of literature on them.

An interesting idea is that multi-dimensional indexes are not just for geographic locations. For example, on an ecommerce website you could use a three-dimensional index on the dimensions (*red*, *green*, *blue*) to search for products in a certain range of colors, or in a database of weather observations you could have a two-dimensional

index on (*date*, *temperature*) in order to efficiently search for all the observations during the year 2013 where the temperature was between 25 and 30℃. With a one-dimensional index, you would have to either scan over all the records from 2013 (regardless of temperature) and then filter them by temperature, or vice versa. A 2D index could narrow down by timestamp and temperature simultaneously. This technique is used by HyperDex [36].

Full-text search and fuzzy indexes

All the indexes discussed so far assume that you have exact data and allow you to query for exact values of a key, or a range of values of a key with a sort order. What they don't allow you to do is search for *similar* keys, such as misspelled words. Such *fuzzy* querying requires different techniques.

For example, full-text search engines commonly allow a search for one word to be expanded to include synonyms of the word, to ignore grammatical variations of words, and to search for occurrences of words near each other in the same document, and support various other features that depend on linguistic analysis of the text. To cope with typos in documents or queries, Lucene is able to search text for words within a certain edit distance (an edit distance of 1 means that one letter has been added, removed, or replaced) [37].

As mentioned in "Making an LSM-tree out of SSTables" on page 78, Lucene uses a SSTable-like structure for its term dictionary. This structure requires a small in-memory index that tells queries at which offset in the sorted file they need to look for a key. In LevelDB, this in-memory index is a sparse collection of some of the keys, but in Lucene, the in-memory index is a finite state automaton over the characters in the keys, similar to a *trie* [38]. This automaton can be transformed into a *Levenshtein automaton*, which supports efficient search for words within a given edit distance [39].

Other fuzzy search techniques go in the direction of document classification and machine learning. See an information retrieval textbook for more detail [e.g., 40].

Keeping everything in memory

The data structures discussed so far in this chapter have all been answers to the limitations of disks. Compared to main memory, disks are awkward to deal with. With both magnetic disks and SSDs, data on disk needs to be laid out carefully if you want good performance on reads and writes. However, we tolerate this awkwardness because disks have two significant advantages: they are durable (their contents are not lost if the power is turned off), and they have a lower cost per gigabyte than RAM.

As RAM becomes cheaper, the cost-per-gigabyte argument is eroded. Many datasets are simply not that big, so it's quite feasible to keep them entirely in memory, poten-

tially distributed across several machines. This has led to the development of *in-memory databases*.

Some in-memory key-value stores, such as Memcached, are intended for caching use only, where it's acceptable for data to be lost if a machine is restarted. But other in-memory databases aim for durability, which can be achieved with special hardware (such as battery-powered RAM), by writing a log of changes to disk, by writing periodic snapshots to disk, or by replicating the in-memory state to other machines.

When an in-memory database is restarted, it needs to reload its state, either from disk or over the network from a replica (unless special hardware is used). Despite writing to disk, it's still an in-memory database, because the disk is merely used as an append-only log for durability, and reads are served entirely from memory. Writing to disk also has operational advantages: files on disk can easily be backed up, inspected, and analyzed by external utilities.

Products such as VoltDB, MemSQL, and Oracle TimesTen are in-memory databases with a relational model, and the vendors claim that they can offer big performance improvements by removing all the overheads associated with managing on-disk data structures [41, 42]. RAMCloud is an open source, in-memory key-value store with durability (using a log-structured approach for the data in memory as well as the data on disk) [43]. Redis and Couchbase provide weak durability by writing to disk asynchronously.

Counterintuitively, the performance advantage of in-memory databases is not due to the fact that they don't need to read from disk. Even a disk-based storage engine may never need to read from disk if you have enough memory, because the operating system caches recently used disk blocks in memory anyway. Rather, they can be faster because they can avoid the overheads of encoding in-memory data structures in a form that can be written to disk [44].

Besides performance, another interesting area for in-memory databases is providing data models that are difficult to implement with disk-based indexes. For example, Redis offers a database-like interface to various data structures such as priority queues and sets. Because it keeps all data in memory, its implementation is comparatively simple.

Recent research indicates that an in-memory database architecture could be extended to support datasets larger than the available memory, without bringing back the overheads of a disk-centric architecture [45]. The so-called *anti-caching* approach works by evicting the least recently used data from memory to disk when there is not enough memory, and loading it back into memory when it is accessed again in the future. This is similar to what operating systems do with virtual memory and swap files, but the database can manage memory more efficiently than the OS, as it can work at the granularity of individual records rather than entire memory pages. This

approach still requires indexes to fit entirely in memory, though (like the Bitcask example at the beginning of the chapter).

Further changes to storage engine design will probably be needed if *non-volatile memory* (NVM) technologies become more widely adopted [46]. At present, this is a new area of research, but it is worth keeping an eye on in the future.

Transaction Processing or Analytics?

In the early days of business data processing, a write to the database typically corresponded to a *commercial transaction* taking place: making a sale, placing an order with a supplier, paying an employee's salary, etc. As databases expanded into areas that didn't involve money changing hands, the term *transaction* nevertheless stuck, referring to a group of reads and writes that form a logical unit.

A transaction needn't necessarily have ACID (atomicity, consistency, isolation, and durability) properties. *Transaction processing* just means allowing clients to make low-latency reads and writes—as opposed to *batch processing* jobs, which only run periodically (for example, once per day). We discuss the ACID properties in Chapter 7 and batch processing in Chapter 10.

Even though databases started being used for many different kinds of data—comments on blog posts, actions in a game, contacts in an address book, etc.—the basic access pattern remained similar to processing business transactions. An application typically looks up a small number of records by some key, using an index. Records are inserted or updated based on the user's input. Because these applications are interactive, the access pattern became known as *online transaction processing* (OLTP).

However, databases also started being increasingly used for *data analytics*, which has very different access patterns. Usually an analytic query needs to scan over a huge number of records, only reading a few columns per record, and calculates aggregate statistics (such as count, sum, or average) rather than returning the raw data to the user. For example, if your data is a table of sales transactions, then analytic queries might be:

- What was the total revenue of each of our stores in January?
- How many more bananas than usual did we sell during our latest promotion?
- Which brand of baby food is most often purchased together with brand X diapers?

These queries are often written by business analysts, and feed into reports that help the management of a company make better decisions (*business intelligence*). In order to differentiate this pattern of using databases from transaction processing, it has been called *online analytic processing* (OLAP) [47].[iv] The difference between OLTP and OLAP is not always clear-cut, but some typical characteristics are listed in Table 3-1.

Table 3-1. Comparing characteristics of transaction processing versus analytic systems

Property	Transaction processing systems (OLTP)	Analytic systems (OLAP)
Main read pattern	Small number of records per query, fetched by key	Aggregate over large number of records
Main write pattern	Random-access, low-latency writes from user input	Bulk import (ETL) or event stream
Primarily used by	End user/customer, via web application	Internal analyst, for decision support
What data represents	Latest state of data (current point in time)	History of events that happened over time
Dataset size	Gigabytes to terabytes	Terabytes to petabytes

At first, the same databases were used for both transaction processing and analytic queries. SQL turned out to be quite flexible in this regard: it works well for OLTP-type queries as well as OLAP-type queries. Nevertheless, in the late 1980s and early 1990s, there was a trend for companies to stop using their OLTP systems for analytics purposes, and to run the analytics on a separate database instead. This separate database was called a *data warehouse*.

Data Warehousing

An enterprise may have dozens of different transaction processing systems: systems powering the customer-facing website, controlling point of sale (checkout) systems in physical stores, tracking inventory in warehouses, planning routes for vehicles, managing suppliers, administering employees, etc. Each of these systems is complex and needs a team of people to maintain it, so the systems end up operating mostly autonomously from each other.

These OLTP systems are usually expected to be highly available and to process transactions with low latency, since they are often critical to the operation of the business. Database administrators therefore closely guard their OLTP databases. They are usually reluctant to let business analysts run ad hoc analytic queries on an OLTP database, since those queries are often expensive, scanning large parts of the dataset, which can harm the performance of concurrently executing transactions.

iv. The meaning of *online* in OLAP is unclear; it probably refers to the fact that queries are not just for predefined reports, but that analysts use the OLAP system interactively for explorative queries.

A *data warehouse*, by contrast, is a separate database that analysts can query to their hearts' content, without affecting OLTP operations [48]. The data warehouse contains a read-only copy of the data in all the various OLTP systems in the company. Data is extracted from OLTP databases (using either a periodic data dump or a continuous stream of updates), transformed into an analysis-friendly schema, cleaned up, and then loaded into the data warehouse. This process of getting data into the warehouse is known as *Extract–Transform–Load* (ETL) and is illustrated in Figure 3-8.

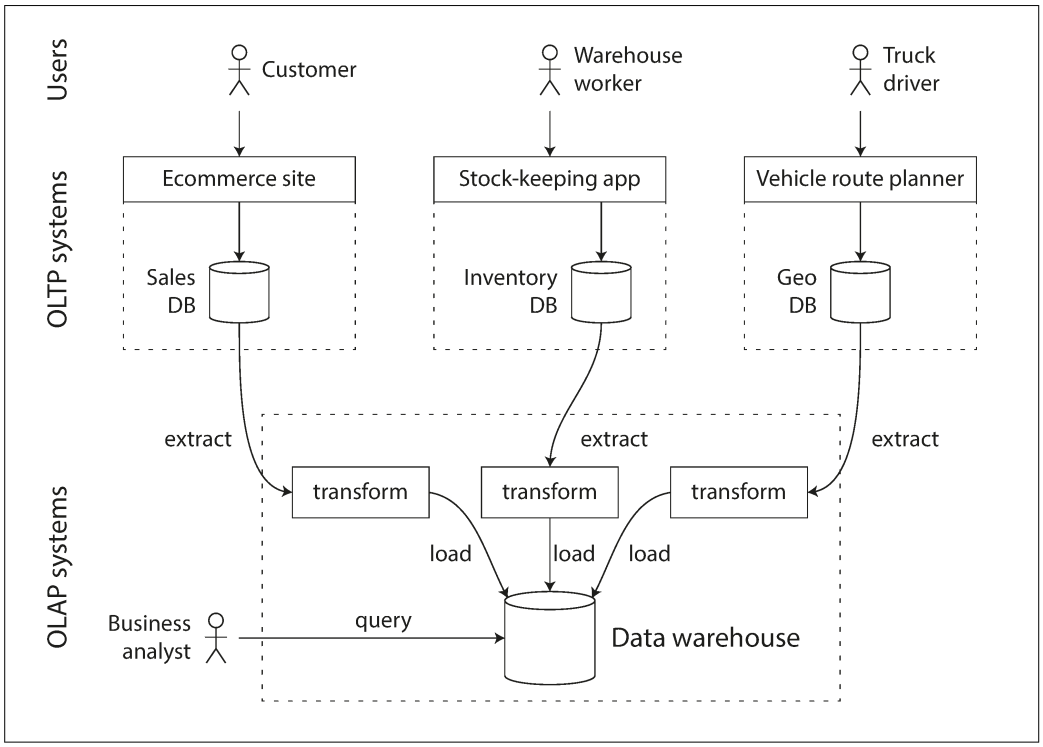


Figure 3-8. Simplified outline of ETL into a data warehouse.

Data warehouses now exist in almost all large enterprises, but in small companies they are almost unheard of. This is probably because most small companies don't have so many different OLTP systems, and most small companies have a small amount of data—small enough that it can be queried in a conventional SQL database, or even analyzed in a spreadsheet. In a large company, a lot of heavy lifting is required to do something that is simple in a small company.

A big advantage of using a separate data warehouse, rather than querying OLTP systems directly for analytics, is that the data warehouse can be optimized for analytic access patterns. It turns out that the indexing algorithms discussed in the first half of this chapter work well for OLTP, but are not very good at answering analytic queries.

In the rest of this chapter we will look at storage engines that are optimized for analytics instead.

The divergence between OLTP databases and data warehouses

The data model of a data warehouse is most commonly relational, because SQL is generally a good fit for analytic queries. There are many graphical data analysis tools that generate SQL queries, visualize the results, and allow analysts to explore the data (through operations such as *drill-down* and *slicing and dicing*).

On the surface, a data warehouse and a relational OLTP database look similar, because they both have a SQL query interface. However, the internals of the systems can look quite different, because they are optimized for very different query patterns. Many database vendors now focus on supporting either transaction processing or analytics workloads, but not both.

Some databases, such as Microsoft SQL Server and SAP HANA, have support for transaction processing and data warehousing in the same product. However, they are increasingly becoming two separate storage and query engines, which happen to be accessible through a common SQL interface [49, 50, 51].

Data warehouse vendors such as Teradata, Vertica, SAP HANA, and ParAccel typically sell their systems under expensive commercial licenses. Amazon RedShift is a hosted version of ParAccel. More recently, a plethora of open source SQL-on-Hadoop projects have emerged; they are young but aiming to compete with commercial data warehouse systems. These include Apache Hive, Spark SQL, Cloudera Impala, Facebook Presto, Apache Tajo, and Apache Drill [52, 53]. Some of them are based on ideas from Google's Dremel [54].

Stars and Snowflakes: Schemas for Analytics

As explored in Chapter 2, a wide range of different data models are used in the realm of transaction processing, depending on the needs of the application. On the other hand, in analytics, there is much less diversity of data models. Many data warehouses are used in a fairly formulaic style, known as a *star schema* (also known as *dimensional modeling* [55]).

The example schema in Figure 3-9 shows a data warehouse that might be found at a grocery retailer. At the center of the schema is a so-called *fact table* (in this example, it is called `fact_sales`). Each row of the fact table represents an event that occurred at a particular time (here, each row represents a customer's purchase of a product). If we were analyzing website traffic rather than retail sales, each row might represent a page view or a click by a user.

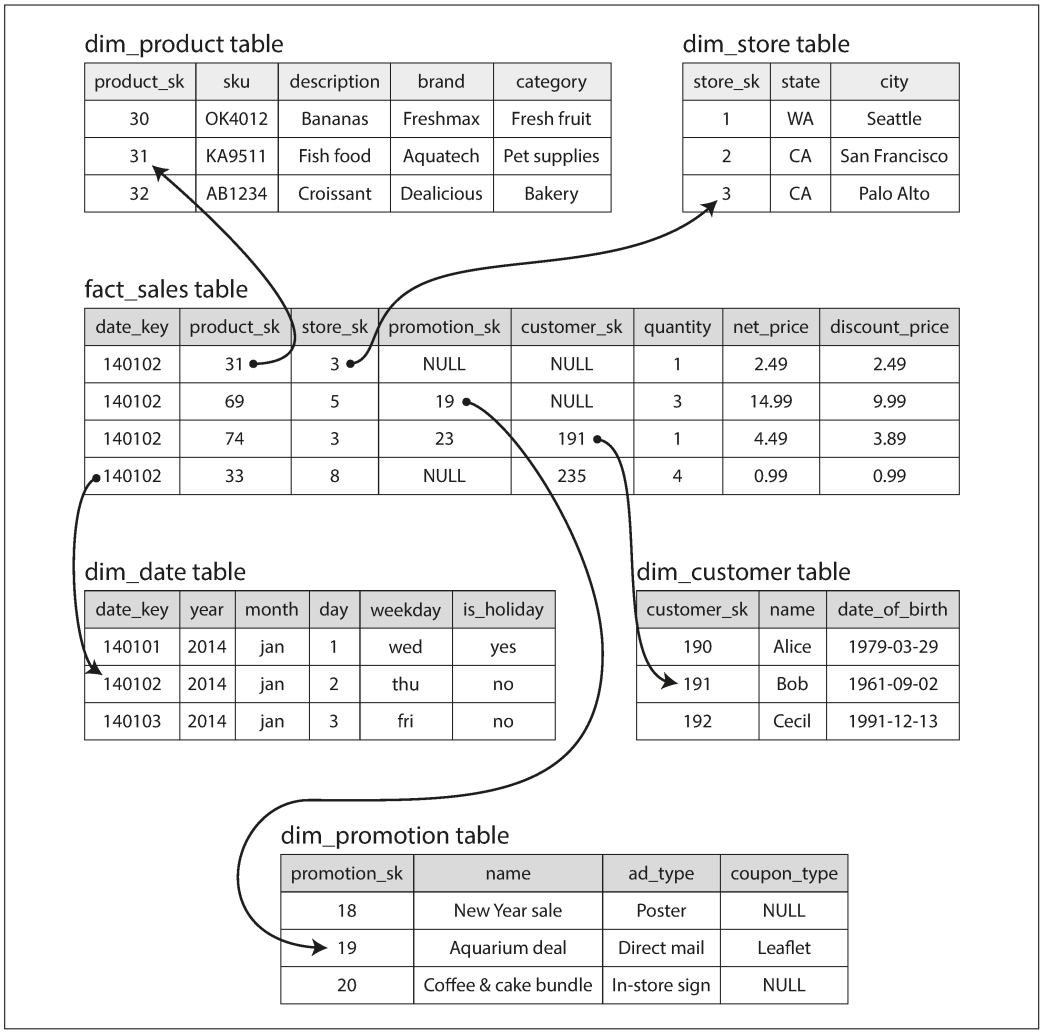


Figure 3-9. Example of a star schema for use in a data warehouse.

Usually, facts are captured as individual events, because this allows maximum flexibility of analysis later. However, this means that the fact table can become extremely large. A big enterprise like Apple, Walmart, or eBay may have tens of petabytes of transaction history in its data warehouse, most of which is in fact tables [56].

Some of the columns in the fact table are attributes, such as the price at which the product was sold and the cost of buying it from the supplier (allowing the profit margin to be calculated). Other columns in the fact table are foreign key references to other tables, called *dimension tables*. As each row in the fact table represents an event, the dimensions represent the *who*, *what*, *where*, *when*, *how*, and *why* of the event.

For example, in Figure 3-9, one of the dimensions is the product that was sold. Each row in the `dim_product` table represents one type of product that is for sale, including

its stock-keeping unit (SKU), description, brand name, category, fat content, package size, etc. Each row in the `fact_sales` table uses a foreign key to indicate which product was sold in that particular transaction. (For simplicity, if the customer buys several different products at once, they are represented as separate rows in the fact table.)

Even date and time are often represented using dimension tables, because this allows additional information about dates (such as public holidays) to be encoded, allowing queries to differentiate between sales on holidays and non-holidays.

The name "star schema" comes from the fact that when the table relationships are visualized, the fact table is in the middle, surrounded by its dimension tables; the connections to these tables are like the rays of a star.

A variation of this template is known as the *snowflake schema*, where dimensions are further broken down into subdimensions. For example, there could be separate tables for brands and product categories, and each row in the `dim_product` table could reference the brand and category as foreign keys, rather than storing them as strings in the `dim_product` table. Snowflake schemas are more normalized than star schemas, but star schemas are often preferred because they are simpler for analysts to work with [55].

In a typical data warehouse, tables are often very wide: fact tables often have over 100 columns, sometimes several hundred [51]. Dimension tables can also be very wide, as they include all the metadata that may be relevant for analysis—for example, the `dim_store` table may include details of which services are offered at each store, whether it has an in-store bakery, the square footage, the date when the store was first opened, when it was last remodeled, how far it is from the nearest highway, etc.

Column-Oriented Storage

If you have trillions of rows and petabytes of data in your fact tables, storing and querying them efficiently becomes a challenging problem. Dimension tables are usually much smaller (millions of rows), so in this section we will concentrate primarily on storage of facts.

Although fact tables are often over 100 columns wide, a typical data warehouse query only accesses 4 or 5 of them at one time (`"SELECT *"` queries are rarely needed for analytics) [51]. Take the query in Example 3-1: it accesses a large number of rows (every occurrence of someone buying fruit or candy during the 2013 calendar year), but it only needs to access three columns of the `fact_sales` table: `date_key`, `product_sk`, and `quantity`. The query ignores all other columns.

Example 3-1. Analyzing whether people are more inclined to buy fresh fruit or candy, depending on the day of the week

```
SELECT
  dim_date.weekday, dim_product.category,
  SUM(fact_sales.quantity) AS quantity_sold
FROM fact_sales
  JOIN dim_date    ON fact_sales.date_key   = dim_date.date_key
  JOIN dim_product ON fact_sales.product_sk = dim_product.product_sk
WHERE
  dim_date.year = 2013 AND
  dim_product.category IN ('Fresh fruit', 'Candy')
GROUP BY
  dim_date.weekday, dim_product.category;
```

How can we execute this query efficiently?

In most OLTP databases, storage is laid out in a *row-oriented* fashion: all the values from one row of a table are stored next to each other. Document databases are similar: an entire document is typically stored as one contiguous sequence of bytes. You can see this in the CSV example of Figure 3-1.

In order to process a query like Example 3-1, you may have indexes on `fact_sales.date_key` and/or `fact_sales.product_sk` that tell the storage engine where to find all the sales for a particular date or for a particular product. But then, a row-oriented storage engine still needs to load all of those rows (each consisting of over 100 attributes) from disk into memory, parse them, and filter out those that don't meet the required conditions. That can take a long time.

The idea behind *column-oriented storage* is simple: don't store all the values from one row together, but store all the values from each *column* together instead. If each column is stored in a separate file, a query only needs to read and parse those columns that are used in that query, which can save a lot of work. This principle is illustrated in Figure 3-10.

Column storage is easiest to understand in a relational data model, but it applies equally to nonrelational data. For example, Parquet [57] is a columnar storage format that supports a document data model, based on Google's Dremel [54].

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	69	4	NULL	NULL	1	13.99	13.99
140102	69	5	19	NULL	3	14.99	9.99
140102	69	5	NULL	191	1	14.99	14.99
140102	74	3	23	202	5	0.99	0.89
140103	31	2	NULL	NULL	1	2.49	2.49
140103	31	3	NULL	NULL	3	14.99	9.99
140103	31	3	21	123	1	49.99	39.99
140103	31	8	NULL	233	1	0.99	0.99

Columnar storage layout:

date_key file contents:	140102, 140102, 140102, 140102, 140103, 140103, 140103, 140103
product_sk file contents:	69, 69, 69, 74, 31, 31, 31, 31
store_sk file contents:	4, 5, 5, 3, 2, 3, 3, 8
promotion_sk file contents:	NULL, 19, NULL, 23, NULL, NULL, 21, NULL
customer_sk file contents:	NULL, NULL, 191, 202, NULL, NULL, 123, 233
quantity file contents:	1, 3, 1, 5, 1, 3, 1, 1
net_price file contents:	13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99
discount_price file contents:	13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99

Figure 3-10. Storing relational data by column, rather than by row.

The column-oriented storage layout relies on each column file containing the rows in the same order. Thus, if you need to reassemble an entire row, you can take the 23rd entry from each of the individual column files and put them together to form the 23rd row of the table.

Column Compression

Besides only loading those columns from disk that are required for a query, we can further reduce the demands on disk throughput by compressing data. Fortunately, column-oriented storage often lends itself very well to compression.

Take a look at the sequences of values for each column in Figure 3-10: they often look quite repetitive, which is a good sign for compression. Depending on the data in the column, different compression techniques can be used. One technique that is particularly effective in data warehouses is *bitmap encoding*, illustrated in Figure 3-11.

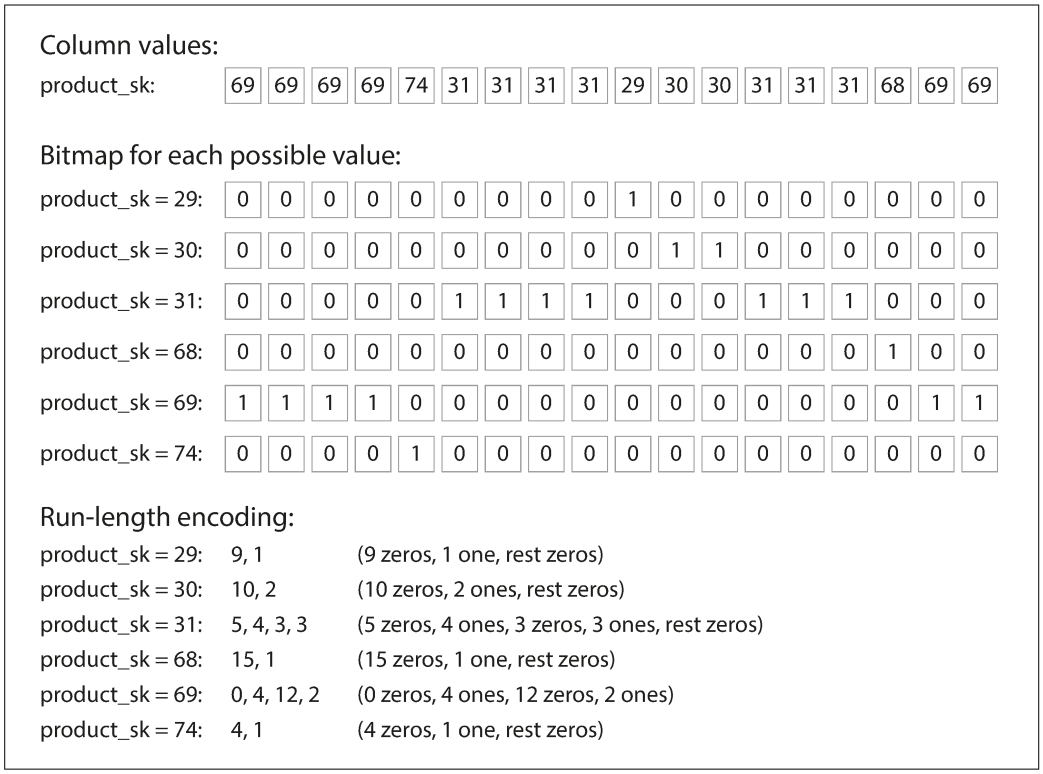


Figure 3-11. Compressed, bitmap-indexed storage of a single column.

Often, the number of distinct values in a column is small compared to the number of rows (for example, a retailer may have billions of sales transactions, but only 100,000 distinct products). We can now take a column with n distinct values and turn it into n separate bitmaps: one bitmap for each distinct value, with one bit for each row. The bit is 1 if the row has that value, and 0 if not.

If n is very small (for example, a *country* column may have approximately 200 distinct values), those bitmaps can be stored with one bit per row. But if n is bigger, there will be a lot of zeros in most of the bitmaps (we say that they are *sparse*). In that case, the bitmaps can additionally be run-length encoded, as shown at the bottom of Figure 3-11. This can make the encoding of a column remarkably compact.

Bitmap indexes such as these are very well suited for the kinds of queries that are common in a data warehouse. For example:

`WHERE product_sk IN (30, 68, 69)`:
: Load the three bitmaps for `product_sk = 30`, `product_sk = 68`, and `product_sk = 69`, and calculate the bitwise *OR* of the three bitmaps, which can be done very efficiently.

`WHERE product_sk = 31 AND store_sk = 3`:
: Load the bitmaps for `product_sk = 31` and `store_sk = 3`, and calculate the bitwise *AND*. This works because the columns contain the rows in the same order, so the *k*th bit in one column's bitmap corresponds to the same row as the *k*th bit in another column's bitmap.

There are also various other compression schemes for different kinds of data, but we won't go into them in detail—see [58] for an overview.

Column-oriented storage and column families

Cassandra and HBase have a concept of *column families*, which they inherited from Bigtable [9]. However, it is very misleading to call them column-oriented: within each column family, they store all columns from a row together, along with a row key, and they do not use column compression. Thus, the Bigtable model is still mostly row-oriented.

Memory bandwidth and vectorized processing

For data warehouse queries that need to scan over millions of rows, a big bottleneck is the bandwidth for getting data from disk into memory. However, that is not the only bottleneck. Developers of analytical databases also worry about efficiently using the bandwidth from main memory into the CPU cache, avoiding branch mispredictions and bubbles in the CPU instruction processing pipeline, and making use of single-instruction-multi-data (SIMD) instructions in modern CPUs [59, 60].

Besides reducing the volume of data that needs to be loaded from disk, column-oriented storage layouts are also good for making efficient use of CPU cycles. For example, the query engine can take a chunk of compressed column data that fits comfortably in the CPU's L1 cache and iterate through it in a tight loop (that is, with no function calls). A CPU can execute such a loop much faster than code that requires a lot of function calls and conditions for each record that is processed. Column compression allows more rows from a column to fit in the same amount of L1 cache. Operators, such as the bitwise *AND* and *OR* described previously, can be designed to operate on such chunks of compressed column data directly. This technique is known as *vectorized processing* [58, 49].

Sort Order in Column Storage

In a column store, it doesn't necessarily matter in which order the rows are stored. It's easiest to store them in the order in which they were inserted, since then inserting a new row just means appending to each of the column files. However, we can choose to impose an order, like we did with SSTables previously, and use that as an indexing mechanism.

Note that it wouldn't make sense to sort each column independently, because then we would no longer know which items in the columns belong to the same row. We can only reconstruct a row because we know that the *k*th item in one column belongs to the same row as the *k*th item in another column.

Rather, the data needs to be sorted an entire row at a time, even though it is stored by column. The administrator of the database can choose the columns by which the table should be sorted, using their knowledge of common queries. For example, if queries often target date ranges, such as the last month, it might make sense to make `date_key` the first sort key. Then the query optimizer can scan only the rows from the last month, which will be much faster than scanning all rows.

A second column can determine the sort order of any rows that have the same value in the first column. For example, if `date_key` is the first sort key in Figure 3-10, it might make sense for `product_sk` to be the second sort key so that all sales for the same product on the same day are grouped together in storage. That will help queries that need to group or filter sales by product within a certain date range.

Another advantage of sorted order is that it can help with compression of columns. If the primary sort column does not have many distinct values, then after sorting, it will have long sequences where the same value is repeated many times in a row. A simple run-length encoding, like we used for the bitmaps in Figure 3-11, could compress that column down to a few kilobytes—even if the table has billions of rows.

That compression effect is strongest on the first sort key. The second and third sort keys will be more jumbled up, and thus not have such long runs of repeated values. Columns further down the sorting priority appear in essentially random order, so they probably won't compress as well. But having the first few columns sorted is still a win overall.

Several different sort orders

A clever extension of this idea was introduced in C-Store and adopted in the commercial data warehouse Vertica [61, 62]. Different queries benefit from different sort orders, so why not store the same data sorted in *several different* ways? Data needs to be replicated to multiple machines anyway, so that you don't lose data if one machine fails. You might as well store that redundant data sorted in different ways so that when you're processing a query, you can use the version that best fits the query pattern.

Having multiple sort orders in a column-oriented store is a bit similar to having multiple secondary indexes in a row-oriented store. But the big difference is that the row-oriented store keeps every row in one place (in the heap file or a clustered index), and secondary indexes just contain pointers to the matching rows. In a column store, there normally aren't any pointers to data elsewhere, only columns containing values.

Writing to Column-Oriented Storage

These optimizations make sense in data warehouses, because most of the load consists of large read-only queries run by analysts. Column-oriented storage, compression, and sorting all help to make those read queries faster. However, they have the downside of making writes more difficult.

An update-in-place approach, like B-trees use, is not possible with compressed columns. If you wanted to insert a row in the middle of a sorted table, you would most likely have to rewrite all the column files. As rows are identified by their position within a column, the insertion has to update all columns consistently.

Fortunately, we have already seen a good solution earlier in this chapter: LSM-trees. All writes first go to an in-memory store, where they are added to a sorted structure and prepared for writing to disk. It doesn't matter whether the in-memory store is row-oriented or column-oriented. When enough writes have accumulated, they are merged with the column files on disk and written to new files in bulk. This is essentially what Vertica does [62].

Queries need to examine both the column data on disk and the recent writes in memory, and combine the two. However, the query optimizer hides this distinction from the user. From an analyst's point of view, data that has been modified with inserts, updates, or deletes is immediately reflected in subsequent queries.

Aggregation: Data Cubes and Materialized Views

Not every data warehouse is necessarily a column store: traditional row-oriented databases and a few other architectures are also used. However, columnar storage can be significantly faster for ad hoc analytical queries, so it is rapidly gaining popularity [51, 63].

Another aspect of data warehouses that is worth mentioning briefly is *materialized aggregates*. As discussed earlier, data warehouse queries often involve an aggregate function, such as `COUNT`, `SUM`, `AVG`, `MIN`, or `MAX` in SQL. If the same aggregates are used by many different queries, it can be wasteful to crunch through the raw data every time. Why not cache some of the counts or sums that queries use most often?

One way of creating such a cache is a *materialized view*. In a relational data model, it is often defined like a standard (virtual) view: a table-like object whose contents are the results of some query. The difference is that a materialized view is an actual copy of the query results, written to disk, whereas a virtual view is just a shortcut for writing queries. When you read from a virtual view, the SQL engine expands it into the view's underlying query on the fly and then processes the expanded query.

When the underlying data changes, a materialized view needs to be updated, because it is a denormalized copy of the data. The database can do that automatically, but

such updates make writes more expensive, which is why materialized views are not often used in OLTP databases. In read-heavy data warehouses they can make more sense (whether or not they actually improve read performance depends on the individual case).

A common special case of a materialized view is known as a *data cube* or *OLAP cube* [64]. It is a grid of aggregates grouped by different dimensions. Figure 3-12 shows an example.

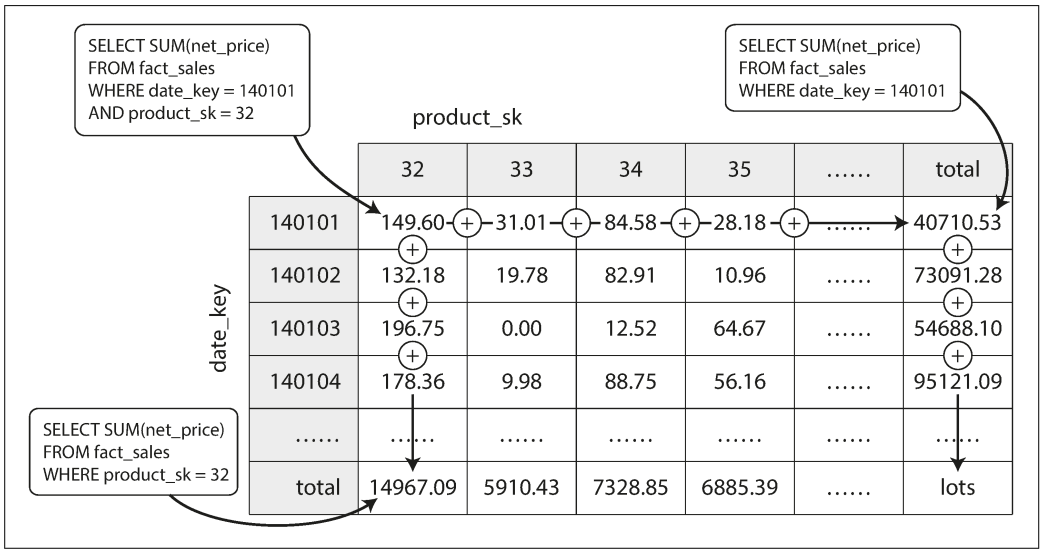


Figure 3-12. Two dimensions of a data cube, aggregating data by summing.

Imagine for now that each fact has foreign keys to only two dimension tables—in Figure 3-12, these are *date* and *product*. You can now draw a two-dimensional table, with dates along one axis and products along the other. Each cell contains the aggregate (e.g., `SUM`) of an attribute (e.g., `net_price`) of all facts with that date-product combination. Then you can apply the same aggregate along each row or column and get a summary that has been reduced by one dimension (the sales by product regardless of date, or the sales by date regardless of product).

In general, facts often have more than two dimensions. In Figure 3-9 there are five dimensions: date, product, store, promotion, and customer. It's a lot harder to imagine what a five-dimensional hypercube would look like, but the principle remains the same: each cell contains the sales for a particular date-product-store-promotion-customer combination. These values can then repeatedly be summarized along each of the dimensions.

The advantage of a materialized data cube is that certain queries become very fast because they have effectively been precomputed. For example, if you want to know

the total sales per store yesterday, you just need to look at the totals along the appropriate dimension—no need to scan millions of rows.

The disadvantage is that a data cube doesn't have the same flexibility as querying the raw data. For example, there is no way of calculating which proportion of sales comes from items that cost more than $100, because the price isn't one of the dimensions. Most data warehouses therefore try to keep as much raw data as possible, and use aggregates such as data cubes only as a performance boost for certain queries.

Summary

In this chapter we tried to get to the bottom of how databases handle storage and retrieval. What happens when you store data in a database, and what does the database do when you query for the data again later?

On a high level, we saw that storage engines fall into two broad categories: those optimized for transaction processing (OLTP), and those optimized for analytics (OLAP). There are big differences between the access patterns in those use cases:

- OLTP systems are typically user-facing, which means that they may see a huge volume of requests. In order to handle the load, applications usually only touch a small number of records in each query. The application requests records using some kind of key, and the storage engine uses an index to find the data for the requested key. Disk seek time is often the bottleneck here.
- Data warehouses and similar analytic systems are less well known, because they are primarily used by business analysts, not by end users. They handle a much lower volume of queries than OLTP systems, but each query is typically very demanding, requiring many millions of records to be scanned in a short time. Disk bandwidth (not seek time) is often the bottleneck here, and column-oriented storage is an increasingly popular solution for this kind of workload.

On the OLTP side, we saw storage engines from two main schools of thought:

- The log-structured school, which only permits appending to files and deleting obsolete files, but never updates a file that has been written. Bitcask, SSTables, LSM-trees, LevelDB, Cassandra, HBase, Lucene, and others belong to this group.
- The update-in-place school, which treats the disk as a set of fixed-size pages that can be overwritten. B-trees are the biggest example of this philosophy, being used in all major relational databases and also many nonrelational ones.

Log-structured storage engines are a comparatively recent development. Their key idea is that they systematically turn random-access writes into sequential writes on disk, which enables higher write throughput due to the performance characteristics of hard drives and SSDs.

Finishing off the OLTP side, we did a brief tour through some more complicated indexing structures, and databases that are optimized for keeping all data in memory.

We then took a detour from the internals of storage engines to look at the high-level architecture of a typical data warehouse. This background illustrated why analytic workloads are so different from OLTP: when your queries require sequentially scanning across a large number of rows, indexes are much less relevant. Instead it becomes important to encode data very compactly, to minimize the amount of data that the query needs to read from disk. We discussed how column-oriented storage helps achieve this goal.

As an application developer, if you're armed with this knowledge about the internals of storage engines, you are in a much better position to know which tool is best suited for your particular application. If you need to adjust a database's tuning parameters, this understanding allows you to imagine what effect a higher or a lower value may have.

Although this chapter couldn't make you an expert in tuning any one particular storage engine, it has hopefully equipped you with enough vocabulary and ideas that you can make sense of the documentation for the database of your choice.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman: *Data Structures and Algorithms*. Addison-Wesley, 1983. ISBN: 978-0-201-00023-8

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein: *Introduction to Algorithms*, 3rd edition. MIT Press, 2009. ISBN: 978-0-262-53305-8

[3] Justin Sheehy and David Smith: "Bitcask: A Log-Structured Hash Table for Fast Key/Value Data," Basho Technologies, April 2010.

[4] Yinan Li, Bingsheng He, Robin Jun Yang, et al.: "Tree Indexing on Solid State Drives," *Proceedings of the VLDB Endowment*, volume 3, number 1, pages 1195–1206, September 2010.

[5] Goetz Graefe: "Modern B-Tree Techniques," *Foundations and Trends in Databases*, volume 3, number 4, pages 203–402, August 2011. doi:10.1561/1900000028

[6] Jeffrey Dean and Sanjay Ghemawat: "LevelDB Implementation Notes," *leveldb.googlecode.com*.

[7] Dhruba Borthakur: "The History of RocksDB," *rocksdb.blogspot.com*, November 24, 2013.

[8] Matteo Bertozzi: "Apache HBase I/O – HFile," *blog.cloudera.com*, June, 29 2012.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al.: "Bigtable: A Distributed Storage System for Structured Data," at *7th USENIX Symposium on Operating System Design and Implementation* (OSDI), November 2006.

[10] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil: "The Log-Structured Merge-Tree (LSM-Tree)," *Acta Informatica*, volume 33, number 4, pages 351–385, June 1996. doi:10.1007/s002360050048

[11] Mendel Rosenblum and John K. Ousterhout: "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, volume 10, number 1, pages 26–52, February 1992. doi:10.1145/146941.146943

[12] Adrien Grand: "What Is in a Lucene Index?," at *Lucene/Solr Revolution*, November 14, 2013.

[13] Deepak Kandepet: "Hacking Lucene—The Index Format," *hackerlabs.org*, October 1, 2011.

[14] Michael McCandless: "Visualizing Lucene's Segment Merges," *blog.mikemccandless.com*, February 11, 2011.

[15] Burton H. Bloom: "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, volume 13, number 7, pages 422–426, July 1970. doi:10.1145/362686.362692

[16] "Operating Cassandra: Compaction," Apache Cassandra Documentation v4.0, 2016.

[17] Rudolf Bayer and Edward M. McCreight: "Organization and Maintenance of Large Ordered Indices," Boeing Scientific Research Laboratories, Mathematical and Information Sciences Laboratory, report no. 20, July 1970.

[18] Douglas Comer: "The Ubiquitous B-Tree," *ACM Computing Surveys*, volume 11, number 2, pages 121–137, June 1979. doi:10.1145/356770.356776

[19] Emmanuel Goossaert: "Coding for SSDs," *codecapsule.com*, February 12, 2014.

[20] C. Mohan and Frank Levine: "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," at *ACM International Conference on Management of Data* (SIGMOD), June 1992. doi:10.1145/130283.130338

[21] Howard Chu: "LDAP at Lightning Speed," at *Build Stuff '14*, November 2014.

[22] Bradley C. Kuszmaul: "A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees," *tokutek.com*, April 22, 2014.

[23] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, et al.: "Designing Access Methods: The RUM Conjecture," at *19th International Conference on Extending Database Technology* (EDBT), March 2016. doi:10.5441/002/edbt.2016.42

[24] Peter Zaitsev: "Innodb Double Write," *percona.com*, August 4, 2006.

[25] Tomas Vondra: "On the Impact of Full-Page Writes," *blog.2ndquadrant.com*, November 23, 2016.

[26] Mark Callaghan: "The Advantages of an LSM vs a B-Tree," *smalldatum.blogspot.co.uk*, January 19, 2016.

[27] Mark Callaghan: "Choosing Between Efficiency and Performance with RocksDB," at *Code Mesh*, November 4, 2016.

[28] Michi Mutsuzaki: "MySQL vs. LevelDB," *github.com*, August 2011.

[29] Benjamin Coverston, Jonathan Ellis, et al.: "CASSANDRA-1608: Redesigned Compaction, *issues.apache.org*, July 2011.

[30] Igor Canadi, Siying Dong, and Mark Callaghan: "RocksDB Tuning Guide," *github.com*, 2016.

[31] *MySQL 5.7 Reference Manual*. Oracle, 2014.

[32] *Books Online for SQL Server 2012*. Microsoft, 2012.

[33] Joe Webb: "Using Covering Indexes to Improve Query Performance," *simple-talk.com*, 29 September 2008.

[34] Frank Ramsak, Volker Markl, Robert Fenk, et al.: "Integrating the UB-Tree into a Database System Kernel," at *26th International Conference on Very Large Data Bases* (VLDB), September 2000.

[35] The PostGIS Development Group: "PostGIS 2.1.2dev Manual," *postgis.net*, 2014.

[36] Robert Escriva, Bernard Wong, and Emin Gün Sirer: "HyperDex: A Distributed, Searchable Key-Value Store," at *ACM SIGCOMM Conference*, August 2012. doi:10.1145/2377677.2377681

[37] Michael McCandless: "Lucene's FuzzyQuery Is 100 Times Faster in 4.0," *blog.mikemccandless.com*, March 24, 2011.

[38] Steffen Heinz, Justin Zobel, and Hugh E. Williams: "Burst Tries: A Fast, Efficient Data Structure for String Keys," *ACM Transactions on Information Systems*, volume 20, number 2, pages 192–223, April 2002. doi:10.1145/506309.506312

[39] Klaus U. Schulz and Stoyan Mihov: "Fast String Correction with Levenshtein Automata," *International Journal on Document Analysis and Recognition*, volume 5, number 1, pages 67–85, November 2002. doi:10.1007/s10032-002-0082-8

[40] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze: *Introduction to Information Retrieval*. Cambridge University Press, 2008. ISBN: 978-0-521-86571-5, available online at *nlp.stanford.edu/IR-book*

[41] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, et al.: "The End of an Architectural Era (It's Time for a Complete Rewrite)," at *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.

[42] "VoltDB Technical Overview White Paper," VoltDB, 2014.

[43] Stephen M. Rumble, Ankita Kejriwal, and John K. Ousterhout: "Log-Structured Memory for DRAM-Based Storage," at *12th USENIX Conference on File and Storage Technologies* (FAST), February 2014.

[44] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker: "OLTP Through the Looking Glass, and What We Found There," at *ACM International Conference on Management of Data* (SIGMOD), June 2008. doi:10.1145/1376616.1376713

[45] Justin DeBrabant, Andrew Pavlo, Stephen Tu, et al.: "Anti-Caching: A New Approach to Database Management System Architecture," *Proceedings of the VLDB Endowment*, volume 6, number 14, pages 1942–1953, September 2013.

[46] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor: "Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems," at *ACM International Conference on Management of Data* (SIGMOD), June 2015. doi:10.1145/2723372.2749441

[47] Edgar F. Codd, S. B. Codd, and C. T. Salley: "Providing OLAP to User-Analysts: An IT Mandate," E. F. Codd Associates, 1993.

[48] Surajit Chaudhuri and Umeshwar Dayal: "An Overview of Data Warehousing and OLAP Technology," *ACM SIGMOD Record*, volume 26, number 1, pages 65–74, March 1997. doi:10.1145/248603.248616

[49] Per-Åke Larson, Cipri Clinciu, Campbell Fraser, et al.: "Enhancements to SQL Server Column Stores," at *ACM International Conference on Management of Data* (SIGMOD), June 2013.

[50] Franz Färber, Norman May, Wolfgang Lehner, et al.: "The SAP HANA Database – An Architecture Overview," *IEEE Data Engineering Bulletin*, volume 35, number 1, pages 28–33, March 2012.

[51] Michael Stonebraker: "The Traditional RDBMS Wisdom Is (Almost Certainly) All Wrong," presentation at *EPFL*, May 2013.

[52] Daniel J. Abadi: "Classifying the SQL-on-Hadoop Solutions," *hadapt.com*, October 2, 2013.

[53] Marcel Kornacker, Alexander Behm, Victor Bittorf, et al.: "Impala: A Modern, Open-Source SQL Engine for Hadoop," at *7th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2015.

[54] Sergey Melnik, Andrey Gubarev, Jing Jing Long, et al.: "Dremel: Interactive Analysis of Web-Scale Datasets," at *36th International Conference on Very Large Data Bases* (VLDB), pages 330–339, September 2010.

[55] Ralph Kimball and Margy Ross: *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd edition. John Wiley & Sons, July 2013. ISBN: 978-1-118-53080-1

[56] Derrick Harris: "Why Apple, eBay, and Walmart Have Some of the Biggest Data Warehouses You've Ever Seen," *gigaom.com*, March 27, 2013.

[57] Julien Le Dem: "Dremel Made Simple with Parquet," *blog.twitter.com*, September 11, 2013.

[58] Daniel J. Abadi, Peter Boncz, Stavros Harizopoulos, et al.: "The Design and Implementation of Modern Column-Oriented Database Systems," *Foundations and Trends in Databases*, volume 5, number 3, pages 197–280, December 2013. doi:10.1561/1900000024

[59] Peter Boncz, Marcin Zukowski, and Niels Nes: "MonetDB/X100: Hyper-Pipelining Query Execution," at *2nd Biennial Conference on Innovative Data Systems Research* (CIDR), January 2005.

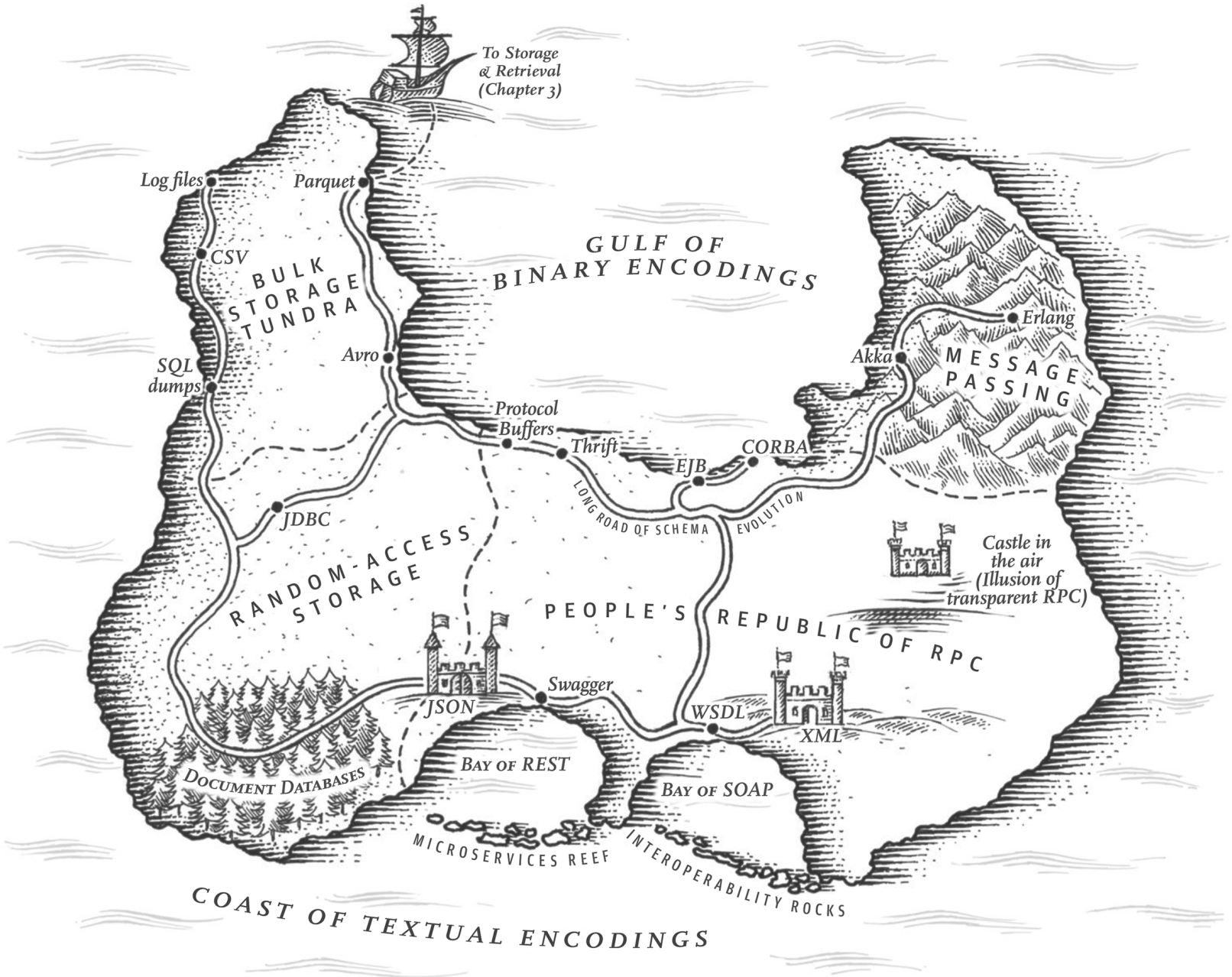
[60] Jingren Zhou and Kenneth A. Ross: "Implementing Database Operations Using SIMD Instructions," at *ACM International Conference on Management of Data* (SIGMOD), pages 145–156, June 2002. doi:10.1145/564691.564709

[61] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, et al.: "C-Store: A Column-oriented DBMS," at *31st International Conference on Very Large Data Bases* (VLDB), pages 553–564, September 2005.

[62] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, et al.: "The Vertica Analytic Database: C-Store 7 Years Later," *Proceedings of the VLDB Endowment*, volume 5, number 12, pages 1790–1801, August 2012.

[63] Julien Le Dem and Nong Li: "Efficient Data Storage for Analytics with Apache Parquet 2.0," at *Hadoop Summit*, San Jose, June 2014.

[64] Jim Gray, Surajit Chaudhuri, Adam Bosworth, et al.: "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals," *Data Mining and Knowledge Discovery*, volume 1, number 1, pages 29–53, March 2007. doi:10.1023/A:1009726021843


To Storage & Retrieval (Chapter 3)
Log files
Parquet
CSV
BULK STORAGE TUNDRA
GULF OF BINARY ENCODINGS
Erlang
Akka
MESSAGE PASSING
Avro
SQL dumps
Protocol Buffers
Thrift
EJB
CORBA
LONG ROAD OF SCHEMA EVOLUTION
JDBC
RANDOM-ACCESS STORAGE
Castle in the air (Illusion of transparent RPC)
PEOPLE'S REPUBLIC OF RPC
Swagger
JSON
WSDL
XML
DOCUMENT DATABASES
BAY OF REST
BAY OF SOAP
MICROSERVICES REEF
INTEROPERABILITY ROCKS
COAST OF TEXTUAL ENCODINGS

CHAPTER 4
Encoding and Evolution

> *Everything changes and nothing stands still.*
>
> —Heraclitus of Ephesus, as quoted by Plato in *Cratylus* (360 BCE)

Applications inevitably change over time. Features are added or modified as new products are launched, user requirements become better understood, or business circumstances change. In Chapter 1 we introduced the idea of *evolvability*: we should aim to build systems that make it easy to adapt to change (see "Evolvability: Making Change Easy" on page 21).

In most cases, a change to an application's features also requires a change to data that it stores: perhaps a new field or record type needs to be captured, or perhaps existing data needs to be presented in a new way.

The data models we discussed in Chapter 2 have different ways of coping with such change. Relational databases generally assume that all data in the database conforms to one schema: although that schema can be changed (through schema migrations; i.e., `ALTER` statements), there is exactly one schema in force at any one point in time. By contrast, schema-on-read ("schemaless") databases don't enforce a schema, so the database can contain a mixture of older and newer data formats written at different times (see "Schema flexibility in the document model" on page 39).

When a data format or schema changes, a corresponding change to application code often needs to happen (for example, you add a new field to a record, and the application code starts reading and writing that field). However, in a large application, code changes often cannot happen instantaneously:

- With server-side applications you may want to perform a *rolling upgrade* (also known as a *staged rollout*), deploying the new version to a few nodes at a time, checking whether the new version is running smoothly, and gradually working your way through all the nodes. This allows new versions to be deployed without service downtime, and thus encourages more frequent releases and better evolvability.
- With client-side applications you're at the mercy of the user, who may not install the update for some time.

This means that old and new versions of the code, and old and new data formats, may potentially all coexist in the system at the same time. In order for the system to continue running smoothly, we need to maintain compatibility in both directions:

Backward compatibility
: Newer code can read data that was written by older code.

Forward compatibility
: Older code can read data that was written by newer code.

Backward compatibility is normally not hard to achieve: as author of the newer code, you know the format of data written by older code, and so you can explicitly handle it (if necessary by simply keeping the old code to read the old data). Forward compatibility can be trickier, because it requires older code to ignore additions made by a newer version of the code.

In this chapter we will look at several formats for encoding data, including JSON, XML, Protocol Buffers, Thrift, and Avro. In particular, we will look at how they handle schema changes and how they support systems where old and new data and code need to coexist. We will then discuss how those formats are used for data storage and for communication: in web services, Representational State Transfer (REST), and remote procedure calls (RPC), as well as message-passing systems such as actors and message queues.

Formats for Encoding Data

Programs usually work with data in (at least) two different representations:

1. In memory, data is kept in objects, structs, lists, arrays, hash tables, trees, and so on. These data structures are optimized for efficient access and manipulation by the CPU (typically using pointers).
2. When you want to write data to a file or send it over the network, you have to encode it as some kind of self-contained sequence of bytes (for example, a JSON document). Since a pointer wouldn't make sense to any other process, this

sequence-of-bytes representation looks quite different from the data structures that are normally used in memory.[i]

Thus, we need some kind of translation between the two representations. The translation from the in-memory representation to a byte sequence is called *encoding* (also known as *serialization* or *marshalling*), and the reverse is called *decoding* (*parsing*, *deserialization*, *unmarshalling*).[ii]

Terminology clash

Serialization is unfortunately also used in the context of transactions (see Chapter 7), with a completely different meaning. To avoid overloading the word we'll stick with *encoding* in this book, even though *serialization* is perhaps a more common term.

As this is such a common problem, there are a myriad different libraries and encoding formats to choose from. Let's do a brief overview.

Language-Specific Formats

Many programming languages come with built-in support for encoding in-memory objects into byte sequences. For example, Java has `java.io.Serializable` [1], Ruby has `Marshal` [2], Python has `pickle` [3], and so on. Many third-party libraries also exist, such as Kryo for Java [4].

These encoding libraries are very convenient, because they allow in-memory objects to be saved and restored with minimal additional code. However, they also have a number of deep problems:

- The encoding is often tied to a particular programming language, and reading the data in another language is very difficult. If you store or transmit data in such an encoding, you are committing yourself to your current programming language for potentially a very long time, and precluding integrating your systems with those of other organizations (which may use different languages).
- In order to restore data in the same object types, the decoding process needs to be able to instantiate arbitrary classes. This is frequently a source of security problems [5]: if an attacker can get your application to decode an arbitrary byte sequence, they can instantiate arbitrary classes, which in turn often allows them to do terrible things such as remotely executing arbitrary code [6, 7].

---

i. With the exception of some special cases, such as certain memory-mapped files or when operating directly on compressed data (as described in "Column Compression" on page 97).

ii. Note that *encoding* has nothing to do with *encryption*. We don't discuss encryption in this book.

- Versioning data is often an afterthought in these libraries: as they are intended for quick and easy encoding of data, they often neglect the inconvenient problems of forward and backward compatibility.
- Efficiency (CPU time taken to encode or decode, and the size of the encoded structure) is also often an afterthought. For example, Java's built-in serialization is notorious for its bad performance and bloated encoding [8].

For these reasons it's generally a bad idea to use your language's built-in encoding for anything other than very transient purposes.

JSON, XML, and Binary Variants

Moving to standardized encodings that can be written and read by many programming languages, JSON and XML are the obvious contenders. They are widely known, widely supported, and almost as widely disliked. XML is often criticized for being too verbose and unnecessarily complicated [9]. JSON's popularity is mainly due to its built-in support in web browsers (by virtue of being a subset of JavaScript) and simplicity relative to XML. CSV is another popular language-independent format, albeit less powerful.

JSON, XML, and CSV are textual formats, and thus somewhat human-readable (although the syntax is a popular topic of debate). Besides the superficial syntactic issues, they also have some subtle problems:

- There is a lot of ambiguity around the encoding of numbers. In XML and CSV, you cannot distinguish between a number and a string that happens to consist of digits (except by referring to an external schema). JSON distinguishes strings and numbers, but it doesn't distinguish integers and floating-point numbers, and it doesn't specify a precision.

  This is a problem when dealing with large numbers; for example, integers greater than 2^{53} cannot be exactly represented in an IEEE 754 double-precision floating-point number, so such numbers become inaccurate when parsed in a language that uses floating-point numbers (such as JavaScript). An example of numbers larger than 2^{53} occurs on Twitter, which uses a 64-bit number to identify each tweet. The JSON returned by Twitter's API includes tweet IDs twice, once as a JSON number and once as a decimal string, to work around the fact that the numbers are not correctly parsed by JavaScript applications [10].
- JSON and XML have good support for Unicode character strings (i.e., human-readable text), but they don't support binary strings (sequences of bytes without a character encoding). Binary strings are a useful feature, so people get around this limitation by encoding the binary data as text using Base64. The schema is then used to indicate that the value should be interpreted as Base64-encoded. This works, but it's somewhat hacky and increases the data size by 33%.

- There is optional schema support for both XML [11] and JSON [12]. These schema languages are quite powerful, and thus quite complicated to learn and implement. Use of XML schemas is fairly widespread, but many JSON-based tools don't bother using schemas. Since the correct interpretation of data (such as numbers and binary strings) depends on information in the schema, applications that don't use XML/JSON schemas need to potentially hardcode the appropriate encoding/decoding logic instead.
- CSV does not have any schema, so it is up to the application to define the meaning of each row and column. If an application change adds a new row or column, you have to handle that change manually. CSV is also a quite vague format (what happens if a value contains a comma or a newline character?). Although its escaping rules have been formally specified [13], not all parsers implement them correctly.

Despite these flaws, JSON, XML, and CSV are good enough for many purposes. It's likely that they will remain popular, especially as data interchange formats (i.e., for sending data from one organization to another). In these situations, as long as people agree on what the format is, it often doesn't matter how pretty or efficient the format is. The difficulty of getting different organizations to agree on *anything* outweighs most other concerns.

Binary encoding

For data that is used only internally within your organization, there is less pressure to use a lowest-common-denominator encoding format. For example, you could choose a format that is more compact or faster to parse. For a small dataset, the gains are negligible, but once you get into the terabytes, the choice of data format can have a big impact.

JSON is less verbose than XML, but both still use a lot of space compared to binary formats. This observation led to the development of a profusion of binary encodings for JSON (MessagePack, BSON, BJSON, UBJSON, BISON, and Smile, to name a few) and for XML (WBXML and Fast Infoset, for example). These formats have been adopted in various niches, but none of them are as widely adopted as the textual versions of JSON and XML.

Some of these formats extend the set of datatypes (e.g., distinguishing integers and floating-point numbers, or adding support for binary strings), but otherwise they keep the JSON/XML data model unchanged. In particular, since they don't prescribe a schema, they need to include all the object field names within the encoded data. That is, in a binary encoding of the JSON document in Example 4-1, they will need to include the strings `userName`, `favoriteNumber`, and `interests` somewhere.

Example 4-1. Example record which we will encode in several binary formats in this chapter

```
{
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

Let's look at an example of MessagePack, a binary encoding for JSON. Figure 4-1 shows the byte sequence that you get if you encode the JSON document in Example 4-1 with MessagePack [14]. The first few bytes are as follows:

1. The first byte, `0x83`, indicates that what follows is an object (top four bits = `0x80`) with three fields (bottom four bits = `0x03`). (In case you're wondering what happens if an object has more than 15 fields, so that the number of fields doesn't fit in four bits, it then gets a different type indicator, and the number of fields is encoded in two or four bytes.)
2. The second byte, `0xa8`, indicates that what follows is a string (top four bits = `0xa0`) that is eight bytes long (bottom four bits = `0x08`).
3. The next eight bytes are the field name `userName` in ASCII. Since the length was indicated previously, there's no need for any marker to tell us where the string ends (or any escaping).
4. The next seven bytes encode the six-letter string value `Martin` with a prefix `0xa6`, and so on.

The binary encoding is 66 bytes long, which is only a little less than the 81 bytes taken by the textual JSON encoding (with whitespace removed). All the binary encodings of JSON are similar in this regard. It's not clear whether such a small space reduction (and perhaps a speedup in parsing) is worth the loss of human-readability.

In the following sections we will see how we can do much better, and encode the same record in just 32 bytes.

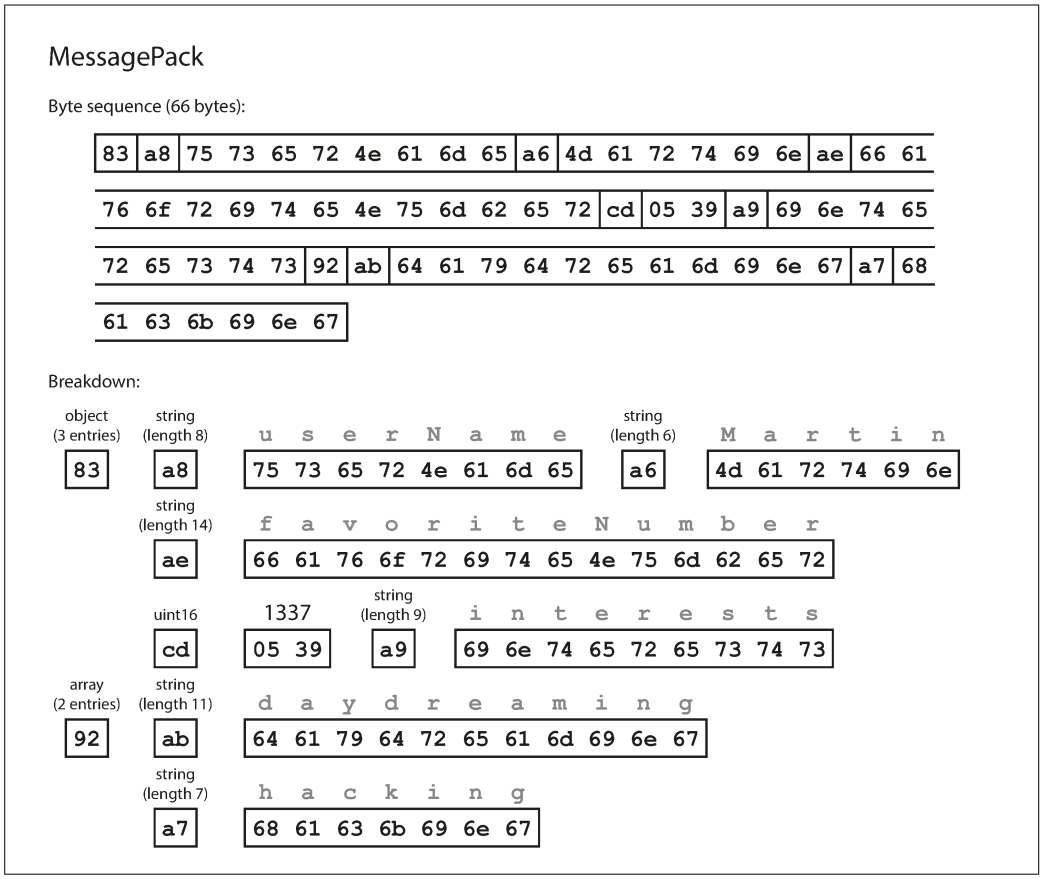


Figure 4-1. Example record (Example 4-1) encoded using MessagePack.

Thrift and Protocol Buffers

Apache Thrift [15] and Protocol Buffers (protobuf) [16] are binary encoding libraries that are based on the same principle. Protocol Buffers was originally developed at Google, Thrift was originally developed at Facebook, and both were made open source in 2007–08 [17].

Both Thrift and Protocol Buffers require a schema for any data that is encoded. To encode the data in Example 4-1 in Thrift, you would describe the schema in the Thrift interface definition language (IDL) like this:

```
struct Person {
  1: required string       userName,
  2: optional i64          favoriteNumber,
  3: optional list<string> interests
}
```

The equivalent schema definition for Protocol Buffers looks very similar:

```
message Person {
    required string user_name       = 1;
    optional int64  favorite_number = 2;
    repeated string interests       = 3;
}
```

Thrift and Protocol Buffers each come with a code generation tool that takes a schema definition like the ones shown here, and produces classes that implement the schema in various programming languages [18]. Your application code can call this generated code to encode or decode records of the schema.

What does data encoded with this schema look like? Confusingly, Thrift has two different binary encoding formats,[iii] called *BinaryProtocol* and *CompactProtocol*, respectively. Let's look at BinaryProtocol first. Encoding Example 4-1 in that format takes 59 bytes, as shown in Figure 4-2 [19].

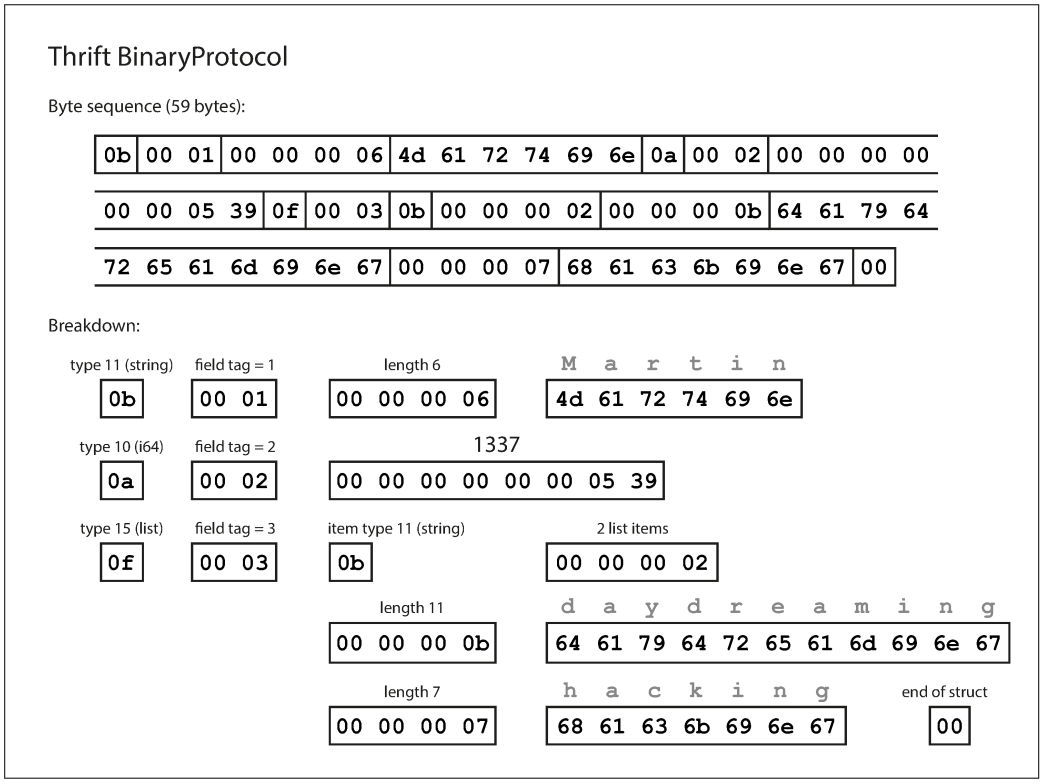


Figure 4-2. Example record encoded using Thrift's BinaryProtocol.

iii. Actually, it has three—BinaryProtocol, CompactProtocol, and DenseProtocol—although DenseProtocol is only supported by the C++ implementation, so it doesn't count as cross-language [18]. Besides those, it also has two different JSON-based encoding formats [19]. What fun!

Similarly to Figure 4-1, each field has a type annotation (to indicate whether it is a string, integer, list, etc.) and, where required, a length indication (length of a string, number of items in a list). The strings that appear in the data ("Martin", "daydreaming", "hacking") are also encoded as ASCII (or rather, UTF-8), similar to before.

The big difference compared to Figure 4-1 is that there are no field names (`userName`, `favoriteNumber`, `interests`). Instead, the encoded data contains *field tags*, which are numbers (`1`, `2`, and `3`). Those are the numbers that appear in the schema definition. Field tags are like aliases for fields—they are a compact way of saying what field we're talking about, without having to spell out the field name.

The Thrift CompactProtocol encoding is semantically equivalent to BinaryProtocol, but as you can see in Figure 4-3, it packs the same information into only 34 bytes. It does this by packing the field type and tag number into a single byte, and by using variable-length integers. Rather than using a full eight bytes for the number 1337, it is encoded in two bytes, with the top bit of each byte used to indicate whether there are still more bytes to come. This means numbers between –64 and 63 are encoded in one byte, numbers between –8192 and 8191 are encoded in two bytes, etc. Bigger numbers use more bytes.

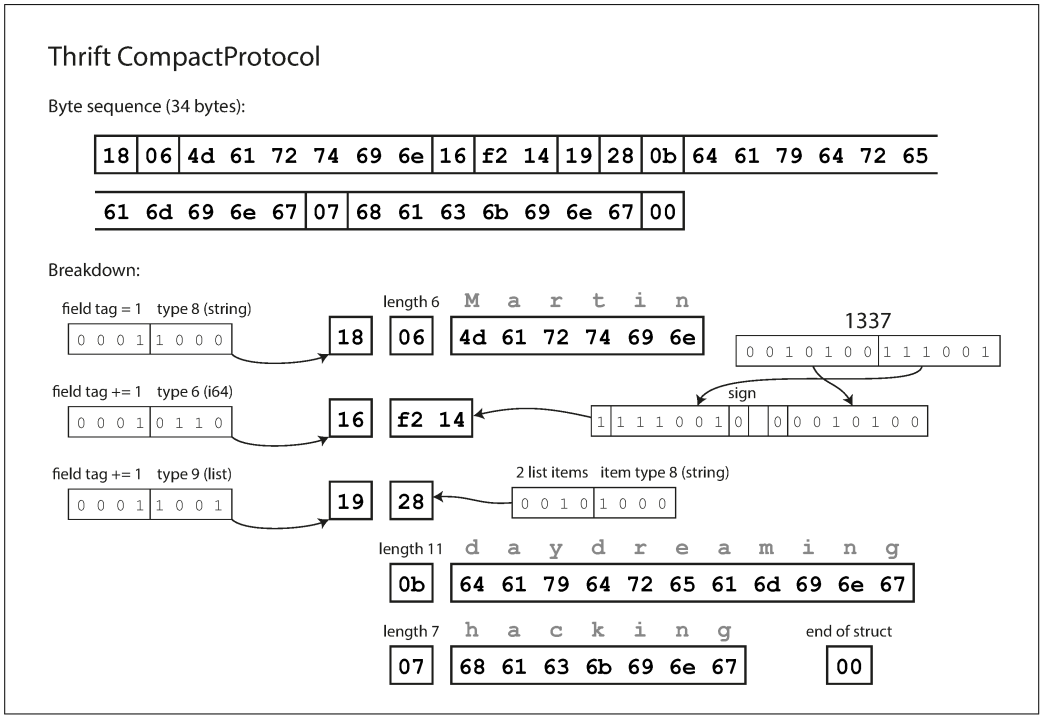


Figure 4-3. Example record encoded using Thrift's CompactProtocol.

Finally, Protocol Buffers (which has only one binary encoding format) encodes the same data as shown in Figure 4-4. It does the bit packing slightly differently, but is

otherwise very similar to Thrift's CompactProtocol. Protocol Buffers fits the same record in 33 bytes.

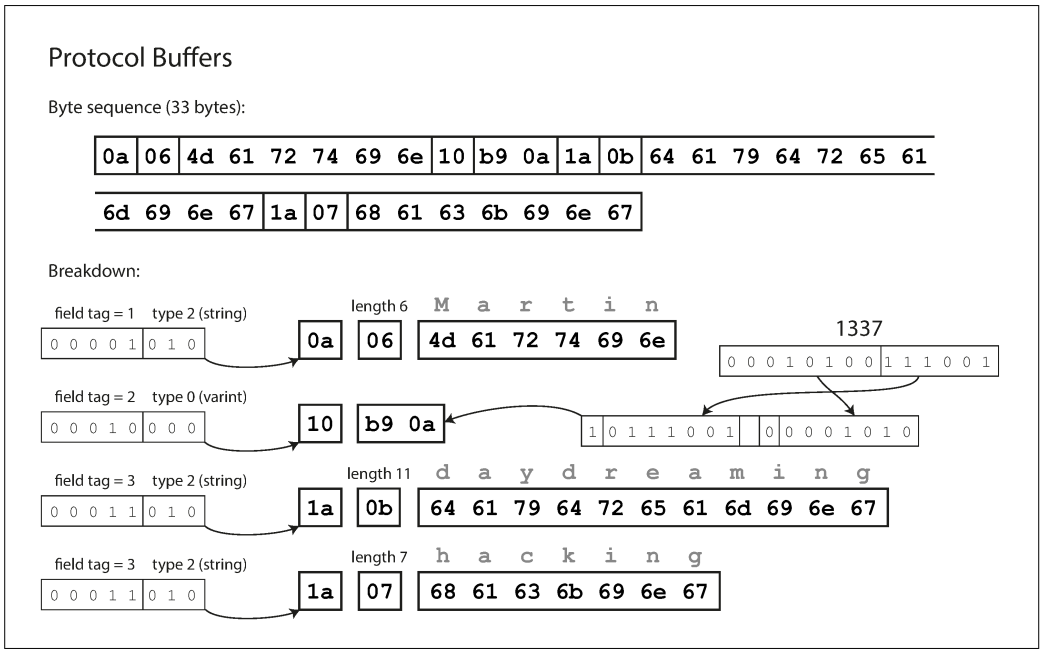


Figure 4-4. Example record encoded using Protocol Buffers.

One detail to note: in the schemas shown earlier, each field was marked either `required` or `optional`, but this makes no difference to how the field is encoded (nothing in the binary data indicates whether a field was required). The difference is simply that `required` enables a runtime check that fails if the field is not set, which can be useful for catching bugs.

Field tags and schema evolution

We said previously that schemas inevitably need to change over time. We call this *schema evolution*. How do Thrift and Protocol Buffers handle schema changes while keeping backward and forward compatibility?

As you can see from the examples, an encoded record is just the concatenation of its encoded fields. Each field is identified by its tag number (the numbers `1`, `2`, `3` in the sample schemas) and annotated with a datatype (e.g., string or integer). If a field value is not set, it is simply omitted from the encoded record. From this you can see that field tags are critical to the meaning of the encoded data. You can change the name of a field in the schema, since the encoded data never refers to field names, but you cannot change a field's tag, since that would make all existing encoded data invalid.

You can add new fields to the schema, provided that you give each field a new tag number. If old code (which doesn't know about the new tag numbers you added) tries to read data written by new code, including a new field with a tag number it doesn't recognize, it can simply ignore that field. The datatype annotation allows the parser to determine how many bytes it needs to skip. This maintains forward compatibility: old code can read records that were written by new code.

What about backward compatibility? As long as each field has a unique tag number, new code can always read old data, because the tag numbers still have the same meaning. The only detail is that if you add a new field, you cannot make it required. If you were to add a field and make it required, that check would fail if new code read data written by old code, because the old code will not have written the new field that you added. Therefore, to maintain backward compatibility, every field you add after the initial deployment of the schema must be optional or have a default value.

Removing a field is just like adding a field, with backward and forward compatibility concerns reversed. That means you can only remove a field that is optional (a required field can never be removed), and you can never use the same tag number again (because you may still have data written somewhere that includes the old tag number, and that field must be ignored by new code).

Datatypes and schema evolution

What about changing the datatype of a field? That may be possible—check the documentation for details—but there is a risk that values will lose precision or get truncated. For example, say you change a 32-bit integer into a 64-bit integer. New code can easily read data written by old code, because the parser can fill in any missing bits with zeros. However, if old code reads data written by new code, the old code is still using a 32-bit variable to hold the value. If the decoded 64-bit value won't fit in 32 bits, it will be truncated.

A curious detail of Protocol Buffers is that it does not have a list or array datatype, but instead has a `repeated` marker for fields (which is a third option alongside `required` and `optional`). As you can see in Figure 4-4, the encoding of a `repeated` field is just what it says on the tin: the same field tag simply appears multiple times in the record. This has the nice effect that it's okay to change an `optional` (single-valued) field into a `repeated` (multi-valued) field. New code reading old data sees a list with zero or one elements (depending on whether the field was present); old code reading new data sees only the last element of the list.

Thrift has a dedicated list datatype, which is parameterized with the datatype of the list elements. This does not allow the same evolution from single-valued to multi-valued as Protocol Buffers does, but it has the advantage of supporting nested lists.

Avro

Apache Avro [20] is another binary encoding format that is interestingly different from Protocol Buffers and Thrift. It was started in 2009 as a subproject of Hadoop, as a result of Thrift not being a good fit for Hadoop's use cases [21].

Avro also uses a schema to specify the structure of the data being encoded. It has two schema languages: one (Avro IDL) intended for human editing, and one (based on JSON) that is more easily machine-readable.

Our example schema, written in Avro IDL, might look like this:

```
record Person {
    string               userName;
    union { null, long } favoriteNumber = null;
    array<string>        interests;
}
```

The equivalent JSON representation of that schema is as follows:

```
{
    "type": "record",
    "name": "Person",
    "fields": [
        {"name": "userName",       "type": "string"},
        {"name": "favoriteNumber", "type": ["null", "long"], "default": null},
        {"name": "interests",      "type": {"type": "array", "items": "string"}}
    ]
}
```

First of all, notice that there are no tag numbers in the schema. If we encode our example record (Example 4-1) using this schema, the Avro binary encoding is just 32 bytes long—the most compact of all the encodings we have seen. The breakdown of the encoded byte sequence is shown in Figure 4-5.

If you examine the byte sequence, you can see that there is nothing to identify fields or their datatypes. The encoding simply consists of values concatenated together. A string is just a length prefix followed by UTF-8 bytes, but there's nothing in the encoded data that tells you that it is a string. It could just as well be an integer, or something else entirely. An integer is encoded using a variable-length encoding (the same as Thrift's CompactProtocol).

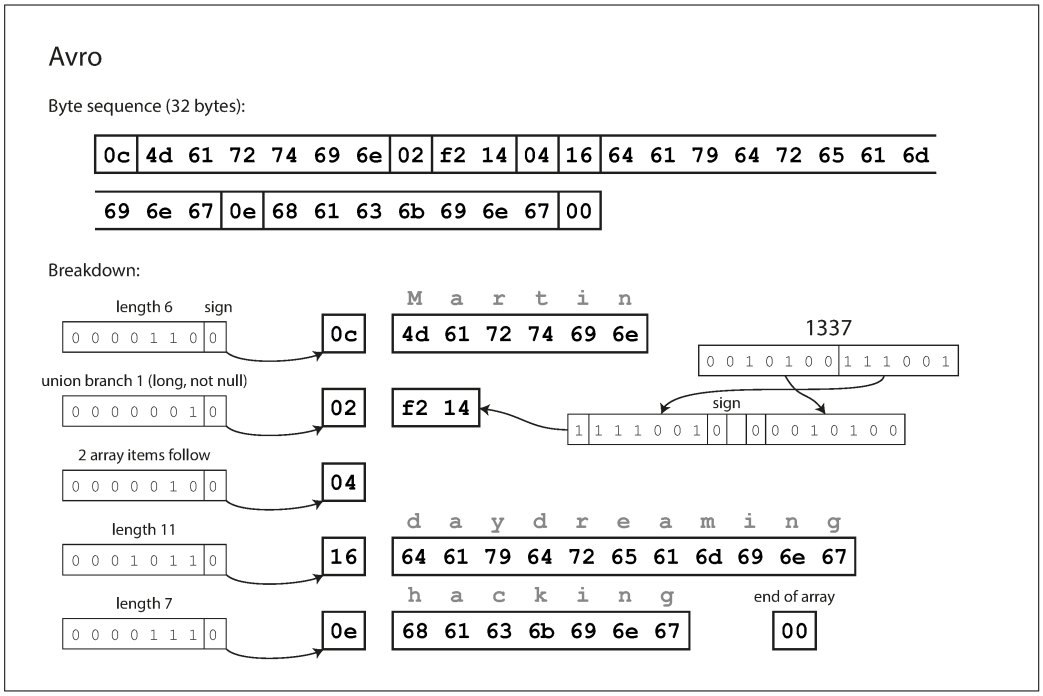


Figure 4-5. Example record encoded using Avro.

To parse the binary data, you go through the fields in the order that they appear in the schema and use the schema to tell you the datatype of each field. This means that the binary data can only be decoded correctly if the code reading the data is using the *exact same schema* as the code that wrote the data. Any mismatch in the schema between the reader and the writer would mean incorrectly decoded data.

So, how does Avro support schema evolution?

The writer's schema and the reader's schema

With Avro, when an application wants to encode some data (to write it to a file or database, to send it over the network, etc.), it encodes the data using whatever version of the schema it knows about—for example, that schema may be compiled into the application. This is known as the *writer's schema*.

When an application wants to decode some data (read it from a file or database, receive it from the network, etc.), it is expecting the data to be in some schema, which is known as the *reader's schema*. That is the schema the application code is relying on —code may have been generated from that schema during the application's build process.

The key idea with Avro is that the writer's schema and the reader's schema *don't have to be the same*—they only need to be compatible. When data is decoded (read), the

Avro library resolves the differences by looking at the writer's schema and the reader's schema side by side and translating the data from the writer's schema into the reader's schema. The Avro specification [20] defines exactly how this resolution works, and it is illustrated in Figure 4-6.

For example, it's no problem if the writer's schema and the reader's schema have their fields in a different order, because the schema resolution matches up the fields by field name. If the code reading the data encounters a field that appears in the writer's schema but not in the reader's schema, it is ignored. If the code reading the data expects some field, but the writer's schema does not contain a field of that name, it is filled in with a default value declared in the reader's schema.

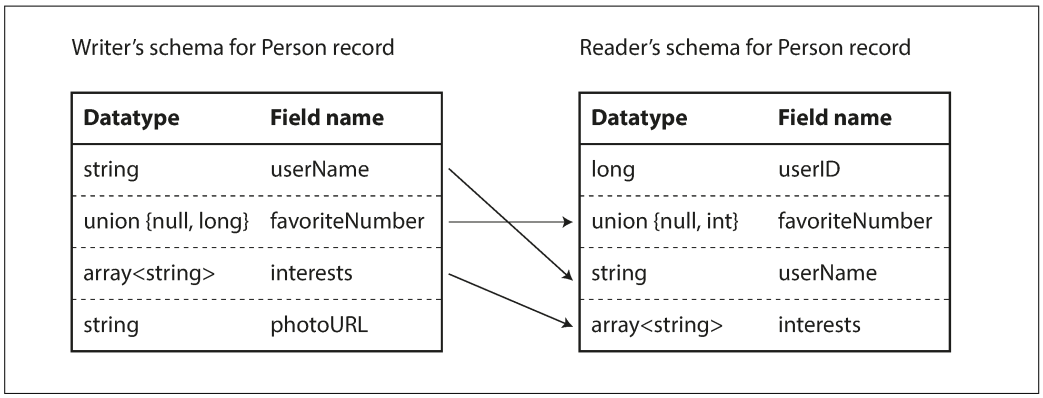


Figure 4-6. An Avro reader resolves differences between the writer's schema and the reader's schema.

Schema evolution rules

With Avro, forward compatibility means that you can have a new version of the schema as writer and an old version of the schema as reader. Conversely, backward compatibility means that you can have a new version of the schema as reader and an old version as writer.

To maintain compatibility, you may only add or remove a field that has a default value. (The field `favoriteNumber` in our Avro schema has a default value of `null`.) For example, say you add a field with a default value, so this new field exists in the new schema but not the old one. When a reader using the new schema reads a record written with the old schema, the default value is filled in for the missing field.

If you were to add a field that has no default value, new readers wouldn't be able to read data written by old writers, so you would break backward compatibility. If you were to remove a field that has no default value, old readers wouldn't be able to read data written by new writers, so you would break forward compatibility.

In some programming languages, `null` is an acceptable default for any variable, but this is not the case in Avro: if you want to allow a field to be null, you have to use a *union type*. For example, `union { null, long, string } field;` indicates that `field` can be a number, or a string, or null. You can only use `null` as a default value if it is one of the branches of the union.[iv] This is a little more verbose than having everything nullable by default, but it helps prevent bugs by being explicit about what can and cannot be null [22].

Consequently, Avro doesn't have `optional` and `required` markers in the same way as Protocol Buffers and Thrift do (it has union types and default values instead).

Changing the datatype of a field is possible, provided that Avro can convert the type. Changing the name of a field is possible but a little tricky: the reader's schema can contain aliases for field names, so it can match an old writer's schema field names against the aliases. This means that changing a field name is backward compatible but not forward compatible. Similarly, adding a branch to a union type is backward compatible but not forward compatible.

But what is the writer's schema?

There is an important question that we've glossed over so far: how does the reader know the writer's schema with which a particular piece of data was encoded? We can't just include the entire schema with every record, because the schema would likely be much bigger than the encoded data, making all the space savings from the binary encoding futile.

The answer depends on the context in which Avro is being used. To give a few examples:

Large file with lots of records
: A common use for Avro—especially in the context of Hadoop—is for storing a large file containing millions of records, all encoded with the same schema. (We will discuss this kind of situation in Chapter 10.) In this case, the writer of that file can just include the writer's schema once at the beginning of the file. Avro specifies a file format (object container files) to do this.

Database with individually written records
: In a database, different records may be written at different points in time using different writer's schemas—you cannot assume that all the records will have the same schema. The simplest solution is to include a version number at the beginning of every encoded record, and to keep a list of schema versions in your data-

iv. To be precise, the default value must be of the type of the *first* branch of the union, although this is a specific limitation of Avro, not a general feature of union types.

base. A reader can fetch a record, extract the version number, and then fetch the writer's schema for that version number from the database. Using that writer's schema, it can decode the rest of the record. (Espresso [23] works this way, for example.)

Sending records over a network connection
: When two processes are communicating over a bidirectional network connection, they can negotiate the schema version on connection setup and then use that schema for the lifetime of the connection. The Avro RPC protocol (see "Dataflow Through Services: REST and RPC" on page 131) works like this.

A database of schema versions is a useful thing to have in any case, since it acts as documentation and gives you a chance to check schema compatibility [24]. As the version number, you could use a simple incrementing integer, or you could use a hash of the schema.

Dynamically generated schemas

One advantage of Avro's approach, compared to Protocol Buffers and Thrift, is that the schema doesn't contain any tag numbers. But why is this important? What's the problem with keeping a couple of numbers in the schema?

The difference is that Avro is friendlier to *dynamically generated* schemas. For example, say you have a relational database whose contents you want to dump to a file, and you want to use a binary format to avoid the aforementioned problems with textual formats (JSON, CSV, SQL). If you use Avro, you can fairly easily generate an Avro schema (in the JSON representation we saw earlier) from the relational schema and encode the database contents using that schema, dumping it all to an Avro object container file [25]. You generate a record schema for each database table, and each column becomes a field in that record. The column name in the database maps to the field name in Avro.

Now, if the database schema changes (for example, a table has one column added and one column removed), you can just generate a new Avro schema from the updated database schema and export data in the new Avro schema. The data export process does not need to pay any attention to the schema change—it can simply do the schema conversion every time it runs. Anyone who reads the new data files will see that the fields of the record have changed, but since the fields are identified by name, the updated writer's schema can still be matched up with the old reader's schema.

By contrast, if you were using Thrift or Protocol Buffers for this purpose, the field tags would likely have to be assigned by hand: every time the database schema changes, an administrator would have to manually update the mapping from database column names to field tags. (It might be possible to automate this, but the schema generator would have to be very careful to not assign previously used field

tags.) This kind of dynamically generated schema simply wasn't a design goal of Thrift or Protocol Buffers, whereas it was for Avro.

Code generation and dynamically typed languages

Thrift and Protocol Buffers rely on code generation: after a schema has been defined, you can generate code that implements this schema in a programming language of your choice. This is useful in statically typed languages such as Java, C++, or C#, because it allows efficient in-memory structures to be used for decoded data, and it allows type checking and autocompletion in IDEs when writing programs that access the data structures.

In dynamically typed programming languages such as JavaScript, Ruby, or Python, there is not much point in generating code, since there is no compile-time type checker to satisfy. Code generation is often frowned upon in these languages, since they otherwise avoid an explicit compilation step. Moreover, in the case of a dynamically generated schema (such as an Avro schema generated from a database table), code generation is an unnecessarily obstacle to getting to the data.

Avro provides optional code generation for statically typed programming languages, but it can be used just as well without any code generation. If you have an object container file (which embeds the writer's schema), you can simply open it using the Avro library and look at the data in the same way as you could look at a JSON file. The file is *self-describing* since it includes all the necessary metadata.

This property is especially useful in conjunction with dynamically typed data processing languages like Apache Pig [26]. In Pig, you can just open some Avro files, start analyzing them, and write derived datasets to output files in Avro format without even thinking about schemas.

The Merits of Schemas

As we saw, Protocol Buffers, Thrift, and Avro all use a schema to describe a binary encoding format. Their schema languages are much simpler than XML Schema or JSON Schema, which support much more detailed validation rules (e.g., "the string value of this field must match this regular expression" or "the integer value of this field must be between 0 and 100"). As Protocol Buffers, Thrift, and Avro are simpler to implement and simpler to use, they have grown to support a fairly wide range of programming languages.

The ideas on which these encodings are based are by no means new. For example, they have a lot in common with ASN.1, a schema definition language that was first standardized in 1984 [27]. It was used to define various network protocols, and its binary encoding (DER) is still used to encode SSL certificates (X.509), for example [28]. ASN.1 supports schema evolution using tag numbers, similar to Protocol Buf-

fers and Thrift [29]. However, it's also very complex and badly documented, so ASN.1 is probably not a good choice for new applications.

Many data systems also implement some kind of proprietary binary encoding for their data. For example, most relational databases have a network protocol over which you can send queries to the database and get back responses. Those protocols are generally specific to a particular database, and the database vendor provides a driver (e.g., using the ODBC or JDBC APIs) that decodes responses from the database's network protocol into in-memory data structures.

So, we can see that although textual data formats such as JSON, XML, and CSV are widespread, binary encodings based on schemas are also a viable option. They have a number of nice properties:

- They can be much more compact than the various "binary JSON" variants, since they can omit field names from the encoded data.
- The schema is a valuable form of documentation, and because the schema is required for decoding, you can be sure that it is up to date (whereas manually maintained documentation may easily diverge from reality).
- Keeping a database of schemas allows you to check forward and backward compatibility of schema changes, before anything is deployed.
- For users of statically typed programming languages, the ability to generate code from the schema is useful, since it enables type checking at compile time.

In summary, schema evolution allows the same kind of flexibility as schemaless/schema-on-read JSON databases provide (see "Schema flexibility in the document model" on page 39), while also providing better guarantees about your data and better tooling.

Modes of Dataflow

At the beginning of this chapter we said that whenever you want to send some data to another process with which you don't share memory—for example, whenever you want to send data over the network or write it to a file—you need to encode it as a sequence of bytes. We then discussed a variety of different encodings for doing this.

We talked about forward and backward compatibility, which are important for evolvability (making change easy by allowing you to upgrade different parts of your system independently, and not having to change everything at once). Compatibility is a relationship between one process that encodes the data, and another process that decodes it.

That's a fairly abstract idea—there are many ways data can flow from one process to another. Who encodes the data, and who decodes it? In the rest of this chapter we will explore some of the most common ways how data flows between processes:

- Via databases (see "Dataflow Through Databases" on page 129)
- Via service calls (see "Dataflow Through Services: REST and RPC" on page 131)
- Via asynchronous message passing (see "Message-Passing Dataflow" on page 136)

Dataflow Through Databases

In a database, the process that writes to the database encodes the data, and the process that reads from the database decodes it. There may just be a single process accessing the database, in which case the reader is simply a later version of the same process—in that case you can think of storing something in the database as *sending a message to your future self*.

Backward compatibility is clearly necessary here; otherwise your future self won't be able to decode what you previously wrote.

In general, it's common for several different processes to be accessing a database at the same time. Those processes might be several different applications or services, or they may simply be several instances of the same service (running in parallel for scalability or fault tolerance). Either way, in an environment where the application is changing, it is likely that some processes accessing the database will be running newer code and some will be running older code—for example because a new version is currently being deployed in a rolling upgrade, so some instances have been updated while others haven't yet.

This means that a value in the database may be written by a *newer* version of the code, and subsequently read by an *older* version of the code that is still running. Thus, forward compatibility is also often required for databases.

However, there is an additional snag. Say you add a field to a record schema, and the newer code writes a value for that new field to the database. Subsequently, an older version of the code (which doesn't yet know about the new field) reads the record, updates it, and writes it back. In this situation, the desirable behavior is usually for the old code to keep the new field intact, even though it couldn't be interpreted.

The encoding formats discussed previously support such preservation of unknown fields, but sometimes you need to take care at an application level, as illustrated in Figure 4-7. For example, if you decode a database value into model objects in the application, and later reencode those model objects, the unknown field might be lost in that translation process. Solving this is not a hard problem; you just need to be aware of it.

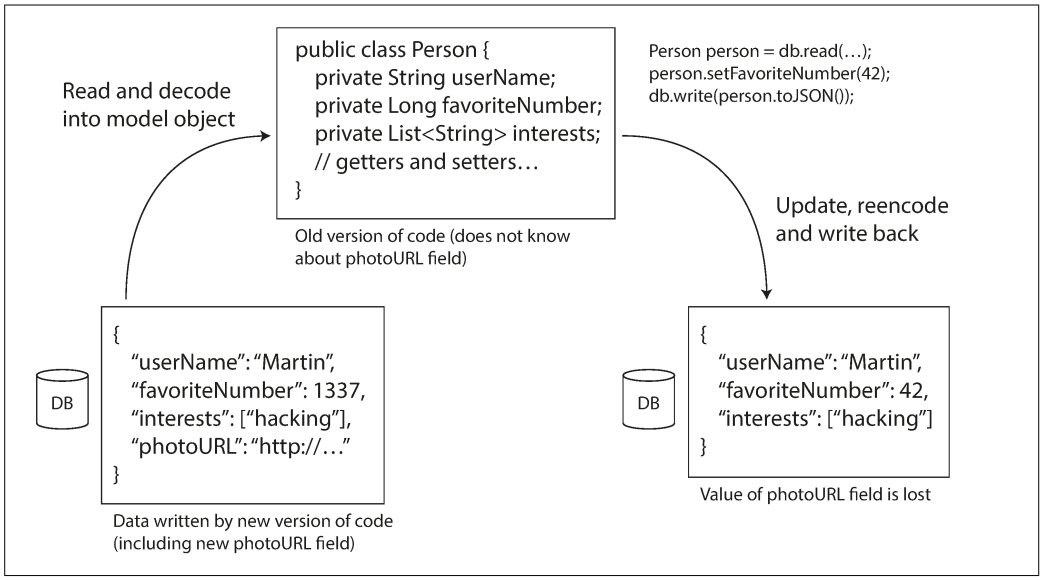


Figure 4-7. When an older version of the application updates data previously written by a newer version of the application, data may be lost if you're not careful.

Different values written at different times

A database generally allows any value to be updated at any time. This means that within a single database you may have some values that were written five milliseconds ago, and some values that were written five years ago.

When you deploy a new version of your application (of a server-side application, at least), you may entirely replace the old version with the new version within a few minutes. The same is not true of database contents: the five-year-old data will still be there, in the original encoding, unless you have explicitly rewritten it since then. This observation is sometimes summed up as *data outlives code*.

Rewriting (*migrating*) data into a new schema is certainly possible, but it's an expensive thing to do on a large dataset, so most databases avoid it if possible. Most relational databases allow simple schema changes, such as adding a new column with a `null` default value, without rewriting existing data.[v] When an old row is read, the database fills in `null`s for any columns that are missing from the encoded data on disk. LinkedIn's document database Espresso uses Avro for storage, allowing it to use Avro's schema evolution rules [23].

v. Except for MySQL, which often rewrites an entire table even though it is not strictly necessary, as mentioned in "Schema flexibility in the document model" on page 39.

Schema evolution thus allows the entire database to appear as if it was encoded with a single schema, even though the underlying storage may contain records encoded with various historical versions of the schema.

Archival storage

Perhaps you take a snapshot of your database from time to time, say for backup purposes or for loading into a data warehouse (see "Data Warehousing" on page 91). In this case, the data dump will typically be encoded using the latest schema, even if the original encoding in the source database contained a mixture of schema versions from different eras. Since you're copying the data anyway, you might as well encode the copy of the data consistently.

As the data dump is written in one go and is thereafter immutable, formats like Avro object container files are a good fit. This is also a good opportunity to encode the data in an analytics-friendly column-oriented format such as Parquet (see "Column Compression" on page 97).

In Chapter 10 we will talk more about using data in archival storage.

Dataflow Through Services: REST and RPC

When you have processes that need to communicate over a network, there are a few different ways of arranging that communication. The most common arrangement is to have two roles: *clients* and *servers*. The servers expose an API over the network, and the clients can connect to the servers to make requests to that API. The API exposed by the server is known as a *service*.

The web works this way: clients (web browsers) make requests to web servers, making `GET` requests to download HTML, CSS, JavaScript, images, etc., and making `POST` requests to submit data to the server. The API consists of a standardized set of protocols and data formats (HTTP, URLs, SSL/TLS, HTML, etc.). Because web browsers, web servers, and website authors mostly agree on these standards, you can use any web browser to access any website (at least in theory!).

Web browsers are not the only type of client. For example, a native app running on a mobile device or a desktop computer can also make network requests to a server, and a client-side JavaScript application running inside a web browser can use XMLHttpRequest to become an HTTP client (this technique is known as *Ajax* [30]). In this case, the server's response is typically not HTML for displaying to a human, but rather data in an encoding that is convenient for further processing by the client-side application code (such as JSON). Although HTTP may be used as the transport protocol, the API implemented on top is application-specific, and the client and server need to agree on the details of that API.

Moreover, a server can itself be a client to another service (for example, a typical web app server acts as client to a database). This approach is often used to decompose a large application into smaller services by area of functionality, such that one service makes a request to another when it requires some functionality or data from that other service. This way of building applications has traditionally been called a *service-oriented architecture* (SOA), more recently refined and rebranded as *microservices architecture* [31, 32].

In some ways, services are similar to databases: they typically allow clients to submit and query data. However, while databases allow arbitrary queries using the query languages we discussed in Chapter 2, services expose an application-specific API that only allows inputs and outputs that are predetermined by the business logic (application code) of the service [33]. This restriction provides a degree of encapsulation: services can impose fine-grained restrictions on what clients can and cannot do.

A key design goal of a service-oriented/microservices architecture is to make the application easier to change and maintain by making services independently deployable and evolvable. For example, each service should be owned by one team, and that team should be able to release new versions of the service frequently, without having to coordinate with other teams. In other words, we should expect old and new versions of servers and clients to be running at the same time, and so the data encoding used by servers and clients must be compatible across versions of the service API—precisely what we've been talking about in this chapter.

Web services

When HTTP is used as the underlying protocol for talking to the service, it is called a *web service*. This is perhaps a slight misnomer, because web services are not only used on the web, but in several different contexts. For example:

1. A client application running on a user's device (e.g., a native app on a mobile device, or JavaScript web app using Ajax) making requests to a service over HTTP. These requests typically go over the public internet.
2. One service making requests to another service owned by the same organization, often located within the same datacenter, as part of a service-oriented/microservices architecture. (Software that supports this kind of use case is sometimes called *middleware*.)
3. One service making requests to a service owned by a different organization, usually via the internet. This is used for data exchange between different organizations' backend systems. This category includes public APIs provided by online services, such as credit card processing systems, or OAuth for shared access to user data.

There are two popular approaches to web services: *REST* and *SOAP*. They are almost diametrically opposed in terms of philosophy, and often the subject of heated debate among their respective proponents.[vi]

REST is not a protocol, but rather a design philosophy that builds upon the principles of HTTP [34, 35]. It emphasizes simple data formats, using URLs for identifying resources and using HTTP features for cache control, authentication, and content type negotiation. REST has been gaining popularity compared to SOAP, at least in the context of cross-organizational service integration [36], and is often associated with microservices [31]. An API designed according to the principles of REST is called *RESTful*.

By contrast, SOAP is an XML-based protocol for making network API requests.[vii] Although it is most commonly used over HTTP, it aims to be independent from HTTP and avoids using most HTTP features. Instead, it comes with a sprawling and complex multitude of related standards (the *web service framework*, known as *WS-**) that add various features [37].

The API of a SOAP web service is described using an XML-based language called the Web Services Description Language, or WSDL. WSDL enables code generation so that a client can access a remote service using local classes and method calls (which are encoded to XML messages and decoded again by the framework). This is useful in statically typed programming languages, but less so in dynamically typed ones (see "Code generation and dynamically typed languages" on page 127).

As WSDL is not designed to be human-readable, and as SOAP messages are often too complex to construct manually, users of SOAP rely heavily on tool support, code generation, and IDEs [38]. For users of programming languages that are not supported by SOAP vendors, integration with SOAP services is difficult.

Even though SOAP and its various extensions are ostensibly standardized, interoperability between different vendors' implementations often causes problems [39]. For all of these reasons, although SOAP is still used in many large enterprises, it has fallen out of favor in most smaller companies.

RESTful APIs tend to favor simpler approaches, typically involving less code generation and automated tooling. A definition format such as OpenAPI, also known as Swagger [40], can be used to describe RESTful APIs and produce documentation.

---

vi. Even within each camp there are plenty of arguments. For example, HATEOAS (*hypermedia as the engine of application state*), often provokes discussions [35].

vii. Despite the similarity of acronyms, SOAP is not a requirement for SOA. SOAP is a particular technology, whereas SOA is a general approach to building systems.

The problems with remote procedure calls (RPCs)

Web services are merely the latest incarnation of a long line of technologies for making API requests over a network, many of which received a lot of hype but have serious problems. Enterprise JavaBeans (EJB) and Java's Remote Method Invocation (RMI) are limited to Java. The Distributed Component Object Model (DCOM) is limited to Microsoft platforms. The Common Object Request Broker Architecture (CORBA) is excessively complex, and does not provide backward or forward compatibility [41].

All of these are based on the idea of a *remote procedure call* (RPC), which has been around since the 1970s [42]. The RPC model tries to make a request to a remote network service look the same as calling a function or method in your programming language, within the same process (this abstraction is called *location transparency*). Although RPC seems convenient at first, the approach is fundamentally flawed [43, 44]. A network request is very different from a local function call:

- A local function call is predictable and either succeeds or fails, depending only on parameters that are under your control. A network request is unpredictable: the request or response may be lost due to a network problem, or the remote machine may be slow or unavailable, and such problems are entirely outside of your control. Network problems are common, so you have to anticipate them, for example by retrying a failed request.
- A local function call either returns a result, or throws an exception, or never returns (because it goes into an infinite loop or the process crashes). A network request has another possible outcome: it may return without a result, due to a *timeout*. In that case, you simply don't know what happened: if you don't get a response from the remote service, you have no way of knowing whether the request got through or not. (We discuss this issue in more detail in Chapter 8.)
- If you retry a failed network request, it could happen that the requests are actually getting through, and only the responses are getting lost. In that case, retrying will cause the action to be performed multiple times, unless you build a mechanism for deduplication (*idempotence*) into the protocol. Local function calls don't have this problem. (We discuss idempotence in more detail in Chapter 11.)
- Every time you call a local function, it normally takes about the same time to execute. A network request is much slower than a function call, and its latency is also wildly variable: at good times it may complete in less than a millisecond, but when the network is congested or the remote service is overloaded it may take many seconds to do exactly the same thing.
- When you call a local function, you can efficiently pass it references (pointers) to objects in local memory. When you make a network request, all those parameters

need to be encoded into a sequence of bytes that can be sent over the network. That's okay if the parameters are primitives like numbers or strings, but quickly becomes problematic with larger objects.

- The client and the service may be implemented in different programming languages, so the RPC framework must translate datatypes from one language into another. This can end up ugly, since not all languages have the same types—recall JavaScript's problems with numbers greater than 2^{53}, for example (see "JSON, XML, and Binary Variants" on page 114). This problem doesn't exist in a single process written in a single language.

All of these factors mean that there's no point trying to make a remote service look too much like a local object in your programming language, because it's a fundamentally different thing. Part of the appeal of REST is that it doesn't try to hide the fact that it's a network protocol (although this doesn't seem to stop people from building RPC libraries on top of REST).

Current directions for RPC

Despite all these problems, RPC isn't going away. Various RPC frameworks have been built on top of all the encodings mentioned in this chapter: for example, Thrift and Avro come with RPC support included, gRPC is an RPC implementation using Protocol Buffers, Finagle also uses Thrift, and Rest.li uses JSON over HTTP.

This new generation of RPC frameworks is more explicit about the fact that a remote request is different from a local function call. For example, Finagle and Rest.li use *futures* (*promises*) to encapsulate asynchronous actions that may fail. Futures also simplify situations where you need to make requests to multiple services in parallel, and combine their results [45]. gRPC supports *streams*, where a call consists of not just one request and one response, but a series of requests and responses over time [46].

Some of these frameworks also provide *service discovery*—that is, allowing a client to find out at which IP address and port number it can find a particular service. We will return to this topic in "Request Routing" on page 214.

Custom RPC protocols with a binary encoding format can achieve better performance than something generic like JSON over REST. However, a RESTful API has other significant advantages: it is good for experimentation and debugging (you can simply make requests to it using a web browser or the command-line tool `curl`, without any code generation or software installation), it is supported by all mainstream programming languages and platforms, and there is a vast ecosystem of tools available (servers, caches, load balancers, proxies, firewalls, monitoring, debugging tools, testing tools, etc.).

For these reasons, REST seems to be the predominant style for public APIs. The main focus of RPC frameworks is on requests between services owned by the same organization, typically within the same datacenter.

Data encoding and evolution for RPC

For evolvability, it is important that RPC clients and servers can be changed and deployed independently. Compared to data flowing through databases (as described in the last section), we can make a simplifying assumption in the case of dataflow through services: it is reasonable to assume that all the servers will be updated first, and all the clients second. Thus, you only need backward compatibility on requests, and forward compatibility on responses.

The backward and forward compatibility properties of an RPC scheme are inherited from whatever encoding it uses:

- Thrift, gRPC (Protocol Buffers), and Avro RPC can be evolved according to the compatibility rules of the respective encoding format.
- In SOAP, requests and responses are specified with XML schemas. These can be evolved, but there are some subtle pitfalls [47].
- RESTful APIs most commonly use JSON (without a formally specified schema) for responses, and JSON or URI-encoded/form-encoded request parameters for requests. Adding optional request parameters and adding new fields to response objects are usually considered changes that maintain compatibility.

Service compatibility is made harder by the fact that RPC is often used for communication across organizational boundaries, so the provider of a service often has no control over its clients and cannot force them to upgrade. Thus, compatibility needs to be maintained for a long time, perhaps indefinitely. If a compatibility-breaking change is required, the service provider often ends up maintaining multiple versions of the service API side by side.

There is no agreement on how API versioning should work (i.e., how a client can indicate which version of the API it wants to use [48]). For RESTful APIs, common approaches are to use a version number in the URL or in the HTTP `Accept` header. For services that use API keys to identify a particular client, another option is to store a client's requested API version on the server and to allow this version selection to be updated through a separate administrative interface [49].

Message-Passing Dataflow

We have been looking at the different ways encoded data flows from one process to another. So far, we've discussed REST and RPC (where one process sends a request over the network to another process and expects a response as quickly as possible),

and databases (where one process writes encoded data, and another process reads it again sometime in the future).

In this final section, we will briefly look at *asynchronous message-passing* systems, which are somewhere between RPC and databases. They are similar to RPC in that a client's request (usually called a *message*) is delivered to another process with low latency. They are similar to databases in that the message is not sent via a direct network connection, but goes via an intermediary called a *message broker* (also called a *message queue* or *message-oriented middleware*), which stores the message temporarily.

Using a message broker has several advantages compared to direct RPC:

- It can act as a buffer if the recipient is unavailable or overloaded, and thus improve system reliability.
- It can automatically redeliver messages to a process that has crashed, and thus prevent messages from being lost.
- It avoids the sender needing to know the IP address and port number of the recipient (which is particularly useful in a cloud deployment where virtual machines often come and go).
- It allows one message to be sent to several recipients.
- It logically decouples the sender from the recipient (the sender just publishes messages and doesn't care who consumes them).

However, a difference compared to RPC is that message-passing communication is usually one-way: a sender normally doesn't expect to receive a reply to its messages. It is possible for a process to send a response, but this would usually be done on a separate channel. This communication pattern is *asynchronous*: the sender doesn't wait for the message to be delivered, but simply sends it and then forgets about it.

Message brokers

In the past, the landscape of message brokers was dominated by commercial enterprise software from companies such as TIBCO, IBM WebSphere, and webMethods. More recently, open source implementations such as RabbitMQ, ActiveMQ, HornetQ, NATS, and Apache Kafka have become popular. We will compare them in more detail in Chapter 11.

The detailed delivery semantics vary by implementation and configuration, but in general, message brokers are used as follows: one process sends a message to a named *queue* or *topic*, and the broker ensures that the message is delivered to one or more *consumers* of or *subscribers* to that queue or topic. There can be many producers and many consumers on the same topic.

A topic provides only one-way dataflow. However, a consumer may itself publish messages to another topic (so you can chain them together, as we shall see in Chapter 11), or to a reply queue that is consumed by the sender of the original message (allowing a request/response dataflow, similar to RPC).

Message brokers typically don't enforce any particular data model—a message is just a sequence of bytes with some metadata, so you can use any encoding format. If the encoding is backward and forward compatible, you have the greatest flexibility to change publishers and consumers independently and deploy them in any order.

If a consumer republishes messages to another topic, you may need to be careful to preserve unknown fields, to prevent the issue described previously in the context of databases (Figure 4-7).

Distributed actor frameworks

The *actor model* is a programming model for concurrency in a single process. Rather than dealing directly with threads (and the associated problems of race conditions, locking, and deadlock), logic is encapsulated in *actors*. Each actor typically represents one client or entity, it may have some local state (which is not shared with any other actor), and it communicates with other actors by sending and receiving asynchronous messages. Message delivery is not guaranteed: in certain error scenarios, messages will be lost. Since each actor processes only one message at a time, it doesn't need to worry about threads, and each actor can be scheduled independently by the framework.

In *distributed actor frameworks*, this programming model is used to scale an application across multiple nodes. The same message-passing mechanism is used, no matter whether the sender and recipient are on the same node or different nodes. If they are on different nodes, the message is transparently encoded into a byte sequence, sent over the network, and decoded on the other side.

Location transparency works better in the actor model than in RPC, because the actor model already assumes that messages may be lost, even within a single process. Although latency over the network is likely higher than within the same process, there is less of a fundamental mismatch between local and remote communication when using the actor model.

A distributed actor framework essentially integrates a message broker and the actor programming model into a single framework. However, if you want to perform rolling upgrades of your actor-based application, you still have to worry about forward and backward compatibility, as messages may be sent from a node running the new version to a node running the old version, and vice versa.

Three popular distributed actor frameworks handle message encoding as follows:

- *Akka* uses Java's built-in serialization by default, which does not provide forward or backward compatibility. However, you can replace it with something like Protocol Buffers, and thus gain the ability to do rolling upgrades [50].
- *Orleans* by default uses a custom data encoding format that does not support rolling upgrade deployments; to deploy a new version of your application, you need to set up a new cluster, move traffic from the old cluster to the new one, and shut down the old one [51, 52]. Like with Akka, custom serialization plug-ins can be used.
- In *Erlang OTP* it is surprisingly hard to make changes to record schemas (despite the system having many features designed for high availability); rolling upgrades are possible but need to be planned carefully [53]. An experimental new `maps` datatype (a JSON-like structure, introduced in Erlang R17 in 2014) may make this easier in the future [54].

Summary

In this chapter we looked at several ways of turning data structures into bytes on the network or bytes on disk. We saw how the details of these encodings affect not only their efficiency, but more importantly also the architecture of applications and your options for deploying them.

In particular, many services need to support rolling upgrades, where a new version of a service is gradually deployed to a few nodes at a time, rather than deploying to all nodes simultaneously. Rolling upgrades allow new versions of a service to be released without downtime (thus encouraging frequent small releases over rare big releases) and make deployments less risky (allowing faulty releases to be detected and rolled back before they affect a large number of users). These properties are hugely beneficial for *evolvability*, the ease of making changes to an application.

During rolling upgrades, or for various other reasons, we must assume that different nodes are running the different versions of our application's code. Thus, it is important that all data flowing around the system is encoded in a way that provides backward compatibility (new code can read old data) and forward compatibility (old code can read new data).

We discussed several data encoding formats and their compatibility properties:

- Programming language–specific encodings are restricted to a single programming language and often fail to provide forward and backward compatibility.
- Textual formats like JSON, XML, and CSV are widespread, and their compatibility depends on how you use them. They have optional schema languages, which are sometimes helpful and sometimes a hindrance. These formats are somewhat

vague about datatypes, so you have to be careful with things like numbers and binary strings.

- Binary schema–driven formats like Thrift, Protocol Buffers, and Avro allow compact, efficient encoding with clearly defined forward and backward compatibility semantics. The schemas can be useful for documentation and code generation in statically typed languages. However, they have the downside that data needs to be decoded before it is human-readable.

We also discussed several modes of dataflow, illustrating different scenarios in which data encodings are important:

- Databases, where the process writing to the database encodes the data and the process reading from the database decodes it
- RPC and REST APIs, where the client encodes a request, the server decodes the request and encodes a response, and the client finally decodes the response
- Asynchronous message passing (using message brokers or actors), where nodes communicate by sending each other messages that are encoded by the sender and decoded by the recipient

We can conclude that with a bit of care, backward/forward compatibility and rolling upgrades are quite achievable. May your application's evolution be rapid and your deployments be frequent.

References

[1] "Java Object Serialization Specification," *docs.oracle.com*, 2010.

[2] "Ruby 2.2.0 API Documentation," *ruby-doc.org*, Dec 2014.

[3] "The Python 3.4.3 Standard Library Reference Manual," *docs.python.org*, February 2015.

[4] "EsotericSoftware/kryo," *github.com*, October 2014.

[5] "CWE-502: Deserialization of Untrusted Data," Common Weakness Enumeration, *cwe.mitre.org*, July 30, 2014.

[6] Steve Breen: "What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability," *foxglovesecurity.com*, November 6, 2015.

[7] Patrick McKenzie: "What the Rails Security Issue Means for Your Startup," *kalzumeus.com*, January 31, 2013.

[8] Eishay Smith: "jvm-serializers wiki," *github.com*, November 2014.

[9] "XML Is a Poor Copy of S-Expressions," *c2.com* wiki.

[10] Matt Harris: "Snowflake: An Update and Some Very Important Information," email to *Twitter Development Talk* mailing list, October 19, 2010.

[11] Shudi (Sandy) Gao, C. M. Sperberg-McQueen, and Henry S. Thompson: "XML Schema 1.1," W3C Recommendation, May 2001.

[12] Francis Galiegue, Kris Zyp, and Gary Court: "JSON Schema," IETF Internet-Draft, February 2013.

[13] Yakov Shafranovich: "RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files," October 2005.

[14] "MessagePack Specification," *msgpack.org*.

[15] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski: "Thrift: Scalable Cross-Language Services Implementation," Facebook technical report, April 2007.

[16] "Protocol Buffers Developer Guide," Google, Inc., *developers.google.com*.

[17] Igor Anishchenko: "Thrift vs Protocol Buffers vs Avro - Biased Comparison," *slideshare.net*, September 17, 2012.

[18] "A Matrix of the Features Each Individual Language Library Supports," *wiki.apache.org*.

[19] Martin Kleppmann: "Schema Evolution in Avro, Protocol Buffers and Thrift," *martin.kleppmann.com*, December 5, 2012.

[20] "Apache Avro 1.7.7 Documentation," *avro.apache.org*, July 2014.

[21] Doug Cutting, Chad Walters, Jim Kellerman, et al.: "[PROPOSAL] New Subproject: Avro," email thread on *hadoop-general* mailing list, *mail-archives.apache.org*, April 2009.

[22] Tony Hoare: "Null References: The Billion Dollar Mistake," at *QCon London*, March 2009.

[23] Aditya Auradkar and Tom Quiggle: "Introducing Espresso—LinkedIn's Hot New Distributed Document Store," *engineering.linkedin.com*, January 21, 2015.

[24] Jay Kreps: "Putting Apache Kafka to Use: A Practical Guide to Building a Stream Data Platform (Part 2)," *blog.confluent.io*, February 25, 2015.

[25] Gwen Shapira: "The Problem of Managing Schemas," *radar.oreilly.com*, November 4, 2014.

[26] "Apache Pig 0.14.0 Documentation," *pig.apache.org*, November 2014.

[27] John Larmouth: *ASN.1 Complete*. Morgan Kaufmann, 1999. ISBN: 978-0-122-33435-1

[28] Russell Housley, Warwick Ford, Tim Polk, and David Solo: "RFC 2459: Internet X.509 Public Key Infrastructure: Certificate and CRL Profile," IETF Network Working Group, Standards Track, January 1999.

[29] Lev Walkin: "Question: Extensibility and Dropping Fields," *lionet.info*, September 21, 2010.

[30] Jesse James Garrett: "Ajax: A New Approach to Web Applications," *adaptivepath.com*, February 18, 2005.

[31] Sam Newman: *Building Microservices*. O'Reilly Media, 2015. ISBN: 978-1-491-95035-7

[32] Chris Richardson: "Microservices: Decomposing Applications for Deployability and Scalability," *infoq.com*, May 25, 2014.

[33] Pat Helland: "Data on the Outside Versus Data on the Inside," at *2nd Biennial Conference on Innovative Data Systems Research* (CIDR), January 2005.

[34] Roy Thomas Fielding: "Architectural Styles and the Design of Network-Based Software Architectures," PhD Thesis, University of California, Irvine, 2000.

[35] Roy Thomas Fielding: "REST APIs Must Be Hypertext-Driven," *roy.gbiv.com*, October 20 2008.

[36] "REST in Peace, SOAP," *royal.pingdom.com*, October 15, 2010.

[37] "Web Services Standards as of Q1 2007," *innoq.com*, February 2007.

[38] Pete Lacey: "The S Stands for Simple," *harmful.cat-v.org*, November 15, 2006.

[39] Stefan Tilkov: "Interview: Pete Lacey Criticizes Web Services," *infoq.com*, December 12, 2006.

[40] "OpenAPI Specification (fka Swagger RESTful API Documentation Specification) Version 2.0," *swagger.io*, September 8, 2014.

[41] Michi Henning: "The Rise and Fall of CORBA," *ACM Queue*, volume 4, number 5, pages 28–34, June 2006. doi:10.1145/1142031.1142044

[42] Andrew D. Birrell and Bruce Jay Nelson: "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems* (TOCS), volume 2, number 1, pages 39–59, February 1984. doi:10.1145/2080.357392

[43] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall: "A Note on Distributed Computing," Sun Microsystems Laboratories, Inc., Technical Report TR-94-29, November 1994.

[44] Steve Vinoski: "Convenience over Correctness," *IEEE Internet Computing*, volume 12, number 4, pages 89–92, July 2008. doi:10.1109/MIC.2008.75

[45] Marius Eriksen: "Your Server as a Function," at *7th Workshop on Programming Languages and Operating Systems* (PLOS), November 2013. doi: 10.1145/2525528.2525538

[46] "grpc-common Documentation," Google, Inc., *github.com*, February 2015.

[47] Aditya Narayan and Irina Singh: "Designing and Versioning Compatible Web Services," *ibm.com*, March 28, 2007.

[48] Troy Hunt: "Your API Versioning Is Wrong, Which Is Why I Decided to Do It 3 Different Wrong Ways," *troyhunt.com*, February 10, 2014.

[49] "API Upgrades," Stripe, Inc., April 2015.

[50] Jonas Bonér: "Upgrade in an Akka Cluster," email to *akka-user* mailing list, *groups.google.com*, August 28, 2013.

[51] Philip A. Bernstein, Sergey Bykov, Alan Geller, et al.: "Orleans: Distributed Virtual Actors for Programmability and Scalability," Microsoft Research Technical Report MSR-TR-2014-41, March 2014.

[52] "Microsoft Project Orleans Documentation," Microsoft Research, *dotnet.github.io*, 2015.

[53] David Mercer, Sean Hinde, Yinso Chen, and Richard A O'Keefe: "beginner: Updating Data Structures," email thread on *erlang-questions* mailing list, *erlang.com*, October 29, 2007.

[54] Fred Hebert: "Postscript: Maps," *learnyousomeerlang.com*, April 9, 2014.

PART II
Distributed Data

> *For a successful technology, reality must take precedence over public relations, for nature cannot be fooled.*
>
> —Richard Feynman, *Rogers Commission Report* (1986)

In Part I of this book, we discussed aspects of data systems that apply when data is stored on a single machine. Now, in Part II, we move up a level and ask: what happens if multiple machines are involved in storage and retrieval of data?

There are various reasons why you might want to distribute a database across multiple machines:

Scalability
: If your data volume, read load, or write load grows bigger than a single machine can handle, you can potentially spread the load across multiple machines.

Fault tolerance/high availability
: If your application needs to continue working even if one machine (or several machines, or the network, or an entire datacenter) goes down, you can use multiple machines to give you redundancy. When one fails, another one can take over.

Latency
: If you have users around the world, you might want to have servers at various locations worldwide so that each user can be served from a datacenter that is geographically close to them. That avoids the users having to wait for network packets to travel halfway around the world.

Scaling to Higher Load

If all you need is to scale to higher load, the simplest approach is to buy a more powerful machine (sometimes called *vertical scaling* or *scaling up*). Many CPUs, many RAM chips, and many disks can be joined together under one operating system, and a fast interconnect allows any CPU to access any part of the memory or disk. In this kind of *shared-memory architecture*, all the components can be treated as a single machine [1].[i]

The problem with a shared-memory approach is that the cost grows faster than linearly: a machine with twice as many CPUs, twice as much RAM, and twice as much disk capacity as another typically costs significantly more than twice as much. And due to bottlenecks, a machine twice the size cannot necessarily handle twice the load.

A shared-memory architecture may offer limited fault tolerance—high-end machines have hot-swappable components (you can replace disks, memory modules, and even CPUs without shutting down the machines)—but it is definitely limited to a single geographic location.

Another approach is the *shared-disk architecture*, which uses several machines with independent CPUs and RAM, but stores data on an array of disks that is shared between the machines, which are connected via a fast network.[ii] This architecture is used for some data warehousing workloads, but contention and the overhead of locking limit the scalability of the shared-disk approach [2].

Shared-Nothing Architectures

By contrast, *shared-nothing architectures* [3] (sometimes called *horizontal scaling* or *scaling out*) have gained a lot of popularity. In this approach, each machine or virtual machine running the database software is called a *node*. Each node uses its CPUs, RAM, and disks independently. Any coordination between nodes is done at the software level, using a conventional network.

No special hardware is required by a shared-nothing system, so you can use whatever machines have the best price/performance ratio. You can potentially distribute data across multiple geographic regions, and thus reduce latency for users and potentially be able to survive the loss of an entire datacenter. With cloud deployments of virtual

---

i. In a large machine, although any CPU can access any part of memory, some banks of memory are closer to one CPU than to others (this is called *nonuniform memory access*, or NUMA [1]). To make efficient use of this architecture, the processing needs to be broken down so that each CPU mostly accesses memory that is nearby—which means that partitioning is still required, even when ostensibly running on one machine.

ii. *Network Attached Storage* (NAS) or *Storage Area Network* (SAN).

machines, you don't need to be operating at Google scale: even for small companies, a multi-region distributed architecture is now feasible.

In this part of the book, we focus on shared-nothing architectures—not because they are necessarily the best choice for every use case, but rather because they require the most caution from you, the application developer. If your data is distributed across multiple nodes, you need to be aware of the constraints and trade-offs that occur in such a distributed system—the database cannot magically hide these from you.

While a distributed shared-nothing architecture has many advantages, it usually also incurs additional complexity for applications and sometimes limits the expressiveness of the data models you can use. In some cases, a simple single-threaded program can perform significantly better than a cluster with over 100 CPU cores [4]. On the other hand, shared-nothing systems can be very powerful. The next few chapters go into details on the issues that arise when data is distributed.

Replication Versus Partitioning

There are two common ways data is distributed across multiple nodes:

Replication
: Keeping a copy of the same data on several different nodes, potentially in different locations. Replication provides redundancy: if some nodes are unavailable, the data can still be served from the remaining nodes. Replication can also help improve performance. We discuss replication in Chapter 5.

Partitioning
: Splitting a big database into smaller subsets called *partitions* so that different partitions can be assigned to different nodes (also known as *sharding*). We discuss partitioning in Chapter 6.

These are separate mechanisms, but they often go hand in hand, as illustrated in Figure II-1.

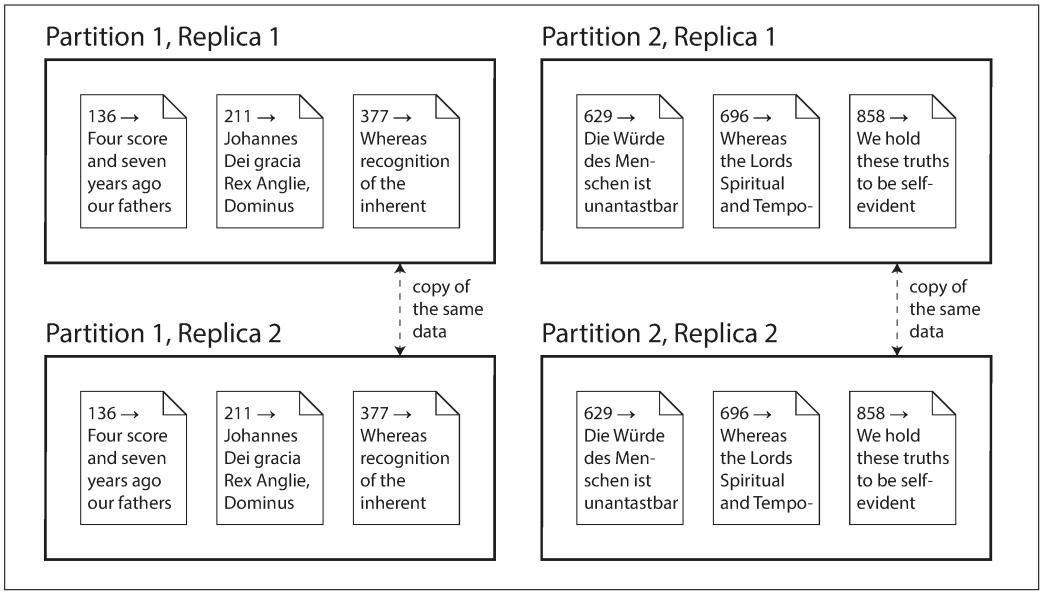


Figure II-1. A database split into two partitions, with two replicas per partition.

With an understanding of those concepts, we can discuss the difficult trade-offs that you need to make in a distributed system. We'll discuss *transactions* in Chapter 7, as that will help you understand all the many things that can go wrong in a data system, and what you can do about them. We'll conclude this part of the book by discussing the fundamental limitations of distributed systems in Chapters 8 and 9.

Later, in Part III of this book, we will discuss how you can take several (potentially distributed) datastores and integrate them into a larger system, satisfying the needs of a complex application. But first, let's talk about distributed data.

References

[1] Ulrich Drepper: "What Every Programmer Should Know About Memory," *akkadia.org*, November 21, 2007.

[2] Ben Stopford: "Shared Nothing vs. Shared Disk Architectures: An Independent View," *benstopford.com*, November 24, 2009.

[3] Michael Stonebraker: "The Case for Shared Nothing," *IEEE Database Engineering Bulletin*, volume 9, number 1, pages 4–9, March 1986.

[4] Frank McSherry, Michael Isard, and Derek G. Murray: "Scalability! But at What COST?," at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.


LOGICAL COAST
RethinkDB
MongoDB
To Consistency & Consensus (Chapter 9)
MySQL
SINGLE-LEADER REPLICATION
FOREST OF CONSISTENCY MODELS
Oracle
PostgreSQL
PHYSICAL COAST
SQL Server
PIT OF FAILOVER
REPLICATION LAG HIGHWAY
BAY OF CAUSALITY
Etherpad
PINNACLES OF CONFLICT RESOLUTION
Google Docs
MULTI-LEADER REPLICATION
QUORUM HARBOR
CouchDB
CRDTs
Calendar sync
Riak
LEADERLESS REPLICATION
EVENTUAL CONSISTENCY BOULEVARD
Voldemort
Cassandra
To Log-structured Storage (Chapter 3)

CHAPTER 5
Replication

> *The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.*
>
> —Douglas Adams, *Mostly Harmless* (1992)

Replication means keeping a copy of the same data on multiple machines that are connected via a network. As discussed in the introduction to Part II, there are several reasons why you might want to replicate data:

- To keep data geographically close to your users (and thus reduce latency)
- To allow the system to continue working even if some of its parts have failed (and thus increase availability)
- To scale out the number of machines that can serve read queries (and thus increase read throughput)

In this chapter we will assume that your dataset is so small that each machine can hold a copy of the entire dataset. In Chapter 6 we will relax that assumption and discuss *partitioning* (*sharding*) of datasets that are too big for a single machine. In later chapters we will discuss various kinds of faults that can occur in a replicated data system, and how to deal with them.

If the data that you're replicating does not change over time, then replication is easy: you just need to copy the data to every node once, and you're done. All of the difficulty in replication lies in handling *changes* to replicated data, and that's what this chapter is about. We will discuss three popular algorithms for replicating changes between nodes: *single-leader*, *multi-leader*, and *leaderless* replication. Almost all distributed databases use one of these three approaches. They all have various pros and cons, which we will examine in detail.

There are many trade-offs to consider with replication: for example, whether to use synchronous or asynchronous replication, and how to handle failed replicas. Those are often configuration options in databases, and although the details vary by database, the general principles are similar across many different implementations. We will discuss the consequences of such choices in this chapter.

Replication of databases is an old topic—the principles haven't changed much since they were studied in the 1970s [1], because the fundamental constraints of networks have remained the same. However, outside of research, many developers continued to assume for a long time that a database consisted of just one node. Mainstream use of distributed databases is more recent. Since many application developers are new to this area, there has been a lot of misunderstanding around issues such as *eventual consistency*. In "Problems with Replication Lag" on page 161 we will get more precise about eventual consistency and discuss things like the *read-your-writes* and *monotonic reads* guarantees.

Leaders and Followers

Each node that stores a copy of the database is called a *replica*. With multiple replicas, a question inevitably arises: how do we ensure that all the data ends up on all the replicas?

Every write to the database needs to be processed by every replica; otherwise, the replicas would no longer contain the same data. The most common solution for this is called *leader-based replication* (also known as *active/passive* or *master–slave replication*) and is illustrated in Figure 5-1. It works as follows:

1. One of the replicas is designated the *leader* (also known as *master* or *primary*). When clients want to write to the database, they must send their requests to the leader, which first writes the new data to its local storage.
2. The other replicas are known as *followers* (*read replicas*, *slaves*, *secondaries*, or *hot standbys*).[i] Whenever the leader writes new data to its local storage, it also sends the data change to all of its followers as part of a *replication log* or *change stream*. Each follower takes the log from the leader and updates its local copy of the database accordingly, by applying all writes in the same order as they were processed on the leader.

i. Different people have different definitions for *hot*, *warm*, and *cold* standby servers. In PostgreSQL, for example, *hot standby* is used to refer to a replica that accepts reads from clients, whereas a *warm standby* processes changes from the leader but doesn't process any queries from clients. For purposes of this book, the difference isn't important.

3. When a client wants to read from the database, it can query either the leader or any of the followers. However, writes are only accepted on the leader (the followers are read-only from the client's point of view).

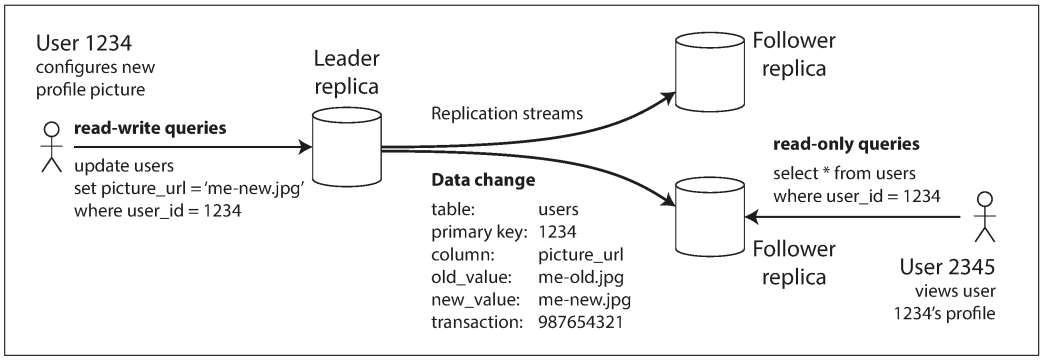


Figure 5-1. Leader-based (master–slave) replication.

This mode of replication is a built-in feature of many relational databases, such as PostgreSQL (since version 9.0), MySQL, Oracle Data Guard [2], and SQL Server's AlwaysOn Availability Groups [3]. It is also used in some nonrelational databases, including MongoDB, RethinkDB, and Espresso [4]. Finally, leader-based replication is not restricted to only databases: distributed message brokers such as Kafka [5] and RabbitMQ highly available queues [6] also use it. Some network filesystems and replicated block devices such as DRBD are similar.

Synchronous Versus Asynchronous Replication

An important detail of a replicated system is whether the replication happens *synchronously* or *asynchronously*. (In relational databases, this is often a configurable option; other systems are often hardcoded to be either one or the other.)

Think about what happens in Figure 5-1, where the user of a website updates their profile image. At some point in time, the client sends the update request to the leader; shortly afterward, it is received by the leader. At some point, the leader forwards the data change to the followers. Eventually, the leader notifies the client that the update was successful.

Figure 5-2 shows the communication between various components of the system: the user's client, the leader, and two followers. Time flows from left to right. A request or response message is shown as a thick arrow.

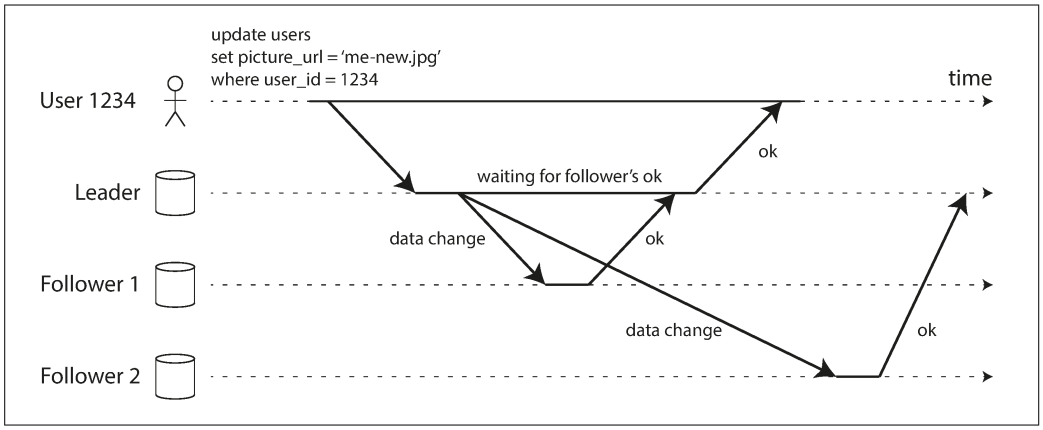


Figure 5-2. Leader-based replication with one synchronous and one asynchronous follower.

In the example of Figure 5-2, the replication to follower 1 is *synchronous*: the leader waits until follower 1 has confirmed that it received the write before reporting success to the user, and before making the write visible to other clients. The replication to follower 2 is *asynchronous*: the leader sends the message, but doesn't wait for a response from the follower.

The diagram shows that there is a substantial delay before follower 2 processes the message. Normally, replication is quite fast: most database systems apply changes to followers in less than a second. However, there is no guarantee of how long it might take. There are circumstances when followers might fall behind the leader by several minutes or more; for example, if a follower is recovering from a failure, if the system is operating near maximum capacity, or if there are network problems between the nodes.

The advantage of synchronous replication is that the follower is guaranteed to have an up-to-date copy of the data that is consistent with the leader. If the leader suddenly fails, we can be sure that the data is still available on the follower. The disadvantage is that if the synchronous follower doesn't respond (because it has crashed, or there is a network fault, or for any other reason), the write cannot be processed. The leader must block all writes and wait until the synchronous replica is available again.

For that reason, it is impractical for all followers to be synchronous: any one node outage would cause the whole system to grind to a halt. In practice, if you enable synchronous replication on a database, it usually means that *one* of the followers is synchronous, and the others are asynchronous. If the synchronous follower becomes unavailable or slow, one of the asynchronous followers is made synchronous. This guarantees that you have an up-to-date copy of the data on at least two nodes: the

leader and one synchronous follower. This configuration is sometimes also called *semi-synchronous* [7].

Often, leader-based replication is configured to be completely asynchronous. In this case, if the leader fails and is not recoverable, any writes that have not yet been replicated to followers are lost. This means that a write is not guaranteed to be durable, even if it has been confirmed to the client. However, a fully asynchronous configuration has the advantage that the leader can continue processing writes, even if all of its followers have fallen behind.

Weakening durability may sound like a bad trade-off, but asynchronous replication is nevertheless widely used, especially if there are many followers or if they are geographically distributed. We will return to this issue in "Problems with Replication Lag" on page 161.

Research on Replication

It can be a serious problem for asynchronously replicated systems to lose data if the leader fails, so researchers have continued investigating replication methods that do not lose data but still provide good performance and availability. For example, *chain replication* [8, 9] is a variant of synchronous replication that has been successfully implemented in a few systems such as Microsoft Azure Storage [10, 11].

There is a strong connection between consistency of replication and *consensus* (getting several nodes to agree on a value), and we will explore this area of theory in more detail in Chapter 9. In this chapter we will concentrate on the simpler forms of replication that are most commonly used in databases in practice.

Setting Up New Followers

From time to time, you need to set up new followers—perhaps to increase the number of replicas, or to replace failed nodes. How do you ensure that the new follower has an accurate copy of the leader's data?

Simply copying data files from one node to another is typically not sufficient: clients are constantly writing to the database, and the data is always in flux, so a standard file copy would see different parts of the database at different points in time. The result might not make any sense.

You could make the files on disk consistent by locking the database (making it unavailable for writes), but that would go against our goal of high availability. Fortunately, setting up a follower can usually be done without downtime. Conceptually, the process looks like this:

1. Take a consistent snapshot of the leader's database at some point in time—if possible, without taking a lock on the entire database. Most databases have this feature, as it is also required for backups. In some cases, third-party tools are needed, such as *innobackupex* for MySQL [12].
2. Copy the snapshot to the new follower node.
3. The follower connects to the leader and requests all the data changes that have happened since the snapshot was taken. This requires that the snapshot is associated with an exact position in the leader's replication log. That position has various names: for example, PostgreSQL calls it the *log sequence number*, and MySQL calls it the *binlog coordinates*.
4. When the follower has processed the backlog of data changes since the snapshot, we say it has *caught up*. It can now continue to process data changes from the leader as they happen.

The practical steps of setting up a follower vary significantly by database. In some systems the process is fully automated, whereas in others it can be a somewhat arcane multi-step workflow that needs to be manually performed by an administrator.

Handling Node Outages

Any node in the system can go down, perhaps unexpectedly due to a fault, but just as likely due to planned maintenance (for example, rebooting a machine to install a kernel security patch). Being able to reboot individual nodes without downtime is a big advantage for operations and maintenance. Thus, our goal is to keep the system as a whole running despite individual node failures, and to keep the impact of a node outage as small as possible.

How do you achieve high availability with leader-based replication?

Follower failure: Catch-up recovery

On its local disk, each follower keeps a log of the data changes it has received from the leader. If a follower crashes and is restarted, or if the network between the leader and the follower is temporarily interrupted, the follower can recover quite easily: from its log, it knows the last transaction that was processed before the fault occurred. Thus, the follower can connect to the leader and request all the data changes that occurred during the time when the follower was disconnected. When it has applied these changes, it has caught up to the leader and can continue receiving a stream of data changes as before.

Leader failure: Failover

Handling a failure of the leader is trickier: one of the followers needs to be promoted to be the new leader, clients need to be reconfigured to send their writes to the new leader, and the other followers need to start consuming data changes from the new leader. This process is called *failover*.

Failover can happen manually (an administrator is notified that the leader has failed and takes the necessary steps to make a new leader) or automatically. An automatic failover process usually consists of the following steps:

1. *Determining that the leader has failed.* There are many things that could potentially go wrong: crashes, power outages, network issues, and more. There is no foolproof way of detecting what has gone wrong, so most systems simply use a timeout: nodes frequently bounce messages back and forth between each other, and if a node doesn't respond for some period of time—say, 30 seconds—it is assumed to be dead. (If the leader is deliberately taken down for planned maintenance, this doesn't apply.)
2. *Choosing a new leader.* This could be done through an election process (where the leader is chosen by a majority of the remaining replicas), or a new leader could be appointed by a previously elected *controller node*. The best candidate for leadership is usually the replica with the most up-to-date data changes from the old leader (to minimize any data loss). Getting all the nodes to agree on a new leader is a consensus problem, discussed in detail in Chapter 9.
3. *Reconfiguring the system to use the new leader.* Clients now need to send their write requests to the new leader (we discuss this in "Request Routing" on page 214). If the old leader comes back, it might still believe that it is the leader, not realizing that the other replicas have forced it to step down. The system needs to ensure that the old leader becomes a follower and recognizes the new leader.

Failover is fraught with things that can go wrong:

- If asynchronous replication is used, the new leader may not have received all the writes from the old leader before it failed. If the former leader rejoins the cluster after a new leader has been chosen, what should happen to those writes? The new leader may have received conflicting writes in the meantime. The most common solution is for the old leader's unreplicated writes to simply be discarded, which may violate clients' durability expectations.
- Discarding writes is especially dangerous if other storage systems outside of the database need to be coordinated with the database contents. For example, in one incident at GitHub [13], an out-of-date MySQL follower was promoted to leader. The database used an autoincrementing counter to assign primary keys to new

rows, but because the new leader's counter lagged behind the old leader's, it reused some primary keys that were previously assigned by the old leader. These primary keys were also used in a Redis store, so the reuse of primary keys resulted in inconsistency between MySQL and Redis, which caused some private data to be disclosed to the wrong users.

- In certain fault scenarios (see Chapter 8), it could happen that two nodes both believe that they are the leader. This situation is called *split brain*, and it is dangerous: if both leaders accept writes, and there is no process for resolving conflicts (see "Multi-Leader Replication" on page 168), data is likely to be lost or corrupted. As a safety catch, some systems have a mechanism to shut down one node if two leaders are detected.[ii] However, if this mechanism is not carefully designed, you can end up with both nodes being shut down [14].
- What is the right timeout before the leader is declared dead? A longer timeout means a longer time to recovery in the case where the leader fails. However, if the timeout is too short, there could be unnecessary failovers. For example, a temporary load spike could cause a node's response time to increase above the timeout, or a network glitch could cause delayed packets. If the system is already struggling with high load or network problems, an unnecessary failover is likely to make the situation worse, not better.

There are no easy solutions to these problems. For this reason, some operations teams prefer to perform failovers manually, even if the software supports automatic failover.

These issues—node failures; unreliable networks; and trade-offs around replica consistency, durability, availability, and latency—are in fact fundamental problems in distributed systems. In Chapter 8 and Chapter 9 we will discuss them in greater depth.

Implementation of Replication Logs

How does leader-based replication work under the hood? Several different replication methods are used in practice, so let's look at each one briefly.

Statement-based replication

In the simplest case, the leader logs every write request (*statement*) that it executes and sends that statement log to its followers. For a relational database, this means that every `INSERT`, `UPDATE`, or `DELETE` statement is forwarded to followers, and each

ii. This approach is known as *fencing* or, more emphatically, *Shoot The Other Node In The Head* (STONITH). We will discuss fencing in more detail in "The leader and the lock" on page 301.

follower parses and executes that SQL statement as if it had been received from a client.

Although this may sound reasonable, there are various ways in which this approach to replication can break down:

- Any statement that calls a nondeterministic function, such as `NOW()` to get the current date and time or `RAND()` to get a random number, is likely to generate a different value on each replica.
- If statements use an autoincrementing column, or if they depend on the existing data in the database (e.g., `UPDATE … WHERE <some condition>`), they must be executed in exactly the same order on each replica, or else they may have a different effect. This can be limiting when there are multiple concurrently executing transactions.
- Statements that have side effects (e.g., triggers, stored procedures, user-defined functions) may result in different side effects occurring on each replica, unless the side effects are absolutely deterministic.

It is possible to work around those issues—for example, the leader can replace any nondeterministic function calls with a fixed return value when the statement is logged so that the followers all get the same value. However, because there are so many edge cases, other replication methods are now generally preferred.

Statement-based replication was used in MySQL before version 5.1. It is still sometimes used today, as it is quite compact, but by default MySQL now switches to row-based replication (discussed shortly) if there is any nondeterminism in a statement. VoltDB uses statement-based replication, and makes it safe by requiring transactions to be deterministic [15].

Write-ahead log (WAL) shipping

In Chapter 3 we discussed how storage engines represent data on disk, and we found that usually every write is appended to a log:

- In the case of a log-structured storage engine (see "SSTables and LSM-Trees" on page 76), this log is the main place for storage. Log segments are compacted and garbage-collected in the background.
- In the case of a B-tree (see "B-Trees" on page 79), which overwrites individual disk blocks, every modification is first written to a write-ahead log so that the index can be restored to a consistent state after a crash.

In either case, the log is an append-only sequence of bytes containing all writes to the database. We can use the exact same log to build a replica on another node: besides writing the log to disk, the leader also sends it across the network to its followers.

When the follower processes this log, it builds a copy of the exact same data structures as found on the leader.

This method of replication is used in PostgreSQL and Oracle, among others [16]. The main disadvantage is that the log describes the data on a very low level: a WAL contains details of which bytes were changed in which disk blocks. This makes replication closely coupled to the storage engine. If the database changes its storage format from one version to another, it is typically not possible to run different versions of the database software on the leader and the followers.

That may seem like a minor implementation detail, but it can have a big operational impact. If the replication protocol allows the follower to use a newer software version than the leader, you can perform a zero-downtime upgrade of the database software by first upgrading the followers and then performing a failover to make one of the upgraded nodes the new leader. If the replication protocol does not allow this version mismatch, as is often the case with WAL shipping, such upgrades require downtime.

Logical (row-based) log replication

An alternative is to use different log formats for replication and for the storage engine, which allows the replication log to be decoupled from the storage engine internals. This kind of replication log is called a *logical log*, to distinguish it from the storage engine's (*physical*) data representation.

A logical log for a relational database is usually a sequence of records describing writes to database tables at the granularity of a row:

- For an inserted row, the log contains the new values of all columns.
- For a deleted row, the log contains enough information to uniquely identify the row that was deleted. Typically this would be the primary key, but if there is no primary key on the table, the old values of all columns need to be logged.
- For an updated row, the log contains enough information to uniquely identify the updated row, and the new values of all columns (or at least the new values of all columns that changed).

A transaction that modifies several rows generates several such log records, followed by a record indicating that the transaction was committed. MySQL's binlog (when configured to use row-based replication) uses this approach [17].

Since a logical log is decoupled from the storage engine internals, it can more easily be kept backward compatible, allowing the leader and the follower to run different versions of the database software, or even different storage engines.

A logical log format is also easier for external applications to parse. This aspect is useful if you want to send the contents of a database to an external system, such as a data

warehouse for offline analysis, or for building custom indexes and caches [18]. This technique is called *change data capture*, and we will return to it in Chapter 11.

Trigger-based replication

The replication approaches described so far are implemented by the database system, without involving any application code. In many cases, that's what you want—but there are some circumstances where more flexibility is needed. For example, if you want to only replicate a subset of the data, or want to replicate from one kind of database to another, or if you need conflict resolution logic (see "Handling Write Conflicts" on page 171), then you may need to move replication up to the application layer.

Some tools, such as Oracle GoldenGate [19], can make data changes available to an application by reading the database log. An alternative is to use features that are available in many relational databases: *triggers* and *stored procedures*.

A trigger lets you register custom application code that is automatically executed when a data change (write transaction) occurs in a database system. The trigger has the opportunity to log this change into a separate table, from which it can be read by an external process. That external process can then apply any necessary application logic and replicate the data change to another system. Databus for Oracle [20] and Bucardo for Postgres [21] work like this, for example.

Trigger-based replication typically has greater overheads than other replication methods, and is more prone to bugs and limitations than the database's built-in replication. However, it can nevertheless be useful due to its flexibility.

Problems with Replication Lag

Being able to tolerate node failures is just one reason for wanting replication. As mentioned in the introduction to Part II, other reasons are scalability (processing more requests than a single machine can handle) and latency (placing replicas geographically closer to users).

Leader-based replication requires all writes to go through a single node, but read-only queries can go to any replica. For workloads that consist of mostly reads and only a small percentage of writes (a common pattern on the web), there is an attractive option: create many followers, and distribute the read requests across those followers. This removes load from the leader and allows read requests to be served by nearby replicas.

In this *read-scaling* architecture, you can increase the capacity for serving read-only requests simply by adding more followers. However, this approach only realistically works with asynchronous replication—if you tried to synchronously replicate to all followers, a single node failure or network outage would make the entire system

unavailable for writing. And the more nodes you have, the likelier it is that one will be down, so a fully synchronous configuration would be very unreliable.

Unfortunately, if an application reads from an *asynchronous* follower, it may see outdated information if the follower has fallen behind. This leads to apparent inconsistencies in the database: if you run the same query on the leader and a follower at the same time, you may get different results, because not all writes have been reflected in the follower. This inconsistency is just a temporary state—if you stop writing to the database and wait a while, the followers will eventually catch up and become consistent with the leader. For that reason, this effect is known as *eventual consistency* [22, 23].[iii]

The term "eventually" is deliberately vague: in general, there is no limit to how far a replica can fall behind. In normal operation, the delay between a write happening on the leader and being reflected on a follower—the *replication lag*—may be only a fraction of a second, and not noticeable in practice. However, if the system is operating near capacity or if there is a problem in the network, the lag can easily increase to several seconds or even minutes.

When the lag is so large, the inconsistencies it introduces are not just a theoretical issue but a real problem for applications. In this section we will highlight three examples of problems that are likely to occur when there is replication lag and outline some approaches to solving them.

Reading Your Own Writes

Many applications let the user submit some data and then view what they have submitted. This might be a record in a customer database, or a comment on a discussion thread, or something else of that sort. When new data is submitted, it must be sent to the leader, but when the user views the data, it can be read from a follower. This is especially appropriate if data is frequently viewed but only occasionally written.

With asynchronous replication, there is a problem, illustrated in Figure 5-3: if the user views the data shortly after making a write, the new data may not yet have reached the replica. To the user, it looks as though the data they submitted was lost, so they will be understandably unhappy.

iii. The term *eventual consistency* was coined by Douglas Terry et al. [24], popularized by Werner Vogels [22], and became the battle cry of many NoSQL projects. However, not only NoSQL databases are eventually consistent: followers in an asynchronously replicated relational database have the same characteristics.

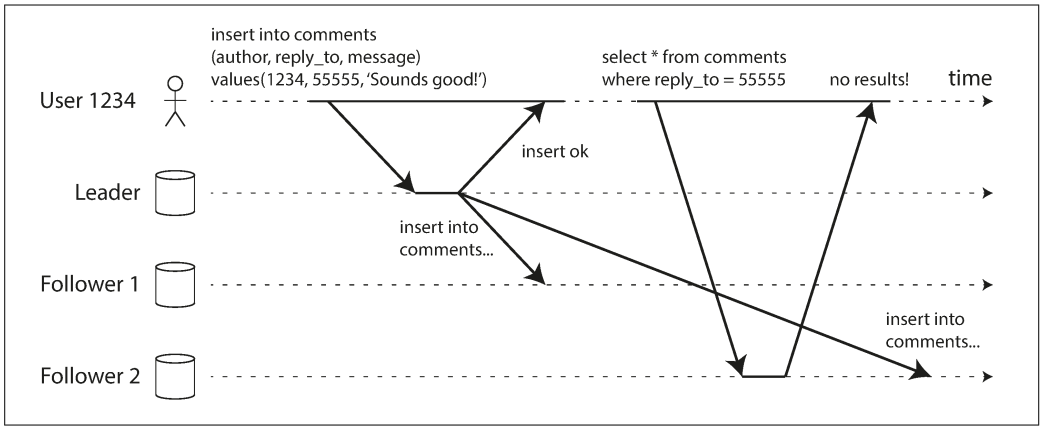


Figure 5-3. A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need read-after-write consistency.

In this situation, we need *read-after-write consistency*, also known as *read-your-writes consistency* [24]. This is a guarantee that if the user reloads the page, they will always see any updates they submitted themselves. It makes no promises about other users: other users' updates may not be visible until some later time. However, it reassures the user that their own input has been saved correctly.

How can we implement read-after-write consistency in a system with leader-based replication? There are various possible techniques. To mention a few:

- When reading something that the user may have modified, read it from the leader; otherwise, read it from a follower. This requires that you have some way of knowing whether something might have been modified, without actually querying it. For example, user profile information on a social network is normally only editable by the owner of the profile, not by anybody else. Thus, a simple rule is: always read the user's own profile from the leader, and any other users' profiles from a follower.
- If most things in the application are potentially editable by the user, that approach won't be effective, as most things would have to be read from the leader (negating the benefit of read scaling). In that case, other criteria may be used to decide whether to read from the leader. For example, you could track the time of the last update and, for one minute after the last update, make all reads from the leader. You could also monitor the replication lag on followers and prevent queries on any follower that is more than one minute behind the leader.
- The client can remember the timestamp of its most recent write—then the system can ensure that the replica serving any reads for that user reflects updates at least until that timestamp. If a replica is not sufficiently up to date, either the read can be handled by another replica or the query can wait until the replica has

caught up. The timestamp could be a *logical timestamp* (something that indicates ordering of writes, such as the log sequence number) or the actual system clock (in which case clock synchronization becomes critical; see "Unreliable Clocks" on page 287).

- If your replicas are distributed across multiple datacenters (for geographical proximity to users or for availability), there is additional complexity. Any request that needs to be served by the leader must be routed to the datacenter that contains the leader.

Another complication arises when the same user is accessing your service from multiple devices, for example a desktop web browser and a mobile app. In this case you may want to provide *cross-device* read-after-write consistency: if the user enters some information on one device and then views it on another device, they should see the information they just entered.

In this case, there are some additional issues to consider:

- Approaches that require remembering the timestamp of the user's last update become more difficult, because the code running on one device doesn't know what updates have happened on the other device. This metadata will need to be centralized.
- If your replicas are distributed across different datacenters, there is no guarantee that connections from different devices will be routed to the same datacenter. (For example, if the user's desktop computer uses the home broadband connection and their mobile device uses the cellular data network, the devices' network routes may be completely different.) If your approach requires reading from the leader, you may first need to route requests from all of a user's devices to the same datacenter.

Monotonic Reads

Our second example of an anomaly that can occur when reading from asynchronous followers is that it's possible for a user to see things *moving backward in time*.

This can happen if a user makes several reads from different replicas. For example, Figure 5-4 shows user 2345 making the same query twice, first to a follower with little lag, then to a follower with greater lag. (This scenario is quite likely if the user refreshes a web page, and each request is routed to a random server.) The first query returns a comment that was recently added by user 1234, but the second query doesn't return anything because the lagging follower has not yet picked up that write. In effect, the second query is observing the system at an earlier point in time than the first query. This wouldn't be so bad if the first query hadn't returned anything, because user 2345 probably wouldn't know that user 1234 had recently added a com-

ment. However, it's very confusing for user 2345 if they first see user 1234's comment appear, and then see it disappear again.

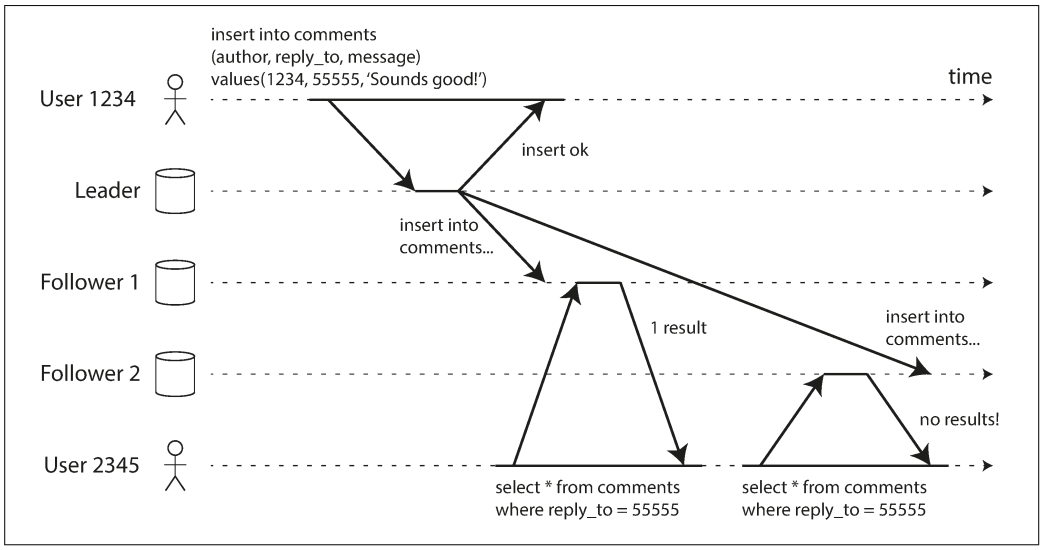


Figure 5-4. A user first reads from a fresh replica, then from a stale replica. Time appears to go backward. To prevent this anomaly, we need monotonic reads.

Monotonic reads [23] is a guarantee that this kind of anomaly does not happen. It's a lesser guarantee than strong consistency, but a stronger guarantee than eventual consistency. When you read data, you may see an old value; monotonic reads only means that if one user makes several reads in sequence, they will not see time go backward—i.e., they will not read older data after having previously read newer data.

One way of achieving monotonic reads is to make sure that each user always makes their reads from the same replica (different users can read from different replicas). For example, the replica can be chosen based on a hash of the user ID, rather than randomly. However, if that replica fails, the user's queries will need to be rerouted to another replica.

Consistent Prefix Reads

Our third example of replication lag anomalies concerns violation of causality. Imagine the following short dialog between Mr. Poons and Mrs. Cake:

Mr. Poons
: How far into the future can you see, Mrs. Cake?

Mrs. Cake
: About ten seconds usually, Mr. Poons.

There is a causal dependency between those two sentences: Mrs. Cake heard Mr. Poons's question and answered it.

Now, imagine a third person is listening to this conversation through followers. The things said by Mrs. Cake go through a follower with little lag, but the things said by Mr. Poons have a longer replication lag (see Figure 5-5). This observer would hear the following:

Mrs. Cake
: About ten seconds usually, Mr. Poons.

Mr. Poons
: How far into the future can you see, Mrs. Cake?

To the observer it looks as though Mrs. Cake is answering the question before Mr. Poons has even asked it. Such psychic powers are impressive, but very confusing [25].

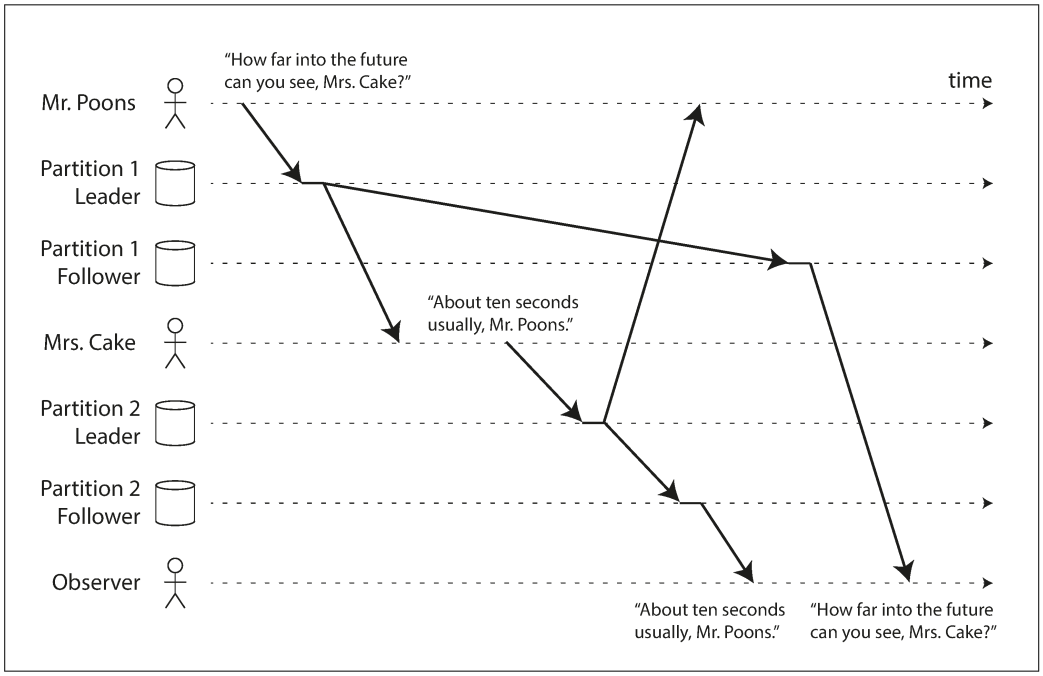


Figure 5-5. If some partitions are replicated slower than others, an observer may see the answer before they see the question.

Preventing this kind of anomaly requires another type of guarantee: *consistent prefix reads* [23]. This guarantee says that if a sequence of writes happens in a certain order, then anyone reading those writes will see them appear in the same order.

This is a particular problem in partitioned (sharded) databases, which we will discuss in Chapter 6. If the database always applies writes in the same order, reads always see a consistent prefix, so this anomaly cannot happen. However, in many distributed

databases, different partitions operate independently, so there is no global ordering of writes: when a user reads from the database, they may see some parts of the database in an older state and some in a newer state.

One solution is to make sure that any writes that are causally related to each other are written to the same partition—but in some applications that cannot be done efficiently. There are also algorithms that explicitly keep track of causal dependencies, a topic that we will return to in "The "happens-before" relationship and concurrency" on page 186.

Solutions for Replication Lag

When working with an eventually consistent system, it is worth thinking about how the application behaves if the replication lag increases to several minutes or even hours. If the answer is "no problem," that's great. However, if the result is a bad experience for users, it's important to design the system to provide a stronger guarantee, such as read-after-write. Pretending that replication is synchronous when in fact it is asynchronous is a recipe for problems down the line.

As discussed earlier, there are ways in which an application can provide a stronger guarantee than the underlying database—for example, by performing certain kinds of reads on the leader. However, dealing with these issues in application code is complex and easy to get wrong.

It would be better if application developers didn't have to worry about subtle replication issues and could just trust their databases to "do the right thing." This is why *transactions* exist: they are a way for a database to provide stronger guarantees so that the application can be simpler.

Single-node transactions have existed for a long time. However, in the move to distributed (replicated and partitioned) databases, many systems have abandoned them, claiming that transactions are too expensive in terms of performance and availability, and asserting that eventual consistency is inevitable in a scalable system. There is some truth in that statement, but it is overly simplistic, and we will develop a more nuanced view over the course of the rest of this book. We will return to the topic of transactions in Chapters 7 and 9, and we will discuss some alternative mechanisms in Part III.

Multi-Leader Replication

So far in this chapter we have only considered replication architectures using a single leader. Although that is a common approach, there are interesting alternatives.

Leader-based replication has one major downside: there is only one leader, and all writes must go through it.[iv] If you can't connect to the leader for any reason, for example due to a network interruption between you and the leader, you can't write to the database.

A natural extension of the leader-based replication model is to allow more than one node to accept writes. Replication still happens in the same way: each node that processes a write must forward that data change to all the other nodes. We call this a *multi-leader* configuration (also known as *master–master* or *active/active replication*). In this setup, each leader simultaneously acts as a follower to the other leaders.

Use Cases for Multi-Leader Replication

It rarely makes sense to use a multi-leader setup within a single datacenter, because the benefits rarely outweigh the added complexity. However, there are some situations in which this configuration is reasonable.

Multi-datacenter operation

Imagine you have a database with replicas in several different datacenters (perhaps so that you can tolerate failure of an entire datacenter, or perhaps in order to be closer to your users). With a normal leader-based replication setup, the leader has to be in *one* of the datacenters, and all writes must go through that datacenter.

In a multi-leader configuration, you can have a leader in *each* datacenter. Figure 5-6 shows what this architecture might look like. Within each datacenter, regular leader–follower replication is used; between datacenters, each datacenter's leader replicates its changes to the leaders in other datacenters.

iv. If the database is partitioned (see Chapter 6), each partition has one leader. Different partitions may have their leaders on different nodes, but each partition must nevertheless have one leader node.

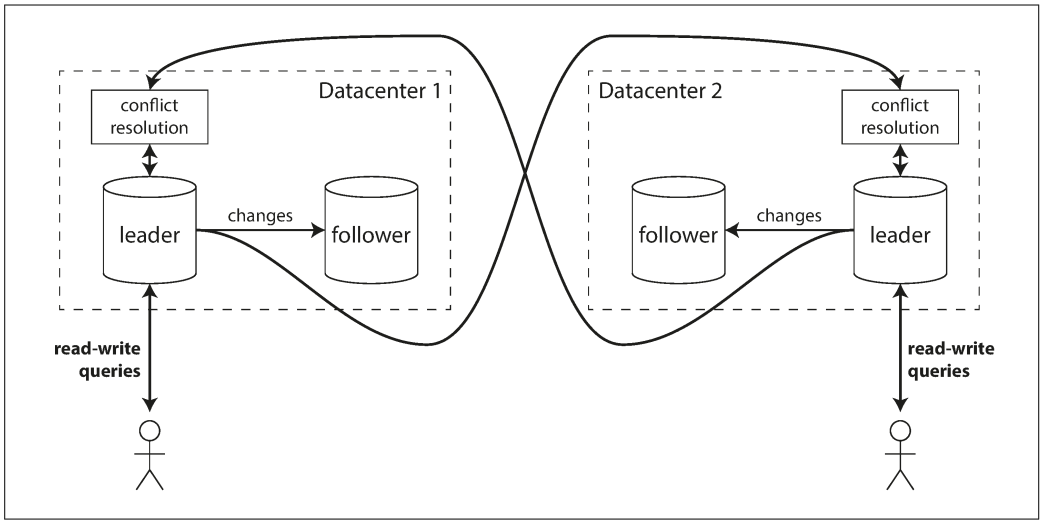


Figure 5-6. Multi-leader replication across multiple datacenters.

Let's compare how the single-leader and multi-leader configurations fare in a multi-datacenter deployment:

Performance
: In a single-leader configuration, every write must go over the internet to the datacenter with the leader. This can add significant latency to writes and might contravene the purpose of having multiple datacenters in the first place. In a multi-leader configuration, every write can be processed in the local datacenter and is replicated asynchronously to the other datacenters. Thus, the inter-datacenter network delay is hidden from users, which means the perceived performance may be better.

Tolerance of datacenter outages
: In a single-leader configuration, if the datacenter with the leader fails, failover can promote a follower in another datacenter to be leader. In a multi-leader configuration, each datacenter can continue operating independently of the others, and replication catches up when the failed datacenter comes back online.

Tolerance of network problems
: Traffic between datacenters usually goes over the public internet, which may be less reliable than the local network within a datacenter. A single-leader configuration is very sensitive to problems in this inter-datacenter link, because writes are made synchronously over this link. A multi-leader configuration with asynchronous replication can usually tolerate network problems better: a temporary network interruption does not prevent writes being processed.

Some databases support multi-leader configurations by default, but it is also often implemented with external tools, such as Tungsten Replicator for MySQL [26], BDR for PostgreSQL [27], and GoldenGate for Oracle [19].

Although multi-leader replication has advantages, it also has a big downside: the same data may be concurrently modified in two different datacenters, and those write conflicts must be resolved (indicated as "conflict resolution" in Figure 5-6). We will discuss this issue in "Handling Write Conflicts" on page 171.

As multi-leader replication is a somewhat retrofitted feature in many databases, there are often subtle configuration pitfalls and surprising interactions with other database features. For example, autoincrementing keys, triggers, and integrity constraints can be problematic. For this reason, multi-leader replication is often considered dangerous territory that should be avoided if possible [28].

Clients with offline operation

Another situation in which multi-leader replication is appropriate is if you have an application that needs to continue to work while it is disconnected from the internet.

For example, consider the calendar apps on your mobile phone, your laptop, and other devices. You need to be able to see your meetings (make read requests) and enter new meetings (make write requests) at any time, regardless of whether your device currently has an internet connection. If you make any changes while you are offline, they need to be synced with a server and your other devices when the device is next online.

In this case, every device has a local database that acts as a leader (it accepts write requests), and there is an asynchronous multi-leader replication process (sync) between the replicas of your calendar on all of your devices. The replication lag may be hours or even days, depending on when you have internet access available.

From an architectural point of view, this setup is essentially the same as multi-leader replication between datacenters, taken to the extreme: each device is a "datacenter," and the network connection between them is extremely unreliable. As the rich history of broken calendar sync implementations demonstrates, multi-leader replication is a tricky thing to get right.

There are tools that aim to make this kind of multi-leader configuration easier. For example, CouchDB is designed for this mode of operation [29].

Collaborative editing

Real-time collaborative editing applications allow several people to edit a document simultaneously. For example, Etherpad [30] and Google Docs [31] allow multiple people to concurrently edit a text document or spreadsheet (the algorithm is briefly discussed in "Automatic Conflict Resolution" on page 174).

We don't usually think of collaborative editing as a database replication problem, but it has a lot in common with the previously mentioned offline editing use case. When one user edits a document, the changes are instantly applied to their local replica (the state of the document in their web browser or client application) and asynchronously replicated to the server and any other users who are editing the same document.

If you want to guarantee that there will be no editing conflicts, the application must obtain a lock on the document before a user can edit it. If another user wants to edit the same document, they first have to wait until the first user has committed their changes and released the lock. This collaboration model is equivalent to single-leader replication with transactions on the leader.

However, for faster collaboration, you may want to make the unit of change very small (e.g., a single keystroke) and avoid locking. This approach allows multiple users to edit simultaneously, but it also brings all the challenges of multi-leader replication, including requiring conflict resolution [32].

Handling Write Conflicts

The biggest problem with multi-leader replication is that write conflicts can occur, which means that conflict resolution is required.

For example, consider a wiki page that is simultaneously being edited by two users, as shown in Figure 5-7. User 1 changes the title of the page from A to B, and user 2 changes the title from A to C at the same time. Each user's change is successfully applied to their local leader. However, when the changes are asynchronously replicated, a conflict is detected [33]. This problem does not occur in a single-leader database.

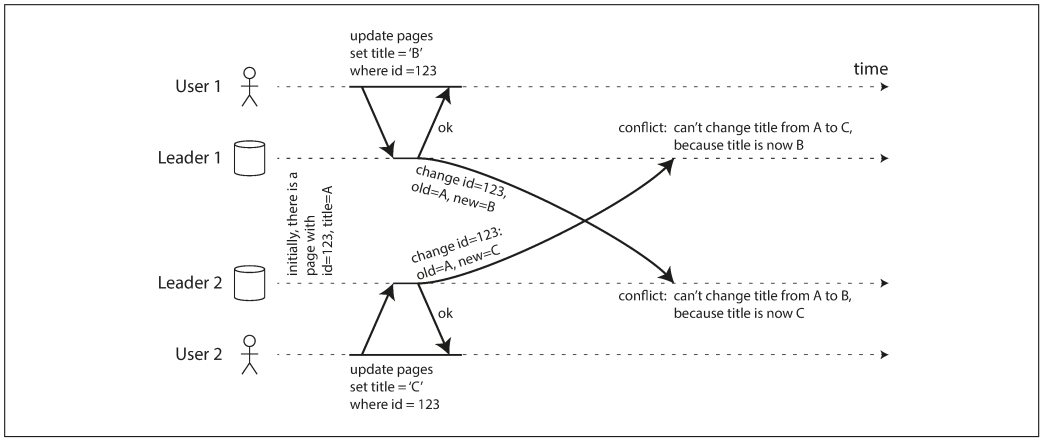


Figure 5-7. A write conflict caused by two leaders concurrently updating the same record.

Synchronous versus asynchronous conflict detection

In a single-leader database, the second writer will either block and wait for the first write to complete, or abort the second write transaction, forcing the user to retry the write. On the other hand, in a multi-leader setup, both writes are successful, and the conflict is only detected asynchronously at some later point in time. At that time, it may be too late to ask the user to resolve the conflict.

In principle, you could make the conflict detection synchronous—i.e., wait for the write to be replicated to all replicas before telling the user that the write was successful. However, by doing so, you would lose the main advantage of multi-leader replication: allowing each replica to accept writes independently. If you want synchronous conflict detection, you might as well just use single-leader replication.

Conflict avoidance

The simplest strategy for dealing with conflicts is to avoid them: if the application can ensure that all writes for a particular record go through the same leader, then conflicts cannot occur. Since many implementations of multi-leader replication handle conflicts quite poorly, avoiding conflicts is a frequently recommended approach [34].

For example, in an application where a user can edit their own data, you can ensure that requests from a particular user are always routed to the same datacenter and use the leader in that datacenter for reading and writing. Different users may have different "home" datacenters (perhaps picked based on geographic proximity to the user), but from any one user's point of view the configuration is essentially single-leader.

However, sometimes you might want to change the designated leader for a record—perhaps because one datacenter has failed and you need to reroute traffic to another datacenter, or perhaps because a user has moved to a different location and is now closer to a different datacenter. In this situation, conflict avoidance breaks down, and you have to deal with the possibility of concurrent writes on different leaders.

Converging toward a consistent state

A single-leader database applies writes in a sequential order: if there are several updates to the same field, the last write determines the final value of the field.

In a multi-leader configuration, there is no defined ordering of writes, so it's not clear what the final value should be. In Figure 5-7, at leader 1 the title is first updated to B and then to C; at leader 2 it is first updated to C and then to B. Neither order is "more correct" than the other.

If each replica simply applied writes in the order that it saw the writes, the database would end up in an inconsistent state: the final value would be C at leader 1 and B at leader 2. That is not acceptable—every replication scheme must ensure that the data is eventually the same in all replicas. Thus, the database must resolve the conflict in a

convergent way, which means that all replicas must arrive at the same final value when all changes have been replicated.

There are various ways of achieving convergent conflict resolution:

- Give each write a unique ID (e.g., a timestamp, a long random number, a UUID, or a hash of the key and value), pick the write with the highest ID as the *winner*, and throw away the other writes. If a timestamp is used, this technique is known as *last write wins* (LWW). Although this approach is popular, it is dangerously prone to data loss [35]. We will discuss LWW in more detail at the end of this chapter ("Detecting Concurrent Writes" on page 184).
- Give each replica a unique ID, and let writes that originated at a higher-numbered replica always take precedence over writes that originated at a lower-numbered replica. This approach also implies data loss.
- Somehow merge the values together—e.g., order them alphabetically and then concatenate them (in Figure 5-7, the merged title might be something like "B/C").
- Record the conflict in an explicit data structure that preserves all information, and write application code that resolves the conflict at some later time (perhaps by prompting the user).

Custom conflict resolution logic

As the most appropriate way of resolving a conflict may depend on the application, most multi-leader replication tools let you write conflict resolution logic using application code. That code may be executed on write or on read:

On write
: As soon as the database system detects a conflict in the log of replicated changes, it calls the conflict handler. For example, Bucardo allows you to write a snippet of Perl for this purpose. This handler typically cannot prompt a user—it runs in a background process and it must execute quickly.

On read
: When a conflict is detected, all the conflicting writes are stored. The next time the data is read, these multiple versions of the data are returned to the application. The application may prompt the user or automatically resolve the conflict, and write the result back to the database. CouchDB works this way, for example.

Note that conflict resolution usually applies at the level of an individual row or document, not for an entire transaction [36]. Thus, if you have a transaction that atomically makes several different writes (see Chapter 7), each write is still considered separately for the purposes of conflict resolution.

Automatic Conflict Resolution

Conflict resolution rules can quickly become complicated, and custom code can be error-prone. Amazon is a frequently cited example of surprising effects due to a conflict resolution handler: for some time, the conflict resolution logic on the shopping cart would preserve items added to the cart, but not items removed from the cart. Thus, customers would sometimes see items reappearing in their carts even though they had previously been removed [37].

There has been some interesting research into automatically resolving conflicts caused by concurrent data modifications. A few lines of research are worth mentioning:

- *Conflict-free replicated datatypes* (CRDTs) [32, 38] are a family of data structures for sets, maps, ordered lists, counters, etc. that can be concurrently edited by multiple users, and which automatically resolve conflicts in sensible ways. Some CRDTs have been implemented in Riak 2.0 [39, 40].
- *Mergeable persistent data structures* [41] track history explicitly, similarly to the Git version control system, and use a three-way merge function (whereas CRDTs use two-way merges).
- *Operational transformation* [42] is the conflict resolution algorithm behind collaborative editing applications such as Etherpad [30] and Google Docs [31]. It was designed particularly for concurrent editing of an ordered list of items, such as the list of characters that constitute a text document.

Implementations of these algorithms in databases are still young, but it's likely that they will be integrated into more replicated data systems in the future. Automatic conflict resolution could make multi-leader data synchronization much simpler for applications to deal with.

What is a conflict?

Some kinds of conflict are obvious. In the example in Figure 5-7, two writes concurrently modified the same field in the same record, setting it to two different values. There is little doubt that this is a conflict.

Other kinds of conflict can be more subtle to detect. For example, consider a meeting room booking system: it tracks which room is booked by which group of people at which time. This application needs to ensure that each room is only booked by one group of people at any one time (i.e., there must not be any overlapping bookings for the same room). In this case, a conflict may arise if two different bookings are created for the same room at the same time. Even if the application checks availability before

allowing a user to make a booking, there can be a conflict if the two bookings are made on two different leaders.

There isn't a quick ready-made answer, but in the following chapters we will trace a path toward a good understanding of this problem. We will see some more examples of conflicts in Chapter 7, and in Chapter 12 we will discuss scalable approaches for detecting and resolving conflicts in a replicated system.

Multi-Leader Replication Topologies

A *replication topology* describes the communication paths along which writes are propagated from one node to another. If you have two leaders, like in Figure 5-7, there is only one plausible topology: leader 1 must send all of its writes to leader 2, and vice versa. With more than two leaders, various different topologies are possible. Some examples are illustrated in Figure 5-8.

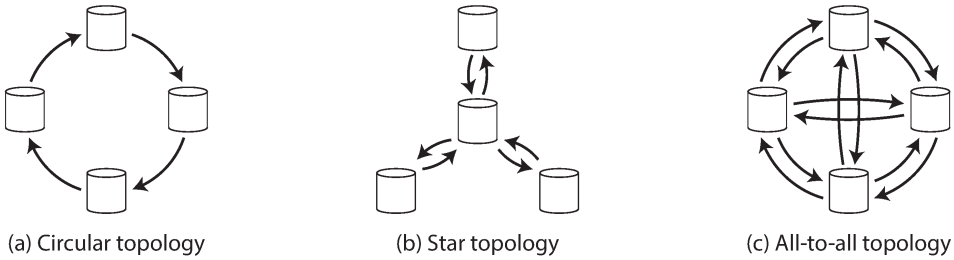


Figure 5-8. Three example topologies in which multi-leader replication can be set up.

The most general topology is *all-to-all* (Figure 5-8 [c]), in which every leader sends its writes to every other leader. However, more restricted topologies are also used: for example, MySQL by default supports only a *circular topology* [34], in which each node receives writes from one node and forwards those writes (plus any writes of its own) to one other node. Another popular topology has the shape of a *star*:[v] one designated root node forwards writes to all of the other nodes. The star topology can be generalized to a tree.

In circular and star topologies, a write may need to pass through several nodes before it reaches all replicas. Therefore, nodes need to forward data changes they receive from other nodes. To prevent infinite replication loops, each node is given a unique identifier, and in the replication log, each write is tagged with the identifiers of all the nodes it has passed through [43]. When a node receives a data change that is tagged

v. Not to be confused with a *star schema* (see "Stars and Snowflakes: Schemas for Analytics" on page 93), which describes the structure of a data model, not the communication topology between nodes.

with its own identifier, that data change is ignored, because the node knows that it has already been processed.

A problem with circular and star topologies is that if just one node fails, it can interrupt the flow of replication messages between other nodes, causing them to be unable to communicate until the node is fixed. The topology could be reconfigured to work around the failed node, but in most deployments such reconfiguration would have to be done manually. The fault tolerance of a more densely connected topology (such as all-to-all) is better because it allows messages to travel along different paths, avoiding a single point of failure.

On the other hand, all-to-all topologies can have issues too. In particular, some network links may be faster than others (e.g., due to network congestion), with the result that some replication messages may "overtake" others, as illustrated in Figure 5-9.

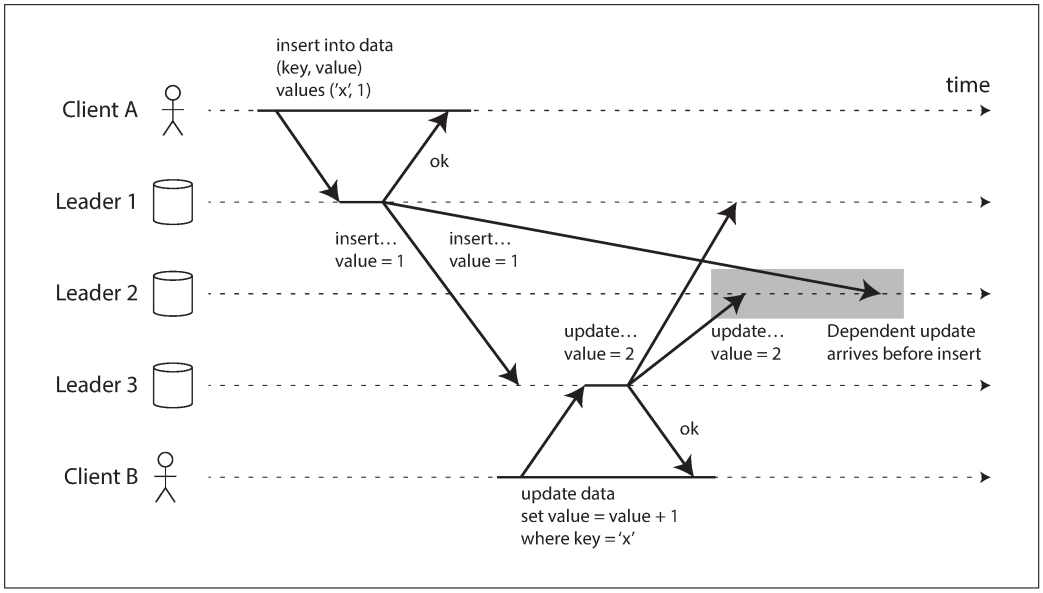


Figure 5-9. With multi-leader replication, writes may arrive in the wrong order at some replicas.

In Figure 5-9, client A inserts a row into a table on leader 1, and client B updates that row on leader 3. However, leader 2 may receive the writes in a different order: it may first receive the update (which, from its point of view, is an update to a row that does not exist in the database) and only later receive the corresponding insert (which should have preceded the update).

This is a problem of causality, similar to the one we saw in "Consistent Prefix Reads" on page 165: the update depends on the prior insert, so we need to make sure that all nodes process the insert first, and then the update. Simply attaching a timestamp to

every write is not sufficient, because clocks cannot be trusted to be sufficiently in sync to correctly order these events at leader 2 (see Chapter 8).

To order these events correctly, a technique called *version vectors* can be used, which we will discuss later in this chapter (see "Detecting Concurrent Writes" on page 184). However, conflict detection techniques are poorly implemented in many multi-leader replication systems. For example, at the time of writing, PostgreSQL BDR does not provide causal ordering of writes [27], and Tungsten Replicator for MySQL doesn't even try to detect conflicts [34].

If you are using a system with multi-leader replication, it is worth being aware of these issues, carefully reading the documentation, and thoroughly testing your database to ensure that it really does provide the guarantees you believe it to have.

Leaderless Replication

The replication approaches we have discussed so far in this chapter—single-leader and multi-leader replication—are based on the idea that a client sends a write request to one node (the leader), and the database system takes care of copying that write to the other replicas. A leader determines the order in which writes should be processed, and followers apply the leader's writes in the same order.

Some data storage systems take a different approach, abandoning the concept of a leader and allowing any replica to directly accept writes from clients. Some of the earliest replicated data systems were leaderless [1, 44], but the idea was mostly forgotten during the era of dominance of relational databases. It once again became a fashionable architecture for databases after Amazon used it for its in-house *Dynamo* system [37].[vi] Riak, Cassandra, and Voldemort are open source datastores with leaderless replication models inspired by Dynamo, so this kind of database is also known as *Dynamo-style*.

In some leaderless implementations, the client directly sends its writes to several replicas, while in others, a coordinator node does this on behalf of the client. However, unlike a leader database, that coordinator does not enforce a particular ordering of writes. As we shall see, this difference in design has profound consequences for the way the database is used.

Writing to the Database When a Node Is Down

Imagine you have a database with three replicas, and one of the replicas is currently unavailable—perhaps it is being rebooted to install a system update. In a leader-based

vi. Dynamo is not available to users outside of Amazon. Confusingly, AWS offers a hosted database product called *DynamoDB*, which uses a completely different architecture: it is based on single-leader replication.

configuration, if you want to continue processing writes, you may need to perform a failover (see "Handling Node Outages" on page 156).

On the other hand, in a leaderless configuration, failover does not exist. Figure 5-10 shows what happens: the client (user 1234) sends the write to all three replicas in parallel, and the two available replicas accept the write but the unavailable replica misses it. Let's say that it's sufficient for two out of three replicas to acknowledge the write: after user 1234 has received two *ok* responses, we consider the write to be successful. The client simply ignores the fact that one of the replicas missed the write.

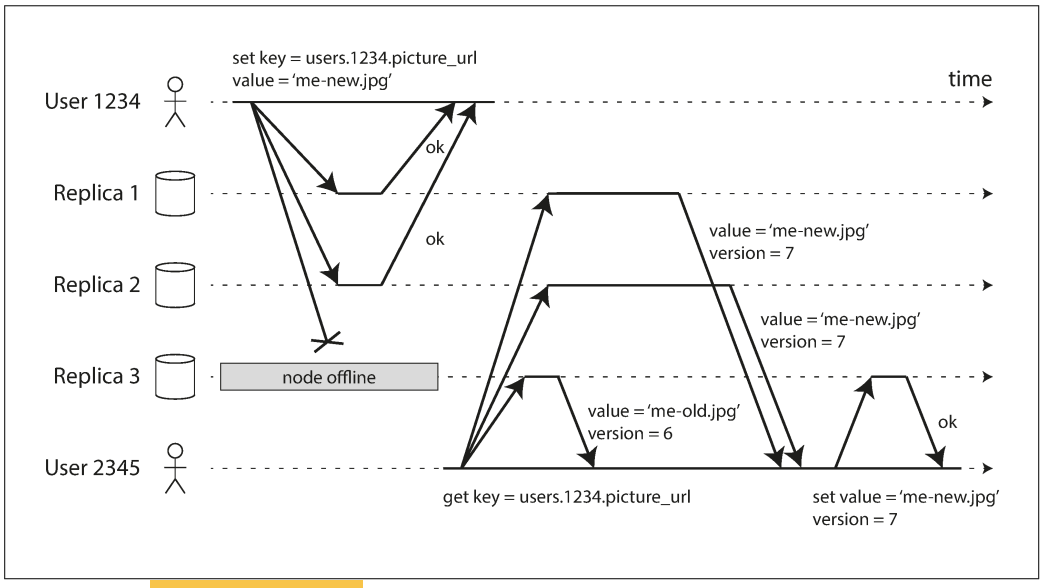


Figure 5-10. A quorum write, quorum read, and read repair after a node outage.

Now imagine that the unavailable node comes back online, and clients start reading from it. Any writes that happened while the node was down are missing from that node. Thus, if you read from that node, you may get *stale* (outdated) values as responses.

To solve that problem, when a client reads from the database, it doesn't just send its request to one replica: *read requests are also sent to several nodes in parallel.* The client may get different responses from different nodes; i.e., the up-to-date value from one node and a stale value from another. Version numbers are used to determine which value is newer (see "Detecting Concurrent Writes" on page 184).

Read repair and anti-entropy

The replication scheme should ensure that eventually all the data is copied to every replica. After an unavailable node comes back online, how does it catch up on the writes that it missed?

Two mechanisms are often used in Dynamo-style datastores:

Read repair
: When a client makes a read from several nodes in parallel, it can detect any stale responses. For example, in Figure 5-10, user 2345 gets a version 6 value from replica 3 and a version 7 value from replicas 1 and 2. The client sees that replica 3 has a stale value and writes the newer value back to that replica. This approach works well for values that are frequently read.

Anti-entropy process
: In addition, some datastores have a background process that constantly looks for differences in the data between replicas and copies any missing data from one replica to another. Unlike the replication log in leader-based replication, this *anti-entropy process* does not copy writes in any particular order, and there may be a significant delay before data is copied.

Not all systems implement both of these; for example, Voldemort currently does not have an anti-entropy process. Note that without an anti-entropy process, values that are rarely read may be missing from some replicas and thus have reduced durability, because read repair is only performed when a value is read by the application.

Quorums for reading and writing

In the example of Figure 5-10, we considered the write to be successful even though it was only processed on two out of three replicas. What if only one out of three replicas accepted the write? How far can we push this?

If we know that every successful write is guaranteed to be present on at least two out of three replicas, that means at most one replica can be stale. Thus, if we read from at least two replicas, we can be sure that at least one of the two is up to date. If the third replica is down or slow to respond, reads can nevertheless continue returning an up-to-date value.

More generally, if there are n replicas, every write must be confirmed by w nodes to be considered successful, and we must query at least r nodes for each read. (In our example, $n = 3$, $w = 2$, $r = 2$.) As long as $w + r > n$, we expect to get an up-to-date value when reading, because at least one of the r nodes we're reading from must be up to date. Reads and writes that obey these r and w values are called *quorum* reads and writes [44].[vii] You can think of r and w as the minimum number of votes required for the read or write to be valid.

vii. Sometimes this kind of quorum is called a *strict quorum*, to contrast with *sloppy quorums* (discussed in "Sloppy Quorums and Hinted Handoff" on page 183).

In Dynamo-style databases, the parameters n, w, and r are typically configurable. A common choice is to make n an odd number (typically 3 or 5) and to set $w = r = (n + 1) / 2$ (rounded up). However, you can vary the numbers as you see fit. For example, a workload with few writes and many reads may benefit from setting $w = n$ and $r = 1$. This makes reads faster, but has the disadvantage that just one failed node causes all database writes to fail.

There may be more than n nodes in the cluster, but any given value is stored only on n nodes. This allows the dataset to be partitioned, supporting datasets that are larger than you can fit on one node. We will return to partitioning in Chapter 6.

The quorum condition, $w + r > n$, allows the system to tolerate unavailable nodes as follows:

- If $w < n$, we can still process writes if a node is unavailable.
- If $r < n$, we can still process reads if a node is unavailable.
- With $n = 3$, $w = 2$, $r = 2$ we can tolerate one unavailable node.
- With $n = 5$, $w = 3$, $r = 3$ we can tolerate two unavailable nodes. This case is illustrated in Figure 5-11.
- Normally, reads and writes are always sent to all n replicas in parallel. The parameters w and r determine how many nodes we wait for—i.e., how many of the n nodes need to report success before we consider the read or write to be successful.

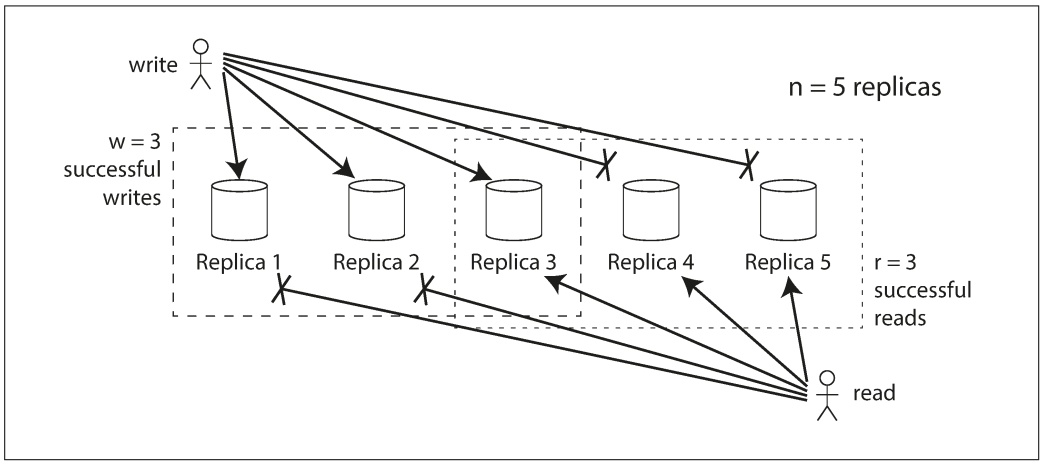


Figure 5-11. If $w + r > n$, at least one of the r replicas you read from must have seen the most recent successful write.

If fewer than the required w or r nodes are available, writes or reads return an error. A node could be unavailable for many reasons: because the node is down (crashed, powered down), due to an error executing the operation (can't write because the disk is full), due to a network interruption between the client and the node, or for any number of other reasons. We only care whether the node returned a successful response and don't need to distinguish between different kinds of fault.

Limitations of Quorum Consistency

If you have n replicas, and you choose w and r such that $w + r > n$, you can generally expect every read to return the most recent value written for a key. This is the case because the set of nodes to which you've written and the set of nodes from which you've read must overlap. That is, among the nodes you read there must be at least one node with the latest value (illustrated in Figure 5-11).

Often, r and w are chosen to be a majority (more than $n/2$) of nodes, because that ensures $w + r > n$ while still tolerating up to $n/2$ node failures. But quorums are not necessarily majorities—it only matters that the sets of nodes used by the read and write operations overlap in at least one node. Other quorum assignments are possible, which allows some flexibility in the design of distributed algorithms [45].

You may also set w and r to smaller numbers, so that $w + r \leq n$ (i.e., the quorum condition is not satisfied). In this case, reads and writes will still be sent to n nodes, but a smaller number of successful responses is required for the operation to succeed.

With a smaller w and r you are more likely to read stale values, because it's more likely that your read didn't include the node with the latest value. On the upside, this configuration allows lower latency and higher availability: if there is a network interruption and many replicas become unreachable, there's a higher chance that you can continue processing reads and writes. Only after the number of reachable replicas falls below w or r does the database become unavailable for writing or reading, respectively.

However, even with $w + r > n$, there are likely to be edge cases where stale values are returned. These depend on the implementation, but possible scenarios include:

- If a sloppy quorum is used (see "Sloppy Quorums and Hinted Handoff" on page 183), the w writes may end up on different nodes than the r reads, so there is no longer a guaranteed overlap between the r nodes and the w nodes [46].
- If two writes occur concurrently, it is not clear which one happened first. In this case, the only safe solution is to merge the concurrent writes (see "Handling Write Conflicts" on page 171). If a winner is picked based on a timestamp (last write wins), writes can be lost due to clock skew [35]. We will return to this topic in "Detecting Concurrent Writes" on page 184.

- If a write happens concurrently with a read, the write may be reflected on only some of the replicas. In this case, it's undetermined whether the read returns the old or the new value.
- If a write succeeded on some replicas but failed on others (for example because the disks on some nodes are full), and overall succeeded on fewer than *w* replicas, it is not rolled back on the replicas where it succeeded. This means that if a write was reported as failed, subsequent reads may or may not return the value from that write [47].
- If a node carrying a new value fails, and its data is restored from a replica carrying an old value, the number of replicas storing the new value may fall below *w*, breaking the quorum condition.
- Even if everything is working correctly, there are edge cases in which you can get unlucky with the timing, as we shall see in "Linearizability and quorums" on page 334.

Thus, although quorums appear to guarantee that a read returns the latest written value, in practice it is not so simple. Dynamo-style databases are generally optimized for use cases that can tolerate eventual consistency. The parameters *w* and *r* allow you to adjust the probability of stale values being read, but it's wise to not take them as absolute guarantees.

In particular, you usually do not get the guarantees discussed in "Problems with Replication Lag" on page 161 (reading your writes, monotonic reads, or consistent prefix reads), so the previously mentioned anomalies can occur in applications. Stronger guarantees generally require transactions or consensus. We will return to these topics in Chapter 7 and Chapter 9.

Monitoring staleness

From an operational perspective, it's important to monitor whether your databases are returning up-to-date results. Even if your application can tolerate stale reads, you need to be aware of the health of your replication. If it falls behind significantly, it should alert you so that you can investigate the cause (for example, a problem in the network or an overloaded node).

For leader-based replication, the database typically exposes metrics for the replication lag, which you can feed into a monitoring system. This is possible because writes are applied to the leader and to followers in the same order, and each node has a position in the replication log (the number of writes it has applied locally). By subtracting a follower's current position from the leader's current position, you can measure the amount of replication lag.

However, in systems with leaderless replication, there is no fixed order in which writes are applied, which makes monitoring more difficult. Moreover, if the database

only uses read repair (no anti-entropy), there is no limit to how old a value might be—if a value is only infrequently read, the value returned by a stale replica may be ancient.

There has been some research on measuring replica staleness in databases with leaderless replication and predicting the expected percentage of stale reads depending on the parameters *n*, *w*, and *r* [48]. This is unfortunately not yet common practice, but it would be good to include staleness measurements in the standard set of metrics for databases. Eventual consistency is a deliberately vague guarantee, but for operability it's important to be able to quantify "eventual."

Sloppy Quorums and Hinted Handoff

Databases with appropriately configured quorums can tolerate the failure of individual nodes without the need for failover. They can also tolerate individual nodes going slow, because requests don't have to wait for all *n* nodes to respond—they can return when *w* or *r* nodes have responded. These characteristics make databases with leaderless replication appealing for use cases that require high availability and low latency, and that can tolerate occasional stale reads.

However, quorums (as described so far) are not as fault-tolerant as they could be. A network interruption can easily cut off a client from a large number of database nodes. Although those nodes are alive, and other clients may be able to connect to them, to a client that is cut off from the database nodes, they might as well be dead. In this situation, it's likely that fewer than *w* or *r* reachable nodes remain, so the client can no longer reach a quorum.

In a large cluster (with significantly more than *n* nodes) it's likely that the client can connect to *some* database nodes during the network interruption, just not to the nodes that it needs to assemble a quorum for a particular value. In that case, database designers face a trade-off:

- Is it better to return errors to all requests for which we cannot reach a quorum of *w* or *r* nodes?
- Or should we accept writes anyway, and write them to some nodes that are reachable but aren't among the *n* nodes on which the value usually lives?

The latter is known as a *sloppy quorum* [37]: writes and reads still require *w* and *r* successful responses, but those may include nodes that are not among the designated *n* "home" nodes for a value. By analogy, if you lock yourself out of your house, you may knock on the neighbor's door and ask whether you may stay on their couch temporarily.

Once the network interruption is fixed, any writes that one node temporarily accepted on behalf of another node are sent to the appropriate "home" nodes. This is

called *hinted handoff*. (Once you find the keys to your house again, your neighbor politely asks you to get off their couch and go home.)

Sloppy quorums are particularly useful for increasing write availability: as long as *any* w nodes are available, the database can accept writes. However, this means that even when $w + r > n$, you cannot be sure to read the latest value for a key, because the latest value may have been temporarily written to some nodes outside of n [47].

Thus, a sloppy quorum actually isn't a quorum at all in the traditional sense. It's only an assurance of durability, namely that the data is stored on w nodes somewhere. There is no guarantee that a read of r nodes will see it until the hinted handoff has completed.

Sloppy quorums are optional in all common Dynamo implementations. In Riak they are enabled by default, and in Cassandra and Voldemort they are disabled by default [46, 49, 50].

Multi-datacenter operation

We previously discussed cross-datacenter replication as a use case for multi-leader replication (see "Multi-Leader Replication" on page 168). Leaderless replication is also suitable for multi-datacenter operation, since it is designed to tolerate conflicting concurrent writes, network interruptions, and latency spikes.

Cassandra and Voldemort implement their multi-datacenter support within the normal leaderless model: the number of replicas n includes nodes in all datacenters, and in the configuration you can specify how many of the n replicas you want to have in each datacenter. Each write from a client is sent to all replicas, regardless of datacenter, but the client usually only waits for acknowledgment from a quorum of nodes within its local datacenter so that it is unaffected by delays and interruptions on the cross-datacenter link. The higher-latency writes to other datacenters are often configured to happen asynchronously, although there is some flexibility in the configuration [50, 51].

Riak keeps all communication between clients and database nodes local to one datacenter, so n describes the number of replicas within one datacenter. Cross-datacenter replication between database clusters happens asynchronously in the background, in a style that is similar to multi-leader replication [52].

Detecting Concurrent Writes

Dynamo-style databases allow several clients to concurrently write to the same key, which means that conflicts will occur even if strict quorums are used. The situation is similar to multi-leader replication (see "Handling Write Conflicts" on page 171), although in Dynamo-style databases conflicts can also arise during read repair or hinted handoff.

The problem is that events may arrive in a different order at different nodes, due to variable network delays and partial failures. For example, Figure 5-12 shows two clients, A and B, simultaneously writing to a key *X* in a three-node datastore:

- Node 1 receives the write from A, but never receives the write from B due to a transient outage.
- Node 2 first receives the write from A, then the write from B.
- Node 3 first receives the write from B, then the write from A.

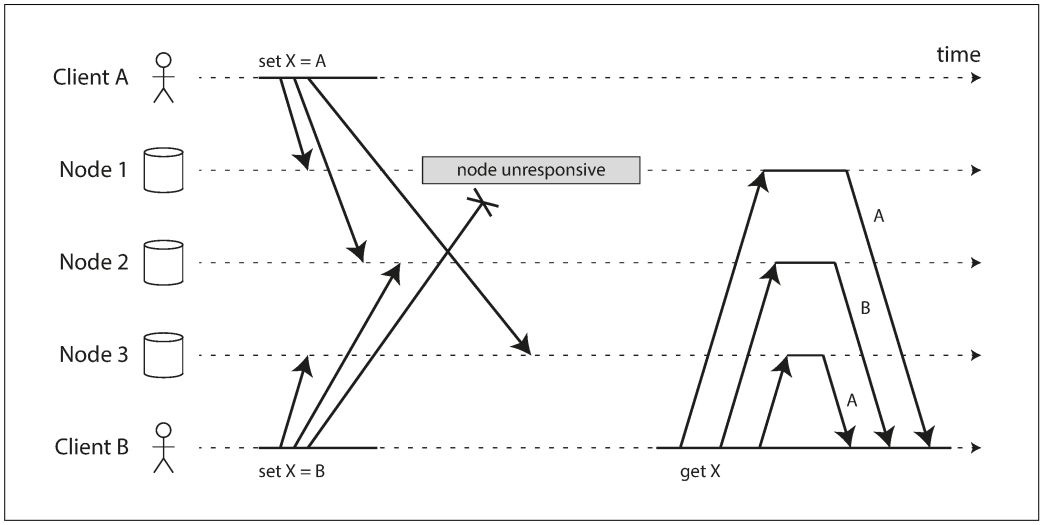


Figure 5-12. Concurrent writes in a Dynamo-style datastore: there is no well-defined ordering.

If each node simply overwrote the value for a key whenever it received a write request from a client, the nodes would become permanently inconsistent, as shown by the final *get* request in Figure 5-12: node 2 thinks that the final value of *X* is B, whereas the other nodes think that the value is A.

In order to become eventually consistent, the replicas should converge toward the same value. How do they do that? One might hope that replicated databases would handle this automatically, but unfortunately most implementations are quite poor: if you want to avoid losing data, you—the application developer—need to know a lot about the internals of your database's conflict handling.

We briefly touched on some techniques for conflict resolution in "Handling Write Conflicts" on page 171. Before we wrap up this chapter, let's explore the issue in a bit more detail.

Last write wins (discarding concurrent writes)

One approach for achieving eventual convergence is to declare that each replica need only store the most "recent" value and allow "older" values to be overwritten and discarded. Then, as long as we have some way of unambiguously determining which write is more "recent," and every write is eventually copied to every replica, the replicas will eventually converge to the same value.

As indicated by the quotes around "recent," this idea is actually quite misleading. In the example of Figure 5-12, neither client knew about the other one when it sent its write requests to the database nodes, so it's not clear which one happened first. In fact, it doesn't really make sense to say that either happened "first": we say the writes are *concurrent*, so their order is undefined.

Even though the writes don't have a natural ordering, we can force an arbitrary order on them. For example, we can attach a timestamp to each write, pick the biggest timestamp as the most "recent," and discard any writes with an earlier timestamp. This conflict resolution algorithm, called *last write wins* (LWW), is the only supported conflict resolution method in Cassandra [53], and an optional feature in Riak [35].

LWW achieves the goal of eventual convergence, but at the cost of durability: if there are several concurrent writes to the same key, even if they were all reported as successful to the client (because they were written to *w* replicas), only one of the writes will survive and the others will be silently discarded. Moreover, LWW may even drop writes that are not concurrent, as we shall discuss in "Timestamps for ordering events" on page 291.

There are some situations, such as caching, in which lost writes are perhaps acceptable. If losing data is not acceptable, LWW is a poor choice for conflict resolution.

The only safe way of using a database with LWW is to ensure that a key is only written once and thereafter treated as immutable, thus avoiding any concurrent updates to the same key. For example, a recommended way of using Cassandra is to use a UUID as the key, thus giving each write operation a unique key [53].

The "happens-before" relationship and concurrency

How do we decide whether two operations are concurrent or not? To develop an intuition, let's look at some examples:

- In Figure 5-9, the two writes are not concurrent: A's insert *happens before* B's increment, because the value incremented by B is the value inserted by A. In other words, B's operation builds upon A's operation, so B's operation must have happened later. We also say that B is *causally dependent* on A.

- On the other hand, the two writes in Figure 5-12 are concurrent: when each client starts the operation, it does not know that another client is also performing an operation on the same key. Thus, there is no causal dependency between the operations.

An operation A *happens before* another operation B if B knows about A, or depends on A, or builds upon A in some way. Whether one operation happens before another operation is the key to defining what concurrency means. In fact, we can simply say that two operations are *concurrent* if neither happens before the other (i.e., neither knows about the other) [54].

Thus, whenever you have two operations A and B, there are three possibilities: either A happened before B, or B happened before A, or A and B are concurrent. What we need is an algorithm to tell us whether two operations are concurrent or not. If one operation happened before another, the later operation should overwrite the earlier operation, but if the operations are concurrent, we have a conflict that needs to be resolved.

Concurrency, Time, and Relativity

It may seem that two operations should be called concurrent if they occur "at the same time"—but in fact, it is not important whether they literally overlap in time. Because of problems with clocks in distributed systems, it is actually quite difficult to tell whether two things happened at exactly the same time—an issue we will discuss in more detail in Chapter 8.

For defining concurrency, exact time doesn't matter: we simply call two operations concurrent if they are both unaware of each other, regardless of the physical time at which they occurred. People sometimes make a connection between this principle and the special theory of relativity in physics [54], which introduced the idea that information cannot travel faster than the speed of light. Consequently, two events that occur some distance apart cannot possibly affect each other if the time between the events is shorter than the time it takes light to travel the distance between them.

In computer systems, two operations might be concurrent even though the speed of light would in principle have allowed one operation to affect the other. For example, if the network was slow or interrupted at the time, two operations can occur some time apart and still be concurrent, because the network problems prevented one operation from being able to know about the other.

Capturing the happens-before relationship

Let's look at an algorithm that determines whether two operations are concurrent, or whether one happened before another. To keep things simple, let's start with a data-

base that has only one replica. Once we have worked out how to do this on a single replica, we can generalize the approach to a leaderless database with multiple replicas.

Figure 5-13 shows two clients concurrently adding items to the same shopping cart. (If that example strikes you as too inane, imagine instead two air traffic controllers concurrently adding aircraft to the sector they are tracking.) Initially, the cart is empty. Between them, the clients make five writes to the database:

1. Client 1 adds `milk` to the cart. This is the first write to that key, so the server successfully stores it and assigns it version 1. The server also echoes the value back to the client, along with the version number.
2. Client 2 adds `eggs` to the cart, not knowing that client 1 concurrently added `milk` (client 2 thought that its `eggs` were the only item in the cart). The server assigns version 2 to this write, and stores `eggs` and `milk` as two separate values. It then returns *both* values to the client, along with the version number of 2.
3. Client 1, oblivious to client 2's write, wants to add `flour` to the cart, so it thinks the current cart contents should be `[milk, flour]`. It sends this value to the server, along with the version number 1 that the server gave client 1 previously. The server can tell from the version number that the write of `[milk, flour]` supersedes the prior value of `[milk]` but that it is concurrent with `[eggs]`. Thus, the server assigns version 3 to `[milk, flour]`, overwrites the version 1 value `[milk]`, but keeps the version 2 value `[eggs]` and returns both remaining values to the client.
4. Meanwhile, client 2 wants to add `ham` to the cart, unaware that client 1 just added `flour`. Client 2 received the two values `[milk]` and `[eggs]` from the server in the last response, so the client now merges those values and adds `ham` to form a new value, `[eggs, milk, ham]`. It sends that value to the server, along with the previous version number 2. The server detects that version 2 overwrites `[eggs]` but is concurrent with `[milk, flour]`, so the two remaining values are `[milk, flour]` with version 3, and `[eggs, milk, ham]` with version 4.
5. Finally, client 1 wants to add `bacon`. It previously received `[milk, flour]` and `[eggs]` from the server at version 3, so it merges those, adds `bacon`, and sends the final value `[milk, flour, eggs, bacon]` to the server, along with the version number 3. This overwrites `[milk, flour]` (note that `[eggs]` was already overwritten in the last step) but is concurrent with `[eggs, milk, ham]`, so the server keeps those two concurrent values.

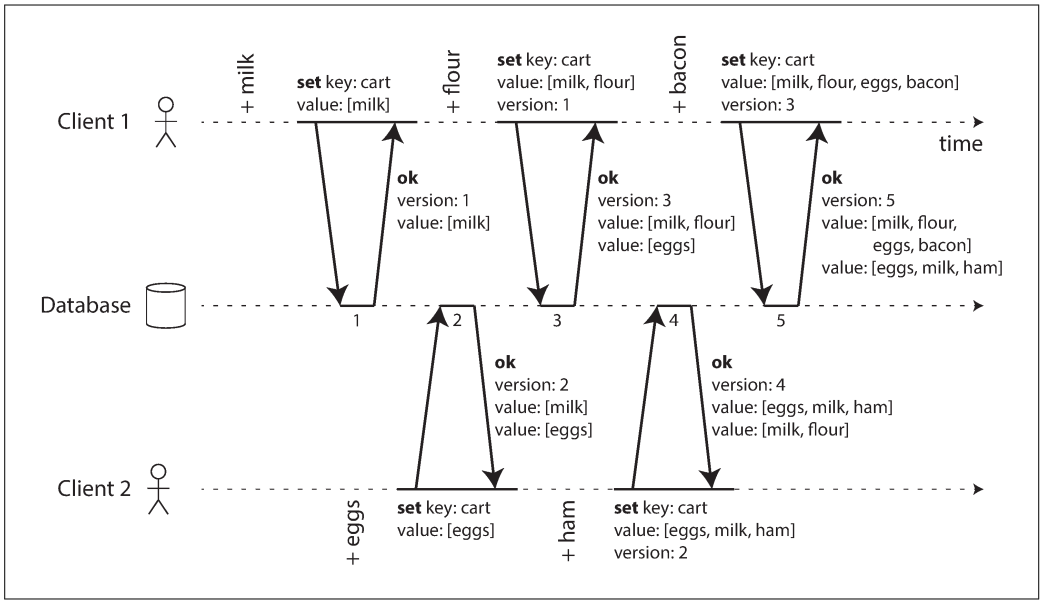


Figure 5-13. Capturing causal dependencies between two clients concurrently editing a shopping cart.

The dataflow between the operations in Figure 5-13 is illustrated graphically in Figure 5-14. The arrows indicate which operation *happened before* which other operation, in the sense that the later operation *knew about* or *depended on* the earlier one. In this example, the clients are never fully up to date with the data on the server, since there is always another operation going on concurrently. But old versions of the value do get overwritten eventually, and no writes are lost.

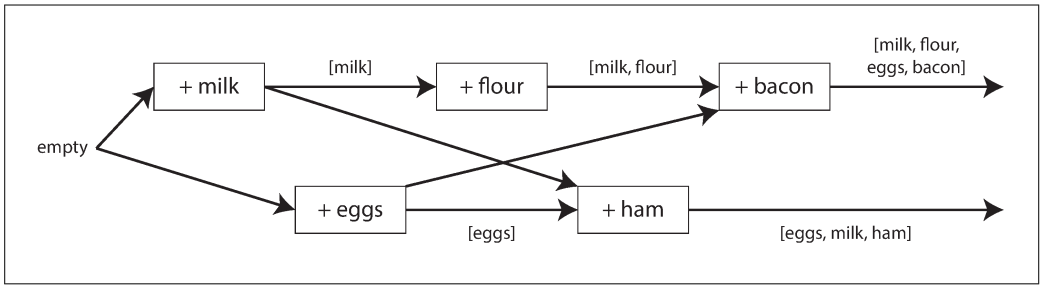


Figure 5-14. Graph of causal dependencies in Figure 5-13.

Note that the server can determine whether two operations are concurrent by looking at the version numbers—it does not need to interpret the value itself (so the value could be any data structure). The algorithm works as follows:

- The server maintains a version number for every key, increments the version number every time that key is written, and stores the new version number along with the value written.
- When a client reads a key, the server returns all values that have not been overwritten, as well as the latest version number. A client must read a key before writing.
- When a client writes a key, it must include the version number from the prior read, and it must merge together all values that it received in the prior read. (The response from a write request can be like a read, returning all current values, which allows us to chain several writes like in the shopping cart example.)
- When the server receives a write with a particular version number, it can overwrite all values with that version number or below (since it knows that they have been merged into the new value), but it must keep all values with a higher version number (because those values are concurrent with the incoming write).

When a write includes the version number from a prior read, that tells us which previous state the write is based on. If you make a write without including a version number, it is concurrent with all other writes, so it will not overwrite anything—it will just be returned as one of the values on subsequent reads.

Merging concurrently written values

This algorithm ensures that no data is silently dropped, but it unfortunately requires that the clients do some extra work: if several operations happen concurrently, clients have to clean up afterward by merging the concurrently written values. Riak calls these concurrent values *siblings*.

Merging sibling values is essentially the same problem as conflict resolution in multi-leader replication, which we discussed previously (see "Handling Write Conflicts" on page 171). A simple approach is to just pick one of the values based on a version number or timestamp (last write wins), but that implies losing data. So, you may need to do something more intelligent in application code.

With the example of a shopping cart, a reasonable approach to merging siblings is to just take the union. In Figure 5-14, the two final siblings are `[milk, flour, eggs, bacon]` and `[eggs, milk, ham]`; note that `milk` and `eggs` appear in both, even though they were each only written once. The merged value might be something like `[milk, flour, eggs, bacon, ham]`, without duplicates.

However, if you want to allow people to also *remove* things from their carts, and not just add things, then taking the union of siblings may not yield the right result: if you merge two sibling carts and an item has been removed in only one of them, then the removed item will reappear in the union of the siblings [37]. To prevent this prob-

lem, an item cannot simply be deleted from the database when it is removed; instead, the system must leave a marker with an appropriate version number to indicate that the item has been removed when merging siblings. Such a deletion marker is known as a *tombstone*. (We previously saw tombstones in the context of log compaction in "Hash Indexes" on page 72.)

As merging siblings in application code is complex and error-prone, there are some efforts to design data structures that can perform this merging automatically, as discussed in "Automatic Conflict Resolution" on page 174. For example, Riak's datatype support uses a family of data structures called CRDTs [38, 39, 55] that can automatically merge siblings in sensible ways, including preserving deletions.

Version vectors

The example in Figure 5-13 used only a single replica. How does the algorithm change when there are multiple replicas, but no leader?

Figure 5-13 uses a single version number to capture dependencies between operations, but that is not sufficient when there are multiple replicas accepting writes concurrently. Instead, we need to use a version number *per replica* as well as per key. Each replica increments its own version number when processing a write, and also keeps track of the version numbers it has seen from each of the other replicas. This information indicates which values to overwrite and which values to keep as siblings.

The collection of version numbers from all the replicas is called a *version vector* [56]. A few variants of this idea are in use, but the most interesting is probably the *dotted version vector* [57], which is used in Riak 2.0 [58, 59]. We won't go into the details, but the way it works is quite similar to what we saw in our cart example.

Like the version numbers in Figure 5-13, version vectors are sent from the database replicas to clients when values are read, and need to be sent back to the database when a value is subsequently written. (Riak encodes the version vector as a string that it calls *causal context*.) The version vector allows the database to distinguish between overwrites and concurrent writes.

Also, like in the single-replica example, the application may need to merge siblings. The version vector structure ensures that it is safe to read from one replica and subsequently write back to another replica. Doing so may result in siblings being created, but no data is lost as long as siblings are merged correctly.

Version vectors and vector clocks

A *version vector* is sometimes also called a *vector clock*, even though they are not quite the same. The difference is subtle—please see the references for details [57, 60, 61]. In brief, when comparing the state of replicas, version vectors are the right data structure to use.

Summary

In this chapter we looked at the issue of replication. Replication can serve several purposes:

High availability
: Keeping the system running, even when one machine (or several machines, or an entire datacenter) goes down

Disconnected operation
: Allowing an application to continue working when there is a network interruption

Latency
: Placing data geographically close to users, so that users can interact with it faster

Scalability
: Being able to handle a higher volume of reads than a single machine could handle, by performing reads on replicas

Despite being a simple goal—keeping a copy of the same data on several machines—replication turns out to be a remarkably tricky problem. It requires carefully thinking about concurrency and about all the things that can go wrong, and dealing with the consequences of those faults. At a minimum, we need to deal with unavailable nodes and network interruptions (and that's not even considering the more insidious kinds of fault, such as silent data corruption due to software bugs).

We discussed three main approaches to replication:

Single-leader replication
: Clients send all writes to a single node (the leader), which sends a stream of data change events to the other replicas (followers). Reads can be performed on any replica, but reads from followers might be stale.

Multi-leader replication
: Clients send each write to one of several leader nodes, any of which can accept writes. The leaders send streams of data change events to each other and to any follower nodes.

Leaderless replication
: Clients send each write to several nodes, and read from several nodes in parallel in order to detect and correct nodes with stale data.

Each approach has advantages and disadvantages. Single-leader replication is popular because it is fairly easy to understand and there is no conflict resolution to worry about. Multi-leader and leaderless replication can be more robust in the presence of

faulty nodes, network interruptions, and latency spikes—at the cost of being harder to reason about and providing only very weak consistency guarantees.

Replication can be synchronous or asynchronous, which has a profound effect on the system behavior when there is a fault. Although asynchronous replication can be fast when the system is running smoothly, it's important to figure out what happens when replication lag increases and servers fail. If a leader fails and you promote an asynchronously updated follower to be the new leader, recently committed data may be lost.

We looked at some strange effects that can be caused by replication lag, and we discussed a few consistency models which are helpful for deciding how an application should behave under replication lag:

Read-after-write consistency
: Users should always see data that they submitted themselves.

Monotonic reads
: After users have seen the data at one point in time, they shouldn't later see the data from some earlier point in time.

Consistent prefix reads
: Users should see the data in a state that makes causal sense: for example, seeing a question and its reply in the correct order.

Finally, we discussed the concurrency issues that are inherent in multi-leader and leaderless replication approaches: because they allow multiple writes to happen concurrently, conflicts may occur. We examined an algorithm that a database might use to determine whether one operation happened before another, or whether they happened concurrently. We also touched on methods for resolving conflicts by merging together concurrent updates.

In the next chapter we will continue looking at data that is distributed across multiple machines, through the counterpart of replication: splitting a large dataset into *partitions*.

References

[1] Bruce G. Lindsay, Patricia Griffiths Selinger, C. Galtieri, et al.: "Notes on Distributed Databases," IBM Research, Research Report RJ2571(33471), July 1979.

[2] "Oracle Active Data Guard Real-Time Data Protection and Availability," Oracle White Paper, June 2013.

[3] "AlwaysOn Availability Groups," in *SQL Server Books Online*, Microsoft, 2012.

[4] Lin Qiao, Kapil Surlaker, Shirshanka Das, et al.: "On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform," at *ACM International Conference on Management of Data* (SIGMOD), June 2013.

[5] Jun Rao: "Intra-Cluster Replication for Apache Kafka," at *ApacheCon North America*, February 2013.

[6] "Highly Available Queues," in *RabbitMQ Server Documentation*, Pivotal Software, Inc., 2014.

[7] Yoshinori Matsunobu: "Semi-Synchronous Replication at Facebook," *yoshinori-matsunobu.blogspot.co.uk*, April 1, 2014.

[8] Robbert van Renesse and Fred B. Schneider: "Chain Replication for Supporting High Throughput and Availability," at *6th USENIX Symposium on Operating System Design and Implementation* (OSDI), December 2004.

[9] Jeff Terrace and Michael J. Freedman: "Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads," at *USENIX Annual Technical Conference* (ATC), June 2009.

[10] Brad Calder, Ju Wang, Aaron Ogus, et al.: "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," at *23rd ACM Symposium on Operating Systems Principles* (SOSP), October 2011.

[11] Andrew Wang: "Windows Azure Storage," *umbrant.com*, February 4, 2016.

[12] "Percona Xtrabackup - Documentation," Percona LLC, 2014.

[13] Jesse Newland: "GitHub Availability This Week," *github.com*, September 14, 2012.

[14] Mark Imbriaco: "Downtime Last Saturday," *github.com*, December 26, 2012.

[15] John Hugg: "'All in' with Determinism for Performance and Testing in Distributed Systems," at *Strange Loop*, September 2015.

[16] Amit Kapila: "WAL Internals of PostgreSQL," at *PostgreSQL Conference* (PGCon), May 2012.

[17] *MySQL Internals Manual*. Oracle, 2014.

[18] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, et al.: "Wormhole: Reliable Pub-Sub to Support Geo-Replicated Internet Services," at *12th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), May 2015.

[19] "Oracle GoldenGate 12c: Real-Time Access to Real-Time Information," Oracle White Paper, October 2013.

[20] Shirshanka Das, Chavdar Botev, Kapil Surlaker, et al.: "All Aboard the Databus!," at *ACM Symposium on Cloud Computing* (SoCC), October 2012.

[21] Greg Sabino Mullane: "Version 5 of Bucardo Database Replication System," *blog.endpoint.com*, June 23, 2014.

[22] Werner Vogels: "Eventually Consistent," *ACM Queue*, volume 6, number 6, pages 14–19, October 2008. doi:10.1145/1466443.1466448

[23] Douglas B. Terry: "Replicated Data Consistency Explained Through Baseball," Microsoft Research, Technical Report MSR-TR-2011-137, October 2011.

[24] Douglas B. Terry, Alan J. Demers, Karin Petersen, et al.: "Session Guarantees for Weakly Consistent Replicated Data," at *3rd International Conference on Parallel and Distributed Information Systems* (PDIS), September 1994. doi:10.1109/PDIS.1994.331722

[25] Terry Pratchett: *Reaper Man: A Discworld Novel*. Victor Gollancz, 1991. ISBN: 978-0-575-04979-6

[26] "Tungsten Replicator," Continuent, Inc., 2014.

[27] "BDR 0.10.0 Documentation," The PostgreSQL Global Development Group, *bdr-project.org*, 2015.

[28] Robert Hodges: "If You *Must* Deploy Multi-Master Replication, Read This First," *scale-out-blog.blogspot.co.uk*, March 30, 2012.

[29] J. Chris Anderson, Jan Lehnardt, and Noah Slater: *CouchDB: The Definitive Guide*. O'Reilly Media, 2010. ISBN: 978-0-596-15589-6

[30] AppJet, Inc.: "Etherpad and EasySync Technical Manual," *github.com*, March 26, 2011.

[31] John Day-Richter: "What's Different About the New Google Docs: Making Collaboration Fast," *googledrive.blogspot.com*, 23 September 2010.

[32] Martin Kleppmann and Alastair R. Beresford: "A Conflict-Free Replicated JSON Datatype," arXiv:1608.03960, August 13, 2016.

[33] Frazer Clement: "Eventual Consistency – Detecting Conflicts," *messagepassing.blogspot.co.uk*, October 20, 2011.

[34] Robert Hodges: "State of the Art for MySQL Multi-Master Replication," at *Percona Live: MySQL Conference & Expo*, April 2013.

[35] John Daily: "Clocks Are Bad, or, Welcome to the Wonderful World of Distributed Systems," *basho.com*, November 12, 2013.

[36] Riley Berton: "Is Bi-Directional Replication (BDR) in Postgres Transactional?," *sdf.org*, January 4, 2016.

[37] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al.: "Dynamo: Amazon's Highly Available Key-Value Store," at *21st ACM Symposium on Operating Systems Principles* (SOSP), October 2007.

[38] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski: "A Comprehensive Study of Convergent and Commutative Replicated Data Types," INRIA Research Report no. 7506, January 2011.

[39] Sam Elliott: "CRDTs: An UPDATE (or Maybe Just a PUT)," at *RICON West*, October 2013.

[40] Russell Brown: "A Bluffers Guide to CRDTs in Riak," *gist.github.com*, October 28, 2013.

[41] Benjamin Farinier, Thomas Gazagnaire, and Anil Madhavapeddy: "Mergeable Persistent Data Structures," at *26es Journées Francophones des Langages Applicatifs* (JFLA), January 2015.

[42] Chengzheng Sun and Clarence Ellis: "Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements," at *ACM Conference on Computer Supported Cooperative Work* (CSCW), November 1998.

[43] Lars Hofhansl: "HBASE-7709: Infinite Loop Possible in Master/Master Replication," *issues.apache.org*, January 29, 2013.

[44] David K. Gifford: "Weighted Voting for Replicated Data," at *7th ACM Symposium on Operating Systems Principles* (SOSP), December 1979. doi:10.1145/800215.806583

[45] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman: "Flexible Paxos: Quorum Intersection Revisited," *arXiv:1608.06696*, August 24, 2016.

[46] Joseph Blomstedt: "Re: Absolute Consistency," email to *riak-users* mailing list, *lists.basho.com*, January 11, 2012.

[47] Joseph Blomstedt: "Bringing Consistency to Riak," at *RICON West*, October 2012.

[48] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, et al.: "Quantifying Eventual Consistency with PBS," *Communications of the ACM*, volume 57, number 8, pages 93–102, August 2014. doi:10.1145/2632792

[49] Jonathan Ellis: "Modern Hinted Handoff," *datastax.com*, December 11, 2012.

[50] "Project Voldemort Wiki," *github.com*, 2013.

[51] "Apache Cassandra 2.0 Documentation," DataStax, Inc., 2014.

[52] "Riak Enterprise: Multi-Datacenter Replication." Technical whitepaper, Basho Technologies, Inc., September 2014.

[53] Jonathan Ellis: "Why Cassandra Doesn't Need Vector Clocks," *datastax.com*, September 2, 2013.

[54] Leslie Lamport: "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, volume 21, number 7, pages 558–565, July 1978. doi:10.1145/359545.359563

[55] Joel Jacobson: "Riak 2.0: Data Types," *blog.joeljacobson.com*, March 23, 2014.

[56] D. Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, et al.: "Detection of Mutual Inconsistency in Distributed Systems," *IEEE Transactions on Software Engineering*, volume 9, number 3, pages 240–247, May 1983. doi:10.1109/TSE.1983.236733

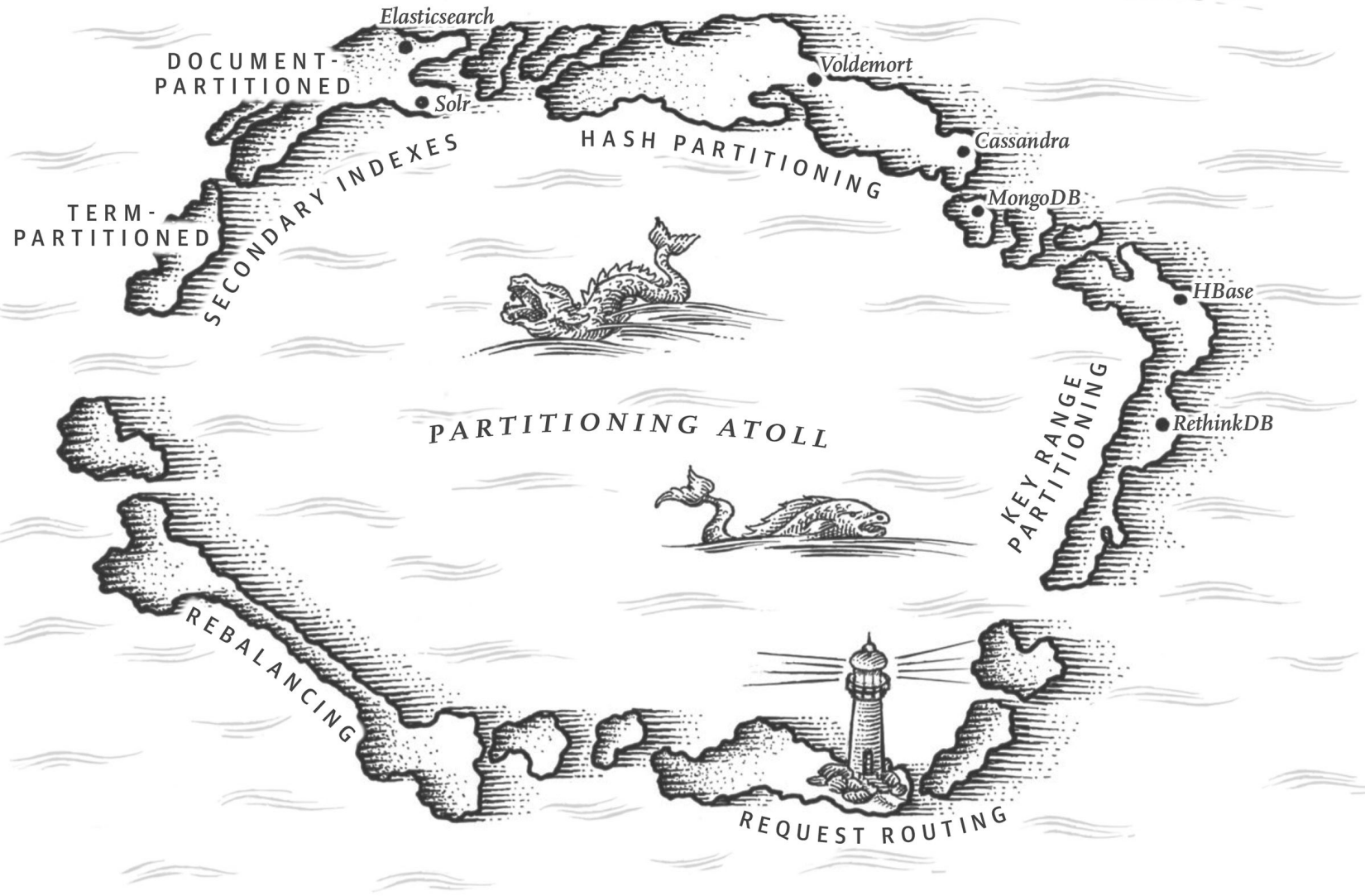
[57] Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida, et al.: "Dotted Version Vectors: Logical Clocks for Optimistic Replication," arXiv:1011.5808, November 26, 2010.

[58] Sean Cribbs: "A Brief History of Time in Riak," at *RICON*, October 2014.

[59] Russell Brown: "Vector Clocks Revisited Part 2: Dotted Version Vectors," *basho.com*, November 10, 2015.

[60] Carlos Baquero: "Version Vectors Are Not Vector Clocks," *haslab.wordpress.com*, July 8, 2011.

[61] Reinhard Schwarz and Friedemann Mattern: "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," *Distributed Computing*, volume 7, number 3, pages 149–174, March 1994. doi:10.1007/BF02277859


Elasticsearch
DOCUMENT-PARTITIONED
Solr
Voldemort
Cassandra
HASH PARTITIONING
MongoDB
TERM-PARTITIONED
SECONDARY INDEXES
HBase
KEY RANGE PARTITIONING
RethinkDB
PARTITIONING ATOLL
REBALANCING
REQUEST ROUTING

CHAPTER 6
Partitioning

> *Clearly, we must break away from the sequential and not limit the computers. We must state definitions and provide for priorities and descriptions of data. We must state relationships, not procedures.*
>
> —Grace Murray Hopper, *Management and the Computer of the Future* (1962)

In Chapter 5 we discussed replication—that is, having multiple copies of the same data on different nodes. For very large datasets, or very high query throughput, that is not sufficient: we need to break the data up into *partitions*, also known as *sharding*.[i]

Terminological confusion

What we call a *partition* here is called a *shard* in MongoDB, Elasticsearch, and SolrCloud; it's known as a *region* in HBase, a *tablet* in Bigtable, a *vnode* in Cassandra and Riak, and a *vBucket* in Couchbase. However, *partitioning* is the most established term, so we'll stick with that.

Normally, partitions are defined in such a way that each piece of data (each record, row, or document) belongs to exactly one partition. There are various ways of achieving this, which we discuss in depth in this chapter. In effect, each partition is a small database of its own, although the database may support operations that touch multiple partitions at the same time.

The main reason for wanting to partition data is *scalability*. Different partitions can be placed on different nodes in a shared-nothing cluster (see the introduction to

i. Partitioning, as discussed in this chapter, is a way of intentionally breaking a large database down into smaller ones. It has nothing to do with *network partitions* (netsplits), a type of fault in the network between nodes. We will discuss such faults in Chapter 8.

Part II for a definition of *shared nothing*). Thus, a large dataset can be distributed across many disks, and the query load can be distributed across many processors.

For queries that operate on a single partition, each node can independently execute the queries for its own partition, so query throughput can be scaled by adding more nodes. Large, complex queries can potentially be parallelized across many nodes, although this gets significantly harder.

Partitioned databases were pioneered in the 1980s by products such as Teradata and Tandem NonStop SQL [1], and more recently rediscovered by NoSQL databases and Hadoop-based data warehouses. Some systems are designed for transactional workloads, and others for analytics (see "Transaction Processing or Analytics?" on page 90): this difference affects how the system is tuned, but the fundamentals of partitioning apply to both kinds of workloads.

In this chapter we will first look at different approaches for partitioning large datasets and observe how the indexing of data interacts with partitioning. We'll then talk about rebalancing, which is necessary if you want to add or remove nodes in your cluster. Finally, we'll get an overview of how databases route requests to the right partitions and execute queries.

Partitioning and Replication

Partitioning is usually combined with replication so that copies of each partition are stored on multiple nodes. This means that, even though each record belongs to exactly one partition, it may still be stored on several different nodes for fault tolerance.

A node may store more than one partition. If a leader–follower replication model is used, the combination of partitioning and replication can look like Figure 6-1. Each partition's leader is assigned to one node, and its followers are assigned to other nodes. Each node may be the leader for some partitions and a follower for other partitions.

Everything we discussed in Chapter 5 about replication of databases applies equally to replication of partitions. The choice of partitioning scheme is mostly independent of the choice of replication scheme, so we will keep things simple and ignore replication in this chapter.

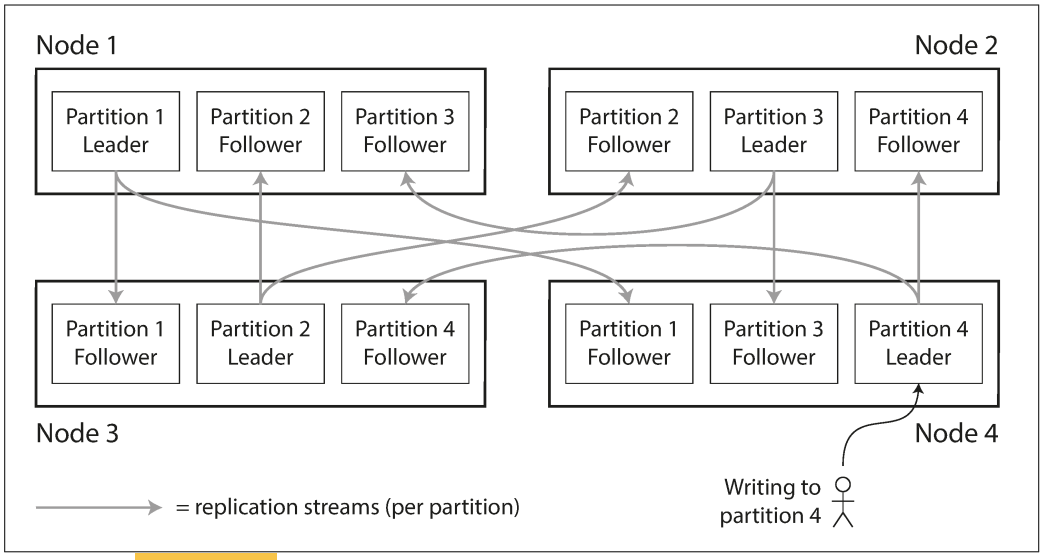


Figure 6-1. Combining replication and partitioning: each node acts as leader for some partitions and follower for other partitions.

Partitioning of Key-Value Data

Say you have a large amount of data, and you want to partition it. How do you decide which records to store on which nodes?

Our goal with partitioning is to spread the data and the query load evenly across nodes. If every node takes a fair share, then—in theory—10 nodes should be able to handle 10 times as much data and 10 times the read and write throughput of a single node (ignoring replication for now).

If the partitioning is unfair, so that some partitions have more data or queries than others, we call it *skewed*. The presence of skew makes partitioning much less effective. In an extreme case, all the load could end up on one partition, so 9 out of 10 nodes are idle and your bottleneck is the single busy node. A partition with disproportionately high load is called a *hot spot*.

The simplest approach for avoiding hot spots would be to assign records to nodes randomly. That would distribute the data quite evenly across the nodes, but it has a big disadvantage: when you're trying to read a particular item, you have no way of knowing which node it is on, so you have to query all nodes in parallel.

We can do better. Let's assume for now that you have a simple key-value data model, in which you always access a record by its primary key. For example, in an old-fashioned paper encyclopedia, you look up an entry by its title; since all the entries are alphabetically sorted by title, you can quickly find the one you're looking for.

Partitioning by Key Range

One way of partitioning is to assign a continuous range of keys (from some minimum to some maximum) to each partition, like the volumes of a paper encyclopedia (Figure 6-2). If you know the boundaries between the ranges, you can easily determine which partition contains a given key. If you also know which partition is assigned to which node, then you can make your request directly to the appropriate node (or, in the case of the encyclopedia, pick the correct book off the shelf).

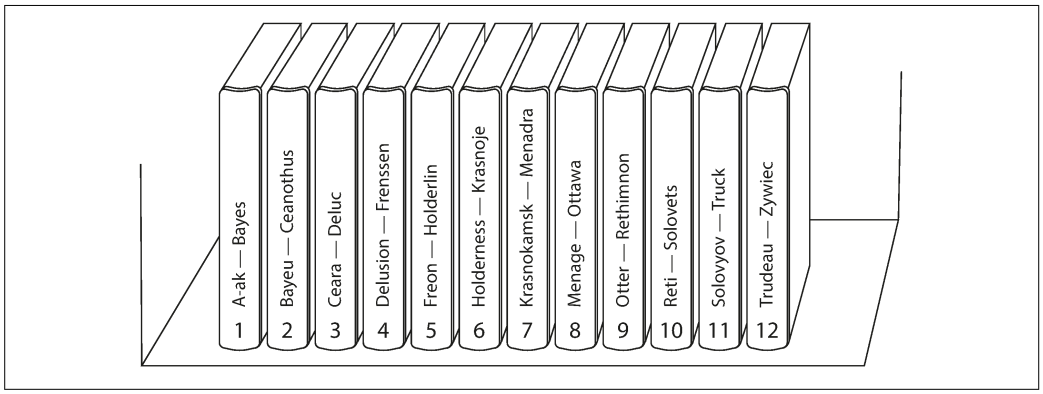


Figure 6-2. A print encyclopedia is partitioned by key range.

The ranges of keys are not necessarily evenly spaced, because your data may not be evenly distributed. For example, in Figure 6-2, volume 1 contains words starting with A and B, but volume 12 contains words starting with T, U, V, X, Y, and Z. Simply having one volume per two letters of the alphabet would lead to some volumes being much bigger than others. In order to distribute the data evenly, the partition boundaries need to adapt to the data.

The partition boundaries might be chosen manually by an administrator, or the database can choose them automatically (we will discuss choices of partition boundaries in more detail in "Rebalancing Partitions" on page 209). This partitioning strategy is used by Bigtable, its open source equivalent HBase [2, 3], RethinkDB, and MongoDB before version 2.4 [4].

Within each partition, we can keep keys in sorted order (see "SSTables and LSM-Trees" on page 76). This has the advantage that range scans are easy, and you can treat the key as a concatenated index in order to fetch several related records in one query (see "Multi-column indexes" on page 87). For example, consider an application that stores data from a network of sensors, where the key is the timestamp of the measurement (*year-month-day-hour-minute-second*). Range scans are very useful in this case, because they let you easily fetch, say, all the readings from a particular month.

However, the downside of key range partitioning is that certain access patterns can lead to hot spots. If the key is a timestamp, then the partitions correspond to ranges of time—e.g., one partition per day. Unfortunately, because we write data from the sensors to the database as the measurements happen, all the writes end up going to the same partition (the one for today), so that partition can be overloaded with writes while others sit idle [5].

To avoid this problem in the sensor database, you need to use something other than the timestamp as the first element of the key. For example, you could prefix each timestamp with the sensor name so that the partitioning is first by sensor name and then by time. Assuming you have many sensors active at the same time, the write load will end up more evenly spread across the partitions. Now, when you want to fetch the values of multiple sensors within a time range, you need to perform a separate range query for each sensor name.

Partitioning by Hash of Key

Because of this risk of skew and hot spots, many distributed datastores use a hash function to determine the partition for a given key.

A good hash function takes skewed data and makes it uniformly distributed. Say you have a 32-bit hash function that takes a string. Whenever you give it a new string, it returns a seemingly random number between 0 and $2^{32} - 1$. Even if the input strings are very similar, their hashes are evenly distributed across that range of numbers.

For partitioning purposes, the hash function need not be cryptographically strong: for example, Cassandra and MongoDB use MD5, and Voldemort uses the Fowler–Noll–Vo function. Many programming languages have simple hash functions built in (as they are used for hash tables), but they may not be suitable for partitioning: for example, in Java's `Object.hashCode()` and Ruby's `Object#hash`, the same key may have a different hash value in different processes [6].

Once you have a suitable hash function for keys, you can assign each partition a range of hashes (rather than a range of keys), and every key whose hash falls within a partition's range will be stored in that partition. This is illustrated in Figure 6-3.

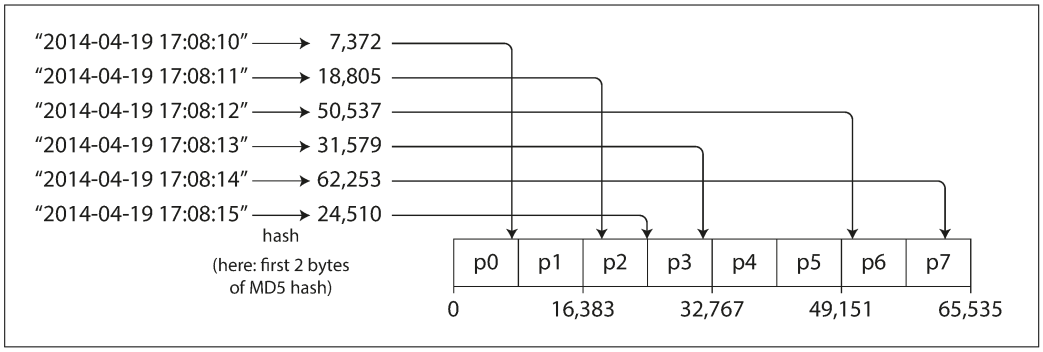


Figure 6-3. Partitioning by hash of key.

This technique is good at distributing keys fairly among the partitions. The partition boundaries can be evenly spaced, or they can be chosen pseudorandomly (in which case the technique is sometimes known as *consistent hashing*).

Consistent Hashing

Consistent hashing, as defined by Karger et al. [7], is a way of evenly distributing load across an internet-wide system of caches such as a content delivery network (CDN). It uses randomly chosen partition boundaries to avoid the need for central control or distributed consensus. Note that *consistent* here has nothing to do with replica consistency (see Chapter 5) or ACID consistency (see Chapter 7), but rather describes a particular approach to rebalancing.

As we shall see in "Rebalancing Partitions" on page 209, this particular approach actually doesn't work very well for databases [8], so it is rarely used in practice (the documentation of some databases still refers to consistent hashing, but it is often inaccurate). Because this is so confusing, it's best to avoid the term *consistent hashing* and just call it *hash partitioning* instead.

Unfortunately however, by using the hash of the key for partitioning we lose a nice property of key-range partitioning: the ability to do efficient range queries. Keys that were once adjacent are now scattered across all the partitions, so their sort order is lost. In MongoDB, if you have enabled hash-based sharding mode, any range query has to be sent to all partitions [4]. Range queries on the primary key are not supported by Riak [9], Couchbase [10], or Voldemort.

Cassandra achieves a compromise between the two partitioning strategies [11, 12, 13]. A table in Cassandra can be declared with a *compound primary key* consisting of several columns. Only the first part of that key is hashed to determine the partition, but the other columns are used as a concatenated index for sorting the data in Cassandra's SSTables. A query therefore cannot search for a range of values within the

first column of a compound key, but if it specifies a fixed value for the first column, it can perform an efficient range scan over the other columns of the key.

The concatenated index approach enables an elegant data model for one-to-many relationships. For example, on a social media site, one user may post many updates. If the primary key for updates is chosen to be `(user_id, update_timestamp)`, then you can efficiently retrieve all updates made by a particular user within some time interval, sorted by timestamp. Different users may be stored on different partitions, but within each user, the updates are stored ordered by timestamp on a single partition.

Skewed Workloads and Relieving Hot Spots

As discussed, hashing a key to determine its partition can help reduce hot spots. However, it can't avoid them entirely: in the extreme case where all reads and writes are for the same key, you still end up with all requests being routed to the same partition.

This kind of workload is perhaps unusual, but not unheard of: for example, on a social media site, a celebrity user with millions of followers may cause a storm of activity when they do something [14]. This event can result in a large volume of writes to the same key (where the key is perhaps the user ID of the celebrity, or the ID of the action that people are commenting on). Hashing the key doesn't help, as the hash of two identical IDs is still the same.

Today, most data systems are not able to automatically compensate for such a highly skewed workload, so it's the responsibility of the application to reduce the skew. For example, if one key is known to be very hot, a simple technique is to add a random number to the beginning or end of the key. Just a two-digit decimal random number would split the writes to the key evenly across 100 different keys, allowing those keys to be distributed to different partitions.

However, having split the writes across different keys, any reads now have to do additional work, as they have to read the data from all 100 keys and combine it. This technique also requires additional bookkeeping: it only makes sense to append the random number for the small number of hot keys; for the vast majority of keys with low write throughput this would be unnecessary overhead. Thus, you also need some way of keeping track of which keys are being split.

Perhaps in the future, data systems will be able to automatically detect and compensate for skewed workloads; but for now, you need to think through the trade-offs for your own application.

Partitioning and Secondary Indexes

The partitioning schemes we have discussed so far rely on a key-value data model. If records are only ever accessed via their primary key, we can determine the partition from that key and use it to route read and write requests to the partition responsible for that key.

The situation becomes more complicated if secondary indexes are involved (see also "Other Indexing Structures" on page 85). A secondary index usually doesn't identify a record uniquely but rather is a way of searching for occurrences of a particular value: find all actions by user `123`, find all articles containing the word `hogwash`, find all cars whose color is `red`, and so on.

Secondary indexes are the bread and butter of relational databases, and they are common in document databases too. Many key-value stores (such as HBase and Voldemort) have avoided secondary indexes because of their added implementation complexity, but some (such as Riak) have started adding them because they are so useful for data modeling. And finally, secondary indexes are the *raison d'être* of search servers such as Solr and Elasticsearch.

The problem with secondary indexes is that they don't map neatly to partitions. There are two main approaches to partitioning a database with secondary indexes: document-based partitioning and term-based partitioning.

Partitioning Secondary Indexes by Document

For example, imagine you are operating a website for selling used cars (illustrated in Figure 6-4). Each listing has a unique ID—call it the *document ID*—and you partition the database by the document ID (for example, IDs 0 to 499 in partition 0, IDs 500 to 999 in partition 1, etc.).

You want to let users search for cars, allowing them to filter by color and by make, so you need a secondary index on `color` and `make` (in a document database these would be fields; in a relational database they would be columns). If you have declared the index, the database can perform the indexing automatically.[ii] For example, whenever a red car is added to the database, the database partition automatically adds it to the list of document IDs for the index entry `color:red`.

ii. If your database only supports a key-value model, you might be tempted to implement a secondary index yourself by creating a mapping from values to document IDs in application code. If you go down this route, you need to take great care to ensure your indexes remain consistent with the underlying data. Race conditions and intermittent write failures (where some changes were saved but others weren't) can very easily cause the data to go out of sync—see "The need for multi-object transactions" on page 231.

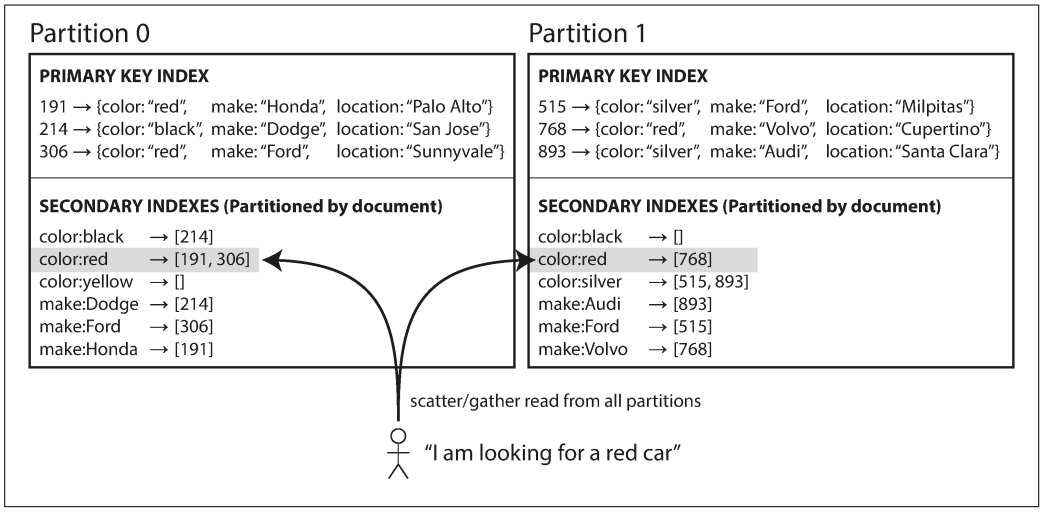


Figure 6-4. Partitioning secondary indexes by document.

In this indexing approach, each partition is completely separate: each partition maintains its own secondary indexes, covering only the documents in that partition. It doesn't care what data is stored in other partitions. Whenever you need to write to the database—to add, remove, or update a document—you only need to deal with the partition that contains the document ID that you are writing. For that reason, a document-partitioned index is also known as a *local index* (as opposed to a *global index*, described in the next section).

However, reading from a document-partitioned index requires care: unless you have done something special with the document IDs, there is no reason why all the cars with a particular color or a particular make would be in the same partition. In Figure 6-4, red cars appear in both partition 0 and partition 1. Thus, if you want to search for red cars, you need to send the query to *all* partitions, and combine all the results you get back.

This approach to querying a partitioned database is sometimes known as *scatter/gather*, and it can make read queries on secondary indexes quite expensive. Even if you query the partitions in parallel, scatter/gather is prone to tail latency amplification (see "Percentiles in Practice" on page 16). Nevertheless, it is widely used: MongoDB, Riak [15], Cassandra [16], Elasticsearch [17], SolrCloud [18], and VoltDB [19] all use document-partitioned secondary indexes. Most database vendors recommend that you structure your partitioning scheme so that secondary index queries can be served from a single partition, but that is not always possible, especially when you're using multiple secondary indexes in a single query (such as filtering cars by color and by make at the same time).

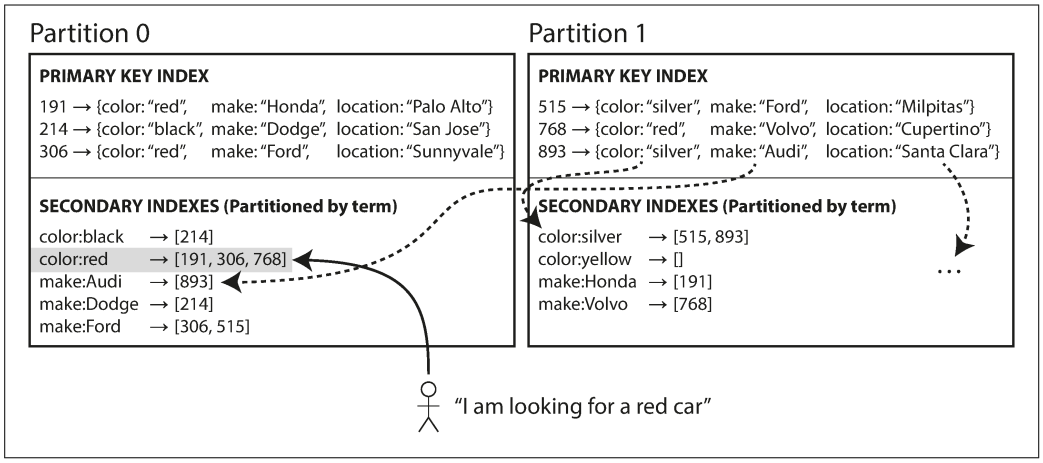


Figure 6-5. Partitioning secondary indexes by term.

Partitioning Secondary Indexes by Term

Rather than each partition having its own secondary index (a *local index*), we can construct a *global index* that covers data in all partitions. However, we can't just store that index on one node, since it would likely become a bottleneck and defeat the purpose of partitioning. A global index must also be partitioned, but it can be partitioned differently from the primary key index.

Figure 6-5 illustrates what this could look like: red cars from all partitions appear under `color:red` in the index, but the index is partitioned so that colors starting with the letters *a* to *r* appear in partition 0 and colors starting with *s* to *z* appear in partition 1. The index on the make of car is partitioned similarly (with the partition boundary being between *f* and *h*).

We call this kind of index *term-partitioned*, because the term we're looking for determines the partition of the index. Here, a term would be `color:red`, for example. The name *term* comes from full-text indexes (a particular kind of secondary index), where the terms are all the words that occur in a document.

As before, we can partition the index by the term itself, or using a hash of the term. Partitioning by the term itself can be useful for range scans (e.g., on a numeric property, such as the asking price of the car), whereas partitioning on a hash of the term gives a more even distribution of load.

The advantage of a global (term-partitioned) index over a document-partitioned index is that it can make reads more efficient: rather than doing scatter/gather over all partitions, a client only needs to make a request to the partition containing the term that it wants. However, the downside of a global index is that writes are slower and more complicated, because a write to a single document may now affect multiple

partitions of the index (every term in the document might be on a different partition, on a different node).

In an ideal world, the index would always be up to date, and every document written to the database would immediately be reflected in the index. However, in a term-partitioned index, that would require a distributed transaction across all partitions affected by a write, which is not supported in all databases (see Chapter 7 and Chapter 9).

In practice, updates to global secondary indexes are often asynchronous (that is, if you read the index shortly after a write, the change you just made may not yet be reflected in the index). For example, Amazon DynamoDB states that its global secondary indexes are updated within a fraction of a second in normal circumstances, but may experience longer propagation delays in cases of faults in the infrastructure [20].

Other uses of global term-partitioned indexes include Riak's search feature [21] and the Oracle data warehouse, which lets you choose between local and global indexing [22]. We will return to the topic of implementing term-partitioned secondary indexes in Chapter 12.

Rebalancing Partitions

Over time, things change in a database:

- The query throughput increases, so you want to add more CPUs to handle the load.
- The dataset size increases, so you want to add more disks and RAM to store it.
- A machine fails, and other machines need to take over the failed machine's responsibilities.

All of these changes call for data and requests to be moved from one node to another. The process of moving load from one node in the cluster to another is called *rebalancing*.

No matter which partitioning scheme is used, rebalancing is usually expected to meet some minimum requirements:

- After rebalancing, the load (data storage, read and write requests) should be shared fairly between the nodes in the cluster.
- While rebalancing is happening, the database should continue accepting reads and writes.
- No more data than necessary should be moved between nodes, to make rebalancing fast and to minimize the network and disk I/O load.

Strategies for Rebalancing

There are a few different ways of assigning partitions to nodes [23]. Let's briefly discuss each in turn.

How not to do it: hash mod N

When partitioning by the hash of a key, we said earlier (Figure 6-3) that it's best to divide the possible hashes into ranges and assign each range to a partition (e.g., assign *key* to partition 0 if $0 \le hash(key) < b_0$, to partition 1 if $b_0 \le hash(key) < b_1$, etc.).

Perhaps you wondered why we don't just use *mod* (the `%` operator in many programming languages). For example, *hash*(*key*) *mod* 10 would return a number between 0 and 9 (if we write the hash as a decimal number, the hash *mod* 10 would be the last digit). If we have 10 nodes, numbered 0 to 9, that seems like an easy way of assigning each key to a node.

The problem with the *mod N* approach is that if the number of nodes *N* changes, most of the keys will need to be moved from one node to another. For example, say *hash*(*key*) = 123456. If you initially have 10 nodes, that key starts out on node 6 (because 123456 *mod* 10 = 6). When you grow to 11 nodes, the key needs to move to node 3 (123456 *mod* 11 = 3), and when you grow to 12 nodes, it needs to move to node 0 (123456 *mod* 12 = 0). Such frequent moves make rebalancing excessively expensive.

We need an approach that doesn't move data around more than necessary.

Fixed number of partitions

Fortunately, there is a fairly simple solution: create many more partitions than there are nodes, and assign several partitions to each node. For example, a database running on a cluster of 10 nodes may be split into 1,000 partitions from the outset so that approximately 100 partitions are assigned to each node.

Now, if a node is added to the cluster, the new node can *steal* a few partitions from every existing node until partitions are fairly distributed once again. This process is illustrated in Figure 6-6. If a node is removed from the cluster, the same happens in reverse.

Only entire partitions are moved between nodes. The number of partitions does not change, nor does the assignment of keys to partitions. The only thing that changes is the assignment of partitions to nodes. This change of assignment is not immediate—it takes some time to transfer a large amount of data over the network—so the old assignment of partitions is used for any reads and writes that happen while the transfer is in progress.

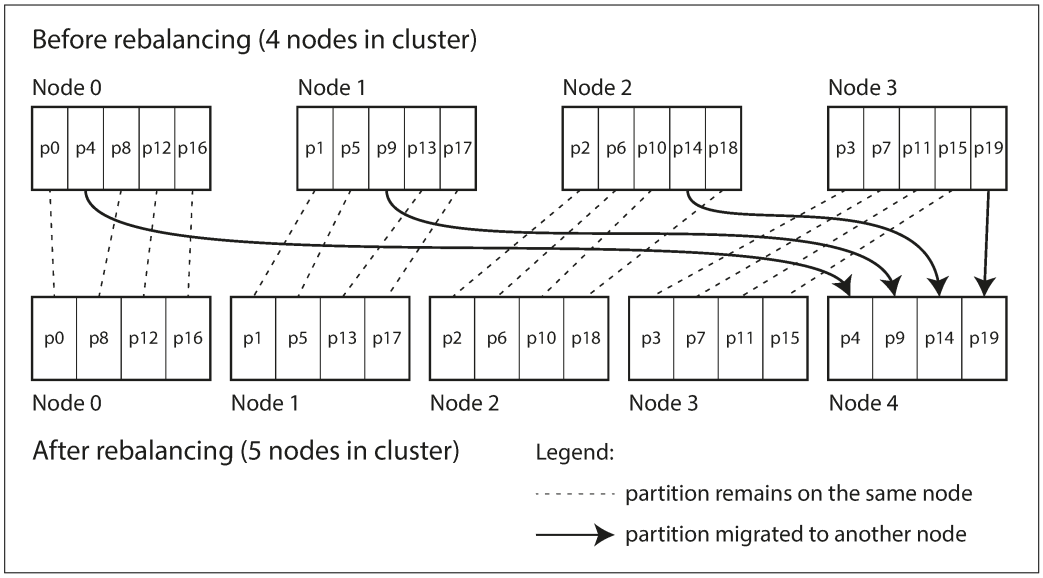


Figure 6-6. Adding a new node to a database cluster with multiple partitions per node.

In principle, you can even account for mismatched hardware in your cluster: by assigning more partitions to nodes that are more powerful, you can force those nodes to take a greater share of the load.

This approach to rebalancing is used in Riak [15], Elasticsearch [24], Couchbase [10], and Voldemort [25].

In this configuration, the number of partitions is usually fixed when the database is first set up and not changed afterward. Although in principle it's possible to split and merge partitions (see the next section), a fixed number of partitions is operationally simpler, and so many fixed-partition databases choose not to implement partition splitting. Thus, the number of partitions configured at the outset is the maximum number of nodes you can have, so you need to choose it high enough to accommodate future growth. However, each partition also has management overhead, so it's counterproductive to choose too high a number.

Choosing the right number of partitions is difficult if the total size of the dataset is highly variable (for example, if it starts small but may grow much larger over time). Since each partition contains a fixed fraction of the total data, the size of each partition grows proportionally to the total amount of data in the cluster. If partitions are very large, rebalancing and recovery from node failures become expensive. But if partitions are too small, they incur too much overhead. The best performance is achieved when the size of partitions is "just right," neither too big nor too small, which can be hard to achieve if the number of partitions is fixed but the dataset size varies.

Dynamic partitioning

For databases that use key range partitioning (see "Partitioning by Key Range" on page 202), a fixed number of partitions with fixed boundaries would be very inconvenient: if you got the boundaries wrong, you could end up with all of the data in one partition and all of the other partitions empty. Reconfiguring the partition boundaries manually would be very tedious.

For that reason, key range–partitioned databases such as HBase and RethinkDB create partitions dynamically. When a partition grows to exceed a configured size (on HBase, the default is 10 GB), it is split into two partitions so that approximately half of the data ends up on each side of the split [26]. Conversely, if lots of data is deleted and a partition shrinks below some threshold, it can be merged with an adjacent partition. This process is similar to what happens at the top level of a B-tree (see "B-Trees" on page 79).

Each partition is assigned to one node, and each node can handle multiple partitions, like in the case of a fixed number of partitions. After a large partition has been split, one of its two halves can be transferred to another node in order to balance the load. In the case of HBase, the transfer of partition files happens through HDFS, the underlying distributed filesystem [3].

An advantage of dynamic partitioning is that the number of partitions adapts to the total data volume. If there is only a small amount of data, a small number of partitions is sufficient, so overheads are small; if there is a huge amount of data, the size of each individual partition is limited to a configurable maximum [23].

However, a caveat is that an empty database starts off with a single partition, since there is no *a priori* information about where to draw the partition boundaries. While the dataset is small—until it hits the point at which the first partition is split—all writes have to be processed by a single node while the other nodes sit idle. To mitigate this issue, HBase and MongoDB allow an initial set of partitions to be configured on an empty database (this is called *pre-splitting*). In the case of key-range partitioning, pre-splitting requires that you already know what the key distribution is going to look like [4, 26].

Dynamic partitioning is not only suitable for key range–partitioned data, but can equally well be used with hash-partitioned data. MongoDB since version 2.4 supports both key-range and hash partitioning, and it splits partitions dynamically in either case.

Partitioning proportionally to nodes

With dynamic partitioning, the number of partitions is proportional to the size of the dataset, since the splitting and merging processes keep the size of each partition between some fixed minimum and maximum. On the other hand, with a fixed num-

ber of partitions, the size of each partition is proportional to the size of the dataset. In both of these cases, the number of partitions is independent of the number of nodes.

A third option, used by Cassandra and Ketama, is to make the number of partitions proportional to the number of nodes—in other words, to have a fixed number of partitions *per node* [23, 27, 28]. In this case, the size of each partition grows proportionally to the dataset size while the number of nodes remains unchanged, but when you increase the number of nodes, the partitions become smaller again. Since a larger data volume generally requires a larger number of nodes to store, this approach also keeps the size of each partition fairly stable.

When a new node joins the cluster, it randomly chooses a fixed number of existing partitions to split, and then takes ownership of one half of each of those split partitions while leaving the other half of each partition in place. The randomization can produce unfair splits, but when averaged over a larger number of partitions (in Cassandra, 256 partitions per node by default), the new node ends up taking a fair share of the load from the existing nodes. Cassandra 3.0 introduced an alternative rebalancing algorithm that avoids unfair splits [29].

Picking partition boundaries randomly requires that hash-based partitioning is used (so the boundaries can be picked from the range of numbers produced by the hash function). Indeed, this approach corresponds most closely to the original definition of consistent hashing [7] (see "Consistent Hashing" on page 204). Newer hash functions can achieve a similar effect with lower metadata overhead [8].

Operations: Automatic or Manual Rebalancing

There is one important question with regard to rebalancing that we have glossed over: does the rebalancing happen automatically or manually?

There is a gradient between fully automatic rebalancing (the system decides automatically when to move partitions from one node to another, without any administrator interaction) and fully manual (the assignment of partitions to nodes is explicitly configured by an administrator, and only changes when the administrator explicitly reconfigures it). For example, Couchbase, Riak, and Voldemort generate a suggested partition assignment automatically, but require an administrator to commit it before it takes effect.

Fully automated rebalancing can be convenient, because there is less operational work to do for normal maintenance. However, it can be unpredictable. Rebalancing is an expensive operation, because it requires rerouting requests and moving a large amount of data from one node to another. If it is not done carefully, this process can overload the network or the nodes and harm the performance of other requests while the rebalancing is in progress.

Such automation can be dangerous in combination with automatic failure detection. For example, say one node is overloaded and is temporarily slow to respond to requests. The other nodes conclude that the overloaded node is dead, and automatically rebalance the cluster to move load away from it. This puts additional load on the overloaded node, other nodes, and the network—making the situation worse and potentially causing a cascading failure.

For that reason, it can be a good thing to have a human in the loop for rebalancing. It's slower than a fully automatic process, but it can help prevent operational surprises.

Request Routing

We have now partitioned our dataset across multiple nodes running on multiple machines. But there remains an open question: when a client wants to make a request, how does it know which node to connect to? As partitions are rebalanced, the assignment of partitions to nodes changes. Somebody needs to stay on top of those changes in order to answer the question: if I want to read or write the key "foo", which IP address and port number do I need to connect to?

This is an instance of a more general problem called *service discovery*, which isn't limited to just databases. Any piece of software that is accessible over a network has this problem, especially if it is aiming for high availability (running in a redundant configuration on multiple machines). Many companies have written their own in-house service discovery tools, and many of these have been released as open source [30].

On a high level, there are a few different approaches to this problem (illustrated in Figure 6-7):

1. Allow clients to contact any node (e.g., via a round-robin load balancer). If that node coincidentally owns the partition to which the request applies, it can handle the request directly; otherwise, it forwards the request to the appropriate node, receives the reply, and passes the reply along to the client.
2. Send all requests from clients to a routing tier first, which determines the node that should handle each request and forwards it accordingly. This routing tier does not itself handle any requests; it only acts as a partition-aware load balancer.
3. Require that clients be aware of the partitioning and the assignment of partitions to nodes. In this case, a client can connect directly to the appropriate node, without any intermediary.

In all cases, the key problem is: how does the component making the routing decision (which may be one of the nodes, or the routing tier, or the client) learn about changes in the assignment of partitions to nodes?

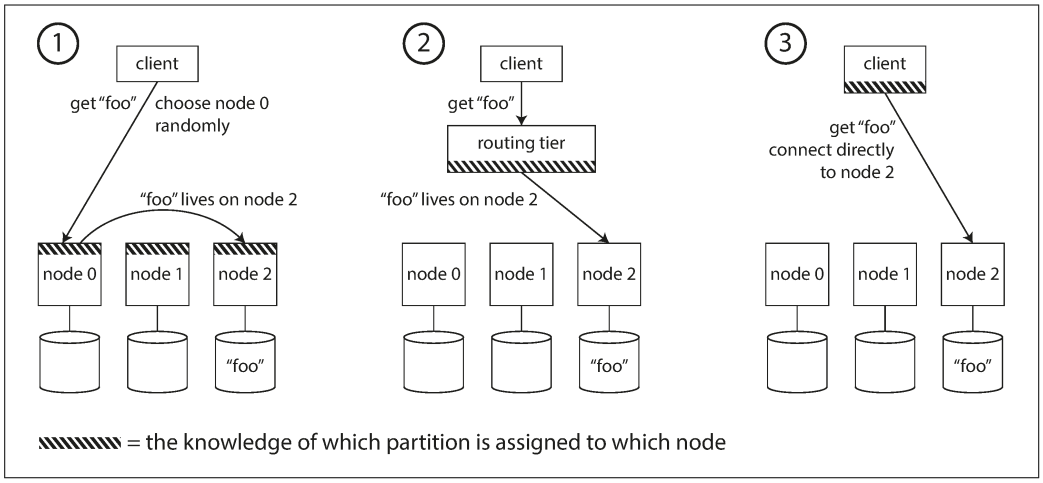


Figure 6-7. Three different ways of routing a request to the right node.

This is a challenging problem, because it is important that all participants agree—otherwise requests would be sent to the wrong nodes and not handled correctly. There are protocols for achieving consensus in a distributed system, but they are hard to implement correctly (see Chapter 9).

Many distributed data systems rely on a separate coordination service such as ZooKeeper to keep track of this cluster metadata, as illustrated in Figure 6-8. Each node registers itself in ZooKeeper, and ZooKeeper maintains the authoritative mapping of partitions to nodes. Other actors, such as the routing tier or the partitioning-aware client, can subscribe to this information in ZooKeeper. Whenever a partition changes ownership, or a node is added or removed, ZooKeeper notifies the routing tier so that it can keep its routing information up to date.

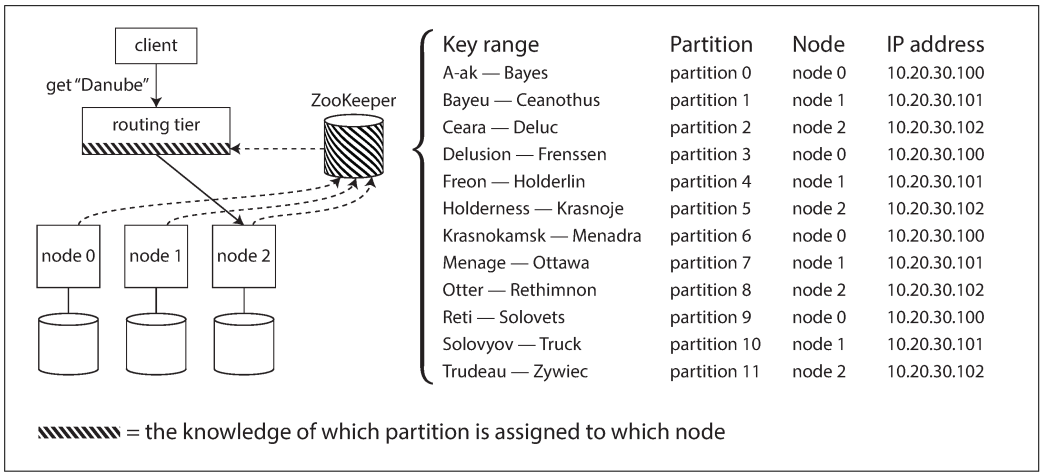


Figure 6-8. Using ZooKeeper to keep track of assignment of partitions to nodes.

For example, LinkedIn's Espresso uses Helix [31] for cluster management (which in turn relies on ZooKeeper), implementing a routing tier as shown in Figure 6-8. HBase, SolrCloud, and Kafka also use ZooKeeper to track partition assignment. MongoDB has a similar architecture, but it relies on its own *config server* implementation and *mongos* daemons as the routing tier.

Cassandra and Riak take a different approach: they use a *gossip protocol* among the nodes to disseminate any changes in cluster state. Requests can be sent to any node, and that node forwards them to the appropriate node for the requested partition (approach 1 in Figure 6-7). This model puts more complexity in the database nodes but avoids the dependency on an external coordination service such as ZooKeeper.

Couchbase does not rebalance automatically, which simplifies the design. Normally it is configured with a routing tier called *moxi*, which learns about routing changes from the cluster nodes [32].

When using a routing tier or when sending requests to a random node, clients still need to find the IP addresses to connect to. These are not as fast-changing as the assignment of partitions to nodes, so it is often sufficient to use DNS for this purpose.

Parallel Query Execution

So far we have focused on very simple queries that read or write a single key (plus scatter/gather queries in the case of document-partitioned secondary indexes). This is about the level of access supported by most NoSQL distributed datastores.

However, *massively parallel processing* (MPP) relational database products, often used for analytics, are much more sophisticated in the types of queries they support. A typical data warehouse query contains several join, filtering, grouping, and aggregation operations. The MPP query optimizer breaks this complex query into a number of execution stages and partitions, many of which can be executed in parallel on different nodes of the database cluster. Queries that involve scanning over large parts of the dataset particularly benefit from such parallel execution.

Fast parallel execution of data warehouse queries is a specialized topic, and given the business importance of analytics, it receives a lot of commercial interest. We will discuss some techniques for parallel query execution in Chapter 10. For a more detailed overview of techniques used in parallel databases, please see the references [1, 33].

Summary

In this chapter we explored different ways of partitioning a large dataset into smaller subsets. Partitioning is necessary when you have so much data that storing and processing it on a single machine is no longer feasible.

The goal of partitioning is to spread the data and query load evenly across multiple machines, avoiding hot spots (nodes with disproportionately high load). This requires choosing a partitioning scheme that is appropriate to your data, and rebalancing the partitions when nodes are added to or removed from the cluster.

We discussed two main approaches to partitioning:

- *Key range partitioning*, where keys are sorted, and a partition owns all the keys from some minimum up to some maximum. Sorting has the advantage that efficient range queries are possible, but there is a risk of hot spots if the application often accesses keys that are close together in the sorted order.

  In this approach, partitions are typically rebalanced dynamically by splitting the range into two subranges when a partition gets too big.
- *Hash partitioning*, where a hash function is applied to each key, and a partition owns a range of hashes. This method destroys the ordering of keys, making range queries inefficient, but may distribute load more evenly.

  When partitioning by hash, it is common to create a fixed number of partitions in advance, to assign several partitions to each node, and to move entire partitions from one node to another when nodes are added or removed. Dynamic partitioning can also be used.

Hybrid approaches are also possible, for example with a compound key: using one part of the key to identify the partition and another part for the sort order.

We also discussed the interaction between partitioning and secondary indexes. A secondary index also needs to be partitioned, and there are two methods:

- *Document-partitioned indexes* (local indexes), where the secondary indexes are stored in the same partition as the primary key and value. This means that only a single partition needs to be updated on write, but a read of the secondary index requires a scatter/gather across all partitions.
- *Term-partitioned indexes* (global indexes), where the secondary indexes are partitioned separately, using the indexed values. An entry in the secondary index may include records from all partitions of the primary key. When a document is written, several partitions of the secondary index need to be updated; however, a read can be served from a single partition.

Finally, we discussed techniques for routing queries to the appropriate partition, which range from simple partition-aware load balancing to sophisticated parallel query execution engines.

By design, every partition operates mostly independently—that's what allows a partitioned database to scale to multiple machines. However, operations that need to write

to several partitions can be difficult to reason about: for example, what happens if the write to one partition succeeds, but another fails? We will address that question in the following chapters.

References

[1] David J. DeWitt and Jim N. Gray: "Parallel Database Systems: The Future of High Performance Database Systems," *Communications of the ACM*, volume 35, number 6, pages 85–98, June 1992. doi:10.1145/129888.129894

[2] Lars George: "HBase vs. BigTable Comparison," *larsgeorge.com*, November 2009.

[3] "The Apache HBase Reference Guide," Apache Software Foundation, *hbase.apache.org*, 2014.

[4] MongoDB, Inc.: "New Hash-Based Sharding Feature in MongoDB 2.4," *blog.mongodb.org*, April 10, 2013.

[5] Ikai Lan: "App Engine Datastore Tip: Monotonically Increasing Values Are Bad," *ikaisays.com*, January 25, 2011.

[6] Martin Kleppmann: "Java's hashCode Is Not Safe for Distributed Systems," *martin.kleppmann.com*, June 18, 2012.

[7] David Karger, Eric Lehman, Tom Leighton, et al.: "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," at *29th Annual ACM Symposium on Theory of Computing* (STOC), pages 654–663, 1997. doi:10.1145/258533.258660

[8] John Lamping and Eric Veach: "A Fast, Minimal Memory, Consistent Hash Algorithm," *arxiv.org*, June 2014.

[9] Eric Redmond: "A Little Riak Book," Version 1.4.0, Basho Technologies, September 2013.

[10] "Couchbase 2.5 Administrator Guide," Couchbase, Inc., 2014.

[11] Avinash Lakshman and Prashant Malik: "Cassandra – A Decentralized Structured Storage System," at *3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware* (LADIS), October 2009.

[12] Jonathan Ellis: "Facebook's Cassandra Paper, Annotated and Compared to Apache Cassandra 2.0," *datastax.com*, September 12, 2013.

[13] "Introduction to Cassandra Query Language," DataStax, Inc., 2014.

[14] Samuel Axon: "3% of Twitter's Servers Dedicated to Justin Bieber," *mashable.com*, September 7, 2010.

[15] "Riak 1.4.8 Docs," Basho Technologies, Inc., 2014.

[16] Richard Low: "The Sweet Spot for Cassandra Secondary Indexing," *wentnet.com*, October 21, 2013.

[17] Zachary Tong: "Customizing Your Document Routing," *elasticsearch.org*, June 3, 2013.

[18] "Apache Solr Reference Guide," Apache Software Foundation, 2014.

[19] Andrew Pavlo: "H-Store Frequently Asked Questions," *hstore.cs.brown.edu*, October 2013.

[20] "Amazon DynamoDB Developer Guide," Amazon Web Services, Inc., 2014.

[21] Rusty Klophaus: "Difference Between 2I and Search," email to *riak-users* mailing list, *lists.basho.com*, October 25, 2011.

[22] Donald K. Burleson: "Object Partitioning in Oracle," *dba-oracle.com*, November 8, 2000.

[23] Eric Evans: "Rethinking Topology in Cassandra," at *ApacheCon Europe*, November 2012.

[24] Rafał Kuć: "Reroute API Explained," *elasticsearchserverbook.com*, September 30, 2013.

[25] "Project Voldemort Documentation," *project-voldemort.com*.

[26] Enis Soztutar: "Apache HBase Region Splitting and Merging," *hortonworks.com*, February 1, 2013.

[27] Brandon Williams: "Virtual Nodes in Cassandra 1.2," *datastax.com*, December 4, 2012.

[28] Richard Jones: "libketama: Consistent Hashing Library for Memcached Clients," *metabrew.com*, April 10, 2007.

[29] Branimir Lambov: "New Token Allocation Algorithm in Cassandra 3.0," *datastax.com*, January 28, 2016.

[30] Jason Wilder: "Open-Source Service Discovery," *jasonwilder.com*, February 2014.

[31] Kishore Gopalakrishna, Shi Lu, Zhen Zhang, et al.: "Untangling Cluster Management with Helix," at *ACM Symposium on Cloud Computing* (SoCC), October 2012. doi:10.1145/2391229.2391248

[32] "Moxi 1.8 Manual," Couchbase, Inc., 2014.

[33] Shivnath Babu and Herodotos Herodotou: "Massively Parallel Databases and MapReduce Systems," *Foundations and Trends in Databases*, volume 5, number 1, pages 1–104, November 2013. doi:10.1561/1900000036


PESSIMISTIC PENINSULA
No Phantoms
Actual serial execution
Two-phase locking
SERIALIZABLE
OPTIMISTIC PENINSULA
Serializable snapshot isolation
No Write Skew
SNAPSHOT ISOLATION
MVCC
No Read Skew
No Lost Updates
Row-level locking
No Dirty Writes
No Dirty Reads
READ COMMITTED
Temple of Multi-Object Atomicity
Atomic commit
To Distributed Transactions (Chapter 9)

CHAPTER 7
Transactions

> *Some authors have claimed that general two-phase commit is too expensive to support, because of the performance or availability problems that it brings. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.*
>
> —James Corbett et al., *Spanner: Google's Globally-Distributed Database* (2012)

In the harsh reality of data systems, many things can go wrong:

- The database software or hardware may fail at any time (including in the middle of a write operation).
- The application may crash at any time (including halfway through a series of operations).
- Interruptions in the network can unexpectedly cut off the application from the database, or one database node from another.
- Several clients may write to the database at the same time, overwriting each other's changes.
- A client may read data that doesn't make sense because it has only partially been updated.
- Race conditions between clients can cause surprising bugs.

In order to be reliable, a system has to deal with these faults and ensure that they don't cause catastrophic failure of the entire system. However, implementing fault-tolerance mechanisms is a lot of work. It requires a lot of careful thinking about all the things that can go wrong, and a lot of testing to ensure that the solution actually works.

For decades, *transactions* have been the mechanism of choice for simplifying these issues. A transaction is a way for an application to group several reads and writes together into a logical unit. Conceptually, all the reads and writes in a transaction are executed as one operation: either the entire transaction succeeds (*commit*) or it fails (*abort*, *rollback*). If it fails, the application can safely retry. With transactions, error handling becomes much simpler for an application, because it doesn't need to worry about partial failure—i.e., the case where some operations succeed and some fail (for whatever reason).

If you have spent years working with transactions, they may seem obvious, but we shouldn't take them for granted. Transactions are not a law of nature; they were created with a purpose, namely to *simplify the programming model* for applications accessing a database. By using transactions, the application is free to ignore certain potential error scenarios and concurrency issues, because the database takes care of them instead (we call these *safety guarantees*).

Not every application needs transactions, and sometimes there are advantages to weakening transactional guarantees or abandoning them entirely (for example, to achieve higher performance or higher availability). Some safety properties can be achieved without transactions.

How do you figure out whether you need transactions? In order to answer that question, we first need to understand exactly what safety guarantees transactions can provide, and what costs are associated with them. Although transactions seem straightforward at first glance, there are actually many subtle but important details that come into play.

In this chapter, we will examine many examples of things that can go wrong, and explore the algorithms that databases use to guard against those issues. We will go especially deep in the area of concurrency control, discussing various kinds of race conditions that can occur and how databases implement isolation levels such as *read committed*, *snapshot isolation*, and *serializability*.

This chapter applies to both single-node and distributed databases; in Chapter 8 we will focus the discussion on the particular challenges that arise only in distributed systems.

The Slippery Concept of a Transaction

Almost all relational databases today, and some nonrelational databases, support transactions. Most of them follow the style that was introduced in 1975 by IBM System R, the first SQL database [1, 2, 3]. Although some implementation details have changed, the general idea has remained virtually the same for 40 years: the transaction support in MySQL, PostgreSQL, Oracle, SQL Server, etc., is uncannily similar to that of System R.

In the late 2000s, nonrelational (NoSQL) databases started gaining popularity. They aimed to improve upon the relational status quo by offering a choice of new data models (see Chapter 2), and by including replication (Chapter 5) and partitioning (Chapter 6) by default. Transactions were the main casualty of this movement: many of this new generation of databases abandoned transactions entirely, or redefined the word to describe a much weaker set of guarantees than had previously been understood [4].

With the hype around this new crop of distributed databases, there emerged a popular belief that transactions were the antithesis of scalability, and that any large-scale system would have to abandon transactions in order to maintain good performance and high availability [5, 6]. On the other hand, transactional guarantees are sometimes presented by database vendors as an essential requirement for "serious applications" with "valuable data." Both viewpoints are pure hyperbole.

The truth is not that simple: like every other technical design choice, transactions have advantages and limitations. In order to understand those trade-offs, let's go into the details of the guarantees that transactions can provide—both in normal operation and in various extreme (but realistic) circumstances.

The Meaning of ACID

The safety guarantees provided by transactions are often described by the well-known acronym *ACID*, which stands for *Atomicity*, *Consistency*, *Isolation*, and *Durability*. It was coined in 1983 by Theo Härder and Andreas Reuter [7] in an effort to establish precise terminology for fault-tolerance mechanisms in databases.

However, in practice, one database's implementation of ACID does not equal another's implementation. For example, as we shall see, there is a lot of ambiguity around the meaning of *isolation* [8]. The high-level idea is sound, but the devil is in the details. Today, when a system claims to be "ACID compliant," it's unclear what guarantees you can actually expect. ACID has unfortunately become mostly a marketing term.

(Systems that do not meet the ACID criteria are sometimes called *BASE*, which stands for *Basically Available*, *Soft state*, and *Eventual consistency* [9]. This is even more vague than the definition of ACID. It seems that the only sensible definition of BASE is "not ACID"; i.e., it can mean almost anything you want.)

Let's dig into the definitions of atomicity, consistency, isolation, and durability, as this will let us refine our idea of transactions.

Atomicity

In general, *atomic* refers to something that cannot be broken down into smaller parts. The word means similar but subtly different things in different branches of comput-

ing. For example, in multi-threaded programming, if one thread executes an atomic operation, that means there is no way that another thread could see the half-finished result of the operation. The system can only be in the state it was before the operation or after the operation, not something in between.

By contrast, in the context of ACID, atomicity is *not* about concurrency. It does not describe what happens if several processes try to access the same data at the same time, because that is covered under the letter *I*, for *isolation* (see "Isolation" on page 225).

Rather, ACID atomicity describes what happens if a client wants to make several writes, but a fault occurs after some of the writes have been processed—for example, a process crashes, a network connection is interrupted, a disk becomes full, or some integrity constraint is violated. If the writes are grouped together into an atomic transaction, and the transaction cannot be completed (*committed*) due to a fault, then the transaction is *aborted* and the database must discard or undo any writes it has made so far in that transaction.

Without atomicity, if an error occurs partway through making multiple changes, it's difficult to know which changes have taken effect and which haven't. The application could try again, but that risks making the same change twice, leading to duplicate or incorrect data. Atomicity simplifies this problem: if a transaction was aborted, the application can be sure that it didn't change anything, so it can safely be retried.

The ability to abort a transaction on error and have all writes from that transaction discarded is the defining feature of ACID atomicity. Perhaps *abortability* would have been a better term than *atomicity*, but we will stick with *atomicity* since that's the usual word.

Consistency

The word *consistency* is terribly overloaded:

- In Chapter 5 we discussed *replica consistency* and the issue of *eventual consistency* that arises in asynchronously replicated systems (see "Problems with Replication Lag" on page 161).
- *Consistent hashing* is an approach to partitioning that some systems use for rebalancing (see "Consistent Hashing" on page 204).
- In the CAP theorem (see Chapter 9), the word *consistency* is used to mean *linearizability* (see "Linearizability" on page 324).
- In the context of ACID, *consistency* refers to an application-specific notion of the database being in a "good state."

It's unfortunate that the same word is used with at least four different meanings.

The idea of ACID consistency is that you have certain statements about your data (*invariants*) that must always be true—for example, in an accounting system, credits and debits across all accounts must always be balanced. If a transaction starts with a database that is valid according to these invariants, and any writes during the transaction preserve the validity, then you can be sure that the invariants are always satisfied.

However, this idea of consistency depends on the application's notion of invariants, and it's the application's responsibility to define its transactions correctly so that they preserve consistency. This is not something that the database can guarantee: if you write bad data that violates your invariants, the database can't stop you. (Some specific kinds of invariants can be checked by the database, for example using foreign key constraints or uniqueness constraints. However, in general, the application defines what data is valid or invalid—the database only stores it.)

Atomicity, isolation, and durability are properties of the database, whereas consistency (in the ACID sense) is a property of the application. The application may rely on the database's atomicity and isolation properties in order to achieve consistency, but it's not up to the database alone. Thus, the letter C doesn't really belong in ACID.[i]

Isolation

Most databases are accessed by several clients at the same time. That is no problem if they are reading and writing different parts of the database, but if they are accessing the same database records, you can run into concurrency problems (race conditions).

Figure 7-1 is a simple example of this kind of problem. Say you have two clients simultaneously incrementing a counter that is stored in a database. Each client needs to read the current value, add 1, and write the new value back (assuming there is no increment operation built into the database). In Figure 7-1 the counter should have increased from 42 to 44, because two increments happened, but it actually only went to 43 because of the race condition.

Isolation in the sense of ACID means that concurrently executing transactions are isolated from each other: they cannot step on each other's toes. The classic database textbooks formalize isolation as *serializability*, which means that each transaction can pretend that it is the only transaction running on the entire database. The database ensures that when the transactions have committed, the result is the same as if they had run *serially* (one after another), even though in reality they may have run concurrently [10].

i. Joe Hellerstein has remarked that the C in ACID was "tossed in to make the acronym work" in Härder and Reuter's paper [7], and that it wasn't considered important at the time.

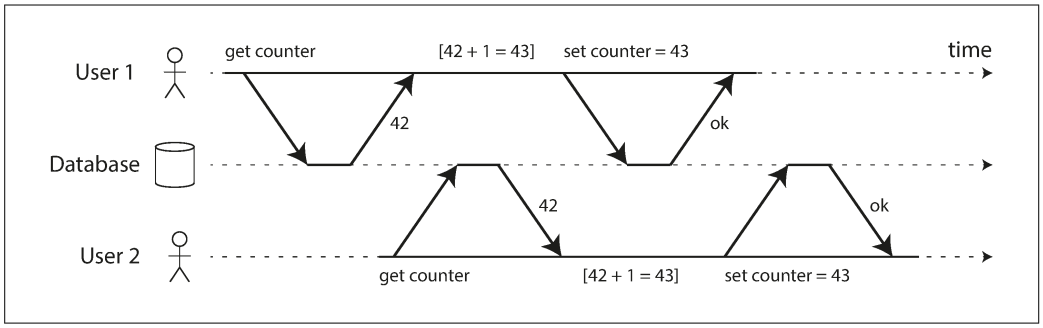


Figure 7-1. A race condition between two clients concurrently incrementing a counter.

However, in practice, serializable isolation is rarely used, because it carries a performance penalty. Some popular databases, such as Oracle 11g, don't even implement it. In Oracle there is an isolation level called "serializable," but it actually implements something called *snapshot isolation*, which is a weaker guarantee than serializability [8, 11]. We will explore snapshot isolation and other forms of isolation in "Weak Isolation Levels" on page 233.

Durability

The purpose of a database system is to provide a safe place where data can be stored without fear of losing it. *Durability* is the promise that once a transaction has committed successfully, any data it has written will not be forgotten, even if there is a hardware fault or the database crashes.

In a single-node database, durability typically means that the data has been written to nonvolatile storage such as a hard drive or SSD. It usually also involves a write-ahead log or similar (see "Making B-trees reliable" on page 82), which allows recovery in the event that the data structures on disk are corrupted. In a replicated database, durability may mean that the data has been successfully copied to some number of nodes. In order to provide a durability guarantee, a database must wait until these writes or replications are complete before reporting a transaction as successfully committed.

As discussed in "Reliability" on page 6, perfect durability does not exist: if all your hard disks and all your backups are destroyed at the same time, there's obviously nothing your database can do to save you.

Replication and Durability

Historically, durability meant writing to an archive tape. Then it was understood as writing to a disk or SSD. More recently, it has been adapted to mean replication. Which implementation is better?

The truth is, nothing is perfect:

- If you write to disk and the machine dies, even though your data isn't lost, it is inaccessible until you either fix the machine or transfer the disk to another machine. Replicated systems can remain available.
- A correlated fault—a power outage or a bug that crashes every node on a particular input—can knock out all replicas at once (see "Reliability" on page 6), losing any data that is only in memory. Writing to disk is therefore still relevant for in-memory databases.
- In an asynchronously replicated system, recent writes may be lost when the leader becomes unavailable (see "Handling Node Outages" on page 156).
- When the power is suddenly cut, SSDs in particular have been shown to sometimes violate the guarantees they are supposed to provide: even `fsync` isn't guaranteed to work correctly [12]. Disk firmware can have bugs, just like any other kind of software [13, 14].
- Subtle interactions between the storage engine and the filesystem implementation can lead to bugs that are hard to track down, and may cause files on disk to be corrupted after a crash [15, 16].
- Data on disk can gradually become corrupted without this being detected [17]. If data has been corrupted for some time, replicas and recent backups may also be corrupted. In this case, you will need to try to restore the data from a historical backup.
- One study of SSDs found that between 30% and 80% of drives develop at least one bad block during the first four years of operation [18]. Magnetic hard drives have a lower rate of bad sectors, but a higher rate of complete failure than SSDs.
- If an SSD is disconnected from power, it can start losing data within a few weeks, depending on the temperature [19].

In practice, there is no one technique that can provide absolute guarantees. There are only various risk-reduction techniques, including writing to disk, replicating to remote machines, and backups—and they can and should be used together. As always, it's wise to take any theoretical "guarantees" with a healthy grain of salt.

Single-Object and Multi-Object Operations

To recap, in ACID, atomicity and isolation describe what the database should do if a client makes several writes within the same transaction:

Atomicity
: If an error occurs halfway through a sequence of writes, the transaction should be aborted, and the writes made up to that point should be discarded. In other words, the database saves you from having to worry about partial failure, by giving an all-or-nothing guarantee.

Isolation
: Concurrently running transactions shouldn't interfere with each other. For example, if one transaction makes several writes, then another transaction should see either all or none of those writes, but not some subset.

These definitions assume that you want to modify several objects (rows, documents, records) at once. Such *multi-object transactions* are often needed if several pieces of data need to be kept in sync. Figure 7-2 shows an example from an email application. To display the number of unread messages for a user, you could query something like:

```
SELECT COUNT(*) FROM emails WHERE recipient_id = 2 AND unread_flag = true
```

However, you might find this query to be too slow if there are many emails, and decide to store the number of unread messages in a separate field (a kind of denormalization). Now, whenever a new message comes in, you have to increment the unread counter as well, and whenever a message is marked as read, you also have to decrement the unread counter.

In Figure 7-2, user 2 experiences an anomaly: the mailbox listing shows an unread message, but the counter shows zero unread messages because the counter increment has not yet happened.[ii] Isolation would have prevented this issue by ensuring that user 2 sees either both the inserted email and the updated counter, or neither, but not an inconsistent halfway point.

ii. Arguably, an incorrect counter in an email application is not a particularly critical problem. Alternatively, think of a customer account balance instead of an unread counter, and a payment transaction instead of an email.

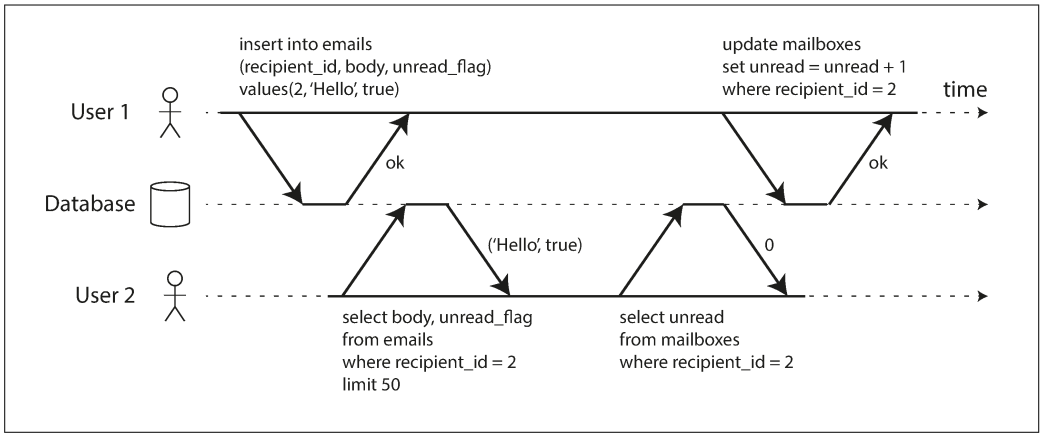


Figure 7-2. Violating isolation: one transaction reads another transaction's uncommitted writes (a "dirty read").

Figure 7-3 illustrates the need for atomicity: if an error occurs somewhere over the course of the transaction, the contents of the mailbox and the unread counter might become out of sync. In an atomic transaction, if the update to the counter fails, the transaction is aborted and the inserted email is rolled back.

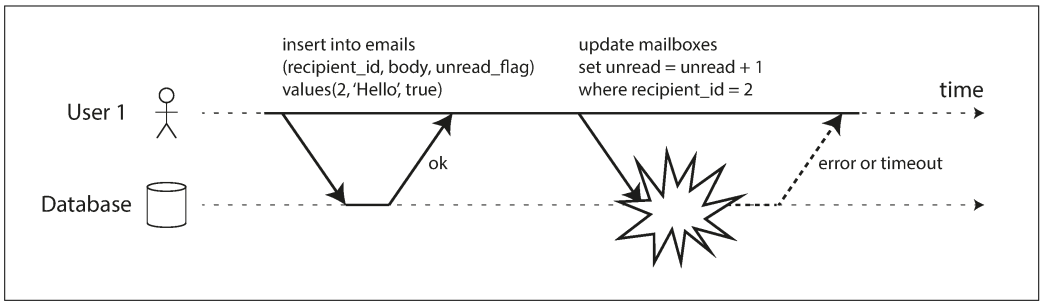


Figure 7-3. Atomicity ensures that if an error occurs any prior writes from that transaction are undone, to avoid an inconsistent state.

Multi-object transactions require some way of determining which read and write operations belong to the same transaction. In relational databases, that is typically done based on the client's TCP connection to the database server: on any particular connection, everything between a `BEGIN TRANSACTION` and a `COMMIT` statement is considered to be part of the same transaction.[iii]

iii. This is not ideal. If the TCP connection is interrupted, the transaction must be aborted. If the interruption happens after the client has requested a commit but before the server acknowledges that the commit happened, the client doesn't know whether the transaction was committed or not. To solve this issue, a transaction manager can group operations by a unique transaction identifier that is not bound to a particular TCP connection. We will return to this topic in "The End-to-End Argument for Databases" on page 516.

On the other hand, many nonrelational databases don't have such a way of grouping operations together. Even if there is a multi-object API (for example, a key-value store may have a *multi-put* operation that updates several keys in one operation), that doesn't necessarily mean it has transaction semantics: the command may succeed for some keys and fail for others, leaving the database in a partially updated state.

Single-object writes

Atomicity and isolation also apply when a single object is being changed. For example, imagine you are writing a 20 KB JSON document to a database:

- If the network connection is interrupted after the first 10 KB have been sent, does the database store that unparseable 10 KB fragment of JSON?
- If the power fails while the database is in the middle of overwriting the previous value on disk, do you end up with the old and new values spliced together?
- If another client reads that document while the write is in progress, will it see a partially updated value?

Those issues would be incredibly confusing, so storage engines almost universally aim to provide atomicity and isolation on the level of a single object (such as a key-value pair) on one node. Atomicity can be implemented using a log for crash recovery (see "Making B-trees reliable" on page 82), and isolation can be implemented using a lock on each object (allowing only one thread to access an object at any one time).

Some databases also provide more complex atomic operations,[iv] such as an increment operation, which removes the need for a read-modify-write cycle like that in Figure 7-1. Similarly popular is a compare-and-set operation, which allows a write to happen only if the value has not been concurrently changed by someone else (see "Compare-and-set" on page 245).

These single-object operations are useful, as they can prevent lost updates when several clients try to write to the same object concurrently (see "Preventing Lost Updates" on page 242). However, they are not transactions in the usual sense of the word. Compare-and-set and other single-object operations have been dubbed "lightweight transactions" or even "ACID" for marketing purposes [20, 21, 22], but that terminology is misleading. A transaction is usually understood as a mechanism for grouping multiple operations on multiple objects into one unit of execution.

iv. Strictly speaking, the term *atomic increment* uses the word *atomic* in the sense of multi-threaded programming. In the context of ACID, it should actually be called *isolated* or *serializable* increment. But that's getting nitpicky.

The need for multi-object transactions

Many distributed datastores have abandoned multi-object transactions because they are difficult to implement across partitions, and they can get in the way in some scenarios where very high availability or performance is required. However, there is nothing that fundamentally prevents transactions in a distributed database, and we will discuss implementations of distributed transactions in Chapter 9.

But do we need multi-object transactions at all? Would it be possible to implement any application with only a key-value data model and single-object operations?

There are some use cases in which single-object inserts, updates, and deletes are sufficient. However, in many other cases writes to several different objects need to be coordinated:

- In a relational data model, a row in one table often has a foreign key reference to a row in another table. (Similarly, in a graph-like data model, a vertex has edges to other vertices.) Multi-object transactions allow you to ensure that these references remain valid: when inserting several records that refer to one another, the foreign keys have to be correct and up to date, or the data becomes nonsensical.
- In a document data model, the fields that need to be updated together are often within the same document, which is treated as a single object—no multi-object transactions are needed when updating a single document. However, document databases lacking join functionality also encourage denormalization (see "Relational Versus Document Databases Today" on page 38). When denormalized information needs to be updated, like in the example of Figure 7-2, you need to update several documents in one go. Transactions are very useful in this situation to prevent denormalized data from going out of sync.
- In databases with secondary indexes (almost everything except pure key-value stores), the indexes also need to be updated every time you change a value. These indexes are different database objects from a transaction point of view: for example, without transaction isolation, it's possible for a record to appear in one index but not another, because the update to the second index hasn't happened yet.

Such applications can still be implemented without transactions. However, error handling becomes much more complicated without atomicity, and the lack of isolation can cause concurrency problems. We will discuss those in "Weak Isolation Levels" on page 233, and explore alternative approaches in Chapter 12.

Handling errors and aborts

A key feature of a transaction is that it can be aborted and safely retried if an error occurred. ACID databases are based on this philosophy: if the database is in danger

of violating its guarantee of atomicity, isolation, or durability, it would rather abandon the transaction entirely than allow it to remain half-finished.

Not all systems follow that philosophy, though. In particular, datastores with leaderless replication (see "Leaderless Replication" on page 177) work much more on a "best effort" basis, which could be summarized as "the database will do as much as it can, and if it runs into an error, it won't undo something it has already done"—so it's the application's responsibility to recover from errors.

Errors will inevitably happen, but many software developers prefer to think only about the happy path rather than the intricacies of error handling. For example, popular object-relational mapping (ORM) frameworks such as Rails's ActiveRecord and Django don't retry aborted transactions—the error usually results in an exception bubbling up the stack, so any user input is thrown away and the user gets an error message. This is a shame, because the whole point of aborts is to enable safe retries.

Although retrying an aborted transaction is a simple and effective error handling mechanism, it isn't perfect:

- If the transaction actually succeeded, but the network failed while the server tried to acknowledge the successful commit to the client (so the client thinks it failed), then retrying the transaction causes it to be performed twice—unless you have an additional application-level deduplication mechanism in place.
- If the error is due to overload, retrying the transaction will make the problem worse, not better. To avoid such feedback cycles, you can limit the number of retries, use exponential backoff, and handle overload-related errors differently from other errors (if possible).
- It is only worth retrying after transient errors (for example due to deadlock, isolation violation, temporary network interruptions, and failover); after a permanent error (e.g., constraint violation) a retry would be pointless.
- If the transaction also has side effects outside of the database, those side effects may happen even if the transaction is aborted. For example, if you're sending an email, you wouldn't want to send the email again every time you retry the transaction. If you want to make sure that several different systems either commit or abort together, two-phase commit can help (we will discuss this in "Atomic Commit and Two-Phase Commit (2PC)" on page 354).
- If the client process fails while retrying, any data it was trying to write to the database is lost.

Weak Isolation Levels

If two transactions don't touch the same data, they can safely be run in parallel, because neither depends on the other. Concurrency issues (race conditions) only come into play when one transaction reads data that is concurrently modified by another transaction, or when two transactions try to simultaneously modify the same data.

Concurrency bugs are hard to find by testing, because such bugs are only triggered when you get unlucky with the timing. Such timing issues might occur very rarely, and are usually difficult to reproduce. Concurrency is also very difficult to reason about, especially in a large application where you don't necessarily know which other pieces of code are accessing the database. Application development is difficult enough if you just have one user at a time; having many concurrent users makes it much harder still, because any piece of data could unexpectedly change at any time.

For that reason, databases have long tried to hide concurrency issues from application developers by providing *transaction isolation*. In theory, isolation should make your life easier by letting you pretend that no concurrency is happening: *serializable* isolation means that the database guarantees that transactions have the same effect as if they ran *serially* (i.e., one at a time, without any concurrency).

In practice, isolation is unfortunately not that simple. Serializable isolation has a performance cost, and many databases don't want to pay that price [8]. It's therefore common for systems to use weaker levels of isolation, which protect against *some* concurrency issues, but not all. Those levels of isolation are much harder to understand, and they can lead to subtle bugs, but they are nevertheless used in practice [23].

Concurrency bugs caused by weak transaction isolation are not just a theoretical problem. They have caused substantial loss of money [24, 25], led to investigation by financial auditors [26], and caused customer data to be corrupted [27]. A popular comment on revelations of such problems is "Use an ACID database if you're handling financial data!"—but that misses the point. Even many popular relational database systems (which are usually considered "ACID") use weak isolation, so they wouldn't necessarily have prevented these bugs from occurring.

Rather than blindly relying on tools, we need to develop a good understanding of the kinds of concurrency problems that exist, and how to prevent them. Then we can build applications that are reliable and correct, using the tools at our disposal.

In this section we will look at several weak (nonserializable) isolation levels that are used in practice, and discuss in detail what kinds of race conditions can and cannot occur, so that you can decide what level is appropriate to your application. Once we've done that, we will discuss serializability in detail (see "Serializability" on page

251). Our discussion of isolation levels will be informal, using examples. If you want rigorous definitions and analyses of their properties, you can find them in the academic literature [28, 29, 30].

Read Committed

The most basic level of transaction isolation is *read committed*.[v] It makes two guarantees:

1. When reading from the database, you will only see data that has been committed (no *dirty reads*).
2. When writing to the database, you will only overwrite data that has been committed (no *dirty writes*).

Let's discuss these two guarantees in more detail.

No dirty reads

Imagine a transaction has written some data to the database, but the transaction has not yet committed or aborted. Can another transaction see that uncommitted data? If yes, that is called a *dirty read* [2].

Transactions running at the read committed isolation level must prevent dirty reads. This means that any writes by a transaction only become visible to others when that transaction commits (and then all of its writes become visible at once). This is illustrated in Figure 7-4, where user 1 has set $x = 3$, but user 2's *get x* still returns the old value, 2, while user 1 has not yet committed.

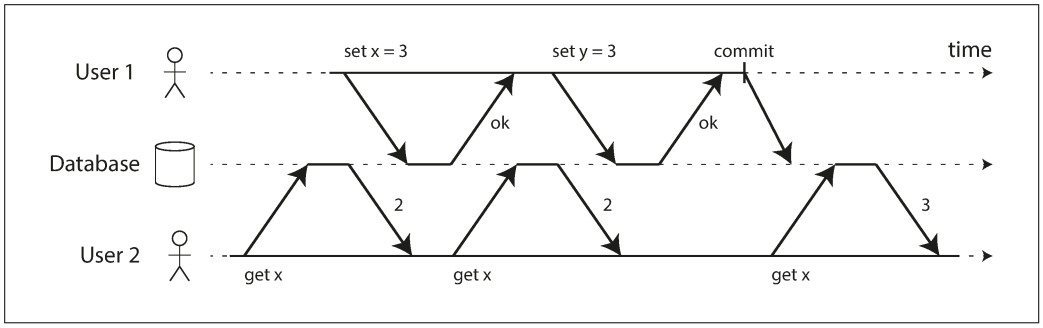


Figure 7-4. No dirty reads: user 2 sees the new value for x only after user 1's transaction has committed.

v. Some databases support an even weaker isolation level called *read uncommitted*. It prevents dirty writes, but does not prevent dirty reads.

There are a few reasons why it's useful to prevent dirty reads:

- If a transaction needs to update several objects, a dirty read means that another transaction may see some of the updates but not others. For example, in Figure 7-2, the user sees the new unread email but not the updated counter. This is a dirty read of the email. Seeing the database in a partially updated state is confusing to users and may cause other transactions to take incorrect decisions.
- If a transaction aborts, any writes it has made need to be rolled back (like in Figure 7-3). If the database allows dirty reads, that means a transaction may see data that is later rolled back—i.e., which is never actually committed to the database. Reasoning about the consequences quickly becomes mind-bending.

No dirty writes

What happens if two transactions concurrently try to update the same object in a database? We don't know in which order the writes will happen, but we normally assume that the later write overwrites the earlier write.

However, what happens if the earlier write is part of a transaction that has not yet committed, so the later write overwrites an uncommitted value? This is called a *dirty write* [28]. Transactions running at the read committed isolation level must prevent dirty writes, usually by delaying the second write until the first write's transaction has committed or aborted.

By preventing dirty writes, this isolation level avoids some kinds of concurrency problems:

- If transactions update multiple objects, dirty writes can lead to a bad outcome. For example, consider Figure 7-5, which illustrates a used car sales website on which two people, Alice and Bob, are simultaneously trying to buy the same car. Buying a car requires two database writes: the listing on the website needs to be updated to reflect the buyer, and the sales invoice needs to be sent to the buyer. In the case of Figure 7-5, the sale is awarded to Bob (because he performs the winning update to the `listings` table), but the invoice is sent to Alice (because she performs the winning update to the `invoices` table). Read committed prevents such mishaps.
- However, read committed does *not* prevent the race condition between two counter increments in Figure 7-1. In this case, the second write happens after the first transaction has committed, so it's not a dirty write. It's still incorrect, but for a different reason—in "Preventing Lost Updates" on page 242 we will discuss how to make such counter increments safe.

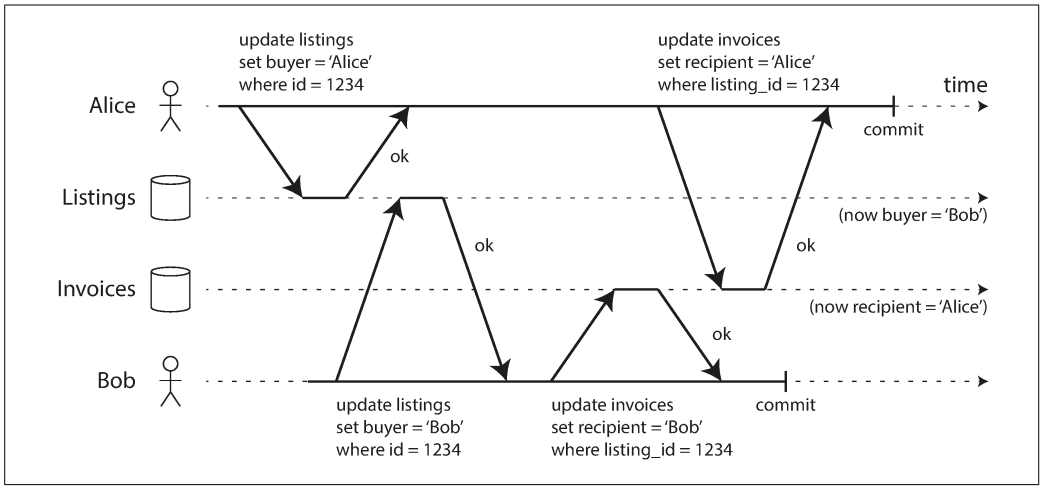


Figure 7-5. With dirty writes, conflicting writes from different transactions can be mixed up.

Implementing read committed

Read committed is a very popular isolation level. It is the default setting in Oracle 11g, PostgreSQL, SQL Server 2012, MemSQL, and many other databases [8].

Most commonly, databases prevent dirty writes by using row-level locks: when a transaction wants to modify a particular object (row or document), it must first acquire a lock on that object. It must then hold that lock until the transaction is committed or aborted. Only one transaction can hold the lock for any given object; if another transaction wants to write to the same object, it must wait until the first transaction is committed or aborted before it can acquire the lock and continue. This locking is done automatically by databases in read committed mode (or stronger isolation levels).

How do we prevent dirty reads? One option would be to use the same lock, and to require any transaction that wants to read an object to briefly acquire the lock and then release it again immediately after reading. This would ensure that a read couldn't happen while an object has a dirty, uncommitted value (because during that time the lock would be held by the transaction that has made the write).

However, the approach of requiring read locks does not work well in practice, because one long-running write transaction can force many read-only transactions to wait until the long-running transaction has completed. This harms the response time of read-only transactions and is bad for operability: a slowdown in one part of an application can have a knock-on effect in a completely different part of the application, due to waiting for locks.

For that reason, most databases[vi] prevent dirty reads using the approach illustrated in Figure 7-4: for every object that is written, the database remembers both the old committed value and the new value set by the transaction that currently holds the write lock. While the transaction is ongoing, any other transactions that read the object are simply given the old value. Only when the new value is committed do transactions switch over to reading the new value.

Snapshot Isolation and Repeatable Read

If you look superficially at read committed isolation, you could be forgiven for thinking that it does everything that a transaction needs to do: it allows aborts (required for atomicity), it prevents reading the incomplete results of transactions, and it prevents concurrent writes from getting intermingled. Indeed, those are useful features, and much stronger guarantees than you can get from a system that has no transactions.

However, there are still plenty of ways in which you can have concurrency bugs when using this isolation level. For example, Figure 7-6 illustrates a problem that can occur with read committed.

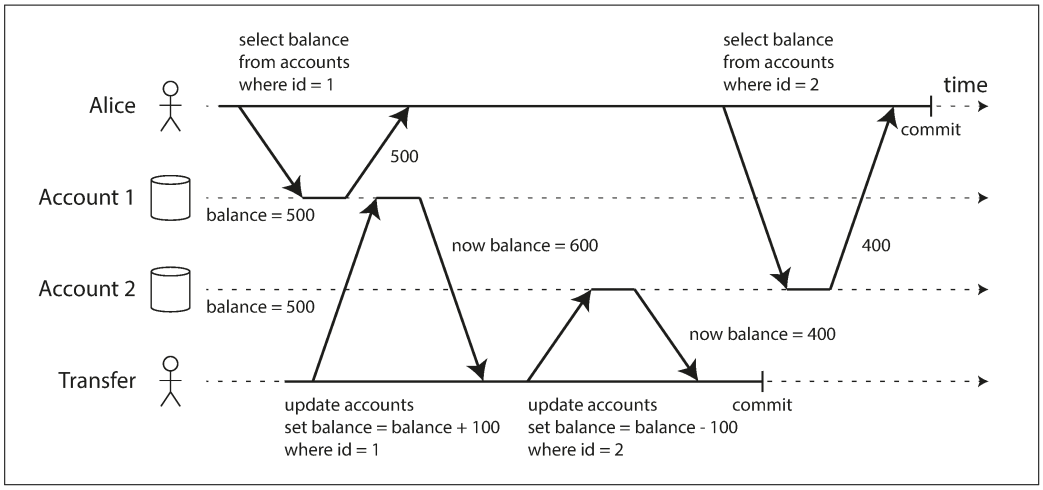


Figure 7-6. Read skew: Alice observes the database in an inconsistent state.

Say Alice has $1,000 of savings at a bank, split across two accounts with $500 each. Now a transaction transfers $100 from one of her accounts to the other. If she is unlucky enough to look at her list of account balances in the same moment as that transaction is being processed, she may see one account balance at a time before the

vi. At the time of writing, the only mainstream databases that use locks for read committed isolation are IBM DB2 and Microsoft SQL Server in the `read_committed_snapshot=off` configuration [23, 36].

incoming payment has arrived (with a balance of $500), and the other account after the outgoing transfer has been made (the new balance being $400). To Alice it now appears as though she only has a total of $900 in her accounts—it seems that $100 has vanished into thin air.

This anomaly is called a *nonrepeatable read* or *read skew*: if Alice were to read the balance of account 1 again at the end of the transaction, she would see a different value ($600) than she saw in her previous query. Read skew is considered acceptable under read committed isolation: the account balances that Alice saw were indeed committed at the time when she read them.

The term *skew* is unfortunately overloaded: we previously used it in the sense of an *unbalanced workload with hot spots* (see "Skewed Workloads and Relieving Hot Spots" on page 205), whereas here it means *timing anomaly*.

In Alice's case, this is not a lasting problem, because she will most likely see consistent account balances if she reloads the online banking website a few seconds later. However, some situations cannot tolerate such temporary inconsistency:

Backups
: Taking a backup requires making a copy of the entire database, which may take hours on a large database. During the time that the backup process is running, writes will continue to be made to the database. Thus, you could end up with some parts of the backup containing an older version of the data, and other parts containing a newer version. If you need to restore from such a backup, the inconsistencies (such as disappearing money) become permanent.

Analytic queries and integrity checks
: Sometimes, you may want to run a query that scans over large parts of the database. Such queries are common in analytics (see "Transaction Processing or Analytics?" on page 90), or may be part of a periodic integrity check that everything is in order (monitoring for data corruption). These queries are likely to return nonsensical results if they observe parts of the database at different points in time.

Snapshot isolation [28] is the most common solution to this problem. The idea is that each transaction reads from a *consistent snapshot* of the database—that is, the transaction sees all the data that was committed in the database at the start of the transaction. Even if the data is subsequently changed by another transaction, each transaction sees only the old data from that particular point in time.

Snapshot isolation is a boon for long-running, read-only queries such as backups and analytics. It is very hard to reason about the meaning of a query if the data on which

it operates is changing at the same time as the query is executing. When a transaction can see a consistent snapshot of the database, frozen at a particular point in time, it is much easier to understand.

Snapshot isolation is a popular feature: it is supported by PostgreSQL, MySQL with the InnoDB storage engine, Oracle, SQL Server, and others [23, 31, 32].

Implementing snapshot isolation

Like read committed isolation, implementations of snapshot isolation typically use write locks to prevent dirty writes (see "Implementing read committed" on page 236), which means that a transaction that makes a write can block the progress of another transaction that writes to the same object. However, reads do not require any locks. From a performance point of view, a key principle of snapshot isolation is *readers never block writers, and writers never block readers*. This allows a database to handle long-running read queries on a consistent snapshot at the same time as processing writes normally, without any lock contention between the two.

To implement snapshot isolation, databases use a generalization of the mechanism we saw for preventing dirty reads in Figure 7-4. The database must potentially keep several different committed versions of an object, because various in-progress transactions may need to see the state of the database at different points in time. Because it maintains several versions of an object side by side, this technique is known as *multi-version concurrency control* (MVCC).

If a database only needed to provide read committed isolation, but not snapshot isolation, it would be sufficient to keep two versions of an object: the committed version and the overwritten-but-not-yet-committed version. However, storage engines that support snapshot isolation typically use MVCC for their read committed isolation level as well. A typical approach is that read committed uses a separate snapshot for each query, while snapshot isolation uses the same snapshot for an entire transaction.

Figure 7-7 illustrates how MVCC-based snapshot isolation is implemented in PostgreSQL [31] (other implementations are similar). When a transaction is started, it is given a unique, always-increasing[vii] transaction ID (`txid`). Whenever a transaction writes anything to the database, the data it writes is tagged with the transaction ID of the writer.

vii. To be precise, transaction IDs are 32-bit integers, so they overflow after approximately 4 billion transactions. PostgreSQL's vacuum process performs cleanup which ensures that overflow does not affect the data.

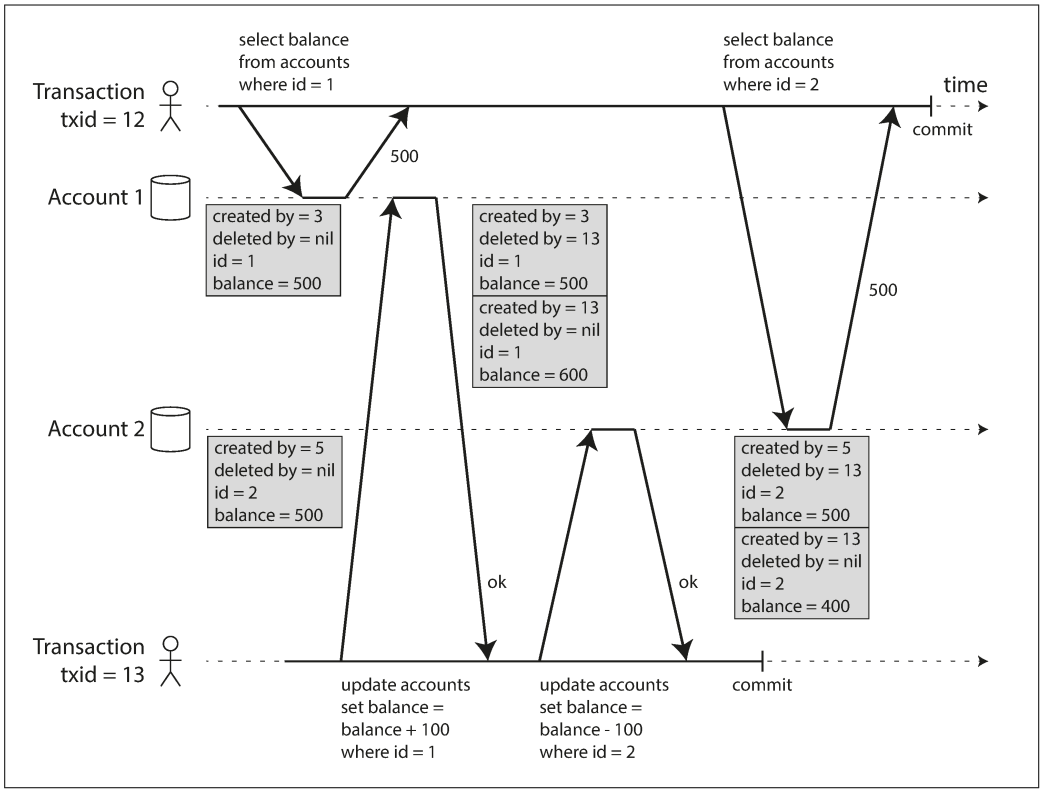


Figure 7-7. Implementing snapshot isolation using multi-version objects.

Each row in a table has a `created_by` field, containing the ID of the transaction that inserted this row into the table. Moreover, each row has a `deleted_by` field, which is initially empty. If a transaction deletes a row, the row isn't actually deleted from the database, but it is marked for deletion by setting the `deleted_by` field to the ID of the transaction that requested the deletion. At some later time, when it is certain that no transaction can any longer access the deleted data, a garbage collection process in the database removes any rows marked for deletion and frees their space.

An update is internally translated into a delete and a create. For example, in Figure 7-7, transaction 13 deducts $100 from account 2, changing the balance from $500 to $400. The `accounts` table now actually contains two rows for account 2: a row with a balance of $500 which was marked as deleted by transaction 13, and a row with a balance of $400 which was created by transaction 13.

Visibility rules for observing a consistent snapshot

When a transaction reads from the database, transaction IDs are used to decide which objects it can see and which are invisible. By carefully defining visibility rules,

the database can present a consistent snapshot of the database to the application. This works as follows:

1. At the start of each transaction, the database makes a list of all the other transactions that are in progress (not yet committed or aborted) at that time. Any writes that those transactions have made are ignored, even if the transactions subsequently commit.
2. Any writes made by aborted transactions are ignored.
3. Any writes made by transactions with a later transaction ID (i.e., which started after the current transaction started) are ignored, regardless of whether those transactions have committed.
4. All other writes are visible to the application's queries.

These rules apply to both creation and deletion of objects. In Figure 7-7, when transaction 12 reads from account 2, it sees a balance of $500 because the deletion of the $500 balance was made by transaction 13 (according to rule 3, transaction 12 cannot see a deletion made by transaction 13), and the creation of the $400 balance is not yet visible (by the same rule).

Put another way, an object is visible if both of the following conditions are true:

- At the time when the reader's transaction started, the transaction that created the object had already committed.
- The object is not marked for deletion, or if it is, the transaction that requested deletion had not yet committed at the time when the reader's transaction started.

A long-running transaction may continue using a snapshot for a long time, continuing to read values that (from other transactions' point of view) have long been overwritten or deleted. By never updating values in place but instead creating a new version every time a value is changed, the database can provide a consistent snapshot while incurring only a small overhead.

Indexes and snapshot isolation

How do indexes work in a multi-version database? One option is to have the index simply point to all versions of an object and require an index query to filter out any object versions that are not visible to the current transaction. When garbage collection removes old object versions that are no longer visible to any transaction, the corresponding index entries can also be removed.

In practice, many implementation details determine the performance of multi-version concurrency control. For example, PostgreSQL has optimizations for avoiding index updates if different versions of the same object can fit on the same page [31].

Another approach is used in CouchDB, Datomic, and LMDB. Although they also use B-trees (see "B-Trees" on page 79), they use an *append-only/copy-on-write* variant that does not overwrite pages of the tree when they are updated, but instead creates a new copy of each modified page. Parent pages, up to the root of the tree, are copied and updated to point to the new versions of their child pages. Any pages that are not affected by a write do not need to be copied, and remain immutable [33, 34, 35].

With append-only B-trees, every write transaction (or batch of transactions) creates a new B-tree root, and a particular root is a consistent snapshot of the database at the point in time when it was created. There is no need to filter out objects based on transaction IDs because subsequent writes cannot modify an existing B-tree; they can only create new tree roots. However, this approach also requires a background process for compaction and garbage collection.

Repeatable read and naming confusion

Snapshot isolation is a useful isolation level, especially for read-only transactions. However, many databases that implement it call it by different names. In Oracle it is called *serializable*, and in PostgreSQL and MySQL it is called *repeatable read* [23].

The reason for this naming confusion is that the SQL standard doesn't have the concept of snapshot isolation, because the standard is based on System R's 1975 definition of isolation levels [2] and snapshot isolation hadn't yet been invented then. Instead, it defines repeatable read, which looks superficially similar to snapshot isolation. PostgreSQL and MySQL call their snapshot isolation level repeatable read because it meets the requirements of the standard, and so they can claim standards compliance.

Unfortunately, the SQL standard's definition of isolation levels is flawed—it is ambiguous, imprecise, and not as implementation-independent as a standard should be [28]. Even though several databases implement repeatable read, there are big differences in the guarantees they actually provide, despite being ostensibly standardized [23]. There has been a formal definition of repeatable read in the research literature [29, 30], but most implementations don't satisfy that formal definition. And to top it off, IBM DB2 uses "repeatable read" to refer to serializability [8].

As a result, nobody really knows what repeatable read means.

Preventing Lost Updates

The read committed and snapshot isolation levels we've discussed so far have been primarily about the guarantees of what a read-only transaction can see in the presence of concurrent writes. We have mostly ignored the issue of two transactions writing concurrently—we have only discussed dirty writes (see "No dirty writes" on page 235), one particular type of write-write conflict that can occur.

There are several other interesting kinds of conflicts that can occur between concurrently writing transactions. The best known of these is the *lost update* problem, illustrated in Figure 7-1 with the example of two concurrent counter increments.

The lost update problem can occur if an application reads some value from the database, modifies it, and writes back the modified value (a *read-modify-write cycle*). If two transactions do this concurrently, one of the modifications can be lost, because the second write does not include the first modification. (We sometimes say that the later write *clobbers* the earlier write.) This pattern occurs in various different scenarios:

- Incrementing a counter or updating an account balance (requires reading the current value, calculating the new value, and writing back the updated value)
- Making a local change to a complex value, e.g., adding an element to a list within a JSON document (requires parsing the document, making the change, and writing back the modified document)
- Two users editing a wiki page at the same time, where each user saves their changes by sending the entire page contents to the server, overwriting whatever is currently in the database

Because this is such a common problem, a variety of solutions have been developed.

Atomic write operations

Many databases provide atomic update operations, which remove the need to implement read-modify-write cycles in application code. They are usually the best solution if your code can be expressed in terms of those operations. For example, the following instruction is concurrency-safe in most relational databases:

```
UPDATE counters SET value = value + 1 WHERE key = 'foo';
```

Similarly, document databases such as MongoDB provide atomic operations for making local modifications to a part of a JSON document, and Redis provides atomic operations for modifying data structures such as priority queues. Not all writes can easily be expressed in terms of atomic operations—for example, updates to a wiki page involve arbitrary text editing[viii]—but in situations where atomic operations can be used, they are usually the best choice.

Atomic operations are usually implemented by taking an exclusive lock on the object when it is read so that no other transaction can read it until the update has been

viii. It is possible, albeit fairly complicated, to express the editing of a text document as a stream of atomic mutations. See "Automatic Conflict Resolution" on page 174 for some pointers.

applied. This technique is sometimes known as *cursor stability* [36, 37]. Another option is to simply force all atomic operations to be executed on a single thread.

Unfortunately, object-relational mapping frameworks make it easy to accidentally write code that performs unsafe read-modify-write cycles instead of using atomic operations provided by the database [38]. That's not a problem if you know what you are doing, but it is potentially a source of subtle bugs that are difficult to find by testing.

Explicit locking

Another option for preventing lost updates, if the database's built-in atomic operations don't provide the necessary functionality, is for the application to explicitly lock objects that are going to be updated. Then the application can perform a read-modify-write cycle, and if any other transaction tries to concurrently read the same object, it is forced to wait until the first read-modify-write cycle has completed.

For example, consider a multiplayer game in which several players can move the same figure concurrently. In this case, an atomic operation may not be sufficient, because the application also needs to ensure that a player's move abides by the rules of the game, which involves some logic that you cannot sensibly implement as a database query. Instead, you may use a lock to prevent two players from concurrently moving the same piece, as illustrated in Example 7-1.

Example 7-1. Explicitly locking rows to prevent lost updates

```
BEGIN TRANSACTION;

SELECT * FROM figures
  WHERE name = 'robot' AND game_id = 222
  FOR UPDATE; ❶

-- Check whether move is valid, then update the position
-- of the piece that was returned by the previous SELECT.
UPDATE figures SET position = 'c4' WHERE id = 1234;

COMMIT;
```

❶ The `FOR UPDATE` clause indicates that the database should take a lock on all rows returned by this query.

This works, but to get it right, you need to carefully think about your application logic. It's easy to forget to add a necessary lock somewhere in the code, and thus introduce a race condition.

Automatically detecting lost updates

Atomic operations and locks are ways of preventing lost updates by forcing the read-modify-write cycles to happen sequentially. An alternative is to allow them to execute in parallel and, if the transaction manager detects a lost update, abort the transaction and force it to retry its read-modify-write cycle.

An advantage of this approach is that databases can perform this check efficiently in conjunction with snapshot isolation. Indeed, PostgreSQL's repeatable read, Oracle's serializable, and SQL Server's snapshot isolation levels automatically detect when a lost update has occurred and abort the offending transaction. However, MySQL/InnoDB's repeatable read does not detect lost updates [23]. Some authors [28, 30] argue that a database must prevent lost updates in order to qualify as providing snapshot isolation, so MySQL does not provide snapshot isolation under this definition.

Lost update detection is a great feature, because it doesn't require application code to use any special database features—you may forget to use a lock or an atomic operation and thus introduce a bug, but lost update detection happens automatically and is thus less error-prone.

Compare-and-set

In databases that don't provide transactions, you sometimes find an atomic compare-and-set operation (previously mentioned in "Single-object writes" on page 230). The purpose of this operation is to avoid lost updates by allowing an update to happen only if the value has not changed since you last read it. If the current value does not match what you previously read, the update has no effect, and the read-modify-write cycle must be retried.

For example, to prevent two users concurrently updating the same wiki page, you might try something like this, expecting the update to occur only if the content of the page hasn't changed since the user started editing it:

```
-- This may or may not be safe, depending on the database implementation
UPDATE wiki_pages SET content = 'new content'
  WHERE id = 1234 AND content = 'old content';
```

If the content has changed and no longer matches `'old content'`, this update will have no effect, so you need to check whether the update took effect and retry if necessary. However, if the database allows the `WHERE` clause to read from an old snapshot, this statement may not prevent lost updates, because the condition may be true even though another concurrent write is occurring. Check whether your database's compare-and-set operation is safe before relying on it.

Conflict resolution and replication

In replicated databases (see Chapter 5), preventing lost updates takes on another dimension: since they have copies of the data on multiple nodes, and the data can potentially be modified concurrently on different nodes, some additional steps need to be taken to prevent lost updates.

Locks and compare-and-set operations assume that there is a single up-to-date copy of the data. However, databases with multi-leader or leaderless replication usually allow several writes to happen concurrently and replicate them asynchronously, so they cannot guarantee that there is a single up-to-date copy of the data. Thus, techniques based on locks or compare-and-set do not apply in this context. (We will revisit this issue in more detail in "Linearizability" on page 324.)

Instead, as discussed in "Detecting Concurrent Writes" on page 184, a common approach in such replicated databases is to allow concurrent writes to create several conflicting versions of a value (also known as *siblings*), and to use application code or special data structures to resolve and merge these versions after the fact.

Atomic operations can work well in a replicated context, especially if they are commutative (i.e., you can apply them in a different order on different replicas, and still get the same result). For example, incrementing a counter or adding an element to a set are commutative operations. That is the idea behind Riak 2.0 datatypes, which prevent lost updates across replicas. When a value is concurrently updated by different clients, Riak automatically merges together the updates in such a way that no updates are lost [39].

On the other hand, the *last write wins* (LWW) conflict resolution method is prone to lost updates, as discussed in "Last write wins (discarding concurrent writes)" on page 186. Unfortunately, LWW is the default in many replicated databases.

Write Skew and Phantoms

In the previous sections we saw *dirty writes* and *lost updates*, two kinds of race conditions that can occur when different transactions concurrently try to write to the same objects. In order to avoid data corruption, those race conditions need to be prevented —either automatically by the database, or by manual safeguards such as using locks or atomic write operations.

However, that is not the end of the list of potential race conditions that can occur between concurrent writes. In this section we will see some subtler examples of conflicts.

To begin, imagine this example: you are writing an application for doctors to manage their on-call shifts at a hospital. The hospital usually tries to have several doctors on call at any one time, but it absolutely must have at least one doctor on call. Doctors

can give up their shifts (e.g., if they are sick themselves), provided that at least one colleague remains on call in that shift [40, 41].

Now imagine that Alice and Bob are the two on-call doctors for a particular shift. Both are feeling unwell, so they both decide to request leave. Unfortunately, they happen to click the button to go off call at approximately the same time. What happens next is illustrated in Figure 7-8.

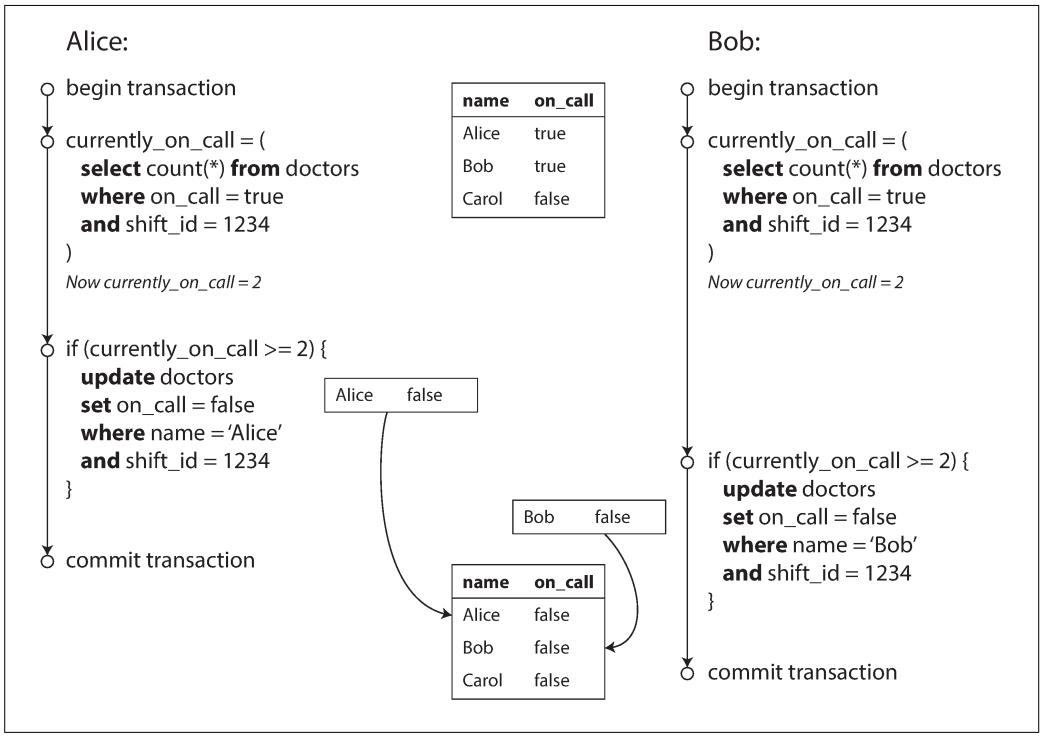


Figure 7-8. Example of write skew causing an application bug.

In each transaction, your application first checks that two or more doctors are currently on call; if yes, it assumes it's safe for one doctor to go off call. Since the database is using snapshot isolation, both checks return 2, so both transactions proceed to the next stage. Alice updates her own record to take herself off call, and Bob updates his own record likewise. Both transactions commit, and now no doctor is on call. Your requirement of having at least one doctor on call has been violated.

Characterizing write skew

This anomaly is called *write skew* [28]. It is neither a dirty write nor a lost update, because the two transactions are updating two different objects (Alice's and Bob's on-call records, respectively). It is less obvious that a conflict occurred here, but it's definitely a race condition: if the two transactions had run one after another, the second

doctor would have been prevented from going off call. The anomalous behavior was only possible because the transactions ran concurrently.

You can think of write skew as a generalization of the lost update problem. Write skew can occur if two transactions read the same objects, and then update some of those objects (different transactions may update different objects). In the special case where different transactions update the same object, you get a dirty write or lost update anomaly (depending on the timing).

We saw that there are various different ways of preventing lost updates. With write skew, our options are more restricted:

- Atomic single-object operations don't help, as multiple objects are involved.
- The automatic detection of lost updates that you find in some implementations of snapshot isolation unfortunately doesn't help either: write skew is not automatically detected in PostgreSQL's repeatable read, MySQL/InnoDB's repeatable read, Oracle's serializable, or SQL Server's snapshot isolation level [23]. Automatically preventing write skew requires true serializable isolation (see "Serializability" on page 251).
- Some databases allow you to configure constraints, which are then enforced by the database (e.g., uniqueness, foreign key constraints, or restrictions on a particular value). However, in order to specify that at least one doctor must be on call, you would need a constraint that involves multiple objects. Most databases do not have built-in support for such constraints, but you may be able to implement them with triggers or materialized views, depending on the database [42].
- If you can't use a serializable isolation level, the second-best option in this case is probably to explicitly lock the rows that the transaction depends on. In the doctors example, you could write something like the following:

```
BEGIN TRANSACTION;

SELECT * FROM doctors
  WHERE on_call = true
  AND shift_id = 1234 FOR UPDATE; ❶

UPDATE doctors
  SET on_call = false
  WHERE name = 'Alice'
  AND shift_id = 1234;

COMMIT;
```

❶ As before, `FOR UPDATE` tells the database to lock all rows returned by this query.

More examples of write skew

Write skew may seem like an esoteric issue at first, but once you're aware of it, you may notice more situations in which it can occur. Here are some more examples:

Meeting room booking system
: Say you want to enforce that there cannot be two bookings for the same meeting room at the same time [43]. When someone wants to make a booking, you first check for any conflicting bookings (i.e., bookings for the same room with an overlapping time range), and if none are found, you create the meeting (see Example 7-2).[ix]

Example 7-2. A meeting room booking system tries to avoid double-booking (not safe under snapshot isolation)

```
BEGIN TRANSACTION;

-- Check for any existing bookings that overlap with the period of noon-1pm
SELECT COUNT(*) FROM bookings
  WHERE room_id = 123 AND
    end_time > '2015-01-01 12:00' AND start_time < '2015-01-01 13:00';

-- If the previous query returned zero:
INSERT INTO bookings
  (room_id, start_time, end_time, user_id)
  VALUES (123, '2015-01-01 12:00', '2015-01-01 13:00', 666);

COMMIT;
```

Unfortunately, snapshot isolation does not prevent another user from concurrently inserting a conflicting meeting. In order to guarantee you won't get scheduling conflicts, you once again need serializable isolation.

Multiplayer game
: In Example 7-1, we used a lock to prevent lost updates (that is, making sure that two players can't move the same figure at the same time). However, the lock doesn't prevent players from moving two different figures to the same position on the board or potentially making some other move that violates the rules of the game. Depending on the kind of rule you are enforcing, you might be able to use a unique constraint, but otherwise you're vulnerable to write skew.

ix. In PostgreSQL you can do this more elegantly using range types, but they are not widely supported in other databases.

Claiming a username
: On a website where each user has a unique username, two users may try to create accounts with the same username at the same time. You may use a transaction to check whether a name is taken and, if not, create an account with that name. However, like in the previous examples, that is not safe under snapshot isolation. Fortunately, a unique constraint is a simple solution here (the second transaction that tries to register the username will be aborted due to violating the constraint).

Preventing double-spending
: A service that allows users to spend money or points needs to check that a user doesn't spend more than they have. You might implement this by inserting a tentative spending item into a user's account, listing all the items in the account, and checking that the sum is positive [44]. With write skew, it could happen that two spending items are inserted concurrently that together cause the balance to go negative, but that neither transaction notices the other.

Phantoms causing write skew

All of these examples follow a similar pattern:

1. A `SELECT` query checks whether some requirement is satisfied by searching for rows that match some search condition (there are at least two doctors on call, there are no existing bookings for that room at that time, the position on the board doesn't already have another figure on it, the username isn't already taken, there is still money in the account).
2. Depending on the result of the first query, the application code decides how to continue (perhaps to go ahead with the operation, or perhaps to report an error to the user and abort).
3. If the application decides to go ahead, it makes a write (`INSERT`, `UPDATE`, or `DELETE`) to the database and commits the transaction.

   The effect of this write changes the precondition of the decision of step 2. In other words, if you were to repeat the `SELECT` query from step 1 after commiting the write, you would get a different result, because the write changed the set of rows matching the search condition (there is now one fewer doctor on call, the meeting room is now booked for that time, the position on the board is now taken by the figure that was moved, the username is now taken, there is now less money in the account).

The steps may occur in a different order. For example, you could first make the write, then the `SELECT` query, and finally decide whether to abort or commit based on the result of the query.

In the case of the doctor on call example, the row being modified in step 3 was one of the rows returned in step 1, so we could make the transaction safe and avoid write skew by locking the rows in step 1 (`SELECT FOR UPDATE`). However, the other four examples are different: they check for the *absence* of rows matching some search condition, and the write *adds* a row matching the same condition. If the query in step 1 doesn't return any rows, `SELECT FOR UPDATE` can't attach locks to anything.

This effect, where a write in one transaction changes the result of a search query in another transaction, is called a *phantom* [3]. Snapshot isolation avoids phantoms in read-only queries, but in read-write transactions like the examples we discussed, phantoms can lead to particularly tricky cases of write skew.

Materializing conflicts

If the problem of phantoms is that there is no object to which we can attach the locks, perhaps we can artificially introduce a lock object into the database?

For example, in the meeting room booking case you could imagine creating a table of time slots and rooms. Each row in this table corresponds to a particular room for a particular time period (say, 15 minutes). You create rows for all possible combinations of rooms and time periods ahead of time, e.g. for the next six months.

Now a transaction that wants to create a booking can lock (`SELECT FOR UPDATE`) the rows in the table that correspond to the desired room and time period. After it has acquired the locks, it can check for overlapping bookings and insert a new booking as before. Note that the additional table isn't used to store information about the booking—it's purely a collection of locks which is used to prevent bookings on the same room and time range from being modified concurrently.

This approach is called *materializing conflicts*, because it takes a phantom and turns it into a lock conflict on a concrete set of rows that exist in the database [11]. Unfortunately, it can be hard and error-prone to figure out how to materialize conflicts, and it's ugly to let a concurrency control mechanism leak into the application data model. For those reasons, materializing conflicts should be considered a last resort if no alternative is possible. A serializable isolation level is much preferable in most cases.

Serializability

In this chapter we have seen several examples of transactions that are prone to race conditions. Some race conditions are prevented by the read committed and snapshot isolation levels, but others are not. We encountered some particularly tricky examples with write skew and phantoms. It's a sad situation:

- Isolation levels are hard to understand, and inconsistently implemented in different databases (e.g., the meaning of "repeatable read" varies significantly).

- If you look at your application code, it's difficult to tell whether it is safe to run at a particular isolation level—especially in a large application, where you might not be aware of all the things that may be happening concurrently.
- There are no good tools to help us detect race conditions. In principle, static analysis may help [26], but research techniques have not yet found their way into practical use. Testing for concurrency issues is hard, because they are usually nondeterministic—problems only occur if you get unlucky with the timing.

This is not a new problem—it has been like this since the 1970s, when weak isolation levels were first introduced [2]. All along, the answer from researchers has been simple: use *serializable* isolation!

Serializable isolation is usually regarded as the strongest isolation level. It guarantees that even though transactions may execute in parallel, the end result is the same as if they had executed one at a time, *serially*, without any concurrency. Thus, the database guarantees that if the transactions behave correctly when run individually, they continue to be correct when run concurrently—in other words, the database prevents *all* possible race conditions.

But if serializable isolation is so much better than the mess of weak isolation levels, then why isn't everyone using it? To answer this question, we need to look at the options for implementing serializability, and how they perform. Most databases that provide serializability today use one of three techniques, which we will explore in the rest of this chapter:

- Literally executing transactions in a serial order (see "Actual Serial Execution" on page 252)
- Two-phase locking (see "Two-Phase Locking (2PL)" on page 257), which for several decades was the only viable option
- Optimistic concurrency control techniques such as serializable snapshot isolation (see "Serializable Snapshot Isolation (SSI)" on page 261)

For now, we will discuss these techniques primarily in the context of single-node databases; in Chapter 9 we will examine how they can be generalized to transactions that involve multiple nodes in a distributed system.

Actual Serial Execution

The simplest way of avoiding concurrency problems is to remove the concurrency entirely: to execute only one transaction at a time, in serial order, on a single thread. By doing so, we completely sidestep the problem of detecting and preventing conflicts between transactions: the resulting isolation is by definition serializable.

Even though this seems like an obvious idea, database designers only fairly recently—around 2007—decided that a single-threaded loop for executing transactions was feasible [45]. If multi-threaded concurrency was considered essential for getting good performance during the previous 30 years, what changed to make single-threaded execution possible?

Two developments caused this rethink:

- RAM became cheap enough that for many use cases is now feasible to keep the entire active dataset in memory (see "Keeping everything in memory" on page 88). When all data that a transaction needs to access is in memory, transactions can execute much faster than if they have to wait for data to be loaded from disk.
- Database designers realized that OLTP transactions are usually short and only make a small number of reads and writes (see "Transaction Processing or Analytics?" on page 90). By contrast, long-running analytic queries are typically read-only, so they can be run on a consistent snapshot (using snapshot isolation) outside of the serial execution loop.

The approach of executing transactions serially is implemented in VoltDB/H-Store, Redis, and Datomic [46, 47, 48]. A system designed for single-threaded execution can sometimes perform better than a system that supports concurrency, because it can avoid the coordination overhead of locking. However, its throughput is limited to that of a single CPU core. In order to make the most of that single thread, transactions need to be structured differently from their traditional form.

Encapsulating transactions in stored procedures

In the early days of databases, the intention was that a database transaction could encompass an entire flow of user activity. For example, booking an airline ticket is a multi-stage process (searching for routes, fares, and available seats; deciding on an itinerary; booking seats on each of the flights of the itinerary; entering passenger details; making payment). Database designers thought that it would be neat if that entire process was one transaction so that it could be committed atomically.

Unfortunately, humans are very slow to make up their minds and respond. If a database transaction needs to wait for input from a user, the database needs to support a potentially huge number of concurrent transactions, most of them idle. Most databases cannot do that efficiently, and so almost all OLTP applications keep transactions short by avoiding interactively waiting for a user within a transaction. On the web, this means that a transaction is committed within the same HTTP request—a transaction does not span multiple requests. A new HTTP request starts a new transaction.

Even though the human has been taken out of the critical path, transactions have continued to be executed in an interactive client/server style, one statement at a time.

An application makes a query, reads the result, perhaps makes another query depending on the result of the first query, and so on. The queries and results are sent back and forth between the application code (running on one machine) and the database server (on another machine).

In this interactive style of transaction, a lot of time is spent in network communication between the application and the database. If you were to disallow concurrency in the database and only process one transaction at a time, the throughput would be dreadful because the database would spend most of its time waiting for the application to issue the next query for the current transaction. In this kind of database, it's necessary to process multiple transactions concurrently in order to get reasonable performance.

For this reason, systems with single-threaded serial transaction processing don't allow interactive multi-statement transactions. Instead, the application must submit the entire transaction code to the database ahead of time, as a *stored procedure*. The differences between these approaches is illustrated in Figure 7-9. Provided that all data required by a transaction is in memory, the stored procedure can execute very fast, without waiting for any network or disk I/O.

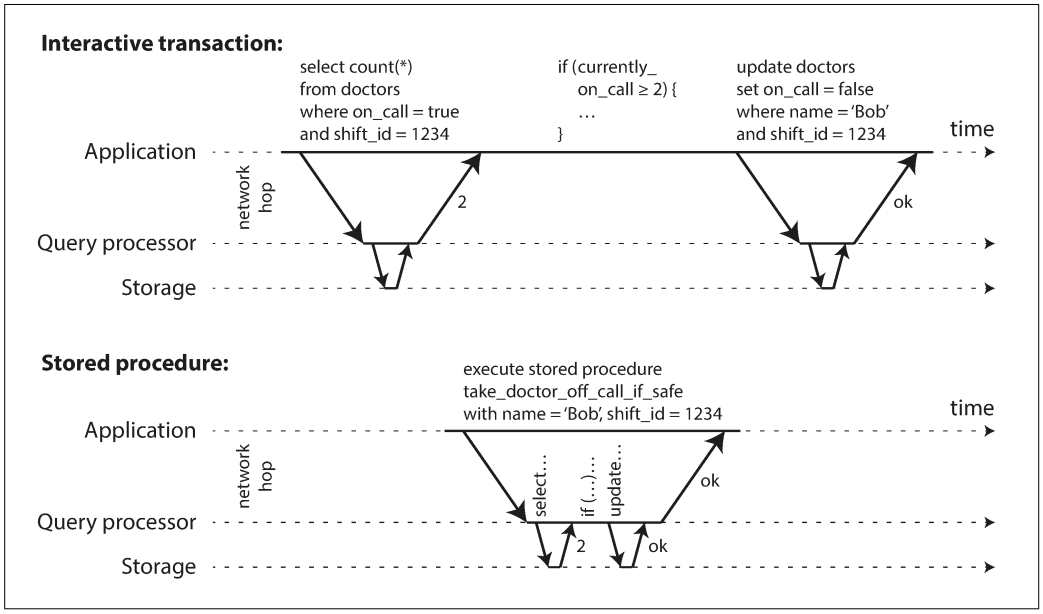


Figure 7-9. The difference between an interactive transaction and a stored procedure (using the example transaction of Figure 7-8).

Pros and cons of stored procedures

Stored procedures have existed for some time in relational databases, and they have been part of the SQL standard (SQL/PSM) since 1999. They have gained a somewhat bad reputation, for various reasons:

- Each database vendor has its own language for stored procedures (Oracle has PL/SQL, SQL Server has T-SQL, PostgreSQL has PL/pgSQL, etc.). These languages haven't kept up with developments in general-purpose programming languages, so they look quite ugly and archaic from today's point of view, and they lack the ecosystem of libraries that you find with most programming languages.
- Code running in a database is difficult to manage: compared to an application server, it's harder to debug, more awkward to keep in version control and deploy, trickier to test, and difficult to integrate with a metrics collection system for monitoring.
- A database is often much more performance-sensitive than an application server, because a single database instance is often shared by many application servers. A badly written stored procedure (e.g., using a lot of memory or CPU time) in a database can cause much more trouble than equivalent badly written code in an application server.

However, those issues can be overcome. Modern implementations of stored procedures have abandoned PL/SQL and use existing general-purpose programming languages instead: VoltDB uses Java or Groovy, Datomic uses Java or Clojure, and Redis uses Lua.

With stored procedures and in-memory data, executing all transactions on a single thread becomes feasible. As they don't need to wait for I/O and they avoid the overhead of other concurrency control mechanisms, they can achieve quite good throughput on a single thread.

VoltDB also uses stored procedures for replication: instead of copying a transaction's writes from one node to another, it executes the same stored procedure on each replica. VoltDB therefore requires that stored procedures are *deterministic* (when run on different nodes, they must produce the same result). If a transaction needs to use the current date and time, for example, it must do so through special deterministic APIs.

Partitioning

Executing all transactions serially makes concurrency control much simpler, but limits the transaction throughput of the database to the speed of a single CPU core on a single machine. Read-only transactions may execute elsewhere, using snapshot isolation, but for applications with high write throughput, the single-threaded transaction processor can become a serious bottleneck.

In order to scale to multiple CPU cores, and multiple nodes, you can potentially partition your data (see Chapter 6), which is supported in VoltDB. If you can find a way of partitioning your dataset so that each transaction only needs to read and write data within a single partition, then each partition can have its own transaction processing thread running independently from the others. In this case, you can give each CPU core its own partition, which allows your transaction throughput to scale linearly with the number of CPU cores [47].

However, for any transaction that needs to access multiple partitions, the database must coordinate the transaction across all the partitions that it touches. The stored procedure needs to be performed in lock-step across all partitions to ensure serializability across the whole system.

Since cross-partition transactions have additional coordination overhead, they are vastly slower than single-partition transactions. VoltDB reports a throughput of about 1,000 cross-partition writes per second, which is orders of magnitude below its single-partition throughput and cannot be increased by adding more machines [49].

Whether transactions can be single-partition depends very much on the structure of the data used by the application. Simple key-value data can often be partitioned very easily, but data with multiple secondary indexes is likely to require a lot of cross-partition coordination (see "Partitioning and Secondary Indexes" on page 206).

Summary of serial execution

Serial execution of transactions has become a viable way of achieving serializable isolation within certain constraints:

- Every transaction must be small and fast, because it takes only one slow transaction to stall all transaction processing.
- It is limited to use cases where the active dataset can fit in memory. Rarely accessed data could potentially be moved to disk, but if it needed to be accessed in a single-threaded transaction, the system would get very slow.[x]
- Write throughput must be low enough to be handled on a single CPU core, or else transactions need to be partitioned without requiring cross-partition coordination.
- Cross-partition transactions are possible, but there is a hard limit to the extent to which they can be used.

x. If a transaction needs to access data that's not in memory, the best solution may be to abort the transaction, asynchronously fetch the data into memory while continuing to process other transactions, and then restart the transaction when the data has been loaded. This approach is known as *anti-caching*, as previously mentioned in "Keeping everything in memory" on page 88.

Two-Phase Locking (2PL)

For around 30 years, there was only one widely used algorithm for serializability in databases: *two-phase locking* (2PL).[xi]

2PL is not 2PC

Note that while two-phase *locking* (2PL) sounds very similar to two-phase *commit* (2PC), they are completely different things. We will discuss 2PC in Chapter 9.

We saw previously that locks are often used to prevent dirty writes (see "No dirty writes" on page 235): if two transactions concurrently try to write to the same object, the lock ensures that the second writer must wait until the first one has finished its transaction (aborted or committed) before it may continue.

Two-phase locking is similar, but makes the lock requirements much stronger. Several transactions are allowed to concurrently read the same object as long as nobody is writing to it. But as soon as anyone wants to write (modify or delete) an object, exclusive access is required:

- If transaction A has read an object and transaction B wants to write to that object, B must wait until A commits or aborts before it can continue. (This ensures that B can't change the object unexpectedly behind A's back.)
- If transaction A has written an object and transaction B wants to read that object, B must wait until A commits or aborts before it can continue. (Reading an old version of the object, like in Figure 7-1, is not acceptable under 2PL.)

In 2PL, writers don't just block other writers; they also block readers and vice versa. Snapshot isolation has the mantra *readers never block writers, and writers never block readers* (see "Implementing snapshot isolation" on page 239), which captures this key difference between snapshot isolation and two-phase locking. On the other hand, because 2PL provides serializability, it protects against all the race conditions discussed earlier, including lost updates and write skew.

Implementation of two-phase locking

2PL is used by the serializable isolation level in MySQL (InnoDB) and SQL Server, and the repeatable read isolation level in DB2 [23, 36].

xi. Sometimes called *strong strict two-phase locking* (SS2PL) to distinguish it from other variants of 2PL.

The blocking of readers and writers is implemented by a having a lock on each object in the database. The lock can either be in *shared mode* or in *exclusive mode*. The lock is used as follows:

- If a transaction wants to read an object, it must first acquire the lock in shared mode. Several transactions are allowed to hold the lock in shared mode simultaneously, but if another transaction already has an exclusive lock on the object, these transactions must wait.
- If a transaction wants to write to an object, it must first acquire the lock in exclusive mode. No other transaction may hold the lock at the same time (either in shared or in exclusive mode), so if there is any existing lock on the object, the transaction must wait.
- If a transaction first reads and then writes an object, it may upgrade its shared lock to an exclusive lock. The upgrade works the same as getting an exclusive lock directly.
- After a transaction has acquired the lock, it must continue to hold the lock until the end of the transaction (commit or abort). This is where the name "two-phase" comes from: the first phase (while the transaction is executing) is when the locks are acquired, and the second phase (at the end of the transaction) is when all the locks are released.

Since so many locks are in use, it can happen quite easily that transaction A is stuck waiting for transaction B to release its lock, and vice versa. This situation is called *deadlock*. The database automatically detects deadlocks between transactions and aborts one of them so that the others can make progress. The aborted transaction needs to be retried by the application.

Performance of two-phase locking

The big downside of two-phase locking, and the reason why it hasn't been used by everybody since the 1970s, is performance: transaction throughput and response times of queries are significantly worse under two-phase locking than under weak isolation.

This is partly due to the overhead of acquiring and releasing all those locks, but more importantly due to reduced concurrency. By design, if two concurrent transactions try to do anything that may in any way result in a race condition, one has to wait for the other to complete.

Traditional relational databases don't limit the duration of a transaction, because they are designed for interactive applications that wait for human input. Consequently, when one transaction has to wait on another, there is no limit on how long it may have to wait. Even if you make sure that you keep all your transactions short, a

queue may form if several transactions want to access the same object, so a transaction may have to wait for several others to complete before it can do anything.

For this reason, databases running 2PL can have quite unstable latencies, and they can be very slow at high percentiles (see "Describing Performance" on page 13) if there is contention in the workload. It may take just one slow transaction, or one transaction that accesses a lot of data and acquires many locks, to cause the rest of the system to grind to a halt. This instability is problematic when robust operation is required.

Although deadlocks can happen with the lock-based read committed isolation level, they occur much more frequently under 2PL serializable isolation (depending on the access patterns of your transaction). This can be an additional performance problem: when a transaction is aborted due to deadlock and is retried, it needs to do its work all over again. If deadlocks are frequent, this can mean significant wasted effort.

Predicate locks

In the preceding description of locks, we glossed over a subtle but important detail. In "Phantoms causing write skew" on page 250 we discussed the problem of *phantoms*—that is, one transaction changing the results of another transaction's search query. A database with serializable isolation must prevent phantoms.

In the meeting room booking example this means that if one transaction has searched for existing bookings for a room within a certain time window (see Example 7-2), another transaction is not allowed to concurrently insert or update another booking for the same room and time range. (It's okay to concurrently insert bookings for other rooms, or for the same room at a different time that doesn't affect the proposed booking.)

How do we implement this? Conceptually, we need a *predicate lock* [3]. It works similarly to the shared/exclusive lock described earlier, but rather than belonging to a particular object (e.g., one row in a table), it belongs to all objects that match some search condition, such as:

```
SELECT * FROM bookings
  WHERE room_id = 123 AND
    end_time   > '2018-01-01 12:00' AND
    start_time < '2018-01-01 13:00';
```

A predicate lock restricts access as follows:

- If transaction A wants to read objects matching some condition, like in that `SELECT` query, it must acquire a shared-mode predicate lock on the conditions of the query. If another transaction B currently has an exclusive lock on any object matching those conditions, A must wait until B releases its lock before it is allowed to make its query.

- If transaction A wants to insert, update, or delete any object, it must first check whether either the old or the new value matches any existing predicate lock. If there is a matching predicate lock held by transaction B, then A must wait until B has committed or aborted before it can continue.

The key idea here is that a predicate lock applies even to objects that do not yet exist in the database, but which might be added in the future (phantoms). If two-phase locking includes predicate locks, the database prevents all forms of write skew and other race conditions, and so its isolation becomes serializable.

Index-range locks

Unfortunately, predicate locks do not perform well: if there are many locks by active transactions, checking for matching locks becomes time-consuming. For that reason, most databases with 2PL actually implement *index-range locking* (also known as *next-key locking*), which is a simplified approximation of predicate locking [41, 50].

It's safe to simplify a predicate by making it match a greater set of objects. For example, if you have a predicate lock for bookings of room 123 between noon and 1 p.m., you can approximate it by locking bookings for room 123 at any time, or you can approximate it by locking all rooms (not just room 123) between noon and 1 p.m. This is safe, because any write that matches the original predicate will definitely also match the approximations.

In the room bookings database you would probably have an index on the `room_id` column, and/or indexes on `start_time` and `end_time` (otherwise the preceding query would be very slow on a large database):

- Say your index is on `room_id`, and the database uses this index to find existing bookings for room 123. Now the database can simply attach a shared lock to this index entry, indicating that a transaction has searched for bookings of room 123.
- Alternatively, if the database uses a time-based index to find existing bookings, it can attach a shared lock to a range of values in that index, indicating that a transaction has searched for bookings that overlap with the time period of noon to 1 p.m. on January 1, 2018.

Either way, an approximation of the search condition is attached to one of the indexes. Now, if another transaction wants to insert, update, or delete a booking for the same room and/or an overlapping time period, it will have to update the same part of the index. In the process of doing so, it will encounter the shared lock, and it will be forced to wait until the lock is released.

This provides effective protection against phantoms and write skew. Index-range locks are not as precise as predicate locks would be (they may lock a bigger range of

objects than is strictly necessary to maintain serializability), but since they have much lower overheads, they are a good compromise.

If there is no suitable index where a range lock can be attached, the database can fall back to a shared lock on the entire table. This will not be good for performance, since it will stop all other transactions writing to the table, but it's a safe fallback position.

Serializable Snapshot Isolation (SSI)

This chapter has painted a bleak picture of concurrency control in databases. On the one hand, we have implementations of serializability that don't perform well (two-phase locking) or don't scale well (serial execution). On the other hand, we have weak isolation levels that have good performance, but are prone to various race conditions (lost updates, write skew, phantoms, etc.). Are serializable isolation and good performance fundamentally at odds with each other?

Perhaps not: an algorithm called *serializable snapshot isolation* (SSI) is very promising. It provides full serializability, but has only a small performance penalty compared to snapshot isolation. SSI is fairly new: it was first described in 2008 [40] and is the subject of Michael Cahill's PhD thesis [51].

Today SSI is used both in single-node databases (the serializable isolation level in PostgreSQL since version 9.1 [41]) and distributed databases (FoundationDB uses a similar algorithm). As SSI is so young compared to other concurrency control mechanisms, it is still proving its performance in practice, but it has the possibility of being fast enough to become the new default in the future.

Pessimistic versus optimistic concurrency control

Two-phase locking is a so-called *pessimistic* concurrency control mechanism: it is based on the principle that if anything might possibly go wrong (as indicated by a lock held by another transaction), it's better to wait until the situation is safe again before doing anything. It is like *mutual exclusion*, which is used to protect data structures in multi-threaded programming.

Serial execution is, in a sense, pessimistic to the extreme: it is essentially equivalent to each transaction having an exclusive lock on the entire database (or one partition of the database) for the duration of the transaction. We compensate for the pessimism by making each transaction very fast to execute, so it only needs to hold the "lock" for a short time.

By contrast, serializable snapshot isolation is an *optimistic* concurrency control technique. Optimistic in this context means that instead of blocking if something potentially dangerous happens, transactions continue anyway, in the hope that everything will turn out all right. When a transaction wants to commit, the database checks whether anything bad happened (i.e., whether isolation was violated); if so, the trans-

action is aborted and has to be retried. Only transactions that executed serializably are allowed to commit.

Optimistic concurrency control is an old idea [52], and its advantages and disadvantages have been debated for a long time [53]. It performs badly if there is high contention (many transactions trying to access the same objects), as this leads to a high proportion of transactions needing to abort. If the system is already close to its maximum throughput, the additional transaction load from retried transactions can make performance worse.

However, if there is enough spare capacity, and if contention between transactions is not too high, optimistic concurrency control techniques tend to perform better than pessimistic ones. Contention can be reduced with commutative atomic operations: for example, if several transactions concurrently want to increment a counter, it doesn't matter in which order the increments are applied (as long as the counter isn't read in the same transaction), so the concurrent increments can all be applied without conflicting.

As the name suggests, SSI is based on snapshot isolation—that is, all reads within a transaction are made from a consistent snapshot of the database (see "Snapshot Isolation and Repeatable Read" on page 237). This is the main difference compared to earlier optimistic concurrency control techniques. On top of snapshot isolation, SSI adds an algorithm for detecting serialization conflicts among writes and determining which transactions to abort.

Decisions based on an outdated premise

When we previously discussed write skew in snapshot isolation (see "Write Skew and Phantoms" on page 246), we observed a recurring pattern: a transaction reads some data from the database, examines the result of the query, and decides to take some action (write to the database) based on the result that it saw. However, under snapshot isolation, the result from the original query may no longer be up-to-date by the time the transaction commits, because the data may have been modified in the meantime.

Put another way, the transaction is taking an action based on a *premise* (a fact that was true at the beginning of the transaction, e.g., "There are currently two doctors on call"). Later, when the transaction wants to commit, the original data may have changed—the premise may no longer be true.

When the application makes a query (e.g., "How many doctors are currently on call?"), the database doesn't know how the application logic uses the result of that query. To be safe, the database needs to assume that any change in the query result (the premise) means that writes in that transaction may be invalid. In other words, there may be a causal dependency between the queries and the writes in the transaction. In order to provide serializable isolation, the database must detect situations in

which a transaction may have acted on an outdated premise and abort the transaction in that case.

How does the database know if a query result might have changed? There are two cases to consider:

- Detecting reads of a stale MVCC object version (uncommitted write occurred before the read)
- Detecting writes that affect prior reads (the write occurs after the read)

Detecting stale MVCC reads

Recall that snapshot isolation is usually implemented by multi-version concurrency control (MVCC; see Figure 7-10). When a transaction reads from a consistent snapshot in an MVCC database, it ignores writes that were made by any other transactions that hadn't yet committed at the time when the snapshot was taken. In Figure 7-10, transaction 43 sees Alice as having `on_call = true`, because transaction 42 (which modified Alice's on-call status) is uncommitted. However, by the time transaction 43 wants to commit, transaction 42 has already committed. This means that the write that was ignored when reading from the consistent snapshot has now taken effect, and transaction 43's premise is no longer true.

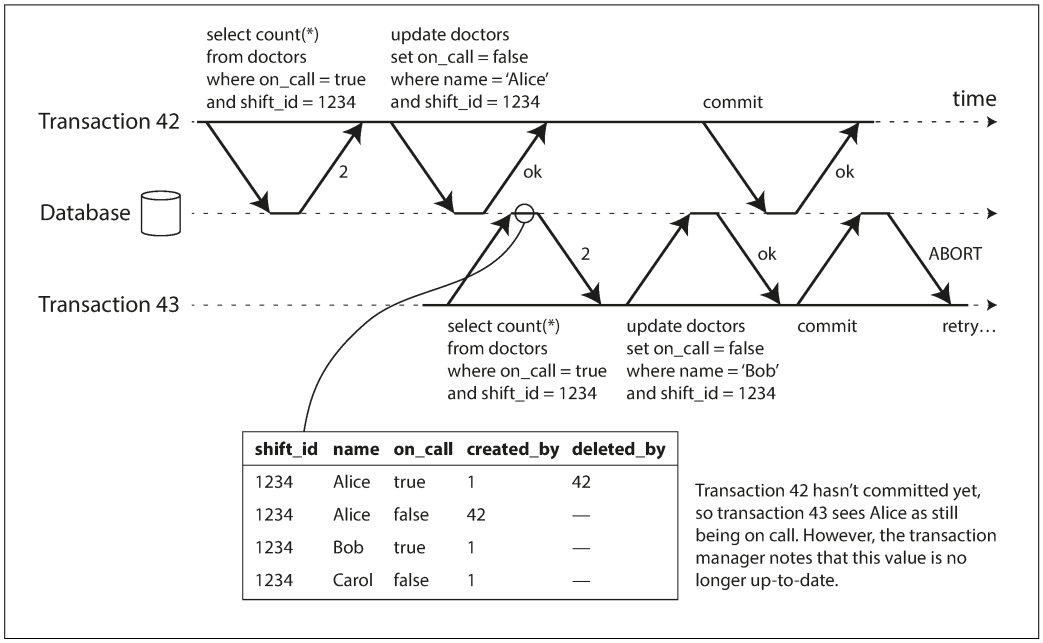


Figure 7-10. Detecting when a transaction reads outdated values from an MVCC snapshot.

In order to prevent this anomaly, the database needs to track when a transaction ignores another transaction's writes due to MVCC visibility rules. When the transaction wants to commit, the database checks whether any of the ignored writes have now been committed. If so, the transaction must be aborted.

Why wait until committing? Why not abort transaction 43 immediately when the stale read is detected? Well, if transaction 43 was a read-only transaction, it wouldn't need to be aborted, because there is no risk of write skew. At the time when transaction 43 makes its read, the database doesn't yet know whether that transaction is going to later perform a write. Moreover, transaction 42 may yet abort or may still be uncommitted at the time when transaction 43 is committed, and so the read may turn out not to have been stale after all. By avoiding unnecessary aborts, SSI preserves snapshot isolation's support for long-running reads from a consistent snapshot.

Detecting writes that affect prior reads

The second case to consider is when another transaction modifies data after it has been read. This case is illustrated in Figure 7-11.

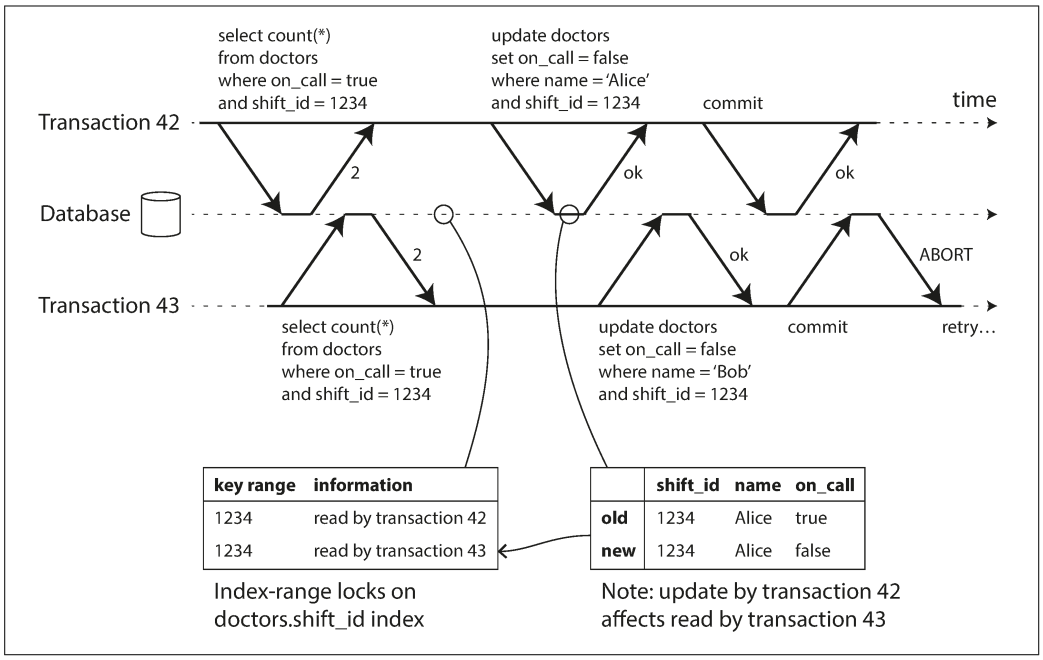


Figure 7-11. In serializable snapshot isolation, detecting when one transaction modifies another transaction's reads.

In the context of two-phase locking we discussed index-range locks (see "Index-range locks" on page 260), which allow the database to lock access to all rows matching some search query, such as `WHERE shift_id = 1234`. We can use a similar technique here, except that SSI locks don't block other transactions.

In Figure 7-11, transactions 42 and 43 both search for on-call doctors during shift `1234`. If there is an index on `shift_id`, the database can use the index entry 1234 to record the fact that transactions 42 and 43 read this data. (If there is no index, this information can be tracked at the table level.) This information only needs to be kept for a while: after a transaction has finished (committed or aborted), and all concurrent transactions have finished, the database can forget what data it read.

When a transaction writes to the database, it must look in the indexes for any other transactions that have recently read the affected data. This process is similar to acquiring a write lock on the affected key range, but rather than blocking until the readers have committed, the lock acts as a tripwire: it simply notifies the transactions that the data they read may no longer be up to date.

In Figure 7-11, transaction 43 notifies transaction 42 that its prior read is outdated, and vice versa. Transaction 42 is first to commit, and it is successful: although transaction 43's write affected 42, 43 hasn't yet committed, so the write has not yet taken effect. However, when transaction 43 wants to commit, the conflicting write from 42 has already been committed, so 43 must abort.

Performance of serializable snapshot isolation

As always, many engineering details affect how well an algorithm works in practice. For example, one trade-off is the granularity at which transactions' reads and writes are tracked. If the database keeps track of each transaction's activity in great detail, it can be precise about which transactions need to abort, but the bookkeeping overhead can become significant. Less detailed tracking is faster, but may lead to more transactions being aborted than strictly necessary.

In some cases, it's okay for a transaction to read information that was overwritten by another transaction: depending on what else happened, it's sometimes possible to prove that the result of the execution is nevertheless serializable. PostgreSQL uses this theory to reduce the number of unnecessary aborts [11, 41].

Compared to two-phase locking, the big advantage of serializable snapshot isolation is that one transaction doesn't need to block waiting for locks held by another transaction. Like under snapshot isolation, writers don't block readers, and vice versa. This design principle makes query latency much more predictable and less variable. In particular, read-only queries can run on a consistent snapshot without requiring any locks, which is very appealing for read-heavy workloads.

Compared to serial execution, serializable snapshot isolation is not limited to the throughput of a single CPU core: FoundationDB distributes the detection of serialization conflicts across multiple machines, allowing it to scale to very high throughput. Even though data may be partitioned across multiple machines, transactions can read and write data in multiple partitions while ensuring serializable isolation [54].

The rate of aborts significantly affects the overall performance of SSI. For example, a transaction that reads and writes data over a long period of time is likely to run into conflicts and abort, so SSI requires that read-write transactions be fairly short (long-running read-only transactions may be okay). However, SSI is probably less sensitive to slow transactions than two-phase locking or serial execution.

Summary

Transactions are an abstraction layer that allows an application to pretend that certain concurrency problems and certain kinds of hardware and software faults don't exist. A large class of errors is reduced down to a simple *transaction abort*, and the application just needs to try again.

In this chapter we saw many examples of problems that transactions help prevent. Not all applications are susceptible to all those problems: an application with very simple access patterns, such as reading and writing only a single record, can probably manage without transactions. However, for more complex access patterns, transactions can hugely reduce the number of potential error cases you need to think about.

Without transactions, various error scenarios (processes crashing, network interruptions, power outages, disk full, unexpected concurrency, etc.) mean that data can become inconsistent in various ways. For example, denormalized data can easily go out of sync with the source data. Without transactions, it becomes very difficult to reason about the effects that complex interacting accesses can have on the database.

In this chapter, we went particularly deep into the topic of concurrency control. We discussed several widely used isolation levels, in particular *read committed*, *snapshot isolation* (sometimes called *repeatable read*), and *serializable*. We characterized those isolation levels by discussing various examples of race conditions:

Dirty reads
: One client reads another client's writes before they have been committed. The read committed isolation level and stronger levels prevent dirty reads.

Dirty writes
: One client overwrites data that another client has written, but not yet committed. Almost all transaction implementations prevent dirty writes.

Read skew (nonrepeatable reads)
: A client sees different parts of the database at different points in time. This issue is most commonly prevented with snapshot isolation, which allows a transaction to read from a consistent snapshot at one point in time. It is usually implemented with *multi-version concurrency control* (MVCC).

Lost updates
: Two clients concurrently perform a read-modify-write cycle. One overwrites the other's write without incorporating its changes, so data is lost. Some implementations of snapshot isolation prevent this anomaly automatically, while others require a manual lock (`SELECT FOR UPDATE`).

Write skew
: A transaction reads something, makes a decision based on the value it saw, and writes the decision to the database. However, by the time the write is made, the premise of the decision is no longer true. Only serializable isolation prevents this anomaly.

Phantom reads
: A transaction reads objects that match some search condition. Another client makes a write that affects the results of that search. Snapshot isolation prevents straightforward phantom reads, but phantoms in the context of write skew require special treatment, such as index-range locks.

Weak isolation levels protect against some of those anomalies but leave you, the application developer, to handle others manually (e.g., using explicit locking). Only serializable isolation protects against all of these issues. We discussed three different approaches to implementing serializable transactions:

Literally executing transactions in a serial order
: If you can make each transaction very fast to execute, and the transaction throughput is low enough to process on a single CPU core, this is a simple and effective option.

Two-phase locking
: For decades this has been the standard way of implementing serializability, but many applications avoid using it because of its performance characteristics.

Serializable snapshot isolation (SSI)
: A fairly new algorithm that avoids most of the downsides of the previous approaches. It uses an optimistic approach, allowing transactions to proceed without blocking. When a transaction wants to commit, it is checked, and it is aborted if the execution was not serializable.

The examples in this chapter used a relational data model. However, as discussed in "The need for multi-object transactions" on page 231, transactions are a valuable database feature, no matter which data model is used.

In this chapter, we explored ideas and algorithms mostly in the context of a database running on a single machine. Transactions in distributed databases open a new set of difficult challenges, which we'll discuss in the next two chapters.

References

[1] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, et al.: "A History and Evaluation of System R," *Communications of the ACM*, volume 24, number 10, pages 632–646, October 1981. doi:10.1145/358769.358784

[2] Jim N. Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger: "Granularity of Locks and Degrees of Consistency in a Shared Data Base," in *Modelling in Data Base Management Systems: Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems*, edited by G. M. Nijssen, pages 364–394, Elsevier/North Holland Publishing, 1976. Also in *Readings in Database Systems*, 4th edition, edited by Joseph M. Hellerstein and Michael Stonebraker, MIT Press, 2005. ISBN: 978-0-262-69314-1

[3] Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie, and Irving L. Traiger: "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, volume 19, number 11, pages 624–633, November 1976.

[4] "ACID Transactions Are Incredibly Helpful," FoundationDB, LLC, 2013.

[5] John D. Cook: "ACID Versus BASE for Database Transactions," *johndcook.com*, July 6, 2009.

[6] Gavin Clarke: "NoSQL's CAP Theorem Busters: We Don't Drop ACID," *theregister.co.uk*, November 22, 2012.

[7] Theo Härder and Andreas Reuter: "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, volume 15, number 4, pages 287–317, December 1983. doi:10.1145/289.291

[8] Peter Bailis, Alan Fekete, Ali Ghodsi, et al.: "HAT, not CAP: Towards Highly Available Transactions," at *14th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2013.

[9] Armando Fox, Steven D. Gribble, Yatin Chawathe, et al.: "Cluster-Based Scalable Network Services," at *16th ACM Symposium on Operating Systems Principles* (SOSP), October 1997.

[10] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman: *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN: 978-0-201-10715-9, available online at *research.microsoft.com*.

[11] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, et al.: "Making Snapshot Isolation Serializable," *ACM Transactions on Database Systems*, volume 30, number 2, pages 492–528, June 2005. doi:10.1145/1071610.1071615

[12] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge: "Understanding the Robustness of SSDs Under Power Fault," at *11th USENIX Conference on File and Storage Technologies* (FAST), February 2013.

[13] Laurie Denness: "SSDs: A Gift and a Curse," *laur.ie*, June 2, 2015.

[14] Adam Surak: "When Solid State Drives Are Not That Solid," *blog.algolia.com*, June 15, 2015.

[15] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, et al.: "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications," at *11th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2014.

[16] Chris Siebenmann: "Unix's File Durability Problem," *utcc.utoronto.ca*, April 14, 2016.

[17] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, et al.: "An Analysis of Data Corruption in the Storage Stack," at *6th USENIX Conference on File and Storage Technologies* (FAST), February 2008.

[18] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant: "Flash Reliability in Production: The Expected and the Unexpected," at *14th USENIX Conference on File and Storage Technologies* (FAST), February 2016.

[19] Don Allison: "SSD Storage – Ignorance of Technology Is No Excuse," *blog.korelogic.com*, March 24, 2015.

[20] Dave Scherer: "Those Are Not Transactions (Cassandra 2.0)," *blog.foundationdb.com*, September 6, 2013.

[21] Kyle Kingsbury: "Call Me Maybe: Cassandra," *aphyr.com*, September 24, 2013.

[22] "ACID Support in Aerospike," Aerospike, Inc., June 2014.

[23] Martin Kleppmann: "Hermitage: Testing the 'I' in ACID," *martin.kleppmann.com*, November 25, 2014.

[24] Tristan D'Agosta: "BTC Stolen from Poloniex," *bitcointalk.org*, March 4, 2014.

[25] bitcointhief2: "How I Stole Roughly 100 BTC from an Exchange and How I Could Have Stolen More!," *reddit.com*, February 2, 2014.

[26] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan: "Automating the Detection of Snapshot Isolation Anomalies," at *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.

[27] Michael Melanson: "Transactions: The Limits of Isolation," *michaelmelanson.net*, March 20, 2014.

[28] Hal Berenson, Philip A. Bernstein, Jim N. Gray, et al.: "A Critique of ANSI SQL Isolation Levels," at *ACM International Conference on Management of Data* (SIGMOD), May 1995.

[29] Atul Adya: "Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions," PhD Thesis, Massachusetts Institute of Technology, March 1999.

[30] Peter Bailis, Aaron Davidson, Alan Fekete, et al.: "Highly Available Transactions: Virtues and Limitations (Extended Version)," at *40th International Conference on Very Large Data Bases* (VLDB), September 2014.

[31] Bruce Momjian: "MVCC Unmasked," *momjian.us*, July 2014.

[32] Annamalai Gurusami: "Repeatable Read Isolation Level in InnoDB – How Consistent Read View Works," *blogs.oracle.com*, January 15, 2013.

[33] Nikita Prokopov: "Unofficial Guide to Datomic Internals," *tonsky.me*, May 6, 2014.

[34] Baron Schwartz: "Immutability, MVCC, and Garbage Collection," *xaprb.com*, December 28, 2013.

[35] J. Chris Anderson, Jan Lehnardt, and Noah Slater: *CouchDB: The Definitive Guide*. O'Reilly Media, 2010. ISBN: 978-0-596-15589-6

[36] Rikdeb Mukherjee: "Isolation in DB2 (Repeatable Read, Read Stability, Cursor Stability, Uncommitted Read) with Examples," *mframes.blogspot.co.uk*, July 4, 2013.

[37] Steve Hilker: "Cursor Stability (CS) – IBM DB2 Community," *toadworld.com*, March 14, 2013.

[38] Nate Wiger: "An Atomic Rant," *nateware.com*, February 18, 2010.

[39] Joel Jacobson: "Riak 2.0: Data Types," *blog.joeljacobson.com*, March 23, 2014.

[40] Michael J. Cahill, Uwe Röhm, and Alan Fekete: "Serializable Isolation for Snapshot Databases," at *ACM International Conference on Management of Data* (SIGMOD), June 2008. doi:10.1145/1376616.1376690

[41] Dan R. K. Ports and Kevin Grittner: "Serializable Snapshot Isolation in PostgreSQL," at *38th International Conference on Very Large Databases* (VLDB), August 2012.

[42] Tony Andrews: "Enforcing Complex Constraints in Oracle," *tonyandrews.blogspot.co.uk*, October 15, 2004.

[43] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, et al.: "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," at *15th ACM*

Symposium on Operating Systems Principles (SOSP), December 1995. doi:10.1145/224056.224070

[44] Gary Fredericks: "Postgres Serializability Bug," *github.com*, September 2015.

[45] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, et al.: "The End of an Architectural Era (It's Time for a Complete Rewrite)," at *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.

[46] John Hugg: "H-Store/VoltDB Architecture vs. CEP Systems and Newer Streaming Architectures," at *Data @Scale Boston*, November 2014.

[47] Robert Kallman, Hideaki Kimura, Jonathan Natkins, et al.: "H-Store: A High-Performance, Distributed Main Memory Transaction Processing System," *Proceedings of the VLDB Endowment*, volume 1, number 2, pages 1496–1499, August 2008.

[48] Rich Hickey: "The Architecture of Datomic," *infoq.com*, November 2, 2012.

[49] John Hugg: "Debunking Myths About the VoltDB In-Memory Database," *voltdb.com*, May 12, 2014.

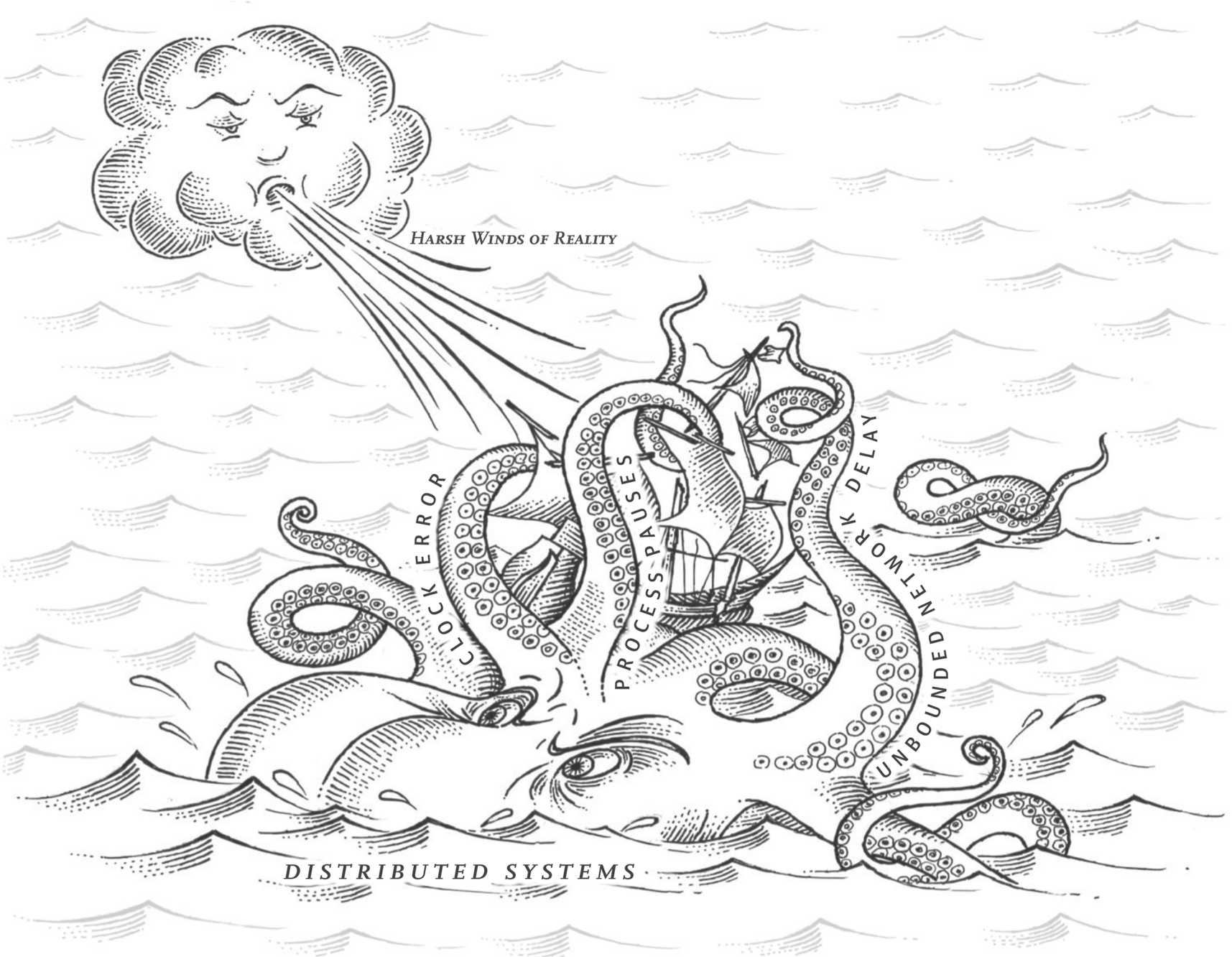
[50] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton: "Architecture of a Database System," *Foundations and Trends in Databases*, volume 1, number 2, pages 141–259, November 2007. doi:10.1561/1900000002

[51] Michael J. Cahill: "Serializable Isolation for Snapshot Databases," PhD Thesis, University of Sydney, July 2009.

[52] D. Z. Badal: "Correctness of Concurrency Control and Implications in Distributed Databases," at *3rd International IEEE Computer Software and Applications Conference* (COMPSAC), November 1979.

[53] Rakesh Agrawal, Michael J. Carey, and Miron Livny: "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Transactions on Database Systems* (TODS), volume 12, number 4, pages 609–654, December 1987. doi:10.1145/32204.32220

[54] Dave Rosenthal: "Databases at 14.4MHz," *blog.foundationdb.com*, December 10, 2014.


Harsh Winds of Reality
Clock Error
Process Pauses
Unbounded Network Delay
Distributed Systems

CHAPTER 8
The Trouble with Distributed Systems

Hey I just met you
The network's laggy
But here's my data
So store it maybe

—Kyle Kingsbury, *Carly Rae Jepsen and the Perils of Network Partitions* (2013)

A recurring theme in the last few chapters has been how systems handle things going wrong. For example, we discussed replica failover ("Handling Node Outages" on page 156), replication lag ("Problems with Replication Lag" on page 161), and concurrency control for transactions ("Weak Isolation Levels" on page 233). As we come to understand various edge cases that can occur in real systems, we get better at handling them.

However, even though we have talked a lot about faults, the last few chapters have still been too optimistic. The reality is even darker. We will now turn our pessimism to the maximum and assume that anything that *can* go wrong *will* go wrong.[i] (Experienced systems operators will tell you that is a reasonable assumption. If you ask nicely, they might tell you some frightening stories while nursing their scars of past battles.)

Working with distributed systems is fundamentally different from writing software on a single computer—and the main difference is that there are lots of new and exciting ways for things to go wrong [1, 2]. In this chapter, we will get a taste of the problems that arise in practice, and an understanding of the things we can and cannot rely on.

i. With one exception: we will assume that faults are *non-Byzantine* (see "Byzantine Faults" on page 304).

In the end, our task as engineers is to build systems that do their job (i.e., meet the guarantees that users are expecting), in spite of everything going wrong. In Chapter 9, we will look at some examples of algorithms that can provide such guarantees in a distributed system. But first, in this chapter, we must understand what challenges we are up against.

This chapter is a thoroughly pessimistic and depressing overview of things that may go wrong in a distributed system. We will look into problems with networks ("Unreliable Networks" on page 277); clocks and timing issues ("Unreliable Clocks" on page 287); and we'll discuss to what degree they are avoidable. The consequences of all these issues are disorienting, so we'll explore how to think about the state of a distributed system and how to reason about things that have happened ("Knowledge, Truth, and Lies" on page 300).

Faults and Partial Failures

When you are writing a program on a single computer, it normally behaves in a fairly predictable way: either it works or it doesn't. Buggy software may give the appearance that the computer is sometimes "having a bad day" (a problem that is often fixed by a reboot), but that is mostly just a consequence of badly written software.

There is no fundamental reason why software on a single computer should be flaky: when the hardware is working correctly, the same operation always produces the same result (it is *deterministic*). If there is a hardware problem (e.g., memory corruption or a loose connector), the consequence is usually a total system failure (e.g., kernel panic, "blue screen of death," failure to start up). An individual computer with good software is usually either fully functional or entirely broken, but not something in between.

This is a deliberate choice in the design of computers: if an internal fault occurs, we prefer a computer to crash completely rather than returning a wrong result, because wrong results are difficult and confusing to deal with. Thus, computers hide the fuzzy physical reality on which they are implemented and present an idealized system model that operates with mathematical perfection. A CPU instruction always does the same thing; if you write some data to memory or disk, that data remains intact and doesn't get randomly corrupted. This design goal of always-correct computation goes all the way back to the very first digital computer [3].

When you are writing software that runs on several computers, connected by a network, the situation is fundamentally different. In distributed systems, we are no longer operating in an idealized system model—we have no choice but to confront the messy reality of the physical world. And in the physical world, a remarkably wide range of things can go wrong, as illustrated by this anecdote [4]:

> In my limited experience I've dealt with long-lived network partitions in a single data center (DC), PDU [power distribution unit] failures, switch failures, accidental power cycles of whole racks, whole-DC backbone failures, whole-DC power failures, and a hypoglycemic driver smashing his Ford pickup truck into a DC's HVAC [heating, ventilation, and air conditioning] system. And I'm not even an ops guy.
>
> —Coda Hale

In a distributed system, there may well be some parts of the system that are broken in some unpredictable way, even though other parts of the system are working fine. This is known as a *partial failure*. The difficulty is that partial failures are *nondeterministic*: if you try to do anything involving multiple nodes and the network, it may sometimes work and sometimes unpredictably fail. As we shall see, you may not even *know* whether something succeeded or not, as the time it takes for a message to travel across a network is also nondeterministic!

This nondeterminism and possibility of partial failures is what makes distributed systems hard to work with [5].

Cloud Computing and Supercomputing

There is a spectrum of philosophies on how to build large-scale computing systems:

- At one end of the scale is the field of *high-performance computing* (HPC). Supercomputers with thousands of CPUs are typically used for computationally intensive scientific computing tasks, such as weather forecasting or molecular dynamics (simulating the movement of atoms and molecules).
- At the other extreme is *cloud computing*, which is not very well defined [6] but is often associated with multi-tenant datacenters, commodity computers connected with an IP network (often Ethernet), elastic/on-demand resource allocation, and metered billing.
- Traditional enterprise datacenters lie somewhere between these extremes.

With these philosophies come very different approaches to handling faults. In a supercomputer, a job typically checkpoints the state of its computation to durable storage from time to time. If one node fails, a common solution is to simply stop the entire cluster workload. After the faulty node is repaired, the computation is restarted from the last checkpoint [7, 8]. Thus, a supercomputer is more like a single-node computer than a distributed system: it deals with partial failure by letting it escalate into total failure—if any part of the system fails, just let everything crash (like a kernel panic on a single machine).

In this book we focus on systems for implementing internet services, which usually look very different from supercomputers:

- Many internet-related applications are *online*, in the sense that they need to be able to serve users with low latency at any time. Making the service unavailable—for example, stopping the cluster for repair—is not acceptable. In contrast, offline (batch) jobs like weather simulations can be stopped and restarted with fairly low impact.
- Supercomputers are typically built from specialized hardware, where each node is quite reliable, and nodes communicate through shared memory and remote direct memory access (RDMA). On the other hand, nodes in cloud services are built from commodity machines, which can provide equivalent performance at lower cost due to economies of scale, but also have higher failure rates.
- Large datacenter networks are often based on IP and Ethernet, arranged in Clos topologies to provide high bisection bandwidth [9]. Supercomputers often use specialized network topologies, such as multi-dimensional meshes and toruses [10], which yield better performance for HPC workloads with known communication patterns.
- The bigger a system gets, the more likely it is that one of its components is broken. Over time, broken things get fixed and new things break, but in a system with thousands of nodes, it is reasonable to assume that *something* is always broken [7]. When the error handling strategy consists of simply giving up, a large system can end up spending a lot of its time recovering from faults rather than doing useful work [8].
- If the system can tolerate failed nodes and still keep working as a whole, that is a very useful feature for operations and maintenance: for example, you can perform a rolling upgrade (see Chapter 4), restarting one node at a time, while the service continues serving users without interruption. In cloud environments, if one virtual machine is not performing well, you can just kill it and request a new one (hoping that the new one will be faster).
- In a geographically distributed deployment (keeping data geographically close to your users to reduce access latency), communication most likely goes over the internet, which is slow and unreliable compared to local networks. Supercomputers generally assume that all of their nodes are close together.

If we want to make distributed systems work, we must accept the possibility of partial failure and build fault-tolerance mechanisms into the software. In other words, we need to build a reliable system from unreliable components. (As discussed in "Reliability" on page 6, there is no such thing as perfect reliability, so we'll need to understand the limits of what we can realistically promise.)

Even in smaller systems consisting of only a few nodes, it's important to think about partial failure. In a small system, it's quite likely that most of the components are working correctly most of the time. However, sooner or later, some part of the system

will become faulty, and the software will have to somehow handle it. The fault handling must be part of the software design, and you (as operator of the software) need to know what behavior to expect from the software in the case of a fault.

It would be unwise to assume that faults are rare and simply hope for the best. It is important to consider a wide range of possible faults—even fairly unlikely ones—and to artificially create such situations in your testing environment to see what happens. In distributed systems, suspicion, pessimism, and paranoia pay off.

Building a Reliable System from Unreliable Components

You may wonder whether this makes any sense—intuitively it may seem like a system can only be as reliable as its least reliable component (its *weakest link*). This is not the case: in fact, it is an old idea in computing to construct a more reliable system from a less reliable underlying base [11]. For example:

- Error-correcting codes allow digital data to be transmitted accurately across a communication channel that occasionally gets some bits wrong, for example due to radio interference on a wireless network [12].
- IP (the Internet Protocol) is unreliable: it may drop, delay, duplicate, or reorder packets. TCP (the Transmission Control Protocol) provides a more reliable transport layer on top of IP: it ensures that missing packets are retransmitted, duplicates are eliminated, and packets are reassembled into the order in which they were sent.

Although the system can be more reliable than its underlying parts, there is always a limit to how much more reliable it can be. For example, error-correcting codes can deal with a small number of single-bit errors, but if your signal is swamped by interference, there is a fundamental limit to how much data you can get through your communication channel [13]. TCP can hide packet loss, duplication, and reordering from you, but it cannot magically remove delays in the network.

Although the more reliable higher-level system is not perfect, it's still useful because it takes care of some of the tricky low-level faults, and so the remaining faults are usually easier to reason about and deal with. We will explore this matter further in "The end-to-end argument" on page 519.

Unreliable Networks

As discussed in the introduction to Part II, the distributed systems we focus on in this book are *shared-nothing systems*: i.e., a bunch of machines connected by a network. The network is the only way those machines can communicate—we assume that each

machine has its own memory and disk, and one machine cannot access another machine's memory or disk (except by making requests to a service over the network).

Shared-nothing is not the only way of building systems, but it has become the dominant approach for building internet services, for several reasons: it's comparatively cheap because it requires no special hardware, it can make use of commoditized cloud computing services, and it can achieve high reliability through redundancy across multiple geographically distributed datacenters.

The internet and most internal networks in datacenters (often Ethernet) are *asynchronous packet networks*. In this kind of network, one node can send a message (a packet) to another node, but the network gives no guarantees as to when it will arrive, or whether it will arrive at all. If you send a request and expect a response, many things could go wrong (some of which are illustrated in Figure 8-1):

1. Your request may have been lost (perhaps someone unplugged a network cable).
2. Your request may be waiting in a queue and will be delivered later (perhaps the network or the recipient is overloaded).
3. The remote node may have failed (perhaps it crashed or it was powered down).
4. The remote node may have temporarily stopped responding (perhaps it is experiencing a long garbage collection pause; see "Process Pauses" on page 295), but it will start responding again later.
5. The remote node may have processed your request, but the response has been lost on the network (perhaps a network switch has been misconfigured).
6. The remote node may have processed your request, but the response has been delayed and will be delivered later (perhaps the network or your own machine is overloaded).

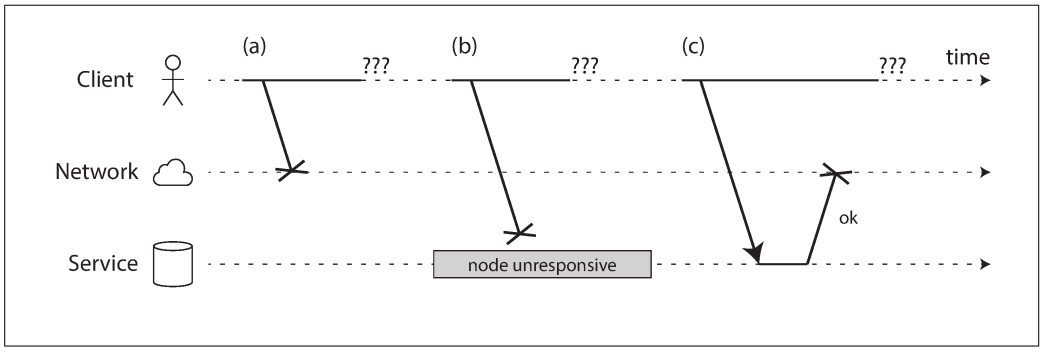


Figure 8-1. If you send a request and don't get a response, it's not possible to distinguish whether (a) the request was lost, (b) the remote node is down, or (c) the response was lost.