O'REILLY®

Designing Data-Intensive Applications

THE BIG IDEAS BEHIND RELIABLE, SCALABLE, AND MAINTAINABLE SYSTEMS

Early Release

RAW & UNEDITED

Martin Kleppmann

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

Editors: Mike Loukides and Ann Spencer
Production Editor: FILL IN PRODUCTION EDITOR

Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition

2015-03-06:	First Early Release
2015-04-01:	Second Early Release
2015-06-18:	Third Early Release
2016-01-26:	Fourth Early Release
2016-05-05:	Fifth Early Release
2016-07-11:	Sixth Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491903094* for release details.

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

978-1-491-90309-4

[FILL IN]

Designing Data-Intensive Applications

Martin Kleppmann

Beijing · Boston · Farnham · Sebastopol · Tokyo

O'REILLY®

Table of Contents

Part I. Foundations of Data Systems

Part II. Distributed Data

Part III. Derived Data

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 towards 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, Dr Dobb's Journal (2012)*

About this Book

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, distributed systems and in the ways we build applications on top of them. There are various driving forces for these developments, including:

- Internet companies such as Google, Yahoo!, Amazon, Facebook, LinkedIn 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 non-relational 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.

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, 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.

We look primarily at the *architecture* of data systems and the ways how they are integrated into data-intensive applications. This book doesn't have space to cover deployment, operations, security, ethics, 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 over-used and under-defined that it is not useful in a serious engineering discussion. This book uses less ambiguous terms, such as single-node vs. distributed systems, or online/interactive vs. offline/batch processing systems.

This book has a bias towards 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 will discuss the fundamental ideas that we need in order to design data-intensive applications. We'll 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 will compare several different data models and query languages, and see how they are appropriate to different situations. In Chapter 3 we will talk about storage engines: how databases arrange data on disk so that you 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 will 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'll first discuss replication (Chapter 5), partitioning/sharding (Chapter 6), and transactions (Chapter 7). We will 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 move up another step and discuss building heterogeneous systems that consist of several different components. As there is no one database for all use cases, applications often need to integrate several different databases, caches, indexes and so on. In Chapter 10 we will start with a batch processing approach, and build upon this in Chapter 11 to describe stream processing. Finally, in ??? we will put everything together and discuss how we can integrate different data systems into reliable, scalable and maintainable applications.

Early Release Status and Feedback

This is an early release copy of *Designing Data-Intensive Applications*. The text, figures and examples are a work in progress, and several chapters are yet to be written. We are releasing the book before it is finished because we hope that it is already useful in its current form, and because we would love your feedback in order to create the best possible finished product.

If you find any errors or glaring omissions, if you find anything confusing, or if you have any ideas for improving the book, please email the author and editors at *feedback@dataintensive.net*.

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 with words like *reliability*, *scalability* and *maintainability*, and how we can try to achieve them.
2. Chapter 2 compares several different data models and query languages — the most visible difference between different 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.

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, *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 which 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. Sometimes it can be hard to combine several 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 1, 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 which 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 data stores that are also used as message queues (Redis), and there are message queues with database-like durability guarantees (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 separate from your main database (such as Elasticsearch or Solr), 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).

When you combine several tools in order to provide a service, the service's interface or 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?

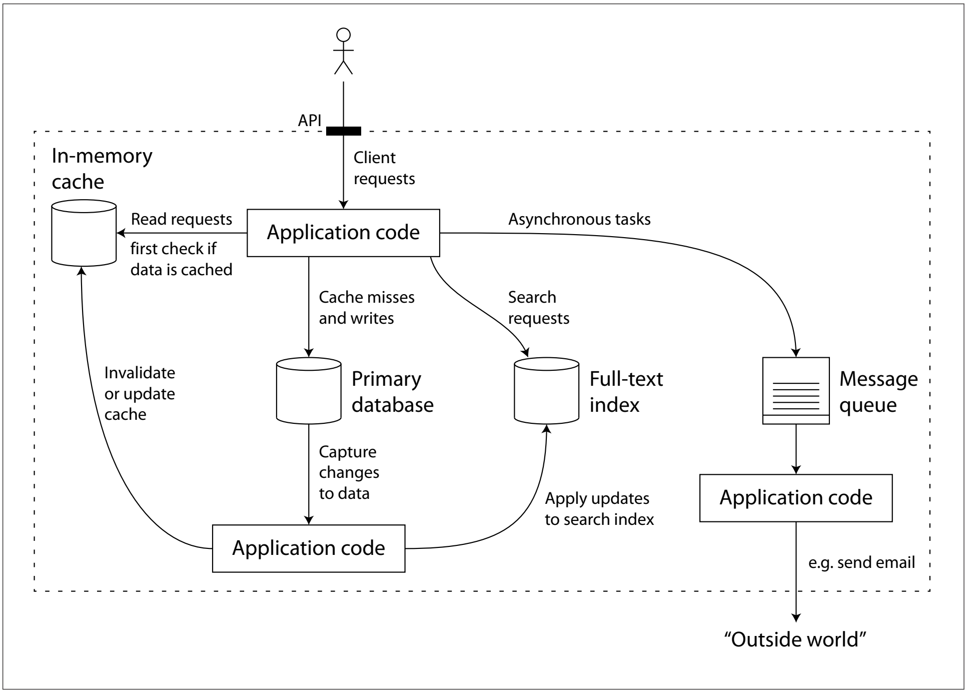


Figure 1-1. One possible architecture for a data system that combines several components.

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 performance) even in the face of *adversity* (hardware or software faults, and even human error). See "Reliability" on page 4.

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 8.

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 16.

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 software 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 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 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 fault*.

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.

Counter-intuitively, 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 fault 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 data centers 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 data centers 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 increase, more applications are using larger numbers of machines, which proportionally increases the rate of hardware faults. Moreover, in some "cloud" platforms such as Amazon Web Services it is fairly common for virtual machine instances to become unavailable without warning [7], as the platform is designed to prioritize flexibility and elasticity[i] over single-machine reliability.

Hence there is a move towards systems that can tolerate the loss of entire machines, by using software fault-tolerance techniques in preference 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.

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 uses up some shared resource — CPU time, memory, disk space or network bandwidth.

[i] Defined in "Approaches for coping with load" on page 15 below.

- A service that the system depends on 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 fault 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 system 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 system 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.
- 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 e-commerce sites can have huge costs in terms of lost revenue and reputation.

Even in "non-critical" applications we have a responsibility to our users. Consider a parent who stores all 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 future. One common reason for degradation is increased load: perhaps it 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 to discuss the question: if the system grows in a particular way, what are our options for coping with the growth? 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: perhaps it’s requests per second to a webserver, 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 (4.6 k requests/sec on average, over 12 k requests/sec at peak).

Home timeline

A user can view tweets recently published by the people they follow (300 k requests/sec).

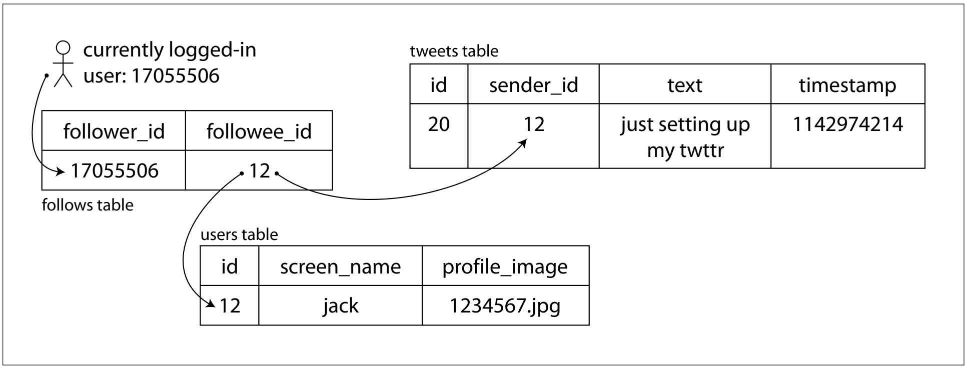
Simply handling 12,000 writes per second (the peak rate for posting tweets) would be fairly easy. However, Twitter’s scaling challenge is not primarily due to tweet volume, but due to *fan-out*[ii] — each user follows many people, and each user is followed by many people. There are broadly two approaches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

1. Posting a tweet simply inserts the new tweet into a global collection of tweets. When a user requests *home timeline*, look up all the people they follow, find all recent tweets for each of those users, and merge them (sorted by time). In a relational database like the one in Figure 1-2, this would be a query along the lines of:

```
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.



id	sender_id	text	timestamp
20	12	just setting up my twttr	1142974214

follower_id	followee_id
17055506	12

id	screen_name	profile_image
12	jack	1234567.jpg

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

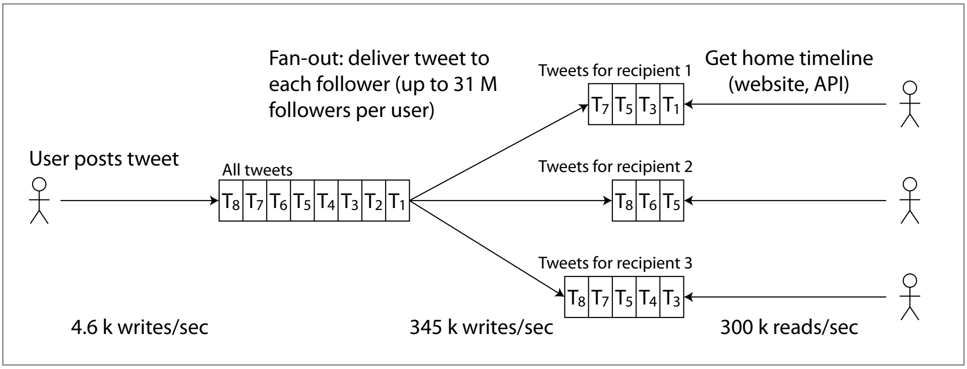


Figure 1-3. Twitter's data pipeline for delivering tweets to followers, with load parameters as of November 2012 [16].

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 4.6 k tweets per second become 345 k 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 5 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 are excepted from this fan-out. Instead, when the home timeline is read, the tweets from celebrities followed by the user are fetched separately and merged with the home timeline when the timeline is read, like in approach 1. This hybrid approach is able to deliver consistently good performance.

Describing performance

Once you have described the load on our 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 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, the *response time* of a service is usually more important — 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 throughput. In practice, the running time is often longer, due to skew (data not being spread evenly across worker processes) or waiting 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 things.

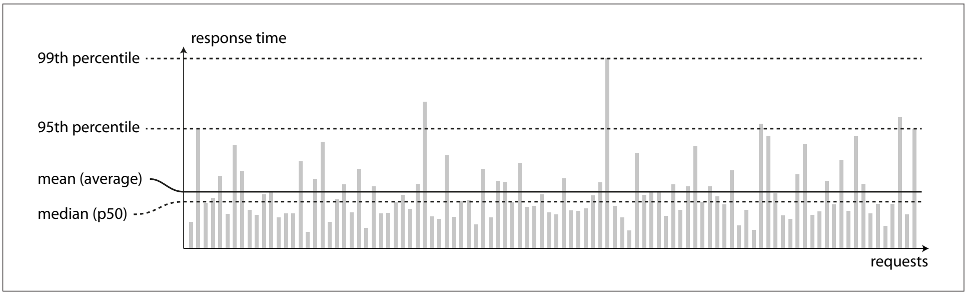


Figure 1-4. Illustrating mean and percentiles: response times for a sample of 100 requests to a service

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 a set of *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 half-way 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 *50th percentile*, and sometimes abbreviated as *p50*. Note that the median is the half-way point for 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* percentile 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 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 made many purchases — i.e. 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 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.

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. For example, a 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. This sets expectations for clients of the service, and allows customers to demand a refund if the SLA is not met.

Queueing delays are often 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 [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 ten 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 which 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].

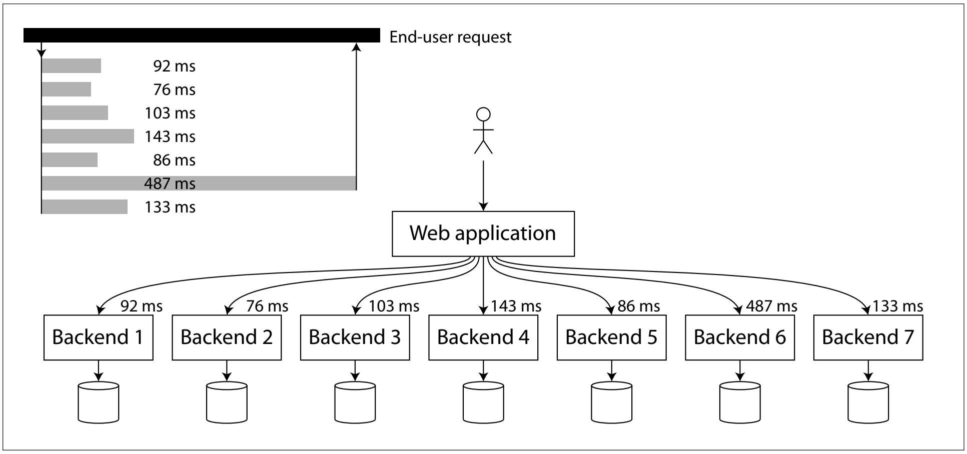


Figure 1-5. When several backend calls are needed to serve a request, it takes just a single slow backend request to slow down the entire end-user request.

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 ten times that load. If you are working on a fast-growing service, it is therefore likely that you will need to re-think your architecture on every order of magnitude load increase — 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, 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 analyses 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 201).

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 application. It is conceivable that distributed data systems will become the default in 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 system, 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 three 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 counter-productive. 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 working 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 in future to make changes to the system, 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 quick answers to 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 does 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 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 effort on high-value activities. Data systems can do various things to make routine tasks easy, including:

- provide visibility into the runtime behavior and internals of the system, with good monitoring;
- good support for automation and integration with standard tools;
- avoid dependency on individual machines (allowing machines to be taken down for maintenance while the system as a whole continues running uninterrupted);
- good documentation and an easy-to-understand operational model ("if I do X, Y will happen");
- 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;
- 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 many 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 re-implementing 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. Much more likely, it is 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 9) 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, when 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, e.g. allow data to be stored, retrieved, searched and processed in various ways), and some *non-functional 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 fault 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 performance. 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 quick answer to making applications reliable, scalable or maintainable. However, there are certain patterns and techniques which keep reappearing in various different kinds of application. In the next few chapters we will take a look at some examples of data systems, and analyze how they work towards 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, 19 July 2011.

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

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

[7] Laurie Voss: "AWS: The Good, the Bad and the Ugly," blog.awe.sm, 18 December 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, 3 July 2012.

[10] Amazon Web Services: "Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region," aws.amazon.com, 29 April 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, 19 March 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, 2 April 2013.

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

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

[17] Martin Fowler: *Patterns of Enterprise Application Architecture*. Addison Wesley, November 2002. ISBN: 978-0321127426

[18] Kelly Sommers: "After all that run around, what caused 500ms disk latency even when we replaced physical server? Slight unbalance in rack causing vibration." twitter.com, 13 November 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," at *Stanford University Data Mining Class* (CS345), December 2006.

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

[22] Jake Brutlag: "Speed matters for Google web search," googleresearch.blogspot.co.uk, 22 June 2009.

[23] Tyler Treat: "Everything You Know About Latency Is Wrong," bravenewgeek.com, 12 December 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), pages 138–149, 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, 7 December 2015.

[29] James Hamilton: “On Designing and Deploying Internet-Scale Servicesx,” 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: 9780201835953

[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), pages 327–330, July 2008. doi:10.1109/COMPSAC.2008.50

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 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 which 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 list of layers above). 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 of layers above).

Relational Model vs. 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* (in SQL: tables), where each relation is an unordered collection of *tuples* (rows).

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 (RDBMS) and SQL had become the tool 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 bank 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, e-commerce, 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 term *NoSQL* is unfortunate, since it doesn't actually refer to any particular technology — it was intended simply as a catchy Twitter hashtag for a meetup on open source, distributed, non-relational 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 re-interpreted 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 of 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 non-relational data stores — 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), 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_info` 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, which allow 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]. PostgreSQL also has vendor-specific extensions for JSON, array datatypes, and more [8].
- Encode jobs, education and contact info as a JSON or XML document, store it on a text column in the database, and to 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.

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

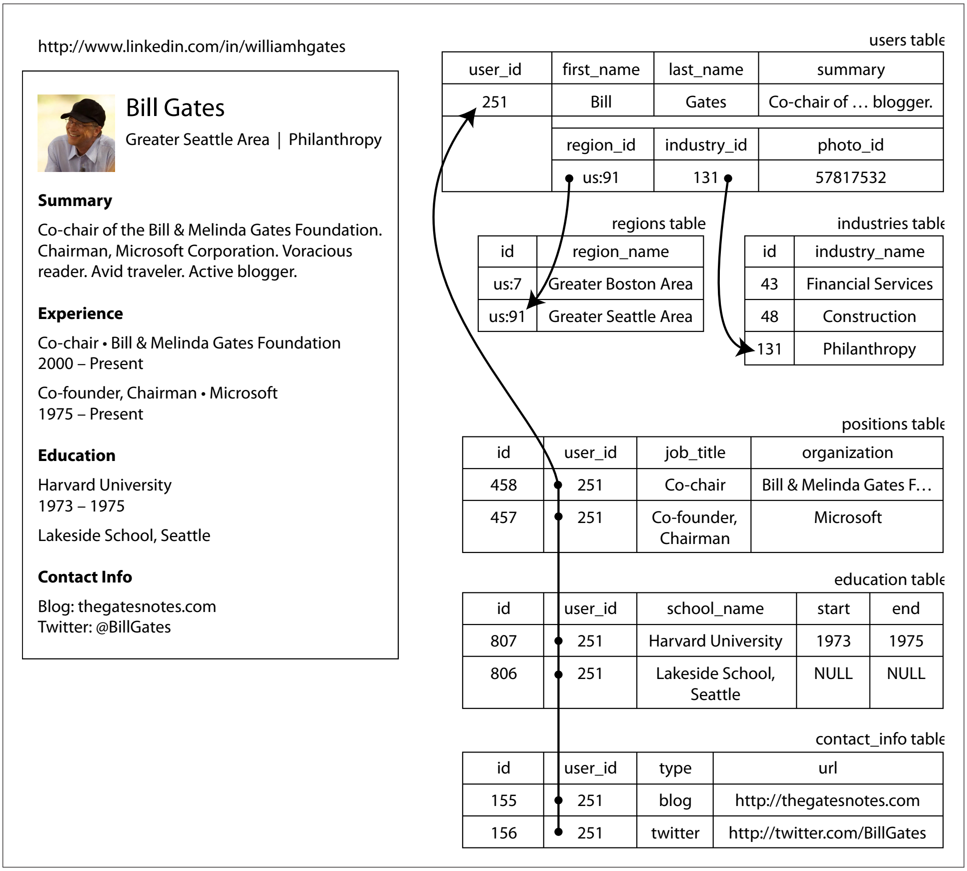
[i] A term borrowed from electronics. Every electric circuit has a certain impedance (resistance to alternating current) on its inputs and outputs. When you connect one circuit's output to another one's input, the power transfer across the connection is maximized if the output and input impedances of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

being much simpler than XML. Document-oriented databases like MongoDB [9], RethinkDB [10], CouchDB [11] and Espresso [12] support this data model.

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 its 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).



http://www.linkedin.com/in/williamhgates

Bill Gates

Greater Seattle Area | Philanthropy

Summary

Co-chair of the Bill & Melinda Gates Foundation. Chairman, Microsoft Corporation. Voracious reader. Avid traveler. Active blogger.

Experience

Co-chair • Bill & Melinda Gates Foundation
2000 – Present

Co-founder, Chairman • Microsoft
1975 – Present

Education

Harvard University
1973 – 1975

Lakeside School, Seattle

Contact Info

Blog: thegatesnotes.com
Twitter: @BillGates

users table

user_id	first_name	last_name	summary
251	Bill	Gates	Co-chair of … blogger.
	region_id	industry_id	photo_id
	us:91	131	57817532

regions table

id	region_name
us:7	Greater Boston Area
us:91	Greater Seattle Area

industries table

id	industry_name
43	Financial Services
48	Construction
131	Philanthropy

positions table

id	user_id	job_title	organization
458	251	Co-chair	Bill & Melinda Gates F…
457	251	Co-founder, Chairman	Microsoft

education table

id	user_id	school_name	start	end
807	251	Harvard University	1973	1975
806	251	Lakeside School, Seattle	NULL	NULL

contact_info table

id	user_id	type	url
155	251	blog	http://thegatesnotes.com
156	251	twitter	http://twitter.com/BillGates

Figure 2-1. Representing a LinkedIn profile using a relational schema.

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"
  }
}
```

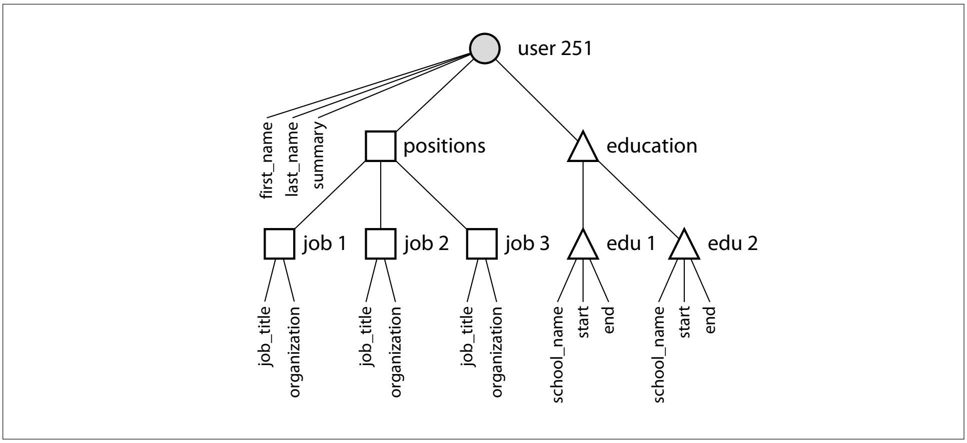


Figure 2-2. One-to-many relationships forming a tree structure.

Many-to-one and many-to-many relationships

In Example 2-1 above, `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,
- The name is stored only in one place, so it is easy to update across the board if it ever needs to be changed (for example, change of a city name due to political events),
- When the site is translated into other languages, the standardized lists can be localized, and so the region and industry can be displayed in the viewer's language,
- Better search — for example, 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 future — and if that information is duplicated, all the redundant copies need to be updated. That incurs overhead on writes, and risks inconsistencies (some copies of the information are updated but others aren't). Removing such duplication is the key idea behind *normalization* in databases.[ii]

> Database administrators and developers love to argue about normalization and denormalization, but we will suspend judgement 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

[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.

[iii] At the time of writing, joins are supported in RethinkDB, not supported in MongoDB, and only supported in pre-declared views in CouchDB.

added to applications. For example, consider some changes we could make to the résumé example:

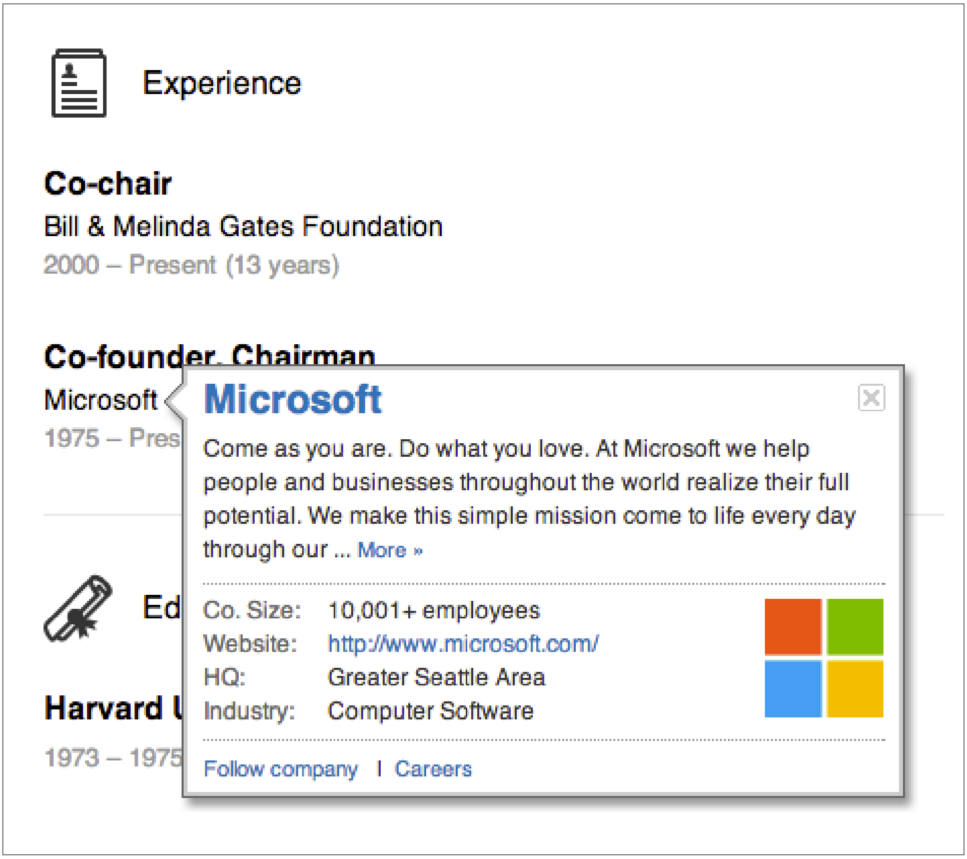
Organizations and schools as entities

In the description above, `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 can link to the organizations and schools that it mentions, and include their logos and other information (see Figure 2-3 for example).

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.

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.



Experience

Co-chair

Bill & Melinda Gates Foundation

2000 – Present (13 years)

Co-founder, Chairman

Microsoft

1975 – Pres

Ed

Harvard U

1973 – 1975

Microsoft

Come as you are. Do what you love. At Microsoft we help people and businesses throughout the world realize their full potential. We make this simple mission come to life every day through our ... More »

Co. Size:	10,001+ employees
Website:	http://www.microsoft.com/
HQ:	Greater Seattle Area
Industry:	Computer Software

Follow company | Careers

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

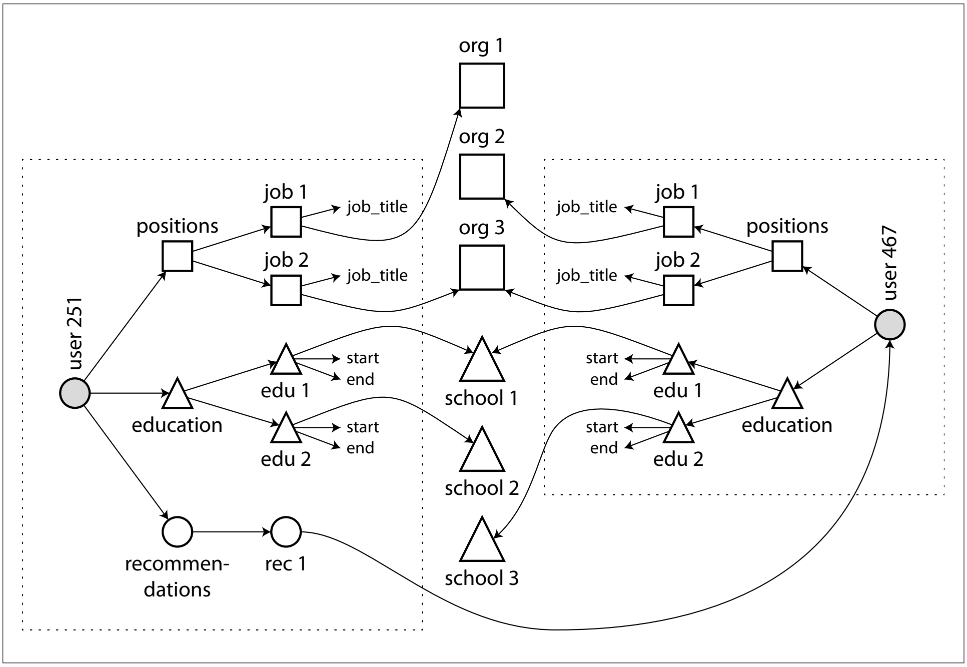


Figure 2-4. Extending résumés with many-to-many relationships.

Are document databases repeating history?

While many-to-many relationships and joins are routinely used in relational databases without thinking twice, 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), a database system 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 above.

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 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 *Conference on Data Systems Languages* (CODASYL), and implemented by several different database vendors, so it is also known as the *CODASYL model* [16].

The CODASYL model is 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 can have multiple parents. For example, there could be one record for the `"Greater Seattle Area"` region, and every user who lives in that region could be linked to it. This allows many-to-one and many-to-many relationships to be modeled.

The links between records in the network model are 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 hand-written 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 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 hand-code 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

[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.

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 vs. 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 of data model.

The main arguments in favor of the document data model are: for some applications it is closer to the data structures used by the application, schema flexibility, and better performance due to locality. The relational model counters by providing better support for joins, 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 probably 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 very awkward, the relational model is acceptable, and graph models (see "Graph-like Data Models" on page 48) 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 data conforms to it) [21].

Schema-on-read is similar to dynamic (run-time) 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 when 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 want to change it to store 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 which 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 — with the exception of MySQL, which copies the entire table on `ALTER TABLE`, which can mean minutes or even hours of downtime when altering a large table. Various tools exist to work around this limitation of MySQL [24, 25].

Running the `UPDATE` statement on a large table is likely to be slow on any database, since every row needs to be re-written. 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 data is heterogeneous, i.e. the items in the collection don't all have the same structure for some reason, 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, or
- 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 re-written — 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 [26]. Oracle allows the same, using a feature called *multi-table index cluster tables* [27]. The *column-family* concept in the Bigtable data model (used in Cassandra and HBase) has a similar purpose in managing locality [28].

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] and IBM DB2 since version 10.5 [29] 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 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 future.

[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: a value in a row doesn't have to just be a primitive datatype like a number or a string, but it could also be a nested relation (table) — and 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.

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:

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

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 it 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 above, 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 [30]. 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, but 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 [31].

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 a `<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` above. 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 code 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 clicked on a different page), the blue color won't be removed, even if the code is re-run — and so the item will remain highlighted until the 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, 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 [32]. A limited form of MapReduce is supported by some NoSQL data stores, 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 have been 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;
```

[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].

❶ The `date_trunc('month', timestamp)` function takes a timestamp and rounds it down to the nearest start of calendar 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:

```
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 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` function 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 re-run 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 [33].

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 *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's expressiveness is similar 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 object: *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, edges indicate which people know each other.

The web graph
  Vertices are web pages, 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, the shortest path in a road network is useful for routing, and PageRank on the web graph can be used to determine the popularity of a web page, and thus its ranking in search results.

In the examples above, 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 object in a single data store. For example, Facebook maintains a single graph with many different types of vertex and edge: 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, etc. [34]

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.

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, 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 [35] and graph processing frameworks like Pregel (see Chapter 10).

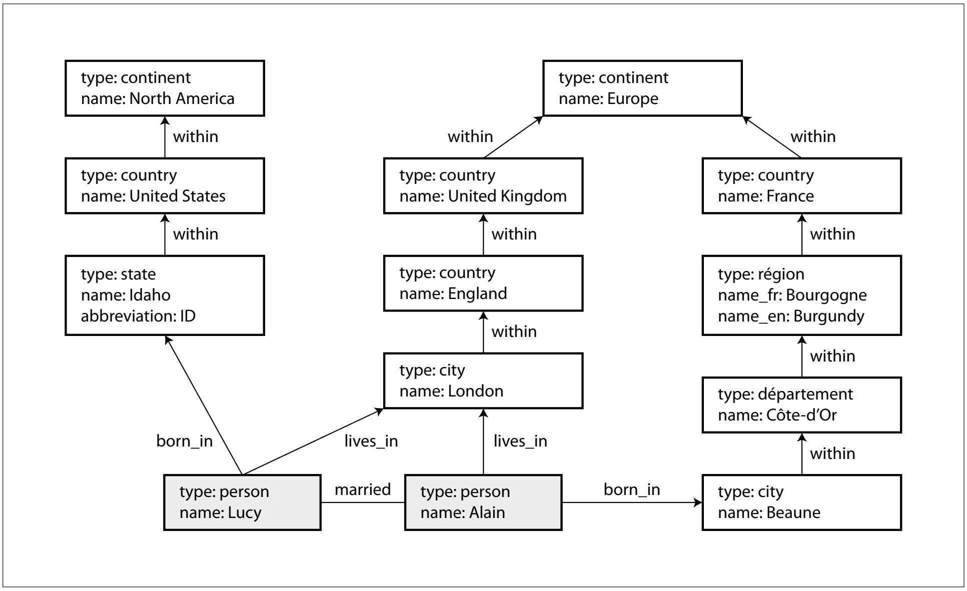


Figure 2-5. Example of graph-structured data (boxes represent vertices, arrows represent edges).

Property graphs

In the property graph model, each vertex consists of:

- a unique identifier,
- a set of outgoing edges,
- a set of incoming edges, and
- 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, and
- 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 on 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 — follow a path through a chain of vertices — both forwards and backwards. (That's why Example 2-2 has indexes on both the `tail_vertex` and the `head_vertex` columns.)
3. By using different labels for different kinds of relationship, 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 can 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 [36]. (It is named after a character in the movie *The Matrix*, and is not related to ciphers in cryptography [37].)

Example 2-3 shows the Cypher query to insert the left-hand 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 labelled `WITHIN`, with `Idaho` as tail node and `USA` as 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 precise: 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 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 labelled 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 above 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 backwards. 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 on one of the locations.

As 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, and 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, i.e. 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.

This idea of variable-length traversal paths in a query can be expressed since SQL: 1999 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 by 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 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 add it to 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.

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 four 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.

Graph databases compared to the network model

In “Are document databases repeating history?” on page 35 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 a property 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 position of the new record in these sets. In a graph database, there is no defined ordering of vertices or edges (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.

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. The object can be 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. The object can be 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.

For example, 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*) [38].

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 [39] is a triple-store that does not claim to have anything to do with the semantic web.[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 — why don't they also publish information as machine-readable data for computers to read? The *Resource Description Framework* (RDF) [40] 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 over-hyped 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 [41] can automatically convert between different RDF formats if necessary.

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>
```

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

```
          </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: rather than a predicate being just `WITHIN` or `LIVES_IN`, it is actually something like `<http://my-company.com/namespace#within>` or `<http://my-company.com/namespace#lives_in>`. The idea behind this: 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 getting confused by `http://` URLs, we use URIs like `urn:example:within` in the examples above. 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 [42]. (It is an acronym for *SPARQL Protocol and RDF Query Language*, and pronounced "sparkle".) It predates Cypher, and since Cypher's pattern-matching is borrowed from SPARQL, they look quite similar [36].

The same query as before — finding people who 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 with the `name` property set to `"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.

The foundation: Datalog

Datalog is a much older language than SPARQL or Cypher, having been studied extensively by academics in the 1980s [43, 44]. It is less well-known among software engineers, but it is nevertheless important, because it provides the foundation that later query languages build upon.

Every year we have more cores within each CPU, more CPUs in each machine, and more machines in a networked cluster. As programmers we still haven't figured out a good answer to the problem of how to best work with all that parallelism. But it has been suggested that declarative languages based on Datalog may be the future for parallel programming, which has rekindled interest in Datalog recently [31]. There has also been a lot of research into evaluating Datalog queries efficiently [43].

In practice, Datalog is used in a few data systems: for example, it is the query language of Datomic [39] and Cascalog [45] is a Datalog implementation for querying large datasets in Hadoop.[viii]

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.

[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 makes no functional difference.

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 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.

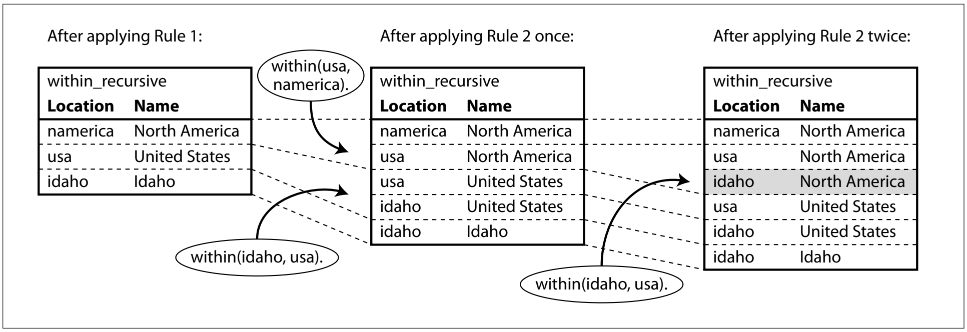
In rules, words that start with an uppercase letter are variables, and predicates are matched like in Cypher and SPARQL. For example, `name(Location, Name)` matches the triple `name(namerica, 'North America')` with variable bindings `Location = namerica` and `Name = 'North America'`.

A rule applies if the system can find a match for *all* predicates on the right-hand side of the `:-` operator. When the rule applies, it's as though the left-hand 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, and generates `within_recursive(namerica, 'North America')`.
2. `within(usa, namerica)` exists in the database, and `within_recursive(namerica, 'North America')` was generated by the previous step, so rule 2 applies, and generates `within_recursive(usa, 'North America')`.
3. `within(idaho, usa)` exists in the database, and `within_recursive(usa, 'North America')` was generated by the previous step, so rule 2 applies, and 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 is illustrated in Figure 2-6.



After applying Rule 1:

within_recursive

Location	Name
namerica	North America
usa	United States
idaho	Idaho

After applying Rule 2 once:

within_recursive

Location	Name
namerica	North America
usa	North America
usa	United States
idaho	United States
idaho	Idaho

After applying Rule 2 twice:

within_recursive

Location	Name
namerica	North America
usa	North America
idaho	North America
usa	United States
idaho	United States
idaho	Idaho

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 Cypher and SPARQL queries above.

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 non-relational "NoSQL" data stores 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, 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.

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 above can handle this kind of usage, which is why researchers have written specialized genome database software like GenBank [46].

- 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 scale, custom solutions are required, to stop the hardware cost spiraling out of control [47].
- *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*, Fourth Edition, MIT Press, pages 2–41, 2005. ISBN: 9780262693141

[3] Pramod J Sadalage and Martin Fowler: *NoSQL Distilled*. Addison-Wesley, August 2012. ISBN: 9780321826626

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

[5] James Phillips: "Surprises in our NoSQL adoption survey," blog.couchbase.com, 8 February 2012.

[6] Michael Wagner: *SQL/XML:2006 – Evaluierung der Standardkonformität ausgewählter Datenbanksysteme*. Diplomica Verlag, Hamburg, 2010. ISBN: 978-3-8366-4609-3

[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," www.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, 11 November 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, Cambridge University Press, pages 19–56, 1984. ISBN: 0521254302

[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, 30 October 2013.

[20] Martin Fowler: "Schemaless data structures," martinfowler.com, 7 January 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] 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.

[27] Donald K Burleson: "Reduce I/O with Oracle cluster tables," dba-oracle.com.

[28] 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.

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

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

[31] 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.

[32] 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.

[33] Craig Kerstiens: "JavaScript in your Postgres," postgres.heroku.com, 5 June 2013.

[34] 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.

[35] "Gremlin graph traversal language," TinkerPop, gremlin.tinkerpop.com, 2013.

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

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

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

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

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

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

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

[43] 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

[44] 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

[45] Nathan Marz: "Cascalog," cascalog.org.

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

[47] Fons Rademakers: "ROOT for Big Data Analysis," at *Workshop on the future of Big Data management*, London, UK, June 2013.

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 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 expore that distinction later in the chapter, in "Transaction Processing or Analytics?" on page 87, and in "Column-oriented storage" on page 93 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. Many databases internally use a *log*,[i] an append-only data file, quite similar to what db_set does. Real databases have more issues to deal with (such as concurrency control, reclaiming disk space so that the log doesn't grow forever, handling errors, partially written records, and so on) but the basic principle is the same. Logs are incredibly useful, and we will come back to them later.

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. But the general idea behind them is: 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 is 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.

[i] 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.

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 example above. 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. This is 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.

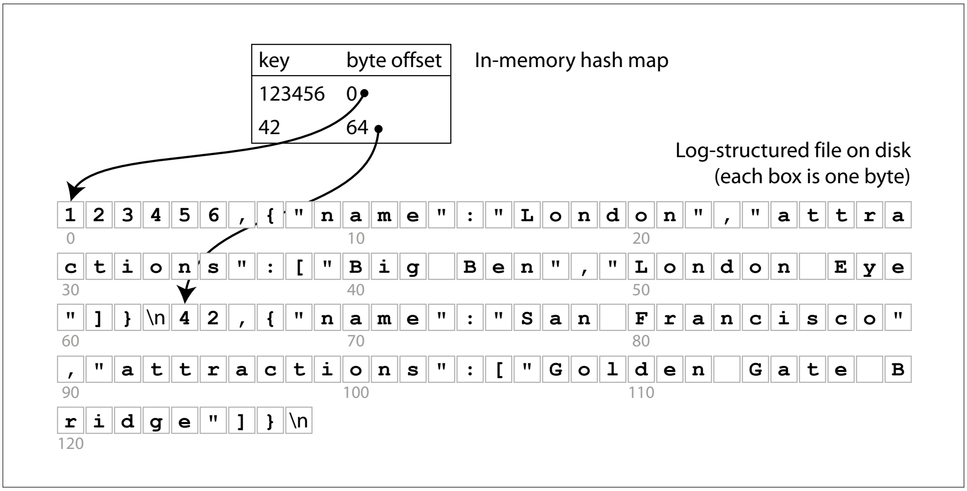

key
byte offset
123456
0
42
64
In-memory hash map
Log-structured file on disk
(each box is one byte)
1 2 3 4 5 6 , { " n a m e " : " L o n d o n " , " a t t r a
0
10
20
c t i o n s " : [" B i g B e n " , " L o n d o n E y e
30
40
50
"] } \n 4 2 , { " n a m e " : " S a n F r a n c i s c o "
60
70
80
, " a t t r a c t i o n s " : [" G o l d e n G a t e B
90
100
110
r i d g e "] } \n
120

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 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 in 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 (so that it's feasible to keep all keys in memory), i.e. you have a large number of writes per key.

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, and to 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.

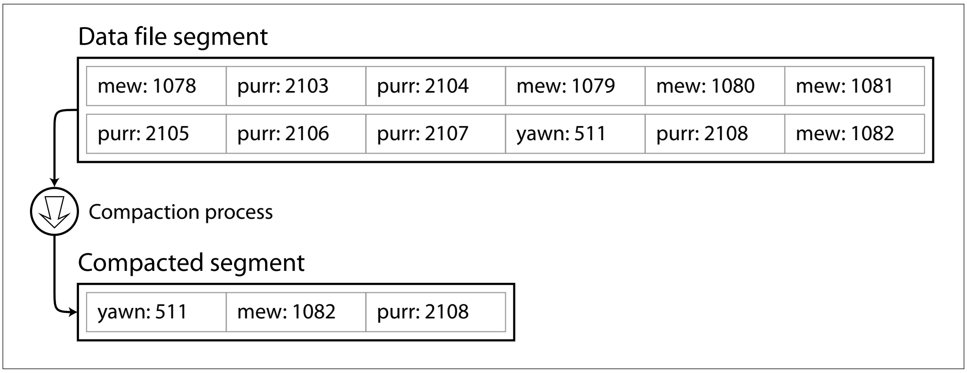


Figure 3-2. Compaction of a key-value update log (counting the number of times each cat video was played), retaining only the most recent value for each key.

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 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.

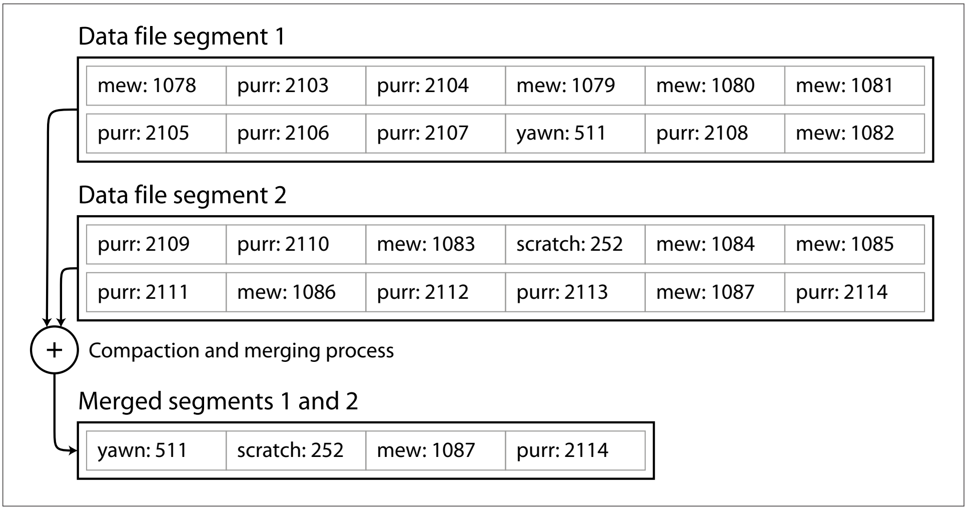


Figure 3-3. Performing compaction and segment merging simultaneously.

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, check the second-most-recent segment, etc. 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. To briefly mention some of the issues that are important in a real implementation:

File format
: CSV is not the best format for a log. It's faster and simpler to use a binary format which 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 quicker.

Partially written records
: The database may crash at any time, including halfway through appending a record to the log. Bitcask files include checksums which allow 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 concurrently read 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. This performance difference applies both to traditional spinning-disk hard drives and to flash-based *solid state drives* (SSDs) [4].
- 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 problems 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 fetch the values for 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 merging 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.

   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 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. So 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).

   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.[ii]

[ii] 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.

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. Nowadays, disk bandwidth is usually a worse bottleneck than CPU, so it is worth spending a few additional CPU cycles to reduce the amount of data you need to write to and read from disk.

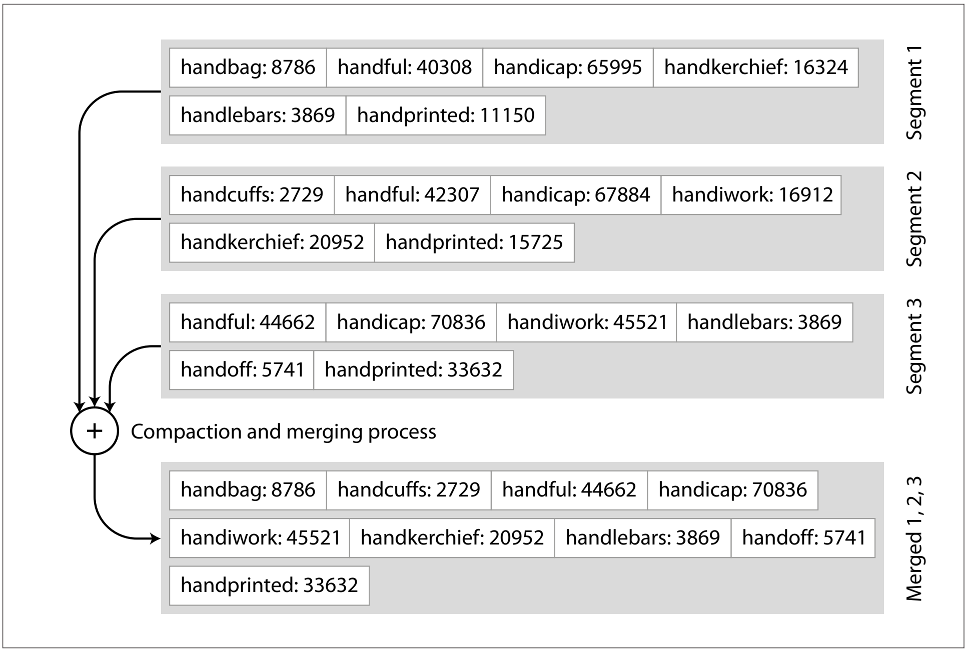


Figure 3-4. Merging several Sorted String Table (SSTable) segments, retaining only the most recent value for each key.

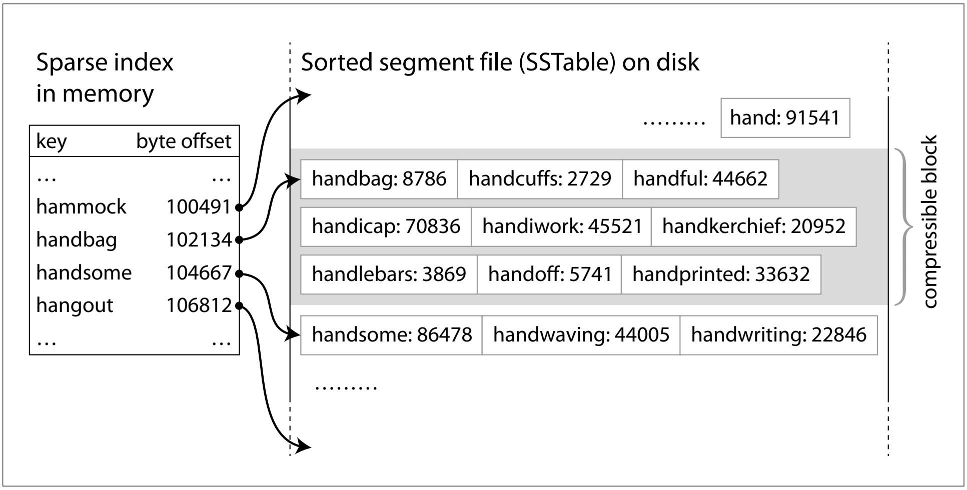


Figure 3-5. A Sorted String Table (SSTable) with in-memory index.

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 next section), 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. When the new SSTable is ready, the memtable can be emptied.
- 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.

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 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* (*LSM-Tree*) [10], building on earlier work on log-structured file systems [11].

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 at its core is 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 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].

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, LevelDB maintains 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 non-existent keys.)

However, the basic idea — keeping a cascade of SSTables that are merged in the background — is simple and effective. Even when the dataset is much bigger than 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

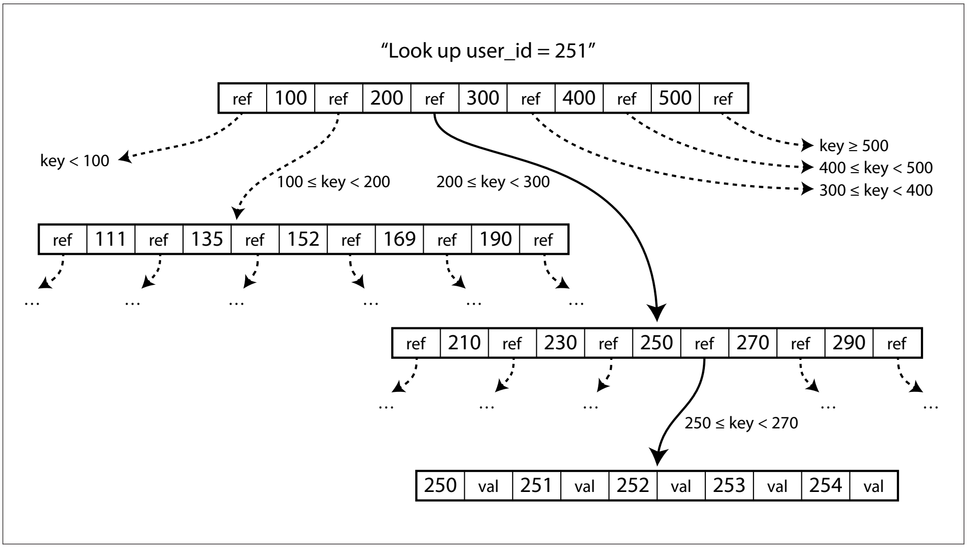
The log-structured indexes we have discussed so far are gaining acceptance, but they are not the most common type of index. The most widely-used indexing structure is quite different: the *B-tree*.

Introduced in 1970 [16] and called "ubiquitous" less than 10 years later [17], B-trees have stood the test of time very well. They remain the standard index implementation in almost all relational databases, and many non-relational 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, and read or write one page at a time. This 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 this to construct a tree of pages, as illustrated in Figure 3-6.



"Look up user_id = 251"
ref 100 ref 200 ref 300 ref 400 ref 500 ref
key < 100
100 ≤ key < 200
200 ≤ key < 300
key ≥ 500
400 ≤ key < 500
300 ≤ key < 400
ref 111 ref 135 ref 152 ref 169 ref 190 ref
...
ref 210 ref 230 ref 250 ref 270 ref 290 ref
250 ≤ key < 270
250 val 251 val 252 val 253 val 254 val

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 k keys and $k + 1$ references to child pages (in Figure 3-6, $k = 5$, but in reality k would typically be in the hundreds). Each child is responsible for a continuous range of keys, and the keys in the root page indicate where the boundaries between those ranges lie.

In the example of 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 which further breaks down the 200–300 range into sub-ranges. Eventually we are 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 each value can be found.

If you want to update the value for an existing key in a B-tree, you search for the leaf page containing that key, and change the value 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.[iii]

This algorithm ensures that the tree remains *balanced*: a B-tree with *n* keys always has a height of *O*(log *n*). Even if the tree is very large, you don't need to follow many page references to find the page you are looking for. (If you're familiar with Red-Black trees or 2-3 trees, B-trees are very similar, except for the larger branching factor.)

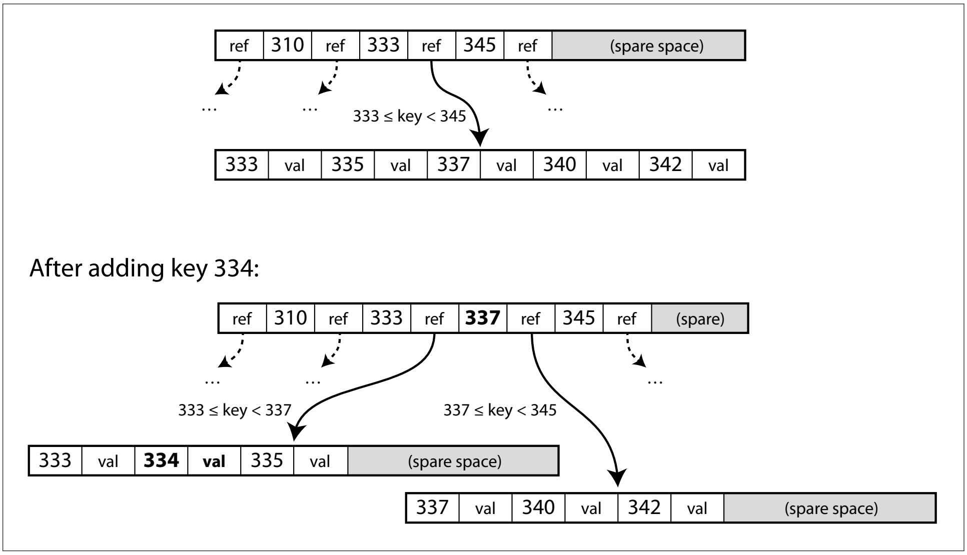


Figure 3-7. Growing a B-tree by splitting a page.

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

Update-in-place vs. append-only logging

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, but it is similarly slow [18].

Moreover, some operations require several different pages to be overwritten. For example, if you split a page because an insertion caused it to be over-full, 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 writing only some of the pages, you end up with a corrupted index (e.g. there may be an *orphan* page which is not a child of any parent).

In order to make the database resilient to crashes, it is normal for B-tree implementations to include an additional data structure on disk: a *write-ahead log* (WAL, also known as *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, 19].

A B-tree index must therefore write every piece of data at least twice: once to the log, and once to the tree page itself (and perhaps again as pages are split). On the other hand, log-structured indexes also re-write data multiple times due to repeated background merging.[iv] It's not clear whether B-trees or LSM-trees are better in this regard — it depends on the workload and the tuning of the storage engine. In the end, there is no alternative to benchmarking systems with your particular workload.

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 back-

[iv] This effect — one write to the database resulting in multiple writes to the disk — is known as *write amplification*, and is of particular concern on SSDs, which can only overwrite blocks a limited number of times before wearing out.

ground 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 [20]. A modified page is written to a different location, and a new version of parent pages in the tree is created, pointing at the new location. This is also useful for concurrency control, as we shall see in "Snapshot isolation and repeatable read" on page 228.
- 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.[v]
- In general, pages could 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 nearby 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* [21] borrow some log-structured ideas to reduce disk seeks (and they have nothing to do with fractals).

Comparing B-trees to LSM-trees

B-tree implementations are generally more mature than LSM-tree implementations, but LSM-trees are very promising due to their performance characteristics.

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

LSM-trees are typically able to sustain much higher throughput of random writes compared to B-trees, because they turn all random writes into sequential writes on the underlying device. This makes them appealing for applications with high write throughput. As a rule of thumb, LSM-trees are typically faster for writes, whereas B-trees are thought to be faster for reads [22]. Actual benchmarks are often inconclusive and sensitive to the details of the workload [23].

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 11) the response time of queries to log-structured storage engines can sometimes be quite high, and B-trees can be more predictable [24].

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 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 in 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` command, 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 two ways: either by making each value in the index a list of matching row identifiers (like a posting 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 could 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) [25]. In SQL Server, you can specify one clustered index per table [26].

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 [27]. This allows some queries to be answered by using the index alone (in which case, the index is said to *cover* the query) [26].

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 north and south pole), 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 [28]. 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 [29]. 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 multidimensional indexes are not just for geographic locations. For example, on an e-commerce 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 [30].

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.

To mention just one example, 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) [31].

As mentioned in "SSTables and LSM-trees" on page 74, Lucene uses a SSTable-like structure for its term dictionary. This structure requires a small in-memory index which 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* [32]. This automaton can be transformed into a *Levenshtein automaton*, which supports efficient search for words within a given edit distance [33].

Other fuzzy search techniques go in the direction of document classification and machine learning. See an information retrieval textbook for more detail [34].

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. Both with 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, potentially 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, which is still unusual), 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.

Vendors such as VoltDB, MemSQL and Oracle TimesTen are in-memory databases with a relational model, and they claim that they can offer big performance improvements by removing all the overheads associated with managing on-disk data structures [35, 36]. 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) [37]. Redis and Couchbase provide weak durability by writing to disk asynchronously.

Counter-intuitively, 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 [38].

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. By keeping 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 memory, without bringing back the overheads of a disk-centric architecture [39]. 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 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 (like the Bitcask example at the beginning of the chapter).

At present, in-memory databases are still a fairly new technology, but they are worth keeping an eye on in 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 ACID 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, or actions in a game, or 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, 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 much 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) [40].[vi] The difference between OLTP

[vi] 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.

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, for example 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.

A *data warehouse*, by contrast, is a separate database that analysts can query to their heart's content, without affecting OLTP operations [41]. 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.

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.

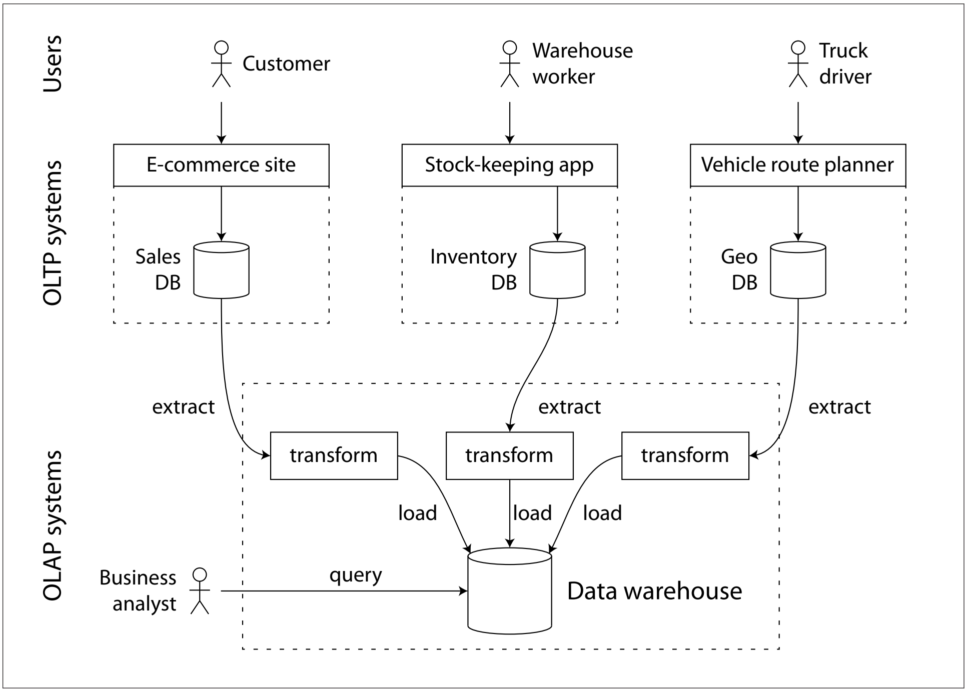


Figure 3-8. Simplified outline of ETL into a data warehouse.

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 which 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 [42, 43, 44].

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, AMPLab's Shark, Cloudera Impala, Hortonworks Stinger, Facebook Presto, Apache Tajo and Apache Drill [45, 46]. Some of them are based on ideas from Google's Dremel [47].

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* [48]).

An 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* (here, `fact_sales`). Each row of the fact table represents an event that occurred at a particular time (here, each row represents a product that was purchased by a customer). If we were analyzing website traffic rather than retail sales, each row may represent a page view or a click by a user.

Usually, facts are captured as individual events, because this allows maximum flexibility of analysis later on. 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 [49].

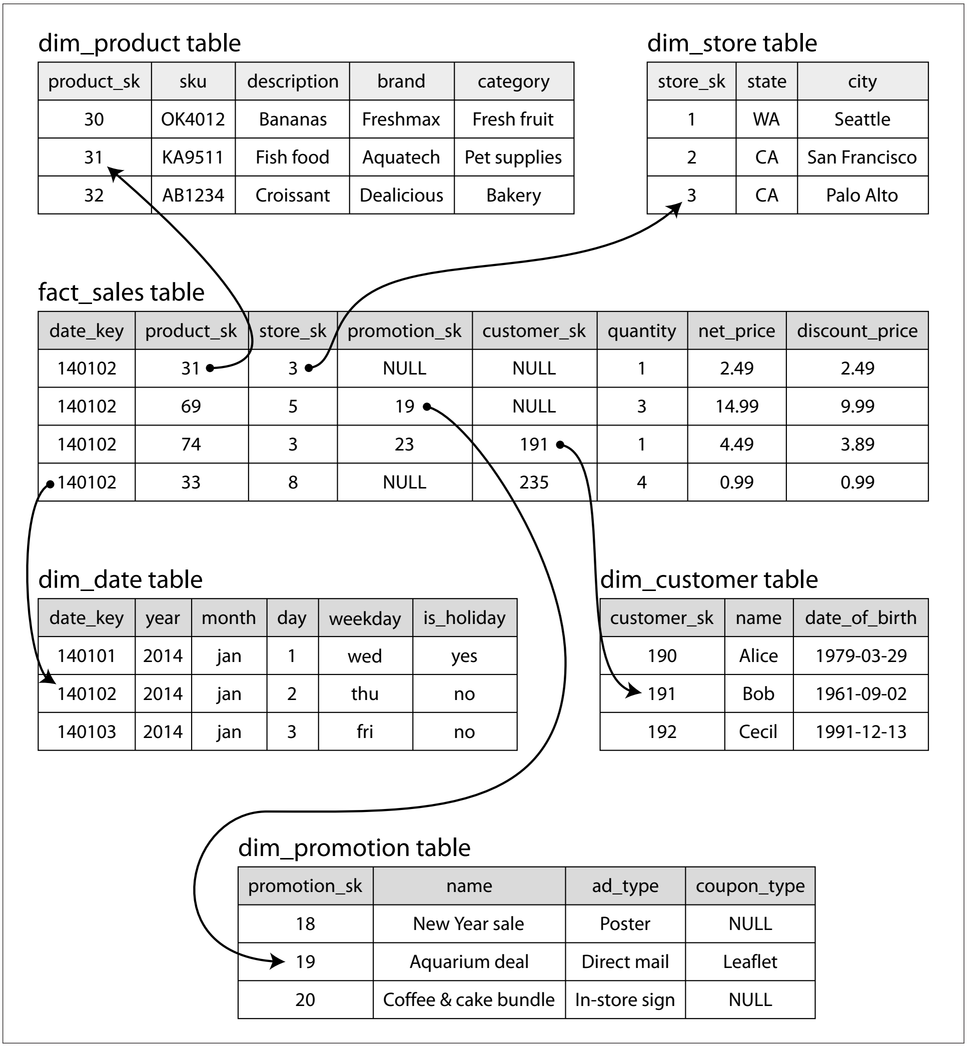
Some of the columns in the fact table are attributes, such as the price at which the product was sold, and the cost of buying it from the supplier (allowing the profit margin to be calculated). Other columns in the fact table are foreign key references to other tables, called *dimension* tables. As each 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 foreign keys to dimension tables like rays of a star.

A variation of this template is known as *snowflake schema*, where dimensions are further broken down into sub-dimensions. For example, there could be separate dimension 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 in this case, star schemas are often preferred because they are simpler for analysts to work with [48].



dim_product table

product_sk	sku	description	brand	category
30	OK4012	Bananas	Freshmax	Fresh fruit
31	KA9511	Fish food	Aquatech	Pet supplies
32	AB1234	Croissant	Dealicious	Bakery

dim_store table

store_sk	state	city
1	WA	Seattle
2	CA	San Francisco
3	CA	Palo Alto

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	31	3	NULL	NULL	1	2.49	2.49
140102	69	5	19	NULL	3	14.99	9.99
140102	74	3	23	191	1	4.49	3.89
140102	33	8	NULL	235	4	0.99	0.99

dim_date table

date_key	year	month	day	weekday	is_holiday
140101	2014	jan	1	wed	yes
140102	2014	jan	2	thu	no
140103	2014	jan	3	fri	no

dim_customer table

customer_sk	name	date_of_birth
190	Alice	1979-03-29
191	Bob	1961-09-02
192	Cecil	1991-12-13

dim_promotion table

promotion_sk	name	ad_type	coupon_type
18	New Year sale	Poster	NULL
19	Aquarium deal	Direct mail	Leaflet
20	Coffee & cake bundle	In-store sign	NULL

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

In a typical data warehouse, tables are often very wide: fact tables often have over 100 columns, sometimes several hundred [44]. Dimension tables can also be very wide, as they include all relevant metadata that may be relevent 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) [44]. 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`. All other columns are ignored by the query.

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

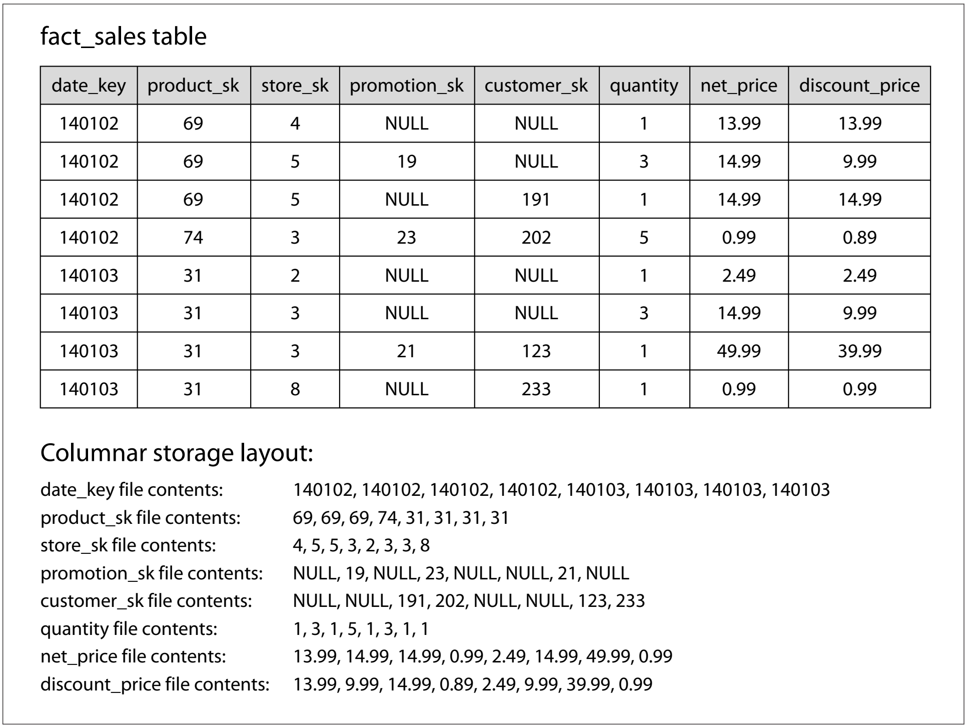
```
SELECT
  dim_date.weekday, dim_product.category,
  SUM(fact_sales.quantity) AS quantity_sold
FROM fact_sales
  JOIN dim_date    ON fact_sales.date_key   = dim_date.date_key
  JOIN dim_product ON fact_sales.product_sk = dim_product.product_sk
WHERE
  dim_date.year = 2013 AND
  dim_product.category 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`, which 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 is illustrated in Figure 3-10.



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.

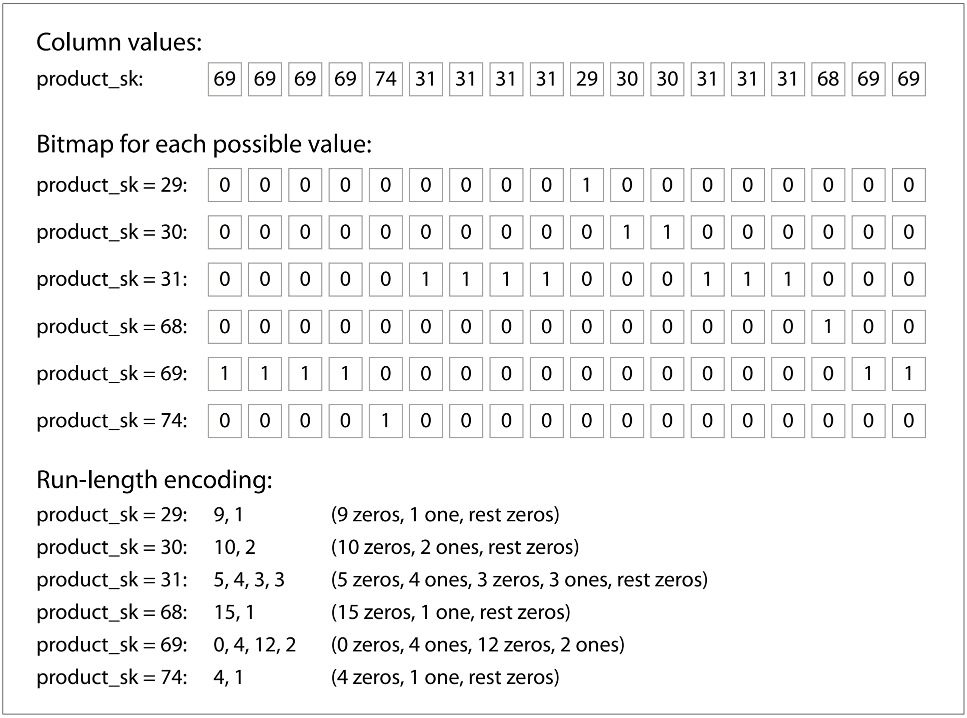
The column-oriented storage layout relies on each column containing the rows in the same order. Thus, if you need to reassemble an entire row, you can take the 23rd entry from each of the individual column files, and put them together to form the 23rd row of the table.

Column storage is easiest to understand in a relational data model, but it applies equally to non-relational data. For example, Parquet [50] is a columnar storage format for a document data model, based on Google's Dremel [47].

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.

In Figure 3-10, see the sequences of values for each column: 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 a *bitmap encoding*, illustrated in Figure 3-11.



Column values:

product_sk:	69	69	69	69	74	31	31	31	31	29	30	30	31	31	31	68	69	69

Bitmap for each possible value:

product_sk = 29:	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
product_sk = 30:	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
product_sk = 31:	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0
product_sk = 68:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
product_sk = 69:	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
product_sk = 74:	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Run-length encoding:

product_sk = 29:	9, 1	(9 zeros, 1 one, rest zeros)
product_sk = 30:	10, 2	(10 zeros, 2 ones, rest zeros)
product_sk = 31:	5, 4, 3, 3	(5 zeros, 4 ones, 3 zeros, 3 ones, rest zeros)
product_sk = 68:	15, 1	(15 zeros, 1 one, rest zeros)
product_sk = 69:	0, 4, 12, 2	(0 zeros, 4 ones, 12 zeros, 2 ones)
product_sk = 74:	4, 1	(4 zeros, 1 one, rest zeros)

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 kind of queries that are common in a data warehouse:

`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 [51] for an overview.

Memory bandwidth and pipelined execution

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 using CPU cycles efficiently: 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 [52, 53].

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. This is 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 above, can be designed to operate on such chunks of compressed column data directly. This technique is known as *vectorized processing* [51, 42].

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 would be much faster than scanning all rows.

A second column can determine the sort order of any rows which 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 which 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 key 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 [54, 55]. 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 [55].

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

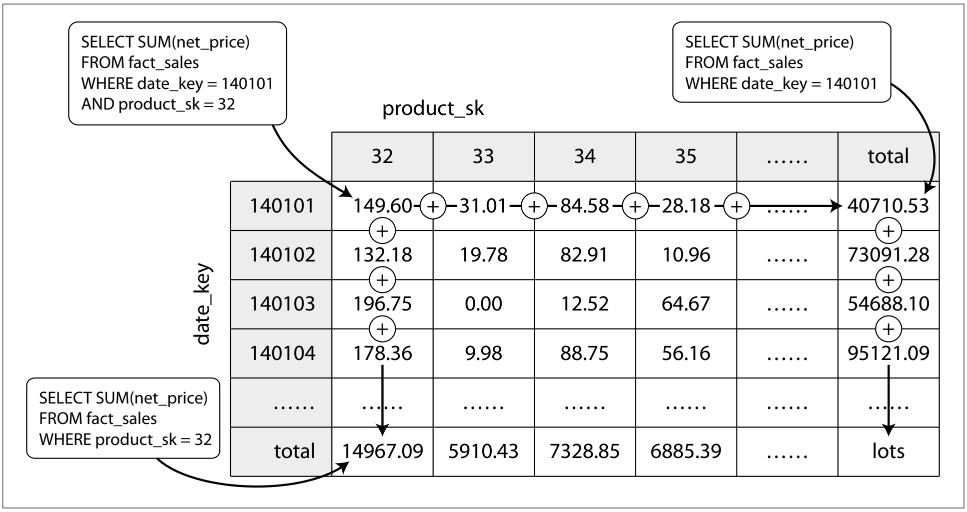
Not every data warehouse is necessarily a column store: traditional row-oriented databases and a few other architectures are also used. However, columnar storage can be significantly faster for ad-hoc analytical queries, so it is rapidly gaining popularity [44, 56].

Another aspect of data warehouses that is worth mentioning briefly is *materialized aggregates*. As discussed above, 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 are used most often by queries?

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* [57]. It is a grid of aggregates grouped by different dimensions. Figure 3-12 shows an example.



```
SELECT SUM(net_price)
FROM fact_sales
WHERE date_key = 140101
AND product_sk = 32
```

```
SELECT SUM(net_price)
FROM fact_sales
WHERE date_key = 140101
```

```
SELECT SUM(net_price)
FROM fact_sales
WHERE product_sk = 32
```

	32	33	34	35	......	total
140101	149.60	31.01	84.58	28.18	......	40710.53
140102	132.18	19.78	82.91	10.96	......	73091.28
140103	196.75	0.00	12.52	64.67	......	54688.10
140104	178.36	9.98	88.75	56.16	......	95121.09
......	......	......	......	......	......	......
total	14967.09	5910.43	7328.85	6885.39	......	lots

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 pre-computed. 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 it 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 the questions of data storage and retrieval. What happens when you store some 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. 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 fixed-size pages which can be overwritten. B-trees are the biggest example of this philosophy, being used in all major relational databases and also many non-relational 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: 0-201-00023-7

[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, Qiong Luo, and Ke Yi: "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," hivedata.com, 26 November 2013.

[8] Matteo Bertozzi: "Apache HBase I/O – HFile," blog.cloudera.com, 29 June 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 2013.

[13] Deepak Kandepet: "Hacking Lucene - the Index Format," hackerlabs.org, 1 October 2011.

[14] Michael McCandless: "Visualizing Lucene's segment merges," blog.mikemccandless.com, 11 February 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] 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.

[17] Douglas Comer: "The Ubiquitous B-tree," *ACM Computing Surveys*, volume 11, number 2, pages 121–137, June 1979. doi:10.1145/356770.356776

[18] Emmanuel Goossaert: "Coding for SSDs," codecapsule.com, 12 February 2014.

[19] 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), pages 371–380, June 1992. doi:10.1145/130283.130338

[20] Howard Chu: "LDAP at Lightning Speed," at *BuildStuffLT*, Vilnius, Lithuania, November 2014.

[21] Bradley C Kuszmaul: "A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees," tokutek.com, 22 April 2014.

[22] 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

[23] Kevin Tseng: "LevelDB Benchmarks," leveldb.googlecode.com, July 2011.

[24] Michi Mutsuzaki: "MySQL vs. LevelDB," github.com, August 2011.

[25] *MySQL 5.7 Reference Manual*. Oracle, 2014.

[26] *Books Online for SQL Server 2012*. Microsoft, 2012.

[27] Joe Webb: "Using Covering Indexes to Improve Query Performance," simple-talk.com, 29 September 2008.

[28] 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.

[29] The PostGIS Development Group: "PostGIS 2.1.2dev Manual," postgis.net, 2014.

[30] 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

[31] Michael McCandless: "Lucene's FuzzyQuery is 100 times faster in 4.0," blog.mikemccandless.com, 24 March 2011.

[32] 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

[33] 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

[34] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze: *Introduction to Information Retrieval*. Cambridge University Press, 2008. ISBN: 0521865719

[35] 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), pages 1150–1160, September 2007.

[36] "VoltDB Technical Overview White Paper," VoltDB, 2014.

[37] 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.

[38] 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

[39] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik: "Anti-Caching: A New Approach to Database Management System Archi-

tecture,” *Proceedings of the VLDB Endowment*, volume 6, number 14, pages 1942–1953, September 2013.

[40] Edgar F Codd, S B Codd, and C T Salley: “Providing OLAP to User-Analysts: An IT Mandate,” E.F. Codd Associates, 1993.

[41] 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

[42] 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.

[43] 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.

[44] Michael Stonebraker: “The Traditional RDBMS Wisdom is (Almost Certainly) All Wrong,” presentation at *EPFL*, May 2013.

[45] Daniel J Abadi: “Classifying the SQL-on-Hadoop Solutions,” hadapt.com, 2 October 2013.

[46] 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.

[47] 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.

[48] Ralph Kimball and Margy Ross: *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd edition. John Wiley & Sons, July 2013. ISBN: 978-1118530801

[49] Derrick Harris: “Why Apple, eBay, and Walmart have some of the biggest data warehouses you’ve ever seen,” gigaom.com, 27 March 2013.

[50] Julien Le Dem: “Dremel made simple with Parquet,” blog.twitter.com, 11 September 2013.

[51] Daniel J Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden: “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

[52] 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.

[53] 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

[54] 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.

[55] 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.

[56] Julien Le Dem and Nong Li: "Efficient data storage for analytics with Apache Parquet 2.0," at *Hadoop Summit*, San Jose, June 2014.

[57] 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

CHAPTER 4

Encoding and Evolution

> Everything changes and nothing stands still.
>
> —Heraclitus of Ephesus, *As quoted by Plato in Cratylus (360 BC)*

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 19).

In most cases, a change of application 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 now 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 *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, REST and 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 preclude 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, e.g. certain memory-mapped files, or operating directly on compressed data (as in "Column compression" on page 94).

[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 criticised for being too verbose and unnecessarily complicated [9]. JSON's popularity is mainly due to its built-in support by 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, 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 a 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.

- 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, as above) depends on information in the schema, applications that don't use XML/JSON schemas need to potentially include additional code to encode/decode data correctly.
- 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 it 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. 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, 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 object field names within the encoded data. That is, in a binary encoding of the JSON document Example 4-1 they still 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 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 3 fields (bottom four bits = `0x03`). (In case you're wondering what happens if an object has more than 15 fields, so 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 8 bytes long (bottom four bits = `0x08`).
3. The next 8 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. 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.

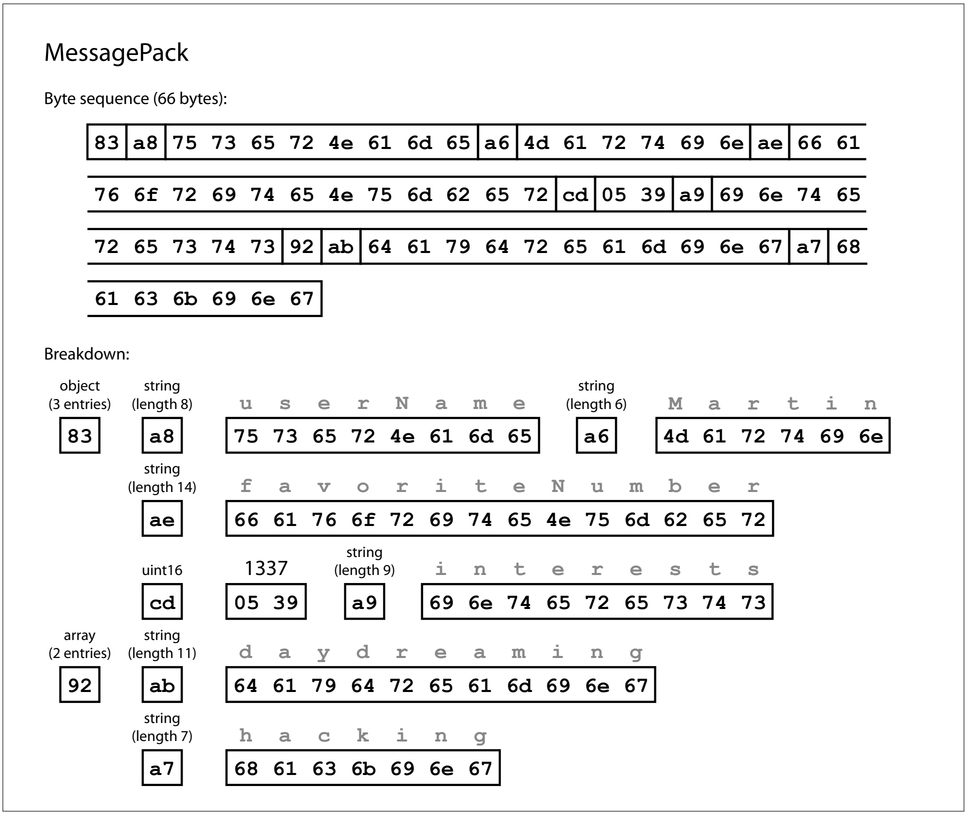


Figure 4-1. Example record (Example 4-1) encoded using MessagePack.

Thrift and Protocol Buffers

Apache Thrift [15] and Protocol Buffers [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 of 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 one above, 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].

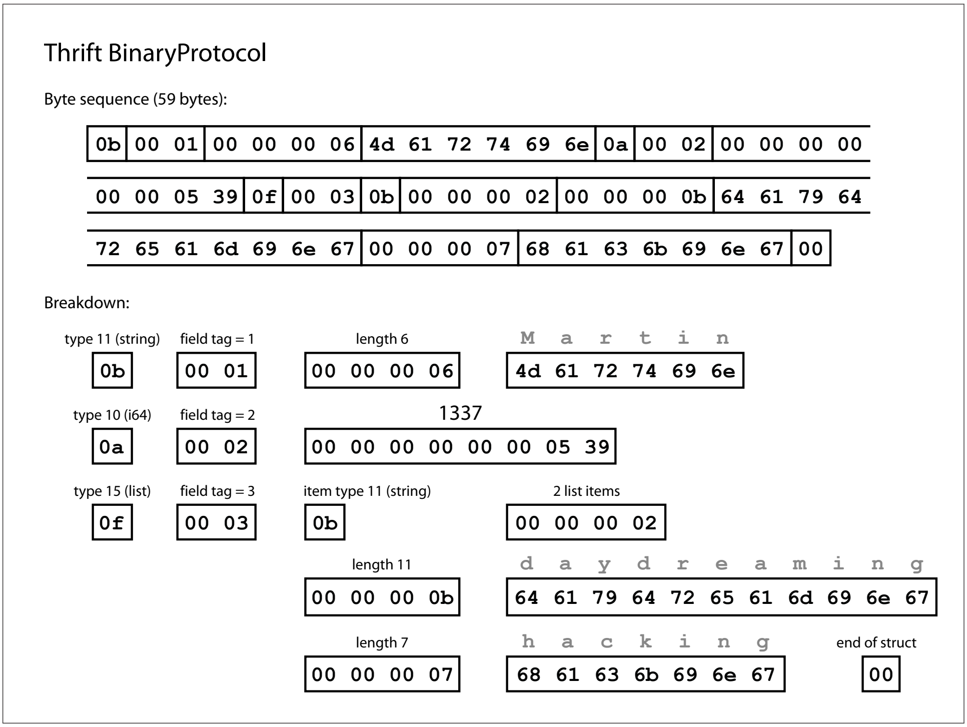


Figure 4-2. Example record encoded using Thrift's BinaryProtocol.

[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 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 above. 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 8 bytes for the number 1337, it is encoded in two bytes, using the top bit of each byte 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.

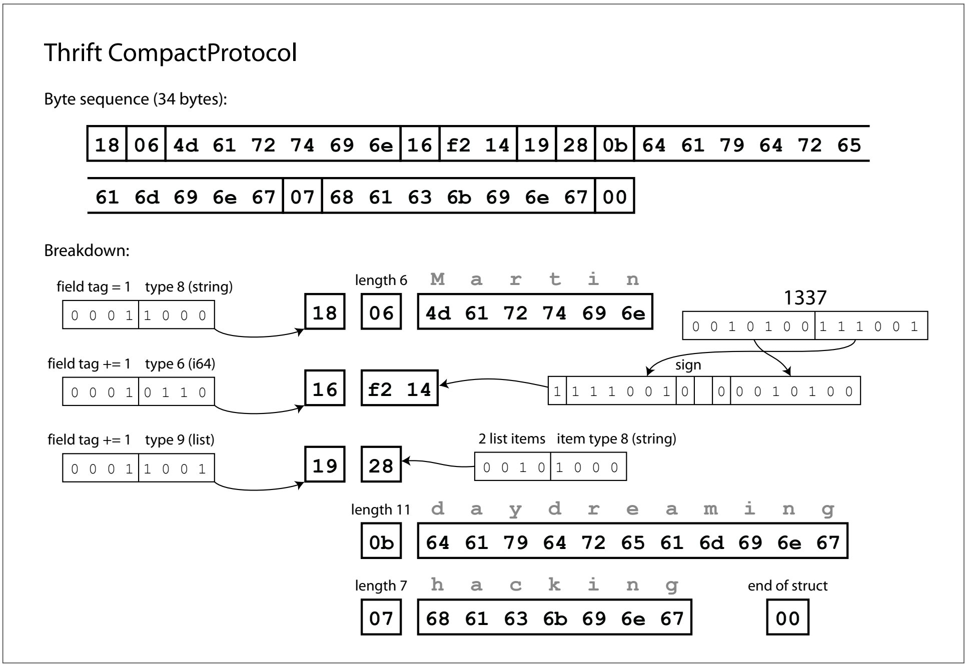


Figure 4-3. Example record encoded using Thrift's CompactProtocol.

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.

One detail to note: in the schemas above, 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.

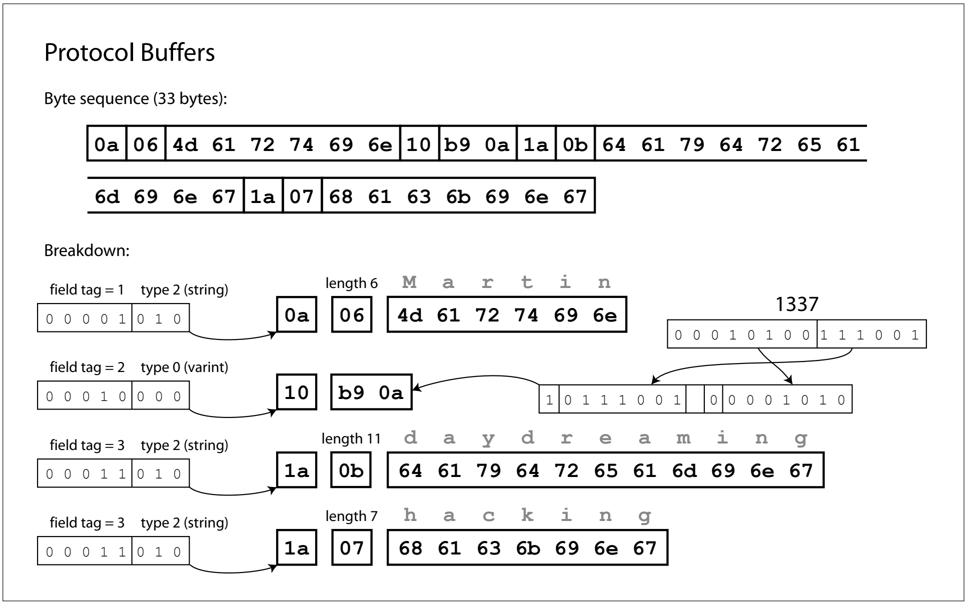


Figure 4-4. Example record encoded using Protocol Buffers.

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 schemas above), 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 new a 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 reads data written by old code, because the old code did not write 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).

Data types 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 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 zero. 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 ok 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 sub-project 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.

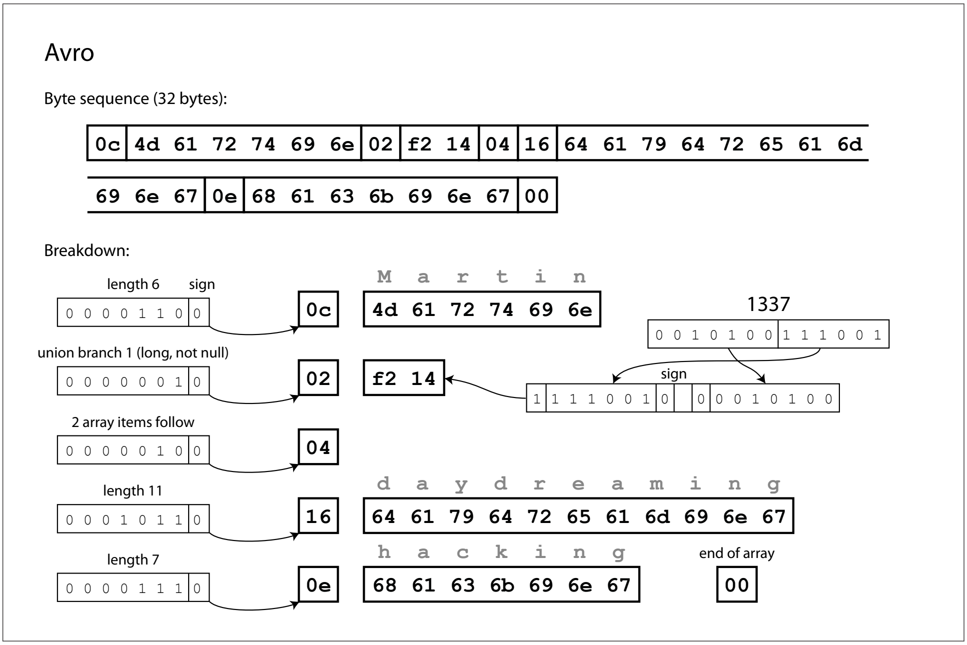


Figure 4-5. Example record encoded using Avro.

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 CompactProtocol).

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?

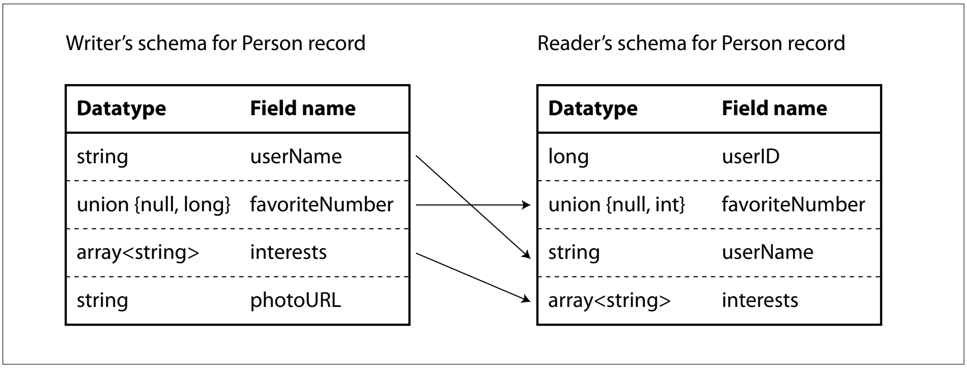
The writer's schema and the reader's schema

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 that 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 a field on the writer's side doesn't have a matching field on the reader's side, it is ignored. If a field on the reader's side doesn't have a matching field on the writer's side, it is filled in with a default value declared in the reader's schema.



Writer's schema for Person record

Datatype	Field name
string	userName
union {null, long}	favoriteNumber
array<string>	interests
string	photoURL

Reader's schema for Person record

Datatype	Field name
long	userID
union {null, int}	favoriteNumber
string	userName
array<string>	interests

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 the Avro schema above 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, because 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 which we 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.

[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.

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 database. 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 continue using that schema for the lifetime of the connection. The Avro RPC protocol (see "Data flow through services: REST and RPC" on page 127) does 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 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 more friendly 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 above) 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 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 compliation 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 is especially useful in conjunction with dynamically typed data processing languages like Apache Pig [26]. In Pig, you can just open some Avro files and start analyzing them 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 Buffers 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 based on schemas. 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) which 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 Data Flow

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 how data can flow from one process to another. Who encodes the data, and who decodes it? In the rest of this chapter we will unpack some of the most common ways how data flows between processes:

- via databases (see "Data flow through databases" on page 125),
- via calls to services (see "Data flow through services: REST and RPC" on page 127), and
- via asynchronous message-passing (see "Message passing data flow" on page 132).

Data flow 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 above 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 re-encode 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.

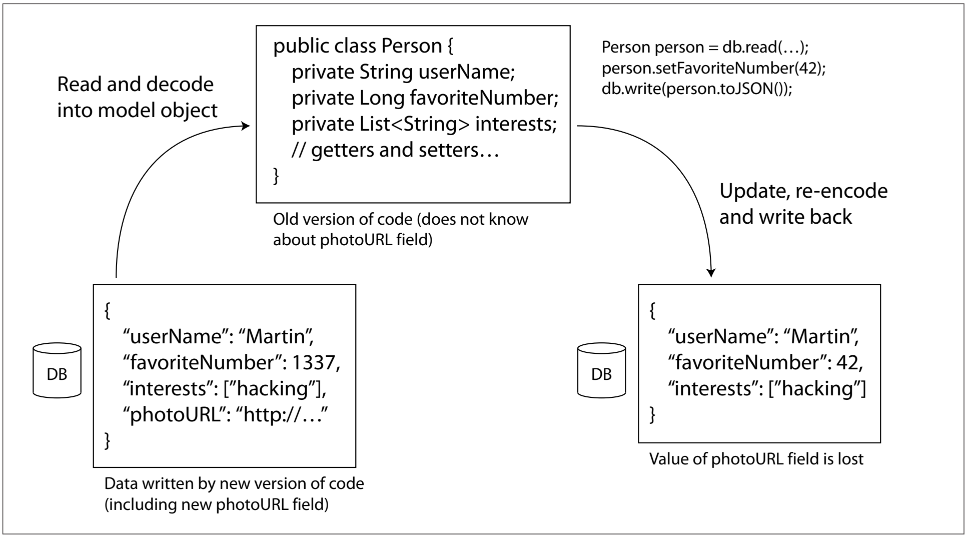

Read and decode
into model object
public class Person {
private String userName;
private Long favoriteNumber;
private List<String> interests;
// getters and setters…
}
Old version of code (does not know about photoURL field)
Person person = db.read(…);
person.setFavoriteNumber(42);
db.write(person.toJSON());
Update, re-encode
and write back
DB
{
"userName": "Martin",
"favoriteNumber": 1337,
"interests": ["hacking"],
"photoURL": "http://…"
}
Data written by new version of code (including new photoURL field)
DB
{
"userName": "Martin",
"favoriteNumber": 42,
"interests": ["hacking"]
}
Value of photoURL field is lost

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 nulls for any columns that are missing from the encoded data on

[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.

disk. LinkedIn's document database Espresso uses Avro for storage, allowing it to use Avro's schema evolution rules [23].

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 88). 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 94).

In Chapter 10 we will talk more about using data in archival storage.

Data flow 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 a 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 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 approach has traditionally been called a *service-oriented architecture* (SOA), more recently refined and rebranded as *microservices architecture* [31, 32].

A key design goal of such an 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:

1. A client application running on a user's device (e.g. 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 SOA/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, for example 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 [33, 34]. 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 [35] 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 [36].

The API of a SOAP web service is described using an XML-based language called WSDL (the RESTful equivalent of WSDL is called Swagger [37]). 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 123).

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.

Remote procedure calls (RPC)

Web services are merely the latest incarnation of a long lineage 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 Remote Method Invocation

[vi] Even within each camp there are plenty of arguments. For example, HATEOAS (*hypermedia as the engine of application state*), often provokes discussions [34].

[vii] Despite the similarity of acronyms, SOAP is not a requirement for SOA. SOAP is a particular technology, whereas SOA is a general approach towards building systems.

(RMI) are limited to Java. DCOM is limited to Microsoft platforms. CORBA is excessively complex, and does not provide backward or forward compatibility [40].

All of these are based on the idea of a *Remote Procedure Call* (RPC), which has been around since the 1970s [41]. RPC 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 is called *location transparency*). Although this seems convenient at first, the approach is fundamentally flawed [42, 43]. 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 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.
- 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, and 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 sequence of bytes that can be sent over the network. That's ok 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 110). 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 a 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 [44]. gRPC supports *streams*, where a call consists of not just one request and one response, but a series of requests and responses over time [45].

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 205.

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 (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 servers and clients of a RPC can be changed and deployed independently. Compared to data flowing through databases (in the last section), we can make a simplifying assumption in the case of data flow 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 [46].
- 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 a 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 [47]. 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 [48].

Message passing data flow

We have been looking at the different ways how 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 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 is what makes it *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 by 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* or *subscribers* of that queue or topic. There can be many producers and many consumers on the same topic.

A topic provides only one-way data flow. 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 re-publishes 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 with threads directly (and the associated problems of race conditions, locking and deadlock), logic is encapsulated in *actors*. Each actor 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 sender and recipient are on the same node or on 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 this 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 [49].
- *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 [50, 51]. Like with Akka, custom serialization plugins 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 [52]. An experimental new `maps` datatype (a JSON-like structure, introduced in Erlang R17 in 2014) may make this easier in future [53].

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 affects 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 which 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 discussed 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 schema 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 data flow, 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, Feb 2015.

[4] "EsotericSoftware/kryo," github.com, Oct 2014.

[5] "CWE-502: Deserialization of Untrusted Data," Common Weakness Enumeration, cwe.mitre.org, 30 July 2014.

[6] Steve Breen: "What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability," foxglovesecurity.com, 6 November 2015.

[7] Patrick McKenzie: "What The Rails Security Issue Means For Your Startup," kalzumeus.com, 31 January 2013.

[8] Eishay Smith: "jvm-serializers wiki," github.com, Nov 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, 19 October 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, 17 September 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, 5 December 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, 21 January 2015.

[24] Jay Kreps: "Putting Apache Kafka to use: a practical guide to building a stream data platform (part 2)," blog.confluent.io, 25 February 2015.

[25] Gwen Shapira: "The problem of managing schemas," radar.oreilly.com, 4 November 2014.

[26] "Apache Pig 0.14.0 Documentation," pig.apache.org, November 2014.

[27] John Larmouth: *ASN.1 Complete*. Morgan Kaufmann, 1999. ISBN: 0-12233-435-3

[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, 21 September 2010.

[30] Jesse James Garrett: "Ajax: A New Approach to Web Applications," adaptivepath.com, 18 February 2005.

[31] Sam Newman: *Building Microservices*. O'Reilly Media, February 2015. ISBN: 978-1-4919-5035-7

[32] Chris Richardson: "Microservices: Decomposing Applications for Deployability and Scalability," infoq.com, 25 May 2014.

[33] Roy Thomas Fielding: "Architectural Styles and the Design of Network-based Software Architectures," PhD Thesis, University of California, Irvine, 2000.

[34] Roy Thomas Fielding: "REST APIs must be hypertext-driven," roy.gbiv.com, 20 October 2008.

[35] "REST in peace, SOAP," royal.pingdom.com, 15 October 2010.

[36] "Web Services Standards as of Q1 2007," innoq.com, February 2007.

[37] "Swagger RESTful API documentation specification," swagger.io, 8 September 2014.

[38] Pete Lacey: "The S stands for Simple," wanderingbarque.com, 15 November 2006.

[39] Stefan Tilkov: "Interview: Pete Lacey Criticizes Web Services," infoq.com, 12 December 2006.

[40] Michi Henning: "The Rise and Fall of CORBA," *ACM Queue*, volume 4, number 5, pages 28–34, June 2006. doi:10.1145/1142031.1142044

[41] 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

[42] 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.

[43] Steve Vinoski: "Convenience Over Correctness," *IEEE Internet Computing*, volume 12, number 4, pages 89–92, July 2008. doi:10.1109/MIC.2008.75

[44] Marius Eriksen: "Your Server as a Function," at *7th Workshop on Programming Languages and Operating Systems* (PLOS), November 2013. doi:10.1145/2525528.2525538

[45] "grpc-common Documentation," Google Inc., github.com, February 2015.

[46] Aditya Narayan and Irina Singh: "Designing and versioning compatible Web services," ibm.com, 28 March 2007.

[47] Troy Hunt: "Your API versioning is wrong, which is why I decided to do it 3 different wrong ways," troyhunt.com, 10 February 2014.

[48] "API upgrades," Stripe, Inc., April 2015.

[49] Jonas Bonér: "Upgrade in an Akka Cluster," email to akka-user mailing list, 28 August 2013.

[50] Philip A Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin: "Orleans: Distributed Virtual Actors for Programmability and Scalability," Microsoft Research Technical Report MSR-TR-2014-41, March 2014.

[51] "Microsoft Project Orleans Documentation," Microsoft Research, dotnet.github.io, 2015.

[52] David Mercer, Sean Hinde, Yinso Chen, and Richard A O'Keefe: "beginner: Updating Data Structures," email thread on erlang-questions mailing list, October 2007.

[53] Fred Hebert: "Postscript: Maps," learnyousomeerlang.com, 9 April 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 have servers at various locations worldwide, so that users can be served from a datacenter that is geographically close to them. That avoids the user 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 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 is super-linear: a machine with twice as many CPUs, twice as much RAM and disk typically costs significantly more than twice as much. 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 machine) 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, 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 *non-uniform 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 II, 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 of the issues that arise when data is distributed.

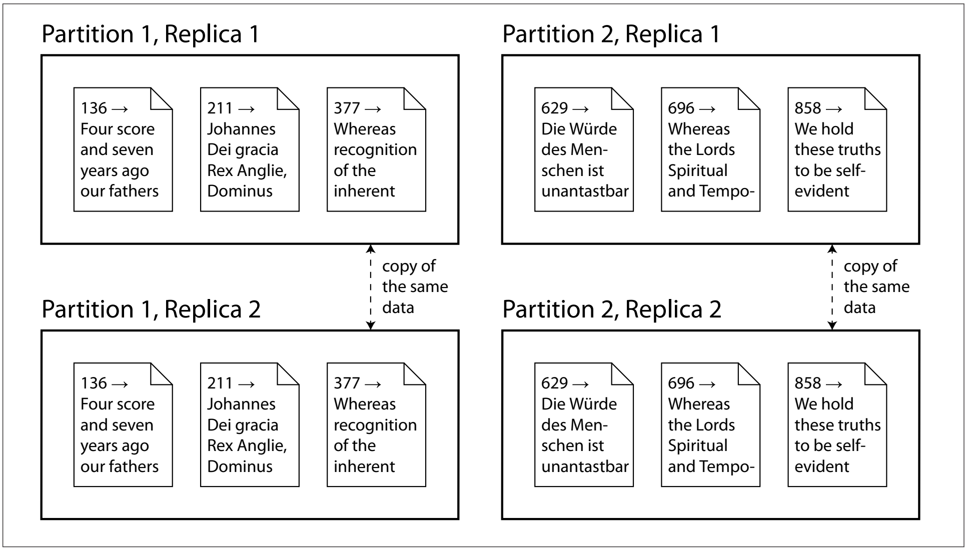
Replication vs. partitioning

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.



Partition 1, Replica 1
136 → Four score and seven years ago our fathers
211 → Johannes Dei gracia Rex Anglie, Dominus
377 → Whereas recognition of the inherent
Partition 2, Replica 1
629 → Die Würde des Men- schen ist unantastbar
696 → Whereas the Lords Spiritual and Tempo-
858 → We hold these truths to be self- evident
copy of the same data
Partition 1, Replica 2
136 → Four score and seven years ago our fathers
211 → Johannes Dei gracia Rex Anglie, Dominus
377 → Whereas recognition of the inherent
copy of the same data
Partition 2, Replica 2
629 → Die Würde des Men- schen ist unantastbar
696 → Whereas the Lords Spiritual and Tempo-
858 → We hold these truths to be self- evident

Figure II-1. A database split into two partitions, with two replicas per partition.

Once we have those in place, 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 us understand all the many things that can go wrong in a data system, and what we can do about it. We'll conclude this part of the book by discussing the fundamental limitations of distributed systems in Chapter 8 and Chapter 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, 21 November 2007.

[2] Ben Stopford: "Shared Nothing vs. Shared Disk Architectures: An Independent View," benstopford.com, 24 November 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.

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 parts of the system have failed (and thus increase availability); or
- 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 fault 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 155 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 request to the leader, which first writes the new data to its local storage.
2. The other replicas are known as *followers* (*read replicas*, *slaves*, 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

[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 which 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.

accordingly, by applying all writes in the same order as they were processed on the leader.

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).

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 non-relational 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 file systems and replicated block devices such as DRBD are similar.

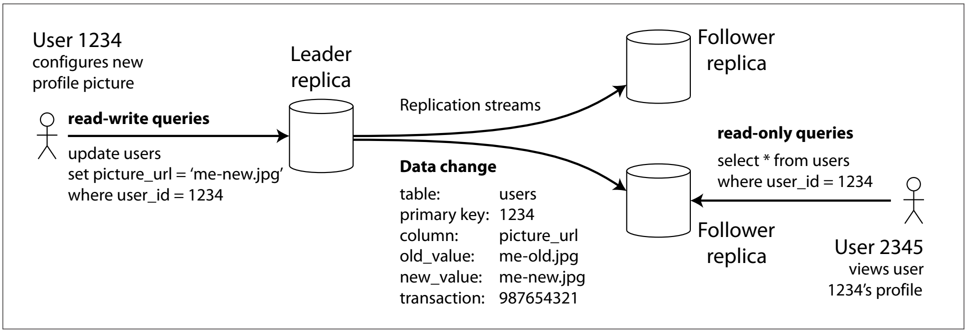


Figure 5-1. Leader-based (master-slave) replication.

Synchronous vs. 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 hard-coded 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 afterwards, 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.

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.

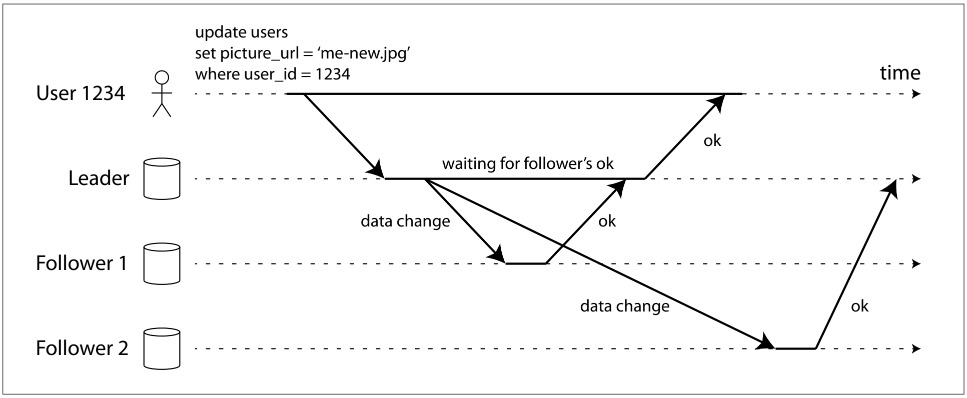


Figure 5-2. Leader-based replication with one synchronous and one asynchronous follower.

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 for 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 goes 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 almost inevitable if there are many followers, or if they are geographically distributed. We will return to this in "Problems With Replication Lag" on page 155.

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 [8].
2. Copy the snapshot to the new follower node.
3. The follower connects to the leader, and requests all data changes that 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 different names: for example, PostgreSQL calls it *log sequence number*, and MySQL calls it *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 failed nodes, 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 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 many 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 either be 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 205). If the old leader comes back, it might still believe that it is 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 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 [9], an out-of-date MySQL follower was promoted to leader. The database used an auto-incrementing counter to assign primary keys to new rows, but because the new leader's counter lagged behind the old leader, 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 161), 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 you're unlucky, you can end up with both nodes being shut down [10].
- 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,

[ii] This is known as *fencing* or, more emphatically, *Shoot The Other Node In The Head* (STONITH).

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 failover 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 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 non-deterministic function, for example `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 auto-incrementing 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, otherwise 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 non-deterministic 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 (see below) if there is any non-determinism in a statement. VoltDB uses statement-based replication, and makes it safe by requiring transactions to be deterministic [11].

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 74), 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 77), which overwrites individual disk blocks, every modification is first written to a write-ahead log (WAL) 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 [12]. 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 block. 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 log replication

An alternative is to use different log formats for replication and for the storage engine. This allows the replication log to be decoupled from the storage engine internals. This is sometimes 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 [13].

Since a logical log is decoupled from the storage engine internals, it can more easily be kept backwards-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 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 [14]. This 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 when 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 164), then you may need to move replication up to the application layer.

Some tools, such as Oracle GoldenGate [15], can do this 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 in a database system so that it is automatically executed when a data change (write transaction) occurs. The trigger has the opportunity to log this change into a separate table, from where 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 [16] and Bucardo for Postgres [17] work like this.

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 before, 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 mostly of 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 reads to be served by a nearby replica.

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 that one is down, so a fully synchronous configuration would be a very unreliable.

Unfortunately, if an application reads from *asynchronous* followers, 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* [18, 19].[iii]

The term *eventually* is deliberately vague: in general, there is no limit 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 minutes.

[iii] The term *eventual consistency* was coined by Douglas Terry *et al.* [20], popularized by Werner Vogels [18], 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.

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 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 have not yet reached the replica. To the user, it looks as though the data they submitted was lost, so they will be understandably unhappy.

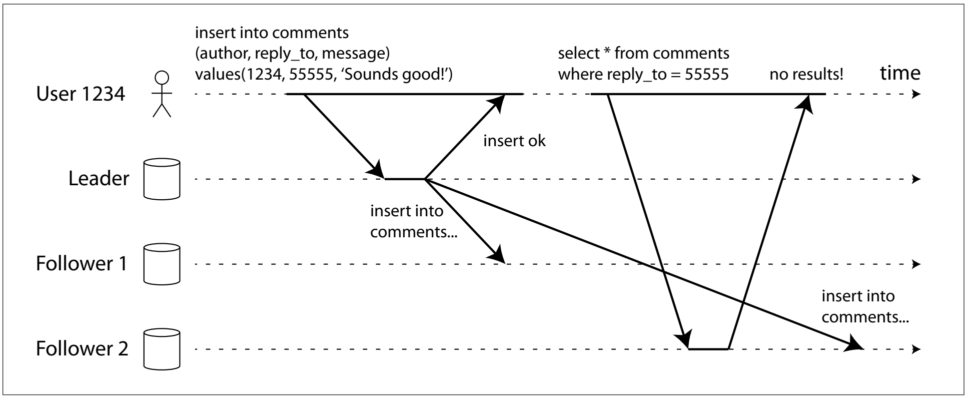


Figure 5-3. A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need read-after-write consistency.

In this situation, we need *read-after-write consistency*, also known as *read-your-writes consistency* [20]. 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 only 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; for one minute after the last update, all reads are made 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.
- Another approach: 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, the read can either 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 278).
- 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 which require remembering a 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 would need to be centralized.
- If your replicas are distributed across different datacenters, there is no guarantee that connections from different devices are routed to the same datacenter. (For example, if the desktop computer uses the home broadband connection and the mobile device uses the cellular data network, their network routes may be com-

pletely 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: it's possible for a user to see things *moving backwards 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. It wouldn't be so bad if the first query hadn't returned anything, because user 2345 probably wouldn't know that user 1234 has recently added a comment. However, it's very confusing for user 2345 if they first see user 1234's comment appear, and then see it disappear again.

Monotonic reads [19] 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 backwards, 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 their user ID, rather than randomly.

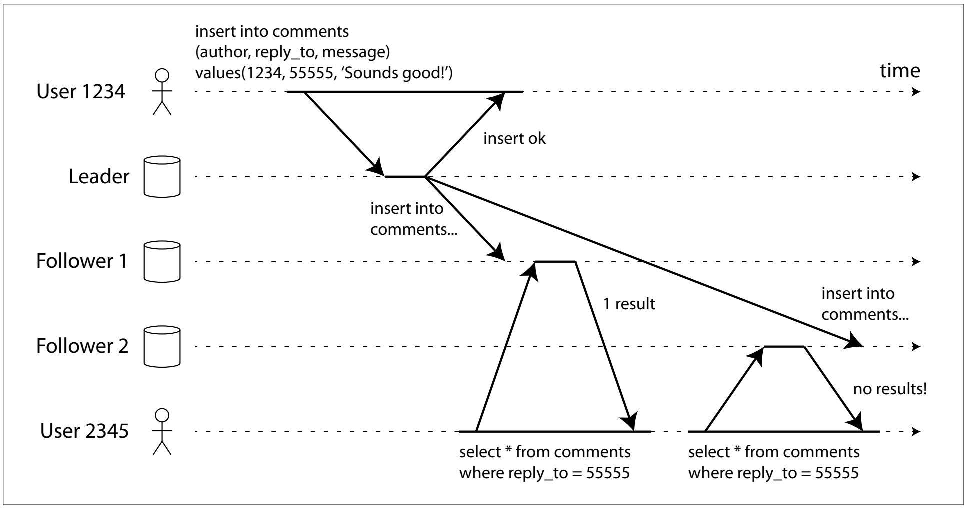


Figure 5-4. A user first reads from a fresh replica, then from a stale replica. Time appears to go backwards. To prevent this anomaly, we need monotonic reads.

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 relationship between those two sentences: Mrs Cake heard Mr Poons' 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 also very confusing [21].

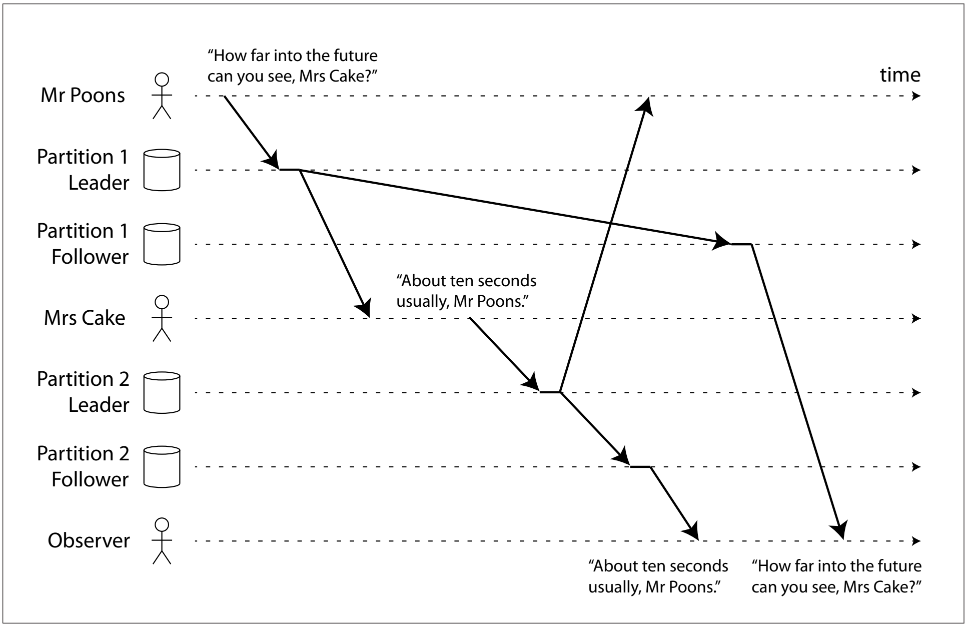


Figure 5-5. If some partitions are replicated slower than others, an observer may see the answer before they see the question.

Preventing this kind of anomaly requires another type of guarantee: *consistent prefix reads* [19]. 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 which are causally related to each other are written to the same partition — but in some applications that can't be done efficiently. In general, ensuring consistent prefix reads requires a kind of distributed transaction with a guarantee such as *snapshot isolation*. We will return to this topic in Chapter 7.

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 above, 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, this adds complexity to the application, and is 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 database 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. That is not necessarily true. We will return to the topic of transactions in Chapter 7.

Multi-leader replication

So far in this chapter we have only considered leader-based replication. Although that is a common replication model, there are interesting alternatives as well.

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 replication* or *active/active*). 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 is a reasonable configuration.

[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.

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.

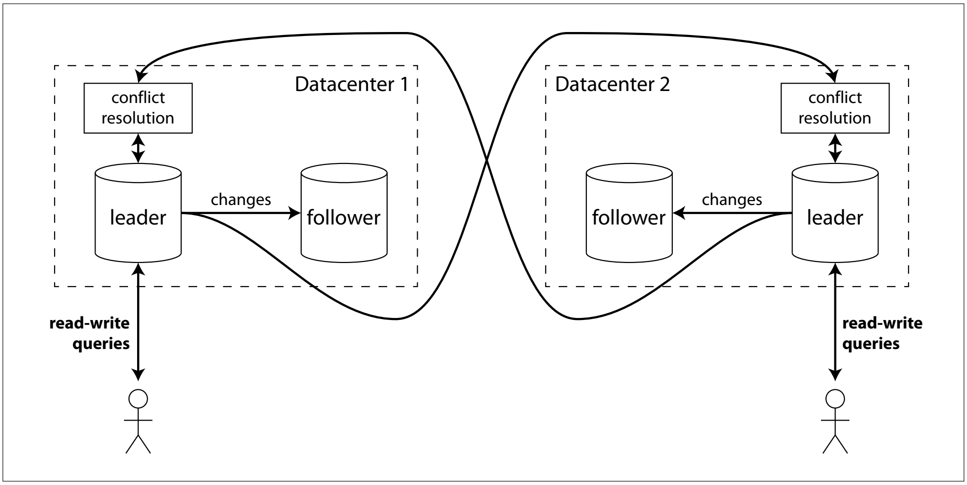


Figure 5-6. Multi-leader replication across multiple datacenters.

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 con-

figuration, 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 database support a multi-leader configuration by default, but it is also often implemented with external tools, such as Tungsten Replicator for MySQL [22], BDR for PostgreSQL [23], and GoldenGate for Oracle [15].

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 in "Handling write conflicts" on page 164.

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, auto-incrementing 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 [24].

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 create 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 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 which aim to make this kind of multi-leader configuration easier. For example, CouchDB is designed for this mode of operation [25].

Collaborative editing

Real-time collaborative editing applications allow several people to edit a document simultaneously. For example, Etherpad [26] and Google Docs [27] allow several people to concurrently edit a text document or spreadsheet (the algorithm is briefly discussed in "Automatic conflict resolution" on page 167).

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 allows multiple users to edit simultaneously, but it also brings all the challenges of multi-leader replication, including requiring conflict resolution.

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 [28]. This problem does not occur in a single-leader database.

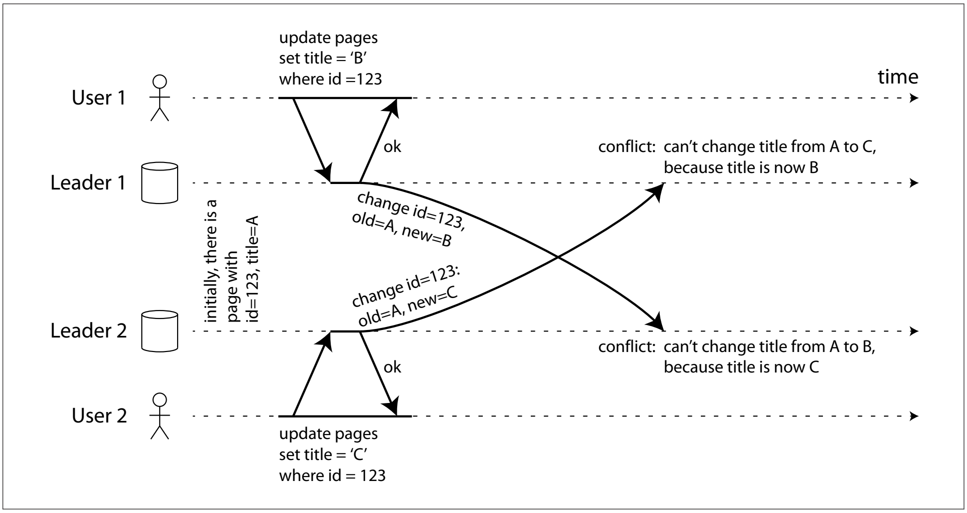

update pages
set title = 'B'
where id =123
time
User 1
ok
Leader 1
conflict: can't change title from A to C, because title is now B
change id=123, old=A, new=B
initially, there is a page with id=123, title=A
change id=123: old=A, new=C
Leader 2
conflict: can't change title from A to B, because title is now C
ok
User 2
update pages
set title = 'C'
where id = 123

Figure 5-7. A write conflict caused by two leaders concurrently updating the same record.

Synchronous vs. 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 confincts is a frequently recommended approach [29].

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 towards 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 technique is popular, it is dangerously prone to data loss [30]. We will discuss LWW in more detail at the end of this chapter ("Detecting concurrent writes" on page 178).
- 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 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 may be something like "B/C").
- Record the conflict in an explicit data structure that preserves all information, and write application code which 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 [31]. Thus, if you have a transaction that atomically makes several different writes (see Chapter 7), each write is still considered separately for 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 re-appearing in their cart even though they had previously been removed [32].

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 data types* (CRDTs) [33] are a family of data structures for sets, maps, ordered lists, counters etc. which can be concurrently edited by multiple users, and which automatically resolve conflicts in sensible ways. Some CRDTs have been implemented in Riak 2.0 [34, 35].
- *Mergeable persistent data structures* [36] 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* [37] is the conflict resolution algorithm behind collaborative editing applications such as Etherpad [26] and Google Docs [27]. It it

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 are still young, but it's likely that they will be integrated into more replicated data systems in 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 one 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.

Solutions to this problem have been proposed [38], but can be hard to implement in practice. For now, detecting conflicts is a question to think about when designing a replicated system, but there isn't a quick ready-made answer.

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.

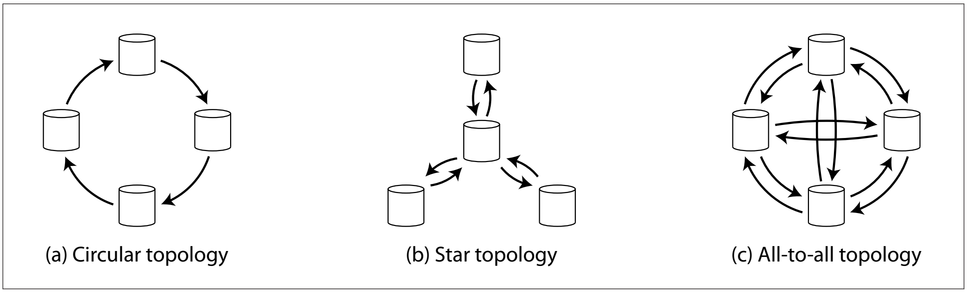


Figure 5-8. Three example topologies in which multi-leader replication can be set up.

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* [29], 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 node it has passed through [39]. When a node receives a data change that is tagged 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.

[v] Not to be confused with a *star schema* (see "Stars and snowflakes: schemas for analytics" on page 90), which describes the structure of a data model, not the communication topology between nodes.

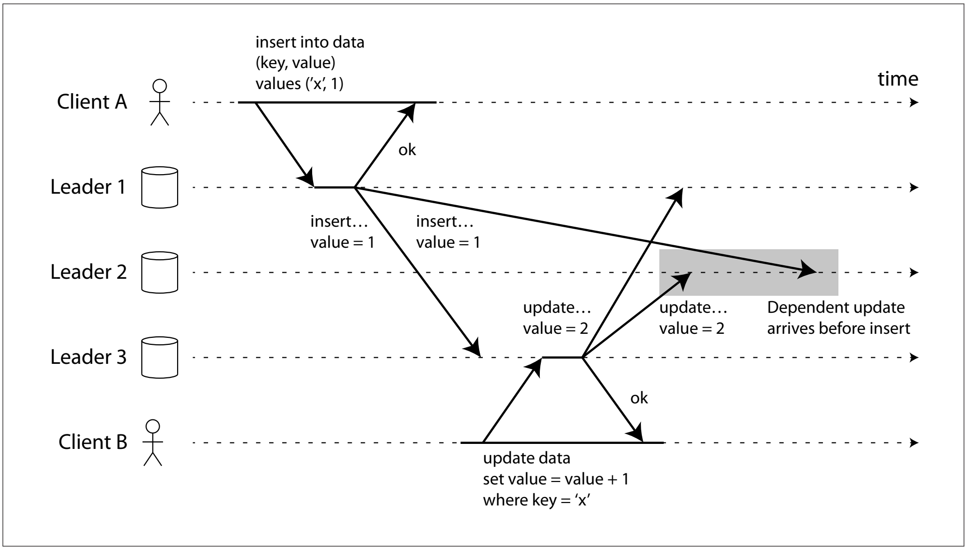


Figure 5-9. With multi-leader replication, writes may arrive in the wrong order at some replicas.

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 159: 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 178). 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 [23], and Tungsten Replicator for MySQL doesn't even try to detect conflicts [29].

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, 40], 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 their in-house *Dynamo* system [32].[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 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 configuration, if you want to continue processing writes, you may need to perform a failover (see "Handling node outages" on page 150).

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, the two available replicas accept the write, but the unavailable replica misses the write. 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.

[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.

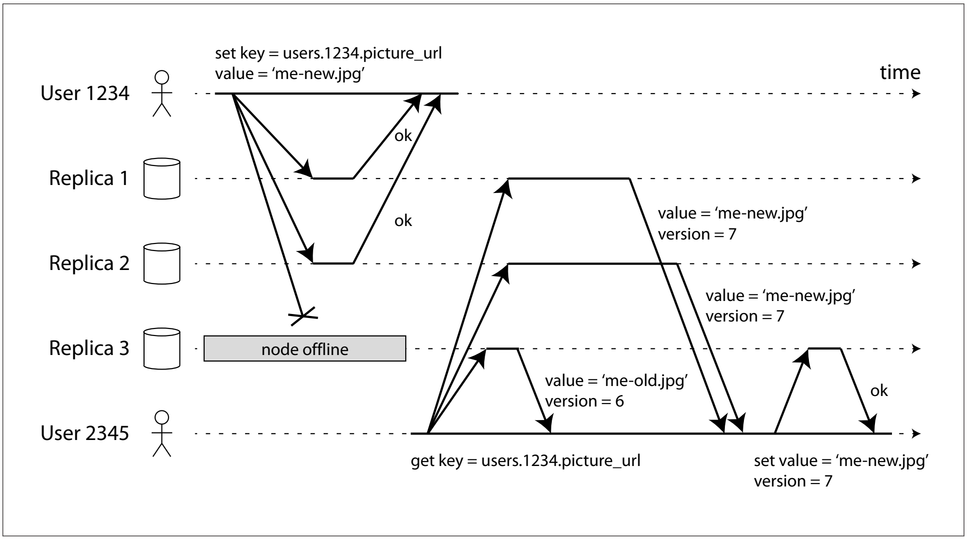


Figure 5-10. A quorum write, quorum read, and read repair after a node outage.

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, e.g. the up-to-date value from one node, and a stale value from another. A version number is used to determine which value is newer (see "Detecting concurrent writes" on page 178).

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 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 [40].[vii] You can think of r and w as the minimum number of votes required for the read or write to be valid.

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.

[vii] Sometimes this kind of quorum is also called *strict quorum*, to contrast with *sloppy quorums* (discussed in "Sloppy quorums and hinted handoff" on page 177).

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, so that the dataset can be 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.

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 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.

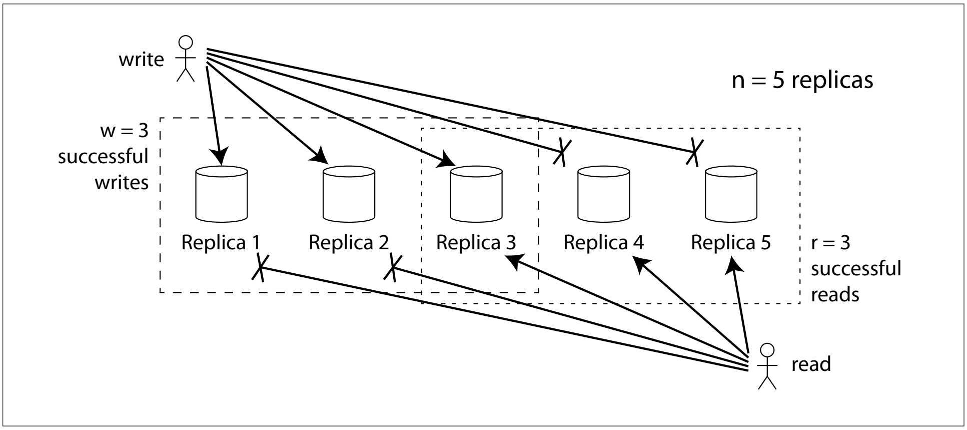


Figure 5-11. If $w + r > n$, at least one of the r replicas you read from must have seen the most recent successful write.

Limitations of quorum consistency

If you have n replicas, and 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 because the set of nodes to which you've written and the set of nodes from which you've read must overlap, i.e. there has to be at least one node with the latest value (illustrated in Figure 5-11). However, as we shall see, there are edge cases in which this is not true.

You may also set w and r to smaller numbers, so that $w + r \leq n$. 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 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, the database becomes unavailable for writing or reading, respectively.

However, even with $w + r > n$, there are likely to be edge cases when 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 177), 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 [41].
- 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 164). If a winner is picked based on a timestamp (*last write wins*), writes can be lost due to clock skew [30]. We will return to this topic in "Detecting concurrent writes" on page 178.
- 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 [42].
- 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 324.

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 155 (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 in Chapter 7 and Chapter 9.

Monitoring staleness

From an operational perspective, it's important to be monitoring 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* [43]. 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 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* [32]: 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 locked 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* [42].

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 [41, 44, 45].

Multi-datacenter operation

We previously discussed cross-datacenter replication as a use case for multi-leader replication (see "Multi-leader replication" on page 161). Leaderless replication is also well suited 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 acknowledgement 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 [45, 46].

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 [47].

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 164), 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*.

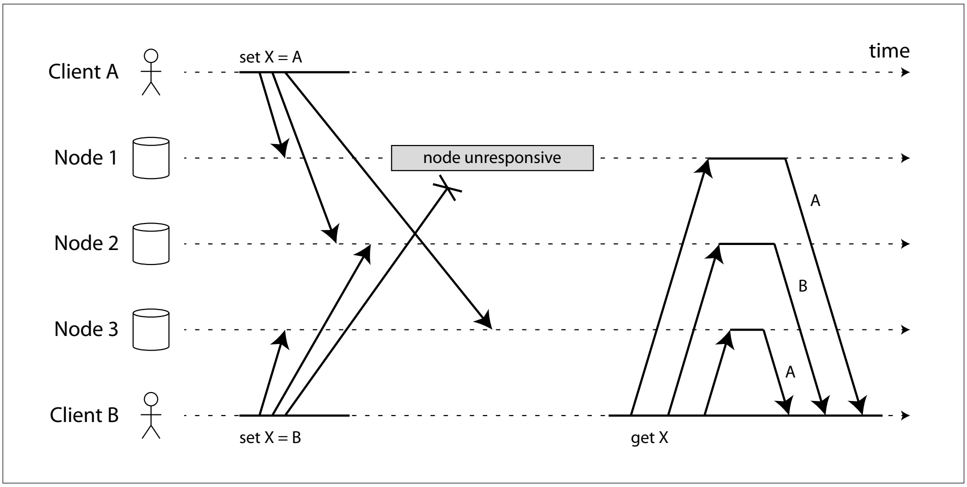


Figure 5-12. Concurrent writes in a Dynamo-style datastore: there is no well-defined ordering.

If each node simply overwrote the value for a key whenever it received a write request from a client, they 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 towards 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 164. 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, then the replicas will eventually converge to the same value.

As indicated by the scare 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 a lower timestamp. This conflict resolution algorithm, called *last write wins* (LWW), is the only supported conflict resolution method in Cassandra [48], and an optional feature in Riak [30].

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 283.

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 key, thus giving each write operation a unique key [48].

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 operation.

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 [49].

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 literally happened at 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 [49], 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.

Tracking happens-before relationships

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 database 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 a shopping cart. 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 the `eggs`, 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 it 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 prior version 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.

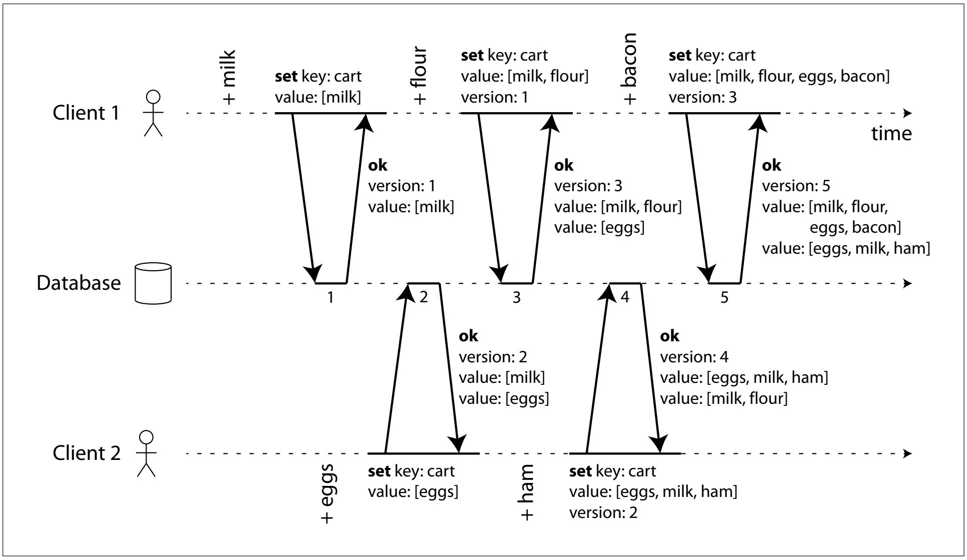


Figure 5-13. Tracking causal dependencies between two clients concurrently editing a shopping cart.

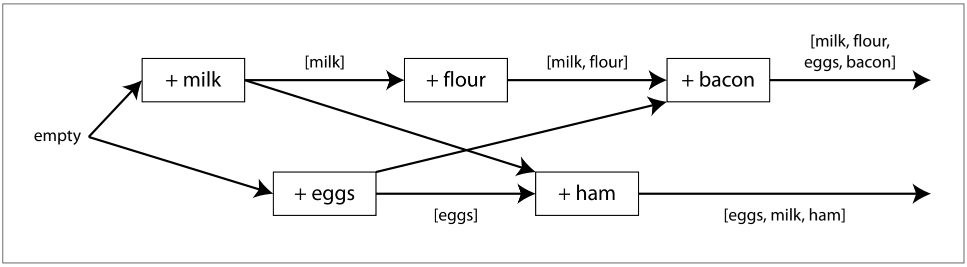


Figure 5-14. Graph of causal dependencies in Figure 5-13.

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.

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 example above.)
- 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 to all other writes, so it will not overwrite anything — it will just be returned as one of the versions 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 happened concurrently, clients have to clean up afterwards 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 164). 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 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 cart, 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 would reappear in the union of the siblings [32].

To prevent this, 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*.

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 167. For example, Riak's datatype support uses a family of data structures called CRDTs [33, 34, 50] 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 track 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 all of the other replicas. It can then use that information to figure out 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* [51]. A few variants of this idea are in use, but the most interesting is probably the *dotted version vector* [52], which is used in Riak 2.0 [53, 54]. We won't go into the details, but the way it works is quite similar to what we saw in the example above.

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. This 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: this 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*. The difference between these data structures is subtle — one way of looking at it is that version vectors are for client-server systems, and vector clocks are for peer-to-peer systems. Please see the references for details [52, 55, 56].

Summary

In this chapter, we looked at the issue of replication, that is: keeping a copy of the same data on several machines. 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 — 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:

1. *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.
2. *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.
3. *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 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*: a user should always see data that they submitted themselves.
- *Monotonic reads*: after a user has 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, 1 April 2014.

[8] "Percona Xtrabackup - Documentation." Percona LLC, 2014.

[9] Jesse Newland: "GitHub availability this week," github.com, 14 September 2012.

[10] Mark Imbriaco: "Downtime last Saturday," github.com, 26 December 2012.

[11] John Hugg: "'All in' with determinism for performance and testing in distributed systems," at *Strange Loop*, September 2015.

[12] Amit Kapila: "WAL Internals Of PostgreSQL," at *PostgreSQL Conference* (PGCon), May 2012.

[13] *MySQL Internals Manual*. Oracle, 2014.

[14] 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.

[15] "Oracle GoldenGate 12c: Real-time access to real-time information." Oracle White Paper, October 2013.

[16] Shirshanka Das, Chavdar Botev, Kapil Surlaker, et al.: "All Aboard the Databus!," at *ACM Symposium on Cloud Computing* (SoCC), October 2012.

[17] Greg Sabino Mullane: "Version 5 of Bucardo database replication system," blog.endpoint.com, 23 June 2014.

[18] Werner Vogels: "Eventually Consistent," *ACM Queue*, volume 6, number 6, pages 14–19, October 2008. doi:10.1145/1466443.1466448

[19] Douglas B Terry: "Replicated Data Consistency Explained Through Baseball," Microsoft Research, Technical Report MSR-TR-2011-137, October 2011.

[20] 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), pages 140–149, September 1994. doi:10.1109/PDIS.1994.331722

[21] Terry Pratchett: *Reaper Man: A Discworld Novel*. Victor Gollancz, 1991. ISBN: 0-575-04979-0

[22] "Tungsten Replicator." Continuent, Inc., 2014.

[23] "BDR 0.10.0 Documentation," The PostgreSQL Global Development Group, bdr-project.org, 2015.

[24] Robert Hodges: "If You *Must* Deploy Multi-Master Replication, Read This First," scale-out-blog.blogspot.co.uk, 30 March 2012.

[25] J Chris Anderson, Jan Lehnardt, and Noah Slater: *CouchDB: The Definitive Guide*. O'Reilly Media, January 2010. ISBN: 978-0-596-15589-6

[26] AppJet, Inc.: "Etherpad and EasySync Technical Manual," github.com, 26 March 2011.

[27] John Day-Richter: "What's different about the new Google Docs: Making collaboration fast," googledrive.blogspot.com, 23 September 2010.

[28] Frazer Clement: "Eventual Consistency - detecting conflicts," messagepassing.blogspot.co.uk, 20 October 2011.

[29] Robert Hodges: "State of the Art for MySQL Multi-Master Replication," at *Percona Live: MySQL Conference & Expo*, April 2013.

[30] John Daily: "Clocks Are Bad, Or, Welcome to the Wonderful World of Distributed Systems," basho.com, 12 November 2013.

[31] Riley Berton: "Is Bi-Directional Replication (BDR) in Postgres Transactional?," sdf.org, 4 January 2016.

[32] 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.

[33] 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.

[34] Sam Elliott: "CRDTs: An UPDATE (or maybe just a PUT)," at *RICON West* (RICON West), October 2013.

[35] Russell Brown: "A Bluffers Guide to CRDTs in Riak," gist.github.com, 28 October 2013.

[36] Benjamin Farinier, Thomas Gazagnaire, and Anil Madhavapeddy: "Mergeable persistent data structures," at *26es Journées Francophones des Langages Applicatifs* (JFLA), January 2015.

[37] 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.

[38] 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), pages 172–182, December 1995. doi: 10.1145/224056.224070

[39] Lars Hofhansl: "HBASE-7709: Infinite loop possible in Master/Master replication," issues.apache.org, 29 January 2013.

[40] David K Gifford: "Weighted Voting for Replicated Data," at *7th ACM Symposium on Operating Systems Principles* (SOSP), pages 150–162, December 1979. doi:10.1145/800215.806583

[41] Joseph Blomstedt: "Re: Absolute consistency," email to riak-users mailing list, lists.basho.com, 11 January 2012.

[42] Joseph Blomstedt: "Bringing Consistency to Riak," at *RICON West*, October 2012.

[43] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica: "Quantifying eventual consistency with PBS," *Communications of the ACM*, volume 57, number 8, pages 93–102, August 2014. doi:10.1145/2632792

[44] Jonathan Ellis: "Modern hinted handoff," datastax.com, 11 December 2012.

[45] "Project Voldemort wiki," github.com, 2013.

[46] "Apache Cassandra 2.0 Documentation," DataStax, Inc., 2014.

[47] "Riak Enterprise: Multi-Datacenter Replication." Technical whitepaper, Basho Technologies Inc., September 2014.

[48] Jonathan Ellis: "Why Cassandra doesn't need vector clocks," datastax.com, 2 September 2013.

[49] 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

[50] Joel Jacobson: "Riak 2.0: Data Types," blog.joeljacobson.com, 23 March 2014.

[51] 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

[52] Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves: "Dotted Version Vectors: Logical Clocks for Optimistic Replication," arxiv:1011.5808, 26 November 2010.

[53] Sean Cribbs: "A brief history of time in Riak," at *RICON*, October 2014.

[54] Russell Brown: "Vector Clocks Revisited Part 2: Dotted Version Vectors," basho.com, 10 November 2015.

[55] Carlos Baquero: "Version Vectors are not Vector Clocks," haslab.wordpress.com, 8 July 2011.

[56] 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

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 also need to break the data up into *partitions*.[i]

Terminological confusion

What we call a *partition* here is called a *shard* in MongoDB, Elasticsearch and SolrCloud, 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 different 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.

[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.

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 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 small 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 87): this difference affects how the system is tuned, but the fundamentals of partitioning apply to both kinds of workload.

In this chapter we will first look at different approaches for partitioning a 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 partition 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 in 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.

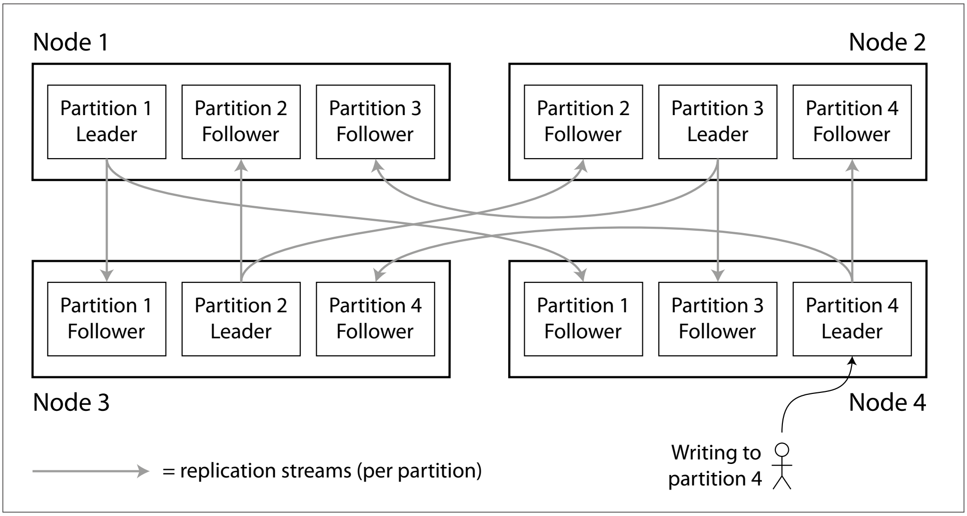


Figure 6-1. Combining replication and partitioning: each node acts as leader for some partitions, and follower for other partitions.

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 node?

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 — ten nodes should be able to handle ten times as much data and ten 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*. This makes the partitioning much less effective. In an extreme case, all the load could end up on one partition — so nine out of ten 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 of avoiding hot spots would be to assign records to nodes randomly. That would distribute the data quite evenly across the nodes, but 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 would 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.

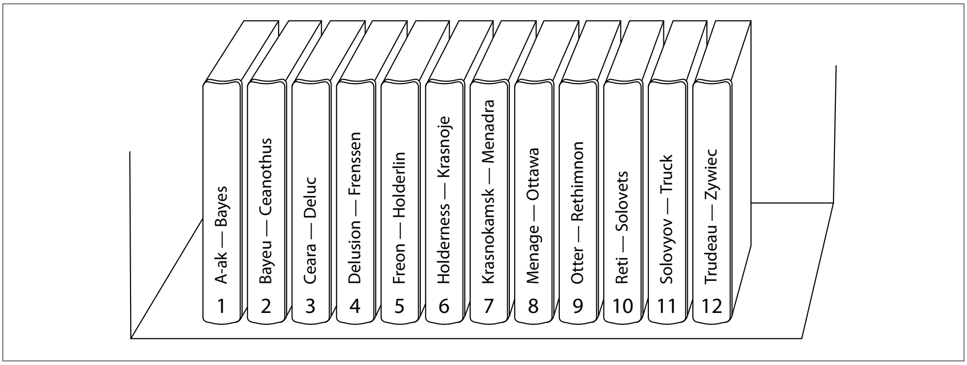


Figure 6-2. A print encyclopedia is partitioned by key range.

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).

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 get to that in "Rebalancing partitions" on page 201. 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 74). 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 84). For example, consider an application that stores data from a network of sensors, where the key is the timestamp of 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 it happens, 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 which 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.[ii] This is illustrated in Figure 6-3.

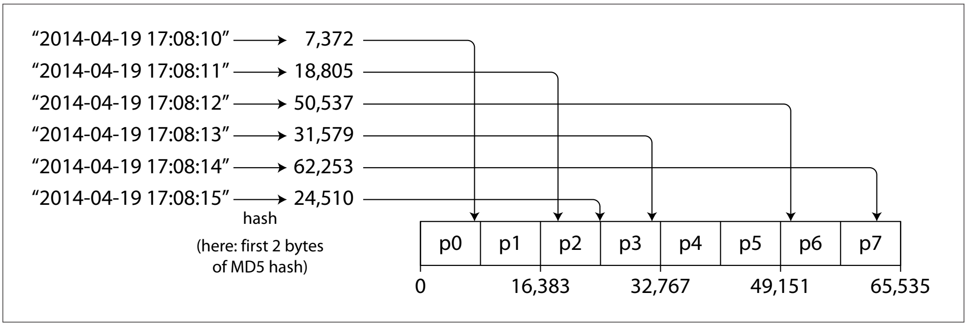


Figure 6-3. Partitioning by hash of key.

You can now 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 technique is good at distributing keys fairly among the partitions. The par-

[ii] It doesn't need to be a cryptographically strong hash function. For example, Cassandra and MongoDB use MD5, and Voldemort uses the Fowler–Noll–Vo function.

tition boundaries can be evenly spaced, or they can be chosen pseudo-randomly (in which case the technique is sometimes known as *consistent hashing*).

Consistent hashing

Consistent hashing, as defined by Karger et al. [6], 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 201, this particular approach actually doesn't work very well for databases [7], and so it is rarely used in practice (the documentation of some databases still refers to consistent hashing, but it is usually inaccurate). Because this is so confusing, it's best to avoid the term *consistent hashing*, and just call it *hash partitioning* instead.

Unfortunately, by using the hash of the key for partitioning, we also lost 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 [8], Couchbase [9] and Voldemort.

Cassandra achieves a compromise between the two partitioning strategies [10, 11, 12]. 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.

This means that a query 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 based on the other columns of the key.

This allows 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 impossible: for example, on a social media site, a celebrity user with millions of followers may cause a storm of activity when they do something [13]. This 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 which keys are being split.

Perhaps in 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.

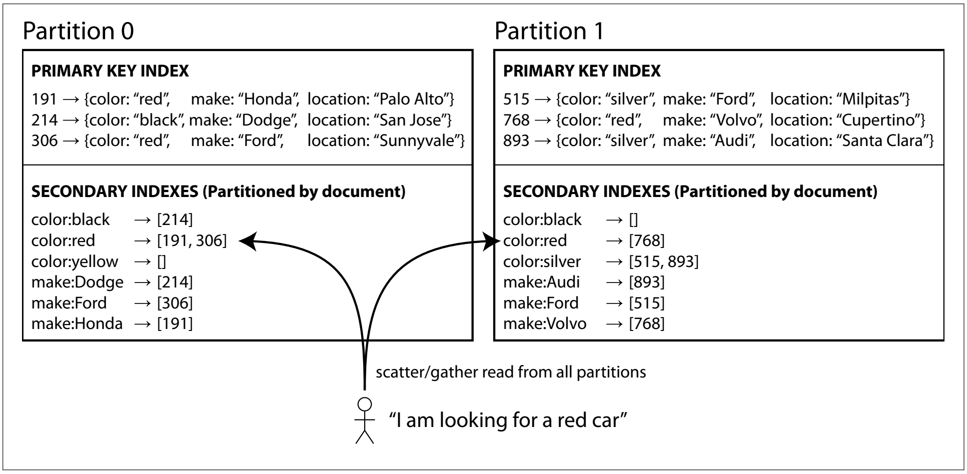
Partitioning and secondary indexes

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 82). A secondary index usually doesn't identify a record uniquely, but rather, it's 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. NoSQL 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.



Partition 0
PRIMARY KEY INDEX
191 → {color: "red", make: "Honda", location: "Palo Alto"}
214 → {color: "black", make: "Dodge", location: "San Jose"}
306 → {color: "red", make: "Ford", location: "Sunnyvale"}
SECONDARY INDEXES (Partitioned by document)
color:black → [214]
color:red → [191, 306]
color:yellow → []
make:Dodge → [214]
make:Ford → [306]
make:Honda → [191]
Partition 1
PRIMARY KEY INDEX
515 → {color: "silver", make: "Ford", location: "Milpitas"}
768 → {color: "red", make: "Volvo", location: "Cupertino"}
893 → {color: "silver", make: "Audi", location: "Santa Clara"}
SECONDARY INDEXES (Partitioned by document)
color:black → []
color:red → [768]
color:silver → [515, 893]
make:Audi → [893]
make:Ford → [515]
make:Volvo → [768]
scatter/gather read from all partitions
"I am looking for a red car"

Figure 6-4. Partitioning secondary indexes by document.

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 *document ID* — and you partition the database by the document ID (for example, IDs 0 to 499 in partition 0, 500 to 999 in partition 1, etc).

Now 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.[iii] 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`.

[iii] If your database only supports a key-value model, you might be tempted to implement a secondary index yourself by creating a value-to-document-ID mapping 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 222.

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 14). Nevertheless, it is widely used: MongoDB, Riak [14], Cassandra [15], Elasticsearch [16], SolrCloud [17], and VoltDB [18] 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).

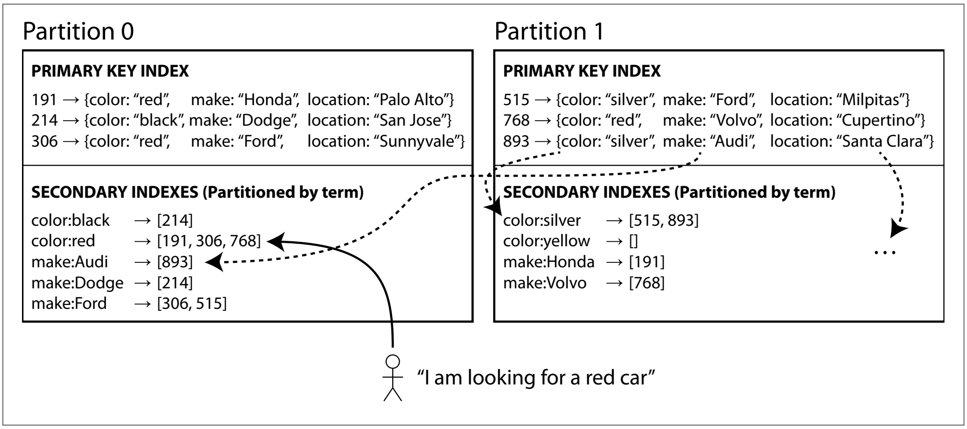

Partition 0
PRIMARY KEY INDEX
191 → {color: "red", make: "Honda", location: "Palo Alto"}
214 → {color: "black", make: "Dodge", location: "San Jose"}
306 → {color: "red", make: "Ford", location: "Sunnyvale"}
SECONDARY INDEXES (Partitioned by term)
color:black → [214]
color:red → [191, 306, 768]
make:Audi → [893]
make:Dodge → [214]
make:Ford → [306, 515]
Partition 1
PRIMARY KEY INDEX
515 → {color: "silver", make: "Ford", location: "Milpitas"}
768 → {color: "red", make: "Volvo", location: "Cupertino"}
893 → {color: "silver", make: "Audi", location: "Santa Clara"}
SECONDARY INDEXES (Partitioned by term)
color:silver → [515, 893]
color:yellow → []
make:Honda → [191]
make:Volvo → [768]
...
"I am looking for a red car"

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 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 *term-based partitioning*, 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 for 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 now 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 case of faults in the infrastructure [19].

Other uses of global term-partitioned indexes include Riak's search feature [20], and Oracle data warehouse, which lets you choose between local and global indexing [21].

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 call for data to be moved from one node to another. The process of moving data around between nodes in the cluster 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.
- Don't move more data than necessary between nodes, to avoid overloading the network.

Strategies for rebalancing

There are a few different ways assigning partitions to nodes [22]. Let's briefly discuss each in turn.

How not to do it: hash mod N

When you're 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 (for example, assign *key* to partition 0 if $0 \leq hash(key) < b_0$, assign *key* to partition 1 if $b_0 \leq hash(key) < b_1$, 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 would need to be moved from one node to another. For example, say *hash*(*key*) = 123456. If you have 10 nodes, that key starts out on node 6 (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). That makes rebalancing excessively expensive.

Thus, we need an approach which 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 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 change. 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.

In principle, you can even take 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 [14], Cassandra since version 1.2 [23], Elasticsearch [24], Couchbase [9] and Voldemort [25].

In this configuration, the number of partitions is usually fixed when the database is first set up, and not changed afterwards. Although in principle it's possible to split and merge partitions (see 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.

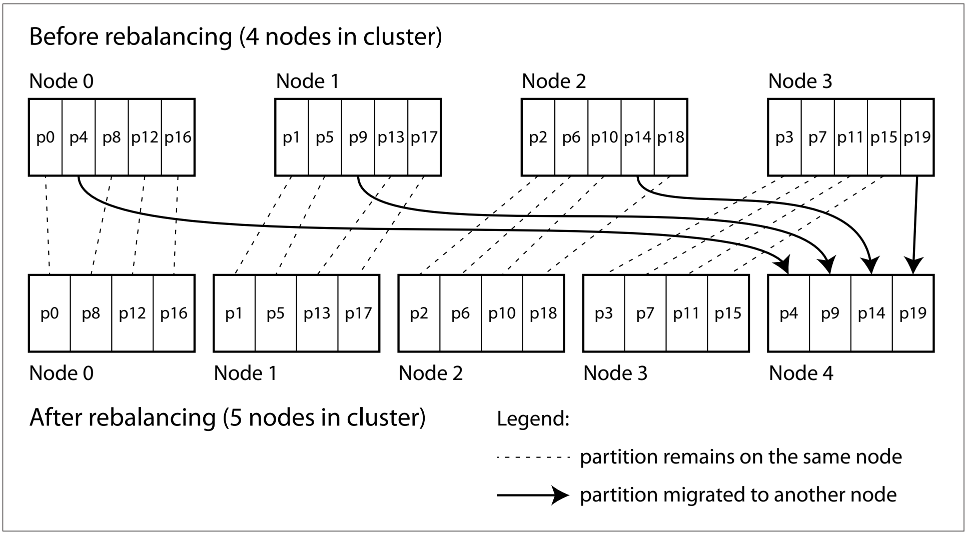


Figure 6-6. Adding a new node to a database cluster with multiple partitions per node.

Dynamic partitioning

A fixed number of partitions works quite well in conjunction with hash partitioning, because the hash function ensures that the keys are distributed uniformly across the whole range of possible hashes.

However, for databases that use key range partitioning (see "Partitioning by key range" on page 194), fixed partition boundaries would be very inconvenient: if you get the boundaries wrong, you could end up with all of the data in one partition, and all of the other partitions being 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 is similar to what happens at the top level of a B-tree (see "B-trees" on page 77).

Each partition is assigned to one node, and each node can handle multiple partitions, like in the previous case. 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 file system [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 [22].

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, 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, this requires that you already know what the key distribution is going to look like [26, 4].

Dynamic partitioning is not only 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, but it splits partitions dynamically in either case.

Other rebalancing strategies

Before version 1.2, Cassandra used consistent hashing with pseudo-random partition boundaries, as originally described by Karger et al. [6] Rather than assigning several small partitions to each node, it used one big partition per node, covering a large continuous range of hashes.

However, this approach suffered from poor load distribution, and made it difficult to add nodes to the cluster: an existing node had to split its range to give half of its data to a new node. This expensive operation was difficult to perform in the background without impacting query performance. For those reasons, Cassandra's partitioning strategy was replaced with the fixed-number-of-partitions approach described above [22].

These problems can be avoided by using a better hash function [7] but in practice, the most widely-used partitioning models are either hashing with a fixed number of partitions, or dynamic partitioning by key range (when range queries are required).

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 re-routing requests and moving a large amount of data from one node to another. If it is not done carefully, this can overload the network or the nodes, and cause performance problems for other systems.

This 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 other nodes and the network, thus potentially overloading more nodes and 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 performing it fully automatically, 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 [27].

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.
2. Send all requests from clients to a routing tier first, which determines the node that should handle the 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.

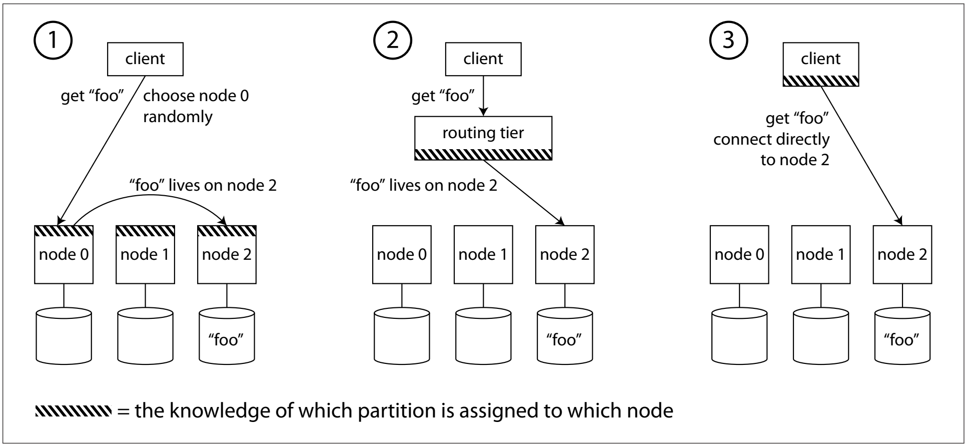


Figure 6-7. Three different ways of routing a request to the right node.

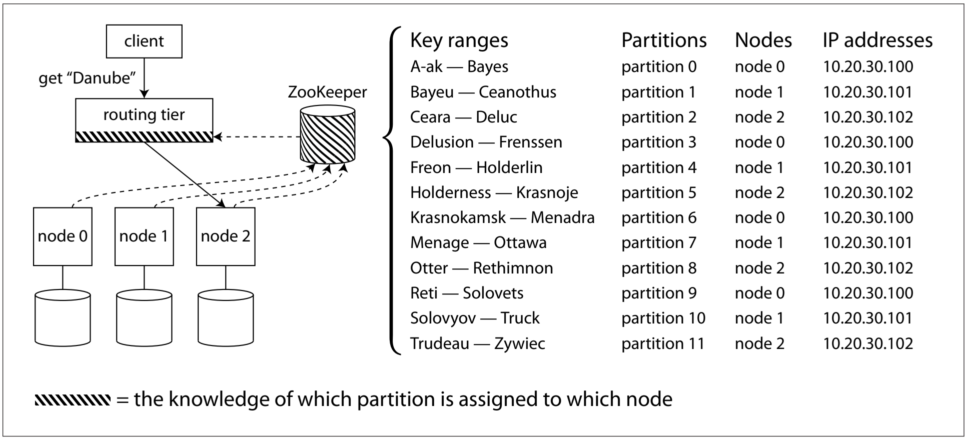
In all cases, the key problem is: how does the component making the routing decision (which may be one of the nodes, or the routing tier, or the client) learn about changes in the assignment of partitions to nodes?

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.

For example, LinkedIn's Espresso uses Helix [28] for cluster management (which in turn relies on ZooKeeper), implementing a routing tier as shown in Figure 6-8. MongoDB has a similar architecture, but relies on its own *config server* implementation rather than ZooKeeper, and *mongos* daemons as routing tier. HBase, SolrCloud and Kafka also use ZooKeeper to track partition assignment.



Key ranges	Partitions	Nodes	IP addresses
A-ak — Bayes	partition 0	node 0	10.20.30.100
Bayeu — Ceanothus	partition 1	node 1	10.20.30.101
Ceara — Deluc	partition 2	node 2	10.20.30.102
Delusion — Frenssen	partition 3	node 0	10.20.30.100
Freon — Holderlin	partition 4	node 1	10.20.30.101
Holderness — Krasnoje	partition 5	node 2	10.20.30.102
Krasnokamsk — Menadra	partition 6	node 0	10.20.30.100
Menage — Ottawa	partition 7	node 1	10.20.30.101
Otter — Rethimnon	partition 8	node 2	10.20.30.102
Reti — Solovets	partition 9	node 0	10.20.30.100
Solovyov — Truck	partition 10	node 1	10.20.30.101
Trudeau — Zywiec	partition 11	node 2	10.20.30.102

▨ = the knowledge of which partition is assigned to which node

Figure 6-8. Using ZooKeeper to keep track of assignment of partitions to nodes.

Cassandra and Riak take a different approach: they use a *gossip protocol* among the nodes to disseminate and agree on 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 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 agreement protocol between nodes. Normally Couchbase is configured with a routing tier called *moxi*, which learns about routing changes through a management connection to the cluster nodes [29].

When using a routing tier or when sending requests to a random node, clients still need to find the IP addresses of the machines to connect to. However, those addresses 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 focussed 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 data stores.

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 joins, 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 gets a lot of commercial interest. We will touch on some techniques for parallel query execution in Chapter 10. For a more detailed overview of techniques used in parallel databases, please see the literature references [1, 30].

Summary

In this chapter we explored different ways of partitioning a large dataset into smaller subsets. This is necessary when you have so much data that storing and processing it on a single machine is no longer feasible.

The main goal of partitioning is to spread the data and the 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 from time to time as nodes are added 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. This 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 sub-ranges 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 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.

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 possibilities for doing this:

- *Document-partitioned index*: 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 index* (global index): 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? In the next chapter we will turn to the topic of *transactions*, which can make multi-write operations easier to reason about.

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, 24 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, 10 April 2013.

[5] Ikai Lan: "App Engine datastore tip: monotonically increasing values are bad," ikaisays.com, 25 January 2011.

[6] 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

[7] John Lamping and Eric Veach: "A Fast, Minimal Memory, Consistent Hash Algorithm," arxiv.org, June 2014.

[8] Eric Redmond: "A Little Riak Book," Version 1.4.0, Basho Technologies, September 2013.

[9] "Couchbase 2.5 Administrator Guide," Couchbase, Inc., 2014.

[10] 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.

[11] Jonathan Ellis: "Facebook's Cassandra paper, annotated and compared to Apache Cassandra 2.0," datastax.com, 12 September 2013.

[12] "CQL for Cassandra 2.0," DataStax, Inc., 2014.

[13] Samuel Axon: "3% of Twitter's Servers Dedicated to Justin Bieber," mashable.com, 7 September 2010.

[14] "Riak 1.4.8 Docs," Basho Technologies, Inc., 2014.

[15] Richard Low: "The sweet spot for Cassandra secondary indexing," wentnet.com, 21 October 2013.

[16] Zachary Tong: "Customizing your document routing," elasticsearch.org, 3 June 2013.

[17] "Apache Solr Reference Guide," Apache Software Foundation, 2014.

[18] Andrew Pavlo: "H-Store Frequently Asked Questions," hstore.cs.brown.edu, October 2013.

[19] "Amazon DynamoDB Developer Guide," Amazon Web Services, Inc., 2014.

[20] Rusty Klophaus: "Difference between 2I and Search," email to riak-users mailing list, lists.basho.com, 25 October 2011.

[21] Donald K Burleson: "Object partitioning in Oracle," dba-oracle.com, 8 November 2000.

[22] Eric Evans: "Rethinking Topology in Cassandra," at *ApacheCon Europe*, November 2012.

[23] Brandon Williams: "Virtual nodes in Cassandra 1.2," datastax.com, 4 December 2012.

[24] Rafał Kuć: "Reroute API Explained," elasticsearchserverbook.com, 30 September 2013.

[25] "Project Voldemort documentation," project-voldemort.com.

[26] Enis Soztutar: "Apache HBase Region Splitting and Merging," hortonworks.com, 1 February 2013.

[27] Jason Wilder: "Open-Source Service Discovery," jasonwilder.com, 4 February 2014.

[28] 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

[29] "Moxi 1.8 Manual," Couchbase, Inc., 2014.

[30] 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

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 others' 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. This makes error handling 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 in order 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 different kinds of race condition 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 non-relational 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 unchanged over 40 years: the transaction support in MySQL, PostgreSQL, Oracle, SQL Server, etc. is uncannily similar to System R.

In the late 2000s, non-relational (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 it 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 unpack 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 computing. For example, in multithreaded 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, but 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 217).

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 part way 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 155).
- In the CAP theorem (see Chapter 9), the word *consistency* is used to mean *linearizability* (see "Linearizability" on page 314).
- 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 three 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 invariant 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 others' 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, and that it wasn't considered important at the time.

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 these isolation levels in "Weak isolation levels" on page 224.

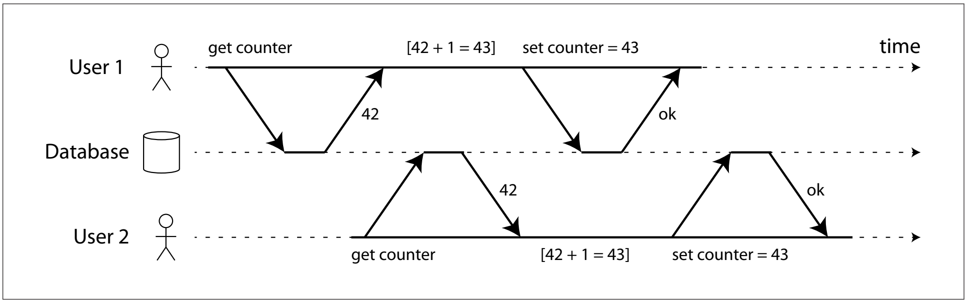


Figure 7-1. A race condition between two clients concurrently incrementing a counter.

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.

On a single-node database, this 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 "Update-in-place vs. append-only logging" on page 80), which allows recovery in case the data structures on disk are corrupted. On 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 4, 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 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 disks 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 4), 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 150).
- 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 become gradually corrupted without being detected [17]. If data has been corrupted for some time, replicas and recent backups may also be corrupted. In this case, you would need to try to restore it from a historical backup.
- If an SSD is disconnected from power, it can start losing data within a few weeks, depending on the temperature [18].

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, perhaps you found this query to be too slow if there are many emails, so you now 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 half-way point.

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 could go 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.

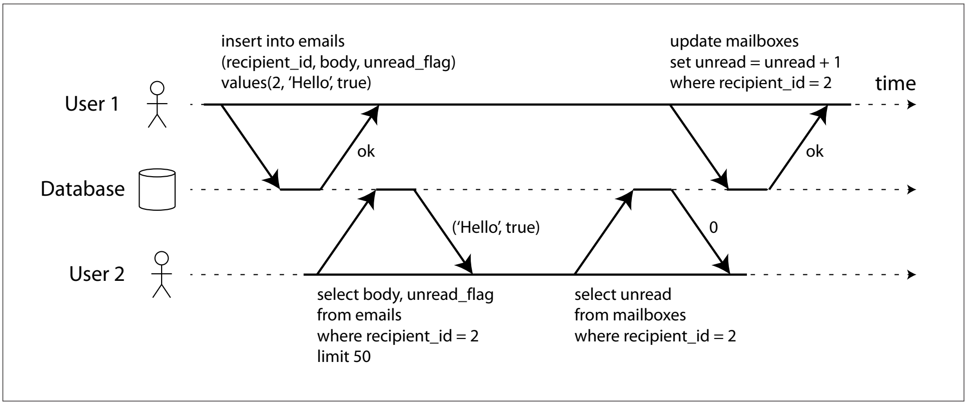


Figure 7-2. Violating isolation: one transaction reads another transaction's uncommitted writes.

[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.

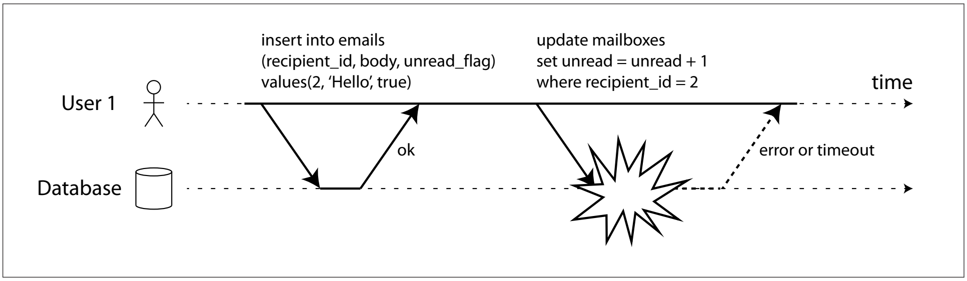


Figure 7-3. Atomicity ensures that if an error occurs, any prior writes from that transaction are undone, to avoid an inconsistent state.

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]

On the other hand, many non-relational 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 the new value 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-

[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 commit, but before the server acknowledges that the commit happened, the client doesn't know whether it was committed or not. To solve this, a transaction manager can group operations by a unique transaction identifier that is not bound to a particular TCP connection.

value pair) on one node. Atomicity can be implemented using a log for crash recovery (see "Update-in-place vs. append-only logging" on page 80), 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[iv] operations, such as an increment operation, which removes the need for a read-modify-write cycle like 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.

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 233). However, they are not transactions in the usual sense of the word. Unfortunately, compare-and-set and other single-object operations have been dubbed "lightweight transactions" or even "ACID" for marketing purposes [19, 20, 21], 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.

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 are 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 you need 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 ensure that these references remain valid: when inserting several records that refer to each other, the foreign keys have to be correct and up-to-date, otherwise 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

[iv] Strictly speaking, *atomic increment* uses the word *atomic* in the sense of multithreaded programming. In the context of ACID, it should actually be called *isolated* or *serializable* increment. But that's getting nitpicky.

transactions are needed when updating a single document. However, document databases lacking join functionality also encourage denormalization (see "Relational vs. 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 a database 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 224.

Handling errors and aborts

A key feature of a transaction is that in the case of a problem, it can be aborted and retried. 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 continue.

Not all systems follow that philosophy: especially datastores with leaderless replication (see "Leaderless replication" on page 171) 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' 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 a feedback cycles, limiting the number of retries, use exponential backoff, and handle overload-related errors differently from other errors (if possible).
- Only transient errors (for example due to deadlock, isolation violation, temporary network interruptions and failover) are worth retrying; retrying a permanent error (e.g. constraint violation) 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, 2-phase commit can help (which we will discuss in "Atomic commit and two-phase commit (2PC)" on page 344).
- If the client process fails while retrying, the data 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 bugs are only triggered when you get unlucky with the timing. This may happen very rarely, and is 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, this 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 [22].

Concurrency bugs caused by weak transaction isolation are not just a theoretical problem. They have caused substantial loss of money [23, 24], have led to investigation by financial auditors [25] and caused customer data to be corrupted [26]. A popular comment on relevations 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 (non-serializable) 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 is appropriate to your application. Once we’ve done that, we will discuss serializability in detail in the next section (“Serializability” on page 242). We will discuss isolation levels informally, using examples. If you want rigorous definitions and analyses of their properties, you can find them in the academic literature [27, 28, 29].

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 one transaction has written some data to the database, but has not yet committed or aborted. Can another transaction see that uncommitted data? If yes, that is called a *dirty read* [2].

[v] Some databases support an even weaker isolation level called *read uncommitted*. It prevents dirty writes, but does not prevent dirty reads.

Transactions running at *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.

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 was later rolled back, i.e. which was never actually committed to the database. Reasoning about the consequences quickly becomes mind-bending.

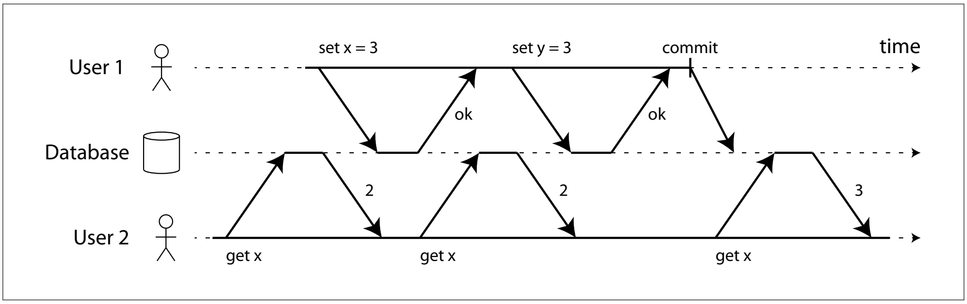


Figure 7-4. No dirty reads: user 2 sees the new value for x only after user 1's transaction has committed.

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* [27]. Transactions running at *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 233 we will discuss how to make such counter increments safe.

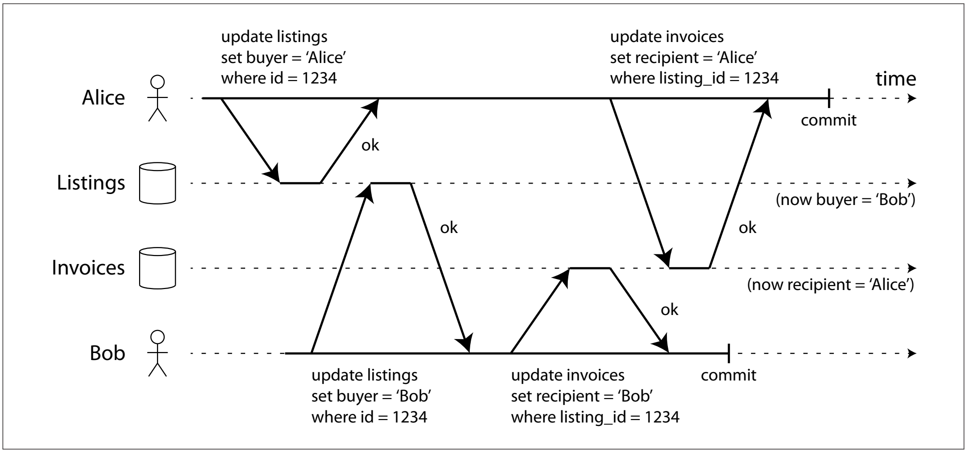


Figure 7-5. With dirty writes, conflicting writes from different transactions can be mixed up.

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 also 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, transactions switch over to reading the new value.

Snapshot isolation and repeatable read

If you look superficially at read committed isolation, you would 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 read committed. For example, Figure 7-6 illustrates a problem that can occur with read committed.

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 another. If she is unlucky 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 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

[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 [22, 35].

appears as though she only has a total of $900 in her accounts — it seems that $100 have vanished into thin air.

This anomaly is called a *non-repeatable 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.

If Alice reloads the online banking website a few seconds later, the $100 will most likely have reappeared, so it's not a lasting problem. 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 87), or may be part of a periodic integrity check that everything is in order (monitoring for data corruption). Such queries are likely to return nonsensical results if they observe parts of the database at different points in time.

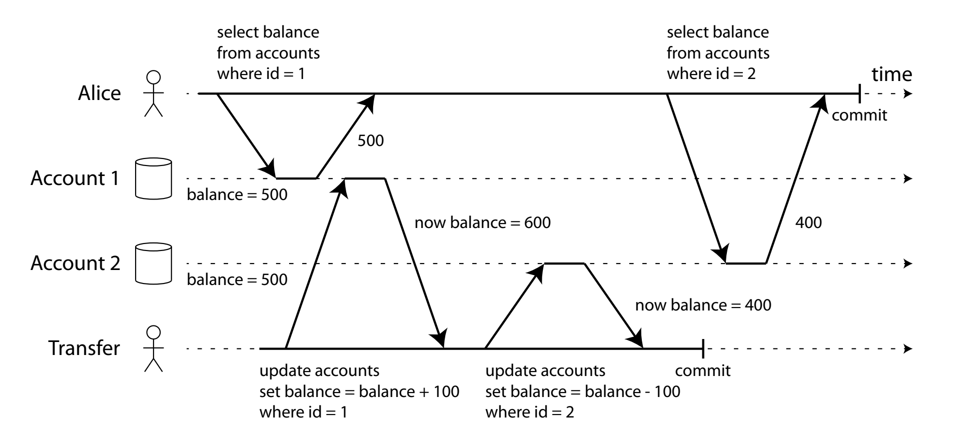

Alice
Account 1
Account 2
Transfer
select balance from accounts where id = 1
500
balance = 500
balance = 500
update accounts set balance = balance + 100 where id = 1
now balance = 600
update accounts set balance = balance - 100 where id = 2
now balance = 400
commit
select balance from accounts where id = 2
400
commit
time

Figure 7-6. Read skew: Alice observes the database in an inconsistent state.

Snapshot isolation, also known as *multiversion concurrency control* (MVCC), is the most common solution to this problem [27]. The idea is that each transaction reads from a *consistent snapshot* of the database, that is, all the data that was committed in the database at a particular point in time. Even if the data is subsequently changed by another transaction, each transaction sees the old data from the time when that transaction started.

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.

Implementing snapshot isolation

Snapshot isolation is a popular feature: it is supported by PostgreSQL, MySQL with the InnoDB storage engine, Oracle, SQL Server, and more [22, 30, 31].

Like read committed, implementations of snapshot isolation typically use write locks to prevent dirty writes (see "Implementing read committed" on page 227), which means that a transaction that makes a write can block the progress of another transaction that writes to the same object. However, locks are not required for reads. 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.

This is implemented as a generalization of the mechanism we saw for preventing dirty reads in Figure 7-4. In order to prevent dirty reads, the database needs to keep at most two versions of an object: the committed version, and the overwritten-but-not-yet-committed version. For snapshot isolation, 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. Hence snapshot isolation is also known as a *multiversion* technique.

Figure 7-7 illustrates how snapshot isolation is implemented in PostgreSQL [30] (other implementations are similar). When a transaction is started, it is given a unique, always-increasing[vii] transaction ID. Whenever a transaction writes anything to the database, the data it writes is tagged with the transaction ID of the writer.

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

[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.

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.

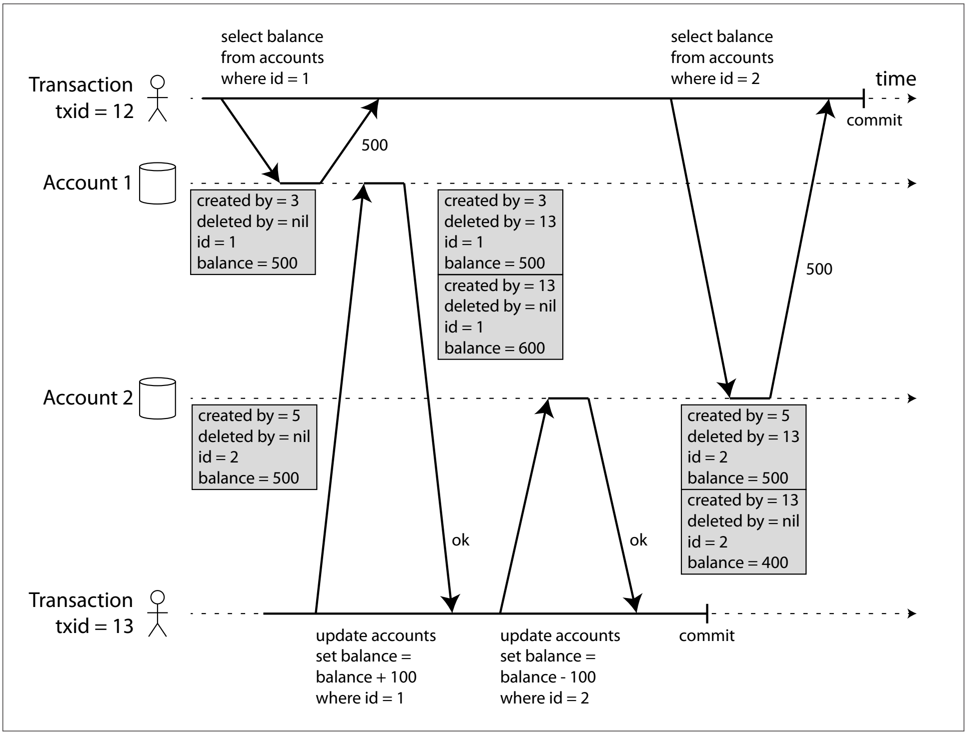


Figure 7-7. Implementing snapshot isolation using multiversion objects.

Visibility rules for observing a consistent snapshot

When a transaction reads from the database, transaction IDs are used to decide which objects a transaction 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 which are in progress (not yet committed or aborted) at that time. Any writes made by one of those transactions are ignored, even if the transaction subsequently commits.
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 that transaction has 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:

- At the time when the reader's transaction started, the transaction which created the object had already committed, and
- The object is not marked for deletion — or if it is, the transaction which 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 which (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 multiversion database? One option is to have the index simply point to all versions of an object, and an index query needs 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 multiversion concurrency control. For example, PostgreSQL has optimizations for avoiding index updates if different versions of the same object can fit on the same page [30].

Another approach is used in CouchDB, Datomic and LMDB. Although they also use B-trees (see "B-trees" on page 77), they use an *append-only/copy-on-write* variant which 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 version of their child pages. Any pages that are not affected by a write do not need to be copied, and remain immutable [32, 33, 34].

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, 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* [22].

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 [27]. Even though several databases implement *repeatable read*, there are big differences in the guarantees they actually provide, despite being ostensibly standardized [22]. There has been a formal definition of *repeatable read* in the research literature [28, 29] 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

Read committed and snapshot isolation, as 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 226), one particular type of write-write conflict that can occur.

There are several other interesting kinds of conflict that can occur between concurrently writing transactions. The most well-known of these is the *lost update* problem, illustrated in Figure 7-1.

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, various different solutions have been developed.

Atomic write operations

Many databases provide atomic update operations, which avoid implementing a read-modify-write cycle in application code. They are usually the best solution if your code can be expressed in terms of those operations. For example, the following is 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 167 for some pointers.

applied. This technique is sometimes known as *cursor stability* [35, 36]. 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 which performs unsafe read-modify-write cycles instead of using atomic operations provided by the database [37]. 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:

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 that a lost update occurred, the transaction is aborted and must 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 transactions. However, MySQL/InnoDB's *repeatable read* does not detect lost updates [22]. Some authors [27, 29] 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 221). The purpose of this operation is to avoid lost updates by allowing an update to only happen 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 only to occur 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 a copy 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 314.)

Instead, as discussed in "Detecting concurrent writes" on page 178, 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 [38].

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 179. Unfortunately, LWW is the default in many replicated databases.

Preventing write skew and phantoms

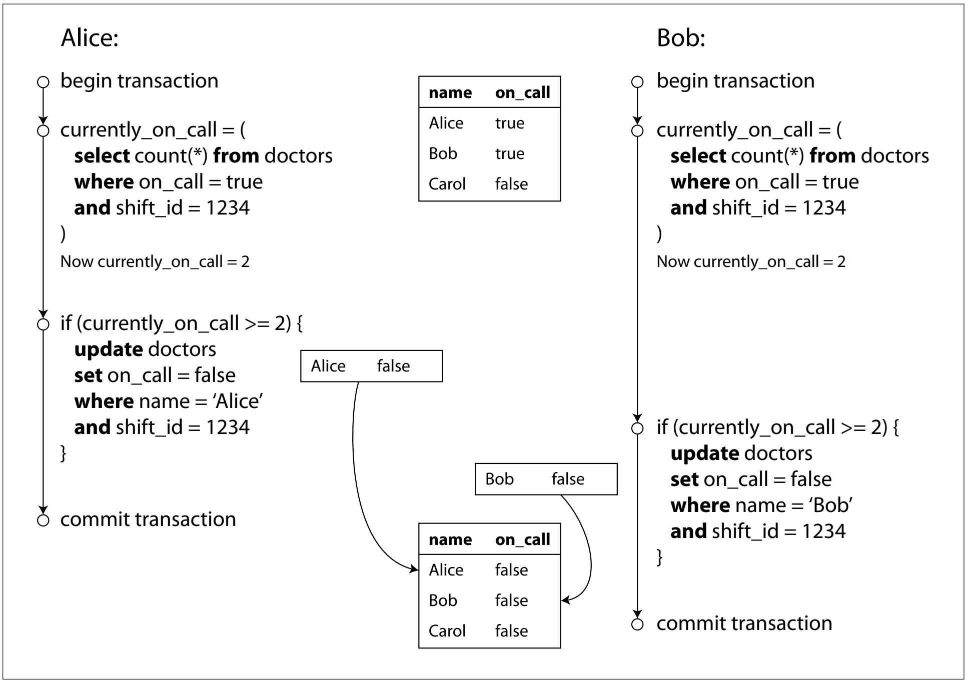
In the previous sections we saw *dirty writes* and *lost updates*, two kinds of race condition that can occur when different transactions concurrently try to write to the same objects. In order to avoid data corruption, those race conditions need to be prevented — either automatically by the database, or by manual safeguards such as using locks or atomic write operations.

However, that is not yet the end of the list of potential race conditions that can occur between concurrent writes. In this section we will see some more subtle 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 shift (e.g. if they are sick themselves) provided that at least one colleague remains on call in that shift [39, 40].

Now imagine that Alice and Bob are the two doctors on-call 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.

In each transaction, your application first checks that two or more doctors are currently on-call; if yes, we think 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.



```
begin transaction

currently_on_call = (
  select count(*) from doctors
  where on_call = true
  and shift_id = 1234
)
Now currently_on_call = 2

if (currently_on_call >= 2) {
  update doctors
  set on_call = false
  where name = 'Alice'
  and shift_id = 1234
}

commit transaction
```

```
begin transaction

currently_on_call = (
  select count(*) from doctors
  where on_call = true
  and shift_id = 1234
)
Now currently_on_call = 2

if (currently_on_call >= 2) {
  update doctors
  set on_call = false
  where name = 'Bob'
  and shift_id = 1234
}

commit transaction
```

name	on_call
Alice	true
Bob	true
Carol	false

Alice	false

Bob	false

name	on_call
Alice	false
Bob	false
Carol	false

Figure 7-8. Example of write skew causing an application bug.

Characterizing write skew

This anomaly is called *write skew* [27]. It is neither a dirty write nor a lost update, because the two transactions are updating two different objects (Alice and Bob's on-call record, 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 lost update. 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, because 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 [22]. Automatically preventing write skew requires true serializable isolation (see "Serializability" on page 242).
- 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. You may be able to implement them with triggers or materialized views, but the result can end up quite hacky [41].
- If you can't use a serializable isolation level, the second-best option in this case is probably to explictly 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 [42]. 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:[ix]

Example 7-2. A meeting room booking system tries to avoid double-booking (not safe under snapshot isolation).

```sql
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 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 two different figures from being moved to the same position on the board, or potentially making some other move which violates the rules of the

[ix] In PostgreSQL you can do this more elegantly using range types, but they are not widely supported in other databases.

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.

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 a 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 [43]. With write skew, it could happen that two spending items are inserted concurrently, which 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 (although the steps may occur in a different order):

1. A `SELECT` query which 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 you wanted 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 to abort).
3. If the application decides to go ahead, it makes a write (`INSERT`, `UPDATE` or `DELETE`) to the database and commits the transaction.
4. If you were to repeat the `SELECT` query from step 1 now, you would now get a different result, because the write in step 3 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).

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 6 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. *repeatable read* means quite different things to different people).

- 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 [25] but research techniques have not yet found their way into practical use. Automated testing for concurrency issues is hard, because they are usually non-deterministic — 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 understand this, 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:

1. Literally executing transactions in a serial order (see "Actual serial execution" on page 243).
2. Two-phase locking (see "Two-phase locking (2PL)" on page 248), which for several decades was the only viable option.
3. Optimistic concurrency control techniques such as serializable snapshot isolation (see "Serializable snapshot isolation (SSI)" on page 252).

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 any concurrency problems is to remove the concurrency entirely: to execute only one transaction at a time, in serial order, on a single thread. This completely sidesteps 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 [44]. If multithreaded concurrency was considered essential for getting good performance during the previous 30 years, what changed to make single-threaded execution feasible?

Two changes caused this rethink:

- RAM has become cheap enough that for many use cases it has become feasible to keep the entire active dataset in memory (see "Keeping everything in memory" on page 85). 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 87). By contrast, long-running analytics 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 [45, 46, 47]. 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; customer decides on 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 mind 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 your 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.

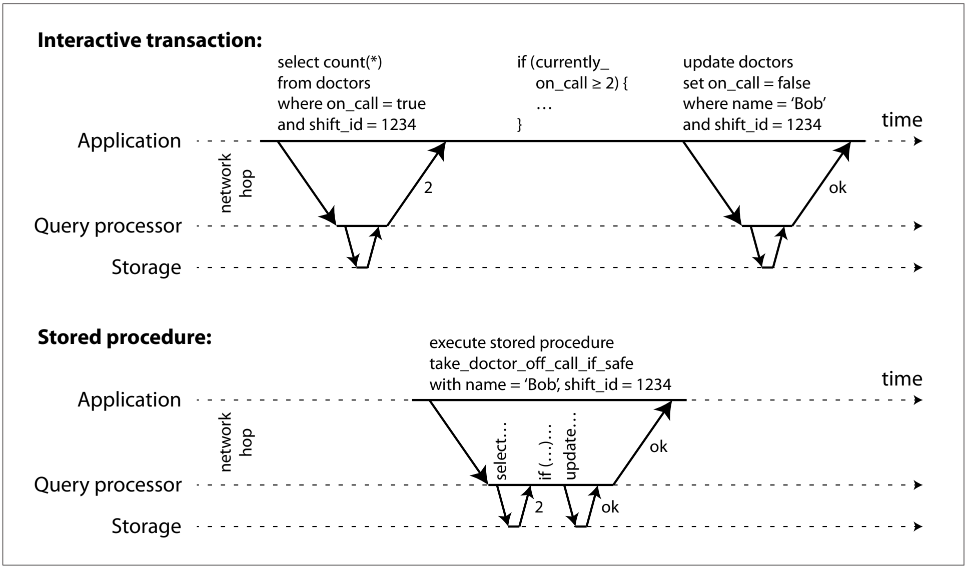


Figure 7-9. The difference between an interactive transaction and a stored procedure (using the example transaction of Figure 7-8).

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 their 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 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 [46].

However, for transactions which need to access multiple partitions, the database must coordinate the transaction across all the partitions that are touched by the transaction. 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. This is orders of magnitude below its single-partition transaction throughput, and cannot be increased by adding more machines [48].

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 197).

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 either low enough to be handled on a single CPU core, or transactions need to be partitioned without requiring cross-partition coordination.

[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 is known as *anti-caching*, as previously mentioned in "Keeping everything in memory" on page 85. It only works if the total amount of data that a query needs to access can be kept in memory.

- Cross-partition transactions are possible, but there is a hard limit to the extent to which they can be used.

Two-phase locking (2PL)

For around 30 years, there was only one widely used algorithm for serializability in databases, and that is *two-phase locking* (2PL).[xi]

> **2PL is not 2PC**
>
> Note that two-phase *locking* (2PL) sounds very similar to two-phase *commit* (2PC), but 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 226): 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, but a reader must also block a writer, and vice versa. Snapshot isolation has the mantra *readers never block writers, and writers never block readers* (see "Implementing snapshot isolation" on page 230), 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 above, including lost updates and write skew.

[xi] Sometimes called *strong strict two-phase locking* (SS2PL) to distinguish it from other variants of 2PL.

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 [22, 35].

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, the transaction 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 (neither in shared nor 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 is called *deadlock*. The database automatically detects deadlock 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 which 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 can be very slow at high percentiles (see "Describing performance" on page 11) 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 even happen with a lock-based *read committed*, deadlocks occur much more frequently under 2PL *serializable* (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 description of locks we glossed over a subtle but important detail. In "Phantoms causing write skew" on page 241 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: 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 ok to concurrently insert bookings for other rooms, or for the same room at a different time which 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 above, 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   > '2015-01-01 12:00' AND
    start_time < '2015-01-01 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 the old or 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 which do not yet exist in the database, but might be added in 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 [40, 49].

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 1pm, 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 1pm. 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 query above 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 1pm on January 1, 2015.

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 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 (i.e. they may lock a bigger range of objects than 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 would not be good for performance, since it would 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 which don't perform well (two-phase locking) or don't scale well (serial execution). On the other hand, we have weak isolation levels which 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 [39] and is the subject of Michael Cahill's PhD thesis [50].

Today SSI is used both in single-node databases (the *serializable* isolation level in PostgreSQL since version 9.1 [40]) and distributed databases (FoundationDB uses a similar algorithm [51]). As SSI so young compared to other concurrency control mechanisms, it is still proving its performance in practice, but it has the chance of being fast enough to become the new default in future.

Pessimistic vs. 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 multithreaded 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 alright. When a transaction wants to commit, the database checks whether anything bad happened (i.e. whether isolation was violated); if so, the transaction is aborted and has to be retried. Only transactions which 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 228). 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 "Preventing write skew and phantoms" on page 237), 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 — i.e. 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 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:

1. Detecting stale MVCC reads (uncommitted write occurred before the read);
2. Detecting writes that affect prior reads (the write occurs after the read).

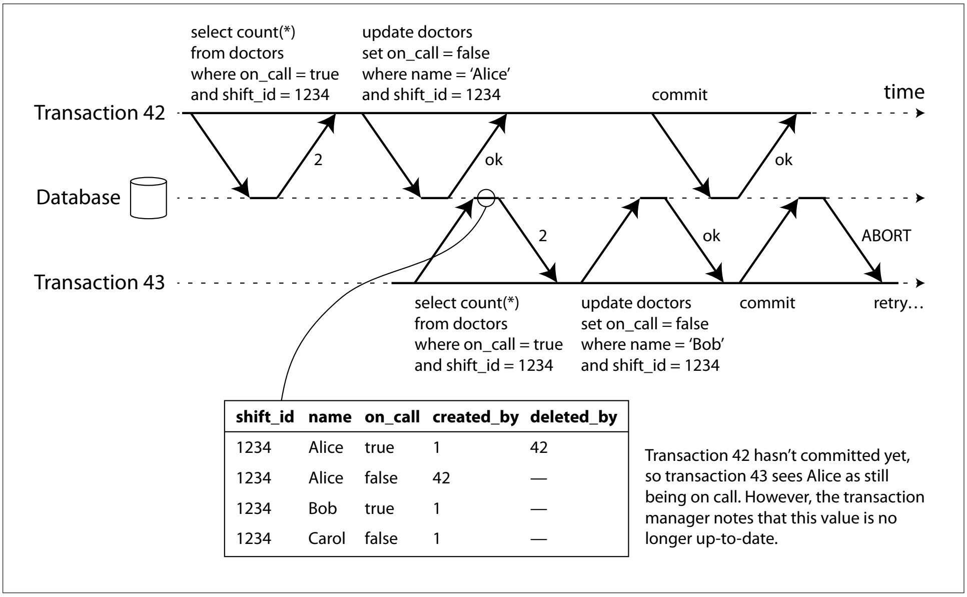
Detecting stale MVCC reads

The first case is illustrated in Figure 7-10. When a transaction reads from a consistent snapshot in a MVCC database, it ignores writes that were made by another transaction which 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 which was ignored when reading from the consistent snapshot has now taken effect, and transaction 43's premise is no longer true.

In order to prevent this, 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 yes, 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. Serializable snapshot isolation needs to support long-running reads from a consistent snapshot, just like regular snapshot isolation, without unnecessary aborts.



shift_id	name	on_call	created_by	deleted_by
1234	Alice	true	1	42
1234	Alice	false	42	—
1234	Bob	true	1	—
1234	Carol	false	1	—

Transaction 42 hasn't committed yet, so transaction 43 sees Alice as still being on call. However, the transaction manager notes that this value is no longer up-to-date.

Figure 7-10. Detecting when a transaction reads outdated values from a MVCC snapshot.

Detecting writes that affect prior reads

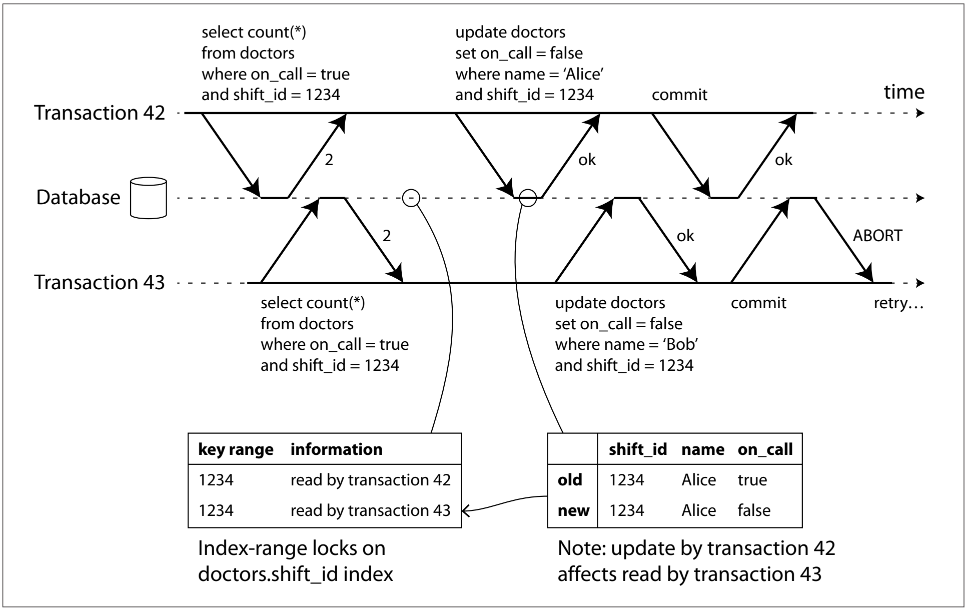
The second case to consider is when another transaction modifies data after it has been read. This case is illustrated in Figure 7-11.

In the context of two-phase locking we discussed index-range locks (see "Index-range locks" on page 251), 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, transaction 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 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.



key range	information
1234	read by transaction 42
1234	read by transaction 43

Index-range locks on doctors.shift_id index

	shift_id	name	on_call
old	1234	Alice	true
new	1234	Alice	false

Note: update by transaction 42 affects read by transaction 43

Figure 7-11. In serializable snapshot isolation, detecting when one transaction modifies another transaction's reads.

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 at 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 ok 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, 40].

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 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 still preserving 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 ok, depending on the implementation). However, SSI is probably less sensitive to slow transactions than two-phase locking or serial execution.

Summary

Transactions are an abstraction layer that allow an application to pretend that certain concurrency problems and certain kinds of hardware and software fault 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.

Many NoSQL systems abandoned transactions in the name of scalability, availability and performance. Unfortunately this means that applications using such data systems either need to implement their own transaction management — which is unlikely, because it's hard to implement correctly — or accept that their data is approximate.

Without transactions, various error scenarios (processes crashing, network interruptions, power outage, 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 effect 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 characterised 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 (non-repeatable reads)
: A client sees different parts of the database at different points in time. This 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.

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.

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, and 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:

1. 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.
2. 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.

3. Serializable snapshot isolation, 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 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 222, transactions are a valuable database feature, no matter which data model is used.

Most of the ideas and algorithms in this chapter apply no matter whether the database is running on a single machine, or replicated and partitioned across multiple machines. However, there is an additional set of difficult challenges that arises if you try to implement transactions in distributed databases. We'll discuss those 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, Elsevier/North Holland Publishing, pages 364–394, 1976. Also in *Readings in Database Systems*, edited by Joseph M. Hellerstein and Michael Stonebraker, 4th edition, 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, 6 July 2009.

[6] Gavin Clarke: "NoSQL's CAP theorem busters: We don't drop ACID," theregister.co.uk, 22 November 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, Joseph M Hellerstein, and Ion Stoica: "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, Eric A Brewer, and Paul Gauthier: "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. Available online at research.microsoft.com. ISBN: 0201107155

[11] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha: "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, 2 June 2015.

[14] Adam Surak: "When Solid State Drives are not that solid," blog.algolia.com, 15 June 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, 14 April 2016.

[17] Lakshmi N Bairavasundaram, Garth R Goodson, Bianca Schroeder, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau: "An Analysis of Data Corruption in the Storage Stack," at *6th USENIX Conference on File and Storage Technologies* (FAST), February 2008.

[18] Don Allison: "SSD Storage — Ignorance of Technology is No Excuse," blog.korelogic.com, 24 March 2015.

[19] Dave Scherer: "Those Are Not Transactions (Cassandra 2.0)," blog.foundationdb.com, 6 September 2013.

[20] Kyle Kingsbury: "Call me maybe: Cassandra," aphyr.com, 24 September 2013.

[21] "ACID Support in Aerospike," Aerospike, Inc., June 2014.

[22] Martin Kleppmann: "Hermitage: Testing the "I" in ACID," martin.kleppmann.com, 25 November 2014.

[23] Tristan D'Agosta: "BTC Stolen from Poloniex," bitcointalk.org, 4 March 2014.

[24] bitcointhief2: "How I stole roughly 100 BTC from an exchange and how I could have stolen more!," reddit.com, 2 February 2014.

[25] 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), pages 1263–1274, September 2007.

[26] Michael Melanson: "Transactions: the limits of isolation," michaelmelanson.net, 20 March 2014.

[27] 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), volume 24, number 2, May 1995. doi:10.1145/568271.223785

[28] Atul Adya: "Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions," PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, March 1999.

[29] 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.

[30] Bruce Momjian: "MVCC Unmasked," momjian.us, July 2014.

[31] Annamalai Gurusami: "Repeatable Read Isolation Level in InnoDB - How Consistent Read View Works," blogs.oracle.com, 15 January 2013.

[32] Nikita Prokopov: "Unofficial guide to Datomic internals," tonsky.me, 6 May 2014.

[33] Baron Schwartz: "Immutability, MVCC, and garbage collection," xaprb.com, 28 November 2013.

[34] J Chris Anderson, Jan Lehnardt, and Noah Slater: *CouchDB: The Definitive Guide*. O'Reilly Media, January 2010. ISBN: 978-0-596-15589-6

[35] Rikdeb Mukherjee: "Isolation in DB2 (repeatable read, read stability, cursor stability, uncommitted read) with examples," mframes.blogspot.co.uk, 4 July 2013.

[36] Steve Hilker: "Cursor Stability (CS) — IBM DB2 Community," toadworld.com, 14 March 2013.

[37] Nate Wiger: "An Atomic Rant," nateware.com, 18 February 2010.

[38] Joel Jacobson: "Riak 2.0: Data Types," blog.joeljacobson.com, 23 March 2014.

[39] Michael J Cahill, Uwe Röhm, and Alan Fekete: "Serializable Isolation for Snapshot Databases," at *ACM International Conference on Management of Data* (SIGMOD), pages 729–738, June 2008. doi:10.1145/1376616.1376690

[40] Dan R K Ports and Kevin Grittner: "Serializable Snapshot Isolation in PostgreSQL," at *38th International Conference on Very Large Databases* (VLDB), volume 5, number 12, pages 1850–1861, August 2012.

[41] Tony Andrews: "Enforcing complex constraints in Oracle," tonyandrews.blogspot.co.uk, 15 October 2004.

[42] 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), pages 172–182, December 1995. doi:10.1145/224056.224070

[43] Gary Fredericks: "Postgres Serializability Bug," github.com, September 2015.

[44] 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), pages 1150–1160, September 2007.

[45] John Hugg: "H-Store/VoltDB architecture vs. CEP systems and newer streaming architectures," at *Data @Scale Boston*, November 2014.

[46] 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.

[47] Rich Hickey: "The Architecture of Datomic," infoq.com, 2 November 2012.

[48] John Hugg: "Debunking Myths about the VoltDB in-memory database," voltdb.com, 12 May 2014.

[49] 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

[50] Michael J Cahill: "Serializable Isolation for Snapshot Databases," PhD Thesis, University of Sydney, July 2009.

[51] FoundationDB: *Concurrency with FoundationDB (Python)*. foundationdb.com, July 2014.

[52] D Z Badal: "Correctness of concurrency control and implications in distributed databases," at *3rd International IEEE Computer Software and Applications Conference* (COMPSAC), pages 588–593, November 1979. doi:10.1109/CMPSAC.1979.762563

[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, 10 December 2014.

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 to discuss how systems handle things going wrong. For example, we discussed replica failover ("Handling node outages" on page 150), replication lag ("Problems With Replication Lag" on page 155) and concurrency control for transactions ("Weak isolation levels" on page 224). 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

[i] With one exception: we will assume that faults are *non-byzantine* — see "Byzantine faults" on page 295.

problems that arise in practice, and an understanding of the things we can and cannot rely on.

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 which 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 269), clocks and timing issues ("Unreliable Clocks" on page 278), 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 happened ("Knowledge, Truth and Lies" on page 291).

Faults and Partial Failures

When you are writing a program on a single computer, it normally behaves in a fairly predictable (*deterministic*) 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. If there is a hardware problem (e.g. memory corruption, 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 return 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!

This non-determinism 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 multitenant 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).

Supercomputers are built from specialized hardware, where each node is quite reliable, and nodes communicate through shared memory and remote direct memory access (RDMA). They often use specialised network topologies, such as multidimensional meshes and toruses [9]. These techniques yield very good performance and sufficient reliability for HPC use cases.

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. This is different from offline (batch) jobs like weather simulations, which can be stopped and restarted with fairly low impact.
- We assume that nodes are built from commodity machines, which can provide equivalent performance at lower cost than specialized hardware (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 [10].
- 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]. If the error handling strategy consists of simply giving up, such a large system would never work.
- 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.

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 4, 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, i.e. 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 re-transmitted, 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.

Unreliable Networks

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 how 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).

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 because someone unplugged the network cable);
2. your request may be waiting in a queue and will be delivered later (perhaps due to the network or the recipient being overloaded);
3. the remote node may have failed (perhaps because it crashed or it was powered down);
4. the remote node may have temporarily stopped responding (perhaps because it is experiencing a long garbage collection pause, see "Process pauses" on page 287), 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 because of a misconfigured switch);
6. the remote node may have processed your request, but the response has been delayed and will be delivered later (perhaps due to the network or your own machine being overloaded).

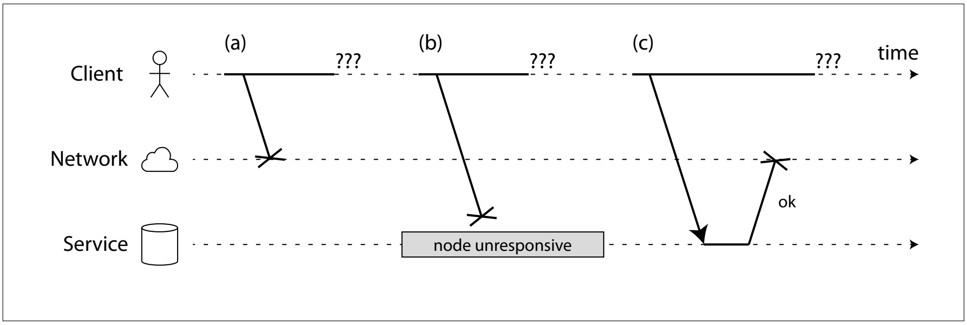


Figure 8-1. If you send a request and don't get a response, it's not possible to distinguish whether (a) the request was lost, (b) the remote node is down, or (c) the response was lost.

The sender can't even tell whether the packet was delivered: the only way to tell whether it arrived is for the recipient to send a response message, which may in turn be lost or delayed. All of the issues above are indistinguishable in an asynchronous network: the only information you have is that you haven't received a response yet. If you send a request to another node and don't receive a response, it is *impossible* to tell why.

The usual way of handling this is a *timeout*: after some time you give up waiting and assume that the response is not going to arrive. However, when a timeout occurs, you still don't know whether the remote node got your request or not (and if the request is still queued somewhere, it may still be delivered to the recipient, even if the sender has given up on it).

Network faults in practice

We have been building computer networks for decades — one might hope that by now we would have figured out how to make them reliable. However, it seems that we have not yet succeeded.

There are some systematic studies, and plenty of anecdotal evidence, showing that network problems can be surprisingly common, even in controlled environments like a datacenter operated by one company [14]. One study in a medium-sized datacenter found about 12 network faults per month, of which half disconnected a single machine, and half disconnected an entire rack [15]. Another study measured the failure rates of components like top-of-rack switches, aggregation switches and load balancers [16]. It found that adding redundant networking gear doesn't reduce faults as much as you'd hope, since it doesn't guard against human error (e.g. misconfigured switches), which is a major cause of outages.

For example, public cloud services such as EC2 are notorious for having frequent transient network glitches [14] and well-managed private datacenter networks can be a more stable environment. Nevertheless, nobody is immune from network problems: for example, a problem during a switch software upgrade could trigger a network topology reconfiguration, during which network packets could be delayed for more than a minute [17]. Other surprising faults include a network interface that sometimes drops all inbound packets, but sends outbound packets successfully [18]: just because a network link works in one direction doesn't guarantee it's also working in the opposite direction.

Network partitions

When one part of the network is cut off from the rest due to a network fault, that is sometimes called a *network partition* or *netsplit*. In this book we'll stick with the more general term *network fault*, to avoid confusion with partitions (shards) of a storage system, as discussed in Chapter 6.

Even if network faults are rare in your environment, the fact that faults *can* occur means that your software needs to be able to handle them. Whenever any communication happens over a network, it may fail — there is no way around it.

If the error handling of network faults is not defined and tested, arbitrarily bad things could happen: for example, the cluster could become deadlocked and permanently unable to serve requests, even when the network recovers [19], or it could even delete all of your data [20]. If software is put in an unanticipated situation, it may do arbitrary unexpected things.

Handling network faults doesn't necessarily mean *tolerating* them: if your network is normally fairly reliable, a valid approach may be to simply show an error message to users while your network is experiencing problems. However, you do need to know how your software reacts to network problems, and ensure that the system can recover from them. It may make sense to deliberately trigger network problems and test the system's response (*chaos monkey*, see "Reliability" on page 4).

Detecting faults

Many systems need to automatically detect faulty nodes, for example:

- a load balancer needs to stop sending requests to a node that is dead (take it *out of rotation*);
- in a distributed database with single-leader replication, if the leader fails, one of the followers needs to be promoted to be the new leader (see "Handling node outages" on page 150).

Unfortunately, the uncertainty about the network makes it difficult to tell whether a node is working or not. In some specific circumstances you might get some feedback to explicitly tell you that something is not working:

- If you can reach the machine on which the node should be running, but no process is listening on the destination port (e.g. because the process crashed), the operating system will helpfully close or refuse TCP connections by sending a `RST` or `FIN` packet in reply. However, if the node crashed while it was handling your request, you have no way of knowing how much data was actually processed by the remote node [21].
- If a node process crashed (or was killed by an administrator), but the node's operating system is still running, a script can notify other nodes about the crash, so that another node can take over quickly without having to wait for a timeout to expire. For example, HBase does this [22].
- If you have access to the management interface of the network switches in your datacenter, you can query them to detect link failures at a hardware level (e.g. if the remote machine is powered down). This option is ruled out if you're connecting via the internet, or if you're in a shared datacenter with no access to the switches themselves, or if you can't reach the management interface due to a network problem.

- If a router is sure that the IP address you're trying to connect to is unreachable, it may reply to you with an ICMP Destination Unreachable packet. However, the router doesn't have a magic failure detection capability either — it is subject to the same limitations as other participants of the network.

Rapid feedback about a remote node being down is useful, but you can't count on it. Even if TCP acknowledges that a packet was delivered, the application may have crashed before handling it. If you want to be sure that a request was successful, you need a positive response from the application itself [23].

Conversely, if something has gone wrong, you may get an error response at some level of the stack, but in general you have to assume that you will get no response at all. You can retry a few times (TCP retries transparently, but you may also retry at the application level), wait for a timeout to elapse, and eventually declare the node dead if you don't hear back within the timeout.

Timeouts and unbounded delays

If a timeout is the only sure way of detecting a fault, then how long should the timeout be? There is unfortunately no simple answer.

A long timeout means a long wait until a node is declared dead (and during this time, users may have to wait, or see error messages). A short timeout detects faults faster, but carries a higher risk of incorrectly declaring a node dead, when in fact it only suffered a temporary slowdown (e.g. due to a load spike on the node or the network).

Prematurely declaring a node dead is problematic: as its responsibilities are transferred to other nodes, additional load is placed on other nodes and the network. If the system is already struggling with high load, declaring nodes dead prematurely can make the problem worse, and even cause a cascading failure (in the extreme case, all nodes may declare each other dead, and everything stops working).

Imagine a fictitious system with a network that guaranteed a maximum delay for packets — every packet is either delivered within some time d, or it is lost, but never takes longer than d. Furthermore, assume that you can guarantee that a non-failed node always handles a request within some time r. In this case, you could guarantee that every successful request receives a response within time $2d + r$ — and if you don't receive a response within that time, you know that either the network or the remote node is not working. If this was true, $2d + r$ would be a reasonable timeout to use.

Unfortunately, most systems we work with have neither of those guarantees: asynchronous networks have *unbounded delays* (that is, they try to deliver packets as quickly as possible, but there is no upper limit on the time the packet may take), and most server implementations cannot guarantee that they can handle requests within some maximum time (see "Response time guarantees" on page 289). For failure detection, it's not sufficient for the system to be fast most of the time: if your timeout is low, it only takes a transient spike in round-trip-times to throw the system off-balance.

Network congestion and queueing

When driving a car, travel times on road networks often vary most due to traffic congestion. Similarly, the variability of packet delays on computer networks is most often due to queueing [24]:

- If several different nodes simultaneously try to send packets to the same destination, the network switch must queue them up and feed them into the destination network link one by one (illustrated in Figure 8-2). On a busy network link, a packet may have to wait a while until it can get a slot (this is called *network congestion*). If there is so much incoming data that the switch queue fills up, the packet is dropped, so it needs to be re-sent — even though the network is functioning fine.
- When a packet reaches the destination machine, if all CPU cores are currently busy, the incoming request from the network is queued by the operating system until the application is ready to handle it. Depending on the load on the machine, this may take an arbitrary length of time.
- In virtualized environments, a running operating system is often paused for tens of milliseconds while another virtual machine uses a CPU core. During this time, the VM cannot consume any data from the network, so the incoming data is queued (buffered) by the virtual machine monitor [25]. This further increases the variability of network delays.
- TCP performs *flow control* (congestion avoidance), in which a node limits its own rate of sending in order to avoid overloading a network link or the receiving node [26]. This means additional queueing at the sender, before the data even enters the network.
- TCP considers a packet to be lost if it is not acknowledged within some timeout (which is calculated from observed round-trip times), and lost packets are automatically retransmitted. Although the application does not see the packet loss and retransmission, it does see the resulting delay (waiting for the timeout to expire, and then waiting for the retransmitted packet to be acknowledged).

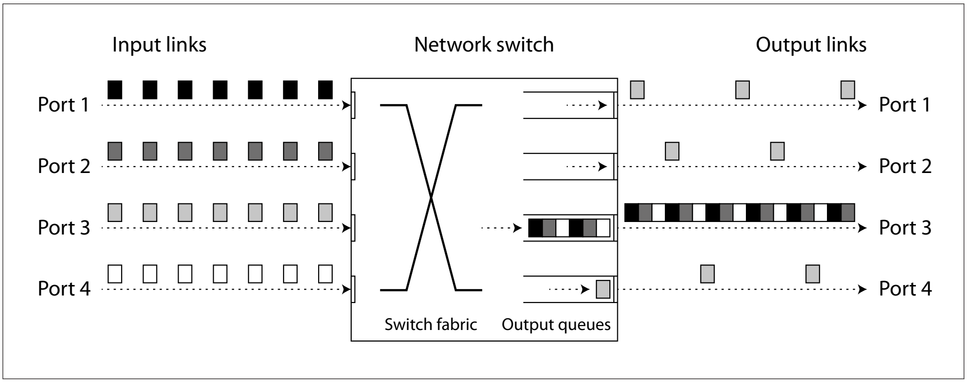


Figure 8-2. If several machines send network traffic to the same destination, its switch queue can fill up. Here, ports 1, 2 and 4 are all trying to send packets to port 3.

TCP versus UDP

Some latency-sensitive applications, such as video conferencing and voice-over-IP, use UDP rather than TCP. It's a trade-off between reliability and variability of delays: as UDP does not perform flow control and does not retransmit lost packets, it avoids some of the reasons for variable network delays (although it is still susceptible to switch queues and scheduling delays).

UDP is a good choice in situations where delayed data is worthless. For example, in a voice-over-IP phone call, there probably isn't enough time to retransmit a lost packet before its data is due to be played over the loudspeakers. In this case, there's no point in retransmitting the packet — the application must instead fill the missing packet's time slot with silence (causing a brief interruption in the sound), and move on in the stream.

All of these factors contribute to the variability of network delays. Queueing delays have an especially wide range when a system is close to its maximum capacity: a system with plenty of spare capacity can easily drain queues, whereas in a highly utilized system, long queues can build up very quickly.

In public clouds and multitenant datacenters, resources are shared among many customers: the network links and switches, and even each machine's network interface and CPUs (when running on virtual machines) are shared. Batch workloads such as MapReduce (see Chapter 10) can easily saturate network links. As you have no control or insight over other customers' usage of the shared resources, network delays can be highly variable if someone near you (a *noisy neighbor*) is using a lot of resources [27, 28].

In such environments, you can only choose timeouts experimentally: measure the distribution of network round-trip times over an extended period, and over many machines, to determine the expected variability of delays. Then, taking into account your application's characteristics, you can determine an appropriate trade-off between failure detection delay and risk of premature timeouts.

Even better, rather than using configured constant timeouts, systems can continually measure response times and their variability (*jitter*), and automatically adjust timeouts according to the observed response time distribution. This can be done with a Phi Accrual failure detector [29], which is used for example in Akka and Cassandra [30]. TCP retransmission timeouts also work similarly [26].

Synchronous vs. asynchronous networks

Distributed systems would be a lot simpler if we could rely on the network to deliver packets with some fixed maximum delay, and not to drop packets. Why can't we solve this at the hardware level, and make the network reliable so that the software doesn't need to worry about it?

To answer this, it's interesting to compare datacenter networks to the traditional fixed-line telephone network (non-cellular, non-Voice-over-IP), which is extremely reliable: delayed audio frames and dropped calls are very rare. A voice call requires a constantly low end-to-end latency, and enough bandwidth to transfer the audio samples of your voice. Wouldn't it be nice to have similar reliability and predictability in computer networks?

When you make a call over the telephone network, it establishes a *circuit*: a fixed, guaranteed amount of bandwidth is allocated for the call, along the entire route between the two callers. This circuit remains in place until the call ends [31]. For example, an ISDN network runs at a fixed rate of 4,000 frames per second. When a call is established, it is allocated 16 bits of space within each frame (in each direction). Thus, for the duration of the call, each side is guaranteed to be able to send exactly 16 bits of audio data every 250 microseconds [32, 33]

This kind of network is *synchronous*: even as data passes through several routers, it does not suffer from queueing, because the 16 bits of space for the call have already been reserved in the next hop of the network. And because there is no queueing, the maximum end-to-end latency of the network is fixed. We call this a *bounded delay*.

Can we not simply make network delays predictable?

Note that a circuit in a telephone network is very different from a TCP connection: a circuit is a fixed amount of reserved bandwidth which nobody else can use while the circuit is established, whereas the packets of a TCP connection opportunistically use whatever network bandwidth is available. You can give TCP a variable-sized block of data (e.g. a web page), and it will try to transfer it in the shortest time possible. While a TCP connection is idle, it doesn't use any bandwidth.[ii]

If datacenter networks and the internet were circuit-switched networks, it would be possible to establish a guaranteed maximum round-trip-time when a circuit was set up. However, they are not: Ethernet and IP are packed-switched protocols, which suffer from queueing and thus unbounded delays in the network.

Why do datacenter networks and the internet use packet switching? The answer is that they are optimized for *bursty traffic*. A circuit is good for an audio or video call, which needs to transfer a fairly constant number of bits per second for the duration of the call. On the other hand, requesting a web page, sending an email or transferring a file doesn't have any particular bandwidth requirement — we just want it to complete as quickly as possible.

If you wanted to transfer a file over a circuit, you would have to guess a bandwidth allocation. If you guess too low, the transfer is unnecessarily slow, leaving network capacity unused. If you guess too high, the circuit cannot be set up (because the network cannot allow a circuit to be created if its bandwidth allocation cannot be guaranteed). Thus, using circuits for bursty data transfers wastes network capacity and makes transfers unnecessarily slow. By contrast, TCP dynamically adapts the rate of data transfer to the available network capacity.

There have been some attempts to build hybrid networks that support both circuit-switching and packet-switching, e.g. ATM.[iii] InfiniBand has some similarities [34]: it implements end-to-end flow control at the link layer, which reduces the need for queueing in the network, although it can still suffer from delays due to link congestion [35]. With careful use *quality of service* (QoS, prioritization and scheduling of packets) and *admission control* (rate-limiting senders), it is possible to emulate circuit-switching on packet networks, or provide statistically bounded delay [24, 31].

However, such quality of service is currently not enabled in multitenant datacenters and public clouds, or when communicating via the internet.[iv] Currently deployed technology does not allow us to make any guarantees about delays or reliability of the

[ii] Maybe except for an occasional keepalive packet, if TCP keepalive is enabled.

[iii] *Asynchronous Transfer Mode* (ATM) was a competitor to Ethernet in the 1980s [31], but it didn't gain much adoption outside of telephone network core switches. It has nothing to do with Automatic Teller Machines (also known as cash machines), despite sharing an acronym. Perhaps, in some parallel universe, the internet is based on something like ATM — in that universe, internet video calls are probably a lot more reliable than they are in ours, because they don't suffer from dropped and delayed packets.

[iv] Peering agreements between internet service providers, and the establishment of routes through BGP, bear closer resemblance to circuit switching than IP itself. At this level, it is possible to buy dedicated bandwidth. However, internet routing operates at the level of networks, not individual connections between hosts, and at a much longer timescale.

network: we have to assume that network congestion, queueing and unbounded delays will happen. Consequently, there's no "correct" value for timeouts — they need to be determined experimentally.

Latency and resource utilization

More generally, you can think of variable delays as a consequence of dynamic resource partitioning.

Say you have a wire between two telephone switches that can carry up to 10,000 simultaneous calls. Each circuit that is switched over this wire occupies one of those call slots. Thus you can think of the wire as a resource that can be shared by up to 10,000 simultaneous users. The resource is divided up in a *static* way: even if you're the only call on the wire right now, and all other 9,999 slots are unused, your circuit is still allocated the same fixed amount of bandwidth as when the wire is fully utilized.

By contrast, the internet shares network bandwidth *dynamically*. Senders push and jostle with each other to get their packets over the wire as quickly as possible, and the network switches decide which packet to send (i.e. the bandwidth allocation) from one moment to the next. This has the downside of queueing, but the advantage is that it maximizes utilization of the wire. The wire has a fixed cost, so if you utilize it better, each byte you send over the wire is cheaper.

A similar situation arises with CPUs: if you share each CPU core dynamically between several threads, one thread sometimes has to wait in the operating system's run queue while another thread is running, so a thread can be paused for varying lengths of time. However, this utilizes the hardware better than if you allocated a static number of CPU cycles to each thread. Better hardware utilization is also a significant motivation for using virtual machines.

Latency guarantees are achievable in certain environments, if resources are statically partitioned (e.g. dedicated hardware and exclusive bandwidth allocations). However, it comes at the cost of reduced utilization — in other words, it is more expensive. On the other hand, multitenancy with dynamic resource partitioning provides better utilization, so it is cheaper, but it has the downside of variable delays.

Variable delays in networks are not a law of nature, but simply the result of a cost-benefit trade-off.

Unreliable Clocks

Clocks and time are important. Applications depend on clocks in various different ways, for example:

- Has this request timed out yet?

- What's the 99th percentile response time of this service?
- How many queries per second did this service handle on average in the last five minutes?
- At what date and time should the reminder email be sent?
- When does this cache entry expire?
- What is the timestamp on this error message in the log file?

The first three examples measure a *duration* (e.g. the time interval between a request being sent and a response being received), whereas the last three describe a *point in time* (an event that occurs on a particular date, at a particular time).

In a distributed system, time is a tricky business, because communication is not instantaneous: it takes time for a message to travel across the network from one machine to another. The time when a message is received is always later than the time when it is sent, but due to variable delays in the network, we don't know how much later. This fact sometimes makes it difficult to determine the order in which things happened when multiple machines are involved.

Moreover, each machine on the network has its own clock, which is an actual hardware device: usually a quartz crystal oscillator. These devices are not perfectly accurate, so each machine has its own notion of time, which may be slightly faster or slower than on other machines. It is possible to synchronize clocks to some degree: most commonly used is the Network Time Protocol (NTP), which allows the computer clock to be adjusted according to the time reported by a group of servers [36]. The servers in turn get their time from a more accurate time source, such as a GPS receiver.

Monotonic vs. time-of-day clocks

Modern computers have at least two different kinds of clock: a *time-of-day clock* and a *monotonic clock*. Although they both measure time, it is important to distinguish the two, since they serve different purposes.

Time-of-day clocks

A time-of-day clock does what you intuitively expect of a clock: it returns the current date and time according to some calendar (also known as *wall-clock time*). For example, `clock_gettime(CLOCK_REALTIME)` on Linux[v] and `System.currentTimeMillis()` in Java return the number of seconds (or milliseconds) since the *epoch*: midnight

[v] Although the clock is called *real-time*, it has nothing to do with real-time operating systems, as discussed in "Response time guarantees" on page 289.

UTC on January 1, 1970, according to the Gregorian calendar, not counting leap seconds. Some systems use other dates as their reference point.

Time-of-day clocks are usually synchronized with NTP, which means that a timestamp from one machine (ideally) means the same as a timestamp on another machine. However, time-of-day clocks also have various oddities, as described in the next section. In particular, if the local clock is too far ahead of the NTP server, it may be forcibly reset and appear to jump back to a previous point in time time. These jumps, as well as the fact that they often ignore leap seconds, make time-of-day clocks unsuitable for measuring elapsed time.

Time-of-day clocks have also historically had quite a coarse-grained resolution, e.g. moving forward in steps of 10 ms on older Windows systems [37]. On recent systems, this is less of a problem.

Monotonic clocks

A monotonic clock is suitable for measuring a duration (time interval), such as a timeout or a service's response time. For example, `clock_gettime(CLOCK_MONOTONIC)` on Linux and `System.nanoTime()` in Java are monotonic clocks. The name comes from the fact that they are guaranteed to always move forwards (whereas a time-of-day clock may jump back in time).

You can check the value of the monotonic clock at one point in time, do something, and then check the clock again at a later time. The *difference* between the two values tells you how much time elapsed between the two checks. However, the *absolute* value of the clock is meaningless: it might be the number of nanoseconds since the computer was started, or something similarly arbitrary. In particular, it makes no sense to compare monotonic clock values from two different computers, because they don't mean the same thing.

On a server with multiple CPU sockets, there may be a separate timer per CPU, which is not necessarily synchronized with other CPUs. Operating systems compensate for this, and try to present a monotonic view of the clock to application threads, even as they are scheduled across different CPUs. However, it is wise to take this guarantee of monotonicity with a pinch of salt [38].

NTP may adjust the frequency at which the monotonic clock moves forward (this is known as *slewing* the clock) if it detects that the computer's local quartz is moving faster or slower than the NTP server. By default, NTP allows the clock rate to be speeded up or slowed down by up to 0.05%, but NTP cannot cause the monotonic clock to jump forwards or backwards. The resolution of monotonic clocks is usually quite good: on most systems they can measure time intervals in microseconds or less.

In a distributed system, using a monotonic clock for measuring elapsed time (e.g. timeouts) is usually fine, because it doesn't assume any synchronization between different nodes' clocks, and is not sensitive to slight inaccuracies of measurement.

Clock synchronization and accuracy

Monotonic clocks don't need synchronization, but time-of-day clocks need to be set according to an NTP or other reliable time source in order to be useful. Unfortunately, this isn't nearly as reliable or accurate as you might hope — hardware clocks and NTP can be fickle beasts. To give just a few examples:

- The quartz clock in a computer is not very accurate: it *drifts* (runs faster or slower than it should). Clock drift varies depending on the temperature of the machine. Google assumes a clock drift of 200 ppm (parts per million) for their servers [39], which is equivalent to 6 ms drift for a clock that is resynchronized with a server every 30 seconds, or 17 seconds drift for a clock that is resynchronized once a day. This limits the best possible accuracy you can achieve, even if everything is working correctly.
- If a computer's clock differs too much from an NTP server, it may refuse to synchronize, or the local clock will be forcibly reset [36]. Any applications observing the time before and after this reset may see time go backwards or suddenly jump forwards.
- If a node is accidentally firewalled off from the NTP server, the misconfiguration may go unnoticed for some time. Anecdotal evidence suggests that this does happen in practice.
- NTP synchronization can only be as good as the network delay, so there is a limit to its accuracy when you're on a congested network with variable packet delays. One experiment showed that a minimum error of 35 ms is achievable when synchronizing over the internet [40], though occasional spikes in network delay lead to errors of around a second. Depending on the configuration, large network delays can cause the NTP client to give up entirely.
- Some NTP servers are wrong or misconfigured, reporting time that is off by hours [41, 42]. NTP clients are quite robust, because they query several servers and ignore outliers. Nevertheless, it's somewhat worrying to bet the correctness of your systems on the time that you were told by a stranger on the internet.
- Leap seconds result in a minute that is 59 seconds or 61 seconds long, which messes up timing assumptions in systems that are not designed with leap seconds in mind [43]. The fact that leap seconds have crashed many large systems [44] shows how easy it is for incorrect assumptions about clocks to sneak into a system. The best way of handling leap seconds may be to make NTP servers "lie", by

performing the leap second adjustment gradually over the course of a day (this is known as *smearing*) [45, 46].

- In virtual machines, the hardware clock is virtualized, which raises additional challenges for applications that need accurate timekeeping [47]. When a CPU core is shared between virtual machines, each VM is paused for tens of milliseconds while another VM is running. This manifests itself as the clock suddenly jumping forwards [25].
- If you run software on devices that you don't fully control (e.g. mobile or embedded devices), you probably cannot trust the device's hardware clock at all. Some users deliberately set their hardware clock to an incorrect date and time, for example to circumvent timing limitations in games. As a result, the clock might be set to a time wildly in the past or the future.

It is possible to achieve very good clock accuracy if you care about it sufficiently to invest significant resources. For example, the MiFID II draft European regulation for financial institutions requires all high-frequency trading funds to synchronize their clocks within 100 microseconds of UTC, in order to help debug market anomalies such as "flash crashes", and to help detect market manipulation [48].

Such accuracy can be achieved using GPS receivers, the Precision Time Protocol [49] and careful deployment and monitoring. However, it requires significant effort and expertise, and there are plenty of ways how clock synchronization can go wrong. If your NTP daemon is misconfigured, or a firewall is blocking NTP traffic, the clock error due to drift can quickly become large.

Relying on synchronized clocks

The problem with clocks is that while they seem simple and easy to use, they have a surprising number of pitfalls: a day may not have exactly 86,400 seconds, time-of-day clocks may move backwards in time, and the time on one node may be quite different from the time on another node.

Earlier in this chapter we discussed networks dropping and arbitrarily delaying packets. Even though networks are well-behaved most of the time, software must be designed on the assumption that the network will occasionally be faulty, and the software must handle such faults gracefully. The same is true with clocks: although they work quite well most of the time, robust software needs to be prepared to deal with incorrect clocks.

Part of the problem is that incorrect clocks easily go unnoticed. If a machine's CPU is defective or its network is misconfigured, it most likely won't work at all, so it will quickly be noticed and fixed. On the other hand, if its quartz clock is defective or its NTP client is misconfigured, most things seem to work fine, even though its clock gradually drifts further and further away from reality. If some piece of software is relying on an accurately synchronized clock, the result is more likely to be silent and subtle data loss than a dramatic crash [50, 51].

Thus, if you use software that requires synchronized clocks, it is essential that you also carefully monitor the clock offsets between all the machines. Any nodes whose clock drifts too far from the others should be declared dead and removed from the cluster. This ensures that you notice the broken clocks before they can cause too much damage.

Timestamps for ordering events

Let's consider one particular situation in which it is tempting, but dangerous, to rely on clocks: ordering of events across multiple nodes. For example, if two clients write to a distributed database, who got there first? Which write is the more recent one?

Figure 8-3 illustrates a dangerous use of time-of-day clocks in a database with multi-leader replication (the example is similar to Figure 5-9). Client A writes $x = 1$ on node 1; the write is replicated to node 3; client B increments x on node 3 (we now have $x = 2$); and finally, both writes are replicated to node 2.

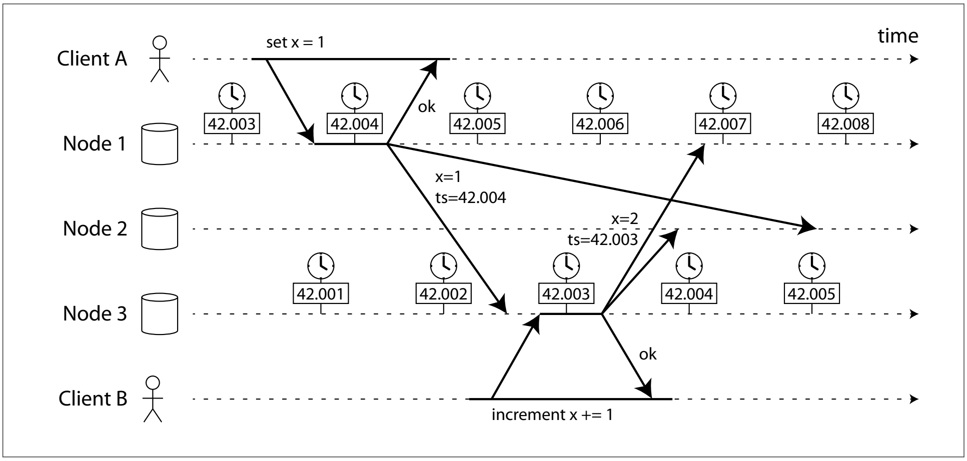


Figure 8-3. The write by client B is causally later than the write by client A, but B's write has an earlier timestamp.

In Figure 8-3, when a write is replicated to other nodes, it is tagged with a timestamp according to the time-of-day clock on the node where the write originated. The clock synchronization is very good in this example: the skew between node 1 and node 3 is less than 3 ms, which is probably better than you can expect in practice.

Nevertheless, the timestamps in Figure 8-3 fail to order the events correctly: the write $x = 1$ has a timestamp of 42.004 seconds, but the write $x = 2$ has a timestamp of 42.003 seconds, even though $x = 2$ occurred unambiguously later. When node 2 receives these two events, it will incorrectly conclude that $x = 1$ is the more recent value, and drop the write $x = 2$. In effect, client B's increment operation has been lost.

This conflict resolution strategy is called *last write wins* (LWW), and it is widely used in both multi-leader replication and leaderless databases such as Cassandra [50] and Riak [51] (see "Last write wins (discarding concurrent writes)" on page 179). Some implementations generate timestamps on the client rather than the server, but this doesn't change the fundamental problems with LWW:

- Database writes can mysteriously disappear: a node with a lagging clock is unable to overwrite values previously written by a node with a fast clock until the clock skew between the nodes has elapsed [51, 52]. This can cause arbitrary amounts of data to be silently dropped without any error being reported to the application.
- LWW cannot distinguish between writes that occurred sequentially in quick succession (in Figure 8-3, client B's increment definitely occurs *after* client A's write) and writes that were truly concurrent (neither writer was aware of the other). Additional causality tracking mechanisms, such as version vectors, are needed in order to prevent violations of causality (see "Detecting concurrent writes" on page 178).
- It is possible for two nodes to independently generate writes with the same timestamp, especially when the clock only has millisecond resolution. An additional tiebreaker value (which can simply be a large random number) is required to resolve such conflicts, but this can also lead to violations of causality [50].

Thus, even though it is tempting to resolve conflicts by keeping the most 'recent' value and discarding others, it's important to be aware that the definition of 'recent' depends on a local time-of-day clock, which may well be incorrect. Even with tightly NTP-synchronized clocks, you could send a packet at timestamp 100 ms (according to the sender's clock), and have it arrive at timestamp 99 ms (according to the recipient's clock) — so it appears as though the packet arrived before it was sent, which is impossible.

Could NTP synchronization be made accurate enough that such incorrect orderings cannot occur? Probably not, because NTP's synchronization accuracy is itself limited by the network round-trip time, in addition to other sources of error such as quartz drift. For correct ordering, you would need the clock source to be significantly more accurate than the thing you are measuring (namely network delay).

So-called *logical clocks* [53], which are based on incrementing counters rather than an oscillating quartz crystal, are a safer alternative for ordering events (see "Detecting concurrent writes" on page 178). Logical clocks do not measure the time of day, only the relative ordering of events (whether one event happened before or after another). Regular time-of-day clocks are also known as *physical clocks*, in contrast to logical clocks. We'll look at ordering a bit more in "Ordering Guarantees" on page 329.

Clock readings have a confidence interval

You may be able to read a machine's time-of-day clock with microsecond or even nanosecond resolution. But even if you can get such a fine-grained measurement, that doesn't mean the value is actually accurate to such precision. In fact, it most likely is not — as mentioned above, clock drift from an imprecise quartz can easily be several milliseconds, even if you synchronize with an NTP server on the local network every minute. With an NTP server on the public internet, the best possible accuracy is probably in the tens of milliseconds, and the error may easily spike to over 100ms when there is network congestion [53].

Thus, it doesn't make sense to think of a clock reading as a point in time — it is more like a range of times, within a confidence interval: for example, a system may be 95% confident that the time now is between 10.3 and 10.5 seconds past the minute, but it doesn't know any more precisely than that [54]. If we only know the time +/– 100 ms, the microsecond digits in the timestamp are essentially meaningless.

The uncertainty bound can be calculated based on your time source. If you have a GPS receiver or atomic (caesium) clock directly attached to your computer, the expected error range is reported by the manufacturer. If you're getting the time from a server, the uncertainty is based on the expected quartz drift since your last sync with the server, plus the NTP server's uncertainty, plus the network round-trip time to the server (to a first approximation, and assuming you trust the server).

Unfortunately, most systems don't expose this uncertainty: for example, when you call `clock_gettime()`, the return value doesn't tell you the expected error of the timestamp, so you don't know if its confidence interval is 5 milliseconds or 5 years.

An interesting exception is Google's *TrueTime* API in Spanner [39], which explicitly reports the confidence interval on the local clock. When you ask it for the current time, you get back two values: [*earliest*, *latest*], which are the *earliest possible* and the *latest possible* timestamp. Based on its uncertainty calculations, the clock knows that the actual current time is somewhere within that interval. The width of the interval depends, among other things, on how long it has been since the local quartz clock was last synchronized with a more accurate clock source.

Synchronized clocks for global snapshots

In "Snapshot isolation and repeatable read" on page 228 we discussed *snapshot isolation*, which is a very useful feature in databases that need to support both small, fast read-write transactions, and also large, long-running read-only transactions (e.g. for backups or analytics). It allows read-only transactions to see the database in a consistent state at a particular point in time, without locking and interfering with read-write transactions.

The most common implementation of snapshot isolation requires a monotonically increasing transaction ID. If a write happened later than the snapshot (i.e. the write has a greater transaction ID than the snapshot), that write is invisible to the snapshot transaction. On a single-node database, a simple counter is sufficient for generating transaction IDs.

However, when a database is distributed across many machines, potentially in multiple datacenters, a global, monotonically increasing transaction ID (across all partitions) is difficult to generate, because it requires coordination. The transaction ID must reflect causality: if transaction B reads a value that was written by transaction A, then B must have a higher transaction ID than A — otherwise the snapshot would not be consistent. With lots of small, rapid transactions, creating transaction IDs in a distributed system becomes an untenable bottleneck.[vi]

Can we use the timestamps from synchronized time-of-day clocks as transaction IDs? If we could get the synchronization good enough, they would have the right properties: later transactions have a higher timestamp. The problem, of course, is the uncertainty about clock accuracy.

Spanner implements distributed snapshot isolation across datacenters in this way [55, 56]. It uses the clock's confidence interval as reported by the TrueTime API, and is based on the following observation: if you have two confidence intervals ($A = [A_{earliest}, A_{latest}]$ and $B = [B_{earliest}, B_{latest}]$) and those two intervals do not overlap (i.e. $A_{earliest} < A_{latest} < B_{earliest} < B_{latest}$), then B definitely happened after A — there can be no doubt. Only if the intervals overlap, we are unsure in which order A and B happened.

In order to ensure that transaction timestamps reflect causality, Spanner deliberately waits for the length of the confidence interval before committing a read-write transaction. By doing so, it ensures that any transaction that may read the data is at a sufficiently later time, so their confidence intervals do not overlap. In order to keep the wait time as short as possible, Spanner needs to keep the clock uncertainty as small as possible; for this purpose, Google deploys a GPS receiver or atomic clock in each datacenter, allowing clocks to be synchronized to within about 7 ms [39].

Using clock synchronization for distributed transaction semantics is an area of active research [53, 57, 58]. These ideas are interesting, but they have not yet been implemented in mainstream databases outside of Google.

[vi] There are distributed sequence number generators such as Twitter's Snowflake, which generate *approximately* monotonically increasing unique IDs in a scalable way (e.g. by allocating blocks of the ID space to different nodes). However, they typically cannot guarantee an ordering that is consistent with causality, because the timescale at which blocks of IDs are assigned is longer than the timescale of database reads and writes. See also "Ordering Guarantees" on page 329.

Process pauses

Let's consider another example of dangerous clock use in a distributed system. Say you have a database with a single leader per partition. Only the leader is allowed to accept writes. How does a node know that it is still leader (that it hasn't been declared dead by the others), and that it may safely accept writes?

One option is for the leader to obtain a *lease* from the other nodes, which is similar to a lock with a timeout [59]. Only one node can hold the lease at any one time — thus, when a node obtains a lease, it knows that it is the leader for some amount of time, until the lease expires. In order to remain leader, the node must periodically renew the lease before it expires. If the node fails, it stops renewing the lease, so another node can take over when it expires.

You can imagine the request-handling loop looking something like this:

```
while (true) {
    request = getIncomingRequest();

    // Ensure that the lease always has at least 10 seconds remaining
    if (lease.expiryTimeMillis - System.currentTimeMillis() < 10000) {
        lease = lease.renew();
    }

    if (lease.isValid()) {
        process(request);
    }
}
```

What's wrong with this code? Firstly, it's relying on synchronized clocks: the expiry time on the lease is set by a different machine (where the expiry may be calculated as the current time plus 30 seconds, for example), and it's being compared to the local system clock. If the clocks are out of sync by more than a few seconds, this code will start doing strange things.

Secondly, even if we change the protocol to only use the local monotonic clock, there is another problem: the code assumes that very little time passes between the point that it checks the time (`System.currentTimeMillis()`) and the time when the request is processed (`process(request)`). Normally this code runs very quickly, so the 10 second buffer is more than enough to ensure that the lease doesn't expire in the middle of processing a request.

However, what if there is an unexpected pause in the execution of the program? For example, imagine the thread stops for 15 seconds around the line `lease.isValid()`, before finally continuing. In that case, it's likely that the lease will have expired by the time the request is processed, and another node has already taken over as leader. However, there is nothing to tell this thread that it was paused for so long, so this code won't notice that the lease has expired until the next iteration of the loop — by which time it may have already done something unsafe by processing the request.

Is it crazy to assume that a thread might be paused for so long? Unfortunately not. There are various reasons why this could happen:

- Many programming language runtimes (such as the Java Virtual Machine) have a *garbage collector* (GC) that occasionally needs to stop all running threads. These *"stop-the-world" GC pauses* have sometimes been known to last for several minutes [60]! Even so-called "concurrent" garbage collectors like the HotSpot JVM's CMS cannot fully run in parallel with the application code — even they need to stop the world from time to time [61]. Although the pauses can often be reduced by changing allocation patterns or tuning GC settings [62], we must assume the worst if we want to offer robust guarantees.
- In virtualized environments, a virtual machine can be *suspended* (pausing the execution of all processes, and saving the contents of memory to disk) and *resumed* (restoring the contents of memory and continuing execution). This pause can occur at any time in a process execution, and can last for an arbitrary length of time. This feature is sometimes used for *live migration* of virtual machines from one host to another without reboot, in which case the length of the pause depends on the rate at which processes are writing to memory [63].
- On end-user devices such as laptops, execution may also be suspended and resumed arbitrarily, e.g. when the user closes the lid of their laptop.
- When the operating system context-switches to another thread, or when the hypervisor switches to a different virtual machine (when running in a virtual machine), the currently running thread can be paused at any arbitrary point in the code. In the case of a virtual machine, the CPU time spent in other virtual machines is known as *steal time*. If the machine is under heavy load, i.e. if there is a long queue of threads waiting to run, it may take some time before the paused thread gets to run again.
- If the application performs synchronous disk access, a thread may be paused waiting for a slow disk I/O operation to complete [64]. In many languages, disk access can happen surprisingly, even if the code doesn't explicitly mention file access — for example, the Java classloader lazily loads class files when they are first used, which could happen at any time in the program execution. I/O pauses and GC pauses may even conspire to combine their delays [65]. If the disk is actually a network file system or network block device (such as Amazon's EBS), the I/O latency is further subject to the variability of network delays [28].
- If the operating system is configured to allow *swapping to disk* (*paging*), a simple memory access may result in a page fault that requires a page from disk to be loaded into memory. The thread is paused while this slow I/O operation takes

place. If memory pressure is high, this may in turn require a different page to be swapped out to disk. In extreme circumstances, the operating system may spend most of its time swapping pages in and out of memory, and getting little actual work done (this is known as *thrashing*). To avoid this, paging is often disabled on server machines (if you would rather kill a process if it runs out of memory, rather than risk thrashing).

- A Unix process can be paused by sending it the `SIGSTOP` signal, which you can do for example by pressing Ctrl+Z in a shell. This immediately stops the process from getting any more CPU cycles until it is resumed with `SIGCONT`, at which point it continues running where it left off. Even if your environment does not normally use `SIGSTOP`, you can imagine the signal being sent accidentally by an operations engineer.

All of these issues can *preempt* the running thread at any point, and resume it at some later time, without the thread even noticing. The problem is similar to making multi-threaded code on a single machine thread-safe: you can't assume anything about timing, because arbitrary context switches and parallelism may occur.

When writing multi-threaded code on a single machine, we have fairly good tools for making it thread-safe: mutexes, semaphores, atomic counters, lock-free data structures, blocking queues, and so on. Unfortunately, these tools don't directly translate to distributed systems, because a distributed system has no shared memory — only messages sent over an unreliable network.

A node in a distributed system must assume that its execution can be paused for a significant length of time at any point, even in the middle of a function. During the pause, the rest of the world keeps moving, and may even declare the paused node dead, because it's not responding. Eventually, the paused node may continue running, without even noticing that it was asleep, until it checks its clock sometime later.

Response time guarantees

In many programming languages and operating systems, threads and processes may pause for an unbounded amount of time, as discussed. Those reasons for pausing *can* be eliminated if you try hard enough.

Some software runs in environments where a failure to respond within a specified time can cause serious damage: computers that control aircraft, rockets, robots, cars and other physical objects must respond quickly and predictably to their sensor inputs. In these systems, there is a specified *deadline* by which the software must respond; if it doesn't meet the deadline, that may cause a failure of the entire system. These are so-called *hard real-time* systems.[vii]

For example, if your car's onboard sensors detect that you are currently experiencing a crash, you wouldn't want the release of the airbag to be delayed due to an inopportune GC pause in the airbag release system.

Providing real-time guarantees in a system requires support from all levels of the software stack: a *real-time operating system* (RTOS) that allows processes to be scheduled with a guaranteed allocation of CPU time in specified intervals; library functions must document their worst-case execution times; dynamic memory allocation may be restricted or disallowed entirely (real-time garbage collectors exist, but the application must still ensure that it doesn't give the GC too much work to do); and an enormous amount of testing and measurement to ensure that guarantees are being met.

All of this requires a large amount of additional work, and severely restricts the range of programming languages, libraries and tools that can be used (since most languages and tools do not provide real-time guarantees). For these reasons, developing real-time systems is very expensive, and they are most commonly used in safety-critical embedded devices. Moreover, real-time is not the same as high performance — in fact, real-time systems may have lower throughput, since they have to prioritize timely responses above all else (see also "Latency and resource utilization" on page 278).

For most server-side data processing systems, real-time requirements are simply not economical or appropriate. Consequently, these systems must suffer the pauses and clock instability that come from operating in a non-real-time environment.

Limiting the impact of garbage collection

The negative effects of process pauses can be mitigated without resorting to expensive real-time scheduling guarantees. Language runtimes have some flexibility around when they schedule garbage collections, because they can track the rate of object allocation and the remaining free memory over time.

An emerging idea is to treat GC pauses like brief planned outages of a node, and to let other nodes handle requests from clients while one node is collecting its garbage. If the runtime can warn the application that a node soon requires a GC pause, the application can stop sending new requests to that node, wait for it to finish processing outstanding requests, and then perform the GC while no requests are in progress.

[vii] In embedded systems, *real-time* means that a system is carefully designed and tested to meet specified timing guarantees in all circumstances. This is in contrast to the more vague use of the term *real-time* on the web, where it describes servers pushing data to clients, and stream processing without hard response time constraints (see Chapter 11).

This hides GC pauses from clients, and reduces the high percentiles of response time [66, 67]. This approach has been implemented in some latency-sensitive financial trading systems [68].

A variant of this idea is to use the garbage collector only for short-lived objects (which are fast to collect), and to restart processes periodically, before they accumulate enough long-lived objects to require a full GC of long-lived objects [61, 69]. One node can be restarted at a time, and traffic can be shifted away from the node before the planned restart, like in a rolling upgrade (see Chapter 4).

These measures cannot fully prevent garbage collection pauses, but they can usefully reduce their impact on the application.

Knowledge, Truth and Lies

So far in this chapter we have explored the ways in which distributed systems are different from programs running on a single computer: there is no shared memory, only message-passing via an unreliable network with variable delays. Systems may suffer from partial failure, unreliable clocks and processing pauses.

The consequences of these issues are profoundly disorienting if you're not used to distributed systems. A node in the network cannot *know* anything for sure — it can only make guesses based on the messages it receives (or doesn't receive) via the network. A node can only find out what state another node is in (what data it has stored, whether it is correctly functioning, etc.) by exchanging messages with it. If a remote node doesn't respond, there is no way of knowing what state it is in, because problems in the network cannot reliably be distinguished from problems at a node.

Discussions of these systems border on the philosophical: What do we know to be true or false in our system? How sure can we be of that knowledge, if the mechanisms for perception and measurement are unreliable? Should software systems obey the laws that we expect of the physical world, such as *cause and effect*?

Fortunately, we don't need to go as far as figuring out the meaning of life. In a distributed system, we can state the assumptions we are making about the behavior (the *system model*), and design the actual system in such a way that it meets those assumptions. Algorithms can be proved to function correctly within a certain system model. This means that reliable behavior is achievable, even if the underlying system model provides very few guarantees.

However, although it is possible to make software well-behaved in an unreliable system model, this is not easy or straightforward. In the rest of this chapter we will further explore the notions of knowledge and truth in distributed systems, which will help us think about the kinds of assumptions we can make, and the guarantees we may want to provide. In Chapter 9 we will proceed to look at some examples of algorithms for distributed systems, which provide particular guarantees under particular assumptions.

The truth is defined by the majority

Imagine a network with an asymmetric fault: a node is able to receive all messages sent to it, but any outgoing messages from that node are dropped or delayed [18]. Even though that node is working perfectly well, and is receiving requests from other nodes, the other nodes cannot hear its responses. After some timeout, the other nodes declare it dead, because they haven't heard from the node. The situation unfolds like a nightmare: the semi-disconnected node is dragged to the graveyard, kicking and screaming "I'm not dead!" — but since nobody can hear its screaming, the funeral procession continues with stoic determination.

In a slightly less nightmarish scenario, the semi-disconnected node may notice that the messages it is sending are not being acknowledged by other nodes, and so it realizes that there must be a fault in the network. Nevertheless, the node is wrongly declared dead by the other nodes, and the semi-disconnected node cannot do anything about it.

As a third scenario, imagine a node that experiences a long stop-the-world garbage collection (GC) pause. All of the node's threads are preempted by the GC and paused for one minute, and consequently, no requests are processed and no responses are sent. The other nodes wait, retry, grow impatient, and eventually declare the node dead and load it onto the hearse. Finally, the GC finishes and the node's threads continue as if nothing had happened. The other nodes are surprised as the supposedly dead node suddenly raises its head out of the coffin, in full health, and starts cheerfully chatting with bystanders. At first, the GCing node doesn't even realize that an entire minute has passed and that it was declared dead — from its perspective, hardly any time passed since it was last talking to the other nodes.

The moral of these stories is that a node cannot necessarily trust its own judgment of a situation. A distributed system cannot exclusively rely on a single node, because a node may fail at any time, potentially leaving the system stuck and unable to recover. Instead, many distributed algorithms rely on a *quorum* (see "Quorums for reading and writing" on page 173): decisions are made by a majority of nodes (e.g. 2 out of 3, or 3 out of 5).

That includes decisions about declaring nodes dead. If a majority of nodes declares another node dead, then it must be considered dead, even if that node still very much feels alive. The individual node must abide by the majority decision, and step down.

The reason for using a majority is simple: assuming there is a fixed number of nodes, the system can still work if individual nodes have failed (with 3 nodes, 1 failure can be tolerated; with 5 nodes, 2 failures can be tolerated). However, it is still safe, because there can only be only one majority in the system — there cannot be two majorities with conflicting decisions at the same time. We will discuss this in more detail when we get to *consensus algorithms* in Chapter 9.

The leader and the lock

Frequently, a system requires there to be only one of some thing, for example:

- Only one node is allowed to be the leader for a database partition, to avoid split-brain (see "Handling node outages" on page 150);
- Only one transaction or client is allowed to hold the lock for a particular resource or object, to prevent concurrently writing to it and corrupting it;
- Only one user is allowed to register a particular username, because a username must uniquely identify a user.

Implementing this in a distributed system requires care: even if a nodes believes that it is *'the chosen one'* (the leader of the partition, the holder of the lock, the request handler of the user who successfully grabbed the username), that doesn't necessarily mean the majority of nodes agrees! A node may have formerly been the leader, but if the other nodes declared it dead in the meantime (e.g. due to a network interruption or GC pause), it may have been demoted and another leader may have already been elected.

If a node continues acting as 'the chosen one', even though the majority of nodes have declared it dead, it could cause problems in a system that is not carefully designed. Such a node could send messages to other nodes in its self-appointed capacity, and if other nodes believe it, the system as a whole may do something incorrect.

For example, Figure 8-4 shows a data corruption bug due to an incorrect implementation of locking. (The bug is not theoretical: HBase used to have this problem [70, 71].) Say you want to ensure that a file in a storage service can only be accessed by one client at a time, because if multiple clients tried to write to it, the file would become corrupted. You try to implement this by requiring a client to obtain a lease from a lock service before accessing the file.

The problem is an example of what we discussed in "Process pauses" on page 287: if the client holding the lease is paused for too long, its lease expires. Another client can obtain a lease for the same file, and start writing to the file. When the paused client comes back, it believes (incorrectly) that it still has a valid lease, and proceeds to also write to the file. As a result, the clients' writes clash and corrupt the file.

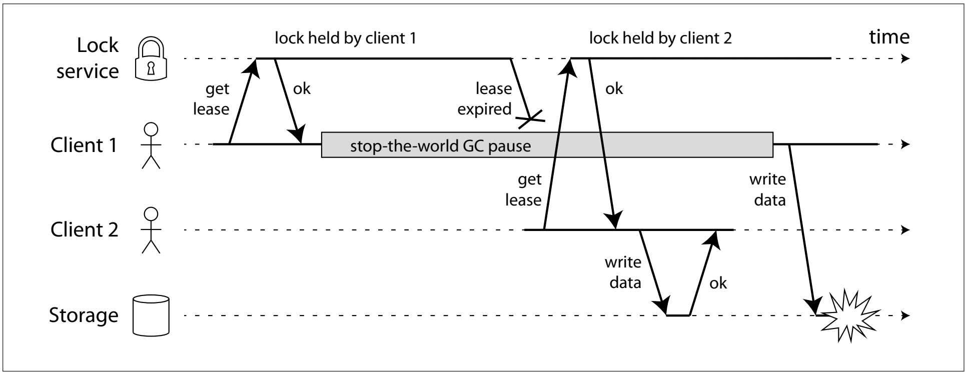


Figure 8-4. Incorrect implementation of a distributed lock: client 1 believes that it still has a valid lease, even though it has expired, and thus corrupts a file in storage.

Fencing tokens

When using a lock or lease to protect access to some resource, such as the file storage in Figure 8-4, we need to ensure that a node that is under a false belief of being 'the chosen one' cannot disrupt the rest of the system. A fairly simple technique that achieves this goal is called *fencing*, and is illustrated in Figure 8-5.

Let's assume that every time the lock server grants a lock or lease, it also returns a *fencing token*, which is a number that increases every time a lock is granted (e.g. incremented by the lock service). We can then require that every time a client sends a write request to the storage service, it must include its current fencing token.

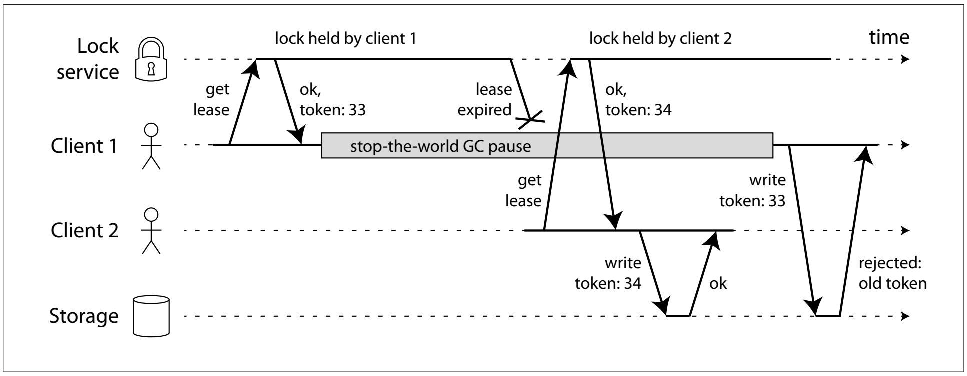


Figure 8-5. Making access to storage safe by allowing writes only in the order of increasing fencing tokens.

In Figure 8-5, client 1 acquires the lease with a token of 33, but then it goes into a long pause and the lease expires. Client 2 acquires the lease with a token of 34 (the number always increases), and then sends its write request to the storage service, including the token of 34. Later, client 1 comes back to life and sends its write to the storage service, including its token value 33. However, the storage server rembers that it has already processed a write with a higher token number (34), and so it rejects the request with token 33.

If ZooKeeper is used as lock service, the transaction ID `zxid` or the node version `cversion` can be used as fencing token. Since they are guaranteed to be monotonically increasing, they have the required properties [70].

Note this requires the resource itself to take an active role in checking tokens, and rejecting any writes on which the token has gone backwards — it is not sufficient to rely on clients checking their lock status themselves. For resources that do not explicitly support fencing tokens, you might still be able work around the limitation (for example, in the case of a file storage service you could include the fencing token in the filename). However, some kind of check is necessary to avoid processing requests outside of the lock's protection.

Checking a token on the server side may seem like a downside, but it is arguably a good thing: it is unwise for a service to assume that its clients will always be well-behaved, because the clients are often run by people whose priorities are very different from the priorities of the people running the service [72]. Thus, it is a good idea for any service to protect itself from accidentally abusive clients.

Byzantine faults

The techniques in the last section, such as fencing tokens, are helpful for detecting and blocking a node that is *inadvertently* acting in error (e.g. because it hasn't yet found out that its lease has expired). However, if the node deliberately wanted to subvert the system's guarantees, it could easily do so by sending messages with a fake fencing token.

In this book we assume that nodes are unreliable but honest: they may be slow or never respond (due to a fault), and their state may be outdated (due to a GC pause or network delays), but we assume that if a node *does* respond, it is telling the 'truth': to the best of its knowledge, it is playing by the rules of the protocol.

Distributed systems problems become much harder if there is a risk that nodes may 'lie' (send arbitrary faulty or corrupted responses) — for example, if a node may claim to have received a particular message from another node, when in fact it didn't. If a node sends untrue messages to other nodes, that is known as a *Byzantine fault*, and the problem of reaching consensus in this untrusting environment is known as the *Byzantine Generals Problem* [73].

The Byzantine Generals Problem

The Byzantine Generals Problem is a generalization of the so-called *Two Generals Problem* [74] which imagines a situation in which two army generals need to agree on a common battle plan. As they have set up camp on two different sites, they can only communicate by messenger, and the messengers sometimes get delayed or captured (like packets in a network). We will discuss this problem of *consensus* in more detail in Chapter 9.

In the Byzantine version of the problem, there are *n* generals who need to agree, and their endeavor is hampered by the fact that there are some traitors in their midst. Most of the generals are loyal, and thus send truthful messages, but the traitors may try to deceive and confuse the others by sending fake or untrue messages (while trying to remain undiscovered). It is not known in advance who the traitors are.

Byzantium was an ancient Greek city that later became Constantinople, in the place which is now Istanbul in Turkey. There isn't any historic evidence that the generals of Byzantium were any more prone to intrigue and conspiracy than those elsewhere. Rather, the name is derived from *byzantine* in the sense of *excessively complicated, bureaucratic, devious*, which was used in politics long before computers [75]. Lamport wanted to choose a nationality that would not offend any readers, and he was advised that calling it *The Albanian Generals Problem* was not such a good idea [76].

A system is *Byzantine fault tolerant* if it continues to operate correctly, even if some of the nodes are malfunctioning and not obeying the protocol, or if malicious attackers are interfering with the network. This is relevant in certain specific circumstances, for example:

- In aerospace environments, the data in a computer's memory or CPU register could become corrupted by radiation, leading it to respond to other nodes in arbitrarily unpredictable ways. Since a system failure would be very expensive (e.g. an aircraft crashing and killing everyone on board, or a rocket colliding with the International Space Station), flight control systems must tolerate Byzantine faults [77, 78].
- In a system with multiple participating organizations, some participants may attempt to cheat or defraud others. In such circumstances, it is not safe for a node to simply trust another node's messages, since they may be sent with malicious intent. For example, systems like the Bitcoin blockchain can be considered

to be a way of getting mutually untrusting parties to agree whether a transaction happened or not, without relying on a central authority [79].

However, in the kinds of systems we discuss in this book, we can usually safely assume that there are no Byzantine faults. In your datacenter, all the nodes are controlled by your organization (so they can hopefully be trusted) and radiation levels are low enough that memory corruption is not a major problem. Protocols for making systems Byzantine fault tolerant are quite complicated [80] and fault-tolerant embedded systems rely on support from the hardware level [77]. In most server-side data systems, the cost of deploying Byzantine fault tolerant solutions makes them impractical.

A bug in the software could be regarded as a Byzantine fault, but if you deploy the same software to all nodes, then a Byzantine fault tolerant algorithm cannot save you. Most Byzantine fault tolerant algorithms require a supermajority of more than two thirds of the nodes to be functioning correctly, i.e. if you have four nodes, at most one may malfunction. To use this approach against bugs, you would have to have four independent implementations of the same software, and hope that a bug only appears in one of the four implementations.

Similarly, it would be appealing if a protocol could protect us from vulnerabilities, security compromises and malicious attacks. Unfortunately, this is not realistic either: in most systems, if an attacker can compromise one node, they can probably compromise all of them, because they are probably running the same software. Thus, traditional mechanisms (authentication, access control, encryption, firewalls and so on) continue to be the main protection against attackers.

Weak forms of lying

Although we assume that nodes are generally honest, it can be worth adding mechanisms to software that guard against weak forms of 'lying' — for example, invalid messages due to hardware issues, software bugs, and misconfiguration. Such protection mechanisms are not full-blown Byzantine fault tolerance, as they would not withstand a determined adversary, but they are nevertheless simple and pragmatic steps towards better reliability. For example:

- Network packets do sometimes get corrupted due to hardware issues or bugs in operating systems, drivers, routers, etc. Usually, corrupted packets are caught by the checksums built into TCP and UDP, but sometimes they evade detection [81, 82, 83]. Simple measures are usually sufficient protection against such corruption, such as checksums in the application-level protocol.
- A publicly accessible application must carefully sanitize any inputs from users, for example checking that a value is within a reasonable range, and limiting the size of strings to prevent denial of service through large memory allocations. An

internal service behind a firewall may be able to get away with less strict checks on inputs, but some basic sanity-checking of values (e.g. in protocol parsing [81]) is a good idea.

- NTP clients can be configured with multiple server addresses. When synchronizing, the client contacts all of them, estimates their errors, and checks that a majority of servers agree on some time range. As long as most of the servers are ok, a misconfigured NTP server that is reporting an incorrect time is detected as an outlier, and is excluded from synchronization [36]. This makes NTP more robust than if it only used a single server.

System model and reality

Many algorithms have been designed to solve distributed systems problems — for example, we will examine solutions for the *consensus* problem in Chapter 9. In order to be useful, these algorithms need to tolerate the various faults of distributed systems that we discussed in this chapter.

Algorithms need to be written in a way that does not depend too heavily on the details of the hardware and software configuration on which they are run. This in turn requires that we somehow formalize the kinds of fault that we expect to happen in a system. We do this by defining a *system model*, which is an abstraction that describes what things an algorithm may assume.

With regard to timing assumptions, three system models are in common use:

Synchronous model

The synchronous model assumes bounded network delay, bounded process pauses, and bounded clock error. This does not imply exactly synchronised clocks or zero network delay; it just means you know that network delay, pauses and clock drift will never exceed some fixed upper bound [84]. As discussed in this chapter, the synchronous model is not a realistic model of most practical systems, because unbounded delays and pauses do occur.

Partially synchronous model

Partial synchrony means that a system behaves like a synchronous system *most of the time*, but it sometimes exceeds the bounds for network delay, process pauses and clock drift [84]. This is a realistic model of many systems: most of the time, network and processes are quite well-behaved, otherwise we would never be able to get anything done — but we have to reckon with the fact that any timing assumptions may be shattered occasionally. When this happens, network delay, pauses and clock error may become arbitrarily large.

Asynchronous model

In the asynchronous model, an algorithm is not allowed to make any timing assumptions whatsoever — in fact, it does not even have a clock (so it cannot use timeouts). Some algorithms can be designed in the asynchronous model, but it is very restrictive.

Moreover, besides timing issues, we have to consider node failures. The three most common system models for nodes are:

Crash-stop faults

In the crash-stop model, an algorithm may assume that a node can fail in only one way, namely by crashing. This means that the node may suddenly stop responding at any moment, and thereafter that node is gone forever — it never comes back.

Crash-recovery faults

We assume that nodes may crash at any moment, and perhaps start responding again after some unknown time. In the crash-recovery model, nodes are assumed to have stable storage (i.e. non-volatile disk storage) which is preserved across crashes, while the in-memory state is assumed to be lost.

Byzantine (arbitrary) faults

Nodes may do absolutely anything, including trying to trick and deceive other nodes, as described in the last section.

For modeling real systems, the partially synchronous model with crash-recovery faults is generally the most useful model. But how do distributed algorithms cope with that model?

Correctness of an algorithm

To define what it means for an algorithm to be correct, we can describe its *properties*. For example, the output of a sorting algorithm has the property that for any two distinct elements of the output list, the element further to the left is smaller than the element further to the right. That is simply a formal way of defining what it means for a list to be sorted.

Similarly, we can write down the properties we want of a distributed algorithm to define what it means to be correct. For example, if we are generating fencing tokens for a lock (see "Fencing tokens" on page 294), we may require the algorithm to have the following properties:

Uniqueness

No two requests for a fencing token return the same value.

Monotonic sequence

If request x returned token t_x, and request y returned token t_y, and x completed before y began, then $t_x < t_y$.

Availability

A node that requests a fencing token and does not crash eventually receives a response.

An algorithm is correct in some system model if it always satisfies its properties in all situations that we assume may occur in that system model. But how does this make sense? If all nodes crash, or all network delays suddenly become infinitely long, then no algorithm will be able to get anything done.

Safety and liveness

To clarify the situation, it is worth distinguishing between two different kinds of properties: *safety* and *liveness* properties. In the example above, *uniqueness* and *monotonic sequence* are safety properties, but *availability* is a liveness property.

What distinguishes the two kinds of property? A giveaway is that liveness properties often include the word "eventually" in their definition. (And yes, you guessed it — *eventual consistency* is a liveness property [85].)

Safety is often informally defined as *nothing bad happens*, and liveness as *something good eventually happens*. However, it's best to not read too much into those informal definitions, because the meaning of good and bad is subjective. The actual definitions of safety and liveness are precise and mathematical [86]:

- If a safety property is violated, we can point at a particular point in time at which it was broken (for example, if the uniqueness property was violated, we can identify the particular operation in which a duplicate fencing token was returned). After a safety property has been violated, the violation cannot be undone — the damage is already done.
- A liveness property works the other way round: it may not hold at some point in time (for example, a node may have sent a request but not yet received a response), but there is always hope that it may be satisfied in future (namely by receiving a response).

An advantage of distinguishing between safety and liveness properties is that they help us deal with difficult system models. For distributed algorithms, it is common to require that safety properties *always* hold, in all possible situations of a system model [84]. That is, even if all nodes crash, or the entire network fails, the algorithm must nevertheless ensure that it does not return a wrong result, i.e. that the safety properties remain satisfied.

However, with liveness properties we are allowed to make caveats: for example, we could say that a request needs to receive a response only if a majority of nodes is not crashed, and only if the network eventually recovers from an outage. The definition of the partially synchronous model requires that eventually the system returns to a synchronous state — that is, any period of network interruption lasts only for a finite duration and is then repaired.

Mapping system models to the real world

Safety and liveness properties and system models are very useful for reasoning about the correctness of a distributed algorithm. However, when implementing an algorithm in practice, the messy facts of reality come back to bite you again, and it becomes clear that the system model is a simplified abstraction of reality.

For example, algorithms in the crash-recovery model generally assume that data in stable storage survives crashes. However, what happens if the data on disk is corrupted, or the data is wiped out due to hardware error or misconfiguration [87]? What happens if a server has a firmware bug, and fails to recognize its hard drives on reboot, even though the drives are correctly attached to the server [88]?

Quorum algorithms (see "Quorums for reading and writing" on page 173) rely on a node remembering the data that it claims to have stored. If a node may suffer from amnesia and forget previously stored data, that breaks the quorum condition, and thus breaks the correctness of the algorithm. Perhaps a new system model is needed, in which we assume that stable storage mostly survives crashes, but may sometimes be lost. But that model then becomes harder to reason about.

The theoretical description of an algorithm can declare that certain things are simply assumed not to happen — and in non-Byzantine systems, we do have to make some assumptions about faults that can and cannot happen. However, a real implementation may still have to include code to handle the case where something happens that was assumed to be impossible, even if that handling boils down to `printf("Sucks to be you")` and `exit(666)` — i.e. letting a human operator clean up the mess [89]. (This is arguably the difference between computer science and software engineering.)

That is not to say that theoretical, abstract system models are worthless — quite the opposite. They are incredibly helpful for distilling down the complexity of real systems to a manageable set of faults that we can reason about, so that we can understand the problem and try to solve it systematically. We can prove algorithms correct by showing that their properties always hold in some system model.

Proving an algorithm correct does not mean its *implementation* on a real system will necessarily always behave correctly. But it's a very good first step, because the theoretical analysis can uncover problems in an algorithm that might remain hidden for a long time in a real system, and that only come to bite you when your assumptions (e.g. about timing) are defeated due to unusual circumstances. Theoretical analysis and empirical testing are equally important.

Summary

In this chapter we have discussed a wide range of problems that can occur in distributed systems, including:

- Whenever you try to send a packet over the network, it may be lost or arbitrarily delayed. Likewise, the reply may be lost or delayed, so if you don't get a reply, you have no idea whether the message got through.
- A node's clock may be significantly out of sync with other nodes (despite your best efforts to set up NTP), it may suddenly jump forward or back in time, and relying on it is dangerous because you most likely don't have a good measure of your clock's error interval.
- A process may pause for a substantial amount of time at any point in its execution (perhaps due to a stop-the-world garbage collector), be declared dead by other nodes, and then come back to life again without realizing that it was paused.

All of these can be summarized as *partial failures*. Whenever software tries to do anything involving other nodes, there is the possibility that it may occasionally fail, or randomly go slow, or not respond at all (and eventually time out). In distributed systems, we try to build tolerance of partial failures into software, so that the system as a whole may continue functioning, even when some of its constituent parts are broken.

To tolerate faults, the first step is to *detect* the fault, but even that is hard. Most systems don't have an accurate mechanism of detecting whether a node has failed, so most distributed algorithms rely on timeouts to determine whether a remote node is still available. However, timeouts can't distinguish between network and node failures, and variable network delay sometimes causes a node to be falsely suspected of crashing. Moreover, sometimes a node can be in a degraded state: for example, a Gigabit network interface could suddenly drop to 1 kilobit/s throughput due to a driver bug [90]. Such a system that is 'limping', but not dead, can be even more difficult to deal with than a cleanly failed node.

Once a fault is detected, making a system tolerate it is not easy either: there is no global variable, no shared memory, no common knowledge or any other kind of shared state between the machines. Nodes can't even agree what time it is, let alone anything more profound. The only way how information can flow from one node to another is by sending it over the unreliable network. Major decisions cannot be safely made by a single node, so we require protocols that enlist the help from other nodes and try to get a majority quorum to agree.

If you're used to writing software in the idealized mathematical perfection of a single computer, where the same operation always deterministically returns the same result, then moving to the messy physical reality of distributed systems can be a bit of a shock. Conversely, distributed systems engineers will often regard a problem as trivial if it can be solved on a single computer [5], and indeed a single computer can do a lot nowadays [91]. If you can avoid opening Pandora's box, and simply keep things on a single machine, it is generally worth doing so.

However, as discussed in the introduction to Part II, scalability is not the only reason for wanting to use a distributed system. Fault tolerance and low latency (by placing data geographically close to users) are equally important goals, and those things cannot be achieved with a single node.

In this chapter we also went on some tangents to explore whether the unreliability of networks, clocks and processes is an inevitable law of nature. We saw that it isn't: it is possible to give hard real-time response guarantees and bounded delay in networks, but doing so is very expensive and results in lower utilization of hardware resources. Most non-safety-critical systems choose cheap and unreliable over expensive and reliable.

We also touched on supercomputers, which assume reliable components, and thus have to be stopped and restarted entirely when a component does fail. By contrast, distributed systems can run forever without being interrupted at the service level, because all faults and maintenance can be handled at the node level — at least in theory. (In practice, if a bad configuration change is rolled out to all nodes, that will still bring a distributed system to its knees.)

This chapter has been all about problems, and has given us a bleak outlook. In the next chapter we will move on to solutions, and discuss some algorithms that have been designed to cope with all the problems in distributed systems.

References

[1] Mark Cavage: "There's Just No Getting around It: You're Building a Distributed System," *ACM Queue*, volume 11, number 4, April 2013. doi: 10.1145/2466486.2482856

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

[3] Sydney Padua: *The Thrilling Adventures of Lovelace and Babbage: The (Mostly) True Story of the First Computer*. Particular Books, April 2015. ISBN: 978-0-141-98151-2

[4] Coda Hale: "You Can't Sacrifice Partition Tolerance," codahale.com, 7 October 2010.

[5] Jeff Hodges: "Notes on Distributed Systems for Young Bloods," somethingsimilar.com, 14 January 2013.

[6] Antonio Regalado: "Who Coined 'Cloud Computing'?," technologyreview.com, 31 October 2011.

[7] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle: "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition," *Synthesis Lectures on Computer Architecture*, volume 8, number 3, Morgan & Claypool Publishers, July 2013. doi:10.2200/S00516ED2V01Y201306CAC024, ISBN: 978-1627050104

[8] David Fiala, Frank Mueller, Christian Engelmann, et al.: "Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing," at *International Conference for High Performance Computing, Networking, Storage and Analysis* (SC12), November 2012.

[9] Glenn K Lockwood: "Hadoop's Uncomfortable Fit in HPC," glennklockwood.blogspot.co.uk, 16 May 2014.

[10] Arjun Singh, Joon Ong, Amit Agarwal, et al.: "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," at *Annual Conference of the ACM Special Interest Group on Data Communication* (SIGCOMM), pages 183–197, August 2015. doi:10.1145/2785956.2787508

[11] John von Neumann: "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," in *Automata Studies (AM-34)*, Claude E Shannon and John McCarthy, Editors. Princeton University Press, 1956. ISBN: 9780691079165

[12] Richard W Hamming: *The Art of Doing Science and Engineering*. Taylor & Francis, 1997. ISBN: 9056995006

[13] Claude E Shannon: "A Mathematical Theory of Communication," *The Bell System Technical Journal*, volume 27, number 3, pages 379–423 and 623–656, July 1948.

[14] Peter Bailis and Kyle Kingsbury: "The Network is Reliable," *ACM Queue*, volume 12, number 7, July 2014. doi:10.1145/2639988.2639988

[15] Joshua B Leners, Trinabh Gupta, Marcos K Aguilera, and Michael Walfish: "Taming uncertainty in distributed systems with help from the network," at *10th European Conference on Computer Systems* (EuroSys), April 2015. doi:10.1145/2741948.2741976

[16] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan: "Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications," at *ACM SIGCOMM Conference*, pages 350–361, August 2011. doi:10.1145/2018436.2018477

[17] Mark Imbriaco: "Downtime last Saturday," github.com, 26 December 2012.

[18] Marc A Donges: "Re: bnx2 cards intermittantly going offline," Message to Linux netdev mailing list, spinics.net, 13 September 2012.

[19] Kyle Kingsbury: "Call me maybe: Elasticsearch," aphyr.com, 15 June 2014.

[20] Salvatore Sanfilippo: "A few arguments about Redis Sentinel properties and fail scenarios," antirez.com, 21 October 2014.

[21] Bert Hubert: "The ultimate SO_LINGER page, or: why is my tcp not reliable," blog.netherlabs.nl, 18 January 2009.

[22] Nicolas Liochon: "CAP: if all you have is a timeout, everything looks like a partition," blog.thislongrun.com, 25 May 2015.

[23] Jerome H Saltzer, David P Reed, and David D Clark: "End-To-End Arguments in System Design," *ACM Transactions on Computer Systems*, volume 2, number 4, pages 277–288, November 1984. doi:10.1145/357401.357402

[24] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, et al.: "Queues Don't Matter When You Can JUMP Them!," at *12th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), May 2015.

[25] Guohui Wang and T S Eugene Ng: "The Impact of Virtualization on Network Performance of Amazon EC2 Data Center," at *29th IEEE International Conference on Computer Communications* (INFOCOM), March 2010. doi:10.1109/INFCOM.2010.5461931

[26] Van Jacobson: "Congestion Avoidance and Control," at *ACM Symposium on Communications Architectures and Protocols* (SIGCOMM), pages 314–329, August 1988. doi:10.1145/52324.52356

[27] Brandon Philips: "etcd: distributed locking and service discovery," at *Strange Loop*, September 2014.

[28] Steve Newman: "A Systematic Look at EC2 I/O," blog.scalyr.com, 16 October 2012.

[29] Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama: "The ϕ accrual failure detector," Japan Advanced Institute of Science and Technology, School of Information Science, Technical Report IS-RR-2004-010, May 2004.

[30] Jeffrey Wang: "Phi Accrual Failure Detector," ternarysearch.blogspot.co.uk, 11 August 2013.

[31] Srinivasan Keshav: *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*. Addison-Wesley Professional, May 1997. ISBN: 0-201-63442-2

[32] "Integrated Services Digital Network," docwiki.cisco.com.

[33] Othmar Kyas: *ATM networks*. International Thomson Publishing, 1995. ISBN: 1-850-32128-0

[34] "InfiniBand FAQ." Mellanox Technologies, 22 December 2014.

[35] Jose Renato Santos, Yoshio Turner, and G (John) Janakiraman: "End-to-End Congestion Control for InfiniBand," at *22nd Annual Joint Conference of the IEEE Computer and Communications Societies* (INFOCOM), April 2003. Also published by HP Laboratories Palo Alto, Tech Report HPL-2002-359. doi:10.1109/INFCOM.2003.1208949

[36] Ulrich Windl, David Dalton, Marc Martinec, and Dale R Worley: "The NTP FAQ and HOWTO," ntp.org, November 2006.

[37] David Holmes: "Inside the Hotspot VM: Clocks, Timers and Scheduling Events - Part I - Windows," blogs.oracle.com, 2 October 2006.

[38] Steve Loughran: "Time on multi-core, multi-socket servers," steveloughran.blogspot.co.uk, 17 September 2015.

[39] 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.

[40] M Caporaloni and R Ambrosini: "How closely can a personal computer clock track the UTC timescale via the internet?," *European Journal of Physics*, volume 23, number 4, pages L17–L21, June 2012. doi:10.1088/0143-0807/23/4/103

[41] Nelson Minar: "A Survey of the NTP Network," xenia.media.mit.edu, December 1999.

[42] Viliam Holub: "Synchronizing Clocks In a Cassandra Cluster Pt. 1 – The Problem," blog.logentries.com, 14 March 2014.

[43] Poul-Henning Kamp: "The one-second war (What time will you die?)," *ACM Queue*, volume 9, number 4, April 2011. doi:10.1145/1966989.1967009

[44] Nelson Minar: "Leap Second crashes half the internet," somebits.com, 3 July 2012.

[45] Christopher Pascoe: "Time, technology and leaping seconds," googleblog.blogspot.co.uk, 15 September 2011.

[46] Mingxue Zhao and Jeff Barr: "Look Before You Leap – The Coming Leap Second and AWS," aws.amazon.com, 18 May 2015.

[47] "Timekeeping in VMware Virtual Machines," Information Guide, VMware, Inc., December 2011.

[48] "MiFID II / MiFIR: Regulatory technical and implementing standards — Annex I (draft)," European Securities and Markets Authority, Report ESMA/2015/1464, September 2015.

[49] Luke Bigum: "Solving MiFID II Clock Synchronisation with minimum spend (part 1)," lmax.com, 27 November 2015.

[50] Kyle Kingsbury: "Call me maybe: Cassandra," aphyr.com, 24 September 2013.

[51] John Daily: "Clocks Are Bad, Or, Welcome to the Wonderful World of Distributed Systems," basho.com, 12 November 2013.

[52] Kyle Kingsbury: "The trouble with timestamps," aphyr.com, 12 October 2013.

[53] Sandeep Kulkarni, Murat Demirbas, Deepak Madeppa, Bharadwaj Avva, and Marcelo Leone: "Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases," University at Buffalo, Computer Science and Engineering Technical Report 2014-04, May 2014.

[54] Justin Sheehy: "There is No Now: Problems with simultaneity in distributed systems," *ACM Queue*, volume 13, number 3, March 2015. doi:10.1145/2733108

[55] Murat Demirbas: "Spanner: Google's Globally-Distributed Database," muratbuffalo.blogspot.co.uk, 4 July 2013.

[56] Dahlia Malkhi and Jean-Philippe Martin: "Spanner's Concurrency Control," *ACM SIGACT News*, volume 44, number 3, pages 73–77, September 2013. doi:10.1145/2527748.2527767

[57] Manuel Bravo, Nuno Diegues, Jingna Zeng, Paolo Romano, and Luís Rodrigues: "On the use of Clocks to Enforce Consistency in the Cloud," *IEEE Data Engineering Bulletin*, volume 38, number 1, pages 18–31, March 2015.

[58] Spencer Kimball: "Living Without Atomic Clocks," cockroachlabs.com, 17 February 2016.

[59] Cary G Gray and David R Cheriton: "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency," at *12th ACM Symposium on Operating Systems Principles* (SOSP), December 1989. doi:10.1145/74850.74870

[60] Todd Lipcon: "Avoiding Full GCs in Apache HBase with MemStore-Local Allocation Buffers: Part 1," blog.cloudera.com, 24 February 2011.

[61] Martin Thompson: "Java Garbage Collection Distilled," mechanical-sympathy.blogspot.co.uk, 16 July 2013.

[62] Alexey Ragozin: "How to tame java GC pauses? Surviving 16GiB heap and greater," java.dzone.com, 28 June 2011.

[63] Christopher Clark, Keir Fraser, Steven Hand, et al.: "Live Migration of Virtual Machines," at *2nd USENIX Symposium on Symposium on Networked Systems Design & Implementation* (NSDI), May 2005.

[64] Mike Shaver: "fsyncers and curveballs," shaver.off.net, 25 May 2008.

[65] Zhenyun Zhuang and Cuong Tran: "Eliminating Large JVM GC Pauses Caused by Background IO Traffic," engineering.linkedin.com, 10 February 2016.

[66] David Terei and Amit Levy: "Blade: A data center garbage collector," arXiv: 1504.02578, 13 April 2015.

[67] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz: "Trash Day: Coordinating Garbage Collection in Distributed Systems," at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.

[68] "Predictable low latency," Cinnober Financial Technology AB, cinnober.com, 24 November 2013.

[69] Martin Fowler: "The LMAX Architecture," martinfowler.com, 12 July 2011.

[70] Flavio P Junqueira and Benjamin Reed: *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, November 2013. ISBN: 978-1-4493-6130-3

[71] Enis Söztutar: "HBase and HDFS: Understanding filesystem usage in HBase," at *HBaseCon*, June 2013.

[72] Caitie McCaffrey: "Clients are jerks: aka how Halo 4 DoSed the services at launch & how we survived," caitiem.com, 23 June 2015.

[73] Leslie Lamport, Robert Shostak, and Marshall Pease: "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems* (TOPLAS), volume 4, number 3, pages 382–401, July 1982. doi:10.1145/357172.357176

[74] Jim N Gray: "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science, volume 60, edited by R Bayer, R M Graham, and G Seegmüller, pages 393–481. Springer-Verlag, 1978. ISBN: 3-540-08755-9

[75] Brian Palmer: "How Complicated Was the Byzantine Empire?," slate.com, 20 October 2011.

[76] Leslie Lamport: "My Writings," research.microsoft.com, 16 December 2014. This page can be found by searching the web for the 23-character string obtained by removing the hyphens from the string `allla-mport-spubso-ntheweb`.

[77] John Rushby: "Bus Architectures for Safety-Critical Embedded Systems," at *1st International Workshop on Embedded Software* (EMSOFT), October 2001.

[78] Jake Edge: "ELC: SpaceX lessons learned," lwn.net, 6 March 2013.

[79] Andrew Miller and Joseph J LaViola, Jr.: "Anonymous Byzantine Consensus from Moderately-Hard Puzzles: A Model for Bitcoin," University of Central Florida, Technical Report CS-TR-14-01, April 2014.

[80] James Mickens: "The Saddest Moment," *USENIX ;login: logout*, May 2013.

[81] Evan Gilman: "The Discovery of Apache ZooKeeper's Poison Packet," pagerduty.com, 7 May 2015.

[82] Jonathan Stone and Craig Partridge: "When The CRC and TCP Checksum Disagree," at *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (SIGCOMM), August 2000. doi:10.1145/347059.347561

[83] Evan Jones: "How both TCP and Ethernet checksums fail," evanjones.ca, 5 October 2015.

[84] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer: "Consensus in the Presence of Partial Synchrony," *Journal of the ACM*, volume 35, number 2, pages 288–323, April 1988. doi:10.1145/42282.42283

[85] Peter Bailis and Ali Ghodsi: "Eventual Consistency Today: Limitations, Extensions, and Beyond," *ACM Queue*, volume 11, number 3, March 2013. doi:10.1145/2460276.2462076

[86] Bowen Alpern and Fred B Schneider: "Defining Liveness," *Information Processing Letters*, volume 21, number 4, pages 181–185, October 1985. doi:10.1016/0020-0190(85)90056-0

[87] Flavio P Junqueira: "Dude, where's my metadata?," fpj.me, 28 May 2015.

[88] Scott Sanders: "January 28th Incident Report," github.com, 3 February 2016.

[89] Jay Kreps: "A Few Notes on Kafka and Jepsen," blog.empathybox.com, 25 September 2013.

[90] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S Gunawi: "Limplock: Understanding the impact of limpware on scale-out cloud systems," at *4th ACM Symposium on Cloud Computing* (SoCC), October 2013. doi:10.1145/2523616.2523627

[91] 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.

CHAPTER 9
Consistency and Consensus

> Is it better to be alive and wrong or right and dead?
>
> —Jay Kreps, *A few notes on Kafka and Jepsen (2013)*

Lots of things can go wrong in distributed systems, as discussed in Chapter 8. The simplest way of handling such faults is to simply let the entire service fail, and show the user an error message. If that solution is unacceptable, we need to find ways of tolerating faults — that is, to keep the service functioning correctly, even if some internal component is faulty.

In this chapter, we will talk about some examples of algorithms and protocols for building fault-tolerant distributed systems. We will assume that all the problems from Chapter 8 can occur: packets can be lost, reordered, duplicated or arbitrarily delayed in the network, clocks are approximate at best, and nodes can pause (e.g. due to garbage collection) or crash at any time.

The best way of building fault tolerant systems is to find some general-purpose abstractions with useful guarantees, implement them once, and then let applications rely on those guarantees. This is the same approach as we used with transactions in Chapter 7: by using a transaction, the application can pretend that there are no crashes (atomicity), that nobody else is concurrently accessing the database (isolation), and that storage devices are perfectly reliable (durability). Even though crashes, race conditions and disk failures do occur, the transaction abstraction hides those problems so that the application doesn't need to worry about them.

We will now continue along the same lines, and seek abstractions that can allow an application to ignore some of the problems with distributed systems. For example, one of the most important abstractions for distributed systems is *consensus*, that is, getting all of the nodes to agree on something. As we shall see in this chapter, reliably reaching consensus in spite of network faults and process failures is a surprisingly tricky problem.

Once you have an implementation of consensus, applications can use it for various purposes. For example, say you have a database with single-leader replication. If the leader dies and you need to failover to another node, the remaining database nodes can use consensus to elect a new leader. As discussed in "Handling node outages" on page 150, it's important that there is only one leader, and that all nodes agree who the leader is. If two nodes both believe that they are the leader, that situation is called *split brain*, and it often leads to data loss. Correct implementations of consensus help avoid such problems.

Later in this chapter, in "Distributed Transactions and Consensus" on page 343, we will look into algorithms to solve consensus and related problems. But first we first need to explore the range of guarantees and abstractions that can be provided in a distributed system.

We need to understand the scope of what can and cannot be done: in some situations, it's possible for the system to tolerate faults and continue working; in other situations, that is not possible. The limits of what is and isn't possible have been explored in depth, both in theoretical proofs and in practical implementations. We will get an overview of those fundamental limits in this chapter.

Researchers in the field of distributed systems have been studying these topics for decades, so there is a lot of material — we'll only be able to scratch the surface. In this book we don't have space to go into details of the formal models and proofs, so we will stick with informal intuitions. The literature references offer plenty of additional depth if you're interested.

Consistency Guarantees

In "Problems With Replication Lag" on page 155 we looked at some timing issues that occur in a replicated database. If you look at two database nodes at the same moment in time, you're likely to see different data on the two nodes, because write requests arrive on different nodes at different times. These inconsistencies occur no matter what replication method the database uses (single-leader, multi-leader or leaderless replication).

Most replicated databases provide at least *eventual consistency*, which means that if you stop writing to the database and wait for some unspecified length of time, then eventually all read requests will return the same value [1]. In other words, the inconsistency is temporary, and it eventually resolves itself (assuming that any faults in the network are also eventually repaired). A better name for eventual consistency may be *convergence*, as we expect all replicas to eventually converge to the same value [2].

However, this is a very weak guarantee — it doesn't say anything about *when* the replicas will converge. Until the time of convergence, reads could return anything or nothing [1]. For example, if you write a value and then immediately read it again, there is no guarantee that you will see the value you just wrote, because the read may be routed to a different replica (see "Reading your own writes" on page 156).

Eventual consistency is hard for application developers because it is so different from the behavior of variables in a normal single-threaded program. If you assign a value to a variable, and then read it shortly afterwards, you don't expect to read back the old value, or for the read to fail. A database looks superficially like a variable that you can read and write, but in fact its semantics is much more complicated [3].

When working with a database that provides only weak guarantees, you need to be constantly aware of its limitations, and not accidentally assume too much. Bugs are often subtle and hard to find by testing, because the application may work well most of the time. The edge cases of eventual consistency only become apparent when there is a fault in the system (e.g. a network interruption) or at high concurrency.

In this chapter we will explore stronger consistency models that data systems may choose to provide. They don't come for free: systems with stronger guarantees may have worse performance, or be less fault-tolerant than systems with weaker guarantees. Nevertheless, stronger guarantees can be appealing because they are easier use correctly. Once you have seen a few different consistency models, you'll be in a better position to decide which one best fits your needs.

There is some similarity between distributed consistency models and the hierarchy of transaction isolation levels we discussed in Chapter 7 (see "Weak isolation levels" on page 224) [4, 5]. Although there is some overlap, they are mostly independent concerns: transaction isolation is primarily about avoiding race conditions due to concurrently executing transactions, whereas distributed consistency is mostly about coordinating the state of replicas in the face of delays and faults.

This chapter covers a broad range of topics, but as we shall see, these areas are in fact deeply linked to each other:

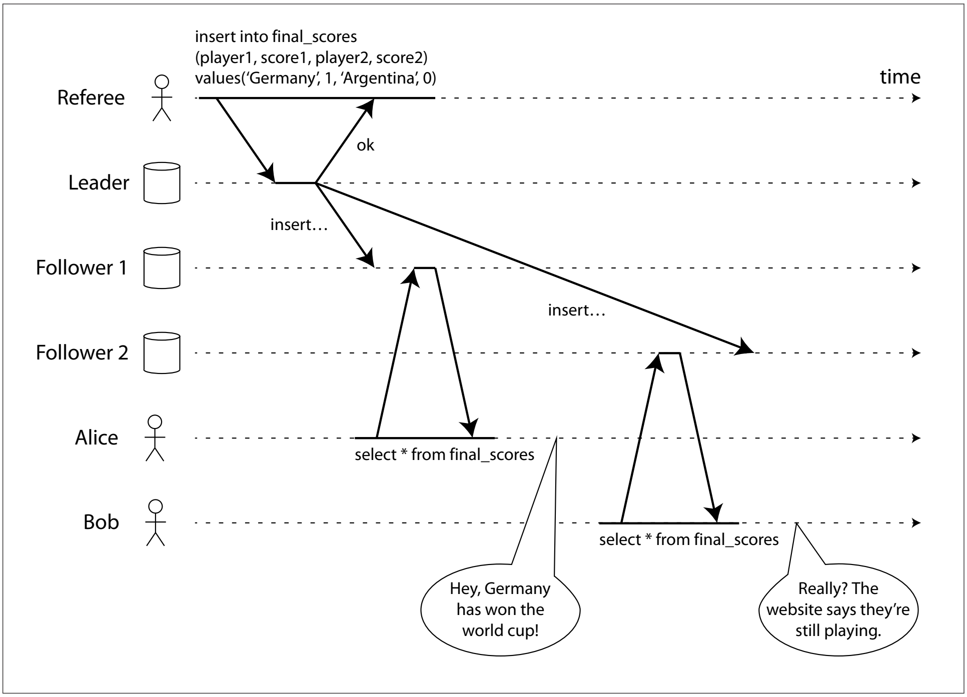
- We will start by looking at one of the strongest consistency models in common use, *linearizability*, and examine its pros and cons.
- We'll then examine the issue of ordering events in a distributed system ("Ordering Guarantees" on page 329), particularly causality and total ordering.
- In the third section ("Distributed Transactions and Consensus" on page 343) we will explore how to atomically commit a distributed transaction, which will finally lead us towards solutions for the consensus problem.

Linearizability

In an eventually consistent database, if you ask two different replicas the same question at the same time, you may get two different answers. That's confusing. Wouldn't it be a lot simpler if the database could give the illusion that there is only one replica, i.e. only one copy of the data? Then every client would have the same view of the data, and you wouldn't have to worry about replication lag.

This is the idea behind *linearizability* [6] (also known as *atomic consistency* [7], *strong consistency*, *immediate consistency*, or *external consistency* [8]). The exact definition of linearizability is quite subtle, and we will explore it in the rest of this section. But the basic idea is to make a system appear as if there was only one copy of the data, and all operations on it are atomic. With this guarantee, even though there may be multiple replicas in reality, the application does not need to worry about them.

In a linearizable system, as soon as one client successfully completes a write, all clients reading from the database must be able to see the value just written. Maintaining the illusion of a single copy of the data means guaranteeing that the value you read is the most recent, up-to-date value, not from a stale cache or replica. In other words, linearizability is a *recency guarantee*. To clarify this idea, let's look at an example of a system that is not linearizable.



insert into final_scores
(player1, score1, player2, score2)
values('Germany', 1, 'Argentina', 0)
time
Referee
ok
Leader
insert...
Follower 1
insert...
Follower 2
Alice
select * from final_scores
Bob
select * from final_scores
Hey, Germany has won the world cup!
Really? The website says they're still playing.

Figure 9-1. This system is not linearizable, causing football fans to be confused.

Figure 9-1 shows an example of a non-linearizable sports website [9]. Alice and Bob are sitting in the same room, both checking their phones to see the outcome of the 2014 football world cup final. Just after the final score is announced, Alice refreshes the page, sees the winner announced, and excitedly tells Bob about it. Bob incredulously hits *reload* on his own phone, but his request goes to a database replica that is lagging, and so his phone shows that the game is still ongoing.

If Alice and Bob had hit reload at the same time, it wouldn't have been surprising if they had got two different query results, because they don't know at exactly what time their respective requests were processed by the server. However, Bob knows that he hit the reload button (initiated his query) *after* he heard Alice exclaim the final score, and therefore he expects his query result to be at least as recent as Alice's. The fact that his query returned a stale result is a violation of linearizability.

What makes a system linearizable?

The basic idea behind linearizability is simple: make a system appear as if there was only a single copy of the data. However, nailing down precisely what that means actually requires some care. In order to understand linearizability better, let's look at some more examples.

Figure 9-2 shows three clients concurrently reading and writing the same key x in a linearizable database. In the distributed systems literature, x is called a *register* — in practice, it could be one key in a key-value store, one row in a relational database, or one document in a document database, for example.

For simplicity, Figure 9-2 shows only the requests from the clients' point of view, not the internals of the database. Each bar is a request made by a client, where the start of a bar is the time when the request was sent, and the end of a bar is when the response was received by the client. Due to variable network delays, a client doesn't know exactly when the database processed its request — it only knows that it must have happened sometime between the client sending the request and receiving the response.[i]

[i] A subtle detail of this diagram is that it assumes the existence of a global clock, represented by the horizontal axis. Even though real systems typically don't have accurate clocks (see "Unreliable Clocks" on page 278), this assumption is ok: for purposes of analyzing a distributed algorithm we may pretend that an accurate global clock exists, as long as the algorithm doesn't have access to it [50]. Instead, the algorithm can only see a mangled approximation of real time, as produced by a quartz oscillator and NTP.

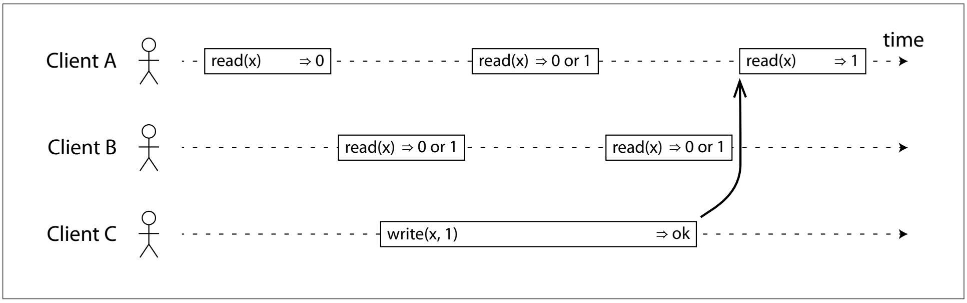


Figure 9-2. If a read request is concurrent with a write request, it may return either the old or the new value.

In this example, the register has two types of operation:

- *read*(*x*) ⇒ *v* means the client requested to read the value of register *x*, and the database returned the value *v*.
- *write*(*x*, *v*) ⇒ *r* means the client requested to set the register *x* to value *v*, and the database returned response *r* (which could be *ok* or *error*).

In Figure 9-2, the value of *x* is initially 0, and client C performs a write request to set it to 1. While this is happening, clients A and B are repeatedly polling the database to read the latest value. What are the possible responses that A and B might get for their read requests?

- The first read operation by client A completes before the write begins, so it must definitely return the old value 0.
- The last read by client A begins after the write has completed, so it must definitely return the new value 1 if the database is linearizable: we know that the write must have been processed sometime between the start and end of the write operation, and the read must have been processed sometime between the start and end of the read operation. If the read started after the write ended, then the read must have been processed after the write, and therefore it must see the new value that was written.
- Any read operations that overlap in time with the write operation might return either 0 or 1, because we don't know whether or not the write has taken effect at the time when the read operation is processed. These operations are *concurrent* with the write.

However, that is not yet sufficient to fully describe linearizability: if reads that are concurrent with a write can return either the old or the new value, then readers could see a value flip back and forth between the old and the new value several times while a write is going on. That is not what we expect of a system that emulates a "single copy of the data".[ii]

To make the system linearizable, we need to add another constraint, illustrated in Figure 9-3.

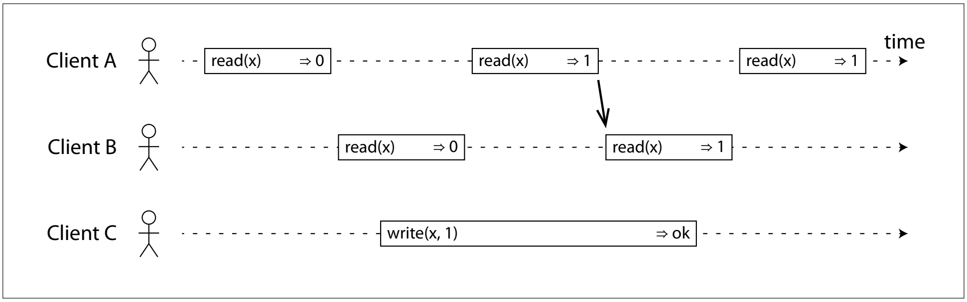


Figure 9-3. After any one read has returned the new value, all following reads (on the same or other clients) must also return the new value.

In a linearizable system we imagine that there must be some point in time (between the start and end of the write operation) at which the value of *x* atomically flips from 0 to 1. Thus, if one client's read returns the new value 1, all subsequent reads must also return the new value, even if the write operation has not yet completed.

This timing dependency is illustrated with an arrow in Figure 9-3. Client A is the first to read the new value 1. Just after A's read returns, B begins a new read. Since B's read occurs strictly after A's read, it must also return 1, even though the write by C is still ongoing. (It's the same situation as with Alice and Bob in Figure 9-1: after Alice has read the new value, Bob also expects to read the new value.)

We can further refine this timing diagram to visualize each operation taking effect atomically at some point in time. A more complex example is shown in Figure 9-4 [10].

[ii] This semantics, in which reads may return either the old or the new value if they are concurrent with a write, is known as a *regular register* [7, 30].

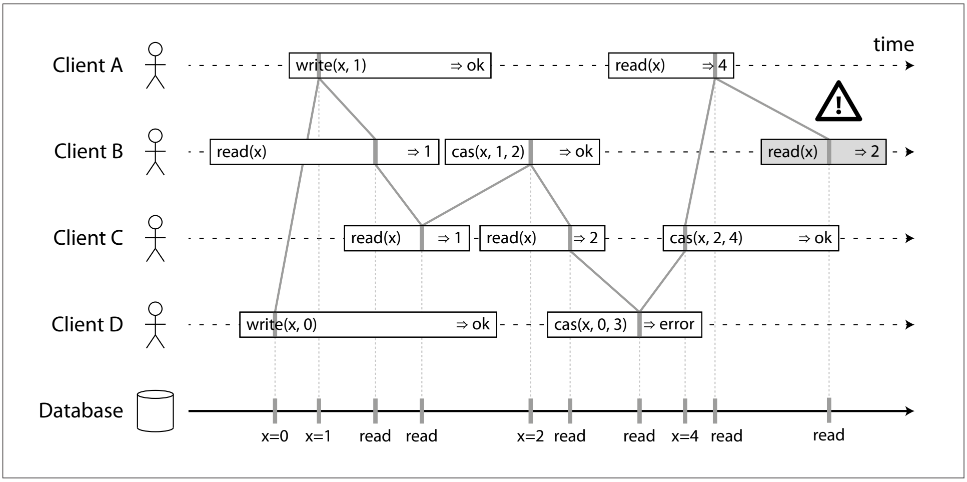


Figure 9-4. Visualizing the points in time at which the reads and writes appear to have taken effect. The final read by B is not linearizable.

In Figure 9-4 we add a third type of operation besides *read* and *write*:

- *cas*(x, v_{old}, v_{new}) ⇒ r means the client requested an atomic *compare-and-set* operation (see "Compare-and-set" on page 236). If the current value of the register x equals v_{old}, it should be atomically set to v_{new}. If $x \neq v_{old}$ then the operation should leave the register unchanged and return an error. r is the database's response (*ok* or *error*).

Each operation in Figure 9-4 is marked with a vertical line (inside the bar for each operation) at the time when we think the operation was executed. Those markers are joined up in a sequential order, and the result must be a valid sequence of reads and writes for a register (every read must return the value set by the most recent write).

The requirement of linearizability is that the lines joining up the operation markers always move forwards in time (from left to right), never backwards. That ensures the recency guarantee we discussed earlier: once a new value has been written or read, all subsequent reads see the value that was written, until it is overwritten again.

There are a few interesting details to point out in Figure 9-4:

- First client B sent a request to read x, then client D sent a request to set x to 0, and then client A sent a request to set x to 1. Nevertheless, the value returned to B's read is 1 (the value written by A). This is ok: it means that the database first processed D's write, then A's write, and finally B's read. Although this is not the order in which the requests were sent, it's an acceptable order, because the three requests are concurrent. Perhaps B's read request was slightly delayed in the network, so it only reached the database after the two writes.

- Client B's read returned 1 before client A received its response from the database, saying that the write of the value 1 was successful. This is also ok: it doesn't mean the value was read before it was written, it just means the "ok" response from the database to client A was slightly delayed in the network.
- This model doesn't assume any transaction isolation: another client may change a value at any time. For example, C first reads 1 and then reads 2, because the value was changed by B between the two reads. An atomic compare-and-set (*cas*) operation can be used to check the value hasn't been concurrently changed by another client: B and C's *cas* requests succeed, but D's *cas* request fails (by the time the database processes it, the value of *x* is no longer 0).
- The final read by client B (in a shaded bar) is not linearizable. The operation is concurrent with C's *cas* write, which updates x from 2 to 4; in the absence of other requests, it would be ok for B's read to return 2. However, client A has already read the new value 4 before B's read started, so B is not allowed to read an older value than A. Again, it's the same situation as with Alice and Bob in Figure 9-1.

That is the intuition behind linearizability; the formal definition [6] describes it more precisely. It is possible (though computationally expensive) to test whether a system's behavior is linearizable by recording the timings of all requests and responses, and checking whether they can be arranged into a valid sequential order [11].

Linearizability vs. serializability

Linearizability is easily confused with serializability (see "Serializability" on page 242), as both words seem to mean something like "can be arranged in a sequential order". However, they are two quite different guarantees, and it is important to distinguish them:

Serializability

Serializability is an isolation property of *transactions*, where every transaction may read and write multiple objects (rows, documents, records) — see "Single-object and multi-object operations" on page 219. It guarantees that transactions behave the same as if they had executed in *some* serial order (each transaction running to completion before the next transaction starts). It is ok for that serial order to be different from the order in which transactions were actually run [12].

Linearizability

Linearizability is a recency guarantee on reads and writes of a register (an *individual object*). It doesn't group operations together into transactions, so it does not prevent problems such as write skew (see "Preventing write skew and phantoms" on page 237), unless you take additional measures such as materializing conflicts (see "Materializing conflicts" on page 242).

A database may provide both serializability and linearizability, and this combination is known as *strict serializability* or *strong one-copy serializability* (strong-1SR) [4, 13]. Implementations of serializability based on 2-phase locking (see "Two-phase locking (2PL)" on page 248) or actual serial execution (see "Actual serial execution" on page 243) are typically linearizable.

However, serializable snapshot isolation (see "Serializable snapshot isolation (SSI)" on page 252) is not linearizable: by design, it makes reads from a consistent snapshot, to avoid lock contention between readers and writers. The whole point of a consistent snapshot is that it does not include writes that are more recent than the snapshot, and thus reads from the snapshot are not linearizable.

Relying on linearizability

In what circumstances is linearizability useful? Viewing the final score of a sporting match is perhaps a frivolous example: a result that is outdated by a few seconds is unlikely to cause any real harm in this situation. However, there a few areas in which linearizability is an important requirement for making a system work correctly.

Locking and leader election

A system that uses single-leader replication needs to ensure that there is indeed only one leader, not several (split brain). One way of electing a leader is to use a lock: every node that starts up tries to acquire the lock, and the one that succeeds becomes the leader [14]. No matter how this lock is implemented, it must be linearizable: all nodes must agree which node owns the lock, otherwise it is useless.

Coordination services like Apache ZooKeeper [15] and etcd [16] are often used to implement distributed locks and leader election. They use consensus algorithms to implement linearizable operations in a fault-tolerant way (we discuss such algorithms later in this chapter, in "Fault-tolerant consensus" on page 355).[iii] There are still many subtle details to implementing locks and leader election correctly (see for example the fencing issue in "The leader and the lock" on page 293), and libraries like Apache Curator [17] help by providing higher-level recipes on top of ZooKeeper. However, a linearizable storage service is the basic foundation for these coordination tasks.

[iii] Strictly speaking, ZooKeeper and etcd provide linearizable writes, but reads may be stale, since by default they can be served by any one of the replicas. You can optionally request a linearizable read: etcd calls this a *quorum read* [16], and in ZooKeeper you need to call `sync()` before the read [15] — see "Implementing linearizable storage using total order broadcast" on page 340.

Distributed locking is also used at a much more granular level in some distributed databases such as Oracle Real Application Clusters (RAC) [18]. RAC uses a lock per disk page, with multiple nodes sharing access to the same disk storage system. Since these linearizable locks are on the critical path of transaction execution, RAC deployments usually have a dedicated cluster interconnect network for communication between database nodes.

Constraints and uniqueness guarantees

Uniqueness constraints are common in databases: for example, a username or email address must uniquely identify one user, and in a file storage service there cannot be two files with the same path and filename. If you want to enforce this constraint as the data is written (i.e. if two people try to concurrently create a user or a file with the same name, one of them will be returned an error), you need linearizability. This situation is actually similar to a lock: when a user registers for your service, you can think of them acquiring a "lock" on their chosen username.

Similar issues arise if you want to ensure that a bank account balance never goes negative, or that you don't sell more items than you have in stock in the warehouse, or that two people don't concurrently book the same seat on a flight or in a theater. These constraints all require there to be a single up-to-date value (the account balance, the stock level, the seat occupancy) that all nodes agree on.

However, in these situations, you may also be able to get away without linearizability:

- If two people concurrently register the same username or book the same seat, you can send one of them an email to apologize, and ask them to choose a different one. This kind of change to correct a mistake is called a *compensating transaction* [19, 20].
- If customers order more items than you have in your warehouse, you can order in more stock, apologize to customers for the delay, and offer them a discount. This is actually the same as you'd have to do if a fork-lift truck runs over one of the items in your warehouse, leaving you with fewer items in stock than you thought you had [21]. Thus, the apology workflow already needs to be part of your business processes anyway, and so it might be unnecessary to require a linearizable constraint on the number of items in stock.
- If someone withdraws more money than they have in their account, you can charge them an overdraft penalty fee and laugh all the way to the bank [22]. By limiting the maximum amount that can be withdrawn per day, the risk to the bank is bounded [23].

Other kinds of constraint, such as foreign key or attribute constraints, can be implemented without requiring linearizability [24].

Cross-channel timing dependencies

Notice a detail in Figure 9-1: if Alice hadn't exclaimed the score, Bob wouldn't have known that the result of his query was stale. He would have just refreshed the page again a few seconds later, and eventually seen the final score. The linearizability violation was only noticed because there was an additional communication channel in the system (Alice's voice to Bob's ears).

Similar situations can arise in computer systems. For example, say you have a website where users can upload a photo, and a background process resizes the photos to lower resolution for faster download (thumbnails). The architecture and data flow of this system is illustrated in Figure 9-5.

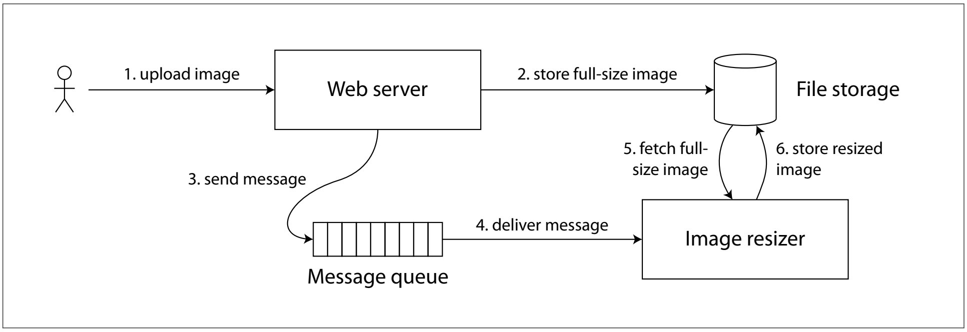


Figure 9-5. The web server and image resizer communicate both through file storage and a message queue, opening the potential for race conditions.

The image resizer needs to be explicitly instructed to perform a resizing job, and this instruction is sent from the web server to the resizer via a message queue (see Chapter 11). The web server doesn't place the entire photo on the queue, since most message brokers are designed for small messages, and a photo may be several megabytes in size. Instead, the photo is first written to a file storage service, and then the instruction to the resizer is placed on the queue.

If the file storage service is linearizable, then this system should work fine. If it is not linearizable, there is the risk of a race condition: the message queue (steps 3 and 4 in Figure 9-5) might be faster than the internal replication inside the storage service. In this case, when the resizer fetches the image (step 5), it might see an old version of the image, or nothing at all. If it processes an old version of the image, the full-size and the resized images in file storage become permanently inconsistent.

This problem arises because there are two different communication channels between the web server and the resizer: the file storage and the message queue. Without the recency guarantee of linearizability, race conditions between these two channels are possible. This is analogous to Figure 9-1, where there was also a race condition between two communication channels: the database replication, and the real-life audio channel between Alice's mouth and Bob's ears.

Linearizability is not the only way of avoiding this race condition, but it's the simplest to understand. If you control the additional communication channel (like in the case of the message queue, but not in the case of Alice and Bob), you can use alternative approaches similar to what we discussed in "Reading your own writes" on page 156, at the cost of additional complexity.

Implementing linearizable systems

Now that we've looked at a few examples in which linearizability is useful, let's think about how we might implement a system that offers linearizable semantics.

Since linearizability essentially means "behave as though there was only a single copy of the data, and all operations on it are atomic", the simplest answer would be to really only use a single copy of the data. However, that approach would not be able to tolerate faults: if the node holding that one copy fails, the data would be lost (or at least inaccessible until the node is brought up again).

The most common approach to making a system fault-tolerant is to use replication. Let's revisit the replication methods from Chapter 5, and compare whether they can be made linearizable:

Single-leader replication (potentially linearizable)

In a system with single-leader replication (see "Leaders and Followers" on page 146), the leader has the primary copy of the data that is used for writes, and the followers maintain backup copies of the data on other nodes. If you make reads from the leader, or from synchronously updated followers, they have the *potential* to be linearizable.[iv] However, not every single-leader database is actually linearizable, either by design (e.g. because it uses snapshot isolation) or due to concurrency bugs [10].

Using the leader for reads relies on the assumption that you know for sure who the leader is. As discussed in "The truth is defined by the majority" on page 292, it is quite possible for a node to think that it is leader, when in fact it is not — and if the delusional leader continues to serve requests, it is likely to violate linearizability [25]. With asynchronous replication, failover may even lose data (see "Handling node outages" on page 150), which violates both durability and linearizability.

[iv] Partitioning (sharding) a single-leader database, so that there is a separate leader per partition, does not affect linearizability, since it is only a single-object guarantee. Cross-partition transactions are a different matter (see "Distributed Transactions and Consensus" on page 343).

Consensus algorithms (linearizable)

Some consensus algorithms, which we will discuss later in this chapter, bear a resemblance to single-leader replication. However, consensus protocols contain measures to prevent split-brain and stale replicas. Thanks to these details, consensus algorithms can implement linearizable storage safely. This is how ZooKeeper [26] and etcd [27] work, for example.

Multi-leader replication (not linearizable)

Systems with multi-leader replication are generally not linearizable, because they concurrently process writes on multiple nodes and asynchronously replicate them to other nodes. For this reason, they can produce conflicting writes that require resolution (see "Handling write conflicts" on page 164).

Leaderless replication (probably not linearizable)

For systems with leaderless replication (Dynamo-style, see "Leaderless replication" on page 171), people sometimes claim that you can obtain "strong consistency" by requiring quorum reads and writes ($w + r > n$). Depending on the exact configuration of the quorums, and depending on how you define strong consistency, this is not quite true.

"Last write wins" conflict resolution methods based on time-of-day clocks (e.g. in Cassandra, see "Relying on synchronized clocks" on page 282) are almost certainly non-linearizable, because clock timestamps cannot be guaranteed to be consistent with actual event ordering due to clock skew. Sloppy quorums ("Sloppy quorums and hinted handoff" on page 177) also ruin any chance of linearizability. Even with strict quorums, non-linearizable behavior is possible, as demonstrated in the next section.

Linearizability and quorums

Intuitively, it seems as though strict quorum reads and writes should be linearizable in a Dynamo-style model. However, when we have variable network delays, it is possible to have race conditions, as demonstrated in Figure 9-6.

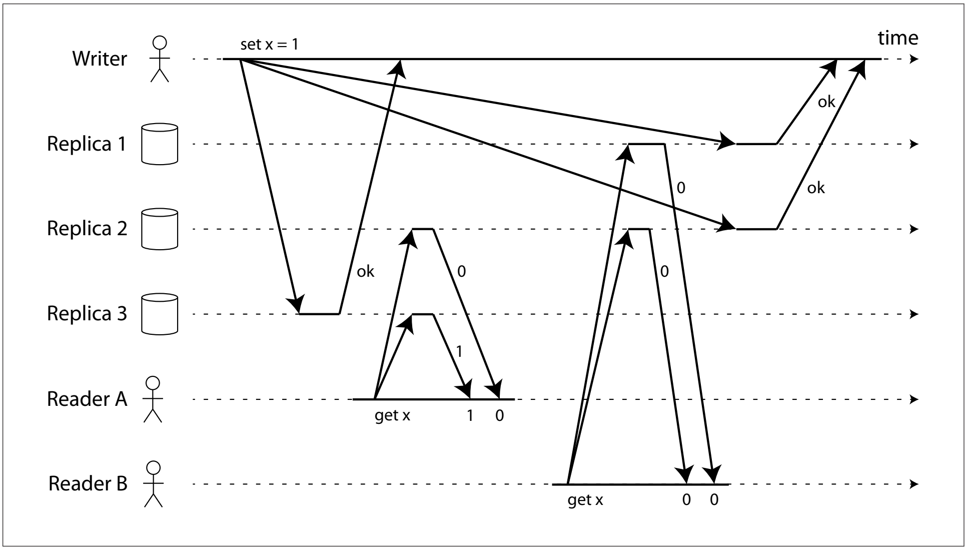


Figure 9-6. A non-linearizable execution, despite using a strict quorum.

In Figure 9-6, the initial value of x is 0, and a writer client is updating x to 1 by sending the write to all three replicas ($n = 3$, $w = 3$). Concurrently, client A reads from a quorum of 2 nodes ($r = 2$) and sees the new value 1 on one of the nodes. Also concurrently with the write, client B reads from a different quorum of 2 nodes, and gets back the old value 0 from both.

The quorum condition is met ($w + r > n$), but this execution is nevertheless not linearizable: B's request begins after A's request completes, but B returns the old value while A returns the new value. (It's once again the Alice and Bob situation from Figure 9-1.)

Interestingly, it *is* possible to make Dynamo-style quorums linearizable at the cost of reduced performance: a reader must perform read repair (see "Read repair and anti-entropy" on page 172) synchronously, before returning results to the application [28], and a writer must read the latest state of a quorum of nodes before sending its writes [29, 30]. However, Riak does not do this due to the performance penalty [31]. Cassandra *does* wait for read repair to complete on quorum reads [32], but it loses linearizability if there are multiple concurrent writes to the same key, due to its use of last-write-wins.

Moreover, only linearizable *read* and *write* operations can be implemented in this model, but a linearizable compare-and-set operation cannot—it requires a consensus algorithm [33].

In summary, it is safest to assume that a leaderless system with Dynamo-style replication does not provide linearizability.

The cost of linearizability

As some replication methods can provide linearizability and others cannot, it is interesting to explore the pros and cons of linearizability in more depth.

We already discussed some use cases for different replication methods in Chapter 5; for example, we saw that multi-leader replication is often a good choice for multi-datacenter replication (see "Multi-datacenter operation" on page 162). An example of such a deployment is illustrated in Figure 9-7.

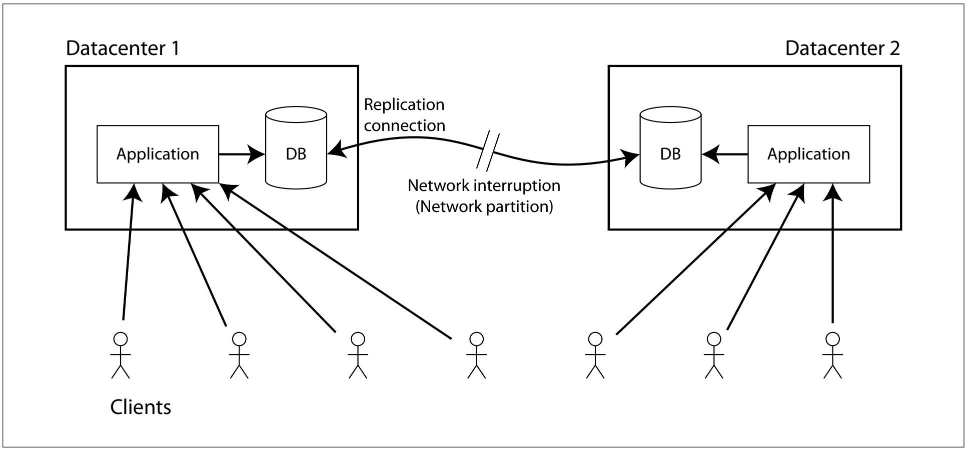


Figure 9-7. A network interruption forcing a choice between linearizability and availability.

Consider what happens if there is a network interruption between the two datacenters. Assume the network within each datacenter is working, and clients can reach the datacenters, but the datacenters cannot connect to each other.

With a multi-leader database, each datacenter can continue operating normally: since writes from one datacenter are asynchronously replicated to the other, the writes are simply queued up and exchanged when network connectivity is restored.

On the other hand, if single-leader replication is used, then the leader must be in one of the datacenters. Any writes and any linearizable reads must be sent to the leader— thus, for any clients connected to a follower datacenter, those read and write requests must be sent synchronously over the network to the leader datacenter.

If the network between datacenters is interrupted in a single-leader setup, clients connected to follower datacenters cannot contact the leader, so they cannot make any writes to the database, nor any linearizable reads. They can still make reads from the follower, but they will be stale (non-linearizable). If the application requires linearizable reads and writes, the network interruption causes the application to become unavailable in the datacenters that cannot contact the leader.

If clients can connect directly to the leader datacenter, then this is not a problem, since the application continues to work normally there. But if clients can only reach a follower datacenter, they will see an outage of the application until the network link is repaired.

The CAP theorem

This issue is not just a consequence of single-leader and multi-leader replication: any linearizable database has this problem, no matter how it is implemented. The issue also isn't specific to multi-datacenter deployments, but can occur on any unreliable network, even within one datacenter (see "Unreliable Networks" on page 269).

The trade-off is as follows:[v]

- If your application *requires* linearizability, and some replicas are disconnected from the other replicas due to a network problem, then some replicas cannot process requests while they are disconnected: they must either wait until the network problem is fixed, or return an error (either way, they become *unavailable*).
- If your application *does not require* linearizability, then it can be written in a way that each replica can process requests independently, even if it is disconnected from other replicas (e.g. multi-leader). In this case, the application can remain *available* in the face of a network problem, but its behavior is not linearizable.

Thus, applications that don't require linearizability can be more tolerant of network problems. This insight is popularly known as the *CAP theorem* [34, 35, 36, 37], named by Eric Brewer in 2000, although the trade-off was already known to designers of distributed databases since the 1970s [22, 38, 39, 40].

CAP was originally proposed as a rule of thumb, without precise definitions, with the goal of starting a discussion about trade-offs in databases. At the time, many distributed databases focused on providing linearizable semantics on a cluster of machines with shared storage [18], and CAP encouraged database engineers to explore a wider design space of distributed shared-nothing systems, which were more suitable for implementing large-scale web services [23]. CAP deserves credit for this culture shift — witness the explosion of new database technologies since the mid-2000s (known as NoSQL).

The CAP theorem, as formally defined [35] is of very narrow scope: it only considers one consistency model (namely linearizability) and one kind of fault (network parti-

[v] These two choices are sometimes known as CP (consistent but not available under network partitions) and AP (available but not consistent under network partitions), respectively. However, this classification scheme has several flaws [9], so it is best avoided.

tions,[vi] i.e. nodes that are alive but disconnected from each other). It doesn't say anything about network delays, dead nodes, or other trade-offs. Thus, although CAP has been historically influential, it has little practical value for designing systems [9, 41].

There are many more interesting impossibility results in distributed systems [42], and CAP has now been superseded by more precise results [2, 43], so it is of mostly historical interest today.

The Unhelpful CAP Theorem

CAP is sometimes presented as *Consistency, Availability, Partition tolerance: pick 2 out of 3*. Unfortunately, putting it this way is misleading [37] because network partitions are a kind of fault, so they aren't something you would normally choose: either they happen or they don't [44].

At times when the network is working correctly, a system can provide both consistency (linearizability) and total availability. When a network fault occurs, you have to choose between either linearizability and total availability. Thus, a better way of phrasing CAP would be *either Consistent or Available when Partitioned* [45].

In discussions of CAP there are several contradictory definitions of the term *availability*, and the formalization as a theorem [35] does not match its usual meaning [41]. Many so-called "highly available" (fault tolerant) systems actually do not meet CAP's idiosyncratic definition of availability. All in all, there is a lot of misunderstanding and confusion around CAP, and it does not help us understand systems better, so CAP is best avoided.

Linearizability and network delays

Although linearizability is a useful guarantee, surprisingly few systems are actually linearizable in practice. For example, even RAM on a modern multicore CPU is not linearizable [46]: if a thread running on one CPU core writes to a memory address, and a thread on another CPU core reads the same address shortly afterwards, it is not guaranteed to read the value written by the first thread (unless a *memory barrier* or *fence* [47] is used).

The reason for this behavior is that every CPU core has its own memory cache and store buffer. Memory access first goes to the cache by default, and any changes are asynchronously written out to main memory. Since accessing data in the cache is

[vi] As discussed in "Network faults in practice" on page 271, this book uses *partitioning* to refer to deliberately breaking down a large dataset into smaller ones (*sharding*, see Chapter 6). By contrast, a network partition is a particular type of network fault, which we normally don't consider separately from other kinds of fault. However, since it's the P in CAP, we can't avoid the confusion in this case.

much faster than going to main memory [48], this feature is essential for good performance on modern CPUs. However, there are now several copies of the data (one in main memory, and perhaps several more in various caches), and these copies are asynchronously updated, so linearizability is lost.

Why make this trade-off? It makes no sense to use the CAP theorem to justify the multi-core memory consistency model: within one computer we usually assume reliable communication, and we don't expect one CPU core to be able to continue operating normally if it is disconnected from the rest of the computer. The reason for dropping linearizability is *performance*, not fault tolerance.

The same is true of many distributed databases that choose not to provide linearizable guarantees: they do so primarily to increase performance, not so much for fault tolerance [49]. Linearizability is slow — and this is true all the time, not only during a network fault.

Can't we maybe find a more efficient implementation of linearizable storage? It seems the answer is no: Attiya and Welch [50] prove that if you want linearizability, the response time of read and write requests is at least proportional to the uncertainty of delays in the network. In a network with highly variable delays, like most computer networks (see "Timeouts and unbounded delays" on page 273), the response time of linearizable reads and writes is inevitably going to be high. A faster algorithm for linearizability does not exist, but weaker consistency models can be much faster, so this is an important trade-off for latency-sensitive systems.

Ordering Guarantees

We said previously that a linearizable register behaves as if there was only a single copy of the data, and that every operation appears to take effect atomically at one point in time. This implies that operations are executed in some well-defined order. We illustrated the ordering in Figure 9-4 by joining up the operations in the order in which they seem to have executed.

Ordering has been a recurring theme in this book, which suggests that it might be an important fundamental idea. Let's briefly recap some of the other contexts in which we have discussed ordering:

- In Chapter 5 we saw that the main purpose of the leader in single-leader replication is to determine the *order of writes* in the replication log — that is, the order in which followers apply those writes. If there is no single leader, conflicts can occur due to concurrent operations (see "Handling write conflicts" on page 164).
- Serializability in Chapter 7 is about ensuring that transactions behave as if they were executed in *some sequential order*. It can be achieved by literally executing

transactions in serial order, or by allowing concurrent execution while preventing serialization conflicts (by locking or aborting).

- The use of timestamps and clocks in distributed systems that we discussed in Chapter 8 (see "Relying on synchronized clocks" on page 282) is another attempt to introduce *order* into a disorderly world, for example to determine which one of two writes happened later.

Interestingly, it turns out that there are deep connections between ordering, linearizability, and consensus. Although this notion is a bit more theoretical and abstract than much of the rest of this book, it is very helpful for clarifying our understanding of what systems can and cannot do. We will explore this topic in the next few sections.

Ordering and causality

There are several reasons why ordering keeps coming up, and one of the reasons is that it helps preserve *causality*. We have already seen several examples over the course of this book where causality has been important:

- In "Consistent prefix reads" on page 159 (Figure 5-5) we saw an example where the observer of a conversation saw first the answer to a question, and then the question being answered. This is confusing because it violates our intuition for cause and effect: if a question is answered, then clearly the question had to be there first, because the person giving the answer must have seen the question (assuming they are not psychic and cannot see into the future). We say that there is a *causal relationship* between the question and the answer.
- A similar pattern appeared in Figure 5-9, where we looked at the replication between three leaders, and noticed that some writes could "overtake" others due to network delays. From the perspective of one of the replicas it would look as though there was an update to a row that did not exist. Causality here means that a row must first be created before it can be updated.
- In "Detecting concurrent writes" on page 178 we observed that if 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. This *happened before* relationship is another expression of causality: if A happened before B, that means B might have known about A, or built upon A, or depended on A. If A and B are concurrent, there is no causal link between them, i.e. we are sure that neither knew about the other.
- In the context of snapshot isolation for transactions ("Snapshot isolation and repeatable read" on page 228), we said that a transaction reads from a consistent snapshot. But what does "consistent" mean in this context? It means *consistent with causality*: if the snapshot contains an answer, it must also contain the ques-

tion being answered [51]. Observing the entire database at a single point in time makes it consistent with causality: the effect of all operations that happened causally before that point in time are visible, but no operations that happened causally afterwards can be seen. Read skew (inconsistent reads, as illustrated in Figure 7-6) means reading data in a state that violates causality.

- Our examples of write skew between transactions (see "Preventing write skew and phantoms" on page 237) also demonstrated causal dependencies: in Figure 7-8, Alice was allowed to go off-call because the transaction thought that Bob is still on-call, and vice versa. In this case, the action of going off-call is causally dependent on the observation of who is currently on-call. Serializable snapshot isolation (see "Serializable snapshot isolation (SSI)" on page 252) detects write skew by tracking the causal dependencies between transactions.
- In the example of Alice and Bob watching football (Figure 9-1), the fact that Bob got a stale result from the server after hearing Alice exclaim the result is a causality violation: Alice's exclamation is causally dependent on the announcement of the score, so Bob should also be able to see the score after hearing Alice. The same pattern reappeared again in "Cross-channel timing dependencies" on page 322 in the guise of an image resizing service.

Looking at it like this, many of the last 200 pages of this book could be boiled down to the three words: *"watch your causality!"*

To be consistent with causality means to obey the ordering imposed by causality. Cause comes before effect. A message is sent before that message is received. The question comes before the answer. And, like in real life, one thing leads to another: one node reads some data and then writes something as a result, another node reads the thing that was written and writes something else in turn, and so on. These chains of causally dependent operations define the causal order in the system — i.e. what happened before what.

The causal order is not a total order

A *total order* allows any two elements to be compared, so if you have two elements, you can always say which one is greater and which one is smaller. For example, natural numbers are totally ordered: if I give you any two numbers, say 5 and 13, you can tell me that 13 is greater than 5.

However, mathematical sets are not totally ordered: is $\{a, b\}$ greater than $\{b, c\}$? Well, you can't really compare them, because neither is a subset of the other. We say they are *incomparable*, and therefore mathematical sets are *partially ordered*: in some cases, one set is greater than another (if one set contains all the elements of another), but in other cases they are incomparable.

The difference between a total order and a partial order is reflected in different database consistency models:

Linearizability
: In a linearizable system, we have a *total order* of operations: if the system behaves as if there is only a single copy of the data, and every operation is atomic, this means that for any two operations we can always say which one happened first. This total ordering is illustrated as a timeline in Figure 9-4.

Causality
: We said that two operations are concurrent if neither happened before the other (see "The "happens-before" relationship and concurrency" on page 180). Put another way, two events are ordered if they are causally related (one happened before the other), but they are incomparable if they are concurrent. This means that causality defines a *partial order*, not a total order: some operations are ordered with respect to each other, but some are incomparable.

Therefore, according to this definition, there are no concurrent operations in a linearizable datastore: there must be a single timeline along which all operations are totally ordered. There might be several requests waiting to be handled, but the datastore ensures that every request is handled atomically at a single point in time, acting on a single copy of the data, along a single timeline, without any concurrency.

Concurrency would mean that the timeline branches and merges again — and in this case, operations on different branches are incomparable, i.e. concurrent. We saw this in Chapter 5: for example, Figure 5-14 is not a straight-line total order, but rather a jumble of different operations going on concurrently. The arrows in the diagram indicate causal dependencies — the partial ordering of operations.

If you are familiar with distributed version control systems such as git, their version histories are very much like the graph of causal dependencies. Often one commit happens after another, in a straight line, but sometimes you get branches (when several people concurrently work on a project), and merges are created when those concurrently created commits are combined.

Linearizability is stronger than causal consistency

So what is the relationship between the causal order and linearizability? The answer is that linearizability *implies* causality: any system that is linearizable will preserve causality correctly [7]. In particular, if there are multiple communication channels in a system (such as the message queue and the file storage service in Figure 9-5), linearizability ensures that causality is automatically preserved without having to do anything special (such as passing around timestamps between different components).

The fact that linearizability ensures causality is what makes linearizable systems simple to understand and appealing. However, as discussed in "The cost of linearizability" on page 326, making a system linearizable can harm its performance and availability, especially if the system has significant network delays (for example if it's geographically distributed). For this reason, many distributed data systems have abandoned all consistency guarantees, which gives them better performance, but can make them difficult to work with.

The good news is that a middle ground is possible. Linearizability is not the only way of preserving causality — there are other ways too. A system can be causally consistent without incurring the performance hit of making it linearizable (in particular, the CAP theorem does not apply). In fact, causal consistency is the strongest possible consistency model that does not slow down due to network delays, and remains available in the face of network failures [2, 43].

In many cases, systems that appear to require linearizability in fact only really require causal consistency, which can be implemented more efficiently. Based on this observation, researchers are exploring new kinds of database that preserve causality, with performance and availability characteristics that are similar to eventually consistent systems [52, 53, 54].

As this research is quite recent, not much of it has yet made its way into production systems, and there are still challenges to be overcome [55, 56]. However, it is a promising direction for future systems.

Tracking causal relationships

We won't go into all the nitty-gritty details of how non-linearizable systems can maintain causal consistency, but just briefly point out some of the key ideas.

In order to maintain causality, you need to know which operation *happened before* which other operation. This is a partial order: concurrent operations may be processed in any order, but if one operation happened before another, then they must be processed in that order on every replica. Thus, when a replica processes an operation, it must ensure that all causally preceding operations (all operations that happened before) have already been processed; if some preceding operation is missing, the later operation must wait until the preceding operation has been processed.

In order to determine causal relationships, we need some way of describing the "knowledge" of a node in the system. If a node had already seen the value X when it issued the write Y, then X and Y may be causally related. The arguments start looking like the kinds of questions you would expect in a criminal investigation of fraud charges: did the CEO *know* about X at the time when they made decision Y?

The techniques for determining which operation happened before which other operation are similar to what we discussed in "Detecting concurrent writes" on page 178. That section discussed causality in a leaderless datastore, where we need to detect concurrent writes to the same key in order to prevent lost updates. Causal consistency goes further: it needs to track causal relationships across the entire database, not just for a single key. Version vectors can be generalized to do this [57].

In order to determine the causal ordering, the database needs to know which version of the data was read by the application. This is why, in Figure 5-13, the version number from the prior operation is passed back to the database on a write. A similar idea appears in the conflict detection of Serializable Snapshot Isolation, as discussed in "Serializable snapshot isolation (SSI)" on page 252: when a transaction wants to commit, the database checks whether the version of the data that it read is still up-to-date. To this end, the database keeps track of which data has been read by which transaction.

Sequence number ordering

Although causality is an important theoretical concept, actually keeping track of all causal dependencies can become impractical. In many applications, clients read lots of data before writing something, and then it is not clear whether the write is causally dependent on all or only some of those prior reads. Explicitly tracking all the data that has been read would mean a large overhead.

However, there is a better way: we can use *sequence numbers* or *timestamps* to order events. A timestamp need not come from a time-of-day clock (physical clock), which have many problems as discussed in "Unreliable Clocks" on page 278. A timestamp could also come from a *logical clock*, which is an algorithm to generate a sequence of numbers to identify operations — typically using counters that are incremented for every operation.

Such sequence numbers or timestamps are compact (only a few bytes in size), and they provide a *total order*: that is, every operation has a unique sequence number, and you can always compare two sequence numbers to determine which is greater (i.e. which operation happened later).

In particular, we can create sequence numbers in a total order that is *consistent with causality*:[vii] we promise that if operation A causally happened before B, then A occurs before B in the total order (A has a lower sequence number than B). Concurrent operations may ordered arbitrarily. Such a total order captures all the causality information, but also imposes more ordering than strictly required by causality.

In a database with single-leader replication (see "Leaders and Followers" on page 146), the replication log defines a total order of write operations that is consistent

[vii] A total order that is *inconsistent* with causality is easy to create, but not very useful. For example, you can generate a random UUID for each operation, and compare UUIDs lexicographically to define the total ordering of operations. This is a valid total order, but the random UUIDs tell you nothing about which operation actually happened first, or whether the operations were concurrent.

with causality. The leader can simply increment a counter for each operation, and thus assign a monotonically increasing sequence number to each operation in the replication log. If a follower applies the writes in the order they appear in the replication log, the state of the follower is always causally consistent (even if it is lagging behind the leader).

Non-causal sequence number generators

If there is not a single leader (perhaps because you are using a multi-leader or leaderless database, or because the database is partitioned), it is less clear how to generate sequence numbers for operations. Various methods are used in practice:

- Each node can generate its own independent set of sequence numbers. For example, if you have two nodes, one node can generate only odd numbers and the other only even numbers. In general, you could reserve some bits in the binary representation of the sequence number to contain a unique node identifier, and this would ensure that two different nodes can never generate the same sequence number.
- Another method is to attach a timestamp from a time-of-day clock (physical clock) to each operation [58]. Such timestamps are not sequential, but if they have sufficiently high resolution, they might be sufficient to totally order operations. This fact is used in the *last write wins* (LWW) conflict resolution method — see "Timestamps for ordering events" on page 283.
- A third option is to pre-allocate blocks of sequence numbers. For example, node A might claim the block of sequence numbers from 1 to 1,000, and node B might claim the block from 1,001 to 2,000. Then each node can independently assign sequence numbers from its block, and allocate a new block when its supply of sequence numbers begins to run low.

These three options perform better and are more scalable than pushing all operations through a single leader that increments a counter. They generate a unique, approximately increasing sequence number for each operation. However, they all have a problem: the sequence numbers they generate are *not consistent with causality*.

The causality problems occur because these sequence number generators do not correctly track ordering of operations across different nodes:

- Each node may process a different number of operations per second. Thus, if one node generates even numbers and the other generates odd numbers, the counter for even numbers may lag behind the counter for odd numbers, or vice versa. If you have an odd-numbered operation and an even-numbered operation, you cannot accurately tell which one causally happened first.

- Timestamps from physical clocks are subject to clock skew, which can make them inconsistent with causality. For example, see Figure 8-3, which shows a scenario in which an operation that happened causally later was actually assigned a lower timestamp.[viii]
- In the case of the block allocator, one operation may be given a sequence number in the range from 1,001 to 2,000, and a causally later operation may be given a number in the range from 1 to 1,000. Here, again, the sequence number is inconsistent with causality.

Lamport timestamps

Although the three sequence number generators above are inconsistent with causality, there is actually a simple method for generating sequence numbers that *is* consistent with causality. It is called a *Lamport timestamp*, proposed in 1978 by Leslie Lamport [59], in what is now one of the most-cited papers in the field of distributed systems.

The use of Lamport timestamps is illustrated in Figure 9-8. Each node has a unique identifier, and each node keeps a counter of the number of operations it has processed. The Lamport timestamp is then simply a pair of (*counter*, *node ID*). Two nodes may sometimes have the same counter value, but by including the node ID in the timestamp, each timestamp is made unique.

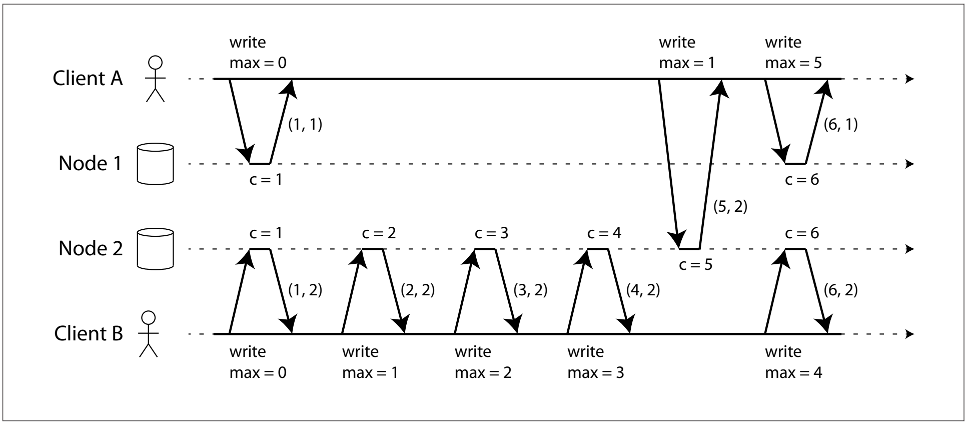


Figure 9-8. Lamport timestamps provide a total ordering consistent with causality.

[viii] It is possible to make physical clock timestamps consistent with causality: in "Synchronized clocks for global snapshots" on page 285 we discussed Google's Spanner, which estimates the expected clock skew and waits out the uncertainty interval before committing a write. This ensures that a causally later transaction is given a greater timestamp. However, most clocks cannot provide the required uncertainty metric.

A Lamport timestamp bears no relationship to a physical time-of-day clock, but it provides total ordering: if you have two timestamps, the one with a greater counter value is the greater timestamp; if the counter values are the same, the one with the greater node ID is the greater timestamp.

So far this is essentially the same as the even/odd counters in the last section. The key idea about Lamport timestamps, which makes them consistent with causality, is the following: every node and every client keeps track of the *maximum* counter value it has seen so far, and includes that maximum on every request. When a node receives an incoming request with a maximum counter greater than its own counter, it immediately increases its own counter to that maximum.

This is shown in Figure 9-8, where client A receives a counter value of 5 from node 2, and then sends that maximum of 5 to node 1. At that time, node 1's counter was only 1, but it was immediately moved forward to 5, so the next operation had an incremented counter value of 6.

As long as the maximum counter value is carried along with every operation, this scheme ensures that the ordering from the Lamport timestamps is consistent with causality, because every causal dependency results in an increased timestamp.

Lamport timestamps are sometimes confused with version vectors, which we saw in "Detecting concurrent writes" on page 178. Although there are some similarities, they have a different purpose: version vectors can distinguish whether two operations are concurrent or whether one is causally dependent on the other, whereas Lamport timestamps always enforce a total ordering. From the total ordering of Lamport timestamps you cannot tell whether two operations are concurrent or whether they are causally dependent.

Timestamp ordering is not sufficient

Although Lamport timestamps define a total order of operations that is consistent with causality, they are not quite sufficient to solve many common problems in distributed systems.

For example, consider a system that needs to ensure that a username uniquely identifies a user account. If two users concurrently try to create an account with the same username, one of the two should succeed and the other should fail. (We touched on this problem previously in "The leader and the lock" on page 293.)

At first glance, it seems as though a total ordering of operations (e.g. using Lamport timestamps) should be sufficient to solve this: if two accounts with the same username are created, pick the one with the lower timestamp as the winner (the one who grabbed the username first), and let the one with the greater timestamp fail. Since timestamps are totally ordered, this comparison is always valid.

This approach works for determining the winner after the fact: once you have collected all the username creation operations in the system, you can compare their timestamps. However, it is not sufficient when a node has just received a request from a user to create a username, and needs to decide *right now* whether the request should succeed or fail. At that moment, the node does not know whether another node is concurrently in the process of creating an account with the same username, and what timestamp that other node may assign to the operation.

In order to be sure that no other node is in the process of concurrently creating an account with the same username and a lower timestamp, you would have to check with every other node to see what it is doing [59]. If one of the other nodes has failed or cannot be reached due to a network problem, this system would grind to a halt. This is not the kind of fault tolerant system that we need.

The problem here is that the total order of operations only emerges after you have collected all of the operations. If another node has generated some operations, but you don't yet know what they are, you cannot construct the final ordering of operations: the unknown operations from the other node may need to be inserted at various positions in the total order.

To conclude: in order to implement something like a uniqueness constraint for usernames, it's not sufficient to have a total ordering of operations — you also need to know when that order is finalized. If you have an operation to create a username, and you are sure that no other node can insert a claim for the same username ahead of your operation in the total order, then you can safely declare the operation successful.

This idea of knowing when your total order is finalized is captured in the topic of *total order broadcast*.

Total order broadcast

If your program runs only on a single CPU core, it is easy to define a total ordering of operations: it is simply the order in which they were executed by the CPU. However, in a distributed system, getting all nodes to agree on the same total ordering of operations is tricky. In the last section we discussed ordering by timestamps or sequence numbers, but found that it is not as powerful as single-leader replication (it is not sufficient for implementing a uniqueness constraint).

As discussed, single-leader replication determines a total order of operations by choosing one node as the leader, and sequencing all operations on a single CPU core on the leader. The challenge then is how to scale the system if the throughput is greater than a single leader can handle, and also how to handle failover in case the leader fails (see "Handling node outages" on page 150). In the distributed systems literature, this problem is known as *total order broadcast* or *atomic broadcast* [30, 60, 61].[ix]

Scope of ordering guarantee

Partitioned databases with a single leader per partition often maintain ordering only per partition, which means they cannot offer consistency guarantees (e.g. consistent snapshots, foreign key references) across partitions. Total ordering across all partitions is possible, but requires additional coordination [62].

Total order broadcast is usually described as a protocol for exchanging messages between nodes. Informally, it requires that two properties are always satisfied:

Reliable delivery
: No messages are lost: if a message is delivered to one node, it is delivered to all nodes.

Totally ordered delivery
: Messages are delivered to every node in the same order.

A correct algorithm for total order broadcast must ensure that the reliability and ordering properties are always satisfied, even if nodes or the network are faulty. Of course, messages will not be delivered while the network is interrupted, but an algorithm can keep retrying so that the messages get through when the network is eventually repaired (and then they must still be delivered in the correct order).

Using total order broadcast

Consensus services such as ZooKeeper and etcd actually implement total order broadcast. This is a hint that there is a strong connection between total order broadcast and consensus, which we will explore later in this chapter.

Total order broadcast is exactly what you need for database replication: if every message represents a write to the database, and every replica processes the same writes in the same order, then the replicas will remain consistent with each other (aside from any temporary replication lag). This principle is known as *state machine replication* [63], and we will return to it in Chapter 11.

Similarly, total order broadcast can be used to implement serializable transactions: as discussed in "Actual serial execution" on page 243, if every message represents a

[ix] The term *atomic broadcast* is traditional, but it is very confusing as it's inconsistent with other uses of the word *atomic*: it has nothing to do with atomicity in ACID transactions, and is only indirectly related to atomic operations (in the sense of multithreaded programming) or atomic registers (linearizable storage). The term *total order multicast* is another synonym.

deterministic transaction to be executed as a stored procedure, and if every node processes those messages in the same order, then the partitions and replicas of the database are kept consistent with each other [64].

An important aspect of total order broadcast is that the order is fixed at the time the messages are delivered: a node is not allowed to retroactively insert a message into an earlier position in the order if subsequent messages have already been delivered. This fact makes total order broadcast stronger than timestamp ordering.

Another way of looking at total order broadcast is that it is a way of creating a *log* (as in a replication log, transaction log or write-ahead log): delivering a message is like appending to the log. Since all nodes must deliver the same messages in the same order, all nodes can read the log and see the same sequence of messages.

Total order broadcast is also useful for implementing a lock service that provides fencing tokens (see "Fencing tokens" on page 294). Every request to acquire the lock is appended as a message to the log, and all messages are sequentially numbered in the order they appear in the log. The sequence number can then serve as fencing token, because it is monotonically increasing. In ZooKeeper, this sequence number is called `zxid` [15].

Implementing linearizable storage using total order broadcast

As illustrated in Figure 9-4, in a linearizable system there is a total order of operations. Does that mean linearizability is the same as total order broadcast? Not quite, but there are close links between the two.[x]

Total order broadcast is asynchronous: messages are guaranteed to be delivered reliably in a fixed order, but there is no guarantee about *when* a message will be delivered (so one recipient may lag behind the others). By contrast, linearizability is a recency guarantee: a read is guaranteed to see the latest value written.

However, if you have total order broadcast, you can build linearizable storage on top of it. For example, you can ensure that usernames uniquely identify user accounts.

Imagine that for every possible username, you can have a linearizable register with an atomic compare-and-set operation. Every register initially has the value `null` (indicating that the username is not taken). When a user wants to create a username, you execute a compare-and-set operation on the register for that username, setting it to the user account ID, under the condition that the previous register value is `null`. If

[x] In a formal sense, a linearizable read/write register is an "easier" problem. Total order broadcast is equivalent to consensus [70], which has no deterministic solution in the asynchronous crash-stop model [73], whereas a linearizable read/write register *can* be implemented in the same system model [28, 29, 30]. However, adding atomic operations such as compare-and-set or increment-and-get to a register make it equivalent to consensus [33]. In practice, the problems of consensus and a linearizable register are rather similar.

multiple users try to concurrently grab the same username, only one of the compare-and-set operations will succeed, because the others will see a value other than `null` (due to linearizability).

You can implement such a linearizable compare-and-set operation as follows, by using total order broadcast as an append-only log [65, 66]:

1. Append a message to the log, tentatively indicating the username you want to claim.
2. Read the log, and wait for the message you appended to be delivered back to you.[xi]
3. Check for any messages claiming the username that you want. If the first message for your desired username is your own message, then you are successful: you can commit the username claim (perhaps by appending another message to the log) and acknowledge it to the client. If the first message for your desired username is from another user, you abort the operation.

Because log entries are delivered to all nodes in the same order, if there are several concurrent writes, all nodes will agree on which one came first. By choosing the first of the conflicting writes as winner, and aborting later ones, all nodes agree whether a write was committed or aborted. A similar approach can be used to implement serializable multi-object transactions on top of a log [65].

While this procedure ensures linearizable writes, it doesn't guarantee linearizable reads — if you read from a store that is asynchronously updated from the log, it may be stale. (To be precise, the procedure above provides *sequential consistency* [50, 67], sometimes also known as *timeline consistency* [68, 69], a slightly weaker guarantee than linearizability.) To make reads linearizable, there are a few options:

- You can sequence reads through the log by appending a message, reading the log, and performing the actual read when the message is delivered back to you. The message's position in the log thus defines the point in time at which the read happens. (Quorum reads in etcd work somewhat like this [16].)
- If the log allows you to fetch the position of the latest log message in a linearizable way, you can query that position, wait for all entries up to that position to be delivered to you, and then perform the read. (This is the idea behind ZooKeeper's `sync()` operation [15].)

[xi] If you don't wait, but acknowledge the write immediately after it has been enqueued, you get something similar to the consistency model of multicore x86 processors [46]. That model is neither linearizable nor sequentially consistent.

- Or you can make your read from a replica that is synchronously updated on writes, and is thus sure to be up-to-date. (This technique is used in chain replication [66].)

Implementing total order broadcast using linearizable storage

The last section showed how to build a linearizable compare-and-set operation from total order broadcast. We can also turn it around, assume that we have linearizable storage, and show how to build total order broadcast from it.

The easiest way is to assume you have a linearizable register that stores an integer and that has an atomic increment-and-get operation [33]. Alternatively, an atomic compare-and-set operation would also do the job.

The algorithm is simple: for every message you want to send through total order broadcast, you increment-and-get the linearizable integer, and then attach the value you got from the register as a sequence number to the message. You can then send the message to all nodes (re-sending any lost messages), and the recipients deliver messages consecutively by sequence number.

Note that unlike Lamport timestamps, the numbers you get from incrementing the linearizable register form a sequence with no gaps. Thus, if a node has delivered message 4 and receives an incoming message with a sequence number of 6, it knows that it must wait for message 5 before it can deliver message 6. The same is not the case with Lamport timestamps — in fact, this is the key difference between total order broadcast and timestamp ordering.

How hard could it be to make a linearizable integer with an atomic increment-and-get operation? As usual, if things never failed, it would be easy: you could just keep it in a variable on one node. The problem lies in handling the situation when network connections to that node are interrupted, and restoring the value when that node fails [62]. In general, if you think hard enough about linearizable sequence number generators, you inevitably end up with a consensus algorithm.

This is no coincidence: it can be proved that a linearizable compare-and-set (or increment-and-get) register and total order broadcast are both *equivalent to consensus* [33, 70]. That is, if you can solve one of these problems, you can transform it into a solution for the others. This is quite a profound and surprising insight!

It is time to finally tackle the consensus problem head-on, which we will do in the rest of this chapter.

Distributed Transactions and Consensus

Consensus is one of the most important and fundamental problems in distributed computing. On the surface, it seems simple: informally, the goal is simply to *get several nodes to agree on something*. You might think that this shouldn't be too hard. Unfortunately, many broken systems have been built in the mistaken belief that this problem is easy.

Although consensus is so important, the section about it appears so late in this book because the topic is quite subtle, and appreciating the subtleties requires some prerequisite knowledge. Even in the academic research community, the understanding of consensus only gradually crystallized over the course of decades, with many misunderstandings along the way. Now that we have discussed replication (Chapter 5), transactions (Chapter 7), system models (Chapter 8), linearizability, and total order broadcast (Chapter 9), we are finally ready to tackle the consensus problem.

There are a number of situations in which it is important for nodes to agree, for example:

Leader election

In a database with single-leader replication, all nodes need to agree which node is the leader. The leadership position might become contested if some nodes can't communicate with others due to a network fault. In this case, consensus is important to avoid a bad failover, resulting in a *split brain* situation in which two nodes both believe to be the leader (see "Handling node outages" on page 150). If there were two leaders, they would both accept writes and their data would diverge, leading to inconsistency and data loss.

Atomic commit

In a database that supports transactions spanning several nodes or partitions, we have the problem that a transaction may fail on some nodes but succeed on others. If we want to maintain transaction atomicity (in the sense of ACID, see "Atomicity" on page 216), we have get all nodes to agree on the outcome of the transaction: either they all abort/rollback (if anything goes wrong) or they all commit (if nothing goes wrong). This instance of consensus is known as the *atomic commit* problem.[xii]

In this section we will first examine the atomic commit problem in more detail. In particular, we will discuss the *2-phase commit* (2PC) algorithm, which is the most

[xii] Atomic commit is formalized slightly differently from consensus: an atomic transaction can commit only if *all* participants vote to commit, and must abort if any participant needs to abort. Consensus is allowed to decide on *any* value that is proposed by one of the participants. However, atomic commit and consensus are reducible to each other [71, 72]. *Non-blocking* atomic commit is harder than consensus — see "Three-phase commit" on page 350.

common way of solving atomic commit, and which is implemented in various databases, messaging systems and application servers. It turns out that 2PC is a kind of consensus algorithm — but not a very good one [71, 72].

By learning from 2PC we will then work our way towards better consensus algorithms, such as those used in ZooKeeper (Zab) and etcd (Raft).

The impossibility of consensus

You may have heard about the FLP result [73] — named after the authors Fischer, Lynch, and Paterson — which proves that there is no algorithm which reliably achieves consensus if there is a risk that a node may crash. In a distributed system, we must assume that nodes may crash, so consensus is impossible. Yet, here we are, discussing algorithms for achieving consensus. What is going on here?

The answer is that the FLP result is proved in a very restrictive theoretical system model, assuming a deterministic algorithm that cannot use any clocks or timeouts. If the algorithm is allowed to use timeouts, or some other way of suspecting nodes as crashed (even if the suspicion is sometimes wrong), then consensus becomes solvable [70]. Even just allowing the algorithm to use random numbers is sufficient to make consensus solvable [74].

Thus, although the FLP result about the impossibility of consensus is of great theoretical importance, distributed systems can usually achieve consensus in practice.

Atomic commit and two-phase commit (2PC)

Recall from Chapter 7 that the purpose of transaction atomicity is to provide simple semantics in the case where something goes wrong in the middle of making several writes. The outcome of a transaction is either a successful *commit* (in which case all of the transaction's writes are made durable), or an *abort* (in which case all of the transaction's writes are rolled back, i.e. undone or discarded).

Atomicity prevents failed transactions from littering the database with half-finished results and half-updated state. This is especially important for multi-object transactions (see "Single-object and multi-object operations" on page 219) and databases that maintain secondary indexes. Each secondary index is a separate data structure from the primary data — thus, if you modify some data, the corresponding change needs to also be made in the secondary index. Atomicity ensures that the secondary index stays consistent with the primary data (if the index became inconsistent with the primary data, it would not be very useful).

From single-node to distributed atomic commit

For transactions that execute at a single database node, atomicity is commonly implemented by the storage engine. When the client asks the database node to commit the transaction, the database makes the transaction's writes durable (typically in a write-ahead log, see "Update-in-place vs. append-only logging" on page 80) and then appends a commit record to the log on disk. If the database crashes in the middle of this process, the transaction is recovered from the log when the node restarts: if the commit record was successfully written to disk before the crash, the transaction is considered committed; if not, any writes from that transaction are rolled back.

Thus, on a single node, transaction commit crucially depends on the *order* in which data is durably written to disk: first the data, then the commit record [75]. The key deciding moment for commit or abort is the moment at which the disk finishes writing the commit record: before that moment, it is still possible to abort (due to a crash), and after that moment, the transaction is committed (even if the database crashes). Thus, it is a single device (the controller of one particular disk drive, attached to one particular node) that makes the commit atomic.

However, what if multiple nodes are involved in a transaction? For example, perhaps you have a multi-object transaction in a partitioned database, or a term-partitioned secondary index (in which the index entry may be on a different node from the primary data, see "Partitioning and secondary indexes" on page 197). Most "NoSQL" distributed datastores do not support such distributed transactions, but various clustered relational systems do (see "Distributed transactions in practice" on page 350).

It is not sufficient to simply send a commit request to all of the nodes, and independently commit the transaction on each one. In doing so, it could easily happen that the commit succeeds on some nodes and fails on other nodes, which would violate the atomicity guarantee:

- some nodes may detect a constraint violation or conflict, making an abort necessary, while other nodes are successfully able to commit;
- some of the commit requests might be lost in the network, eventually aborting due to a timeout, while other commit requests get through;
- some nodes may crash before the commit record is fully written, and roll back on recovery, while others succeed.

If some nodes commit the transaction but others abort it, the nodes become inconsistent with each other (like in Figure 7-3). And once a transaction has been committed on one node, it cannot be retracted again if it later turns out that it was aborted on another node.

Once data has been committed, it becomes visible to other transactions, and thus other clients may start relying on that data. For this reason, a transaction commit must be irrevocable — you are not allowed to change your mind and retroactively abort a transaction after it has been committed. This is the basis of *read committed* isolation — see "Read committed" on page 225. (It is possible for the effects of a committed transaction to later be undone by another, *compensating transaction* [19, 20]. However, from the database's point of view this is a separate transaction, and thus any cross-transaction correctness requirements are the application's problem.)

Introduction to two-phase commit

Two-phase commit (2PC) is an algorithm for achieving atomic transaction commit across multiple nodes, i.e. to ensure that either all nodes commit, or all nodes abort. It is a classic algorithm in distributed databases [13, 39, 76]. 2PC is used internally in some databases, and also made available to applications in the form of *XA transactions* [77, 78] (which are supported by the Java Transaction API, for example), or WS-AtomicTransaction for SOAP web services [79, 80].

Don't confuse 2PC and 2PL

Two-phase *commit* (2PC) and two-phase *locking* (see "Two-phase locking (2PL)" on page 248) are two very different things. 2PC provides atomic commit in a distributed database, 2PL provides serializable isolation. To avoid confusion, it's best to think of them as entirely separate concepts, and to ignore the unfortunate similarity in name.

The basic flow of 2PC is illustrated in Figure 9-9. Instead of a single commit request, as with a single-node transaction, the commit/abort process in 2PC is split into two phases (hence the name).

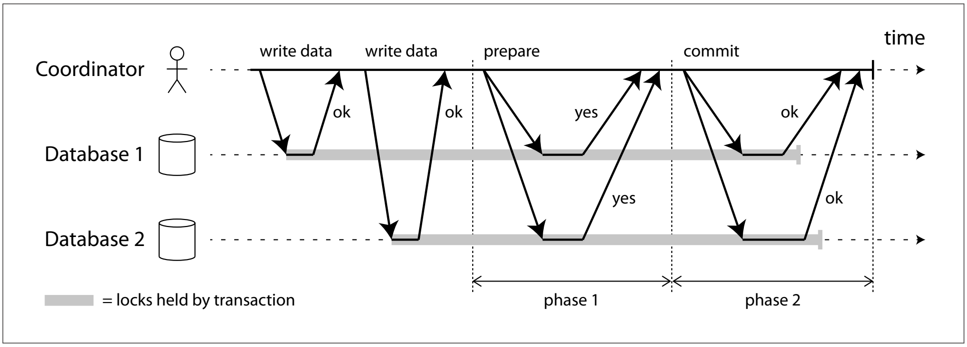


Figure 9-9. A successful execution of two-phase commit (2PC).

Two-phase commit uses a new component that does not normally appear in single-node transactions: a *coordinator* (also known as *transaction manager*). The coordinator is often implemented as a library within the same application process that is requesting the transaction (e.g. embedded in a Java EE container), but it can also be a separate process or service. Examples of such coordinators include Narayana, JOTM, BTM or MSDTC.

A 2PC transaction begins with the application reading and writing data on multiple database nodes, as normal. We call these database nodes *participants* in the transaction. When the application is ready to commit, the coordinator begins phase 1: it sends a *prepare* request to each of the nodes, asking them whether they are able to commit. The coordinator then tracks the responses from the participants:

- If all participants reply "yes", indicating they are ready to commit, then the coordinator sends out a *commit* request in phase 2, and the commit actually takes place.
- If any of the participants replies "no", the coordinator sends an *abort* request to all nodes in phase 2.

This process is somewhat like the traditional marriage ceremony in western cultures: the minister asks the bride and groom individually whether they want to marry the other, and typically receives the answer "I do" from each. After receiving both acknowledgements, the minister pronounces the couple husband and wife: the transaction is committed, and the happy fact is broadcast to all attendees. If either bride or groom does not say yes, the ceremony is aborted [19].

A system of promises

From this short description it might not be clear why two-phase commit ensures atomicity, while one-phase commit across several nodes does not. Surely the *prepare* and *commit* requests can just as easily be lost in the two-phase case. What makes 2PC different?

To understand this, we have to break down the process in a bit more detail:

1. When the application wants to begin a distributed transaction, it requests a transaction ID from the coordinator. This transaction ID is globally unique.
2. The application begins a single-node transaction on each of the participants, and attaches the globally unique transaction ID to the single-node transaction. All reads and writes are done in one of these single-node transactions. If anything goes wrong at this stage (for example, a node crashes or a request times out), the coordinator or any of the participants can abort.
3. When the application is ready to commit, the coordinator sends a *prepare* request to all participants, tagged with the global transaction ID. If any of these requests fails or times out, the coordinator sends an *abort* request for that transaction ID to all participants.

4. When a participant receives the *prepare* request, it makes sure that it can definitely commit the transaction in all circumstances. This includes writing all transaction data to disk (a crash, a power failure or running out of disk space are not acceptable excuses for refusing to commit later), and checking for any conflicts or constraint violations. By replying "yes" to the coordinator, the node promises to commit the transaction without error if requested. In other words, the participant gives up the right to abort the transaction, but without actually committing it.
5. When the coordinator has received responses to all *prepare* requests, it makes the definitive decision whether to commit or abort the transaction (committing only if all participants voted "yes"). The coordinator must write that decision to its transaction log on disk, so that it knows which way it decided in case it subsequently crashes. This is called the *commit point*.
6. Once the coordinator's decision has been written to disk, the *commit* or *abort* request is sent to all participants. If the request fails or times out, the coordinator must retry forever until it succeeds. There is no more going back: if the decision was to commit, that decision must be enforced, no matter how many retries it takes. If a participant has crashed in the meantime, the transaction will be committed when it recovers — since the participant voted "yes", it cannot refuse to commit when it recovers.

Thus, the protocol contains two crucial "points of no return": when a participant votes "yes", it promises that it is definitely able to commit later (although the coordinator may still choose to abort); and once the coordinator decides, that decision is final. Those promises ensure the atomicity of 2PC. (Single-node atomic commit lumps these two events into one: writing the commit record to the transaction log.)

Returning to the marriage analogy: before saying "I do", you and your bride/groom have the freedom to abort the transaction by saying "no way!", or something to that effect. However, after saying "I do", you cannot retract that statement. If you faint after saying "I do", and you don't hear the minister speak the words "you are now husband and wife", that doesn't change the fact that the transaction was committed. When you recover consciousness later, you can find out whether you are married or not by querying the minister for the status of your global transaction ID, or you can wait for the minister's next retry of the *commit* request (since the retries will have continued throughout your period of unconsciousness).

Coordinator failure

We have discussed what happens if one of the participants or the network fails during 2PC: if any of the *prepare* requests fails or times out, the coordinator aborts the transaction; if any of the *commit* or *abort* requests fail, the coordinator retries them indefinitely. However, it is less clear what happens if the coordinator crashes.

If the coordinator fails before sending the *prepare* requests, a participant can safely abort the transaction. But once the participant has received a *prepare* request and voted "yes", it can no longer abort unilaterally — it must wait to hear back from the coordinator whether the transaction was committed or aborted. If the coordinator crashes or the network fails at this point, the participant can do nothing but wait. A participant's transaction in this state is called *in doubt* or *uncertain*.

The situation is illustrated in Figure 9-10. In this particular example, the coordinator actually decided to commit, and database 2 received the commit request. However, the coordinator crashed before it could send the commit request to database 1, and so database 1 does not know whether to commit or abort. Even a timeout does not help here: if database 1 unilaterally aborts after a timeout, it will end up inconsistent with database 2, which has committed. Similarly, it is not safe to unilaterally commit, because another participant may have aborted.

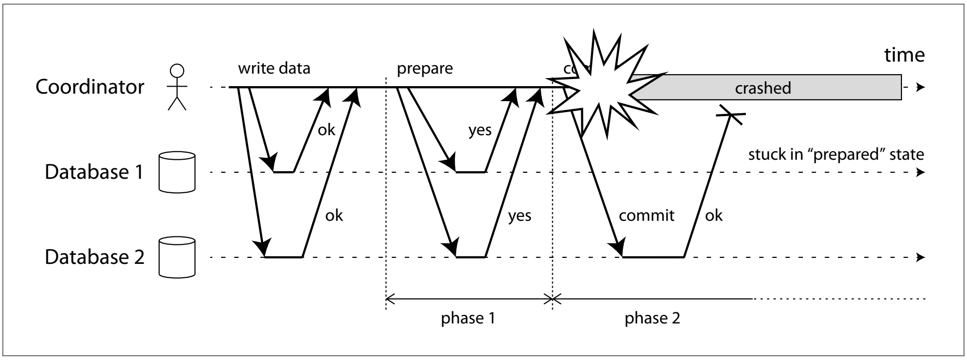


Figure 9-10. The coordinator crashes after participants vote "yes". Database 1 does not know whether to commit or abort.

Without hearing from the coordinator, the participant has no way of knowing whether to commit or abort. In principle, the participants could communicate among themselves to find out how each participant voted and come to some agreement, but that is not part of the 2PC protocol.

The only way how 2PC can complete is by waiting for the coordinator to recover. This is why the coordinator must write its commit or abort decision to a transaction log on disk before sending commit or abort requests to participants: when the coordinator recovers, it determines the status of all in-doubt transactions by reading its transaction log. Any transactions that don't have a commit record in the coordinator's log are aborted. Thus, the commit point of 2PC reduces down to a regular single-node atomic commit on the coordinator.

Three-phase commit

Two-phase commit is called a *blocking* atomic commit protocol due to the fact that 2PC can become stuck waiting for the coordinator to recover. In theory, it is possible to make an atomic commit protocol *non-blocking*, so that it does not get stuck if a node fails. However, making this work in practice is not so straightforward.

As alternative to 2PC, an algorithm called *three-phase commit* (3PC) has been proposed [13, 81]. However, the standard formulation of 3PC assumes a network with bounded delay and nodes with bounded response times; in most practical systems with unbounded network delay and process pauses (see Chapter 8) it cannot guarantee atomicity.

In general, non-blocking atomic commit requires a *perfect failure detector* [70, 72], i.e. a reliable mechanism for telling whether a node is crashed or not. In a network with unbounded delay, a timeout is not a reliable failure detector, because a request may time out due to a network problem even if no node has crashed. For this reason, 2PC continues to be used, despite the known problem of blocking on coordinator failure.

Distributed transactions in practice

Distributed transactions, especially those implemented with two-phase commit, have a mixed reputation. On the one hand, they are seen as providing an important safety guarantee that would be hard to achieve otherwise; on the other hand, they are criticized for causing operational problems, killing performance, and promising more than they can deliver [82, 83, 84, 85]. Many "cloud" services choose not to implement distributed transactions due to the operational problems they cause [86, 87].

Some implementations of distributed transactions carry a heavy performance penalty — for example, distributed transactions in MySQL are reported to be over 10 times slower than single-node transactions [88], so it is not surprising when people advise against using them. Much of the performance cost inherent in 2-phase commit is due to the additional disk forcing (fsync) that is required for crash recovery [89], and the additional network round-trips.

However, rather than dismissing distributed transactions outright, we should examine them in some more detail, because there are important lessons to be learned from them. To begin, we should be precise about what we mean with distributed transactions. Two quite different types of distributed transaction are often conflated:

Database-internal distributed transactions

Some distributed databases (i.e. databases that use replication and partitioning in their standard configuration) support internal transactions among the nodes of that database. For example, VoltDB, FoundationDB, and MySQL Cluster's NDB storage engine have such internal transaction support. In this case, all the nodes participating in the transaction are running the same database software.

Heterogeneous distributed transactions

In a *heterogeneous* transaction, the participants are two or more different technologies: for example, two databases from different vendors, or even non-database systems such as message brokers. A distributed transaction across these systems must ensure atomic commit, even though they may be entirely different under the hood.

Database-internal transactions do not have to be compatible with any other system, and so they can use any protocol and apply optimizations specific to that particular technology. For that reason, database-internal distributed transactions can often work quite well. On the other hand, transactions spanning heterogeneous technologies are a lot more challenging.

Exactly-once message processing

Heterogeneous distributed transactions allow diverse systems to be integrated in powerful ways. For example, a message from a message queue can be acknowledged as processed if and only if the database transaction for processing the message was successfully committed. This is implemented by atomically committing the message acknowledgement and the database writes in a single transaction. With distributed transaction support, this is possible, even if the message broker and the database are two unrelated technologies running on different machines.

If either the message delivery or the database transaction fails, both are aborted, and so the message broker may safely redeliver the message later. Thus, by atomically committing the message and the side-effects of its processing, we can ensure that the message is *effectively* processed exactly once, even if it required a few retries before it succeeded.

Such a distributed transaction is only possible if all systems affected by the transaction are able to use the same atomic commit protocol. For example, say a side-effect of processing a message is to send an email, and the email server does not support two-phase commit. Then it could happen that the email is sent two or more times if message processing fails and is retried. But if all side-effects of processing a message are rolled back on transaction abort, then the processing step can safely be retried as if nothing had happened.

We will return to the topic of exactly-once message processing in Chapter 11. Let's look first at the atomic commit protocol that allows such heterogeneous distributed transactions.

XA Transactions

X/Open XA (short for *eXtended Architecture*) is a standard for implementing two-phase commit across heterogeneous technologies [77, 78]. It was introduced in 1991 and has been widely implemented: XA is supported by many traditional relational databases (including PostgreSQL, MySQL, DB2, SQL Server and Oracle) and message brokers (including ActiveMQ, HornetQ, MSMQ and IBM MQ).

XA is not a network protocol — it is merely a C API for interfacing with a transaction coordinator. Bindings for this API exist in other languages; for example, in the world of Java EE applications, XA transactions are implemented using the Java Transaction API (JTA), which in turn is supported by many drivers for databases using Java Database Connectivity (JDBC), and drivers for message brokers using the Java Message Service (JMS) APIs.

XA assumes that your application uses a network driver or client library to communicate with the participant databases or messaging services. If the driver supports XA, that means it calls the XA API to find out whether an operation should be part of a distributed transaction — and if so, it sends the necessary information to the database server. The driver also exposes callbacks through which the coordinator can ask the participant to prepare, commit or abort.

The transaction coordinator implements the XA API. The standard does not specify how it should be implemented, but in practice the coordinator is often simply a library that is loaded into the same process as the application issuing the transaction (not a separate service). It keeps track of the participants in a transaction, collects responses to *prepare* requests (via callback into the driver), and uses a log on the local disk to keep track of the commit/abort outcome of each transaction.

If the application process crashes, or the machine on which the application is running dies, the coordinator goes with it. Any participants with prepared but uncommitted transactions are then stuck in doubt. Since the coordinator's log is on the application server's local disk, that server must be restarted, and the coordinator library must read the log to recover the commit/abort outcome of each transaction. Only then can the coordinator use the database driver's XA callbacks to ask participants to commit or abort, as appropriate. The database server cannot contact the coordinator directly, since all communication must go via its client library.

Holding locks while in doubt

Why do we care so much about a transaction being stuck in doubt? Can't the rest of the system just get on with its work, and ignore the in-doubt transaction that will be cleaned up eventually?

The problem is with *locking*. As discussed in "Read committed" on page 225, database transactions usually take a row-level exclusive lock on any rows they modify, to prevent dirty writes. In addition, if you want serializable isolation, a database using 2-phase locking would also have to take a shared lock on any rows *read* by the transaction (see "Two-phase locking (2PL)" on page 248).

The database cannot release those locks until the transaction commits or aborts (illustrated as a shaded area in Figure 9-9). Therefore, when using 2-phase commit, a transaction must hold onto the locks throughout the time is in doubt. If the coordinator has crashed and takes 20 minutes to start up again, those locks will be held for 20 minutes. If the coordinator's log is entirely lost for some reason, those locks will be held forever — at least until the situation is manually resolved by an administrator.

While those locks are held, no other transaction can write those rows. Depending on the database, perhaps other transactions are even blocked from reading those rows. Thus, other transactions cannot simply continue with their business — if they want to access that same data, they will be blocked. This can cause large parts of your application to become unavailable until the in-doubt transaction is resolved.

Recovering from coordinator failure

In theory, if the coordinator crashes and is restarted, it should cleanly recover its state from the log and resolve any in-doubt transactions. However, in practice, *orphaned* in-doubt transactions do occur [90, 91] — that is, transactions for which the coordinator cannot decide the outcome for whatever reason, e.g. because the transaction log is lost or corrupted due to a software bug. These transactions cannot be resolved automatically, so they sit forever in the database, holding locks and blocking other transactions.

Even rebooting your database servers would not fix this problem, since a correct implementation of 2PC must preserve the locks of an in-doubt transaction even across reboots (otherwise it would risk violating the atomicity guarantee). It's a sticky situation.

The only way out is for an administrator to manually decide whether to commit or rollback the transactions. The administrator must examine the participants of each in-doubt transaction, determine whether any participant has committed or aborted already, and then apply the same outcome to the other participants. This is potentially a lot of manual effort, and most likely needs to be done under high stress and time pressure during a serious production outage (why otherwise would the coordinator be in such a bad state?).

Many XA implementations have an emergency escape hatch called *heuristic decisions*: allowing a participant to unilaterally decide to abort or commit an in-doubt transaction without a definitive decision from the coordinator [77, 78, 92]. To be clear, *heuristic* here is a euphemism for *probably breaking atomicity*, since it violates the system of promises in 2-phase commit. Thus, heuristic decisions are intended only for getting out of catastrophic situations, and not for regular use.

Limitations of distributed transactions

XA Transactions solve the real and important problem of keeping several participant data systems consistent with each other, but as we have seen, they also introduce major operational problems. In particular, the key realization is that the transaction coordinator is itself a kind of database (in which transaction outcomes are stored), and so it needs to be approached with the same care as any other important database.

- If the coordinator is not replicated, but runs only on a single machine, it is a single point of failure for the entire system (since its failure causes other application servers to block on locks held by in-doubt transactions). Surprisingly, many coordinator implementations are not highly available by default, or have only rudimentary replication support.
- Many server-side applications are developed in a stateless model (as favored by HTTP), with all persistent state stored in a database, which has the advantage that application servers can be added and removed at will. However, when the coordinator is part of the application server, it changes the nature of the deployment: suddenly the coordinator's logs become a crucial part of the durable system state — as important as the databases themselves, since the coordinator logs are required in order to recover in-doubt transactions after a crash. Such application servers are no longer stateless.
- Since XA needs to be compatible with a wide range of data systems, it is necessarily a lowest common denominator. For example, it cannot detect deadlocks across different systems (since that would require a standardized protocol for systems to exchange information on the locks that each transaction is waiting for), and it does not work with Serializable Snapshot Isolation (see "Serializable snapshot isolation (SSI)" on page 252, since that would require a protocol for identifying conflicts across different systems).
- For database-internal distributed transactions (not XA), the limitations are not so great — for example, a distributed version of SSI is possible. However, there remains the problem that for 2PC to successfully commit a transaction, *all* participants must respond. Consequently, if *any* part of the system is broken, the transaction also fails. Distributed transactions thus have a tendency of *amplifying failures*, which runs counter to our goal of building fault tolerant systems.

Do these facts mean we should give up all hope of keeping several systems consistent with each other? Not quite — there are alternative methods that allow us to achieve the same thing without the pain of heterogeneous distributed transactions. We will return to these in Chapter 11 and ???. But first we should wrap up the topic of consensus.

Fault-tolerant consensus

Informally, consensus means getting several nodes to agree on something. For example, if several people concurrently try to book the last seat on an airplane, or the same seat in a theater, or try to register an account with the same username, then a consensus algorithm could be used to determine which one of these mutually incompatible operations should be the winner.

The consensus problem is normally formalized as follows: one or more nodes may *propose* values, and the consensus algorithm *decides* on one of those values. In the seat-booking example, when several customers are concurrently trying to buy the last seat, each node handling a customer request may propose the ID of the customer it is serving, and the decision indicates which one of those customers got the seat.

In this formalism, a consensus algorithm must satisfy the following properties [30]:[xiii]

Uniform agreement
    No two nodes decide differently.

Integrity
    No node decides twice.

Validity
    If a node decides value *v*, then *v* was proposed by some node.

Termination
    Every node that does not crash eventually decides some value.

The *uniform agreement* and *integrity* properties define the core idea of consensus: everyone decides on the same outcome, and once you have decided, you cannot change your mind. The *validity* property exists mostly to rule out trivial solutions: for example, you could have an algorithm that always decides `null`, no matter what was proposed — it would satisfy the agreement and integrity properties, but not the validity property.

If you don't care about fault tolerance, then satisfying the first three properties is easy: you can just hard-code one node to be the "dictator", and let that node make all of the decisions. However, if that one node fails, then the system can no longer make any decisions. This is, in fact, what we saw in the case of 2-phase commit: if the coordinator fails, in-doubt participants cannot decide whether to commit or abort.

[xiii] This particular variant of consensus is called *uniform consensus*, which is equivalent to regular consensus in asynchronous systems with unreliable failure detectors [72]. The academic literature usually refers to *processes* rather than *nodes*, but we use *nodes* here for consistency with the rest of this book.

The *termination* property formalizes the idea of fault tolerance. It essentially says that a consensus algorithm cannot simply sit around and do nothing forever — in other words, it must make progress. Even if some nodes fail, the other nodes must still reach a decision. (Termination is a liveness property, whereas the other three are safety properties — see "Safety and liveness" on page 300.)

The system model of consensus assumes that when a node "crashes", it suddenly disappears and never comes back. (Instead of a software crash, imagine that there is an earthquake, and the datacenter containing your node is destroyed by a landslide. You must assume that your node is buried under 30 feet of mud and is never going to come back online.) In this system model, any algorithm that has to wait for a node to recover is not going to be able to satisfy the termination property. In particular, 2PC does not meet the requirements for termination.

Of course, if *all* nodes crash and none of them are running, then it is not possible for any algorithm to decide anything. There is a limit to the number of failures that an algorithm can tolerate: in fact, it can be proved that any consensus algorithm requires at least a majority of nodes to be functioning correctly in order to assure termination [70].

Thus, the termination property is subject to the assumption that fewer than half of the nodes are crashed or unreachable. However, most implementations of consensus ensure that the safety properties — agreement, integrity and validity — are always met, even if a majority of nodes fails or there is a severe network problem [93]. Thus, a large-scale outage can stop the system from being able to process requests, but it cannot corrupt the consensus system by causing it to make invalid decisions.

Most consensus algorithms assume that there are no Byzantine faults, as discussed in "Byzantine faults" on page 295. That is, if a node does not correctly follow the protocol (for example, if it sends contradictory messages to different nodes), it may break the safety properties of the protocol. It is possible to make consensus robust against Byzantine faults as long as fewer than one third of the nodes are Byzantine-faulty [30, 94], but we don't discuss those algorithms here as they are rarely used in practice.

Consensus algorithms and total order broadcast

The best-known fault-tolerant consensus algorithms are Viewstamped Replication (VSR) [95, 96], Paxos [97, 98, 99, 100], Raft [27, 101, 102] and Zab [15, 26, 103]. There are quite a few similarities between these algorithms, but they are not the same [104]. In this book we won't go into full details of the algorithms: it's sufficient to be aware of some of the high-level ideas that these algorithms have in common, unless you're implementing a consensus system yourself (which is probably not advisable — it's hard [99, 105]).

Most of these algorithms actually don't directly use the formal model described above (proposing and deciding on a single value, while satisfying the agreement, integrity, validity and termination properties). Instead, they decide on a *sequence* of values, which makes them *total order broadcast* algorithms, as discussed previously in this chapter (see "Total order broadcast" on page 338).

Remember that total order broadcast requires messages to be delivered exactly once, in the same order, to all nodes. If you think about it, this is equivalent to performing several rounds of consensus: in each round, nodes propose the message that they want to send next, and then decide on the next message to be delivered in the total order [70].

Thus, total order broadcast is equivalent to repeated rounds of consensus (each consensus decision corresponding to one message delivery):

- Due to the agreement property of consensus, all nodes decide to deliver the same messages in the same order.
- Due to the integrity property, messages are not duplicated.
- Due to the validity property, messages are not corrupted and not fabricated out of thin air.
- Due to the termination property, messages are not lost.

Viewstamped Replication, Raft and Zab implement total order broadcast directly, because that is more efficient than doing repeated rounds of one-value-at-a-time consensus. In the case of Paxos, this optimization is known as Multi-Paxos.

Single-master replication and consensus

In Chapter 5 we discussed single-leader replication (see "Leaders and Followers" on page 146), which takes all the writes to the leader and applies them to the followers in the same order, thus keeping replicas up-to-date. Isn't this essentially total order broadcast? How come we didn't have to worry about consensus in Chapter 5?

The answer comes down to how the leader is chosen. If the leader is manually chosen and configured by the humans in your operations team, you essentially have a 'consensus algorithm' of the dictatorial variety: only one node is allowed to accept writes (i.e. make decisions about the order of writes in the replication log), and if that node goes down, the system becomes unavailable for writes until the operators manually configure a different node to be the leader. Such a system can work well in practice, but it does not satisfy the termination property of consensus because it requires human intervention in order to make progress.

Some databases perform automatic leader election and *failover*, promoting a follower to be the new leader if the old leader fails (see "Handling node outages" on page 150). This brings us closer to fault-tolerant total order broadcast, and thus solving consensus.

However, there is a problem: we previously discussed the problem of *split brain*, and said that all nodes need to agree who the leader is — otherwise two different nodes could each believe to be the leader, and consequently get the database into an inconsistent state. Thus, we need consensus in order to elect a leader. But if the consensus algorithms above are actually total order broadcast algorithms, and total order broadcast is like single-leader replication, and single-leader replication requires a leader, then…

It seems that in order to elect a leader, we first need a leader. In order to solve consensus, we must first solve consensus. How do we break out of this conundrum?

Epoch numbering and quorums

All of the above consensus protocols internally use a leader in some form or another, but they don't guarantee that the leader is unique. Instead, they can make a weaker guarantee: the protocols define an *epoch number* (called *ballot number* in Paxos, *view number* in Viewstamped Replication, and *term number* in Raft), and guarantee that within each epoch, the leader is unique.

Every time the current leader is thought to be dead, a vote is started among the nodes to elect a new leader. This election is given an incremented epoch number, and thus epoch numbers are totally ordered and monotonically increasing. If there is a conflict between two different leaders in two different epochs (perhaps because the previous leader actually wasn't dead after all), then the leader with the higher epoch number prevails.

Before a leader is allowed to decide anything, it must first check that there isn't some other leader with a higher epoch number which might take a conflicting decision. How does a leader know that it hasn't been ousted by another node? Recall "The truth is defined by the majority" on page 292: a node cannot necessarily trust its own judgment — just because a node thinks that it is leader does not mean that other nodes accept it as leader — so it must rely on votes from a majority.

This is done using a *quorum* of nodes (see "Quorums for reading and writing" on page 173). For every decision that a leader wants to make, it must send the proposed value to the other nodes, and wait for a majority of nodes to respond in favor of the proposal. A node votes in favor of a proposal only if it is not aware of any other leader with a higher epoch.

Since a node requires a majority of votes to become leader, and a proposal requires a majority to be decided, we can be sure that at least one of the nodes voting on a proposal will have seen a leader election if one has happened. Therefore, if a majority of nodes are in favor of the proposal, the leader can be sure that it still holds the leadership, and therefore it can safely decide the proposed value.

This voting process looks superficially similar to 2-Phase Commit. The biggest difference is that fault-tolerant consensus algorithms only require votes from a majority of nodes, whereas 2PC requires a *yes* vote from *every* participant. Moreover, consensus algorithms define a recovery process by which nodes can get into a consistent state after a new leader is elected, ensuring that the safety properties are always met. These differences are key to the fault tolerance of a consensus algorithm.

Limitations of consensus

Consensus algorithms are a huge breakthrough for distributed systems: they bring concrete safety properties (agreement, integrity and validity) to systems where everything else is uncertain, and they nevertheless remain fault-tolerant (able to make progress as long as a majority of nodes is working and reachable). They provide total order broadcast, and therefore they can also implement linearizable atomic operations in a fault-tolerant way (see "Implementing linearizable storage using total order broadcast" on page 340).

Nevertheless, they are not used everywhere, because the benefits come at a cost.

The process by which nodes vote on proposals before they are decided is a kind of synchronous replication. As discussed in "Synchronous vs. asynchronous replication" on page 147, databases are often configured to use asynchronous replication. In this configuration, some committed data can potentially be lost on failover — but many people choose to accept this risk for the sake of better performance.

Consensus systems always require a strict majority to operate. This means you need a minimum of three nodes in order to tolerate one failure (the remaining two out of three form a majority), or a minimum of five nodes to tolerate two failures (the remaining three out of five form a majority). If a network failure cuts off some nodes from the rest, only the majority portion of the network can make progress, and the rest is blocked (see also "The cost of linearizability" on page 326).

Most consensus algorithms assume a fixed set of nodes that participate in voting, which means that you can't just add or remove nodes in the cluster. *Dynamic membership* extensions to consensus algorithms allow the set of nodes in the cluster to change over time, but they are much less well understood than static-membership algorithms.

Consensus systems generally rely on timeouts to detect failed nodes. In environments with highly variable network delays, especially geographically distributed systems, it often happens that a node falsely believes the leader to have failed due to a transient network issue. Although this does not harm the safety properties, frequent leader elections result in terrible performance, because the system can end up spending more time choosing a leader than doing any useful work.

Sometimes, consensus algorithms are particularly sensitive to network problems. For example, Raft has been shown to have unpleasant edge cases [106]: if the entire network is working correctly, except for one particular network link that is consistently unreliable, Raft can get into situations where leadership continually bounces between two nodes, or the current leader is continually forced to resign, so the system effectively never makes progress. Other consensus algorithms have similar problems, and designing algorithms that are more robust to unreliable networks is still an open research problem.

Membership and coordination services

If you look at projects like ZooKeeper or etcd, you see them described as "distributed key-value stores" or "coordination and configuration services". The API looks pretty much like that of a database: you can read and write the value for a given key, and iterate over keys. So if they're basically databases, why do they go to all this effort of implementing a consensus algorithm? What makes them different from any other kind of database?

To understand this, it is helpful to briefly explore how a service like ZooKeeper is used. As an application developer, you rarely need to use ZooKeeper directly, because it is actually not well suited as a general-purpose database. It is more likely that you end up relying on it indirectly via some other project: for example, HBase, Hadoop YARN, OpenStack Nova and Kafka all rely on ZooKeeper running in the background. What is it that these projects get from it?

ZooKeeper and etcd are designed to hold small amounts of data that can fit entirely in memory (although they still write to disk for durability) — so you wouldn't want to store all of your application's data here. That small amount of data is replicated across all the nodes using a fault-tolerant total order broadcast algorithm. As discussed previously, total order broadcast is just what you need for database replication: if each message represents a write to the database, applying the same writes in the same order keeps replicas consistent with each other.

ZooKeeper is modeled after Google's Chubby lock service [14, 99], implementing not only total order broadcast (and hence consensus), but also an interesting set of other features that turns out to be particularly useful when building distributed systems:

Linearizable atomic operations

Using an atomic compare-and-set operation you can implement a lock: if several nodes concurrently try to perform the same operation, only one of them will succeed. By using consensus, the operation is guaranteed to be atomic and linearizable, even if a node fails or the network is interrupted at any point. A distributed lock is usually implemented as a *lease*, which has an expiry time so that it is eventually released in case the client fails (see "Process pauses" on page 287).

Total ordering of operations

As discussed in "The leader and the lock" on page 293, when some resource is protected by a lock or lease, you need a *fencing token* to prevent clients from conflicting with each other in the case of a process pause. The fencing token is some number that monotonically increases every time the lock is acquired. ZooKeeper provides this by totally ordering all operations, and giving each operation a monotonically increasing transaction ID `zxid` and version number `cversion` [15].

Failure detection

Clients maintain a long-lived session on ZooKeeper servers, and periodically exchange heartbeats to check that the other node is still alive. Even if the connection is temporarily interrupted, or a ZooKeeper node fails, the session remains active. However, if the heartbeats cease for a duration that is longer than the session timeout, ZooKeeper declares the session to be dead. Any locks held by a session can be configured to be automatically deleted when the session times out (ZooKeeper calls these *ephemeral nodes*).

Event notifications

Not only can one client read locks and values that were created by another client, but it can also watch them for changes. Thus, a client can find out when another client joins the cluster (based on the value it writes to ZooKeeper), or if another client fails (because its session times out and its ephemeral nodes disappear). By subscribing to notifications, a client avoids having to frequently poll to find out about changes.

Of these features, only the linearizable atomic operations really require consensus. However, it is the combination of these features that makes systems like ZooKeeper so useful for distributed coordination.

Allocating work to nodes

One example in which the ZooKeeper/Chubby model works well is if you have several instances of a process or service, and one of them needs to be chosen as leader or primary. If the leader fails, one of the other nodes should take over. This is of course useful for single-leader databases, but also for job schedulers and similar stateful systems.

Another example is where you have some partitioned resource (database, message streams, file storage, distributed actor system, etc.) and need to decide which partition to assign to which node. As new nodes join the cluster, some of the partitions need to be moved from existing nodes to the new nodes, in order to rebalance the load (see "Rebalancing partitions" on page 201). As nodes are removed or fail, other nodes need to take over the failed nodes' work.

These kinds of task can be achieved by judicious use of atomic operations, ephemeral nodes and notifications in ZooKeeper. If done correctly, this allows the application to automatically recover from faults without human intervention. It is still not easy — libraries such as Apache Curator [17] have sprung up to provide higher-level tools on top of the ZooKeeper client API. But it is still much better than attempting to implement the necessary consensus algorithms from scratch, which has a poor success record [107].

An application may initially run only on a single node, but eventually may grow to thousands of nodes. Trying to perform majority votes over so many nodes would be terribly inefficient. Instead, ZooKeeper runs on a fixed number of nodes (usually 3 or 5) and performs its majority votes among those nodes while supporting a potentially large number of clients. Thus, ZooKeeper provides a way of 'outsourcing' some of the work of coordinating nodes (consensus, operation ordering and failure detection) to an external service.

Normally, the kind of data managed by ZooKeeper is quite slow-changing: it represents information like "the node running on 10.1.1.23 is the leader for partition 7", which may change on a timescale of minutes or hours. It is not intended for storing the runtime state of the application, which may change thousands or even millions of times per second. If application state needs to be replicated from one node to another, other tools such as Apache BookKeeper can be used [108].

Service discovery

ZooKeeper, etcd and Consul are also often used for *service discovery*, that is, to find out which IP address you need to connect to in order to reach a particular service. In 'cloud' datacenter environments, where it is common for virtual machines to continually come and go, you often don't know the IP addresses of your services ahead of time. Instead, you can configure your services such that when they start up, they register their network endpoint in a service registry, where they can then be found by other services.

However, it is less clear whether service discovery actually requires consensus. DNS is the traditional way of looking up the IP address for a service name, and it uses multiple layers of caching to achieve good performance and availability. Reads from DNS are absolutely not linearizable, and it is usually not considered problematic if the results are a little stale [109]. It is more important that DNS is reliably available and robust to network interruptions.

Although service discovery does not require consensus, leader election does. Thus, if your consensus system already knows who the leader is, then it can make sense to also use that information to help other services discover who the leader is. For this purpose, some consensus systems support read-only caching replicas. These replicas asynchronously receive the log of all decisions of the consensus algorithm, but do not actively participate in voting. They are therefore able to serve read requests that do not need to be linearizable.

Membership services

ZooKeeper and friends can be seen as part of a long history of research into *membership services*, which goes back to the 1980s and has been important for building highly reliable systems, e.g. for air traffic control [110].

A membership service determines which nodes are currently active and live members of a cluster. As we saw throughout Chapter 8, due to unbounded network delays it's not possible to reliably detect whether another node has failed. However, if you couple failure detection with consensus, nodes can come to an agreement about which node is alive and which is not.

It could still happen that a node is incorrectly declared dead by consensus, even though it is actually alive. But it is nevertheless very useful for a system to have agreement on which nodes constitute the current membership. For example, choosing a leader could mean simply choosing the lowest-numbered among the current members. This is not possible if different nodes have divergent opinions on who the current members are.

Summary

In this chapter we examined the topics of consistency and consensus from several different angles. We looked in depth at linearizability, a popular consistency model: its goal is to make replicated data appear as though there was only a single copy, and to make all operations act on it atomically. Although linearizability is appealing because it is easy to understand — it makes a database behave like a variable in a single-threaded program — it has the downside of being slow, especially in environments with large network delays.

We also explored causality, which imposes an ordering on events in a system (what happened before what, based on cause and effect). Unlike linearizability, which puts all operations in a single, totally ordered timeline, causality provides us with a weaker consistency model: some things can be concurrent, so the version history is like a timeline with branching and merging. Causal consistency does not have the coordination overhead of linearizability, and is much less sensitive to network problems.

However, even if we capture the causal ordering (for example using Lamport timestamps), we saw that some things cannot be implemented this way: in "Timestamp ordering is not sufficient" on page 337 we considered the example of ensuring that a username is unique, and rejecting concurrent registrations for the same username. If one node is going to accept a registration, it needs to somehow know that another node isn't concurrently in the process of registering the same name. This problem led us towards *consensus*.

We saw that achieving consensus means to *decide* something in a way that all nodes agree what was decided, and such that the decision is irrevocable. With some digging, it turns out that a wide range of problems are actually reducible to consensus, and equivalent to each other (in the sense that if you have a solution for one of them, you can easily transform it into a solution for one of the others):

Linearizable compare-and-set register
: The register needs to atomically *decide* whether to set its value, based on whether its current value equals the parameter given in the operation.

Atomic transaction commit
: A database must *decide* whether to commit or abort a distributed transaction.

Total order broadcast
: The messaging system must *decide* on the order in which to deliver messages.

Locks and leases
: When several clients are racing to grab a lock or lease, the lock *decides* which one successfully acquired it.

Membership/coordination service
: Given a failure detector (e.g. timeouts), the system must *decide* which nodes are alive, and which should be considered dead because their session timed out.

Uniqueness constraint
: When several transactions concurrently try to create conflicting records with the same key, the constraint must *decide* which one to allow and which should fail with a constraint violation.

All of these are straightforward if you only have a single node, or if you are willing to assign the decision-making capability to a single node. This is what happens in a single-leader database: all the power to make decisions is vested in the leader, which is why such databases are able to provide linearizable operations, uniqueness constraints, a totally ordered replication log, and more.

However, if that single leader fails, or if a network interruption makes the leader unreachable, such a system becomes unable to make any progress. There are three ways of handling that situation:

1. Wait for the leader to recover, and accept that the system will be blocked in the meantime. Many XA/JTA transaction coordinators choose this option. This approach does not solve consensus because it does not satisfy the termination property: if the leader does not recover, the system can be blocked forever.

2. Manually failover by having humans choose a new leader node, and reconfiguring the system to use it. Many relational databases take this approach. It is a kind of consensus by "act of God" — the human operator, outside of the computer system, makes the decision. The speed of failover is limited by the speed at which humans can act, which is generally slower than computers.
3. Use an algorithm to automatically choose a new leader. This requires a consensus algorithm; any system that performs automatic failover without using a proven consensus algorithm is likely to behave badly in adverse network conditions [107].

Although a single-leader database can provide linearizability without executing a consensus algorithm on every write, it still requires consensus to maintain its leadership and for leadership changes. Thus, in some sense, having a leader only "kicks the can down the road": consensus is still required, only in a different place, and less frequently. The good news is that fault-tolerant algorithms and systems for consensus exist, and we briefly discussed them in this chapter.

Tools like ZooKeeper play an important role in providing an "outsourced" consensus, failure detection and membership service that applications can use. It's not easy to use, but it is much better than trying to develop your own algorithms that can withstand all the problems discussed in Chapter 8. If you find yourself wanting to do one of those things that is reducible to consensus, and you want it to be fault-tolerant, then it is advisable to use something like ZooKeeper.

Nevertheless, not every system necessarily requires consensus: for example, leaderless and multi-leader replication systems typically do not use global consensus. The conflicts that occur in these systems (see "Handling write conflicts" on page 164) are a consequence of not having consensus across different leaders. But maybe that's ok: maybe we simply need to cope without linearizability, and learn to work better with data that has branching and merging version histories.

This chapter referenced a large body of research on the theory of distributed systems. Although the theoretical papers and proofs are not always easy to understand, and sometimes make unrealistic assumptions, they are incredibly valuable for informing practical work in this field: they help us reason about what can and cannot be done, and help us find the counter-intuitive ways in which distributed systems are often flawed. If you have the time, the references are well worth exploring.

This brings us to the end of Part II of this book, in which we covered replication (Chapter 5), partitioning (Chapter 6), transactions (Chapter 7), distributed system failure models (Chapter 8), and finally consistency and consensus (Chapter 9). Now that we have laid a firm foundation of theory, in Part III we will turn once again to more practical systems, and discuss how to build powerful applications from heterogeneous building blocks.

References

[1] Peter Bailis and Ali Ghodsi: "Eventual Consistency Today: Limitations, Extensions, and Beyond," *ACM Queue*, volume 11, number 3, March 2013. doi: 10.1145/2460276.2462076

[2] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin: "Consistency, Availability, and Convergence," University of Texas at Austin, Department of Computer Science, Tech Report UTCS TR-11-22, May 2011.

[3] Alex Scotti: "Adventures in building your own database," at *All Your Base*, November 2015.

[4] Peter Bailis, Aaron Davidson, Alan Fekete, et al.: "Highly Available Transactions: Virtues and Limitations," at *40th International Conference on Very Large Data Bases* (VLDB), September 2014. Extended version published as pre-print arXiv:1302.0309 [cs.DB].

[5] Paolo Viotti and Marko Vukolić: "Consistency in Non-Transactional Distributed Storage Systems," *arXiv:1512.00168*, 12 April 2016.

[6] Maurice P Herlihy and Jeannette M Wing: "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems* (TOPLAS), volume 12, number 3, pages 463–492, July 1990. doi: 10.1145/78969.78972

[7] Leslie Lamport: "On interprocess communication," *Distributed Computing*, volume 1, number 2, pages 77–101, June 1986. doi:10.1007/BF01786228

[8] David K Gifford: "Information Storage in a Decentralized Computer System," Xerox Palo Alto Research Centers, CSL-81-8, June 1981.

[9] Martin Kleppmann: "Please stop calling databases CP or AP," martin.kleppmann.com, 11 May 2015.

[10] Kyle Kingsbury: "Call me maybe: MongoDB stale reads," aphyr.com, 20 April 2015.

[11] Kyle Kingsbury: "Computational techniques in Knossos," aphyr.com, 17 May 2014.

[12] Peter Bailis: "Linearizability versus Serializability," bailis.org, 24 September 2014.

[13] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman: *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. Available online at research.microsoft.com. ISBN: 0201107155

[14] Mike Burrows: "The Chubby lock service for loosely-coupled distributed systems," at *7th USENIX Symposium on Operating System Design and Implementation* (OSDI), November 2006.

[15] Flavio P Junqueira and Benjamin Reed: *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, November 2013. ISBN: 978-1-4493-6130-3

[16] "etcd 2.0.12 Documentation," CoreOS, Inc., 2015.

[17] "Apache Curator," Apache Software Foundation, curator.apache.org, 2015.

[18] Morali Vallath: *Oracle 10g RAC Grid, Services & Clustering*. Elsevier Digital Press, May 2006. ISBN: 978-1-55558-321-7

[19] Jim Gray: "The Transaction Concept: Virtues and Limitations," at *7th International Conference on Very Large Data Bases* (VLDB), pages 144–154, September 1981.

[20] Hector Garcia-Molina and Kenneth Salem: "Sagas," at *ACM International Conference on Management of Data* (SIGMOD), pages 249–259, May 1987. doi:10.1145/38713.38742

[21] Pat Helland: "Memories, Guesses, and Apologies," blogs.msdn.com, 15 May 2007.

[22] Susan B Davidson, Hector Garcia-Molina, and Dale Skeen: "Consistency in Partitioned Networks," *ACM Computing Surveys*, volume 17, number 3, pages 341–370, September 1985. doi:10.1145/5505.5508

[23] Eric A Brewer: "NoSQL: Past, Present, Future," at *QCon San Francisco*, November 2012.

[24] Peter Bailis, Alan Fekete, Michael J Franklin, et al.: "Coordination-Avoiding Database Systems," *Proceedings of the VLDB Endowment*, volume 8, number 3, pages 185–196, November 2014.

[25] Kyle Kingsbury: "Call me maybe: etcd and Consul," aphyr.com, 9 June 2014.

[26] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini: "Zab: High-performance broadcast for primary-backup systems," at *41st IEEE International Conference on Dependable Systems and Networks* (DSN), pages 245–256, June 2011. doi:10.1109/DSN.2011.5958223

[27] Diego Ongaro and John K Ousterhout: "In Search of an Understandable Consensus Algorithm (Extended Version)," at *USENIX Annual Technical Conference* (ATC), June 2014.

[28] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev: "Sharing memory robustly in message-passing systems," *Journal of the ACM*, volume 42, number 1, pages 124–142, January 1995. doi:10.1145/200836.200869

[29] Nancy Lynch and Alex Shvartsman: "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts," at *27th Annual International Symposium on Fault-Tolerant Computing* (FTCS), pages 272–281, June 1997. doi:10.1109/FTCS.1997.614100

[30] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues: *Introduction to Reliable and Secure Distributed Programming*, Second Edition. Springer, February 2011. ISBN: 978-3-642-15259-7, doi:10.1007/978-3-642-15260-3

[31] Sam Elliott, Mark Allen, and Martin Kleppmann: personal communication, thread on twitter.com, 15 October 2015.

[32] Niklas Ekström, Mikhail Panchenko, and Jonathan Ellis: "Possible issue with read repair?," email thread on cassandra-dev mailing list, October 2012.

[33] Maurice P Herlihy: "Wait-Free Synchronization," *ACM Transactions on Programming Languages and Systems* (TOPLAS), volume 13, number 1, pages 124–149, January 1991. doi:10.1145/114005.102808

[34] Armando Fox and Eric A Brewer: "Harvest, Yield, and Scalable Tolerant Systems," at *7th Workshop on Hot Topics in Operating Systems* (HotOS), pages 174–178, March 1999. doi:10.1109/HOTOS.1999.798396

[35] Seth Gilbert and Nancy Lynch: "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," *ACM SIGACT News*, volume 33, number 2, pages 51–59, 2002. doi:10.1145/564585.564601

[36] Seth Gilbert and Nancy Lynch: "Perspectives on the CAP Theorem," *IEEE Computer Magazine*, volume 45, number 2, pages 30–36, February 2012. doi:10.1109/MC.2011.389

[37] Eric A Brewer: "CAP Twelve Years Later: How the "Rules" Have Changed," *IEEE Computer Magazine*, volume 45, number 2, pages 23–29, February 2012. doi:10.1109/MC.2012.37

[38] Paul R Johnson and Robert H Thomas: "RFC 677: The Maintenance of Duplicate Databases," Network Working Group, 27 January 1975.

[39] Bruce G Lindsay, Patricia Griffiths Selinger, C Galtieri, et al.: "Notes on Distributed Databases," IBM Research, Research Report RJ2571(33471), July 1979.

[40] Michael J Fischer and Alan Michael: "Sacrificing serializability to attain high availability of data in an unreliable network," at *1st ACM Symposium on Principles of Database Systems* (PODS), pages 70–75, March 1982. doi:10.1145/588111.588124

[41] Martin Kleppmann: "A Critique of the CAP Theorem," arXiv:1509.05393, 17 September 2015.

[42] Nancy A Lynch: "A Hundred Impossibility Proofs for Distributed Computing," at *8th ACM Symposium on Principles of Distributed Computing* (PODC), pages 1–28, August 1989. doi:10.1145/72981.72982

[43] Hagit Attiya, Faith Ellen, and Adam Morrison: "Limitations of Highly-Available Eventually-Consistent Data Stores," at *ACM Symposium on Principles of Distributed Computing* (PODC), July 2015. doi:10.1145/2767386.2767419

[44] Henry Robinson: "CAP Confusion: Problems with "partition tolerance"," blog.cloudera.com, 26 April 2010.

[45] Adrian Cockcroft: "Migrating to Microservices," at *QCon London*, March 2014.

[46] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen: "x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors," *Communications of the ACM*, volume 53, number 7, pages 89–97, July 2010. doi:10.1145/1785414.1785443

[47] Martin Thompson: "Memory Barriers/Fences," mechanical-sympathy.blogspot.co.uk, 24 July 2011.

[48] Ulrich Drepper: "What Every Programmer Should Know About Memory," akkadia.org, 21 November 2007.

[49] Daniel J Abadi: "Consistency Tradeoffs in Modern Distributed Database System Design," *IEEE Computer Magazine*, volume 45, number 2, pages 37–42, February 2012. doi:10.1109/MC.2012.33

[50] Hagit Attiya and Jennifer L Welch: "Sequential Consistency versus Linearizability," *ACM Transactions on Computer Systems* (TOCS), volume 12, number 2, pages 91–122, May 1994. doi:10.1145/176575.176576

[51] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto: "Causal memory: definitions, implementation, and programming," *Distributed Computing*, volume 9, number 1, pages 37–49, March 1995. doi:10.1007/BF01784241

[52] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen: "Stronger Semantics for Low-Latency Geo-Replicated Storage," at *10th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), April 2013.

[53] Marek Zawirski, Annette Bieniusa, Valter Balegas, et al.: "SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine," INRIA Research Report 8347, August 2013.

[54] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica: "Bolt-on Causal Consistency," at *ACM International Conference on Management of Data* (SIGMOD), June 2013.

[55] Philippe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan: "Challenges to Adopting Stronger Consistency at Scale," at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.

[56] Peter Bailis: "Causality Is Expensive (and What To Do About It)," bailis.org, 5 February 2014.

[57] Ricardo Gonçalves, Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte: "Concise Server-Wide Causality Management for Eventually Consistent Data Stores," at *15th IFIP International Conference on Distributed Applications and Interoperable Systems* (DAIS), pages 66–79, June 2015. doi:10.1007/978-3-319-19129-4_6

[58] Rob Conery: "A Better ID Generator For PostgreSQL," rob.conery.io, 29 May 2014.

[59] 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

[60] Xavier Défago, André Schiper, and Péter Urbán: "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, volume 36, number 4, pages 372–421, December 2004. doi:10.1145/1041680.1041682

[61] Hagit Attiya and Jennifer Welch: *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, Second Edition. John Wiley & Sons, June 2004. ISBN: 9780471453246, doi:10.1002/0471478210

[62] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, et al.: "CORFU: A Shared Log Design for Flash Clusters," at *9th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), April 2012.

[63] Fred B Schneider: "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, volume 22, number 4, pages 299–319, December 1990.

[64] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, et al.: "Calvin: Fast Distributed Transactions for Partitioned Database Systems," at *ACM International Conference on Management of Data* (SIGMOD), May 2012.

[65] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, et al.: "Tango: Distributed Data Structures over a Shared Log," at *24th ACM Symposium on Operating Systems Principles* (SOSP), pages 325–340, November 2013. doi:10.1145/2517349.2522732

[66] 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.

[67] Leslie Lamport: "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, volume 28, number 9, pages 690–691, September 1979. doi:10.1109/TC.1979.1675439

[68] Enis Söztutar, Devaraj Das, and Carter Shanklin: "Apache HBase High Availability at the Next Level," hortonworks.com, 22 January 2015.

[69] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, et al.: "PNUTS: Yahoo!'s Hosted Data Serving Platform," at *34th International Conference on Very Large Data Bases* (VLDB), August 2008. doi:10.14778/1454159.1454167

[70] Tushar Deepak Chandra and Sam Toueg: "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, volume 43, number 2, pages 225–267, March 1996. doi:10.1145/226643.226647

[71] Jim N Gray and Leslie Lamport: "Consensus on Transaction Commit," *ACM Transactions on Database Systems* (TODS), volume 31, number 1, pages 133–160, March 2006. doi:10.1145/1132863.1132867

[72] Rachid Guerraoui: "Revisiting the relationship between non-blocking atomic commitment and consensus," at *9th International Workshop on Distributed Algorithms* (WDAG), pages 87–100, September 1995. doi:10.1007/BFb0022140

[73] Michael J Fischer, Nancy Lynch, and Michael S Paterson: "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, volume 32, number 2, pages 374–382, April 1985. doi:10.1145/3149.214121

[74] Michael Ben-Or: "Another advantage of free choice: Completely asynchronous agreement protocols," at *2nd ACM Symposium on Principles of Distributed Computing* (PODC), August 1983. doi:10.1145/800221.806707

[75] 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.

[76] C Mohan, Bruce G Lindsay, and Ron Obermarck: "Transaction Management in the R* Distributed Database Management System," *ACM Transactions on Database Systems*, volume 11, number 4, pages 378–396, December 1986. doi:10.1145/7239.7266

[77] "Distributed Transaction Processing: The XA Specification," X/Open Company Ltd, Technical Standard XO/CAE/91/300, December 1991. ISBN: 1872630243

[78] Mike Spille: "XA Exposed, Part II," jroller.com, 3 April 2004.

[79] Ivan Silva Neto and Francisco Reverbel: "Lessons Learned from Implementing WS-Coordination and WS-AtomicTransaction," at *7th IEEE/ACIS International*

Conference on Computer and Information Science (ICIS), May 2008. doi:10.1109/ICIS.2008.75

[80] James E Johnson, David E Langworthy, Leslie Lamport, and Friedrich H Vogt: "Formal Specification of a Web Services Protocol," at *1st International Workshop on Web Services and Formal Methods* (WS-FM), February 2004. doi:10.1016/j.entcs.2004.02.022

[81] Dale Skeen: "Nonblocking commit protocols," at *ACM International Conference on Management of Data* (SIGMOD), April 1981. doi:10.1145/582318.582339

[82] Gregor Hohpe: "Your Coffee Shop Doesn't Use Two-Phase Commit," *IEEE Software*, volume 22, number 2, pages 64–66, March 2005. doi:10.1109/MS.2005.52

[83] Pat Helland: "Life beyond Distributed Transactions: an Apostate's Opinion," at *3rd Biennial Conference on Innovative Data Systems Research* (CIDR), pages 132–141, January 2007.

[84] Jonathan Oliver: "My Beef with MSDTC and Two-Phase Commits," blog.jonathanoliver.com, 4 April 2011.

[85] Oren Eini (Ahende Rahien): "The fallacy of distributed transactions," ayende.com, 17 July 2014.

[86] Clemens Vasters: "Transactions in Windows Azure (with Service Bus) – An Email Discussion," vasters.com, 30 July 2012.

[87] "Transactions in Azure," NServiceBus Documentation, Particular Software, 2015.

[88] Randy Wigginton, Ryan Lowe, Marcos Albe, and Fernando Ipar: "Distributed Transactions in MySQL," at *MySQL Conference and Expo*, April 2013.

[89] Mike Spille: "XA Exposed, Part I," jroller.com, 3 April 2004.

[90] Ajmer Dhariwal: "Orphaned MSDTC transactions (-2 spids)," eraofdata.com, 12 December 2008.

[91] Paul Randal: "Real world story of DBCC PAGE saving the day," sqlskills.com, 19 June 2013.

[92] "in-doubt xact resolution Server Configuration Option", SQL Server 2016 documentation, Microsoft Inc., 2016.

[93] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer: "Consensus in the Presence of Partial Synchrony," *Journal of the ACM*, volume 35, number 2, pages 288–323, April 1988. doi:10.1145/42282.42283

[94] Miguel Castro and Barbara H Liskov: "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Transactions on Computer Systems*, volume 20, number 4, pages 396–461, November 2002. doi:10.1145/571637.571640

[95] Brian M Oki and Barbara H Liskov: "Viewstamped Replication: A new primary copy method to support highly-available distributed systems," at *7th ACM Symposium on Principles of Distributed Computing* (PODC), August 1988. doi: 10.1145/62546.62549

[96] Barbara H Liskov and James Cowling: "Viewstamped Replication Revisited," Massachusetts Institute of Technology, Tech Report MIT-CSAIL-TR-2012-021, July 2012.

[97] Leslie Lamport: "The Part-Time Parliament," *ACM Transactions on Computer Systems*, volume 16, number 2, pages 133–169, May 1998. doi:10.1145/279227.279229

[98] Leslie Lamport: "Paxos Made Simple," *ACM SIGACT News*, volume 32, number 4, pages 51–58, December 2001.

[99] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone: "Paxos Made Live — An Engineering Perspective," at *26th ACM Symposium on Principles of Distributed Computing* (PODC), June 2007.

[100] Robbert van Renesse: "Paxos made moderately complex," cs.cornell.edu, March 2011.

[101] Diego Ongaro: "Consensus: Bridging Theory and Practice," PhD Thesis, Stanford University, August 2014.

[102] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft: "Raft Refloated: Do We Have Consensus?," *ACM SIGOPS Operating Systems Review*, volume 49, number 1, pages 12–21, January 2015. doi:10.1145/2723872.2723876

[103] André Medeiros: "ZooKeeper's atomic broadcast protocol: Theory and practice," Aalto University School of Science, March 2012.

[104] Robbert van Renesse, Nicolas Schiper, and Fred B Schneider: "Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab," *IEEE Transactions on Dependable and Secure Computing*, volume 12, number 4, pages 472–484, September 2014. doi: 10.1109/TDSC.2014.2355848

[105] Will Portnoy: "Lessons Learned from Implementing Paxos," blog.willportnoy.com, 14 June 2012.

[106] Heidi Howard and Jon Crowcroft: "Coracle: Evaluating Consensus at the Internet Edge," at *Annual Conference of the ACM Special Interest Group on Data Communication* (SIGCOMM), pages 85–86, August 2015. doi:10.1145/2829988.2790010

[107] Kyle Kingsbury: "Call me maybe: Elasticsearch 1.5.0," aphyr.com, 27 April 2015.

[108] Ivan Kelly: "BookKeeper Tutorial," github.com, October 2014.

[109] Camille Fournier: "Consensus Systems for the Skeptical Architect," at *Craft Conference*, Budapest, Hungary, April 2015.

[110] Kenneth P Birman: "A History of the Virtual Synchrony Replication Model," in *Replication: Theory and Practice*, Springer LNCS volume 5959, chapter 6, pages 91–120, 2010. ISBN: 978-3-642-11293-5, doi:10.1007/978-3-642-11294-2_6

PART III
Derived Data

In Part I and Part II of this book, we assembled from the ground up all the major considerations that go into a distributed database, from the layout of data on disk, all the way to the limits of distributed consistency in the presence of faults. However, this discussion assumed that there was only one database in the application.

In reality, data systems are often more complex. In a large application you often need to be able to access and process data in many different ways, and there is no one database which can satisfy all those different needs simultaneously. Applications thus commonly use a combination of several different datastores, indexes, caches, analytics systems etc. and implement mechanisms for moving data from one store to another.

In this final Part III of the book, we will examine the issues around integrating multiple different data systems, potentially with different data models and optimized for different access patterns, into one coherent application architecture. This aspect of system-building is often overlooked by vendors who claim that their product can satisfy all your needs. In reality, integrating disparate systems is one of the most important things that needs to be done in a non-trivial application.

Systems of Record and Derived Data

On a high level, systems that store and process data can be grouped into two broad categories:

Systems of record

A system of record, also known as *source of truth*, holds the authoritative version of your data. When new data comes in, e.g. as user input, it is first written here. Each fact is represented exactly once (the representation is typically *normalized*). If there is any discrepancy between another system and the system of record, then the value in the system of record is (by definition) the correct one.

Derived data systems

Data in a derived system is the result of taking some existing data from another system and transforming or processing it in some way. If you lose derived data, you can re-create it from the original source. A classic example is a cache: data can be served from the cache if present, but if the cache doesn't contain what you need, you can fall back to the underlying database. Denormalized values, indexes and materialized views also fall in this category. In recommendation systems, predictive summary data is often derived from usage logs.

Technically speaking, derived data is *redundant*, in the sense that it duplicates existing information. However, it is often essential for getting good performance on read queries. It is often *denormalized*. You can derive several different datasets from a single source, enabling you to look at the data from different "points of view".

Not all systems make a clear distinction between systems of record and derived data in their architecture, but it's a very helpful distinction to make, because it clarifies the dataflow through your system: it makes explicit which parts of the system have which inputs and which outputs, and how they depend on each other.

Most databases, storage engines and query languages are not inherently a system of record or a derived system. A database is just a tool: how you use it is up to you. The distinction between system of record and derived data system depends not on the tool, but on how you use it in your application.

By being clear about which data is derived from which other data, you can bring clarity to an otherwise confusing system architecture. This point will be a running theme throughout Part III of this book.

Overview of Chapters

We will start in Chapter 10 by examining batch-oriented dataflow systems such as MapReduce, and show how they give us good tools and principles for building large-scale data systems. In Chapter 11 we will take those ideas and apply them to data streams, which allow us to do the same kinds of things with lower delays. ??? concludes the book by exploring ideas about how we might build reliable, scalable and maintainable applications in future.

CHAPTER 10

Batch Processing

> A system cannot be successful if it is too strongly influenced by a single person. Once the initial design is complete and fairly robust, the real test begins as people with many different viewpoints undertake their own experiments.
>
> —Donald Knuth

In the first two parts of this book we talked a lot about *requests* and *queries*, and the corresponding *responses* or *results*. This style of data processing is assumed in many modern data systems: you ask for something, or you send an instruction, and some time later the system (hopefully) gives you an answer. Databases, caches, search indexes, web servers, and many other systems work this way.

In such *online* systems, whether it's a web browser requesting a page, or a service calling a remote API, we generally assume that the request is triggered by a human user, and that the user is waiting for the response. They shouldn't have to wait too long for the response, so we pay a lot of attention to the *response time* of these systems (see "Describing performance" on page 11).

The web, and increasing numbers of HTTP-/REST-based APIs, have made the request/response style of interaction so common that it's easy to take it for granted. But we should remember that it's not the only way of building systems, and that other approaches have their merits too. Let's distinguish three different types of system:

Services (online systems)
: A service waits for a request or instruction from a client to arrive. When one is received, the service tries to handle it as quickly as possible and sends a response back. Response time is usually the primary measure of performance of a service,

and availability is often very important (if the client can't reach the service, the user will probably get an error message).

Batch processing systems (offline systems)

A batch processing system takes a large amount of input data, runs a *job* to process it, and produces some output data. Jobs often take a while (from a few minutes to several days), so there normally isn't a user waiting for the job to finish. Instead, batch jobs are often scheduled to run periodically (for example, once a day). The primary performance measure of a batch job is usually *throughput* (the time it takes to crunch through an input dataset of a certain size). We discuss batch processing in this chapter.

Stream processing systems (near-real-time systems)

Stream processing is somewhere between online and offline/batch (so it is sometimes called *near-real-time* or *nearline* processing). Like batch processing, a stream processor consumes inputs and produces outputs (rather than responding to requests). However, a job operates on events shortly after they happen, whereas a batch job operates on a fixed set of input data. This difference allows stream processing systems to have lower latency than the equivalent batch systems. As stream processing builds upon batch processing, we discuss it in Chapter 11.

As we shall see in this chapter, batch processing is an important building block in our quest to build reliable, scalable and maintainable applications. For example, MapReduce, a batch-processing algorithm published in 2004 [1] was (perhaps over-enthusiastically) called *"the algorithm that makes Google so massively scalable"* [2]. It was subsequently implemented in various open source data systems, including Hadoop, CouchDB and MongoDB.

However, MapReduce is in some ways also a step backwards from more sophisticated parallel processing techniques that were developed for data warehouses many years before [3, 4]. Although the importance of MapReduce is now declining [5], it is still worth understanding, because it provides a clear picture of why and how batch processing is useful.

In fact, batch processing is a very old form of computing. Long before programmable digital computers were invented, punch card tabulating machines — such as the Hollerith machines used in the 1890 US Census [6] — implemented a semi-mechanized form of batch processing to compute aggregate statistics from large inputs. And MapReduce bears an uncanny resemblance to the electromechanical IBM card-sorting machines that were widely used for business data processing in the 1940s and 1950s [7]. As usual, history has a tendency of repeating itself.

In this chapter, we will look at MapReduce and several other batch processing algorithms and frameworks, and explore how they are used in modern data systems. But first, to get started, we will look at data processing using standard Unix tools. Even if you are already familiar with them, it's worth reminding ourselves of the Unix philosophy, because the ideas and approaches from Unix carry over directly to large-scale, heterogeneous distributed data systems.

Batch Processing with Unix Tools

Let's start with a simple example. Say you have a web server that appends a line to a log file every time it serves a request. For example, using the nginx default access log format, one line of the log might look like this:

```
216.58.210.78 - - [27/Feb/2015:17:55:11 +0000] "GET /css/typography.css HTTP/1.1"
200 3377 "http://martin.kleppmann.com/" "Mozilla/5.0 (Macintosh; Intel Mac OS X
10_9_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.115 Safari/537.36"
```

(That is actually one line, it's only broken onto multiple lines here for readability.) There's a lot of information in that one line. In order to interpret it, you need to look at the definition of the log format, which is as follows:

```
$remote_addr - $remote_user [$time_local] "$request"
$status $body_bytes_sent "$http_referer" "$http_user_agent"
```

This line of the log indicates that on 27 February 2015 at 17:55:11 UTC, the server received a request for the file `/css/typography.css` from the client IP address 216.58.210.78. The user was not authenticated, so `$remote_user` is set to a dash (`-`). The response status was 200, i.e. the request was successful, and the response was 3377 bytes in size. The file was embedded in the page at URL `http://martin.kleppmann.com/` and the web browser was Chrome 40.

Simple log analysis

Various tools can take these log files and produce pretty reports about your website traffic, but for the sake of exercise, let's build our own, using basic Unix tools. For example, say you want to find the 5 most popular pages on your website. You can do this in a Unix shell as follows:[i]

```
cat /var/log/nginx/access.log | ❶
  awk '{print $7}' | ❷
  sort             | ❸
  uniq -c          | ❹
  sort -r -n       | ❺
  head -n 5          ❻
```

[i] Some people love to point out that `cat` is unnecessary here, as the input file could be given directly as an argument to `awk`. However, the linear pipeline is more apparent when written like this.

❶ Read the log file.

❷ Split each line into fields by whitespace, and output only the seventh such field from each line, which happens to be the requested URL. In the example above, this would be `/css/typography.css`.

❸ Alphabetically `sort` the list of requested URLs. If some URL has been requested *n* times, then after sorting, the file contains the same URL repeated *n* times in a row.

❹ The `uniq` command filters out repeated lines in its input by checking whether two adjacent lines are the same. The `-c` option tells it to also output a counter: for every distinct URL, it reports how many times that URL appeared in the input.

❺ The second `sort` sorts by the number (`-n`) at the start of each line, which is the number of times the URL was requested. It then returns the results in reverse (`-r`) order, i.e. with the largest number first.

❻ Finally, `head` outputs just the first 5 lines (`-n 5`) of input, and discards the rest.

The output of that series of commands looks something like this:

```
4189 /favicon.ico
3631 /2013/05/24/improving-security-of-ssh-private-keys.html
2124 /2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html
1369 /
 915 /css/typography.css
```

Although the above command line looks a bit obscure if you're unfamiliar with Unix tools, it is incredibly powerful. It will process gigabytes of log files in a matter of seconds, and you can easily modify the analysis to suit your needs. For example, if you want to omit CSS files from the report, change the `awk` argument to `'$7 !~ /\.css$/ {print $7}'`. If you want to count top client IP addresses instead of top pages, change the `awk` argument to `'{print $1}'`. And so on.

We don't have space in this book to explain Unix tools in detail, but they are very much worth learning about. Surprisingly many data analyses can be done in a few minutes using some combination of `awk`, `sed`, `grep`, `sort`, `uniq` and `xargs`, and they perform surprisingly well [8].

Chain of commands vs. custom program

Instead of this chain of Unix commands, you could imagine writing a simple program to do the same thing. For example, in Ruby, it might look something like this:

```
counts = Hash.new(0) ❶
```

```
File.open('/var/log/nginx/access.log') do |file|
  file.each do |line|
    url = line.split[6] ❷
    counts[url] += 1 ❸
  end
end

top5 = counts.map{|url, count| [count, url] }.sort.reverse[0...5] ❹
top5.each{|count, url| puts "#{count} #{url}" } ❺
```

❶ `counts` is a hash table that keeps a counter for the number of times we've seen each URL. A counter is zero by default.

❷ From each line of the log, we take the URL to be the seventh whitespace-separated field (the array index here is 6 because Ruby's arrays are zero-indexed).

❸ Increment the counter for the URL in the current line of the log.

❹ Sort the hash table contents by counter value descending, and take the top 5 entries.

❺ Print out those top 5 entries.

This is not as concise as the chain of Unix pipes, but it's fairly readable, and which of the two you prefer is partly a matter of taste. However, besides the superficial syntactic differences between the two, there is a big difference in the execution flow, which becomes apparent if you run this analysis on a large file.

Sorting vs. in-memory aggregation

The Ruby script keeps an in-memory hash table of URLs, where each URL is mapped to the number of times it has been seen. The Unix pipeline example does not have such a hash table, but instead relies on sorting a list of URLs in which multiple occurrences of the same URL are simply repeated.

Which approach is better? It depends how many different URLs you have. For most small to mid-sized websites, you can probably fit all distinct URLs, and a counter for each URL, in (say) 1 GB of memory. In this example, the *working set* of the job (the amount of memory to which the job needs random access) depends only on the number of distinct URLs: if there are a million log entries for a single URL, the space required in the hash table is still just one URL plus the size of the counter. If this working set is small enough, an in-memory hash table works fine — even on a laptop.

On the other hand, if the job's working set is larger than memory, the sorting approach has the advantage that it can make efficient use of disks. It's the same principle as we discussed in "SSTables and LSM-trees" on page 74: chunks of data can be sorted in memory, written out to disk as segment files, and then multiple sorted segments can be merged into a larger sorted file. Mergesort has sequential access patterns that perform well on disks. (Remember that optimizing for sequential I/O was a recurring theme in Chapter 3. The same pattern reappears here.)

The `sort` utility in GNU Coreutils (Linux) automatically handles larger-than-memory datasets by spilling to disk, and automatically parallelizes sorting across multiple CPU cores [9]. This means that the simple chain of Unix commands above easily scales to large datasets, without running out of memory. The bottleneck is likely to be the rate at which the input file can be read from disk.

The Unix philosophy

It's no coincidence that we were able to analyze a log file quite easily, using a chain of commands like in the example above: this was in fact one of the key design ideas of Unix, and it remains astonishingly relevant today. Let's look at it in some more depth so that we can borrow ideas from Unix [10].

Doug McIlroy, the inventor of Unix pipes, first described them like this in 1964 [11]: *"We should have some ways of connecting programs like [a] garden hose — screw in another segment when it becomes necessary to massage data in another way. This is the way of I/O also."* The plumbing analogy stuck, and the idea of connecting programs with pipes became part of what is now known as the *Unix philosophy* — a set of design principles that became popular among the developers and users of Unix. The philosophy was described in 1978 as follows [12, 13]:

> 1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".
> 2. Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
> 3. Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
> 4. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

This approach — automation, rapid prototyping, incremental iteration, being friendly to experimentation, and breaking down large projects into manageable chunks — sounds remarkably like the Agile and DevOps movements of today. Surprisingly little has changed in four decades.

The `sort` tool is a great example of a program that does one thing well. It is arguably a better sorting implementation than most programming languages have in their standard library (which do not spill to disk and do not use multiple threads, even when that would be beneficial). And yet, sort is barely useful in isolation. It only becomes powerful in combination with the other Unix tools, such as uniq. It would have been easy for the implementer of sort to add uniq as a feature, but they resisted the temptation.

A Unix shell like bash lets us easily *compose* these small programs into surprisingly powerful data processing jobs. Even though many of these programs are written by different groups of people, they can be joined together in flexible ways. What does Unix do to enable this composability?

A uniform interface

If you expect the output of one program to become the input to another program, that means those programs must use the same data format — in other words, a compatible interface. If you want to be able to connect *any* program's output to *any* program's input, that means that *all* programs must use the same input/output interface.

In Unix, that interface is a file (or, more precisely, a file descriptor). A file is just an ordered sequence of bytes. Because that is such a simple interface, many different things can be represented using the same interface: an actual file on the filesystem, a communication channel to another process (Unix socket, stdin, stdout), a device driver (say /dev/audio or /dev/lp0), a socket representing a TCP connection, and so on. It's easy to take this for granted, but it's actually quite remarkable that these very different things can share a uniform interface, so they can easily be plugged together.[ii]

By convention, many (but not all) Unix programs treat this sequence of bytes as ASCII text. The log analysis example above used this fact: awk, sort, uniq and head all treat their input file as a list of records separated by the \n (newline, ASCII 0x0A) character. The choice of \n is arbitrary — arguably, the ASCII record separator 0x1E would have been a better choice, since it's intended for this purpose [14] — but in any case, the fact that all these programs have standardized on using the same record separator allows them to interoperate.

The parsing of each record (i.e. a line of input) is more vague. Unix tools commonly split a line into fields by whitespace or tab characters, but CSV (comma-separated), pipe-separated and other encodings are also used. Even a fairly simple tool like xargs has half a dozen command-line options for specifying how its input should be parsed.

[ii] Another example of a uniform interface is URLs and HTTP, the foundations of the web. A URL identifies a particular thing (resource) on a website, and you can link to any URL from any other website. A user with a web browser can thus seamlessly jump between websites by following links, even though the servers may be operated by entirely unrelated organizations. This seems obvious today, but it was a key insight towards making the web the success that it is today.

The uniform interface of ASCII text mostly works, but it's not exactly beautiful: our log analysis example used {print $7} to extract the URL, which is not very readable. In an ideal world this could have perhaps been {print $request_url} or something of that sort. We will return to this idea later.

Although it's not perfect, even decades later, the uniform interface of Unix is still something remarkable. Not many pieces of software interoperate and compose as well as Unix tools do: you can't easily pipe the contents of your email account and your online shopping history through a custom analysis tool, into a spreadsheet, and post the results to a social network or a wiki. Today it's an exception, not the norm, to have programs that work together as smoothly as Unix tools do.

Even databases with the *same data model* often don't make it easy to get data out of one and into the other. This leads to balkanization of data.

Separation of logic and wiring

Another characteristic feature of Unix tools is their use of standard input (stdin) and standard output (stdout). If you run a program and don't specify anything else, stdin comes from the keyboard and stdout goes to the screen. However, you can also take input from a file, and/or redirect output to a file. Pipes let you attach the stdout of one process to the stdin of another process (with a small in-memory buffer, and without writing the entire intermediate data stream to disk).

Of course a program can still read and write files directly if it needs to, but the Unix approach works best if a program doesn't worry about particular file paths, and simply uses stdin and stdout. This allows a shell user to wire up the input and output in whatever way they want; the program doesn't know or care where the input is coming from and where the output is going to. (One could say this is a form of *loose coupling*, *late binding* [15] or *inversion of control* [16].) Separating the input/output wiring from the program logic makes it easier to compose small tools into bigger systems.

You can even write your own programs and combine them with the tools provided by the operating system. Your program just needs to read input from stdin, and write output to stdout, and it can participate in data processing pipelines. In the log analysis example, you could write a tool that translates user-agent strings into more sensible browser identifiers, or a tool that translates IP addresses into country codes, and simply plug it into the pipeline. The sort program doesn't care whether it's communicating with another part of the operating system or with a program written by you.

However, there are limits to what you can do with stdin and stdout. Programs that need multiple inputs or outputs are possible but tricky. You can't pipe a program's output into a network connection [17, 18].[iii] If a program directly opens files for reading and writing, or starts another program as a subprocess, or opens a network connection, then that I/O is wired up by the program itself. It can still be configurable (through command-line options, for example) but the flexibility of wiring up inputs and outputs in a shell is reduced.

Transparency and experimentation

Part of what makes Unix tools so successful is that they make it quite easy to see what is going on:

- The input files to Unix commands are normally treated as immutable. This means you can run the commands as often as you want, trying various command-line options, without damaging the input files.
- You can end the pipeline at any point, pipe the output into `less`, and look at it to see if it has the expected form. This is great for debugging.
- You can write the output of one pipeline stage to a file, and use that file as input to the next stage. This allows you to restart the later stage without re-running the entire pipeline.

Thus, even though Unix tools are quite blunt, simple tools compared to a query optimizer of a relational database, they remain amazingly useful, especially for experimentation.

However, the biggest limitation of Unix tools is that they run only on a single machine — and that's where tools like Hadoop come in.

MapReduce and Distributed Filesystems

MapReduce is a bit like Unix tools, but distributed across potentially thousands of machines. Like Unix tools, it is a fairly blunt, brute-force, but surprisingly effective tool. A single MapReduce job is comparable to a single Unix process: it takes one or more inputs and produces one or more outputs.

As with most Unix tools, running a MapReduce job normally does not modify the input, and does not have any side-effects other than producing the output. The output files are written once, in a sequential fashion (not modifying any existing part of a file once it has been written).

[iii] Except by using a separate tool, such as `netcat` or `curl`. Unix started out trying to represent everything as files, but the BSD sockets API deviated from that convention [17]. The research operating systems *Plan 9* and *Inferno* are more consistent in their use of files: they represent a TCP connection as a file in `/net/tcp` [18].

While Unix tools use `stdin` and `stdout` as input and output, MapReduce jobs read and write files on a distributed filesystem. In Hadoop's implementation of MapReduce, that filesystem is called HDFS (Hadoop Distributed FileSystem), an open source re-implementation of Google's GFS [19].

HDFS is based on the *shared-nothing* principle: every general-purpose machine in a datacenter has some disks attached to it, and HDFS creates (conceptually) one big filesystem that can use the space on all those disks. HDFS runs a daemon process on each machine, exposing a network service that allows other nodes to access files stored on that machine. A central server called *NameNode* keeps track of which file blocks are stored on which machine.

In order to tolerate machine and disk failures, file blocks are replicated on multiple machines. Replication may mean simply several copies of the same data on multiple machines, as in Chapter 5, or an *erasure coding* scheme such as Reed-Solomon codes, which allows lost data to be recovered with lower storage overhead than full replication [20]. The techniques are similar to RAID, which provides redundancy across several disks attached to the same machine; the difference is that in a distributed filesystem, file access and replication goes via a conventional datacenter network without special hardware.

The shared-nothing approach is in contrast to *Network Attached Storage* (NAS) and *Storage Area Network* (SAN) architectures, where shared storage is provided by a centralized storage appliance. Such appliances often use custom hardware and special network infrastructure such as Fibre Channel. (In a SAN/NAS architecture, individual machines may still have their own disks, but they are only used by the local operating system and not as shared storage.)

HDFS has scaled well: at the time of writing, the biggest HDFS deployments run on tens of thousands of machines, with combined storage capacity of hundreds of petabytes [21]. Such large scale has become viable because the cost of data storage and access on HDFS, using commodity hardware and open source software, is much lower than the equivalent capacity on a dedicated storage appliance [22].

Various other distributed filesystems besides HDFS exist, such as GlusterFS or the Quantcast File System [23]. Object storage services such as Amazon S3, Azure Blob Storage and OpenStack Swift [24] are similar in many ways.[iv] In this chapter we will mostly use HDFS as running example, but the principles apply to any distributed filesystem.

[iv] One difference is that with HDFS, computing tasks can be scheduled to run on the machine that stores a copy of a particular file, whereas object stores usually keep storage and computation separate. Reading from local disk has a performance advantage if network bandwidth is a bottleneck. Note that if erasure coding is used, the locality advantage is lost, because the data from several machines must be combined in order to reconstitute the original file [23].

MapReduce job execution

MapReduce is a programming framework with which you can write code to process large datasets in a distributed filesystem like HDFS. The easiest way of understanding it is by referring back to the web server log analysis example in "Simple log analysis" on page 379. The pattern of data processing in MapReduce is very similar to this example:

1. Read a set of input files, and break it up into *records*. In the web server log example, each record is one line in the log (that is, `\n` is the record separator).
2. Extract a *key* and *value* from each input record. In the example above, this is done by the command `awk '{print $7}'`: it extracts the URL (`$7`) as key, and leaves the value empty.
3. Sort all of the key-value pairs by key. In the example above, this is done by the first `sort` command.
4. Iterate over the sorted list of key-value pairs. If there are multiple occurrences of the same key, the sorting has made them adjacent in the list, so it is easy to combine those values without having to keep a lot of state in memory. In the example above, this is done by the command `uniq -c`, which counts the number of adjacent records with the same key.

Those four steps can be performed by one MapReduce job. When you write a MapReduce job, you write two callback functions, *map* and *reduce* (see also "MapReduce querying" on page 45):

Map

The map function is called once for every input record, and its job is to extract the key and value from the input record. In the web server log example, `awk '{print $7}'` is effectively the mapper. In this case, one input record results in one key-value pair, but it's also ok for the mapper to produce zero or more key-value pairs for one input record.

Reduce

The MapReduce framework collects all the key-value pairs with the same key, and calls the reduce function with an iterator over that collection of values with the same key. In the web server log example, `uniq -c` is effectively the reducer. The reducer can produce output records (such as the number of occurrences of the same URL).

Note that step 3, the `sort` step, is implicit in MapReduce — you don't have to write it, because the output from the mapper is always sorted before it is given to the reducer. This sorting step is arguably the most important aspect of MapReduce.

In the web server log example, we had a second `sort` command in step 5, which ranked URLs by number of requests. In MapReduce, if you need a second sorting stage, you can implement it by writing a second MapReduce job, and using the output of the first job as input to the second job. Viewed like this, the role of the mapper is to prepare the data into a form that is suitable for sorting, and the role of the reducer is to process the data that has been sorted.

Distributed execution of MapReduce

The main difference to pipelines of Unix commands is that MapReduce can parallelize a computation across many machines, without having to write code to explicitly handle the parallelism. The *map* and *reduce* functions only operate on one record at a time; they don't need to know where their input is coming from or their output is going to, so the framework can handle the complexities of moving the data between machines.

It is possible to use standard Unix tools as mappers and reducers in a distributed computation [25], but more commonly they are implemented as functions in a conventional programming language. In Hadoop MapReduce, the mapper and reducer are each a Java class that implements a particular interface. In MongoDB and CouchDB, mappers and reducers are JavaScript functions (see "MapReduce querying" on page 45).

Figure 10-1 shows the dataflow in a Hadoop MapReduce job. Its parallelization is based on partitioning (see Chapter 6): the input to a job is typically a directory in HDFS, and each file or file block within the input directory is considered to be a separate partition that can be processed by a separate mapper task (marked by m 1, m 2, and m 3 in Figure 10-1).

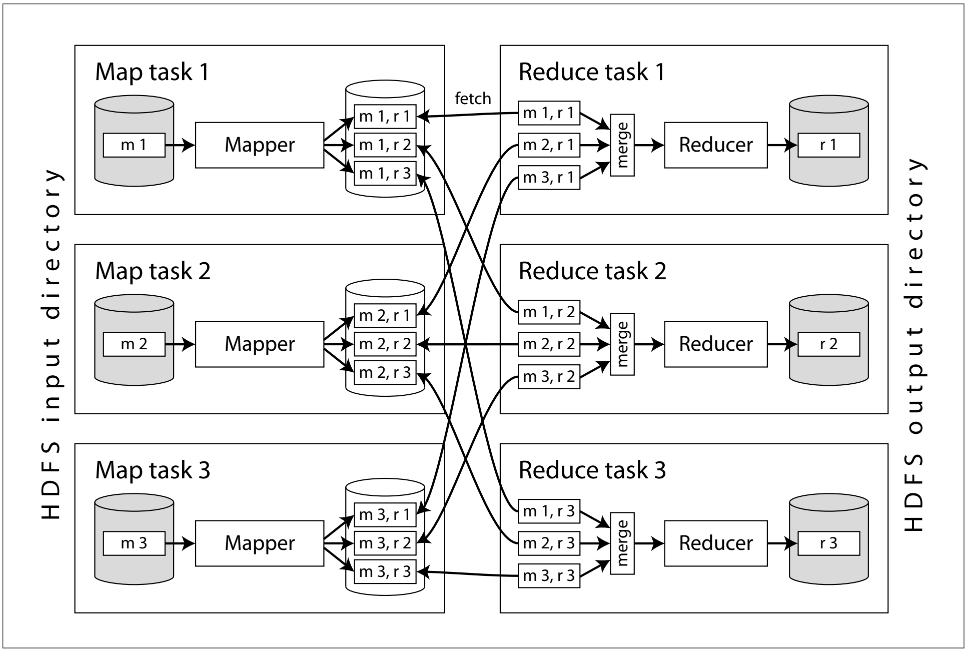


Figure 10-1. A MapReduce job with three mappers and three reducers.

Each input file is typically hundreds of megabytes in size. The MapReduce scheduler (not shown in the diagram) tries to run each mapper on one of the machines that stores a replica of the input file, provided that machine has enough spare RAM and CPU to run the mapper task [26]. This principle is known as *putting the computation near the data* [27]: it saves copying the input file over the network, reducing network load and increasing locality.

In most cases, the application code that should run in the mapper task is not yet present on the machine that is assigned the task of running it, so the MapReduce framework first copies the code (e.g. jar files) to the appropriate machines. It then starts the map task and begins reading the input file, passing one record at a time to the mapper callback. The output of the mapper consists of key-value pairs.

The reduce side of the computation is also partitioned. While the number of map tasks is determined by the number of input file blocks, the number of reduce tasks is configured by the job author (it can be different from the number of map tasks). To ensure that all key-value pairs with the same key end up at the same reducer, the framework uses a hash of the key to determine which reduce task should receive a particular key-value pair (see "Partitioning by hash of key" on page 195).

The key-value pairs must be sorted, but the dataset is likely too large to be sorted with a conventional sorting algorithm on a single machine. Instead, the sorting is performed in stages. First, each mapper task partitions its output by reducer, based on the hash of the key. Each of these partitions is written to a sorted file on the mapper's local disk, using a technique similar to what we discussed in "SSTables and LSM-trees" on page 74.

When a mapper has finished reading its input file and writing its sorted output files, the MapReduce scheduler notifies the reducers that they can start fetching the output files from the mappers. The reducers connect to each of the mappers and download the file of sorted key-value pairs for their partition. The process of partitioning by reducer, sorting and copying data partitions from mappers to reducers is known as *the shuffle* [26] (a confusing term — unlike shuffling a deck of cards, there is no randomness in MapReduce).

The reduce task takes the files from the mappers and merges them together, preserving the sort order. Thus, if different mappers produced records with the same key, they will be adjacent in the merged reducer input.

The reducer function is called with a key and an iterator that incrementally scans over all records with the same key (which may in some cases be bigger than what can fit in memory). The reducer can use arbitrary logic to process these records, and generate any number of output records in the process. These output records are written to a file on HDFS (usually, one copy on the local disk of the machine running the reducer, with replicas on other machines).

MapReduce workflows

The range of problems you can solve with a single MapReduce job are limited. Referring back to the log analysis example, a single MapReduce job could determine the number of page views per URL, but not the most popular URLs, since that requires a second round of sorting.

Thus, it is very common for MapReduce jobs to be chained together into *workflows*, such that the output of one job becomes the input to the next job. The Hadoop MapReduce framework does not have any particular support for workflows, so this chaining is done implicitly by directory name: the first job must be configured to write its output to a designated directory in HDFS, and the second job must be configured to read that same directory name as its input. From the MapReduce framework's point of view, they are two independent jobs.

Chained MapReduce jobs are therefore less like pipelines of Unix commands (which pass the output of one process as input to another process directly, using only a small in-memory buffer), and more like a sequence of commands where each command's output is written to a temporary file, and the next command reads from the temporary file. This design has advantages and disadvantages, which we will discuss in "Materialization of intermediate state" on page 407.

A batch job's output is only considered valid when the job has completed successfully (MapReduce discards the partial output of a failed job). Therefore, one job in a workflow can only start when the prior jobs — that is, the jobs that produce its input directories — have completed successfully. To handle these dependencies between job executions, various workflow schedulers for Hadoop have been developed, including Oozie, Azkaban, Luigi, Airflow, and Pinball [28].

These schedulers also have management features that are useful when maintaining a large collection of batch jobs. Workflows consisting of 50 to 100 MapReduce jobs are common when building recommendation systems [29], and in a large organization, many different teams may be running different jobs that read each others' output. Tool support is important for managing such complex dataflows.

Various higher-level tools for Hadoop, such as Pig [30], Hive [31], Cascading [32], Crunch [33], and FlumeJava [34], also set up workflows of multiple MapReduce stages that are automatically wired together appropriately.

Reduce-side joins and grouping

We discussed joins in Chapter 2 in the context of data models and query languages, but we have not said much about how joins are actually executed. It is time that we pick up that thread again.

In many datasets it is common for one record to have a association to another record: a *foreign key* in a relational model, a *document reference* in a document model, or an *edge* in a graph model. A join is necessary whenever you have some code that needs to access records on both sides of that association (both the record that holds the reference and the record being referenced). As discussed in Chapter 2, denormalization can reduce the need for joins, but generally not remove joins entirely.[v]

In a database, if you execute a query that involves only a small number of records, the database would typically use an *index* to quickly locate the records of interest (see Chapter 3). If the query involves joins, it may require multiple index lookups. However, MapReduce has no concept of indexes — at least not in the usual sense.

When a MapReduce job is given a set of files as input, it reads the entire content of all of those files; a database would call this operation a *full table scan*. If you only want to read a small number of records, a full table scan is outrageously expensive compared to an index lookup. However, in analytic queries (see "Transaction Processing or Analytics?" on page 87) it is common to want to calculate aggregates over a large

[v] The joins we talk about in this book are generally *equi-joins*, the most common type of join, in which a record is associated with other records that have *an identical value* in a particular field (such as an ID). Some databases support more general types of join, for example using a less-than operator instead of an equality operator, but we will not go into them here.

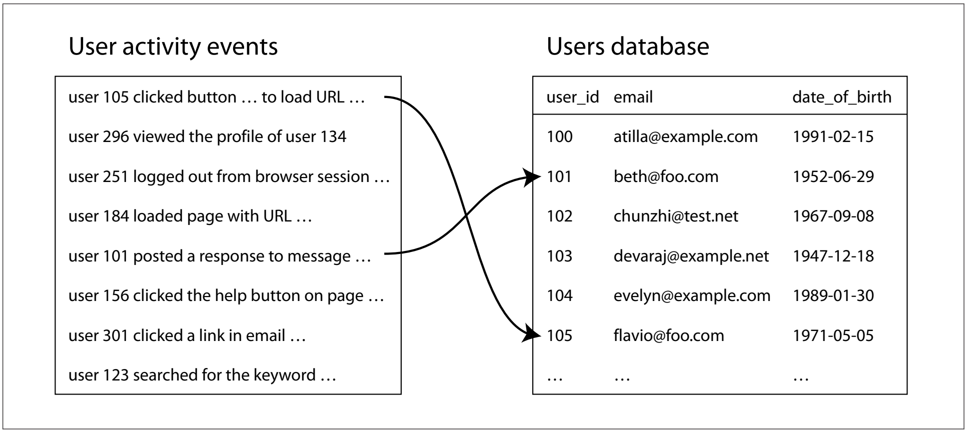
number of records. In this case, scanning the entire input might be quite a reasonable thing to do, especially if you can parallelize the processing across multiple machines.

When we talk about joins in the context of batch processing, we mean resolving all occurrences of some association within a dataset. For example, we assume that a job is processing the data for all users simultaneously, not merely looking up the data for one particular user (which would be done far more efficiently with an index).

Example: analysis of user activity events

A typical example of a join in a batch job is illustrated in Figure 10-2. On the left is a log of events describing the things that logged-in users did on a website (known as *activity events* or *clickstream data*), and on the right is a database of users. You can think of this example as being part of a star schema (see "Stars and snowflakes: schemas for analytics" on page 90): the log of events is the fact table, and the user database is one of the dimensions.

An analytics task may need to correlate user activity with user profile information: for example, if the profile contains the user's age or date of birth, the system could determine which pages are most popular with which age groups. However, the activity events contain only the user ID, not the full user profile information. Embedding that profile information in every single activity event would most likely be too wasteful. Therefore, the activity events need to be joined with the user profile database.



User activity events

user 105 clicked button … to load URL …
user 296 viewed the profile of user 134
user 251 logged out from browser session …
user 184 loaded page with URL …
user 101 posted a response to message …
user 156 clicked the help button on page …
user 301 clicked a link in email …
user 123 searched for the keyword …

Users database

user_id	email	date_of_birth
100	atilla@example.com	1991-02-15
101	beth@foo.com	1952-06-29
102	chunzhi@test.net	1967-09-08
103	devaraj@example.net	1947-12-18
104	evelyn@example.com	1989-01-30
105	flavio@foo.com	1971-05-05
…	…	…

Figure 10-2. A join between a log of user activity events and a database of user profiles.

The simplest implementation of this join would go over the activity events one by one, and query the user database (on a remote server) for every user ID it encounters. This is possible, but it would most likely suffer from very poor performance: the processing throughput would be limited by the round-trip time to the database server, the effectiveness of a local cache would depend very much on the distribution of data, and running a large number of queries in parallel could easily overwhelm the database [35].

In order to achieve good throughput in a batch process, the computation must be local to one machine as much as possible. Making random-access requests over the network for every record you want to process is too slow. Moreover, querying a remote database would mean that the batch job becomes non-deterministic, because the data in the remote database might change.

Thus, a better approach would be to take a copy of the users database (for example, extracted from a database backup using an ETL process, see "Data warehousing" on page 88) and to put it in the same distributed filesystem as the log of user activity events. You then have the user database in one set of files in HDFS, and the user activity records in another set of files. MapReduce can be used to bring together all of the relevant records in the same place and process them efficiently.

Sort-merge joins

Recall that the purpose of the *map* function is to extract a key and value from each input record. In the case of Figure 10-2, this key would be the user ID: one set of mappers would go over the activity events (extracting the user ID as key, and the activity event as value), while another set of mappers would go over the user database (extracting the user ID as key, and the user's date of birth as value). This process is illustrated in Figure 10-3.

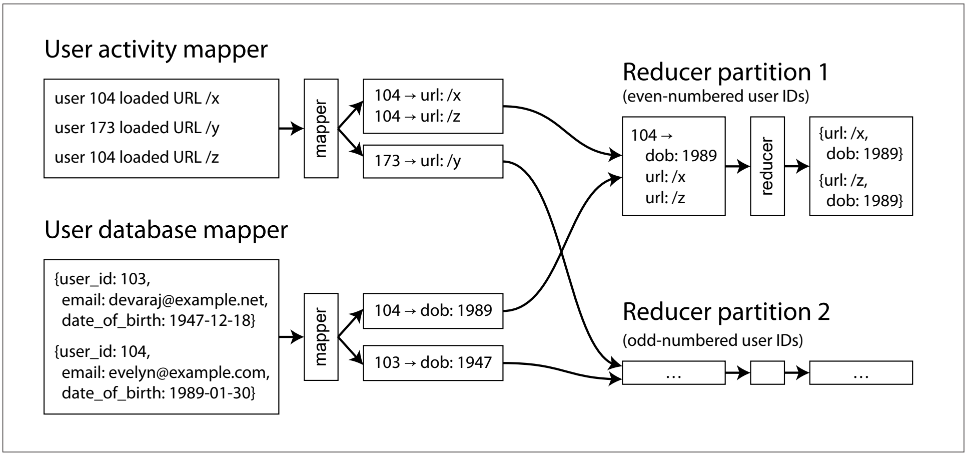


Figure 10-3. A reduce-side sort-merge join on user ID. If the input datasets are partitioned into multiple files, each could be processed with multiple mappers in parallel.

When the MapReduce framework partitions the mapper output by key, and then sorts the key-value pairs, the effect is that all the activity events and the user record with the same user ID become adjacent to each other in the reducer input. The Map-

Reduce job can even arrange the records to be sorted such that the reducer always sees the record from the user database first, followed by the activity events in timestamp order — this is known as a *secondary sort* [26].

The reducer can then perform the actual join logic easily: the reduce function is called once for every user ID, and thanks to the secondary sort, the first value is expected to be the date-of-birth record from the user database. The reducer remembers the date of birth in a local variable, and then iterates over the activity events with the same user ID, outputting pairs of *viewed-url* and *viewer-age-in-years*. Subsequent MapReduce jobs could then calculate the distribution of viewer ages for each URL, and cluster by age group.

Since the reducer processes all of the records for a particular user ID in one go, it only needs to keep one user record in memory at any one time, and it never needs to make any requests over the network. This algorithm is known as a *sort-merge join*, since mapper output is sorted by key, and the reducers then merge together the sorted lists of records from both sides of the join.

Bringing related data together in the same place

The mappers and the sorting process have made sure that all the necessary data to perform the join operation for a particular user ID has been brought together in the same place: a single call to the *reduce* function. Having lined up all the required data in advance, the reducer can be a fairly simple, single-threaded piece of code that can churn through records with high throughput and low memory overhead.

One way of looking at this architecture is that mappers "send messages" to the reducers. When a mapper emits a key-value pair, the key is like the destination address to which the value should be delivered. Even though the key is just an arbitrary string (not a physical network address like an IP address and port number), it acts like an address: all key-value pairs with the same key will be delivered to the same destination (a call to the reduce function).

Using the MapReduce programming model has separated the physical network communication aspects of the computation (getting the data to the right machine) from the application logic (processing the data once you have it). This is in contrast to the typical use of databases, where a request to fetch data from a database often occurs somewhere deeply inside a piece of application code [36]. Since MapReduce handles all network communication, it also shields the application code from having to worry about partial failures, such as the crash of another node: MapReduce transparently retries failed tasks without affecting the application logic.

GROUP BY

Besides joins, another common use of the "bringing related data to the same place" pattern is grouping records by some key (as in the `GROUP BY` clause in SQL). All records with the same key form a group, and the next step is often to perform some kind of aggregation within each group: for example, to count the number of records in each group (`COUNT(*)` in SQL), to add up the values in one particular field of the grouped records (`SUM(fieldname)`), to pick the top *k* records according to some ranking function, etc.

The simplest way of implementing such a grouping operation with MapReduce is to set up the mappers so that the key-value pairs they produce use the desired grouping key. The partitioning and sorting process then brings together all the records with the same key in the same reducer. Thus, grouping and joining look quite similar when implemented on top of MapReduce.

Another common use for grouping is collating all the activity events for a particular user session, in order to find out the sequence of actions that the user took — a process called *sessionization* [37]. For example, such analysis could be used to work out whether users who were shown a new version of your website are more likely to make a purchase than those who were shown the old version (A/B testing), or to calculate whether some marketing activity is worthwhile.

If you have multiple web servers handling user requests, the activity events for a particular user are most likely scattered across various different servers' log files. You can implement sessionization by using a session cookie, user ID or similar identifier as grouping key, and bringing all the activity events for a particular user together in one place, while distributing different users' events across different partitions.

Handling skew

The pattern of "bringing all records with the same key to the same place" breaks down if there is a very large amount of data related to a single key. For example, in a social network, most users might be connected to a few hundred people at most, but a small number of celebrities may have many millions of followers. Such disproportionately active database records are known as *linchpin objects* [38].

Collecting all activity related to a celebrity (e.g. replies to something they posted) in a single reducer can lead to significant *skew*, that is, one reducer that must process significantly more records than the others (see "Skewed workloads and relieving hot spots" on page 196). Since a MapReduce job is only complete when all of its mappers and reducers have completed, any subsequent jobs must wait for the slowest reducer to complete before they can start.

To compensate for skewed input to a join, you can change the partitioning of mapper outputs: when processing the skewed input files, send records relating to a linchpin object to a *random* reducer (in contrast to conventional MapReduce, which chooses a reducer deterministically based on a hash of the key). For the other input to the join, records relating to a linchpin object need to be sent to *all* reducers [39].

This technique is known as *skew join* in Pig and Hive, and *sharded join* in Crunch. It ensures that each reducer receives some of the records from the skewed input, and also the record from the input with which it is being joined. Thus, the burden of processing linchpin objects is evenly shared among the reducers, at the cost of having to replicate the other join input to multiple reducers.

When grouping records by a skewed key, you can handle skewed input by performing the grouping in two stages. The first MapReduce stage sends records to a random reducer, so that each reducer performs the grouping on a subset of records for the skewed key, and outputs a more compact aggregated value per key. The second MapReduce job then combines the values from all of the first-stage reducers into a single value per key.

Map-side joins

The join algorithms described in the last section perform the actual join logic in the reducers, and are hence known as *reduce-side joins*. The mappers take the role of preparing the input data: extracting the key and value from each input record, assigning the key-value pairs to a reducer partition, and sorting by key.

The reduce-side approach has the advantage that you do not need to make any assumptions about the input data: whatever its properties and structure, the mappers can prepare the data to be ready for joining. However, the downside is that all that sorting, copying to reducers, and merging of reducer inputs can be quite expensive. Depending on the available memory buffers, data may be written to disk several times as it passes through the stages of MapReduce [37].

On the other hand, if you *can* make certain assumptions about your input data, it is possible to make joins faster by using a so-called *map-side join*. It uses a cut-down MapReduce job in which there are no reducers and no sorting. Instead, each mapper simply reads one input file block from HDFS and writes one output file to HDFS — that is all.

Broadcast hash joins

The simplest way of performing a map-side join applies in the case where a large dataset is joined with a small dataset. In particular, the small dataset needs to be small enough that it can be loaded entirely into memory in each of the mappers.

For example, imagine in the case of Figure 10-2 that the user database is small enough to fit in memory. In this case, when a mapper starts up, it would first read the user database from HDFS into an in-memory hash table. Once this is done, the mapper can scan over the user activity events, and simply look up the user ID for each event in the hash table.[vi]

There can still be several mapper tasks, one for each file block of the large input to the join (in the example of Figure 10-2, the activity events are the large input). Each of these mappers loads the small input entirely into memory.

This simple but effective algorithm is called a *broadcast hash join*: the word *broadcast* reflects the fact that each mapper for a partition of the large input reads the entirety of the small input (so the small input is effectively 'broadcast' to all partitions of the large input), and the word *hash* reflects its use of a hash table. This join method is supported by Pig (under the name "replicated join"), Hive ("MapJoin"), Cascading, and Crunch. It is also used in data warehouse query engines such as Impala [40].

Instead of loading the small join input into an in-memory hash table, an alternative is to store the small join input in a read-only index on the local disk [41]. The frequently-used parts of this index will remain in the operating system's page cache, so this approach can provide random-access lookups almost as fast as an in-memory hash table, but without actually requiring the dataset to fit in memory.

Partitioned hash joins

If the inputs to the map-side join are partitioned in the same way, then the hash join approach can be applied to each partition independently. In the case of Figure 10-2, you might arrange for both the activity events and the user database to each be partitioned based on the last decimal digit of the user ID (so there are ten partitions on either side). For example, mapper 3 first loads all users with an ID ending in 3 into a hash table, and then scans over all the activity events for users whose ID ends in 3.

If the partitioning is done correctly, you can be sure that all the records you might want to join are located in the same-numbered partition, and so it is sufficient for each mapper to only read one partition from each of the input datasets. This has the advantage that each mapper only needs to load a smaller amount of data into its hash table.

This approach only works if both of the join's inputs have the same number of partitions, with records assigned to partitions based on the same key and the same hash function. If the inputs are generated by prior MapReduce jobs that already perform this grouping, then this can be a reasonable assumption to make.

Partitioned hash joins are known as *bucketed map joins* in Hive [37].

[vi] This example assumes that there is exactly one entry for each key in the hash table, which is probably true with a user database (a user ID uniquely identifies a user). In general, the hash table may need to contain several entries with the same key, and the join operator would output all matches for a key.

Map-side merge joins

Another variant of a map-side join applies if the input datasets are not only partitioned in the same way, but also *sorted* based on the same key. In this case, it does not matter whether the inputs are small enough to fit in memory, because a mapper can perform the same merging operation that would normally be done by a reducer: reading both input files incrementally, in order of ascending key, and matching records with the same key.

If a map-side merge join is possible, that probably means that prior MapReduce jobs brought the input datasets into this partitioned and sorted form in the first place. In principle, this join could have been placed in the reduce stage of the prior job. However, it may still be appropriate to perform the merge join in a separate map-only job, for example if the partitioned and sorted datasets are also needed for other purposes besides this particular join.

MapReduce workflows with map-side joins

When the output of a MapReduce join is consumed by downstream jobs, the choice of map-side or reduce-side join affects the structure of the output. The output of a reduce-side join is partitioned and sorted by the join key, whereas the output of a map-side join is partitioned and sorted in the same way as the large input (since one map task is started for each file block of the join's large input, regardless of whether a partitioned or broadcast join is used).

As discussed, map-side joins also make more assumptions about the size, sorting and partitioning of their input datasets. Knowing about the physical data layout of datasets in HDFS becomes important when optimizing join strategies: it is not sufficient to just know the encoding format and the name of the directory in which the data is stored, but also the number of partitions, and the keys by which it is partitioned and sorted.

In the Hadoop ecosystem, this kind of metadata about the partitioning of datasets is often maintained in HCatalog and the Hive metastore [37].

The output of batch workflows

We have talked a lot about the various algorithms for implementing workflows of MapReduce jobs, but we neglected an important question: what is the result of all of that processing, once it is done? Why are we running all these jobs in the first place?

In the case of database queries, we distinguished transaction-processing (OLTP) purposes from analytic purposes (see "Transaction Processing or Analytics?" on page 87). We saw that OLTP queries generally look up a small number of records by key, using indexes, in order to present them to a user, for example on a web page. On the other hand, analytic queries often scan over a large number of records, performing groupings and aggregations, and the output often has the form of a report: a graph showing the change of a metric over time, or the top 10 items according to some ranking, or a breakdown of some quantity into subcategories. The consumer of such a report is often an analyst or a manager who needs to make business decisions.

Where does batch processing fit in? It is neither transaction processing, nor is it analytics. It is closer to analytics, in that a batch process typically scans over large portions of an input dataset. However, a workflow of MapReduce jobs is not the same as a SQL query used for analytic purposes (see "Comparing MapReduce to distributed databases" on page 402). The output of a batch process is often not a report, but some other kind of structure.

Building search indexes

Google's original use of MapReduce was to build indexes for their search engine, which was implemented as a workflow of five to ten MapReduce jobs [1]. Although they later moved away from using MapReduce for this purpose [42], it helps to understand MapReduce if you look at it through the lens of building a search index. (Even today, Hadoop MapReduce remains a good way of building indexes for Lucene/Solr [43].)

We saw briefly in "SSTables and LSM-trees" on page 74 how a full-text search index such as Lucene works: it is a file in which you can efficiently look up a particular keyword (the term dictionary), and find the list of all the document IDs containing that keyword (the postings list). This is a very simplified view of a search index — in reality it requires various additional data, in order to rank search results by relevance, correct misspellings, resolve synonyms, and so on — but the principle holds.

If you need to perform full-text search over a fixed set of documents, then a batch process is a very effective way of building the indexes: the mappers partition the set of documents as needed, each reducer builds the index for its partition, and the index files are written to HDFS. Building such document-partitioned indexes (see "Partitioning and secondary indexes" on page 197) parallelizes very well. Since querying a search index by keyword is a read-only operation, these index files are immutable once they have been created.

If the indexed set of documents changes, one option is to periodically re-run the entire indexing workflow for the entire set of document, and replace the previous index files wholesale with the new index files when it is done. That can be computationally expensive if only a small number of documents have changed, but it has the advantage that the indexing process is very easy to reason about: documents in, indexes out.

Alternatively, it is possible to build indexes incrementally. As discussed in Chapter 3, if you want to add, remove or update documents in an index, Lucene does this by writing out new segment files, and asynchronously merging and compacting segment files in the background. We will see more on such incremental processing in Chapter 11.

Key-value stores as batch process output

Search indexes are just one example of the possible outputs of a batch processing workflow. Another common use for batch processing is to build machine learning systems such as classifiers (e.g. spam filters, anomaly detection, image recognition) and recommendation systems (e.g. people you may know, products you may be interested in, or related searches [29]).

The output of those batch jobs is often some kind of database: for example, a database that can be queried by user ID to obtain suggested friends for that user, or a database that can be queried by product ID to get a list of related products [44].

These databases need to be queried from the web application that handles user requests, which is usually separate from the Hadoop infrastructure. So how does the output from the batch process get back into a database where the web application can query it?

The most obvious choice might be to use the client library for your favorite database directly within a mapper or reducer, and to write from the batch job directly to the database server, one record at a time. This will work (assuming your firewall rules allow direct access from your Hadoop environment to your production databases), but it is a bad idea for several reasons:

- As discussed previously in the context of joins, making a network request for every single record is orders of magnitude slower than the normal throughput of a batch task. Even if the client library supports batching, performance is likely to be poor.
- MapReduce jobs often run many tasks in parallel. If all the mappers or reducers concurrently write to the same output database, with a rate expected of a batch process, that database can easily be overwhelmed, and its performance for queries is likely to suffer. This can in turn cause operational problems in other parts of the system [35].
- Normally, MapReduce provides a clean all-or-nothing guarantee for job output: if a job succeeds, the result is the output of running every task exactly once, even if some tasks failed and had to be retried along the way; if the entire job fails, no output is produced. However, writing to an external system from inside a job produces externally visible side-effects which cannot be hidden in this way. Thus, you have to worry about the results from partially completed jobs being visible to other systems, and the complexities of Hadoop task attempts and speculative execution.

A much better solution is to build a brand-new database *inside* the batch job, and write it as files to the job's output directory in HDFS, just like the search indexes in the last section. Those data files are then immutable once written, and can be loaded in bulk into servers that handle read-only queries. Various key-value stores support building database files in MapReduce jobs, including Voldemort [45], Terrapin [46], ElephantDB [47], and HBase bulk loading [48].

Building these database files is a good use of MapReduce: using a mapper to extract a key, and then sorting by that key, is already a lot of the work required to build an index. Since most of these key-value stores are read-only (the files can only be written once by a batch job and are then immutable), the data structures are quite simple. For example, they do not require a WAL (see "Update-in-place vs. append-only logging" on page 80).

When loading data into Voldemort, the server continues serving requests to the old data files while the new data files are copied from HDFS to the server's local disk. Once the copying is complete, the server atomically switches over to querying the new files. If anything goes wrong in this process, it can easily switch back to the old files again, since they are still there and immutable [45].

Philosophy of batch process outputs

The Unix philosophy that we discussed earlier in this chapter ("The Unix philosophy" on page 382) encourages experimentation by being very explicit about dataflow: a program reads its input and writes its output. In the process, the input is left unchanged, any previous output is completely replaced with the new output, and there are no other side-effects. This means that you can re-run a command as often as you like, tweaking or debugging it, without messing up the state of your system.

The handling of output from MapReduce jobs follows the same philosophy. By treating inputs as immutable and avoiding side-effects (such as writing to external databases), batch jobs not only achieve good performance, but also become much easier to maintain:

- If you introduce a bug into the code and the output is wrong or corrupted, you can simply roll back to a previous version of the code, re-run the job, and the output will be correct again. Or even simpler, you can keep the old output in a different directory, and simply switch back to it. Databases with read/write transactions do not have this property: if you deploy buggy code that writes bad data to the database, then rolling back the code will do nothing to fix the data in the database. (The idea of being able to recover from buggy code has been called *human fault tolerance* [49].)
- As a consequence of this ease of rolling back, feature development can proceed more quickly than in an environment where mistakes could mean irreversible

damage. This principle of *minimizing irreversibility* is beneficial for agile software development [50].

- If a map or reduce task fails, the MapReduce framework automatically re-schedules it and runs it again on the same input. If the failure is due to a bug in the code, it will keep crashing and eventually cause the job to fail after a few attempts, but if the failure is due to a transient issue, the fault is tolerated. This automatic retry is only safe because inputs are immutable, and outputs from failed tasks are discarded by the MapReduce framework.
- The same set of files can be used as input for various different jobs, including monitoring jobs that calculate metrics and evaluate whether a job's output has the expected characteristics (for example, by comparing it to the output from the previous run, and measuring discrepancies).
- Like Unix tools, MapReduce jobs separate logic from wiring (configuring the input and output directories), which provides a separation of concerns and potential reuse of code: one team can focus on implementing a job that does one thing well, while other teams can decide where and when to run that job.

In these areas, the design principles that worked well for Unix also seem to be working well for Hadoop. But Unix and Hadoop also differ in some ways: for example, because most Unix tools assume untyped text files, everything has to do a lot of input parsing (our log analysis example at the beginning of the chapter used `{print $7}` to extract the URL).

On Hadoop, some of those low-value syntactic conversions are eliminated by using more structured file formats: Avro (see "Avro" on page 118) and Parquet (see "Column-oriented storage" on page 93) are often used, as they provide efficient schema-based encoding, and allow evolution of their schemas over time (see Chapter 4).

Comparing MapReduce to distributed databases

At a most abstract level, databases, Hadoop, and operating systems all perform the same functions: they store some data, and they allow you to process and query that data [51]. A database stores data in records of some data model (rows in tables, documents, vertices in a graph, etc.) while an operating system's filesystem stores data in files — but at their core, both are "information management" systems [52].

As we have seen, Hadoop is somewhat like a distributed version of Unix, where HDFS is the filesystem, and MapReduce is a quirky implementation of a Unix process (which happens to always run the `sort` utility between the map phase and the reduce phase). We saw how you can implement various join and grouping operations on top of these primitives.

When the MapReduce paper [1] was published, it was — in some sense — not at all new. All of the processing and parallel join algorithms that we discussed in the last few sections had already been implemented in so-called *massively parallel processing* (MPP) databases more than a decade previously [3, 39]. For example, the Gamma database machine, Teradata, and Tandem NonStop SQL were pioneers in this area [53].

The biggest difference is that MPP databases focus on parallel execution of analytic SQL queries on a cluster of machines, while the combination of MapReduce and a distributed filesystem [19] provides something much more like a general-purpose operating system that can run arbitrary programs.

Diversity of storage

Databases require you to structure data according to a particular model (e.g. relational or documents), whereas files in a distributed filesystem are just byte sequences, which can be written using any data model and encoding. They might be collections of database records, but they can equally well be images, videos, sensor readings, sparse matrices, feature vectors, genome sequences, or any other kind of data.

To put it bluntly, Hadoop opened up the possibility of indiscriminately dumping data into HDFS, and only later figuring out how to process it further [54]. By contrast, MPP databases typically require careful up-front modeling of the data and query patterns before importing the data into the database's proprietary storage format.

From a purist point of view, it may seem that this careful modeling and import is desirable, because it means users of the database have better-quality data to work with. However, in practice, it appears that simply making data available quickly — even if it is in a quirky, difficult-to-use, raw format — is often more valuable than trying to decide on the ideal data model up-front [55].

The idea is similar to a data warehouse (see "Data warehousing" on page 88): simply bringing data from various parts of a large organization together in one place is valuable, because it enables joins across datasets that were previously disparate. The careful schema design required by a MPP database slows down that centralized data collection; collecting data in its raw form, and worrying about schema design later, allows the data collection to be speeded up (a concept sometimes known as a "data lake" or "enterprise data hub" [56]).

Indiscriminate data dumping shifts the burden of interpreting the data: instead of forcing the producer of a dataset to bring it into a standardized format, the interpretation of the data becomes the consumer's problem (the schema-on-read approach, see "Schema flexibility in the document model" on page 39). This can be an advantage if the producer and consumers are different teams with different priorities. There may not even be one ideal data model, but rather different views onto the data that are suitable for different purposes. Simply dumping data in its raw form allows for several such transformations. This approach has been dubbed the *sushi principle*: "raw data is better" [57].

Thus, Hadoop has often been used for implementing ETL processes (see "Data warehousing" on page 88): data from transaction-processing systems is dumped into HDFS in some raw form, and then MapReduce jobs are written to clean up that data, transform it into a relational form, and import it into a MPP data warehouse for analytic purposes. Data modeling still happens, but it is in a separate step, decoupled from the data collection. This is possible because HDFS supports data encoded in any format.

Diversity of processing models

MPP databases are monolithic, tightly-integrated pieces of software that take care of storage layout on disk, query planning, scheduling and execution. Since these components can all be tuned and optimized for the specific needs of the database, the system as a whole can achieve very good performance on the types of queries for which it is designed. Moreover, the SQL query language allows expressive queries and elegant semantics without having to write code, making it accessible to graphical tools used by business analysts, such as Tableau.

On the other hand, not all kinds of processing can be sensibly expressed as SQL queries. For example, if you are building machine learning and recommendation systems, or full-text search indexes with relevance ranking models, or performing image analysis, you most likely need a more general model of data processing. These kinds of processing are often very specific to a particular application (e.g. feature engineering for machine learning, natural language models for machine translation, risk estimation functions for fraud prediction), so they inevitably require writing code, not just queries.

MapReduce gave engineers the ability to easily run their own code over large datasets. If you have HDFS and MapReduce, you *can* build a SQL query execution engine on top of it, and indeed this is what the Hive project did [31]. However, you can also write many other forms of batch process that do not lend themselves to being expressed as a SQL query.

Subsequently, people found that MapReduce was too limiting and performed too badly for some types of processing, so various other processing models were developed on top of Hadoop (we will see some of them in "Beyond MapReduce" on page 406). Having two processing models, SQL and MapReduce, was not enough: even more different models were needed! And due to the openness of the Hadoop platform, it was feasible to implement a whole range of different approaches, which would not have been possible within the confines of a monolithic MPP database [58].

Crucially, those various different processing models can be run on a single shared-use cluster of machines, all accessing the same files on HDFS. In the Hadoop approach, there is no need to import the data into several different specialized systems for different kinds of processing: the system is flexible enough to support a diverse set of workloads within the same cluster. Not having to move data around makes it a lot easier to derive value from the data, and a lot easier to experiment with new processing models.

The Hadoop ecosystem includes both random-access OLTP databases such as HBase (see "SSTables and LSM-trees" on page 74), and also MPP-style analytic databases such as Impala [40]. Neither HBase nor Impala uses MapReduce, but both use HDFS for storage. They are very different approaches to accessing and processing data, but they can nevertheless coexist and be integrated in the same system.

Designing for frequent faults

When comparing MapReduce to MPP databases, two more differences in design approach stands out: the handling of faults, and the use of memory and disk. Batch processes are less sensitive to faults than online systems, because they do not immediately affect users if they fail, and they can always be run again.

If a node crashes while a query is executing, most MPP databases abort the entire query, and either let the user resubmit the query or automatically run it again [3]. As queries normally run for a few seconds or a few minutes at most, this is acceptable, since the cost of retrying is not too great. MPP databases also prefer to keep as much data as possible in memory (e.g. using hash joins), to avoid the cost of reading from disk.

On the other hand, MapReduce can tolerate the failure of a map or reduce task without affecting the job as a whole, by retrying work at the granularity of an individual task. It is also very eager to write data to disk, partly for fault tolerance, and partly on the assumption that the dataset will be too big to fit in memory anyway.

The MapReduce approach is more appropriate for larger jobs: jobs that process so much data and run for such a long time that they are likely to experience at least one task failure along the way. In that case, re-running the entire job due to a single task failure would be wasteful. Even if recovery at the granularity of an individual task introduces overheads that make fault-free processing slower, it can still be a reasonable trade-off if the rate of task failures is high enough.

But how realistic are these assumptions? In most clusters, machine failures do occur, but they are not very frequent — probably rare enough that most jobs would not experience a machine failure. Is it really worth incurring significant overheads for the sake of fault tolerance?

To understand the reasons for MapReduce's sparing use of memory and task-level recovery, it is helpful to look at the environment for which MapReduce was originally designed. Google has mixed-use datacenters, in which online production services and offline batch jobs run on the same machines. Every task has a resource allocation (CPU cores, RAM, disk space, etc.) that is enforced using containers. Every task also has a priority, and if a higher-priority task needs more resources, lower-priority tasks on the same machine can be terminated (preempted) in order to free up resources. Priority also determines pricing of the computing resources: teams must pay for the resources they use, and higher-priority processes cost more [59].

This architecture allows non-production (low-priority) computing resources to be over-committed, because the system knows that it can reclaim the resources if necessary. This in turn allows better utilization of machines and greater efficiency compared to systems that segregate production and non-production tasks. However, as MapReduce jobs run at low priority, they run the risk of being preempted at any time because a higher-priority process requires its resources. Batch jobs effectively "pick up the scraps under the table", using any computing resources that remain after the high-priority processes have taken what they need.

At Google, a MapReduce task that runs for an hour has an approximately 5% risk of being terminated to make space for a higher-priority process. This rate is more than an order of magnitude higher than the rate of failures due to hardware issues, machine reboot or other reasons [59]. At this rate of preemptions, if a job has 100 tasks that run for 10 minutes, there is a risk greater than 50% that at least one task will be terminated before it is finished.

And this is why MapReduce is designed to tolerate frequent unexpected task termination: it's not because the hardware is particularly unreliable, it's because the freedom to arbitrarily terminate processes enables better resource utilization in a computing cluster.

Among open source cluster schedulers, preemption is less widely used. YARN's CapacityScheduler supports preemption for balancing the resource allocation of different queues [58], but general priority preemption is not supported in YARN, Mesos or Kubernetes at the time of writing [60]. In an environment where tasks are not so often terminated, the design decisions of MapReduce make less sense. In the next section, we will look at some alternatives to MapReduce that make different design decisions.

Beyond MapReduce

Although MapReduce became very popular and received a lot of hype in the late 2000s, it is just one among many possible programming models for distributed systems. Depending on the volume of data, the structure of the data, and the type of processing being done with it, other tools may be more appropriate for expressing a computation.

We nevertheless spent a lot of time in this chapter discussing MapReduce because it is a useful learning tool, as it is a fairly clear and simple abstraction on top of a distributed filesystem. That is, *simple* in the sense of being able to understand what it is doing, not in the sense of being easy to use. Quite the opposite: implementing a complex processing job using the raw MapReduce APIs is actually quite hard and laborious — for instance, you would need to implement the above join algorithms from scratch [37].

In response to the difficulty of using MapReduce directly, various higher-level programming models (Pig, Hive, Cascading, Crunch) were created as abstractions on top of MapReduce. If you understand how MapReduce works, they are fairly easy to learn, and their higher-level constructs make many common batch processing tasks significantly easier to implement.

However, there are also problems with the MapReduce execution model itself, which are not fixed by adding another level of abstraction, and which manifest themselves as poor performance for some kinds of processing. On the one hand, MapReduce is very robust: you can use it to process almost arbitrarily large quantities of data on an unreliable multitenant system with frequent task terminations, and it will still get the job done, albeit slowly. On the other hand, other tools are sometimes orders of magnitude faster for some kinds of processing.

In the rest of this chapter, we will look at some of those alternatives for batch processing. In Chapter 11 we will move to stream processing, which can be regarded as another way of speeding up batch processing.

Materialization of intermediate state

As discussed previously, every MapReduce job is independent from every other job. The main contact points of a job with the rest of the world are its input and output directories on HDFS. If you want the output of one job to become the input to a second job, you need to configure the second job's input directory to be the same as the first job's output directory, and an external workflow scheduler must start the second job only once the first job has completed.

This setup is reasonable if the output from the first job is a dataset that you want to publish widely within your organization. In that case, you need to be able to refer to it by name, and reuse it as input to several different jobs (including jobs developed by other teams). Publishing data to a well-known location in HDFS allows loose coupling, so that jobs don't need to know who is producing their input or consuming their output (see "Separation of logic and wiring" on page 384).

However, in many cases, you know that the output of one job is only ever used as input to one other job, which is maintained by the same team. In this case, the files on HDFS are simply *intermediate state*: a means of passing data from one job to the next. In the complex workflows used to build recommendation systems, consisting of 50 or 100 MapReduce jobs [29], there is a lot of such intermediate state.

The process of writing out this intermediate state to HDFS is called *materialization*. (We came across the term previously in the context of materialized views, in "Aggregation: Data cubes and materialized views" on page 98. It means to eagerly compute the result of some operation and to write it out, rather than computing it on demand when requested.)

By contrast, the log analysis example at the beginning of the chapter used Unix pipes to connect the output of one command with the input of another. Pipes do not fully materialize the intermediate state, but instead *stream* the output to the input incrementally, using only a small in-memory buffer.

MapReduce's approach of fully materializing intermediate state has downsides compared to Unix pipes:

- A MapReduce job can only start when all tasks in the preceding jobs (that generate its inputs) have completed, whereas processes connected by a Unix pipe are started at the same time, with output being consumed as soon as it is produced. Skew or varying load on different machines mean that a job often has a few straggler tasks that take much longer to complete than the others. Having to wait until all of the preceding job's tasks have completed slows down the execution of the workflow as a whole.
- Mappers are often redundant: they just read back the same file that was just written by a reducer, and prepare it for the next stage of partitioning and sorting. In many cases, the mapper code could be part of the previous reducer: if the reducer output was partitioned and sorted in the same way as mapper output, then reducers could be chained together directly, without interleaving with mapper stages.

Dataflow engines

In order to fix these problems with MapReduce, several new execution engines for distributed batch computations were developed, the most well-known of which are Spark [61, 62], Tez [63, 64] and Flink [65, 66]. There are various differences in the way they are designed, but they have one thing in common: they handle an entire workflow as one job, rather than breaking it up into independent sub-jobs.

Since they explicitly model the flow of data through several processing stages, these systems are known as *dataflow engines*. Like MapReduce, they work by repeatedly calling a user-defined function to process one record at a time on a single thread. They parallelize work by partitioning inputs, and they copy the output of one function over the network to become the input to another function.

The difference is that these functions need not take the strict roles of alternating map and reduce, but instead can be assembled in more flexible ways. We call these functions *operators*, and the dataflow engine provides several different options for connecting one operator's output to another's input:

- One option is to re-partition and sort records by key, like in the shuffle stage of MapReduce (see "Distributed execution of MapReduce" on page 388). This enables sort-merge joins and grouping in the same way as MapReduce.
- Another possibility is to take several inputs and to partition them in the same way, but to skip the sorting. This saves effort on partitioned hash joins, where the partitioning of records is important but the order is irrelevant, because building the hash table randomizes the order anyway.
- For broadcast hash joins, the same output from one operator can be sent to all partitions of the join operator.

This style of processing engine is based on research systems like Dryad [67] and Nephele [68], and it offers several advantages compared to the MapReduce model:

- Expensive work such as sorting need only be inserted in places where it is actually required, rather than always happening by default between every map and reduce stage.
- This approach avoids unnecessary map tasks, since the work done by a mapper can often be incorporated into the preceding reduce operator (because a mapper does not change the partitioning of a dataset).
- Because all joins and data dependencies in a workflow are explicitly declared, the scheduler has an overview of what data is required where, so it can make locality optimizations. For example, it can try to place the task that consumes some data on the same machine as the task that produces it, so that the data can be exchanged through a shared memory buffer, rather than having to copy it over the network.
- It is usually sufficient for intermediate state between operators to be kept in memory or written to local disk, which requires less I/O than writing it to HDFS (where it must be replicated to several machines and written to disk on each replica). MapReduce already does this for mapper output, but dataflow engines generalize the idea to all intermediate state.
- Operators can start executing as soon as their input is ready; there is no need to wait for the entire preceding stage to finish before the next one starts.
- Existing JVM processes can be reused to run new operators, which reduces startup overheads.

Dataflow engines can be used to implement the same computations as MapReduce workflows, and they usually execute significantly faster due to the optimizations above. Since operators are a generalization of map and reduce, the same processing code can run on either execution engine: workflows implemented in Pig, Hive or Cascading can be switched from MapReduce to Tez with a simple configuration change, without modifying code [64].

Tez is a fairly thin library that relies on the YARN shuffle service for the actual copying of data between nodes [58], whereas Spark and Flink are big frameworks that include their own network communication, scheduling, and user-facing APIs. We will discuss those high-level APIs shortly.

Fault tolerance

An advantage of fully materializing intermediate state to HDFS is that it is durable, which makes fault tolerance fairly easy in MapReduce: if a task fails, it can just be restarted on another machine, and read the same input again from HDFS.

Spark, Flink and Tez avoid writing intermediate state to HDFS, so they take a different approach to tolerating faults: if a machine fails and the intermediate state on that machine is lost, it is recomputed from other data that is still available (a prior intermediary stage if possible, or otherwise the original input data, which is normally on HDFS).

To enable this recomputation, the framework must keep track of how a given piece of data was computed — which input partitions it used, and which operators were applied to it. Spark uses the RDD ("resilient distributed dataset") abstraction for tracking the ancestry of data [61], while Flink checkpoints operator state, allowing it to resume running an operator that ran into a fault during its execution [66].

When recomputing data, it is important to know whether the computation is *deterministic*: that is, given the same input data, do the operators always produce the same output? This matters if some of the lost data has already been sent to downstream operators. If the operator is restarted and the recomputed data is not the same as the original lost data, it becomes very hard for downstream operators to resolve the contradictions between the old and new data. The solution in the case of non-deterministic operators is normally to kill the downstream operators as well, and run them again on the new data.

In order to avoid such cascading faults, it is better to make operators deterministic. Note however that it is easy for non-deterministic behavior to accidentally creep in: for example, many programming languages do not guarantee any particular order when iterating over elements of a hash table, many probabilistic and statistical algorithms explicitly rely on using random numbers, and any use of the system clock or external data sources is non-deterministic. Such causes of non-determinism need to be removed in order to reliably recover from faults, for example by generating pseudo-random numbers using a fixed seed.

Recovering from faults by recomputing data is not always the right answer: if the intermediate data is much smaller than the source data, or if the computation is very CPU-intensive, it is probably cheaper to materialize the intermediate data to HDFS than to recompute it.

Discussion of materialization

Returning to the Unix analogy, we saw that MapReduce is like writing the output of each command to a temporary file, whereas dataflow engines look much more like Unix pipes. Flink is especially built around the idea of pipelined execution, that is, incrementally passing the output of an operator to other operators, and not waiting for the input to be complete before starting to process it.

A sorting operation inevitably needs to consume its entire input before it can produce any output — because it's possible that the very last input record is the one with the lowest key, and thus needs to be the very first output record. Any operator that requires sorting will thus need to accumulate state, at least temporarily. But many other parts of a workflow can be executed in a pipelined manner.

When the job completes, its output needs to go somewhere durable so that users can find it and use it — most likely, it is written to HDFS again. Thus, when using a dataflow engine, materialized datasets on HDFS are still usually the inputs and the final outputs of a job. Like with MapReduce, the inputs are immutable and the output is completely replaced. The improvement over MapReduce is that you save yourself writing all the intermediate state to HDFS as well.

Graphs and iterative processing

In "Graph-like Data Models" on page 48 we discussed using graphs for modelling data, and using graph query languages to traverse the edges and vertices in a graph. The discussion in Chapter 2 was focused around OLTP-style use: quickly executing queries to find a small number of vertices matching certain criteria.

It is also interesting to look at graphs in a batch processing context, where the goal is to perform some kind of offline processing or analysis on an entire graph. This need often arises in machine learning applications such as recommendation engines, or in ranking systems. For example, one of the most famous graph analysis algorithms is PageRank [69], which tries to estimate the popularity of a web page based on what other web pages link to it. It is used as part of the formula that determines the order in which a web search engine presents its results.

> Dataflow engines like Spark, Flink and Tez (see "Materialization of intermediate state" on page 407) typically arrange the operators in a job as a directed acyclic graph (DAG). This is not the same as graph processing: in dataflow engines, the *flow of data from one operator to another* is structured as a graph, while the data itself typically consists of relational-style tuples. In graph processing, the *data itself* has the form of a graph. Another unfortunate naming confusion!

Many graph algorithms are expressed by traversing one edge at a time, joining one vertex with an adjacent vertex in order to propagate some information, and repeating until some condition is met — for example, until there are no more edges to follow, or until some metric converges. We previously saw an example in Figure 2-6, which made a list of all the locations in North America by repeatedly following edges indicating which location is within which other location (this kind of algorithm is called a *transitive closure*).

It is possible to store a graph in HDFS (in files containing lists of vertices and edges), but this idea of "repeating until done" cannot be expressed in plain MapReduce, since it only performs a single pass over the data. This kind of algorithm is thus often implemented in an *iterative* style:

1. An external scheduler runs a batch process to calculate one step of the algorithm;
2. When the batch process completes, the scheduler checks whether it has finished (based on the completion condition, e.g. there are no more edges to follow, or the change compared to the last iteration is below some threshold);
3. If it has not yet finished, the scheduler goes back to step 1 and runs another round of the batch process.

This approach works, but implementing it with MapReduce is often very inefficient, because MapReduce does not account for the iterative nature of the algorithm: it will always read the entire input dataset and produce a completely new output dataset, even if only a small part of the graph has changed compared to the last iteration.

The Pregel processing model

As an optimization for batch processing graphs, the *bulk synchronous parallel* (BSP) model of computation [70] has become popular. Among others, it is implemented by Apache Giraph [37], Spark's GraphX API, and Flink's Gelly API [71]. It is also known as the *Pregel* model, as Google's Pregel paper popularized this approach for processing graphs [72].

Recall that in MapReduce, mappers conceptually "send a message" to a particular call of the reduce function, because the framework collects together all the mapper outputs with the same key. A similar idea is behind Pregel: one vertex can "send a message" to another vertex, and typically those messages are sent along the edges in a graph.

In each iteration, a function is called for each vertex, passing it all the messages that were sent to it — much like a call to the reduce function. The difference to MapReduce is that in the Pregel model, a vertex remembers its state in memory from one iteration to the next, so the function only needs to process new incoming messages. If no messages are being sent in some part of the graph, no work needs to be done.

It's a bit similar to the actor model (see "Distributed actor frameworks" on page 134), if you think of each vertex as an actor, except that vertex state and messages between vertices are fault-tolerant and durable, and that communication proceeds in fixed rounds: at every iteration, the framework delivers all messages sent in the previous iteration. Actors normally have no such timing guarantee.

Fault tolerance

The fact that vertices can only communicate by message passing (not by querying each other directly) helps improve the performance of Pregel jobs, since messages can be batched and there is less waiting for communication. The only waiting is between iterations: since the Pregel model guarantees that all messages sent in one iteration are delivered in the next iteration, the prior iteration must completely finish, and all of its messages must be copied over the network, before the next one can start.

Even though the underlying network may drop, duplicate or arbitrarily delay messages (see "Unreliable Networks" on page 269), Pregel implementations guarantee that messages are processed exactly once at their destination vertex in the following iteration. Like in MapReduce, the framework transparently recovers from faults, in order to simplify the programming model for algorithms on top of Pregel.

This fault tolerance is achieved by periodically checkpointing the state of all vertices at the end of an iteration, i.e. writing their full state to durable storage. If a node fails and its in-memory state is lost, the simplest solution is to roll back the entire graph computation to the last checkpoint and to restart the computation. If the algorithm is deterministic and messages are logged, it is also possible to selectively recover only the partition that was lost (like we previously discussed for dataflow engines) [72].

Parallel execution

A vertex does not need to know on which physical machine it is executing; when it sends messages to other vertices, it simply sends them to a vertex ID. It is up to the framework to partition the graph — i.e. to decide which vertex runs on which machine, and how to route messages over the network so that they end up in the right place.

Because the programming model deals with just one vertex at a time (sometimes called *think like a vertex*) the framework may partition the graph in arbitrary ways. Ideally it would be partitioned such that vertices are co-located on the same machine if they need to communicate a lot. However, finding such an optimized partitioning is hard — in practice, the graph is often simply partitioned by hash of vertex ID, making no attempt to group related vertices together.

As a result, graph algorithms often have a lot of cross-machine communication overhead, and the intermediate state (messages sent between nodes) is often bigger than the original graph. The overhead of sending messages over the network can significantly slow down distributed graph algorithms.

For this reason, if your graph can fit in memory on a single computer, it's quite likely that a single-machine (maybe even single-threaded) algorithm will outperform a distributed batch process [73, 74]. Even if the graph is bigger than memory, but can fit on the disks of a single computer, single-machine processing using a framework such as GraphChi is a viable option [75]. If the graph is too big to fit on a single machine, a distributed approach such as Pregel is unavoidable; efficiently parallelizing graph algorithms is an area of ongoing research [76].

High-level APIs and languages

Over the years since MapReduce first became popular, the execution engines for distributed batch processing have matured. By now, the infrastructure has become robust enough to store and process many petabytes of data on clusters of over 10,000 machines. As the problem of physically operating batch processes at such scale has been considered more or less solved, attention has turned to other areas: improving the programming model, improving the efficiency of processing, and broadening the set of problems that these technologies can solve.

As discussed previously, higher-level languages and APIs such as Hive, Pig, Cascading and Crunch became popular because programming MapReduce jobs by hand is quite laborious. As Tez emerged, these high-level languages had the additional benefit of being able to move to the new dataflow execution engine without having to rewrite job code. Spark and Flink also include their own high-level dataflow APIs, often taking inspiration from FlumeJava [34].

These dataflow APIs generally use relational-style building blocks to express a computation: joining datasets on the value of some field, grouping tuples by key, filtering by some condition, and aggregating tuples by counting, summing, or other functions. Internally, these operations are implemented using the various join and grouping algorithms that we discussed earlier in this chapter.

Besides the obvious advantage of requiring less code, these high-level interfaces also allow interactive use, in which you write analysis code incrementally in a shell, and run it frequently to observe what it is doing. This style of development is very helpful when exploring a dataset and experimenting with approaches for processing it. It is also reminiscent of the Unix philosophy, which we discussed in "The Unix philosophy" on page 382.

Moreover, these high-level interfaces not only make the humans using the system more productive, but they also improve the job execution efficiency at a machine level.

The move towards declarative query languages

An advantage of specifying joins as relational operators, rather than writing them out by hand, is that the framework can analyze the properties of the join inputs, and automatically decide which of the aforementioned join algorithms would be most suitable. Hive, Spark and Flink have cost-based query optimizers that can do this, and even change the order of joins so that the amount of intermediate state is minimized [66, 77, 78, 79].

The choice of join algorithm can make a big difference to the performance of a batch job, and it is nice not to have to understand and remember all the various join algorithms we discussed in this chapter. This is possible if joins are specified in a *declarative* way: the application simply states which joins are required, and the query optimizer decides how they should best be executed. We previously came across this idea in "Query Languages for Data" on page 42.

However, in other ways, MapReduce and its dataflow successors are very different from the fully declarative query model of SQL. MapReduce was built around the idea of function callbacks: for each record or group of records, a user-defined function (the mapper or reducer) is called, and that function is free to call arbitrary code in order to decide what to output. This approach has the advantage that you can draw upon a large ecosystem of existing libraries to do things like parsing, natural language analysis, image analysis, and numerical or statistical algorithms.

The freedom to easily run arbitrary code is what long distinguished batch processing systems of MapReduce heritage from MPP databases (see "Comparing MapReduce to distributed databases" on page 402); although databases have facilities for writing user-defined functions, they are often cumbersome to use, and not well integrated with package managers and dependency management systems that are widely used in most programming languages (such as Maven for Java, npm for JavaScript, Rubygems for Ruby, etc).

However, dataflow engines found that there are also advantages to incorporating more declarative features in other areas besides joins: for example, if a callback function contains only a simple filtering condition, or it just selects some fields from a record, then there is significant CPU overhead in calling the function on every record.

If such simple filtering and mapping operations are expressed in a declarative way, the query optimizer can take advantage of column-oriented storage layouts (see "Column-oriented storage" on page 93) and read only the required columns from disk. Hive, Spark DataFrames and Impala also use vectorized execution (see "Memory bandwidth and pipelined execution" on page 96): iterating over data in a tight inner loop that is friendly to CPU caches, and avoiding function calls. Spark generates JVM bytecode [79] and Impala uses LLVM to generate native code for these inner loops [40].

By incorporating declarative aspects in their high-level APIs, and having query optimizers that can take advantage of them during execution, batch processing frameworks begin to look more like MPP databases (and can achieve comparable performance). At the same time, by having the extensibility of being able to run arbitrary code and read data in arbitrary formats, they retain their flexibility advantage.

Specialization for different domains

While the extensibility of being able to run arbitrary code is useful, there are also many common cases where standard processing patterns keep reoccurring, and so it is worth having reusable implementations of the common building blocks. Traditionally, MPP databases have served the needs of business intelligence analysts and business reporting, but that is just one among many domains in which batch processing is used.

Another domain of increasing importance is statistical and numerical algorithms, which are needed for machine learning applications. Reusable implementations are emerging: for example, Mahout implements various algorithms for machine learning on top of MapReduce, Spark and Flink, while MADlib implements similar functionality inside a relational MPP database (Greenplum/HAWK) [55].

Also useful are spatial algorithms, for example *k-nearest neighbors* [80] which searches for items that are close to a given item in some multidimensional space — a kind of similarity search. Approximate search is also important for genome analysis algorithms, which need to find strings that are similar but not identical [81].

Batch processing engines are being used for distributed execution of algorithms from an increasingly wide range of domains. As batch processing systems gain built-in functionality and high-level declarative operators, and as MPP databases become more programmable and flexible, the two are beginning to look more alike: in the end, they are all just systems for storing and processing data.

Summary

In this chapter we explored the topic of batch processing. We started by looking at Unix tools such as `awk`, `grep` and `sort`, and we saw how the design philosophy of those tools is carried forward into MapReduce and more recent dataflow engines. Some of those design principles include: inputs are immutable, outputs are intended to become the input to another (as yet unknown) program, and complex problems are solved by composing small tools that "do one thing well".

In the Unix world, the uniform interface that allows one program to be composed with another is files and pipes; in MapReduce, that interface is a distributed filesystem. We saw that dataflow engines add their own pipe-like data transport mechanisms to avoid materializing intermediate state to the distributed filesystem, but the initial input and the final output of a job is still usually HDFS.

The the two main problem that distributed batch processing frameworks need to solve are partitioning and fault tolerance:

Partitioning

In MapReduce, mappers are partitioned according to input file blocks. The output of mappers is re-partitioned, sorted and merged into a configurable number of reducer partitions. The purpose of this process is to bring all the related data, e.g. all the records with the same key, together in the same place.

Post-MapReduce dataflow engines try to avoid sorting unless it is required, but they otherwise take a broadly similar approach to partitioning.

Fault tolerance

MapReduce frequently writes to disk, which makes it easy for a job to recover from an individual failed task without restarting the entire job, but which slows down execution in the failure-free case. Dataflow engines perform less materialization of intermediate state and keep more in memory, which means that they need to recompute more data if a node fails. Deterministic operators reduce the amount of data that needs to be recomputed.

We discussed several join algorithms for MapReduce, most of which are also internally used in MPP databases and dataflow engines. They also provide a good illustration of how partitioned algorithms work:

Sort-merge joins

Each of the inputs being joined goes through a mapper that extracts the join key. By partitioning, sorting and merging, all the records with the same key end up going to the same call of the reduce function. This function can then output the joined records.

Broadcast hash joins

One of the two join inputs is small, so it is not partitioned and it can be entirely loaded into a hash table. Thus, you can start a mapper for each partition of the large join input, load the hash table for the small input into each mapper, and then scan over the large input one record at a time, querying the hash table for each record.

Partitioned hash joins
If the two join inputs are partitioned in the same way (using the same key, same hash function, and same number of partitions), then the hash-table approach can be used independently for each partition.

Distributed batch processing engines have a deliberately restricted programming model: callback functions (such as mappers and reducers) are assumed to be stateless, and to have no externally visible side-effects besides their designated output. This allows the framework to hide some of the hard distributed systems problems: in the face of crashes and network issues, tasks can be retried safely, and the output from any failed tasks is discarded. If several tasks for a partition succeed, only one of them actually makes its output visible.

Therefore, your code in a batch processing job does not need to worry about implementing fault tolerance mechanisms: the framework can guarantee that the final output of a job is the same as if no faults had occurred, even though in reality various tasks perhaps had to be retried. These reliable semantics are much stronger than what you usually have in online services that handle user requests, and that write to databases as a side-effect of processing a request.

The distinguishing feature of a batch processing jobs is that it reads some input data and produces some output data, without modifying the input — in other words, the output is derived from the input. Crucially, the input data is *bounded*: it has a known, fixed size — for example, it consists of a set of log files at some point in time, or a snapshot of a database's contents. Because it is bounded, a job knows when it has finished reading the entire input, and so a job eventually completes when it is done.

In the next chapter, we will turn to stream processing, in which the input is *unbounded* — that is, you still have a job, but its inputs are never-ending streams of data. In this case, a job is never complete, because at any time there may still be more work coming in. We shall see that stream and batch processing are similar at some level, but the assumption of unbounded streams also changes a lot about how we build systems.

References

[1] 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.

[2] Joel Spolsky: "The Perils of JavaSchools," joelonsoftware.com, 25 December 2005.

[3] 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

[4] David J DeWitt and Michael Stonebraker: "MapReduce: A major step backwards," originally published at databasecolumn.vertica.com, 17 January 2008.

[5] Henry Robinson: "The Elephant was a Trojan Horse: On the Death of Map-Reduce at Google," the-paper-trail.org, 25 June 2014.

[6] "The Hollerith Machine," United States Census Bureau, census.gov.

[7] "IBM 82, 83, and 84 Sorters Reference Manual," Edition A24-1034-1, International Business Machines Corporation, July 1962.

[8] Adam Drake: "Command-line tools can be 235x faster than your Hadoop cluster," aadrake.com, 25 January 2014.

[9] "GNU Coreutils 8.23 Documentation," Free Software Foundation, Inc., 2014.

[10] Martin Kleppmann: "Kafka, Samza, and the Unix philosophy of distributed data," martin.kleppmann.com, 5 August 2015.

[11] Doug McIlroy: Internal Bell Labs memo, October 1964. Cited in: Dennis M Richie: "Advice from Doug McIlroy," cm.bell-labs.com.

[12] M D McIlroy, E N Pinson, and B A Tague: "UNIX Time-Sharing System: Foreword," *The Bell System Technical Journal*, volume 57, number 6, pages 1899–1904, July 1978.

[13] Eric S Raymond: *The Art of UNIX Programming*. Addison-Wesley, September 2003. ISBN: 0-13-142901-9

[14] Ronald Duncan: "Text File formats – ASCII Delimited Text – Not CSV or TAB delimited text," ronaldduncan.wordpress.com, 31 October 2009.

[15] Alan Kay: "Is "Software Engineering" an Oxymoron?," tinlizzie.org.

[16] Martin Fowler: "InversionOfControl," martinfowler.com, 26 June 2005.

[17] Daniel J Bernstein: "Two file descriptors for sockets," cr.yp.to.

[18] Rob Pike and Dennis M Ritchie: "The Styx Architecture for Distributed Systems," *Bell Labs Technical Journal*, volume 4, number 2, pages 146–152, April 1999.

[19] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: "The Google File System," at *19th ACM Symposium on Operating Systems Principles* (SOSP), October 2003. doi:10.1145/945445.945450

[20] Zhe Zhang, Andrew Wang, Kai Zheng, Uma Maheswara G, and Vinayakumar B: "Introduction to HDFS Erasure Coding in Apache Hadoop," blog.cloudera.com, 23 September 2015.

[21] Peter Cnudde: "Hadoop Turns 10," yahoohadoop.tumblr.com, 5 February 2016.

[22] Eric Baldeschwieler: "Thinking about the HDFS vs. Other Storage Technologies," hortonworks.com, 25 July 2012.

[23] Michael Ovsiannikov, Silvius Rus, Damian Reeves, et al.: "The Quantcast File System," *Proceedings of the VLDB Endowment*, volume 6, number 11, pages 1092–1101, August 2013. doi:10.14778/2536222.2536234

[24] "OpenStack Swift 2.6.1 developer documentation," OpenStack Foundation, docs.openstack.org, March 2016.

[25] Brendan Gregg: "Manta: Unix Meets Map Reduce," dtrace.org, 25 June 2013.

[26] Tom White: *Hadoop: The Definitive Guide*, 4th edition. O'Reilly Media, March 2015. ISBN: 978-1-4919-0163-2

[27] Jim N Gray: "Distributed Computing Economics," Microsoft Research Tech Report MSR-TR-2003-24, March 2003.

[28] Márton Trencséni: "Luigi vs Airflow vs Pinball," bytepawn.com, 6 February 2016.

[29] Roshan Sumbaly, Jay Kreps, and Sam Shah: "The "Big Data" Ecosystem at LinkedIn," at *ACM International Conference on Management of Data* (SIGMOD), July 2013. doi:10.1145/2463676.2463707

[30] Alan F Gates, Olga Natkovich, Shubham Chopra, et al.: "Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience," at *35th International Conference on Very Large Data Bases* (VLDB), August 2009.

[31] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, et al.: "Hive – A Petabyte Scale Data Warehouse Using Hadoop," at *26th IEEE International Conference on Data Engineering* (ICDE), March 2010. doi:10.1109/ICDE.2010.5447738

[32] "Cascading 3.0 User Guide," Concurrent, Inc., docs.cascading.org, January 2016.

[33] "Apache Crunch User Guide," Apache Software Foundation, crunch.apache.org.

[34] Craig Chambers, Ashish Raniwala, Frances Perry, et al.: "FlumeJava: Easy, Efficient Data-Parallel Pipelines," at *31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 363–375, June 2010. doi:10.1145/1806596.1806638

[35] Jay Kreps: "Why local state is a fundamental primitive in stream processing," oreilly.com, 31 July 2014.

[36] Martin Kleppmann: "Rethinking caching in web apps," martin.kleppmann.com, 1 October 2012.

[37] Mark Grover, Ted Malaska, Jonathan Seidman, and Gwen Shapira: *Hadoop Application Architectures*. O'Reilly Media, June 2015. ISBN: 978-1-4919-0004-8

[38] Philippe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan: "Challenges to Adopting Stronger Consistency at Scale," at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.

[39] David J DeWitt, Jeffrey F Naughton, Donovan A Schneider, and S Seshadri: "Practical Skew Handling in Parallel Joins," at *18th International Conference on Very Large Data Bases* (VLDB), pages 27–40, August 1992.

[40] 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.

[41] Matthieu Monsch: "Open-sourcing PalDB, a lightweight companion for storing side data," engineering.linkedin.com, 26 October 2015.

[42] Daniel Peng and Frank Dabek: "Large-scale Incremental Processing Using Distributed Transactions and Notifications," at *9th USENIX conference on Operating Systems Design and Implementation* (OSDI), October 2010.

[43] "Cloudera Search User Guide, Cloudera, Inc., September 2015.

[44] Lili Wu, Sam Shah, Sean Choi, Mitul Tiwari, and Christian Posse: "The Browsemaps: Collaborative Filtering at LinkedIn," at *6th Workshop on Recommender Systems and the Social Web* (RSWeb), October 2014.

[45] Roshan Sumbaly, Jay Kreps, Lei Gao, et al.: "Serving Large-scale Batch Computed Data with Project Voldemort," at *10th USENIX Conference on File and Storage Technologies* (FAST), February 2012.

[46] Varun Sharma: "Open-sourcing Terrapin: A serving system for batch generated data," engineering.pinterest.com, 14 September 2015.

[47] Nathan Marz: "ElephantDB," slideshare.net, 30 May 2011.

[48] Jean-Daniel (JD) Cryans: "How-to: Use HBase Bulk Loading, and Why," blog.cloudera.com, 27 September 2013.

[49] Nathan Marz: "How to beat the CAP theorem," nathanmarz.com, 13 October 2011.

[50] Molly Bartlett Dishman and Martin Fowler: "Agile Architecture," at *O'Reilly Software Architecture Conference*, March 2015.

[51] Dennis M Ritchie and Ken Thompson: "The UNIX Time-Sharing System," *Communications of the ACM*, volume 17, number 7, July 1974. doi:10.1145/361011.361061

[52] Eric A Brewer and Joseph M Hellerstein: "CS262a: Advanced Topics in Computer Systems," lecture notes, *University of California, Berkeley*, cs.berkeley.edu, August 2011.

[53] 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

[54] Jay Kreps: "But the multi-tenancy thing is actually really really hard," tweetstorm, twitter.com, 31 October 2014.

[55] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M Hellerstein, and Caleb Welton: "MAD Skills: New Analysis Practices for Big Data," *Proceedings of the VLDB Endowment*, volume 2, number 2, pages 1481–1492, August 2009. doi:10.14778/1687553.1687576

[56] Paige Roberts: "To Schema On Read or to Schema On Write, That is the Hadoop Data Lake Question," adaptivesystemsinc.com, 2 July 2015.

[57] Bobby Johnson and Joseph Adler: "The Sushi Principle: Raw Data Is Better," at *Strata+Hadoop World*, San Jose, CA, February 2015.

[58] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, et al.: "Apache Hadoop YARN: Yet Another Resource Negotiator," at *4th ACM Symposium on Cloud Computing* (SoCC), October 2013. doi:10.1145/2523616.2523633

[59] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, et al.: "Large-scale cluster management at Google with Borg," at *10th European Conference on Computer Systems* (EuroSys), April 2015. doi:10.1145/2741948.2741964

[60] Malte Schwarzkopf: "The evolution of cluster scheduler architectures," firmament.io, 9 March 2016.

[61] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, et al.: "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," at *9th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), April 2012.

[62] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia: *Learning Spark*. O'Reilly Media, January 2015. ISBN: 978-1-4493-5904-1

[63] Bikas Saha and Hitesh Shah: "Apache Tez: Accelerating Hadoop Query Processing," at *Hadoop Summit*, San Jose, CA, June 2014.

[64] Bikas Saha, Hitesh Shah, Siddharth Seth, et al.: "Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications," at *ACM International Conference on Management of Data* (SIGMOD), pages 1357–1369, June 2015. doi:10.1145/2723372.2742790

[65] Kostas Tzoumas: "Apache Flink: API, runtime, and project roadmap," slideshare.net, 14 January 2015.

[66] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, et al.: "The Stratosphere platform for big data analytics," *The VLDB Journal*, volume 23, number 6, pages 939–964, May 2014. doi:10.1007/s00778-014-0357-y

[67] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly: "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," at *European Conference on Computer Systems* (EuroSys), pages 59–72, March 2007. doi:10.1145/1272996.1273005

[68] Daniel Warneke and Odej Kao: "Nephele: Efficient Parallel Data Processing in the Cloud," at *2nd Workshop on Many-Task Computing on Grids and Supercomputers* (MTAGS), November 2009. doi:10.1145/1646468.1646476

[69] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd: "The PageRank Citation Ranking: Bringing Order to the Web," Stanford InfoLab Technical Report 422, 1999.

[70] Leslie G Valiant: "A bridging model for parallel computation," *Communications of the ACM*, volume 33, number 8, pages 103–111, August 1990. doi:10.1145/79173.79181

[71] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl: "Spinning Fast Iterative Data Flows," *Proceedings of the VLDB Endowment*, volume 5, number 11, July 2012. doi:10.14778/2350229.2350245

[72] Grzegorz Malewicz, Matthew H Austern, Aart J C Bik, et al.: "Pregel: A System for Large-Scale Graph Processing," at *ACM International Conference on Management of Data* (SIGMOD), pages 135–146, June 2010. doi:10.1145/1807167.1807184

[73] 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.

[74] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, et al.: "Musketeer: all for one, one for all in data processing systems," at *10th European Conference on Computer Systems* (EuroSys), April 2015. doi:10.1145/2741948.2741968

[75] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin: "GraphChi: Large-Scale Graph Computation on Just a PC," at *10th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2012.

[76] Andrew Lenharth, Donald Nguyen, and Keshav Pingali: "Parallel graph analytics," *Communications of the ACM*, volume 59, number 5, pages 78–87, May 2016. doi:10.1145/2901919

[77] Fabian Hüske: "Peeking into Apache Flink's Engine Room," flink.apache.org, 13 March 2015.

[78] Mostafa Mokhtar: "Hive 0.14 Cost Based Optimizer (CBO) Technical Overview," hortonworks.com, 2 March 2015.

[79] Michael Armbrust, Reynold S Xin, Cheng Lian, et al.: "Spark SQL: Relational Data Processing in Spark," at *ACM International Conference on Management of Data* (SIGMOD), June 2015. doi:10.1145/2723372.2742797

[80] Daniel Blazevski: "Planting Quadtrees for Apache Flink," insightdataengineering.com, 25 March 2016.

[81] Tom White: "Genome Analysis Toolkit: Now Using Apache Spark for Data Processing," blog.cloudera.com, 6 April 2016.

CHAPTER 11

Stream Processing

> A complex system that works is invariably found to have evolved from a simple system that works. The inverse proposition also appears to be true: A complex system designed from scratch never works and cannot be made to work.
>
> —John Gall, *Systemantics (1975)*

In Chapter 10 we discussed batch processing — techniques that read a set of files as input, and produce a new set of output files. The output is a form of *derived data*, that is, a dataset that can be re-created by running the batch process again if necessary. We saw how this simple but powerful idea can be used to create search indexes, recommendation systems, analytics and more.

However, one big assumption remained throughout Chapter 10: that the input is *bounded*, i.e. of a known and finite size, so the batch process knows when it has finished reading its input. For example, the sorting operation that is central to MapReduce must read its entire input before it can start producing output: it could happen that the very last input record is the one with the lowest key, and thus needs to be the very first output record, so starting the output early is not an option.

In reality, a lot of data is *unbounded* because it arrives gradually over time: your users produced data yesterday and today, and they will continue to produce more data tomorrow. Unless you go out of business, this process never ends, and so the data is never "complete" in any meaningful way [1]. Thus, batch processors must artificially divide the data into chunks of fixed duration: for example, processing a day's worth of data at the end of every day, or processing an hour's worth of data at the end of every hour.

The problem with daily batch processes is that changes in the input are only reflected in the output a day later, which is too slow for many impatient users. To reduce the delay, we can run the processing more frequently — say, processing a second's worth of data at the end of every second — or even continuously, abandoning the fixed time-slices entirely, and simply processing every event as it happens. That is the idea behind *stream processing*.

In general, a stream refers to data that is incrementally made available over time. The concept appears in many places: in the `stdin` and `stdout` of Unix, programming languages (lazy lists) [2], filesystem APIs (such as Java's FileInputStream), TCP connections, delivering audio and video over the internet, and so on.

In this chapter we will look at *event streams* as a data management mechanism: the unbounded, incrementally-processed counterpart to the batch data we saw in the last chapter. We will first discuss how streams are represented, stored and transmitted over a network. In "Databases and streams" on page 436 we will investigate the relationship between streams and databases. And finally, in "Processing Streams" on page 448 we will explore approaches and tools for processing those streams continually, and ways how they can be used to build applications.

Transmitting Event Streams

In the batch processing world, the inputs and outputs of a job are files (perhaps on a distributed filesystem). What does the streaming equivalent look like?

When the input is a file (a sequence of bytes), the first processing step is usually to parse it into a sequence of records. In a stream processing context, a record is more commonly known as an *event*, but it is essentially the same thing: a small, self-contained, immutable object containing the details of something that happened at some point in time. An event usually contains a timestamp indicating when it happened.

For example, the thing that happened might be an action that a user took, such as viewing a page or making a purchase. It might also originate from a machine, such as a periodic measurement from a temperature sensor, or a CPU utilization metric. In the example of "Batch Processing with Unix Tools" on page 379, each line of the web server log is an event.

An event may be encoded as a text string, or JSON, or perhaps some binary form, as discussed in Chapter 4. This allows you to store an event, for example by appending it to a file, inserting it into a relational table, or writing it to a document database. It also allows you to send the event over the network to another node in order to process it.

In batch processing, a file is written once and then potentially read by multiple jobs. Analogously, in streaming terminology, an event is generated once by a *producer* (also known as *publisher* or *sender*), and then potentially processed by multiple *con-*

sumers (*subscribers* or *recipients*). In a filesystem, a filename identifies a set of related records; in a streaming system, related events are usually grouped together into a *topic* or *stream*.

In principle, a file or database is sufficient to connect producers and consumers: a producer writes every event that it generates to the datastore, and each consumer periodically polls the datastore to check for events that appeared since it last ran. This is essentially what a batch process does when it processes a day's worth of data at the end of every day.

However, when moving towards continual processing with low delays, polling becomes expensive if the datastore is not designed for this kind of usage. The more often you poll, the lower the percentage of requests that return new events, and thus the higher the overheads become. Instead, it is better for consumers to be notified when new events appear.

Databases have traditionally not supported this kind of notification mechanism very well: relational databases commonly have *triggers*, which can react to a change (e.g. a row being inserted into a table), but they are very limited in what they can do, and have been somewhat of an afterthought in database design [3, 4]. Instead, specialized tools were developed for the purpose of delivering event notifications.

Messaging systems

A common approach for notifying consumers about new events is to use a *messaging system*: a producer sends a message containing the event, which is then pushed to consumers. We touched on these systems previously in "Message passing data flow" on page 132, but we will now go into more detail.

A direct communication channel like a Unix pipe or TCP connection between producer and consumer would be a simple way of implementing a messaging system. However, most messaging systems expand on this basic model. In particular, Unix pipes and TCP connect exactly one sender with one recipient, whereas a messaging system allows multiple producer nodes to send messages to the same topic, and allows multiple consumer nodes to receive messages in a topic.

Within this *publish-subscribe* model, different systems take a wide range of approaches, and there is no one right answer for all purposes. To differentiate the systems, it is particularly helpful to ask the following two questions:

1. *What happens if the producers send messages faster than the consumers can process them?* Broadly speaking, there are three options: the system can drop messages, buffer messages in a queue, or apply *backpressure* (block the producer from sending more messages). For example, Unix pipes and TCP use backpres-

sure: they have a small fixed-size buffer, and if it fills up, the sender is blocked until the recipient takes data out of the buffer.

If messages are buffered in a queue, it is important to understand what happens as that queue grows. Does the system crash if the queue no longer fits in memory? Or does it write messages to disk, but how does that affect performance?

2. *What happens if nodes crash or temporarily go offline? Are any messages lost?* As with databases, durability may require some combination of writing to disk and/or replication (see "Replication and durability" on page 218), which has a cost. If you can afford to sometimes lose messages, you can probably get higher throughput and lower latency on the same hardware.

Whether message loss is acceptable depends very much on the application. For example, with sensor readings and metrics that are transmitted periodically, an occasional missing data point is perhaps not important, since an updated value will be sent a short time later anyway. However, beware that if a large number of messages is dropped, it may not be immediately apparent that the metrics are incorrect [5]. If you are counting events, it is more important that they are delivered reliably, since every lost message means incorrect counters.

A nice property of the batch processing systems in Chapter 10 is that they provide a strong reliability guarantee: failed tasks are automatically retried, and partial output from failed tasks is automatically discarded. This means the output is the same as if no failures had occurred, which helps simplify the programming model. Later in this chapter we will examine how we can provide similar guarantees in a streaming context.

Direct messaging from producers to consumers

A number of messaging systems use direct network communication between producers and consumers, without going via intermediary nodes:

- UDP multicast is widely used in the financial industry for streams such as stock market feeds, where low latency is important [6]. Although UDP itself is unreliable, application-level protocols can recover lost packets (the producer must remember packets it has sent, so that it can retransmit them on demand).
- Brokerless messaging libraries such as ZeroMQ [7] and nanomsg take a similar approach, implementing publish-subscribe messaging over TCP or IP multicast.
- StatsD [8] and Brubeck [5] use unreliable UDP messaging for collecting metrics from all machines on the network, and monitoring them. (In the StatsD protocol, counter metrics are only correct if all messages are received; using UDP makes the metrics at best approximate [9]. See also "TCP versus UDP" on page 275.)

- If the consumer exposes a service on the network, producers can make a direct HTTP or RPC request (see "Data flow through services: REST and RPC" on page 127) to push messages to the consumer. This is the idea behind webhooks [10], a pattern in which a callback URL of one service is registered with another service, and it makes a request to that URL whenever an event occurs.

Although these direct messaging systems work well in the situations for which they are designed, they generally require the application code to be aware of the possibility of message loss. The faults they can tolerate are quite limited: even if the protocols detect and retransmit packets that are lost in the network, they generally assume that producers and consumers are constantly online.

If a consumer is offline, it may miss messages that were sent while it is unreachable. Some protocols allow the producer to retry failed message deliveries, but this approach may break down if the producer crashes, losing the buffer of messages that it was supposed to retry.

Message brokers

A widely-used alternative is to send messages via a *message broker* (also known as *message queue*), which is essentially a kind of database that is optimized for handling message streams [11]. It runs as a server, with producers and consumers connecting to it as clients. Producers write messages to the broker, and consumers receive them by reading them from the broker.

By centralizing the data in the broker, these systems can more easily tolerate clients that come and go (connect, disconnect and crash), and the question of durability is moved to the broker instead. Some message brokers only keep messages in memory, while others (depending on configuration) write them to disk so that they are not lost in case of a broker crash. Faced with slow consumers, they generally allow unbounded queueing (as opposed to dropping messages or backpressure), although this may also depend on the configuration.

A consequence of queueing is also that consumers are generally *asynchronous*: when a producer sends a message, it normally only waits for the broker to confirm that it has buffered the message, but it does not wait for the message to be processed by consumers. The delivery to consumers will happen at some undetermined future point in time — often within a fraction of a second, but sometimes significantly later if there is a queue backlog.

Message brokers compared to databases

Some message brokers can even participate in 2-phase commit protocols using XA or JTA (see "Distributed transactions in practice" on page 350). This makes them quite similar in nature to databases, although there are still important practical differences between message brokers and databases:

- Databases usually keep data forever until it is explicitly deleted, whereas most message brokers automatically delete a message when it has been successfully delivered to its consumers. Such message brokers are not suitable for long-term data storage.
- Since they quickly delete messages, most message brokers assume that their working set is fairly small, i.e. the queues are short. If the broker needs to buffer a lot of messages because the consumers are slow (perhaps spilling messages to disk if they no longer fit in memory), each individual message takes longer to process, and the overall throughput may degrade [12].
- Databases often support secondary indexes and various ways of searching for data, while message brokers often support some way of subscribing to a subset of topics matching some pattern. The mechanisms are different, but both are essentially ways for a client to select the portion of the data that it wants to know about.
- When querying a database, the result is typically based on a point-in-time snapshot of the data; if another client subsequently writes something to the database that changes the query result, the first client does not find out that its prior result is now outdated (unless it repeats the query, i.e. polls for changes). By contrast, message brokers do not support arbitrary queries, but they do notify clients when data changes (i.e. when new messages become available).

This is the traditional view of message brokers, which is encapsulated in standards like JMS [13] and AMQP [14], and implemented in software like RabbitMQ, ActiveMQ, HornetQ, Qpid, TIBCO Enterprise Message Service, IBM MQ, and Azure Service Bus.

Multiple consumers

When multiple consumers are reading messages in the same topic, two main patterns of messaging are used, as illustrated in Figure 11-1:

Load balancing

Each message is delivered to *one* of the consumers, so the consumers can share the work of processing the messages in the topic. The broker may assign messages to consumers arbitrarily. This is useful when the messages are expensive to process, and so you want to be able to add consumers to parallelize the processing. (In AMQP, this is done by having multiple clients consuming from the same queue, and in JMS it is called a *shared subscription*.)

Fan-out

Each message is delivered to *all* of the consumers. This allows several independent consumers to each "tune in" to the same broadcast of messages, without affecting each other — the streaming equivalent of having several different batch jobs that read the same input file. (This is done with topic subscriptions in JMS, and exchange bindings in AMQP.)

The two patterns can be combined: for example, two separate groups of consumers may each subscribe to a topic, such that each group collectively receives all messages, but within each group only one of the nodes receives each message.

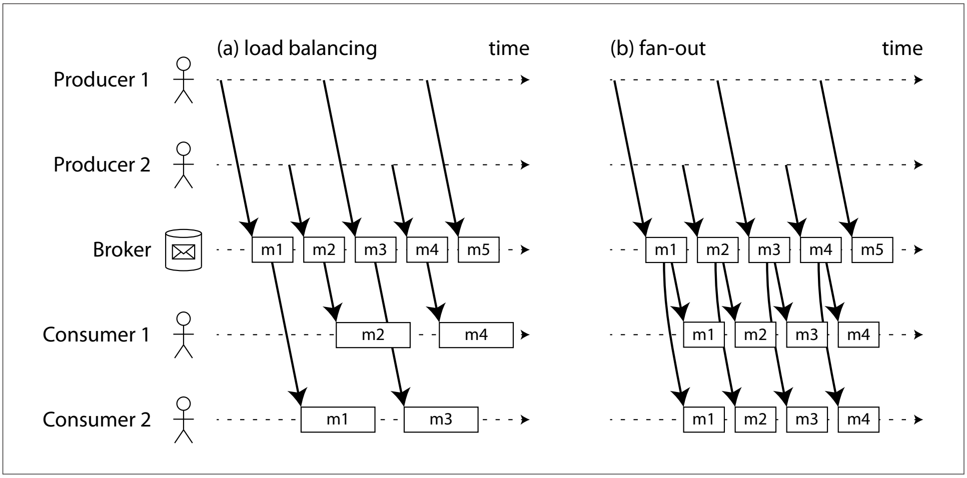


Figure 11-1. (a) Load balancing: sharing the work of consuming a topic among consumers; (b) fan-out: delivering each message to multiple consumers.

Acknowledgements and redelivery

Consumers may crash at any time, so it could happen that a broker delivers a message to a consumer, but the consumer never processes it, or only partially processes it before crashing. In order to ensure that the message is not lost, message brokers use *acknowledgements*: a client must explicitly tell the broker when it has finished processing a message, so that the broker can remove it from the queue.

If the connection to a client is closed or times out without the broker receiving an acknowledgement, it assumes that the message was not processed, and therefore it delivers the message again to another consumer. (Note that it could happen that the message actually *was* fully processed, but the acknowledgement was lost in the network. To handle this case requires an atomic commit protocol, as discussed in "Distributed transactions in practice" on page 350.)

When combined with load balancing, this redelivery behavior has an interesting effect on the ordering of messages. In Figure 11-2, the consumers generally process messages in the order they were sent by producers. However, consumer 2 crashes while processing message m3, at the same time as consumer 1 is processing message m4. The unacknowledged message m3 is subsequently redelivered to consumer 1, with the result that consumer 1 processes messages in the order m4, m3, m5. Thus, m3 and m4 are not delivered in the same order as they were sent by producer 1.

Even if the message broker otherwise tries to preserve the order of messages (as required by both the JMS and AMQP standards), the combination of load balancing with redelivery inevitably leads to messages being reordered. This is not a problem if messages are completely independent of each other, but it can be important if there are causal dependencies between messages, as we shall see later in the chapter.

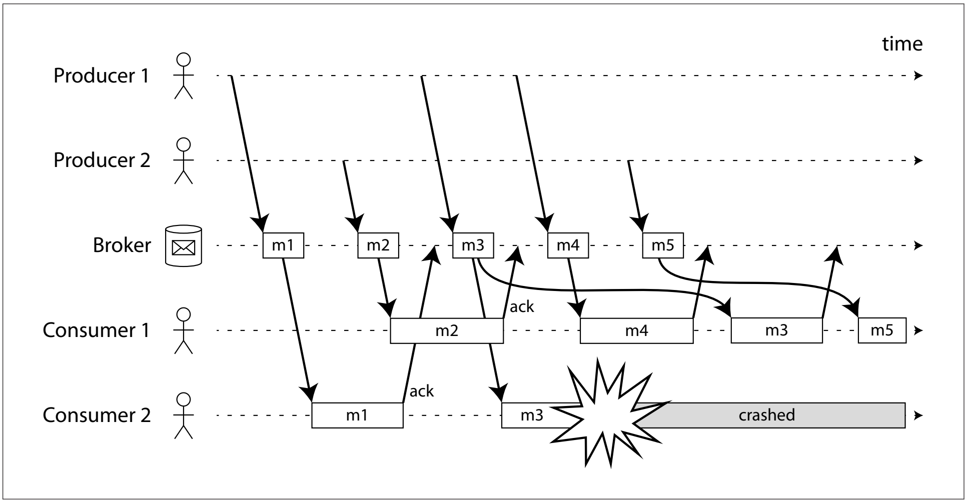


Figure 11-2. Consumer 2 crashes while processing m3, so it is redelivered to consumer 1 at a later time.

Partitioned logs

Sending a packet over a network, or making a request to a network service, is normally a transient operation that leaves no permanent trace. Although it is possible to record it permanently (using packet capture and logging), we normally don't think of it that way. Even message brokers that durably write messages to disk quickly delete them again after they have been delivered to consumers, because they are built around a transient messaging mindset.

Databases and filesystems take the opposite approach: everything that is written to a database or file is normally expected to be permanently recorded, at least until someone explicitly chooses to delete it again.

This difference in mindset has a big impact on how derived data is created. A key feature of batch processes, as discussed in Chapter 10, is that you can run them repeatedly, experimenting with the processing steps, without risk of damaging the input (since the input is read-only). This is not the case with AMQP/JMS-style messaging: receiving a message is destructive if processing it causes it to be deleted from the broker, so you cannot run the same consumer again and expect to get the same result.

If you add a new consumer to a messaging system, it typically only starts receiving messages from the time it was registered, but no prior messages (since they are already gone). Contrast this with files and databases, where you can add a new client at any time, and it can read data written arbitrarily far in the past (as long as it has not been overwritten or deleted).

Why can we not have a hybrid, combining the durable storage approach of databases with the low-latency notification facilities of messaging? This is the idea behind *log-based message brokers*.

Using logs for message storage

A log is simply an append-only sequence of records on disk. We previously discussed logs in the context of log-structured storage engines and write-ahead logs in Chapter 3, and in the context of replication in "Implementation of replication logs" on page 152.

The same structure can be used to implement a message broker: a producer sends a message by appending it to the end of the log, and a consumer receives messages by reading the log sequentially. If a consumer reaches the end of the log, it waits for a notification that a new message has been appended. The Unix tool `tail -f`, which watches a file for data being appended, essentially works like this.

In order to scale to higher throughput than a single disk can offer, the log can be *partitioned* (in the sense of Chapter 6). Different partitions can then be hosted on different machines, and several partitions can be grouped together to a topic. This approach is illustrated in Figure 11-3.

Within each partition, the broker assigns a monotonically increasing sequence number, or *offset*, to every message (in Figure 11-3, the numbers in boxes are message offsets). Such a sequence number makes sense because a partition is append-only, so the messages within a partition are totally ordered. There is no ordering guarantee across different partitions.

Apache Kafka [15, 16], Amazon Kinesis Streams [17] and Twitter's DistributedLog [18, 19] are log-based message brokers that work like this. Even though they write all messages to disk, they able to achieve throughput of millions of messages per second by partitioning across multiple machines, and fault tolerance by replicating messages [20, 21].

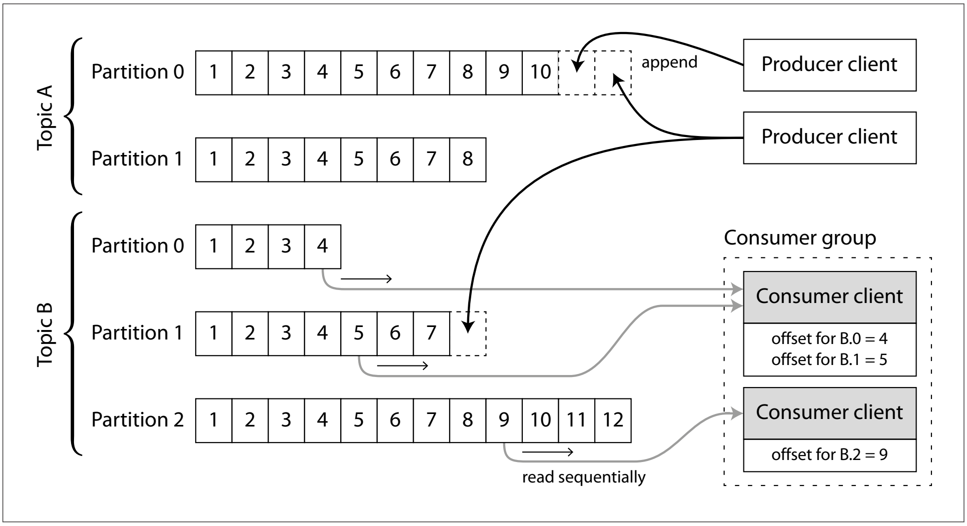


Figure 11-3. Producers send messages by appending them to a topic-partition file, and consumers read these files sequentially.

Logs compared to traditional messaging

The log-based approach trivially supports fan-out messaging, because several consumers can independently read the log without affecting each other. To achieve load balancing across a group of consumers, instead of assigning individual messages to consumer clients, the broker can assign entire partitions to nodes in the consumer group.

Each client then consumes *all* the messages in the partitions it has been assigned. Typically, when a consumer has been assigned a log partition, it reads the messages in the partition sequentially, in a straightforward single-threaded manner. This coarse-grained load balancing approach has some downsides:

- The number of nodes sharing the work of consuming a topic can be at most the number of log partitions in that topic, because messages within the same partition are delivered to the same node.[i]

[i] It’s possible to create a load balancing scheme in which two consumers share the work of processing a partition by having both read the full set of messages, but one only processes messages with even-numbered offsets while the other processes odd-numbered offsets. Or you could spread message processing over a thread pool, but that approach complicates consumer offset management. In general, single-threaded processing of a partition is preferable.

- If a single message is slow to process, it holds up the processing of subsequent messages in that partition (a form of head-of-line blocking, see "Describing performance" on page 11).

Thus, in situations where messages may be expensive to process and you want to parallelize processing on a message-by-message basis, and where message ordering is not so important, the JMS/AMQP style of message broker is preferable. On the other hand, in situations with high message throughput, where each message is fast to process and where message ordering is important, the log-based approach works very well.

Consumer offsets

Consuming a partition sequentially makes it easy to tell which messages have been processed: all messages with an offset less than a consumer's current offset have already been processed, and all messages with a greater offset have not yet been seen. Thus, the broker does not need to track acknowledgements for every single message — it only needs to periodically record the consumer offsets. The reduced bookkeeping overhead, and the opportunities for batching and pipelining in this approach, help increase the throughput of the system.

This offset is in fact very similar to the *log sequence number* that is commonly found in single-leader database replication, and which we discussed in "Setting up new followers" on page 149. In database replication, the log sequence number allows a follower to reconnect to a leader after it has become disconnected, and resume replication without skipping any writes. Exactly the same principle is used here: the message broker behaves like a leader database, and the consumer like a follower.

If a consumer node fails, another node in the consumer group is assigned the failed consumer's partitions, and it starts consuming messages at the last recorded offset. If the consumer had processed subsequent messages, but not yet recorded their offset, those messages will be processed a second time on restart. We will discuss ways of dealing with this issue later in the chapter.

Disk space usage

If you only ever append to the log, you will eventually run out of disk space. To reclaim disk space, the log is actually divided into segments, and from time to time old segments are deleted or moved to archive storage. (We discuss a more sophisticated way of freeing disk space later.)

This means that if a slow consumer cannot keep up with the rate of messages, and it falls so far behind that its consumer offset points to a deleted segment, it will miss some of the messages. Effectively, the log implements a bounded-size buffer that discards old messages when it gets full, also known as a *circular buffer* or *ring buffer*. However, since that buffer is on disk, it can be quite large.

Let's do a back-of-the-envelope calculation. At the time of writing, a typical large hard drive has a capacity of 6 TB and a sequential write throughput of 150 MB/s. If you are writing messages at the fastest possible rate, it takes about 11 hours to fill the drive. Thus, the disk can buffer 11 hours worth of messages, after which it will start overwriting old messages. This ratio remains the same, even if you use many hard drives and machines. In practice, deployments rarely use the full write bandwidth of the disk, so the log can typically keep a buffer of several days or even weeks worth of messages.

You can monitor how far a consumer is behind the head of the log, and raise an alert if it falls behind. As the buffer is large, there is enough time for a human to fix the slow consumer and allow it to catch up before it starts missing messages. And even if the consumer does fall too far behind, it only affects itself, but it does not disrupt the service for other consumers, which is a big operational advantage.

Replaying old messages

We noted previously that with AMQP and JMS-style message brokers, processing and acknowledging messages is a destructive operation, since it causes the messages to be deleted on the broker. On the other hand, in a log-based message broker, consuming messages is more like reading from a file: it is a read-only operation that does not change the log.

The only side-effect of processing, besides any output of the consumer, is that the consumer offset moves forward. But the offset is under the consumer's control, so it can easily be manipulated if necessary: for example, you can start a copy of a consumer with yesterday's offsets, and write the output to a different location, in order to re-process the last day's worth of messages. You can repeat this any number of times, varying the processing code.

This aspect makes log-based messaging more like the batch processes of the last chapter, where derived data is clearly separated from input data through a repeatable transformation process. It allows more experimentation and easier recovery from errors and bugs, making it a good tool for integrating data flows within an organization [22].

Databases and streams

We have drawn some comparisons between message brokers and databases. Even though they have traditionally been considered to be separate categories of tools, we saw that log-based message brokers have been successful in taking ideas from databases and applying them to messaging. We can also go in reverse: take ideas from messaging and streams, and apply them to databases.

We said previously that an event is a record of something that happened at some point in time. The thing that happened may be a user action (e.g. typing a search query), or a sensor reading, but it may also be a *write to a database*. The fact that something was written to a database is an event that can be captured, stored and processed. This suggests that the connection between databases and streams runs deeper than just the physical storage of logs on disk — it is quite fundamental.

In fact, a replication log (see "Implementation of replication logs" on page 152) is a stream of database write events, produced by the leader as it processes transactions. The followers apply that stream of writes to their own copy of the database, and thus end up with an up-to-date copy of the same data. The events in the replication log describe the data changes that occurred.

We also came across the *state machine replication* principle in "Total order broadcast" on page 338, which states: if every event represents a write to the database, and every replica processes the same events in the same order, then the replicas will all end up in the same final state. (Processing an event is assumed to be a deterministic operation.) It's just another case of event streams!

In this section we will first look at a problem that arises in heterogeneous data systems, and then explore how we can solve it by bringing ideas from event streams to databases.

Keeping systems in sync

As we have seen throughout this book, there is no single system that can satisfy all data storage, querying and processing needs. In practice, most non-trivial web applications need to combine several different technologies in order to satisfy their requirements: for example, using an OLTP database to serve user requests, a cache to speed up common requests, a full-text index to handle search queries, and a data warehouse for analytics. Each of these has its own copy of the data, stored in its own representation that is optimized for its purposes.

As the same data appears in all these different places, they need to be kept in sync with each other: if an item is updated in the database, it also needs to be updated in the cache, search indexes and the data warehouse. With data warehouses, this is usually done through ETL processes (see "Data warehousing" on page 88) — often by taking a full copy of a database, transforming it, and bulk-loading it into the data warehouse. In other words, a batch process.

If periodic full database dumps are too slow, an alternative that is sometimes used is *dual writes*, in which the application code explicitly writes to each of the systems when data changes: for example, first writing to the database, then updating the search index, then invalidating the cache entries.

However, dual writes have some serious problems, one of which is a race condition illustrated in Figure 11-4. In this example, two clients concurrently want to update an item X: client 1 wants to set the value to A, and client 2 wants to set it to B. Both clients first write the new value to the database, then write it to the search index.

Due to unlucky timing, the requests are interleaved: the database first sees the write from client 1 setting the value to A, then the write from client 2 setting the value to B, so the final value in the database is B. The search index first sees the write from client 2, then client 1, so the final value in the search index is A. The two systems are now permanently inconsistent with each other, even though no error occurred in the execution.

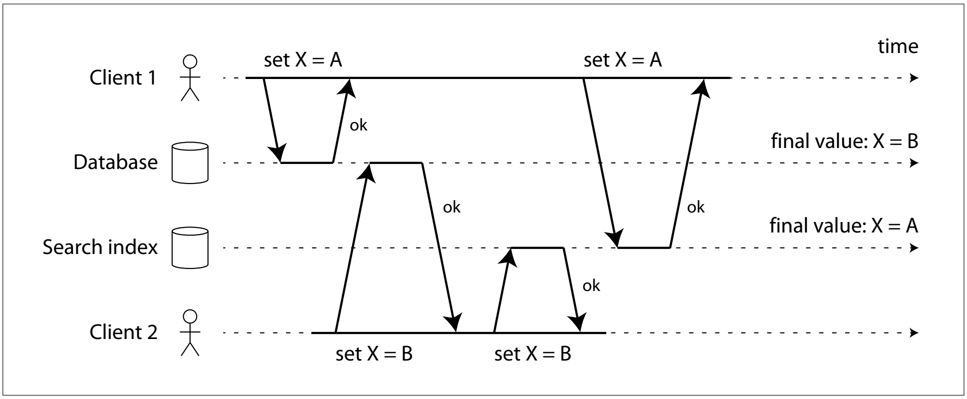


Figure 11-4. In the database, X is first set to A and then to B, while at the search index the writes arrive in the opposite order.

Unless you have some additional concurrency tracking mechanism, such as the version vectors we discussed in "Detecting concurrent writes" on page 178, you will not even notice that concurrent writes occurred—one value will simply silently overwrite another value.

Another problem with dual writes is that one of the writes may fail while the other succeeds. This is a fault-tolerance problem rather than a concurrency problem, but it also has the effect of the two systems becoming inconsistent with each other. Ensuring that they either both succeed or both fail is a case of the atomic commit problem, which is expensive to solve (see "Atomic commit and two-phase commit (2PC)" on page 344).

If you only have one replicated database with a single leader, then that leader determines the order of writes, so the state machine replication approach works among replicas of the database. However, in Figure 11-4 there isn't a single leader: the database may have a leader and the search index may have a leader, but neither follows the other, and so conflicts can occur (see "Multi-leader replication" on page 161).

The situation would be better if there really was only one leader, for example the database, and if we could make the search index a follower of the database. But is this possible in practice?

Change data capture

The problem with most databases' replication logs is that they have long been considered to be an internal implementation detail of the database, not a public API. Clients are supposed to query the database through its data model and query language, not parse the replication logs and try to extract data from them.

For decades, many databases simply did not have a documented way of getting the log of changes written to it. For this reason it was difficult to take all the changes made in a database and replicate them to a different storage technology such as a search index, cache or data warehouse.

More recently, there has been growing interest in *change data capture* (CDC), which is the process of observing all data changes written to a database, and extracting them in a form in which they can be replicated to other systems. CDC is especially interesting if changes are made available as a stream, immediately as they are written.

For example, you can capture the changes in a database and continually apply the same changes to a search index. If the log of changes is applied in the same order, you can expect the data in the search index to match the data in the database. The search index and any other derived data systems are just consumers of the change stream, as illustrated in Figure 11-5.

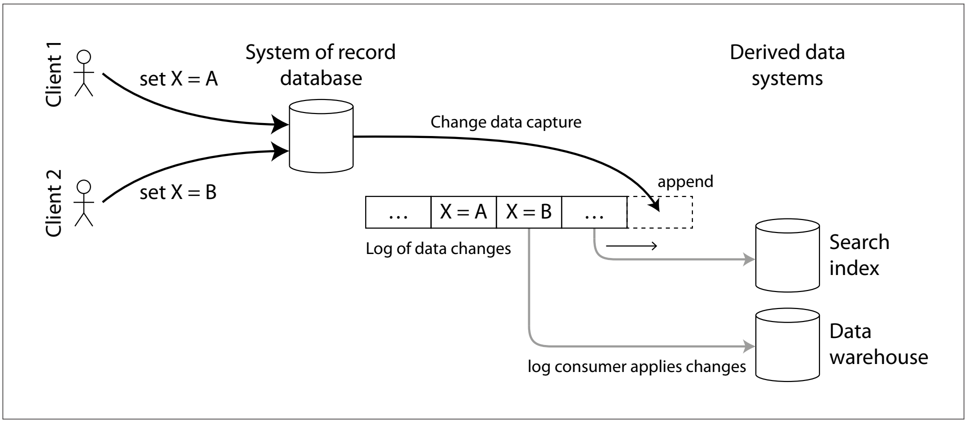


Figure 11-5. Taking data in the order it was written to one database, and applying the changes to other systems in the same order.

Implementing change data capture

We can call the log consumers *derived data systems*, as discussed in the introduction to Part III: the data stored in the search index and the data warehouse is just another view onto the data in the system of record. Change data capture is a mechanism for ensuring that all changes made to the system of record are also reflected in the derived data systems, so that the derived systems have an accurate copy of the data.

Essentially, change data capture makes one database the leader (the one from which the changes are captured), and turns the others into followers. A log-based message broker is well suited for transporting the change events from the source database, since it preserves the ordering of messages (avoiding the reordering issue of Figure 11-2).

Database triggers can be used to implement change data capture (see "Trigger-based replication" on page 154), by registering triggers that observe all changes to data tables and add an entry to a changelog table. However, they tend to be fragile and have significant performance overheads. Parsing the replication log can be a more robust approach, although it also comes with challenges, such as handling schema changes.

LinkedIn's Databus [23], Facebook's Wormhole [24] and Yahoo's Sherpa [25] use this idea at large scale. Bottled Water implements CDC for PostgreSQL using an API that decodes the write-ahead log [26], Maxwell and Debezium do something similar for MySQL by parsing the binlog [27, 28], Mongoriver reads the MongoDB oplog [29, 30], and GoldenGate provides similar facilities for Oracle [31, 32].

Like message brokers, change data capture is usually asynchronous: the system of record database does not wait for the change to be applied to consumers before committing it. This has the operational advantage that adding a slow consumer does not slow down the system of record, but it has the downside that all the issues of replication lag apply (see "Problems With Replication Lag" on page 155).

Initial snapshot

If you have the log of all changes that were ever made to a database, you can reconstruct the entire state of the database by replaying the log. However, in many cases, keeping all changes forever would require too much disk space, and replaying it would take too long, so the log needs to be truncated.

Building a new full-text index, for example, requires a full copy of the entire database — it is not sufficient to only apply a log of recent changes, since it would be missing items that were not recently updated. Thus, if you don't have the entire log history, you need to start with a consistent snapshot, as previously discussed in "Setting up new followers" on page 149.

The snapshot of the database must correspond to a known position or offset in the change log, so that you know at which point to start applying changes after the snapshot has been processed. Some CDC tools integrate this snapshot facility, while others leave it as a manual operation.

Log compaction

If you can only keep a limited amount of log history, you need to go through the snapshot process every time you want to add a new derived data system. However, *log compaction* provides a good alternative.

We discussed log compaction previously in "Hash indexes" on page 70, in the context of log-structured storage engines (see Figure 3-2 for an example). The principle is simple: the storage engine periodically looks for log records with the same key, throws away any duplicates and keeps only the most recent update for each key. This compaction and merging process runs in the background.

An update with a special null value indicates that a key was deleted. But as long as a key is not overwritten or deleted, it stays in the log forever. The disk space required for such a compacted log depends only on the current contents of the database, not the number of writes that have ever occurred in the database. If the same key is frequently overwritten, previous values will eventually be garbage-collected, and only the latest value will be retained.

The same idea works in the context of log-based message brokers and change data capture. If the CDC system is set up such that every change has a primary key, and every update for a key replaces the previous value for that key, then it's sufficient to keep just the most recent write for a particular key.

Now, whenever you want to rebuild a derived data system such as a search index, you can start a new consumer from offset 0 of the log-compacted topic, and sequentially scan over all messages in the log. The log is guaranteed to contain the most recent value for every key in the database (and maybe some older values) — in other words, it can obtain a full copy of the database contents without having to take another snapshot on the CDC source database.

This log compaction feature is supported by Apache Kafka. As we shall see later in this chapter, it allows the message broker to be used for durable storage, not just for transient messaging.

API support for change streams

Increasingly, databases are beginning to support change streams as a first-class interface, rather than the retrofitted and reverse-engineered efforts that change data capture has long been. For example, RethinkDB allows queries to subscribe to notifications when the results of the query change [33], Firebase [34] and CouchDB

[35] provide data synchronization based on a change feed that is also made available to applications, and Meteor uses the MongoDB oplog to subscribe to data changes and update the user interface [36].

Kafka Connect [37] is an effort to integrate change data capture tools for a wide range of database systems with Kafka. Once the stream of change events is in Kafka, it can be used to update derived data systems such as search indexes, and also feed into stream processing systems as discussed later in this chapter.

Event sourcing

There are some parallels between the ideas we've discussed here and *event sourcing*, a technique that was developed in the Domain-Driven Design (DDD) community [38, 39, 40]. We will discuss event sourcing briefly, because it incorporates some useful and relevant ideas for streaming systems.

Similarly to change data capture, event sourcing involves storing all changes to the application state as a log of change events. The biggest difference is that event sourcing applies the idea at a different level of abstraction:

- In change data capture, the application uses the database in a mutable way, updating and deleting records at will. The log of changes is extracted from the database at a low level (e.g. by parsing the replication log), which ensures that the order of writes extracted from the database matches the order in which they were actually written, avoiding the race condition in Figure 11-4. The application writing to the database does not need to be aware that CDC is occurring.
- In event sourcing, the application logic is explicitly built on the basis of immutable events that are written to an event log. In this case, the event store is append-only, and updates or deletes are discouraged or prohibited. Events are carefully designed to mirror things that happened at the application level.

Event sourcing is similar to the chronicle data model [41], and there are also similarities between an event log and the fact table that you find in a star schema (see "Stars and snowflakes: schemas for analytics" on page 90).

Specialized databases such as Event Store [42] have been developed to support applications using event sourcing, but in general the approach is independent of any particular tool. A conventional database or a log-based message broker can also be used to build applications in this style.

Deriving current state from the event log

An event log by itself is not very useful, because users generally expect to see the current state of the system, not the history of modifications. For example, on a shopping website, users expect to be able to see the current contents of their cart, not an append-only list of all the changes they have ever made to their cart.

Thus, applications that use event sourcing need to take the log of events (representing the data *written* to the system) and transform it into application state that is suitable for showing to user (the way in which data is *read* from the system [43]). This transformation can use arbitrary logic, but it should be deterministic, so that you can run it again and derive the same application state from the event log.

Like with change data capture, replaying the event log allows you to reconstruct the current state of the system. However, log compaction needs to be handled differently:

- A CDC event for the update of a record typically contains the entire new version of the record, so the current value for a primary key is entirely determined by the most recent event for that primary key, and log compaction can discard previous events for the same key.
- On the other hand, with event sourcing, events are modeled at a higher level: an event typically expresses the intent of a user action, not the mechanics of the state update that occurred as a result of the action. In this case, later events typically do not override prior events, and so you need the full history of events to reconstruct the final state. Log compaction is not possible in the same way.

Applications that use event sourcing typically have some mechanism for storing snapshots of the current state that is derived from the log of events, so they don't need to repeatedly re-process the full log. However, this is only a performance optimization to speed up reads and recovery from crashes; the intention is that the system is able to store all raw events forever, and re-process the full event log whenever required. This is a reasonable assumption for all but the very largest applications.

Commands and events

The event sourcing philosophy is careful to distinguish between *events* and *commands* [44]. When a request from a user first arrives, it is initially a command: at this point it may still fail, for example because some integrity condition is violated. The application must first validate that it can execute the command. If the validation is successful and the command is accepted, it becomes an event, which is durable and immutable.

For example, if a user tries to register a particular username, or reserve a seat on an airplane or in a theater, then the application needs to check that the username or seat is not already taken. (We previously discussed this example in "Fault-tolerant consensus" on page 355.) When that check has succeeded, the application can generate an event to indicate that a particular username was registered by a particular user ID, or that a particular seat has been reserved for a particular customer.

At the point when the event is generated, it becomes a *fact*. Even if the customer later decides to change or cancel the reservation, the fact remains true that they formerly held a reservation for a particular seat, and the change or cancellation is a separate event that is added later.

A consumer of the event stream is not allowed to reject an event: by the time the consumer sees the event, it is already an immutable part of the log, and it may have already been seen by other consumers. Thus, any validation of a command needs to happen synchronously, before it becomes an event.

Alternatively, the user request to reserve a seat could be split into two events: first a tentative reservation, and then a separate confirmation event once the reservation has been validated (as discussed in "Implementing linearizable storage using total order broadcast" on page 340). This allows the validation to take place in an asynchronous process.

State, streams, and immutability

We saw in Chapter 10 that batch processing benefits from the immutability of its input files, so you can run experimental processing jobs on existing input files without fear of damaging them. This principle of immutability is also what makes event sourcing and change data capture so powerful.

We normally think of databases as storing the current state of the application — this representation is optimized for reads, and it is usually the most convenient for serving queries. The nature of state is that it changes, which is why databases support updating and deleting data as well as inserting it. How does this fit with immutability?

Whenever you have state that changes, that state is the result of the events that mutated it over time. For example, your list of currently available seats is the result of the reservations you have processed, the current account balance is the result of the credits and debits on the account, and the response time graph for your web server is an aggregation of the individual response times of all web requests that occurred.

No matter how the state changes, there was always a sequence of events that caused those changes. Even as things are done and undone, the fact remains true that those events occurred. The important thing to realize is that mutable state and an append-only log of immutable events do not contradict each other: they are two sides of the same coin. The log of all changes, the *changelog*, represents the evolution of state over time.

If you are mathematically inclined, you might say that the application state is what you get when you integrate an event stream over time, and a change stream is what you get when you differentiate the state by time, as shown in Figure 11-6 [45, 46].

The analogy has limitations (for example, the second derivative of state does not seem to be meaningful), but it's a useful starting point for thinking about data.

$$state(now) = \int_{t=0}^{now} stream(t)\ \mathrm{d}t \qquad\qquad stream(t) = \frac{\mathrm{d}\, state(t)}{\mathrm{d}t}$$

Figure 11-6. The relationship between the current application state and an event stream.

If you store the changelog durably, that simply has the effect of making the state reproducible. If you consider the log of events to be your system of record, and any mutable state as being derived from it, it becomes easier to reason about the flow of data through a system. As Pat Helland puts it [47]:

> Transaction logs record all the changes made to the database. High-speed appends are the only way to change the log. From this perspective, the contents of the database hold a caching of the latest record values in the logs. The truth is the log. The database is a cache of a subset of the log. That cached subset happens to be the latest value of each record and index value from the log.

Advantages of immutable events

Immutability in databases is an old idea. For example, accountants have been using immutability for centuries in financial bookkeeping. When a transaction occurs, it is recorded in an append-only *ledger*, which is essentially a log of events describing money, goods or services that have changed hands. The accounts, such as profit and loss or the balance sheet, are derived from the transactions in the ledger by adding them up [48].

If a mistake is made, accountants don't erase or change the incorrect transaction in the ledger — instead, they add another transaction that compensates for the mistake, for example refunding an incorrect charge. The incorrect transaction still remains in the ledger forever, because it might be important for auditing reasons. If incorrect figures, derived from the incorrect ledger, have already been published, then the figures for the next accounting period include a correction. This process is entirely normal in accounting [49].

Although such auditability is particularly important in financial systems, it is also beneficial for many other systems that are not subject to such strict regulation. As discussed in "Philosophy of batch process outputs" on page 401, if you accidentally deploy buggy code that writes bad data to a database, recovery is much harder if the code is able to destructively overwrite data. With an append-only log of immutable events, it is much easier to see what happened and recover.

Immutable events also capture more information than just the current state. For example, on a shopping website, a customer may add an item to their cart and then remove it again. Although the second event cancels out the first event from the point of view of order fulfillment, it may be useful to know for analytics purposes that the customer was considering a particular item but then decided against it. Perhaps they will choose to buy it in future, or perhaps they found a substitute. This information is recorded in an event log, but would be lost in a database that deletes items when they are removed from the cart [38].

Deriving several views from the same event log

Moreover, by separating mutable state from the immutable event log, you can derive several different read-oriented representations from the same log of events. This works just like having multiple consumers of a stream (Figure 11-5).

Having an explicit translation step from an event log to a database makes it easier to evolve your application over time: if you want to introduce a new feature that presents your existing data in some new way, you can use the event log to build a separate read-optimized view for the new feature, and run it alongside the existing systems without having to modify them. Running old and new systems side-by-side is often easier than performing a complicated schema migration in an existing system. Once the old system is no longer needed, you can simply shut it down [43, 50].

Storing data is normally quite straightforward if you don't have to worry about how it is going to be queried and accessed; a lot of the complexities of schema design, indexing and storage engines are the result of wanting to support certain query and access patterns (see Chapter 3). For this reason, you gain a lot of flexibility by separating the form in which data is written from the form it is read, and by allowing several different read views. This idea is sometimes known as *command query responsibility segregation* or CQRS [38, 51, 52].

The traditional approach to database and schema design is based on the fallacy that data must be written in the same form as it will be queried. Debates about normalization and denormalization (see "Many-to-one and many-to-many relationships" on page 31) become largely irrelevant if you can translate data from a write-optimized event log to read-optimized application state: it is entirely reasonable to denormalize data in the read-optimized views, as the translation process gives you a mechanism for keeping it up-to-date.

In "Describing load" on page 9 we discussed Twitter's home timelines, a cache of recently-written tweets by the people a particular user is following (like a mailbox). This is another example of read-optimized state: home timelines are highly denormalized, since your tweets are duplicated in all of the timelines of the people following you. However, the fan-out service keeps this duplicated state in sync with new tweets and new following relationships, which keeps the duplication manageable.

Concurrency control

The biggest downside of event sourcing and change data capture is that the consumers of the event log are usually asynchronous, so there is a possibility that a user may make a write to the log, then read from a log-derived view, and find that their write has not yet been reflected in the read view. We discussed this problem and potential solutions previously in "Reading your own writes" on page 156.

One solution would be to perform the updates of the read view synchronously with appending the event to the log. This requires a transaction to combine the writes into an atomic unit, so you either need to keep the event log and the read view in the same storage system, or you need a distributed transaction across the different systems. Alternatively, you could use the approach discussed in "Implementing linearizable storage using total order broadcast" on page 340.

On the other hand, deriving the current state from an event log also simplifies some aspects of concurrency control. Much of the need for multi-object transactions (see "Single-object and multi-object operations" on page 219) stems from a single user action requiring data to be changed in several different places. With event sourcing, you can design an event such that it is a self-contained description of a user action — so the user action requires only a single write in one place, namely appending the event to the log, which is easy to make atomic.

If the event log and the application state are partitioned in the same way (for example, processing an event for a customer in partition 3 only requires updating partition 3 of the application state), then a straightforward single-threaded log consumer needs no concurrency control for writes — by construction, it only processes a single event at a time (see also "Actual serial execution" on page 243). The log removes the nondeterminism of concurrency by defining a serial order of events in a partition [22]. If an event touches multiple state partitions, a bit more work is required, which we will discuss later in this chapter.

Immutability and deletion

Many systems that don't use an event-sourced model nevertheless rely on immutability: for example, Datomic uses immutable data structures to query snapshots of a database, including historical snapshots from past points in time (see "Indexes and snapshot isolation" on page 232). Version control systems such as git, Mercurial and Fossil also rely on immutable data to preserve version history of files.

Storage is now so cheap that keeping all events forever is a viable option for all but the very largest systems. However, there may still be circumstances where you need data to really be deleted, in spite of all immutability. For example, privacy regulations may require deleting a user's personal information after they close their account, or an accidental leak of sensitive information may need to be contained.

In these circumstances, it's not sufficient to just append another event to the log to indicate that the prior data should be considered deleted — you actually want to rewrite history and pretend that the data was never written in the first place. For example, Datomic calls this feature *excision* [53], and the Fossil version control system has a similar concept called *shunning* [54].

Truly deleting data is surprisingly hard [55], since copies can live in many places: for example, storage engines, filesystems and SSDs often write to a new location rather than overwriting in-place [47], and backups are often deliberately immutable to prevent accidental deletion or corruption. Deletion is more a matter of "making it harder to retrieve the data" than a matter of "making it impossible to retrieve the data". Nevertheless, it is sometimes required.

Processing Streams

So far in this chapter we have talked about where streams come from (user activity events, sensors, and writes to databases), and we have talked about how streams are transported (through direct messaging, via message brokers, and event logs).

What remains is to discuss what you can do with the stream once you have it — namely, you can process it. Broadly, there are three options:

1. You can take the data in the events and write it to a database, cache, search index or similar storage system, from where it can then be queried by other clients. As shown in Figure 11-5, this is a good way of keeping a database in sync with changes happening in other parts of the system — especially if the stream consumer is the only client writing to the database. Writing to a storage system is the streaming equivalent of what we discussed in "The output of batch workflows" on page 398.
2. You push the events to users in some way, for example by sending email alerts or push notifications, or by streaming the events to a realtime dashboard where they are visualized. In this case, a human is the ultimate consumer of the stream.
3. You can process one or more input streams to produce one or more output streams. Streams may go through a pipeline consisting of several such processing stages, before they eventually end up at an output (option 1 or 2).

In the rest of this chapter, we will discuss option 3: processing streams to produce other, derived streams. A piece of code that processes streams like this is known as an *operator* or a *job*. It is closely related to the Unix processes and the MapReduce jobs we discussed in Chapter 10, and the pattern of dataflow is similar: a stream processor consumes input streams in a read-only fashion, and writes its output to a different location in an append-only fashion.

The patterns for partitioning and parallelization in stream processors are also very similar to MapReduce and the dataflow engines we saw in Chapter 10, so we won't repeat that topic here. Basic mapping operations such as transforming and filtering records also work the same.

The one crucial difference to batch jobs is that a stream never ends. This difference has many implications: as discussed at the start of this chapter, sorting does not make sense with an unbounded dataset, and so sort-merge joins (see "Reduce-side joins and grouping" on page 391) do not apply. Fault-tolerance mechanisms must also change: with a batch job that has been running for a few minutes, a failed task can simply be restarted from the beginning — but with a stream job that has been running for several years, restarting from the beginning may not be a viable option.

Uses of stream processing

Stream processing has long been used for monitoring purposes, where an organization wants to be alerted if certain things happen. For example:

- fraud detection systems need to determine if the usage patterns of a credit card have unexpectedly changed, and block the card if it is likely to have been stolen;
- trading systems need to examine price changes in a financial market and execute trades according to specified rules;
- manufacturing systems need to monitor the status of machines in a factory, and quickly identify the problem if there is a malfunction;
- military and intelligence systems need to track the activities of a potential aggressor, and raise the alarm if there are signs of an attack.

These kinds of application require quite sophisticated pattern-matching and correlations. However, other uses of stream processing have also emerged over time. In this section we will briefly compare and contrast some of these applications.

Complex event processing

Complex event processing (CEP) is an approach developed in the 1990s for analyzing event streams, especially geared towards the kind of application that requires searching for certain event patterns [56, 57]. Similarly to the way that a regular expression allows you to search for certain patterns of characters in a string, CEP allows you to specify rules to search for certain patterns of events in a stream.

CEP systems often use a high-level declarative query language like SQL, or a graphical user interface, to describe the pattern of events that should be detected. These queries are submitted to a processing engine that consumes the input streams and internally maintains a state machine that performs the required matching. When a match is found, the engine emits a *complex event* (hence the name) with the details of the event pattern that was detected.

In these systems, the relationship between queries and data is reversed compared to normal databases. Usually, a database stores data persistently, and treats queries as transient: when a query comes in, the database searches for data matching the query, and then forgets about the query. CEP engines are the other way round: queries are stored long-term, and events from the input streams continuously flow past the queries in search of a query that matches the event.

Implementations of CEP include Esper [58], IBM InfoSphere Streams [59], Apama, TIBCO StreamBase, and SQLstream.

Stream analytics

Another area in which stream processing is used is for *analytics* on streams. The boundary between CEP and stream analytics is blurry, but as a general rule, analytics tends to be less interested in finding specific event sequences, and is more oriented towards aggregations and statistical metrics over a large number of events, for example:

- measuring the rate of some type of event (how often it occurs per time interval),
- calculating the rolling average of a value over some time period, or
- comparing current statistics to previous time intervals (e.g. to detect trends, or to alert on metrics that are unusually high or low compared to the same time last week).

Such statistics are usually computed over fixed time intervals — for example, you might want to know the average number of queries per second to a service over the last five minutes, and their 99th percentile response time during that period. Averaging over a few minutes smoothes out irrelevant fluctuations from one second to the next, while still giving you a timely picture of any changes in traffic pattern. The time interval over which you aggregate is known as a *window*, and we will look into windowing in more detail in "Reasoning about time" on page 452.

Stream analytics systems sometimes use probabilistic algorithms, such as bloom filters (which we encountered in "SSTables and LSM-trees" on page 74) for set membership, HyperLogLog [60] for cardinality estimation, and various percentile estimation algorithms (see "Percentiles in Practice" on page 14). Probabilistic algorithms produce approximate results, but have the advantage of requiring significantly less memory in the stream processor than exact algorithms. This use of approximation algorithms sometimes leads people to believe that stream processing systems are always lossy and inexact, but that is wrong: there is nothing inherently approximate about stream processing [61].

Many open source distributed stream processing frameworks are designed with analytics in mind: for example, Apache Storm, Spark Streaming, Flink, Concord, Samza, and Kafka Streams [62]. Hosted services include Google Cloud Dataflow and Azure Stream Analytics.

Maintaining materialized views

We saw in "Databases and streams" on page 436 that a stream of changes to a database can be used to keep derived data systems, such as caches, search indexes and data warehouses, up-to-date with a source database. We can regard these examples as specific cases of maintaining *materialized views* (see "Aggregation: Data cubes and materialized views" on page 98): deriving an alternative view onto some dataset, so that you can query it efficiently, and updating that view whenever the underlying data changes [46].

Similarly, in event sourcing, application state is maintained by applying a log of events; here the application state is also a kind of materialized view. Unlike stream analytics scenarios, it is usually not sufficient to consider only events within some time window: building the materialized view requires *all* events that ever happened — perhaps after log compaction has discarded obsolete events (see "Log compaction" on page 441). In effect, you need a window that stretches all the way back to the beginning of time.

In principle, any stream processor could be used for materialized view maintenance, although the need to maintain events forever runs counter to the assumptions of some analytics-oriented frameworks. Samza and Kafka Streams support this kind of usage, building upon Kafka's support for log compaction [63].

Search on streams

Besides CEP, which allows searching for patterns consisting of multiple events, there is also sometimes a need to search for individual events based on complex criteria, such as full-text search queries.

For example, media monitoring services subscribe to feeds of news articles and broadcasts from media outlets, and search for any news mentioning companies, products or topics of interest. This is done by formulating a search query in advance, and then continually matching the stream of news items against this query. Similar features exist on some websites: for example, users of real estate websites can ask to be notified when a new property matching their search criteria appears on the market.

Conventional search engines first index the documents and then run queries over the index. By contrast, searching a stream turns the processing on its head: the queries are stored, and the documents run past the queries. In the simplest case, you can test every document against every query, although this can get slow if you have a large number of queries. To optimize the process, it is possible to index the queries as well as the documents, and thus narrow down the set of queries that may match [64].

Message passing and RPC

In "Message passing data flow" on page 132 we discussed message-passing systems as an alternative to RPC, i.e. as a mechanism for services to communicate, as used for example in the actor model. Although these systems are also based on messages and events, we normally don't think of them as stream processors:

- Actor frameworks are primarily a mechanism for managing concurrency and distributed execution of communicating modules, whereas stream processing is primarily a data management mechanism.
- Communication between actors is often ephemeral and one-to-one, whereas event logs are durable and multi-subscriber.
- Actors can communicate in arbitrary ways (including cyclic request-response patterns), but stream processors are usually set up in acyclic pipelines where every stream is the output of one particular job, and derived from a well-defined set of input streams.

That said, there is some cross-over area between RPC-like systems and stream processing. For example, Apache Storm has a feature called *distributed RPC*, which allows user queries to be farmed out to a set of nodes that also process event streams; these queries are then interleaved with events from the input streams, and results can be aggregated and sent back to the user [65]. It is also possible to build stream processors on top of actor frameworks, although it is worth examining the fault tolerance guarantees that this approach can provide.

Reasoning about time

Stream processors often need to deal with time, especially when used for analytics purposes, which often use time windows such as "last five minutes". It might seem that the meaning of "last five minutes" should be unambiguous and clear, but unfortunately the notion is surprisingly tricky.

In a batch process, the processing tasks rapidly crunch through a large collection of historical events. If some kind of breakdown by time needs to happen, the batch process needs to look at the timestamp embedded in each event. There is no point in looking at the system clock of the machines running the batch process, because the time at which the process is run has nothing to do with the time at which the events actually occurred.

A batch process may read a year worth of historical events within a few minutes; in most cases, the timeline of interest is the year of history, not the few minutes of processing. Moreover, using the timestamp in the events allows the processing to be deterministic: running the same process again on the same input yields the same result (see "Fault tolerance" on page 410).

On the other hand, many stream processing frameworks use the local system clock on the processing machine (the *processing time*) to determine windowing [66]. This approach has the advantage of being simple, and it is reasonable if the delay between event creation and event processing is negligibly short. However, it breaks down if there is any significant processing lag, i.e. if the processing may happen noticeably later than the time at which the event actually occurred.

Event time versus processing time

There are many reasons why processing may be delayed: network faults (see "Unreliable Networks" on page 269), a performance issue leading to contention in the message broker or processor, a restart of the stream consumer, or re-processing past events (see "Replaying old messages" on page 436) while recovering from a fault, or after fixing a bug in the code.

Moreover, message delays also lead to unpredictable ordering of messages. For example, say a user first makes one web request (which is handled by web server A), and then a second request (which is handled by server B). A and B emit events describing the requests they handled, but B's event reaches the message broker before A's event does. Now stream processors will first see the B event and then the A event, even though they actually occurred in the opposite order.

If it helps to have an analogy, consider the Star Wars movies: Episode IV was released in 1977, Episode V in 1980, and Episode VI in 1983, followed by episodes I, II and III in 1999, 2002 and 2005 respectively, and Episode VII in 2015 [67]. If you watched the movies in the order they came out, the order in which you processed the movies is inconsistent with the order of their narrative. (The episode number is like the event timestamp, and the date when you watched the movie is the processing time.) As humans, we are able to cope with such discontinuities, but stream processing algorithms need to be specifically written to accommodate such timing and ordering issues.

Confusing event time and processing time leads to bad data. For example, say you have a stream processor that measures the rate of requests (counting the number of requests per second). If you redeploy the stream processor, it may be shut down for a minute, and process the backlog of events when it comes back up. If you measure the rate based on the processing time, it will look as if there was a sudden anomalous spike of requests while processing the backlog, when in fact the real rate of requests was steady (Figure 11-7).

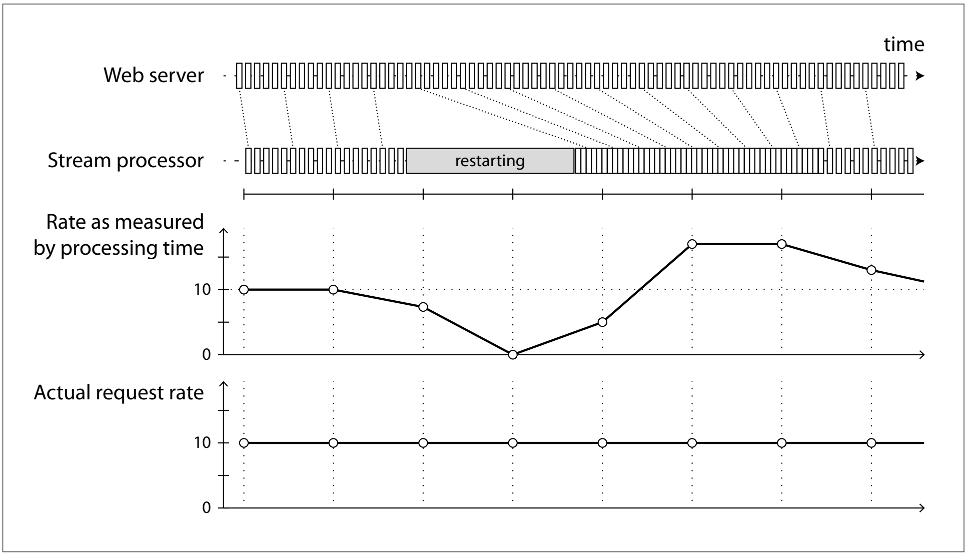


Figure 11-7. Windowing by processing time introduces artifacts due to variations in processing rate.

Knowing when you're ready

A tricky problem when defining windows in terms of event time is that you can never be sure when you have received all of the events for a particular window, or whether there are some events still to come.

For example, say you're grouping events into one-minute windows, so that you can count the number of requests per minute. You have counted some number of events with timestamps that fall in the 37th minute of the hour, and then time has moved on, and now most of the incoming events fall within the 38th and 39th minute of the hour. When do you declare that you have finished the window for the 37th minute, and output its counter value?

In general, it's impossible to be sure that you have received all the events for a window. You can time out and declare a window ready after you have not seen any new events for a while, but it could still happen that some events were buffered on another machine somewhere, which couldn't yet be sent due to a network interruption, but which will turn up later.

If you are using windows based on event timestamps, you need to be able to handle such *straggler* events that arrive after the window has already been declared complete. Broadly, you have two options [1]:

1. Ignore the straggler events, as they are probably a small percentage of events in normal circumstances. You can track the number of dropped events as a metric, and alert if you start dropping a significant amount of data.
2. Recalculate the value for the window with the stragglers included, and issue a *correction* by publishing the updated value (possibly retracting the previous output first).

In some cases it is possible to use a *low watermark* to indicate things like "from now on there will be no more messages with a timestamp earlier than *t*", and to have consumers wait for the watermark as an indication that the window is ready [68]. However, if timestamps are generated by clients, you cannot be sure whether there are still any pending events somewhere in the system, so stragglers (with a timestamp older than the low watermark) are still possible.

Whose clock are you using, anyway?

Assigning timestamps to events is even more difficult when events can be buffered at several points in the system. For example, consider a mobile app that reports events for usage metrics to a server. The app may be used while offline, in which case it will buffer events locally on the device, and send them to a server when an internet connection is next available (hours or days later). To any consumers of this stream, the events will appear as extremely delayed stragglers.

In this context, the timestamp on the events should really be the time at which the user interaction occurred, according to the mobile device's local clock. However, the clock on a user-controlled device often cannot be trusted, as it may be accidentally or deliberately set to the wrong time (see "Clock synchronization and accuracy" on page 281). The time at which the event was received by the server (according to the server's clock) is more likely to be accurate, but less meaningful in terms of describing the user interaction.

To adjust for incorrect device clocks, one approach is to log three timestamps [69]:

- the time at which the event occurred, according to the device clock;
- the time at which the event was sent to the server, according to the device clock;
- the time at which the event was received by the server, according to the server clock.

This allows you to estimate the offset between the device clock and the server clock (assuming the network delay is negligible compared to the required timestamp accuracy), and thus estimate the true time at which the event actually occurred (assuming the device clock offset does not change between the time the event occurred and the time it was sent to the server).

This problem is not unique to stream processing — batch processing suffers from exactly the same issue. It is just more noticeable in a streaming context, where we are more aware of the passage of time.

Types of window

Once you know how the timestamp of an event should be determined, the next step is to decide how windows over time periods should be defined. The window can then be used for aggregations, for example to count events, or to calculate the average of values within the window. Several types of window are in common use [66, 70]:

Tumbling window
: A tumbling window has a fixed length, and every event belongs to exactly one window. For example, if you have a 1-minute tumbling window, all the events with timestamps between 10:03:00 and 10:03:59 are grouped into one window, events between 10:04:00 and 10:04:59 into the next window, and so on. You could implement a 1-minute tumbling window by taking each event timestamp and rounding it to the nearest minute.

Hopping window
: A hopping window also has a fixed length, but allows windows to overlap in order to provide some smoothing. For example, a 5-minute window with a hop size of 1 minute would contain the events between 10:03:00 and 10:07:59, then the next window would cover events between 10:04:00 and 10:08:59, and so on. You can implement this hopping window by first calculating 1-minute tumbling windows, and then aggregating over several adjacent windows.

Sliding window
: A sliding window contains all the events that occur within some interval of each other. For example, a 5-minute sliding window would cover events at 10:03:39 and 10:08:12, because they are less than 5 minutes apart (note that tumbling and hopping 5-minute windows would not have put these two events in the same window, as they use fixed boundaries). A sliding window can be implemented by keeping a buffer of events sorted by time, and removing old events when they expire from the window.

Session window
: Unlike the other window types, a session window has no fixed duration. Instead, it is defined by grouping together all events for the same user that occur closely together in time, and the window ends when the user has been inactive for some time (for example, if there have been no events for 30 minutes). Sessionization is a common requirement for website analytics (see "GROUP BY" on page 394).

Stream joins

In Chapter 10 we discussed how batch jobs can join datasets by key, and how such joins are an important part of data pipelines. Since stream processing generalizes data pipelines to incremental processing of unbounded datasets, there is exactly the same need for joins on streams.

However, the fact that new events can appear anytime on a stream makes joins on streams more challenging than in batch jobs. To understand the situation better, let's distinguish three different types of join: *stream-stream* joins, *stream-table* joins, and *table-table* joins [71]. In the following sections we'll illustrate each by example.

Stream-stream join (window join)

Say you have a search feature on your website, and you are working on improving the ranking of search results. Every time someone types a search query, you log an event containing the query and the results returned. Every time someone clicks one of the search results, you log another event recording the click. In order to calculate the click-through rate for each search result, you need to bring together the events for the search action and the click action, which are connected by having the same session ID [72].

The click may never come if the user abandons their search, and even if it comes, the time between the search and the click may be highly variable: in many cases it might be a few seconds, but it could be as long as days or weeks (if a user runs a search, forgets about that browser tab, and then returns to the tab and clicks a result sometime later). Due to variable network delays, the click event may even arrive before the search event. You can choose a suitable window for the join — for example, you may choose to join a click with a search if they occur at most one hour apart.

Note that embedding the details of the search in the click event is not equivalent to joining the events: that would only tell you about the cases where the user clicked a search result, but not about the searches where the user did not click any of the results. In order to measure search quality you need accurate click-through rates, for which you need both the search events and the click events.

To implement this type of join, a stream processor needs to maintain *state*: for example, all the events that occurred in the last hour, indexed by session ID. Whenever a search event or click event occurs, it is added to the appropriate index, and it also checks the other index to see if another event for the same session ID has already arrived.

Stream-table join (stream enrichment)

In "Example: analysis of user activity events" on page 392 (Figure 10-2) we saw an example of a batch job joining two datasets: a set of user activity events and a database of user profiles. It is natural to think of the user activity events as a stream, and to perform the same join on a continuous basis in a stream processor: the input is a stream of activity events containing a user ID, and the output is a stream of *enriched* activity events in which the user ID has been augmented with profile information about the user.

To perform this join, the stream process needs to look at one activity event at a time, look up the event's user ID in the database, and add the profile information to the activity event. The database lookup could be implemented by querying a remote database; however, as discussed in "Example: analysis of user activity events" on page 392, this is likely to be slow, and risks overloading the remote database [63].

Another approach is to load a copy of the database into the stream processor, so that it can be queried locally without a network round-trip. This is very similar to the hash joins we discussed in "Map-side joins" on page 396: the local copy of the database might be an in-memory hash table if it is small enough, or an index on local disk.

The difference to batch jobs is that a batch job uses a point-in-time snapshot of the database as input, whereas a stream processor is long-running, and the contents of the database is likely to change over time, so the local copy of the data needs to be kept up-to-date. This issue can be solved by change data capture: the stream processor can subscribe to a changelog of the profiles database as well as the stream of activity events. When a profile is created or modified, the stream processor updates its local copy.

Table-table join (materialized view maintenance)

Consider the Twitter timeline example that we discussed in "Describing load" on page 9. We said that when a user wants to view their home timeline, it is too expensive to iterate over all the people the user is following, find their recent tweets, and merge them.

Instead, we want a timeline cache: a kind of per-user "inbox" to which tweets are written as they are sent, so that reading the timeline is a single lookup. Materializing and maintaining this cache requires the following event processing:

- When user u sends a new tweet, it is added to the timeline of every user who is following u.
- When a user deletes a tweet, it is removed from all users' timelines.
- When user u_1 starts following user u_2, recent tweets by u_2 are added to u_1's timeline.
- When user u_1 unfollows user u_2, tweets by u_2 are removed from u_1's timeline.

To implement this cache maintenance in a stream processor, you need streams of events for tweets (sending and deleting) and for follow relationships (following and unfollowing). The stream process needs to maintain a database containing the set of followers for each user, so that it knows which timelines need to be updated when a new tweet arrives [73].

Another way of looking at this stream process is that it maintains a materialized view for a query that joins two tables (tweets and follows), something like the following:

```
SELECT follows.follower_id AS timeline_id,
  array_agg(tweets.* ORDER BY tweets.timestamp DESC)
FROM tweets
JOIN follows ON follows.followee_id = tweets.sender_id
GROUP BY follows.follower_id
```

The join of the streams corresponds directly to the join of the tables in that query. The timelines are effectively a cache of the result of this query, updated every time the underlying tables change.[ii]

Time-dependence of joins

The three types of join above (stream-stream, stream-table and table-table) have a lot in common: they all require the stream processor to maintain some state (search and click events, user profiles, or follower list) based on one join input, and query that state on messages from the other join input.

The order of the events that maintain the state is important (it matters whether you first follow and then unfollow, or the other way round). In a partitioned log, the ordering of events within a single partition is preserved, but there is typically no ordering guarantee across different streams or partitions.

This raises a question: if events on different streams happen around a similar time, in which order are they processed? In the stream-table join example, if a user updates their profile, which activity events are joined with the old profile (processed before the profile update), and which are joined with the new profile (processed after the profile update)? Put another way: if state changes over time, and you join with some state, what point in time do you use for the join [41]?

If the ordering of events across streams is undetermined, the join becomes nondeterministic, which means you cannot re-run the same job on the same input and necessarily get the same result: the events on the input streams may be interleaved in a different way when you run the job again. It is possible for a stream processor to log

[ii] If you regard a stream as the derivative of a table, as in Figure 11-6, and regard a join as a product of two tables $u \cdot v$, something interesting happens: the stream of changes to the materialized join follows the product rule $(u \cdot v)' = u'v + uv'$. In words: whenever the tweets change, it is joined with the current follows, and whenever the follows change, it is joined with the current tweets [45, 46].

the interleaving of messages [74], but this alone is not enough when recovering from faults, as we shall see in the next section.

Fault tolerance

In the final section of this chapter, let's consider how stream processors can tolerate faults. We saw in Chapter 10 that batch processing frameworks can tolerate faults fairly easily: if a task in a MapReduce job fails, it can simply be started again on another machine, and the output of the failed task is discarded. This is possible because input files are immutable, each task writes its output to a separate file on HDFS, and output is only made visible when a task completes successfully.

In particular, the batch approach to fault tolerance ensures that the output of the batch job is the same as if nothing had gone wrong, even if in fact some tasks did fail. It appears as though every input record was processed exactly once — no records are skipped, and none are processed twice. Although restarting tasks means that records may in fact be processed multiple times, the *visible effect* in the output is as if they had only been processed once, a principle known as *exactly-once semantics*.

The same issue of fault tolerance arises in stream processing, but it is less straightforward to handle: waiting until a task is finished before making its output visible is not an option, because a stream is infinite and so you can never finish processing it.

Microbatching and checkpointing

One solution is to break the stream into small blocks, and treat each block like a miniature batch process. This approach is called *microbatching*, and it is used in Spark Streaming [75]. The batch size is typically around 1 second, which is a performance compromise: smaller batches incur greater scheduling and coordination overhead, while larger batches mean a longer delay before results of the stream processor become visible.

Microbatching also implicitly provides a tumbling window equal to the batch size (windowed by processing time, not event timestamps); any jobs that require larger windows need to explicitly carry over state from one microbatch to the next.

A variant approach, used in Apache Flink, periodically generates rolling checkpoints of state and writes them to durable storage [76, 77]. If a stream operator crashes, it can restart from its most recent checkpoint, and discard any output generated between the last checkpoint and the crash. The checkpoints are triggered by barriers in the message stream, similar to the boundaries between microbatches, but without forcing a particular window size.

Within the confines of the stream processing framework, the microbatching and checkpointing approaches provide the same exactly-once semantics as batch processing. However, as soon as output leaves the stream processor (for example, by writing to a database, sending messages to an external message broker, or sending emails), the framework is no longer able to discard the output of a failed batch.

In this case, restarting a failed task causes the external side-effect to happen twice, and microbatching or checkpointing alone is not sufficient to prevent this problem.

Atomic commit revisited

In order to give the appearance of exactly-once processing in the presence of faults, we need to ensure that all outputs and side-effects of processing an event take effect *if and only if* the processing is successful. Those effects include any messages sent to downstream operators or external messaging systems (including email or push notifications), any database writes, any changes to operator state, and any acknowledgement of input messages (including moving the consumer offset forward in a log-based message broker).

Those things either all need to happen atomically, or none of them must happen, but they should not go out of sync with each other. If this approach sounds familiar, it is because we discussed it in "Exactly-once message processing" on page 351 in the context of distributed transactions and 2-phase commit.

In Chapter 9 we discussed the problems in the traditional implementation of distributed transactions, such as XA. However, in more restricted environments it is possible to implement such an atomic commit facility efficiently. This approach is used in Google Cloud Dataflow [68, 76], and there are plans to add similar features to Apache Kafka [78]. The approach relies on writing the transaction commit as a single object to a fault-tolerant datastore, since a single-object write can be made atomic fairly easily (see "Single-object writes" on page 221).

Idempotence

Our goal is to discard the partial output of any failed tasks, so that they can be safely retried without taking effect twice. Distributed transactions are one way of achieving that goal, but another way is to rely on *idempotence* [79].

An idempotent operation is one that you can perform multiple times, and it has the same effect as if you performed it only once. For example, setting a key in a key-value store to some fixed value is idempotent (writing the value again simply overwrites the value with an identical value), whereas incrementing a counter is not idempotent (performing the increment again means the value is incremented twice).

Even if an operation is not naturally idempotent, it can often be made idempotent with a bit of extra metadata. For example, when consuming messages from Kafka, every message has a persistent, monotonically increasing offset. When writing a value to an external database, you can include the offset of the message that triggered the last write with the value. Thus, you can tell whether an update has already been applied, and avoid performing the same update again.

The state handling in Storm's Trident is based on a similar idea [65]. Relying on idempotence implies several assumptions: restarting a failed task must replay the same messages in the same order (a log-based message broker does this), the processing must be deterministic, and no other node may concurrently update the same value [80].

When failing over from one processing node to another, fencing may be required (see "The leader and the lock" on page 293) to prevent interference from a node that is thought to be dead but is actually alive. Despite all those caveats, idempotent operations can be an effective way of achieving exactly-once semantics with only a small overhead.

Rebuilding state after a failure

Any stream process that requires state — for example, any windowed aggregations (such as counters, averages and histograms) and any tables and indexes used for joins — must ensure that this state can be recovered after a failure.

One option is to keep the state in a remote datastore and replicate it, although (as discussed in "Stream-table join (stream enrichment)" on page 457) having to query a remote database for each individual message can be slow. A better option is to keep state local to the stream processor, and to replicate it periodically. Then, when the stream processor is recovering from a failure, the new task can read the replicated state, and resume processing without data loss.

For example, Flink periodically captures snapshots of operator state and writes them to durable storage such as HDFS [76, 77]; Samza and Kafka Streams replicate state changes by sending them to a dedicated Kafka topic with log compaction, similar to change data capture [71, 81].

In some cases, it may not even be necessary to replicate the state, because it can be rebuilt from the input streams. For example, if the state consists of aggregations over a fairly short window, it may be fast enough to simply replay the input events corresponding to that window. If the state is a local replica of a database, maintained by change data capture, the database can also be rebuilt from the log-compacted change stream (see "Log compaction" on page 441).

Summary

In this chapter we have discussed event streams, what purposes they serve and how to process them. In some ways, stream processing is very much like the batch processing we discussed in Chapter 10, but done continuously on an unbounded (never-ending) stream rather than on a fixed-size input. From this perspective, message brokers and event logs serve as the streaming equivalent of a filesystem.

We spent some time comparing two types of message broker:

AMQP/JMS-style message brokers

The broker assigns individual messages to consumers, and consumers acknowledge individual messages when they have been successfully processed. Messages are deleted from the broker once they have been acknowledged. This approach is appropriate as an asynchronous form of RPC (see also "Message passing data flow" on page 132), for example in a task queue, where the exact order of message processing is not important, and where there is no need to go back and read old messages again after they have been processed.

Log-based message brokers

The broker assigns all messages in a partition to the same consumer node, and always delivers messages in the same order. Parallelism is achieved through partitioning, and consumers track their progress by checkpointing the offset of the last message they have processed. The broker retains messages on disk, so it is possible to jump back and re-read old messages if necessary.

The log-based approach has similarities to replication logs found in databases (see Chapter 5) and log-structured storage engines (see Chapter 3). We saw that this approach is especially appropriate for stream processing systems that consume input streams and generate derived state or derived output streams.

In terms of where streams come from, we discussed several possibilities: user activity events, sensors providing periodic readings, and data feeds (e.g. market data in finance) are naturally represented as streams. We saw that it can also be useful to think of the writes to a database as a stream: it can capture the changelog, i.e. the history of all changes made to a database, either implicitly through change data capture or explicitly through event sourcing. Log compaction allows the stream to retain a full copy of the contents of a database.

Representing databases as streams opens up powerful opportunities for integrating systems. You can keep derived data systems such as search indexes, caches and analytics systems continually up-to-date by consuming the log of changes and applying them to the derived system. You can even build fresh views onto existing data by starting from scratch and consuming the log of changes from the beginning all the way to the present.

The facilities for maintaining state as streams and replaying messages are also the basis for the techniques that enable streaming joins and fault tolerance in various stream processing frameworks. We discussed several purposes of stream processing, including searching for event patterns (complex event processing), computing windowed aggregations (stream analytics), and keeping derived data systems up-to-date (materialized views).

We then discussed the difficulties of reasoning about time in a stream processor, including the distinction between processing time and event timestamps, and the problem of dealing with straggler events that arrive after you thought your window was complete.

We distinguished three types of join that may appear in stream processes:

Stream-stream joins
: Matching two events that occur within some window of time — for example, two actions taken by the same user within 30 minutes of each other.

Stream-table joins
: One input stream consists of activity events, while the other is a database changelog. The changelog keeps a local copy of the database up-to-date, while activity events query the database and output an enriched activity event.

Table-table joins
: Both input streams are database changelogs. In this case, every change on one side is joined with the latest state of the other side. The result is a stream of changes to the materialized view of the two tables.

Finally, we discussed techniques for achieving fault tolerance and exactly-once semantics in a stream processor. As with batch processing, we need to discard the partial output of any failed tasks. However, since a stream process is long-running and produces output continuously, we can't simply discard all output. Instead, a finer-grained recovery mechanism can be used, based on microbatching, checkpointing, transactions, or idempotent writes.

References

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, et al.: "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," *Proceedings of the VLDB Endowment*, volume 8, number 12, pages 1792–1803, August 2015. doi:10.14778/2824032.2824076

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman: *Structure and Interpretation of Computer Programs*, Second Edition. MIT Press, July 1996. ISBN: 9780262510875, online at mitpress.mit.edu.

[3] Joseph M Hellerstein and Michael Stonebraker: *Readings in Database Systems*, Fourth Edition. MIT Press, January 2005. ISBN: 978-0-262-69314-1, redbook.cs.berkeley.edu.

[4] Don Carney, Uğur Çetintemel, Mitch Cherniack, et al.: "Monitoring Streams – A New Class of Data Management Applications," at *28th International Conference on Very Large Data Bases* (VLDB), August 2002.

[5] Vicent Martí: "Brubeck, a statsd-compatible metrics aggregator," githubengineering.com, 15 June 2015.

[6] Seth Lowenberger: "MoldUDP64 Protocol Specification V 1.00," nasdaqtrader.com, July 2009.

[7] Pieter Hintjens: *ZeroMQ – The Guide*. O'Reilly Media, March 2013. ISBN: 978-1-4493-3404-8, online at zguide.zeromq.org.

[8] Ian Malpass: "Measure Anything, Measure Everything," codeascraft.com, 15 February 2011.

[9] Dieter Plaetinck: "25 graphite, grafana and statsd gotchas," blog.raintank.io, 3 March 2016.

[10] Jeff Lindsay: "Web hooks to revolutionize the web," progrium.com, 3 May 2007.

[11] Jim N Gray: "Queues Are Databases," Microsoft Research Technical Report MSR-TR-95-56, December 1995.

[12] Matthew Sackman: "Pushing Back," lshift.net, 5 May 2016.

[13] Mark Hapner, Rich Burridge, Rahul Sharma, et al.: "JSR-343 Java Message Service (JMS) 2.0 Specification," jms-spec.java.net, March 2013.

[14] Sanjay Aiyagari, Matthew Arrott, Mark Atwell, et al.: "AMQP: Advanced Message Queuing Protocol Specification," Version 0-9-1, November 2008.

[15] "Apache Kafka 0.9 documentation," kafka.apache.org, November 2015.

[16] Jay Kreps, Neha Narkhede, and Jun Rao: "Kafka: a Distributed Messaging System for Log Processing," at *6th International Workshop on Networking Meets Databases* (NetDB), June 2011.

[17] "Amazon Kinesis Streams Developer Guide," docs.aws.amazon.com, April 2016.

[18] Leigh Stewart and Sijie Guo: "Building DistributedLog: Twitter's high-performance replicated log service," blog.twitter.com, 16 September 2015.

[19] "DistributedLog documentation," Twitter Inc., distributedlog.io, May 2016.

[20] Jay Kreps: "Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines)," engineering.linkedin.com, 27 April 2014.

[21] Kartik Paramasivam: "How We're Improving and Advancing Kafka at LinkedIn," engineering.linkedin.com, 2 September 2015.

[22] Jay Kreps: "The Log: What every software engineer should know about real-time data's unifying abstraction," engineering.linkedin.com, 16 December 2013.

[23] Shirshanka Das, Chavdar Botev, Kapil Surlaker, et al.: "All Aboard the Databus!," at *3rd ACM Symposium on Cloud Computing* (SoCC), October 2012.

[24] 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.

[25] P P S Narayan: "Sherpa update," developer.yahoo.com, 8 June 2010.

[26] Martin Kleppmann: "Bottled Water: Real-time integration of PostgreSQL and Kafka," martin.kleppmann.com, 23 April 2015.

[27] Ben Osheroff: "Introducing Maxwell, a mysql-to-kafka binlog processor," developer.zendesk.com, 20 August 2015.

[28] Randall Hauch: "Debezium 0.2.1 Released," debezium.io, 10 June 2016.

[29] "Mongoriver," Stripe, Inc., github.com, September 2014.

[30] Dan Harvey: "Change Data Capture with Mongo + Kafka," at *Hadoop Users Group UK*, August 2015.

[31] "Oracle GoldenGate 12c: Real-time access to real-time information," Oracle White Paper, March 2015.

[32] "Oracle GoldenGate Fundamentals: How Oracle GoldenGate Works," Oracle Corporation, youtube.com, November 2012.

[33] Slava Akhmechet: "Advancing the realtime web," rethinkdb.com, 27 January 2015.

[34] "Firebase Realtime Database Documentation," Google Inc., firebase.google.com, May 2016.

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

[36] Matt DeBergalis: "Meteor 0.7.0: Scalable database queries using MongoDB oplog instead of poll-and-diff," info.meteor.com, 17 December 2013.

[37] Neha Narkhede: "Announcing Kafka Connect: Building large-scale low-latency data pipelines," confluent.io, 18 February 2016.

[38] Greg Young: "CQRS and Event Sourcing," at *Code on the Beach*, August 2014.

[39] Martin Fowler: "Event Sourcing," martinfowler.com, 12 December 2005.

[40] Vaughn Vernon: *Implementing Domain-Driven Design*. Addison-Wesley Professional, February 2013. ISBN: 0321834577

[41] H V Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz: "View Maintenance Issues for the Chronicle Data Model," at *14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (PODS), pages 113–124, May 1995. doi:10.1145/212433.220201

[42] "Event Store 3.5.0 Documentation," Event Store LLP, docs.geteventstore.com, February 2016.

[43] Martin Kleppmann: *Making Sense of Stream Processing*. O'Reilly Media, May 2016.

[44] Sander Mak: "Event-sourced architectures with Akka," at *JavaOne*, September 2014.

[45] Julian Hyde: personal communication, June 2016.

[46] Ashish Gupta and Inderpal Singh Mumick: *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, May 1999. ISBN: 9780262571227

[47] Pat Helland: "Immutability Changes Everything," at *7th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2015.

[48] Martin Kleppmann: "Accounting for Computer Scientists," martin.kleppmann.com, 7 March 2011.

[49] Pat Helland: "Accountants Don't Use Erasers," blogs.msdn.com, 14 June 2007.

[50] Kartik Paramasivam: "Stream Processing Hard Problems – Part 1: Killing Lambda," engineering.linkedin.com, 27 June 2016.

[51] Martin Fowler: "CQRS," martinfowler.com, 14 July 2011.

[52] Greg Young: "CQRS Documents," cqrs.files.wordpress.com, November 2010.

[53] "Datomic Development Resources: Excision," Cognitect, Inc, docs.datomic.com.

[54] "Fossil Documentation: Deleting Content From Fossil," fossil-scm.org, 2016.

[55] Jay Kreps: "The irony of distributed systems is that data loss is really easy but deleting data is surprisingly hard." twitter.com, 30 March 2015.

[56] David C Luckham: "What's the Difference Between ESP and CEP?," complexevents.com, 1 August 2006.

[57] Srinath Perera: "How is stream processing and complex event processing (CEP) different?," quora.com, 3 December 2015.

[58] "Esper Reference, Version 5.4.0," EsperTech Inc., espertech.com, April 2016.

[59] Zubair Nabi, Eric Bouillet, Andrew Bainbridge, and Chris Thomas: "Of Streams and Storms," IBM technical report, developer.ibm.com, April 2014.

[60] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier: "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm," at *Conference on Analysis of Algorithms* (AofA), pages 137–156, June 2007.

[61] Jay Kreps: "Questioning the Lambda Architecture," oreilly.com, 2 July 2014.

[62] Ian Hellström: "An Overview of Apache Streaming Technologies," databaseline.wordpress.com, 12 March 2016.

[63] Jay Kreps: "Why local state is a fundamental primitive in stream processing," oreilly.com, 31 July 2014.

[64] Alan Woodward and Martin Kleppmann: "Real-time full-text search with Luwak and Samza," martin.kleppmann.com, 13 April 2015.

[65] "Apache Storm 1.0.1 Documentation," storm.apache.org, May 2016.

[66] Tyler Akidau: "The world beyond batch: Streaming 102," oreilly.com, 20 January 2016.

[67] Stephan Ewen: "Streaming Analytics with Apache Flink," at *Kafka Summit*, April 2016.

[68] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, et al.: "MillWheel: Fault-Tolerant Stream Processing at Internet Scale," at *39th International Conference on Very Large Data Bases* (VLDB), pages 734–746, August 2013.

[69] Alex Dean: "Improving Snowplow's understanding of time," snowplowanalytics.com, 15 September 2015.

[70] "Windowing (Azure Stream Analytics)," Microsoft Azure Reference, msdn.microsoft.com, April 2016.

[71] "State Management," Apache Samza 0.10 Documentation, samza.apache.org, December 2015.

[72] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, et al.: "Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams," at *ACM International Conference on Management of Data* (SIGMOD), June 2013. doi:10.1145/2463676.2465272

[73] Martin Kleppmann: "Samza newsfeed demo," github.com, September 2014.

[74] Ben Kirwin: "Doing the Impossible: Exactly-once Messaging Patterns in Kafka," ben.kirw.in, 28 November 2014.

[75] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica: "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters," at *4th USENIX Conference in Hot Topics in Cloud Computing* (HotCloud), June 2012.

[76] Kostas Tzoumas, Stephan Ewen, and Robert Metzger: "High-throughput, low-latency, and exactly-once stream processing with Apache Flink," data-artisans.com, 5 August 2015.

[77] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas: "Lightweight Asynchronous Snapshots for Distributed Dataflows," *arXiv:1506.08603 [cs.DC]*, 29 June 2015.

[78] Flavio Junqueira: "Making sense of exactly-once semantics," at *Strata+Hadoop World London*, June 2016.

[79] Pat Helland: "Idempotence is not a medical condition," *Communications of the ACM*, volume 55, number 5, page 56, May 2012. doi:10.1145/2160718.2160734

[80] Jay Kreps: "Re: Trying to achieve deterministic behavior on recovery/rewind," email to samza-dev mailing list, 9 September 2014.

[81] Adam Warski: "Kafka Streams - how does it fit the stream processing landscape?," softwaremill.com, 1 June 2016.