

O'REILLY®

Second
Edition

Designing Data-Intensive Applications

The Big Ideas Behind Reliable, Scalable,
and Maintainable Systems

Early
Release

RAW &
UNEDITED



Martin Kleppmann
& Chris Riccomini

O'REILLY®

Second
Edition

Designing Data-Intensive Applications

The Big Ideas Behind Reliable, Scalable,
and Maintainable Systems

Early
Release

RAW &
UNEDITED



Martin Kleppmann
& Chris Riccomini

Designing Data-Intensive Applications

The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

SECOND EDITION

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Martin Kleppmann and Chris Riccomini



Designing Data-Intensive Applications

by Martin Kleppmann and Chris Riccomini

Copyright © 2026 Martin Kleppmann and Chris Riccomini. All rights reserved.

Printed in the United States of America.

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

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

Acquisitions Editor: Aaron
Black

Development Editor: Melissa Potter	Interior Designer: David Futato
---------------------------------------	------------------------------------

Production Editor: Katherine Tozer	Cover Designer: Karen Montgomery
---------------------------------------	-------------------------------------

- March 2017: First Edition
- December 2025: Second Edition

Revision History for the Early Release

- 2024-08-27: First Release
- 2024-11-11: Second Release
- 2025-01-10: Third Release
- 2025-02-18: Fourth Release
- 2025-03-12: Fifth Release
- 2025-04-08: Sixth Release

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

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Designing Data-Intensive Applications*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors 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.

978-1-098-11900-3

Dedication

To everyone using technology and data to address the world's biggest problems.

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

—[Alan Kay](#), in interview with Dr Dobb's
Journal (2012)

Brief Table of Contents (*Not Yet Final*)

Chapter 1: Tradeoffs in Data Systems Architecture (available)

Chapter 2: Defining NonFunctional Requirements (available)

Chapter 3: Data Models and Query Languages (available)

Chapter 4: Storage and Retrieval (available)

Chapter 5: Encoding and Evolution (available)

Chapter 6: Replication (available)

Chapter 7: Partitioning (available)

Chapter 8: Transactions (available)

Chapter 9: The Trouble with Distributed Systems (unavailable)

Chapter 10: Consistency and Consensus (unavailable)

Chapter 11: Batch Processing (unavailable)

Chapter 12: Stream Processing (unavailable)

Chapter 13: Doing the Right Thing (unavailable)

...

Chapter 1. Trade-offs in Data Systems Architecture

There are no solutions, there are only trade-offs. [...] But you try to get the best trade-off you can get, and that's all you can hope for.

—[Thomas Sowell](#), Interview with Fred Barnes
(2005)

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. The GitHub repo for this book is <https://github.com/ept/ddia2-feedback>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out on GitHub.

Data is central to much application development today. With web and mobile apps, software as a service (SaaS), and cloud services, it has become normal to store data from many different users in a shared server-based data infrastructure. Data from user activity, business transactions, devices and sensors needs to be stored and made available for analysis. As users interact with an application, they both read the data that is stored, and also generate more data.

Small amounts of data, which can be stored and processed on a single machine, are often fairly easy to deal with. However, as the data volume or the rate of queries grows, it needs to be distributed across multiple machines, which introduces many challenges. As the needs of the application become more complex, it is no longer sufficient to store everything in one system, but it might be necessary to combine multiple storage or processing systems that provide different capabilities.

We call an application *data-intensive* if data management is one of the primary challenges in developing the application [1]. While in *compute-intensive* systems the challenge is parallelizing some very large computation, in data-intensive applications we usually worry more about things like storing and processing large data volumes, managing changes to data,

ensuring consistency in the face of failures and concurrency, and making sure services are highly available.

Such applications are typically built from standard building blocks that provide commonly needed functionality. For example, many applications need to:

- Store data so that they, or another application, can find it again later (*databases*)
- Remember the result of an expensive operation, to speed up reads (*caches*)
- Allow users to search data by keyword or filter it in various ways (*search indexes*)
- Handle events and data changes as soon as they occur (*stream processing*)
- Periodically crunch a large amount of accumulated data (*batch processing*)

In building an application we typically take several software systems or services, such as databases or APIs, and glue them together with some application code. If you are doing exactly what the data systems were designed for, then this process can be quite easy.

However, as your application becomes more ambitious, challenges arise. There are many database systems with different characteristics, suitable for different purposes—how do you choose which one to use? There are various approaches to caching, several ways of building search indexes, and so on—how do you reason about their trade-offs? You need to figure out which tools and which approaches are the most appropriate for the task at hand, and it can be difficult to combine tools when you need to do something that a single tool cannot do alone.

This book is a guide to help you make decisions about which technologies to use and how to combine them. As you will see, there is no one approach that is fundamentally better than others; everything has pros and cons. With this book, you will learn to ask the right questions to evaluate and compare data systems, so that you can figure out which approach will best serve the needs of your particular application.

We will start our journey by looking at some of the ways that data is typically used in organizations today. Many of the ideas here have their origin in *enterprise software* (i.e., the software needs and engineering practices of large organizations, such as big corporations and governments), since historically, only large organizations had the large data volumes that required

sophisticated technical solutions. If your data volume is small enough, you can simply keep it in a spreadsheet! However, more recently it has also become common for smaller companies and startups to manage large data volumes and build data-intensive systems.

One of the key challenges with data systems is that different people need to do very different things with data. If you are working at a company, you and your team will have one set of priorities, while another team may have entirely different goals, even though you might be working with the same dataset! Moreover, those goals might not be explicitly articulated, which can lead to misunderstandings and disagreement about the right approach.

To help you understand what choices you can make, this chapter compares several contrasting concepts, and explores their trade-offs:

- the difference between operational and analytical systems ([“Analytical versus Operational Systems”](#));
- pros and cons of cloud services and self-hosted systems ([“Cloud versus Self-Hosting”](#));
- when to move from single-node systems to distributed systems ([“Distributed versus Single-Node Systems”](#)); and

- balancing the needs of the business and the rights of the user ([Data Systems, Law, and Society](#)).

Moreover, this chapter will provide you with terminology that we will need for the rest of the book.

TERMINOLOGY: FRONTENDS AND BACKENDS

Much of what we will discuss in this book relates to *backend development*. To explain that term: for web applications, the client-side code (which runs in a web browser) is called the *frontend*, and the server-side code that handles user requests is known as the *backend*. Mobile apps are similar to frontends in that they provide user interfaces, which often communicate over the Internet with a server-side backend. Frontends sometimes manage data locally on the user's device [2], but the greatest data infrastructure challenges often lie in the backend: a frontend only needs to handle one user's data, whereas the backend manages data on behalf of *all* of the users.

A backend service is often reachable via HTTP (sometimes WebSocket); it usually consists of some application code that reads and writes data in one or more databases, and sometimes interfaces with additional data systems such as caches or message queues (which we might collectively call *data infrastructure*). The application code is often *stateless* (i.e., when it finishes handling one HTTP request, it forgets everything about that request), and any information that needs to persist from one request to another needs to be stored either on the client, or in the server-side data infrastructure.

Analytical versus Operational Systems

If you are working on data systems in an enterprise, you are likely to encounter several different types of people who work with data. The first type are *backend engineers* who build services that handle requests for reading and updating data; these services often serve external users, either directly or indirectly via other services (see [“Microservices and Serverless”](#)). Sometimes services are for internal use by other parts of the organization.

In addition to the teams managing backend services, two other groups of people typically require access to an organization’s data: *business analysts*, who generate reports about the activities of the organization in order to help the management make better decisions (*business intelligence* or *BI*), and *data scientists*, who look for novel insights in data or who create user-facing product features that are enabled by data analysis and machine learning/AI (for example, “people who bought X also bought Y” recommendations on an e-commerce website, predictive analytics such as risk scoring or spam filtering, and ranking of search results).

Although business analysts and data scientists tend to use different tools and operate in different ways, they have some things in common: both perform *analytics*, which means they look at the data that the users and backend services have generated, but they generally do not modify this data (except perhaps for fixing mistakes). They might create derived datasets in which the original data has been processed in some way. This has led to a split between two types of systems—a distinction that we will use throughout this book:

- *Operational systems* consist of the backend services and data infrastructure where data is created, for example by serving external users. Here, the application code both reads and modifies the data in its databases, based on the actions performed by the users.
- *Analytical systems* serve the needs of business analysts and data scientists. They contain a read-only copy of the data from the operational systems, and they are optimized for the types of data processing that are needed for analytics.

As we shall see in the next section, operational and analytical systems are often kept separate, for good reasons. As these systems have matured, two new specialized roles have emerged: *data engineers* and *analytics engineers*. Data engineers are the people who know how to integrate the operational and

the analytical systems, and who take responsibility for the organization's data infrastructure more widely [3]. Analytics engineers model and transform data to make it more useful for the business analysts and data scientists in an organization [4].

Many engineers specialize on either the operational or the analytical side. However, this book covers both operational and analytical data systems, since both play an important role in the lifecycle of data within an organization. We will explore in-depth the data infrastructure that is used to deliver services both to internal and external users, so that you can work better with your colleagues on the other side of this divide.

Characterizing Transaction Processing and 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.

NOTE

[Chapter 8](#) explores in detail what we mean with a transaction. This chapter uses the term loosely to refer to low-latency reads and writes.

Even though databases started being used for many different kinds of data—posts on social media, moves in a game, contacts in an address book, and many others—the basic access pattern remained similar to processing business transactions. An operational system typically looks up a small number of records by some key (this is called a *point query*). Records are inserted, updated, or deleted based on the user’s input. Because these applications are interactive, this access pattern became known as *online transaction processing* (OLTP).

However, databases also started being increasingly used for analytics, which has very different access patterns compared to OLTP. Usually an analytic query scans over a huge number of records, and calculates aggregate statistics (such as count, sum, or average) rather than returning the individual records to the user. For example, a business analyst at a supermarket chain may want to answer analytic queries such as:

- What was the total revenue of each of our stores in January?

- How many more bananas than usual did we sell during our latest promotion?
- Which brand of baby food is most often purchased together with brand X diapers?

The reports that result from these types of queries are important for business intelligence, helping the management decide what to do next. In order to differentiate this pattern of using databases from transaction processing, it has been called *online analytic processing* (OLAP) [5]. The difference between OLTP and analytics is not always clear-cut, but some typical characteristics are listed in [Table 1-1](#).

Table 1-1. Comparing characteristics of operational and analytic systems

Property	Operational systems (OLTP)	Analytical systems (OLAP)
Main read pattern	Point queries (fetch individual records by key)	Aggregate over large number of records
Main write pattern	Create, update, and delete individual records	Bulk import (ETL) or event stream
Human user example	End user of web/mobile application	Internal analyst, for decision support
Machine use example	Checking if an action is authorized	Detecting fraud/abuse patterns
Type of queries	Fixed set of queries, predefined by application	Analyst can make arbitrary queries
Data represents	Latest state of data (current point in time)	History of events that happened over time

Property	Operational systems (OLTP)	Analytical systems (OLAP)
	time)	time
Dataset size	Gigabytes to terabytes	Terabytes to petabytes

NOTE

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.

With operational systems, users are generally not allowed to construct custom SQL queries and run them on the database, since that would potentially allow them to read or modify data that they do not have permission to access. Moreover, they might write queries that are expensive to execute, and hence affect the database performance for other users. For these reasons, OLTP systems mostly run a fixed set of queries that are baked into the application code, and use one-off custom queries only occasionally for maintenance or troubleshooting. On the other hand, analytic databases usually give their users the freedom to write arbitrary SQL queries by hand, or to generate

queries automatically using a data visualization or dashboard tool such as Tableau, Looker, or Microsoft Power BI.

There is also a type of systems that is designed for analytical workloads (queries that aggregate over many records) but that are embedded into user-facing products. This category is known as *product analytics* or *real-time analytics*, and systems designed for this type of use include Pinot, Druid, and ClickHouse [6].

Data Warehousing

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 both types of 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 system instead. This separate database was called a *data warehouse*.

A large enterprise may have dozens, even hundreds, of online transaction processing systems: systems powering the customer-facing website, controlling point of sale (checkout) systems in physical stores, tracking inventory in warehouses, planning routes for vehicles, managing suppliers, administering

employees, and performing many other tasks. Each of these systems is complex and needs a team of people to maintain it, so these systems end up operating mostly independently from each other.

It is usually undesirable for business analysts and data scientists to directly query these OLTP systems, for several reasons:

- the data of interest may be spread across multiple operational systems, making it difficult to combine those datasets in a single query (a problem known as *data silos*);
- the kinds of schemas and data layouts that are good for OLTP are less well suited for analytics (see [“Stars and Snowflakes: Schemas for Analytics”](#));
- analytic queries can be quite expensive, and running them on an OLTP database would impact the performance for other users; and
- the OLTP systems might reside in a separate network that users are not allowed direct access to for security or compliance reasons.

A *data warehouse*, by contrast, is a separate database that analysts can query to their hearts' content, without affecting OLTP operations [7]. As we shall see in [Chapter 4](#), data

warehouses often store data in a way that is very different from OLTP databases, in order to optimize for the types of queries that are common in analytics.

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 data warehouse is known as *Extract–Transform–Load* (ETL) and is illustrated in [Figure 1-1](#). Sometimes the order of the *transform* and *load* steps is swapped (i.e., the transformation is done in the data warehouse, after loading), resulting in *ELT*.

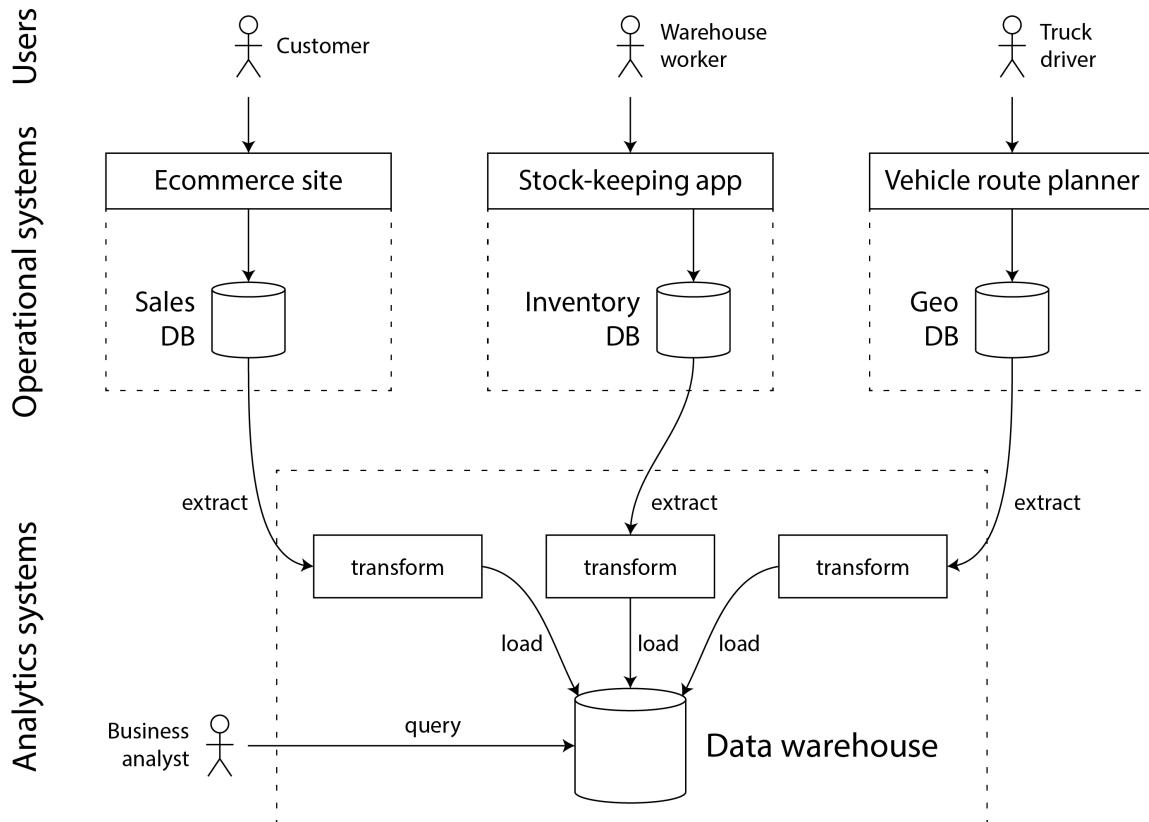


Figure 1-1. Simplified outline of ETL into a data warehouse.

In some cases the data sources of the ETL processes are external SaaS products such as customer relationship management (CRM), email marketing, or credit card processing systems. In those cases, you do not have direct access to the original database, since it is accessible only via the software vendor's API. Bringing the data from these external systems into your own data warehouse can enable analyses that are not possible via the SaaS API. ETL for SaaS APIs is often implemented by specialist data connector services such as Fivetran, Singer, or AirByte.

Some database systems offer *hybrid transactional/analytic processing* (HTAP), which aims to enable OLTP and analytics in a single system without requiring ETL from one system into another [8, 9]. However, many HTAP systems internally consist of an OLTP system coupled with a separate analytical system, hidden behind a common interface—so the distinction between the two remains important for understanding how these systems work.

Moreover, even though HTAP exists, it is common to have a separation between transactional and analytic systems due to their different goals and requirements. In particular, it is considered good practice for each operational system to have its own database (see “[Microservices and Serverless](#)”), leading to hundreds of separate operational databases; on the other hand, an enterprise usually has a single data warehouse, so that business analysts can combine data from several operational systems in a single query.

HTAP therefore does not replace data warehouses. Rather, it is useful in scenarios where the same application needs to both perform analytics queries that scan a large number of rows, and also read and update individual records with low latency. Fraud detection can involve such workloads, for example [10].

The separation between operational and analytical systems is part of a wider trend: as workloads have become more demanding, systems have become more specialized and optimized for particular workloads. General-purpose systems can handle small data volumes comfortably, but the greater the scale, the more specialized systems tend to become [11].

From data warehouse to data lake

A data warehouse often uses a *relational* data model that is queried through SQL (see [Chapter 3](#)), perhaps using specialized business intelligence software. This model works well for the types of queries that business analysts need to make, but it is less well suited to the needs of data scientists, who might need to perform tasks such as:

- Transform data into a form that is suitable for training a machine learning model; often this requires turning the rows and columns of a database table into a vector or matrix of numerical values called *features*. The process of performing this transformation in a way that maximizes the performance of the trained model is called *feature engineering*, and it often requires custom code that is difficult to express using SQL.

- Take textual data (e.g., reviews of a product) and use natural language processing techniques to try to extract structured information from it (e.g., the sentiment of the author, or which topics they mention). Similarly, they might need to extract structured information from photos using computer vision techniques.

Although there have been efforts to add machine learning operators to a SQL data model [12] and to build efficient machine learning systems on top of a relational foundation [13], many data scientists prefer not to work in a relational database such as a data warehouse. Instead, many prefer to use Python data analysis libraries such as pandas and scikit-learn, statistical analysis languages such as R, and distributed analytics frameworks such as Spark [14]. We discuss these further in “[Dataframes, Matrices, and Arrays](#)”.

Consequently, organizations face a need to make data available in a form that is suitable for use by data scientists. The answer is a *data lake*: a centralized data repository that holds a copy of any data that might be useful for analysis, obtained from operational systems via ETL processes. The difference from a data warehouse is that a data lake simply contains files, without imposing any particular file format or data model. Files in a data lake might be collections of database records, encoded

using a file format such as Avro or Parquet (see [Chapter 5](#)), but they can equally well contain text, images, videos, sensor readings, sparse matrices, feature vectors, genome sequences, or any other kind of data [15]. Besides being more flexible, this is also often cheaper than relational data storage, since the data lake can use commoditized file storage such as object stores (see [“Cloud-Native System Architecture”](#)).

ETL processes have been generalized to *data pipelines*, and in some cases the data lake has become an intermediate stop on the path from the operational systems to the data warehouse. The data lake contains data in a “raw” form produced by the operational systems, without the transformation into a relational data warehouse schema. This approach has the advantage that each consumer of the data can transform the raw data into a form that best suits their needs. It has been dubbed the *sushi principle*: “raw data is better” [16].

Besides loading data from a data lake into a separate data warehouse, it is also possible to run typical data warehousing workloads (SQL queries and business analytics) directly on the files in the data lake, alongside data science/machine learning workloads. This architecture is known as a *data lakehouse*, and it requires a query execution engine and a metadata (e.g., schema management) layer that extend the data lake’s file

storage [17]. Apache Hive, Spark SQL, Presto, and Trino are examples of this approach.

Beyond the data lake

As analytics practices have matured, organizations have been increasingly paying attention to the management and operations of analytics systems and data pipelines, as captured for example in the DataOps manifesto [18]. Part of this are issues of governance, privacy, and compliance with regulation such as GDPR and CCPA, which we discuss in [Data Systems, Law, and Society](#) and [Link to Come].

Moreover, analytical data is increasingly made available not only as files and relational tables, but also as streams of events (see [Link to Come]). With file-based data analysis you can re-run the analysis periodically (e.g., daily) in order to respond to changes in the data, but stream processing allows analytics systems to respond to events much faster, on the order of seconds. Depending on the application and how time-sensitive it is, a stream processing approach can be valuable, for example to identify and block potentially fraudulent or abusive activity.

In some cases the outputs of analytics systems are made available to operational systems (a process sometimes known

as *reverse ETL* [19]). For example, a machine-learning model that was trained on data in an analytics system may be deployed to production, so that it can generate recommendations for end-users, such as “people who bought X also bought Y”. Such deployed outputs of analytics systems are also known as *data products* [20]. Machine learning models can be deployed to operational systems using specialized tools such as TFX, Kubeflow, or MLflow.

Systems of Record and Derived Data

Related to the distinction between operational and analytical systems, this book also distinguishes between *systems of record* and *derived data systems*. These terms are useful because they can help you clarify the flow of data through a system:

Systems of record

A system of record, also known as *source of truth*, holds the authoritative or *canonical* version of some 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*; see [“Normalization, Denormalization, and Joins”](#)). 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 recreate 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, materialized views, transformed data representations, and models trained on a dataset also fall into this category.

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. You can derive several different datasets from a single source, enabling you to look at the data from different “points of view.”

Analytical systems are usually derived data systems, because they are consumers of data created elsewhere. Operational services may contain a mixture of systems of record and derived data systems. The systems of record are the primary

databases to which data is first written, whereas the derived data systems are the indexes and caches that speed up common read operations, especially for queries that the system of record cannot answer efficiently.

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.

When the data in one system is derived from the data in another, you need a process for updating the derived data when the original in the system of record changes. Unfortunately, many databases are designed based on the assumption that your application only ever needs to use that one database, and they do not make it easy to integrate multiple systems in order to propagate such updates. In [Link to Come] we will discuss approaches to *data integration*, which allow us to compose multiple data systems to achieve things that one system alone cannot do.

That brings us to the end of our comparison of analytics and transaction processing. In the next section, we will examine another trade-off that you might have already seen debated multiple times.

Cloud versus Self-Hosting

With anything that an organization needs to do, one of the first questions is: should it be done in-house, or should it be outsourced? Should you build or should you buy?

Ultimately, this is a question about business priorities. The received management wisdom is that things that are a core competency or a competitive advantage of your organization should be done in-house, whereas things that are non-core, routine, or commonplace should be left to a vendor [21]. To give an extreme example, most companies do not generate their own electricity (unless they are an energy company, and leaving aside emergency backup power), since it is cheaper to buy electricity from the grid.

With software, two important decisions to be made are who builds the software and who deploys it. There is a spectrum of possibilities that outsource each decision to various degrees, as

illustrated in [Figure 1-2](#). At one extreme is bespoke software that you write and run in-house; at the other extreme are widely-used cloud services or Software as a Service (SaaS) products that are implemented and operated by an external vendor, and which you only access through a web interface or API.

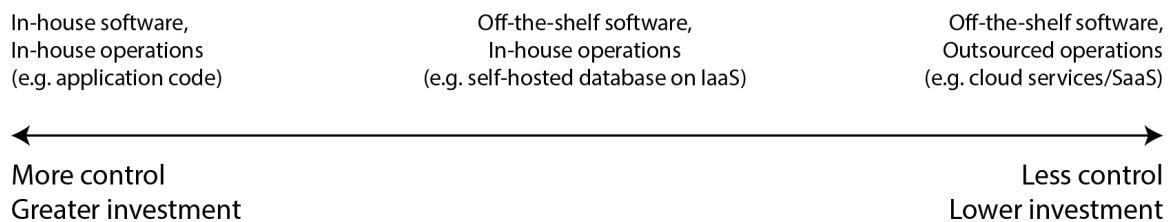


Figure 1-2. A spectrum of types of software and its operations.

The middle ground is off-the-shelf software (open source or commercial) that you *self-host*, i.e., deploy yourself—for example, if you download MySQL and install it on a server you control. This could be on your own hardware (often called *on-premises*, even if the server is actually in a rented datacenter rack and not literally on your own premises), or on a virtual machine in the cloud (*Infrastructure as a Service* or IaaS). There are still more points along this spectrum, e.g., taking open source software and running a modified version of it.

Separately from this spectrum there is also the question of *how* you deploy services, either in the cloud or on-premises—for

example, whether you use an orchestration framework such as Kubernetes. However, choice of deployment tooling is out of scope of this book, since other factors have a greater influence on the architecture of data systems.

Pros and Cons of Cloud Services

Using a cloud service, rather than running comparable software yourself, essentially outsources the operation of that software to the cloud provider. There are good arguments for and against cloud services. Cloud providers claim that using their services saves you time and money, and allows you to move faster compared to setting up your own infrastructure.

Whether a cloud service is actually cheaper and easier than self-hosting depends very much on your skills and the workload on your systems. If you already have experience setting up and operating the systems you need, and if your load is quite predictable (i.e., the number of machines you need does not fluctuate wildly), then it's often cheaper to buy your own machines and run the software on them yourself [22, 23].

On the other hand, if you need a system that you don't already know how to deploy and operate, then adopting a cloud service is often easier and quicker than learning to manage the system

yourself. If you have to hire and train staff specifically to maintain and operate the system, that can get very expensive. You still need an operations team when you're using the cloud (see [“Operations in the Cloud Era”](#)), but outsourcing the basic system administration can free up your team to focus on higher-level concerns.

When you outsource the operation of a system to a company that specializes in running that service, that can potentially result in a better service, since the provider gains operational expertise from providing the service to many customers. On the other hand, if you run the service yourself, you can configure and tune it to perform well on your particular workload; it is unlikely that a cloud service would be willing to make such customizations on your behalf.

Cloud services are particularly valuable if the load on your systems varies a lot over time. If you provision your machines to be able to handle peak load, but those computing resources are idle most of the time, the system becomes less cost-effective. In this situation, cloud services have the advantage that they can make it easier to scale your computing resources up or down in response to changes in demand.

For example, analytics systems often have extremely variable load: running a large analytical query quickly requires a lot of computing resources in parallel, but once the query completes, those resources sit idle until the user makes the next query. Predefined queries (e.g., for daily reports) can be enqueued and scheduled to smooth out the load, but for interactive queries, the faster you want them to complete, the more variable the workload becomes. If your dataset is so large that querying it quickly requires significant computing resources, using the cloud can save money, since you can return unused resources to the provider rather than leaving them idle. For smaller datasets, this difference is less significant.

The biggest downside of a cloud service is that you have no control over it:

- If it is lacking a feature you need, all you can do is to politely ask the vendor whether they will add it; you generally cannot implement it yourself.
- If the service goes down, all you can do is to wait for it to recover.
- If you are using the service in a way that triggers a bug or causes performance problems, it will be difficult for you to diagnose the issue. With software that you run yourself, you can get performance metrics and debugging information

from the operating system to help you understand its behavior, and you can look at the server logs, but with a service hosted by a vendor you usually do not have access to these internals.

- Moreover, if the service shuts down or becomes unacceptably expensive, or if the vendor decides to change their product in a way you don't like, you are at their mercy—continuing to run an old version of the software is usually not an option, so you will be forced to migrate to an alternative service [24]. This risk is mitigated if there are alternative services that expose a compatible API, but for many cloud services there are no standard APIs, which raises the cost of switching, making vendor lock-in a problem.
- The cloud provider needs to be trusted to keep the data secure, which can complicate the process of complying with privacy and security regulations.

Despite all these risks, it has become more and more popular for organizations to build new applications on top of cloud services, or adopting a hybrid approach in which cloud services are used for some aspects of a system. However, cloud services will not subsume all in-house data systems: many older systems predate the cloud, and for any services that have specialist requirements that existing cloud services cannot meet, in-house systems remain necessary. For example, very latency-sensitive

applications such as high-frequency trading require full control of the hardware.

Cloud-Native System Architecture

Besides having a different economic model (subscribing to a service instead of buying hardware and licensing software to run on it), the rise of the cloud has also had a profound effect on how data systems are implemented on a technical level. The term *cloud-native* is used to describe an architecture that is designed to take advantage of cloud services.

In principle, almost any software that you can self-host could also be provided as a cloud service, and indeed such managed services are now available for many popular data systems. However, systems that have been designed from the ground up to be cloud-native have been shown to have several advantages: better performance on the same hardware, faster recovery from failures, being able to quickly scale computing resources to match the load, and supporting larger datasets [[25](#), [26](#), [27](#)]. [Table 1-2](#) lists some examples of both types of systems.

Table 1-2. Examples of self-hosted and cloud-native database systems

Category	Self-hosted systems	Cloud-native systems
Operational/OLTP	MySQL, PostgreSQL, MongoDB	AWS Aurora [25], Azure SQL DB Hyperscale [26], Google Cloud Spanner
Analytical/OLAP	Teradata, ClickHouse, Spark	Snowflake [27], Google BigQuery, Azure Synapse Analytics

Layering of cloud services

Many self-hosted data systems have very simple system requirements: they run on a conventional operating system such as Linux or Windows, they store their data as files on the filesystem, and they communicate via standard network protocols such as TCP/IP. A few systems depend on special hardware such as GPUs (for machine learning) or RDMA network interfaces, but on the whole, self-hosted software tends to use very generic computing resources: CPU, RAM, a filesystem, and an IP network.

In a cloud, this type of software can be run on an Infrastructure-as-a-Service environment, using one or more virtual machines (or *instances*) with a certain allocation of CPUs, memory, disk, and network bandwidth. Compared to physical machines, cloud instances can be provisioned faster and they come in a greater variety of sizes, but otherwise they are similar to a traditional computer: you can run any software you like on it, but you are responsible for administering it yourself.

In contrast, the key idea of cloud-native services is to use not only the computing resources managed by your operating system, but also to build upon lower-level cloud services to create higher-level services. For example:

- *Object storage* services such as Amazon S3, Azure Blob Storage, and Cloudflare R2 store large files. They provide more limited APIs than a typical filesystem (basic file reads and writes), but they have the advantage that they hide the underlying physical machines: the service automatically distributes the data across many machines, so that you don't have to worry about running out of disk space on any one machine. Even if some machines or their disks fail entirely, no data is lost.

- Many other services are in turn built upon object storage and other cloud services: for example, Snowflake is a cloud-based analytic database (data warehouse) that relies on S3 for data storage [27], and some other services in turn build upon Snowflake.

As always with abstractions in computing, there is no one right answer to what you should use. As a general rule, higher-level abstractions tend to be more oriented towards particular use cases. If your needs match the situations for which a higher-level system is designed, using the existing higher-level system will probably provide what you need with much less hassle than building it yourself from lower-level systems. On the other hand, if there is no high-level system that meets your needs, then building it yourself from lower-level components is the only option.

Separation of storage and compute

In traditional computing, disk storage is regarded as durable (we assume that once something is written to disk, it will not be lost); to tolerate the failure of an individual hard disk, RAID is often used to maintain copies of the data on several disks. In the cloud, compute instances (virtual machines) may also have local disks attached, but cloud-native systems typically treat

these disks more like an ephemeral cache, and less like long-term storage. This is because the local disk becomes inaccessible if the associated instance fails, or if the instance is replaced with a bigger or a smaller one (on a different physical machine) in order to adapt to changes in load.

As an alternative to local disks, cloud services also offer virtual disk storage that can be detached from one instance and attached to a different one (Amazon EBS, Azure managed disks, and persistent disks in Google Cloud). Such a virtual disk is not actually a physical disk, but rather a cloud service provided by a separate set of machines, which emulates the behavior of a disk (a *block device*, where each block is typically 4 KiB in size). This technology makes it possible to run traditional disk-based software in the cloud, but the block device emulation introduces overheads that can be avoided in systems that are designed from the ground up for the cloud [25]. It also makes the application very sensitive to network glitches, since every I/O on the virtual block device is actually a network call.

To address this problem, cloud-native services generally avoid using virtual disks, and instead build on dedicated storage services that are optimized for particular workloads. Object storage services such as S3 are designed for long-term storage of fairly large files, ranging from hundreds of kilobytes to

several gigabytes in size. The individual rows or values stored in a database are typically much smaller than this; cloud databases therefore typically manage smaller values in a separate service, and store larger data blocks (containing many individual values) in an object store [26, 28]. We will see ways of doing this in [Chapter 4](#).

In a traditional systems architecture, the same computer is responsible for both storage (disk) and computation (CPU and RAM), but in cloud-native systems, these two responsibilities have become somewhat separated or *disaggregated* [9, 27, 29, 30]: for example, S3 only stores files, and if you want to analyze that data, you will have to run the analysis code somewhere outside of S3. This implies transferring the data over the network, which we will discuss further in [“Distributed versus Single-Node Systems”](#).

Moreover, cloud-native systems are often *multitenant*, which means that rather than having a separate machine for each customer, data and computation from several different customers are handled on the same shared hardware by the same service [31]. Multitenancy can enable better hardware utilization, easier scalability, and easier management by the cloud provider, but it also requires careful engineering to

ensure that one customer's activity does not affect the performance or security of the system for other customers [32].

Operations in the Cloud Era

Traditionally, the people managing an organization's server-side data infrastructure were known as *database administrators* (DBAs) or *system administrators* (sysadmins). More recently, many organizations have tried to integrate the roles of software development and operations into teams with a shared responsibility for both backend services and data infrastructure; the *DevOps* philosophy has guided this trend. *Site Reliability Engineers* (SREs) are Google's implementation of this idea [33].

The role of operations is to ensure services are reliably delivered to users (including configuring infrastructure and deploying applications), and to ensure a stable production environment (including monitoring and diagnosing any problems that may affect reliability). For self-hosted systems, operations traditionally involves a significant amount of work at the level of individual machines, such as capacity planning (e.g., monitoring available disk space and adding more disks before you run out of space), provisioning new machines,

moving services from one machine to another, and installing operating system patches.

Many cloud services present an API that hides the individual machines that actually implement the service. For example, cloud storage replaces fixed-size disks with *metered billing*, where you can store data without planning your capacity needs in advance, and you are then charged based on the space actually used. Moreover, many cloud services remain highly available, even when individual machines have failed (see [“Reliability and Fault Tolerance”](#)).

This shift in emphasis from individual machines to services has been accompanied by a change in the role of operations. The high-level goal of providing a reliable service remains the same, but the processes and tools have evolved. The DevOps/SRE philosophy places greater emphasis on:

- automation—preferring repeatable processes over manual one-off jobs,
- preferring ephemeral virtual machines and services over long running servers,
- enabling frequent application updates,
- learning from incidents, and

- preserving the organization's knowledge about the system, even as individual people come and go [34].

With the rise of cloud services, there has been a bifurcation of roles: operations teams at infrastructure companies specialize in the details of providing a reliable service to a large number of customers, while the customers of the service spend as little time and effort as possible on infrastructure [35].

Customers of cloud services still require operations, but they focus on different aspects, such as choosing the most appropriate service for a given task, integrating different services with each other, and migrating from one service to another. Even though metered billing removes the need for capacity planning in the traditional sense, it's still important to know what resources you are using for which purpose, so that you don't waste money on cloud resources that are not needed: capacity planning becomes financial planning, and performance optimization becomes cost optimization [36]. Moreover, cloud services do have resource limits or *quotas* (such as the maximum number of processes you can run concurrently), which you need to know about and plan for before you run into them [37].

Adopting a cloud service can be easier and quicker than running your own infrastructure, although even here there is a cost in learning how to use it, and perhaps working around its limitations. Integration between different services becomes a particular challenge as a growing number of vendors offers an ever broader range of cloud services targeting different use cases [38, 39]. ETL (see “[Data Warehousing](#)”) is only part of the story; operational cloud services also need to be integrated with each other. At present, there is a lack of standards that would facilitate this sort of integration, so it often involves significant manual effort.

Other operational aspects that cannot fully be outsourced to cloud services include maintaining the security of an application and the libraries it uses, managing the interactions between your own services, monitoring the load on your services, and tracking down the cause of problems such as performance degradations or outages. While the cloud is changing the role of operations, the need for operations is as great as ever.

Distributed versus Single-Node

Systems

A system that involves several machines communicating via a network is called a *distributed system*. Each of the processes participating in a distributed system is called a *node*. There are various reasons why you might want a system to be distributed:

Inherently distributed systems

If an application involves two or more interacting users, each using their own device, then the system is unavoidably distributed: the communication between the devices will have to go via a network.

Requests between cloud services

If data is stored in one service but processed in another, it must be transferred over the network from one service to the other.

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. See “[Reliability and Fault Tolerance](#)” and [Chapter 6](#) on replication.

Scalability

If your data volume or computing requirements grow bigger than a single machine can handle, you can potentially spread the load across multiple machines. See “[Scalability](#)”.

Latency

If you have users around the world, you might want to have servers in various regions worldwide so that each user can be served from a server that is geographically close to them. That avoids the users having to wait for network packets to travel halfway around the world to answer their requests. See “[Describing Performance](#)”.

Elasticity

If your application is busy at some times and idle at other times, a cloud deployment can scale up or down to meet the demand, so that you pay only for resources you are actively using. This is more difficult on a single machine, which needs to be provisioned to handle the maximum load, even at times when it is barely used.

Using specialized hardware

Different parts of the system can take advantage of different types of hardware to match their workload. For example, an object store may use machines with many disks but few CPUs, whereas a data analysis system may use machines with lots of CPU and memory but no disks, and a machine learning system may use machines with GPUs (which are much more efficient than CPUs for training deep neural networks and other machine learning tasks).

Legal compliance

Some countries have data residency laws that require data about people in their jurisdiction to be stored and processed geographically within that country [40]. The scope of these rules varies—for example, in some cases it applies only to medical or financial data, while other cases are broader. A service with users in several such jurisdictions will therefore have to distribute their data across servers in several locations.

These reasons apply both to services that you write yourself (application code) and services consisting of off-the-shelf software (such as databases).

Problems with Distributed Systems

Distributed systems also have downsides. Every request and API call that goes via the network needs to deal with the possibility of failure: the network may be interrupted, or the service may be overloaded or crashed, and therefore any request may time out without receiving a response. In this case, we don't know whether the service received the request, and simply retrying it might not be safe. We will discuss these problems in detail in [Link to Come].

Although datacenter networks are fast, making a call to another service is still vastly slower than calling a function in the same process [41]. When operating on large volumes of data, rather than transferring the data from storage to a separate machine that processes it, it can be faster to bring the computation to the machine that already has the data [42]. More nodes are not always faster: in some cases, a simple single-threaded program on one computer can perform significantly better than a cluster with over 100 CPU cores [43].

Troubleshooting a distributed system is often difficult: if the system is slow to respond, how do you figure out where the problem lies? Techniques for diagnosing problems in distributed systems are developed under the heading of

observability [44, 45], which involves collecting data about the execution of a system, and allowing it to be queried in ways that allows both high-level metrics and individual events to be analyzed. *Tracing* tools such as OpenTelemetry, Zipkin, and Jaeger allow you to track which client called which server for which operation, and how long each call took [46].

Databases provide various mechanisms for ensuring data consistency, as we shall see in [Chapter 6](#) and [Chapter 8](#). However, when each service has its own database, maintaining consistency of data across those different services becomes the application's problem. Distributed transactions, which we explore in [Link to Come], are a possible technique for ensuring consistency, but they are rarely used in a microservices context because they run counter to the goal of making services independent from each other, and many databases don't support them [47].

For all these reasons, if you can do something on a single machine, this is often much simpler and cheaper compared to setting up a distributed system [23, 43, 48]. CPUs, memory, and disks have grown larger, faster, and more reliable. When combined with single-node databases such as DuckDB, SQLite, and KùzuDB, many workloads can now run on a single node. We will explore more on this topic in [Chapter 4](#).

Microservices and Serverless

The most common way of distributing a system across multiple machines is to divide them into clients and servers, and let the clients make requests to the servers. Most commonly HTTP is used for this communication, as we will discuss in [“Dataflow Through Services: REST and RPC”](#). The same process may be both a server (handling incoming requests) and a client (making outbound requests to other services).

This way of building applications has traditionally been called a *service-oriented architecture* (SOA); more recently the idea has been refined into a *microservices* architecture [49, 50]. In this architecture, a service has one well-defined purpose (for example, in the case of S3, this would be file storage); each service exposes an API that can be called by clients via the network, and each service has one team that is responsible for its maintenance. A complex application can thus be decomposed into multiple interacting services, each managed by a separate team.

There are several advantages to breaking down a complex piece of software into multiple services: each service can be updated independently, reducing coordination effort among teams; each service can be assigned the hardware resources it needs; and by

hiding the implementation details behind an API, the service owners are free to change the implementation without affecting clients. In terms of data storage, it is common for each service to have its own databases, and not to share databases between services: sharing a database would effectively make the entire database structure a part of the service's API, and then that structure would be difficult to change. Shared databases could also cause one service's queries to negatively impact the performance of other services.

On the other hand, having many services can itself breed complexity: each service requires infrastructure for deploying new releases, adjusting the allocated hardware resources to match the load, collecting logs, monitoring service health, and alerting an on-call engineer in the case of a problem.

Orchestration frameworks such as Kubernetes have become a popular way of deploying services, since they provide a foundation for this infrastructure. Testing a service during development can be complicated, since you also need to run all the other services that it depends on.

Microservice APIs can be challenging to evolve. Clients that call an API expect the API to have certain fields. Developers might wish to add or remove fields to an API as business needs change, but doing so can cause clients to fail. Worse still, such

failures are often not discovered until late in the development cycle when the updated service API is deployed to a staging or production environment. API description standards such as OpenAPI and gRPC help manage the relationship between client and server APIs; we discuss these further in [Chapter 5](#).

Microservices are primarily a technical solution to a people problem: allowing different teams to make progress independently without having to coordinate with each other. This is valuable in a large company, but in a small company where there are not many teams, using microservices is likely to be unnecessary overhead, and it is preferable to implement the application in the simplest way possible [\[49\]](#).

Serverless, or *function-as-a-service* (FaaS), is another approach to deploying services, in which the management of the infrastructure is outsourced to a cloud vendor [\[32\]](#). When using virtual machines, you have to explicitly choose when to start up or shut down an instance; in contrast, with the serverless model, the cloud provider automatically allocates and frees hardware resources as needed, based on the incoming requests to your service [\[51\]](#). Serverless deployment shifts more of the operational burden to cloud providers and enables flexible billing by usage rather than machine instances. To offer such benefits, many serverless infrastructure providers impose a

time limit on function execution, limit runtime environments, and might suffer from slow start times when a function is first invoked. The term “serverless” can also be misleading: each serverless function execution still runs on a server, but subsequent executions might run on a different one. Moreover, infrastructure such as BigQuery and various Kafka offerings have adopted “serverless” terminology to signal that their services auto-scale and that they bill by usage rather than machine instances.

Just like cloud storage replaced capacity planning (deciding in advance how many disks to buy) with a metered billing model, the serverless approach is bringing metered billing to code execution: you only pay for the time that your application code is actually running, rather than having to provision resources in advance.

Cloud Computing versus Supercomputing

Cloud computing is not the only way of building large-scale computing systems; an alternative is *high-performance computing* (HPC), also known as *supercomputing*. Although there are overlaps, HPC often has different priorities and uses different techniques compared to cloud computing and enterprise datacenter systems. Some of those differences are:

- Supercomputers are typically used for computationally intensive scientific computing tasks, such as weather forecasting, climate modeling, molecular dynamics (simulating the movement of atoms and molecules), complex optimization problems, and solving partial differential equations. On the other hand, cloud computing tends to be used for online services, business data systems, and similar systems that need to serve user requests with high availability.
- A supercomputer typically runs large batch jobs that checkpoint the state of their computation to disk from time to time. If a node fails, a common solution is to simply stop the entire cluster workload, repair the faulty node, and then restart the computation from the last checkpoint [52, 53]. With cloud services, it is usually not desirable to stop the entire cluster, since the services need to continually serve users with minimal interruptions.
- Supercomputer nodes typically communicate through shared memory and remote direct memory access (RDMA), which support high bandwidth and low latency, but assume a high level of trust among the users of the system [54]. In cloud computing, the network and the machines are often shared by mutually untrusting organizations, requiring stronger

security mechanisms such as resource isolation (e.g., virtual machines), encryption and authentication.

- Cloud datacenter networks are often based on IP and Ethernet, arranged in Clos topologies to provide high bisection bandwidth—a commonly used measure of a network’s overall performance [52, 55]. Supercomputers often use specialized network topologies, such as multi-dimensional meshes and toruses [56], which yield better performance for HPC workloads with known communication patterns.
- Cloud computing allows nodes to be distributed across multiple geographic regions, whereas supercomputers generally assume that all of their nodes are close together.

Large-scale analytics systems sometimes share some characteristics with supercomputing, which is why it can be worth knowing about these techniques if you are working in this area. However, this book is mostly concerned with services that need to be continually available, as discussed in “[Reliability and Fault Tolerance](#)”.

Data Systems, Law, and Society

So far you've seen in this chapter that the architecture of data systems is influenced not only by technical goals and requirements, but also by the human needs of the organizations that they support. Increasingly, data systems engineers are realizing that serving the needs of their own business is not enough: we also have a responsibility towards society at large.

One particular concern are systems that store data about people and their behavior. Since 2018 the *General Data Protection Regulation* (GDPR) has given residents of many European countries greater control and legal rights over their personal data, and similar privacy regulation has been adopted in various other countries and states around the world, including for example the California Consumer Privacy Act (CCPA). Regulations around AI, such as the *EU AI Act*, place further restrictions on how personal data can be used.

Moreover, even in areas that are not directly subject to regulation, there is increasing recognition of the effects that computer systems have on people and society. Social media has changed how individuals consume news, which influences their political opinions and hence may affect the outcome of

elections. Automated systems increasingly make decisions that have profound consequences for individuals, such as deciding who should be given a loan or insurance coverage, who should be invited to a job interview, or who should be suspected of a crime [57].

Everyone who works on such systems shares a responsibility for considering the ethical impact and ensuring that they comply with relevant law. It is not necessary for everybody to become an expert in law and ethics, but a basic awareness of legal and ethical principles is just as important as, say, some foundational knowledge in distributed systems.

Legal considerations are influencing the very foundations of how data systems are being designed [58]. For example, the GDPR grants individuals the right to have their data erased on request (sometimes known as the *right to be forgotten*). However, as we shall see in this book, many data systems rely on immutable constructs such as append-only logs as part of their design; how can we ensure deletion of some data in the middle of a file that is supposed to be immutable? How do we handle deletion of data that has been incorporated into derived datasets (see “[Systems of Record and Derived Data](#)”), such as training data for machine learning models? Answering these questions creates new engineering challenges.

At present we don't have clear guidelines on which particular technologies or system architectures should be considered "GDPR-compliant" or not. The regulation deliberately does not mandate particular technologies, because these may quickly change as technology progresses. Instead, the legal texts set out high-level principles that are subject to interpretation. This means that there are no simple answers to the question of how to comply with privacy regulation, but we will look at some of the technologies in this book through this lens.

In general, we store data because we think that its value is greater than the costs of storing it. However, it is worth remembering that the costs of storage are not just the bill you pay for Amazon S3 or another service: the cost-benefit calculation should also take into account the risks of liability and reputational damage if the data were to be leaked or compromised by adversaries, and the risk of legal costs and fines if the storage and processing of the data is found not to be compliant with the law [48].

Governments or police forces might also compel companies to hand over data. When there is a risk that the data may reveal criminalized behaviors (for example, homosexuality in several Middle Eastern and African countries, or seeking an abortion in several US states), storing that data creates real safety risks for

users. Travel to an abortion clinic, for example, could easily be revealed by location data, perhaps even by a log of the user's IP addresses over time (which indicate approximate location).

Once all the risks are taken into account, it might be reasonable to decide that some data is simply not worth storing, and that it should therefore be deleted. This principle of *data minimization* (sometimes known by the German term *Datensparsamkeit*) runs counter to the "big data" philosophy of storing lots of data speculatively in case it turns out to be useful in the future [59]. But it fits with the GDPR, which mandates that personal data may only be collected for a specified, explicit purpose, that this data may not later be used for any other purpose, and that the data must not be kept for longer than necessary for the purposes for which it was collected [60].

Businesses have also taken notice of privacy and safety concerns. Credit card companies require payment processing businesses to adhere to strict payment card industry (PCI) standards. Processors undergo frequent evaluations from independent auditors to verify continued compliance. Software vendors have also seen increased scrutiny. Many buyers now require their vendors to comply with Service Organization Control (SOC) Type 2 standards. As with PCI compliance, vendors undergo third party audits to verify adherence.

Generally, it is important to balance the needs of your business against the needs of the people whose data you are collecting and processing. There is much more to this topic; in [Link to Come] we will go deeper into the topics of ethics and legal compliance, including the problems of bias and discrimination.

Summary

The theme of this chapter has been to understand trade-offs: that is, to recognize that for many questions there is not one right answer, but several different approaches that each have various pros and cons. We explored some of the most important choices that affect the architecture of data systems, and introduced terminology that will be needed throughout the rest of this book.

We started by making a distinction between operational (transaction-processing, OLTP) and analytical (OLAP) systems, and saw their different characteristics: not only managing different types of data with different access patterns, but also serving different audiences. We encountered the concept of a data warehouse and data lake, which receive data feeds from operational systems via ETL. In [Chapter 4](#) we will see that operational and analytical systems often use very different

internal data layouts because of the different types of queries they need to serve.

We then compared cloud services, a comparatively recent development, to the traditional paradigm of self-hosted software that has previously dominated data systems architecture. Which of these approaches is more cost-effective depends a lot on your particular situation, but it's undeniable that cloud-native approaches are bringing big changes to the way data systems are architected, for example in the way they separate storage and compute.

Cloud systems are intrinsically distributed, and we briefly examined some of the trade-offs of distributed systems compared to using a single machine. There are situations in which you can't avoid going distributed, but it's advisable not to rush into making a system distributed if it's possible to keep it on a single machine. In [Link to Come] and [Link to Come] we will cover the challenges with distributed systems in more detail.

Finally, we saw that data systems architecture is determined not only by the needs of the business deploying the system, but also by privacy regulation that protects the rights of the people whose data is being processed—an aspect that many engineers

are prone to ignoring. How we translate legal requirements into technical implementations is not yet well understood, but it's important to keep this question in mind as we move through the rest of this book.

FOOTNOTES

REFERENCES

Richard T. Kouzes, Gordon A. Anderson, Stephen T. Elbert, Ian Gorton, and Deborah K. Gracio. [The Changing Paradigm of Data-Intensive Computing](#). *IEEE Computer*, volume 42, issue 1, January 2009. [doi:10.1109/MC.2009.26](https://doi.org/10.1109/MC.2009.26)

Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. [Local-first software: you own your data, in spite of the cloud](#). At *2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Onward!), October 2019. [doi:10.1145/3359591.3359737](https://doi.org/10.1145/3359591.3359737)

Joe Reis and Matt Housley. [Fundamentals of Data Engineering](#). O'Reilly Media, 2022. ISBN: 9781098108304

Rui Pedro Machado and Helder Russa. [Analytics Engineering with SQL and dbt](#). O'Reilly Media, 2023. ISBN: 9781098142384

Edgar F. Codd, S. B. Codd, and C. T. Salley. [Providing OLAP to User-Analysts: An IT Mandate](#). E. F. Codd Associates, 1993. Archived at perma.cc/RKX8-2GEE

Chinmay Soman and Neha Pawar. [Comparing Three Real-Time OLAP Databases: Apache Pinot, Apache Druid, and ClickHouse](#). startree.ai, April 2023. Archived at perma.cc/8BZP-VWPA

Surajit Chaudhuri and Umeshwar Dayal. [An Overview of Data Warehousing and OLAP Technology](#). *ACM SIGMOD Record*, volume 26, issue 1, pages 65–74, March 1997. [doi:10.1145/248603.248616](#)

Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. [Hybrid Transactional/Analytical Processing: A Survey](#). At *ACM International Conference on Management of Data* (SIGMOD), May 2017. [doi:10.1145/3035918.3054784](#)

Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, and Nikita Shamgunov. [Cloud-Native Transactions and Analytics in SingleStore](#). At *International Conference on Management of Data* (SIGMOD), June 2022. [doi:10.1145/3514221.3526055](#)

Chao Zhang, Guoliang Li, Jintao Zhang, Xinning Zhang, and Jianhua Feng. [HTAP Databases: A Survey](#). *IEEE Transactions on Knowledge and Data Engineering*, April 2024. [doi:10.1109/TKDE.2024.3389693](#)

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. [doi:10.1109/ICDE.2005.1](#)

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, issue 2, pages 1481–1492, August 2009. [doi:10.14778/1687553.1687576](#)

Dan Olteanu. [The Relational Data Borg is Learning](#). *Proceedings of the VLDB Endowment*, volume 13, issue 12, August 2020. [doi:10.14778/3415478.3415572](#)

Matt Bornstein, Martin Casado, and Jennifer Li. [Emerging Architectures for Modern Data Infrastructure: 2020](#). *future.a16z.com*, October 2020. Archived at [perma.cc/LF8W-KDCC](#)

Martin Fowler. [DataLake](#). *martinfowler.com*, February 2015. Archived at [perma.cc/4WKN-CZUK](#)

Bobby Johnson and Joseph Adler. [The Sushi Principle: Raw Data Is Better](#). At *Strata+Hadoop World*, February 2015.

Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. [Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics](#). At *11th Annual Conference on Innovative Data Systems Research* (CIDR), January 2021.

DataKitchen, Inc. [The DataOps Manifesto](#). *dataopsmanifesto.org*, 2017. Archived at [perma.cc/3F5N-FUQ4](#)

Tejas Manohar. [What is Reverse ETL: A Definition & Why It's Taking Off](#). *hightouch.io*, November 2021. Archived at [perma.cc/A7TN-GLYJ](#)

Simon O'Regan. [Designing Data Products](#). *towardsdatascience.com*, August 2018. Archived at [perma.cc/HU67-3RV8](#)

Camille Fournier. [Why is it so hard to decide to buy?](#) *skamille.medium.com*, July 2021. Archived at [perma.cc/6VSG-HQ5X](#)

David Heinemeier Hansson. [Why we're leaving the cloud](#). *world.hey.com*, October 2022. Archived at [perma.cc/82E6-UJ65](#)

Nima Badizadegan. [Use One Big Server](#). *specbranch.com*, August 2022. Archived at [perma.cc/M8NB-95UK](#)

Steve Yegge. [Dear Google Cloud: Your Deprecation Policy is Killing You](#). *steveyegge.medium.com*, August 2020. Archived at [perma.cc/KQP9-SPGU](#)

Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili,

and Xiaofeng Bao. [Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases](#). At *ACM International Conference on Management of Data* (SIGMOD), pages 1041–1052, May 2017. [doi:10.1145/3035918.3056101](#)

Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade.

[Socrates: The New SQL Server in the Cloud](#). At *ACM International Conference on Management of Data* (SIGMOD), pages 1743–1756, June 2019.

[doi:10.1145/3299869.3314047](#)

Midhul Vuppala, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. [Building An Elastic Query Engine on Disaggregated Storage](#). At *17th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), February 2020.

Colin Breck. [Predicting the Future of Distributed Systems](#). blog.colinbreck.com, August 2024. Archived at [perma.cc/K5FC-4XX2](#)

Gwen Shapira. [Compute-Storage Separation Explained](#). thenile.dev, January 2023. Archived at [perma.cc/QCV3-XJNZ](#)

Ravi Murthy and Gurmeet Goindi. [AlloyDB for PostgreSQL under the hood: Intelligent, database-aware storage](#). cloud.google.com, May 2022. Archived at [archive.org](#)

Jack Vanlightly. [The Architecture of Serverless Data Systems](#). jack-vanlightly.com, November 2023. Archived at [perma.cc/UDV4-TNJ5](#)

Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, David A. Patterson.

[Cloud Programming Simplified: A Berkeley View on Serverless Computing](#). arxiv.org, February 2019.

Betsy Beyer, Jennifer Petoff, Chris Jones, and Niall Richard Murphy. [Site Reliability Engineering: How Google Runs Production Systems](#). O'Reilly Media, 2016. ISBN: 9781491929124

Thomas Limoncelli. [The Time I Stole \\$10,000 from Bell Labs](#). ACM Queue, volume 18, issue 5, November 2020. [doi:10.1145/3434571.3434773](#)

Charity Majors. [The Future of Ops Jobs](#). acldguru.com, August 2020. Archived at [perma.cc/GRU2-CZG3](#)

Boris Cherkasky. [\(Over\)Pay As You Go for Your Datastore](#). medium.com, September 2021. Archived at [perma.cc/Q8TV-2AM2](#)

Shlomi Kushchi. [Serverless Doesn't Mean DevOpsLess or NoOps](#). thenewstack.io, February 2023. Archived at [perma.cc/3NJR-AYYU](#)

Erik Bernhardsson. [Storm in the stratosphere: how the cloud will be reshuffled](#). erikbern.com, November 2021. Archived at [perma.cc/SYB2-99P3](#)

Benn Stancil. [The data OS](#). benn.substack.com, September 2021. Archived at [perma.cc/WQ43-FHS6](#)

Maria Korolov. [Data residency laws pushing companies toward residency as a service](#). csoonline.com, January 2022. Archived at [perma.cc/CHE4-XZZ2](#)

Kousik Nath. [These are the numbers every computer engineer should know](#). freeCodeCamp.org, September 2019. Archived at [perma.cc/RW73-36RL](#)

Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. [Serverless Computing: One Step](#)

[Forward, Two Steps Back](#). At *Conference on Innovative Data Systems Research* (CIDR), January 2019.

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.

Cindy Sridharan. [Distributed Systems Observability: A Guide to Building Robust Systems](#). Report, O'Reilly Media, May 2018. Archived at perma.cc/M6JL-XKCM

Charity Majors. [Observability — A 3-Year Retrospective](#). *thenewstack.io*, August 2019. Archived at perma.cc/CG62-TJWL

Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. [Dapper, a Large-Scale Distributed Systems Tracing Infrastructure](#). Google Technical Report dapper-2010-1, April 2010. Archived at perma.cc/K7KU-2TMH

Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. [Data management in microservices: State of the practice, challenges, and research directions](#). *Proceedings of the VLDB Endowment*, volume 14, issue 13, pages 3348–3361, September 2021. [doi:10.14778/3484224.3484232](https://doi.org/10.14778/3484224.3484232)

Jordan Tigani. [Big Data is Dead](#). *motherduck.com*, February 2023. Archived at perma.cc/HT4Q-K77U

Sam Newman. [Building Microservices, second edition](#). O'Reilly Media, 2021. ISBN: 9781492034025

Chris Richardson. [Microservices: Decomposing Applications for Deployability and Scalability](#). *infoq.com*, May 2014. Archived at perma.cc/CKN4-YEQ2

Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, Ricardo

Bianchini. [Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider](#). At *USENIX Annual Technical Conference (ATC)*, July 2020.

Luiz André Barroso, Urs Hözle, and Parthasarathy Ranganathan. [The Datacenter as a Computer: Designing Warehouse-Scale Machines](#), third edition. Morgan & Claypool Synthesis Lectures on Computer Architecture, October 2018.
[doi:10.2200/S00874ED3V01Y201809CAC046](#)

David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. [Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing](#),” at *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2012.
[doi:10.1109/SC.2012.49](#)

Anna Kornfeld Simpson, Adriana Szekeres, Jacob Nelson, and Irene Zhang. [Securing RDMA for High-Performance Datacenter Storage Systems](#). At *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, July 2020.

Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hözle, Stephen Stuart, and Amin Vahdat. [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)*, August 2015. [doi:10.1145/2785956.2787508](#)

Glenn K. Lockwood. [Hadoop’s Uncomfortable Fit in HPC](#).
glenchklockwood.blogspot.co.uk, May 2014. Archived at [perma.cc/S8XX-Y67B](#)

Cathy O’Neil: *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*. Crown Publishing, 2016. ISBN: 9780553418811

Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaran. [Understanding and Benchmarking the Impact of GDPR on Database Systems](#). *Proceedings of the VLDB Endowment*, volume 13, issue 7, pages 1064–1077, March 2020. [doi:10.14778/3384345.3384354](https://doi.org/10.14778/3384345.3384354)

Martin Fowler. [Datensparsamkeit](#). *martinfowler.com*, December 2013. Archived at perma.cc/R9QX-CME6

[Regulation \(EU\) 2016/679 of the European Parliament and of the Council of 27 April 2016 \(General Data Protection Regulation\)](#). *Official Journal of the European Union L 119/1*, May 2016.

Chapter 2. Defining Nonfunctional Requirements

The Internet was done so well that most people think of it as a natural resource like the Pacific Ocean, rather than something that was man-made. When was the last time a technology with a scale like that was so error-free?

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

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. The GitHub repo for this book is <https://github.com/ept/ddia2-feedback>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out on GitHub.

If you are building an application, you will be driven by a list of requirements. At the top of your list is most likely the functionality that the application must offer: what screens and what buttons you need, and what each operation is supposed to do in order to fulfill the purpose of your software. These are your *functional requirements*.

In addition, you probably also have some *nonfunctional requirements*: for example, the app should be fast, reliable, secure, legally compliant, and easy to maintain. These requirements might not be explicitly written down, because they may seem somewhat obvious, but they are just as important as the app's functionality: an app that is unbearably slow or unreliable might as well not exist.

Many nonfunctional requirements, such as security, fall outside the scope of this book. But there are a few nonfunctional requirements that we will consider, and this chapter will help you articulate them for your own systems:

- How to define and measure the *performance* of a system (see [“Describing Performance”](#));
- What it means for a service to be *reliable*—namely, continuing to work correctly, even when things go wrong (see [“Reliability and Fault Tolerance”](#));

- Allowing a system to be *scalable* by having efficient ways of adding computing capacity as the load on the system grows (see “[Scalability](#)”); and
- Making it easier to maintain a system in the long term (see “[Maintainability](#)”).

The terminology introduced in this chapter will also be useful in the following chapters, when we go into the details of how data-intensive systems are implemented. However, abstract definitions can be quite dry; to make the ideas more concrete, we will start this chapter with a case study of how a social networking service might work, which will provide practical examples of performance and scalability.

Case Study: Social Network Home Timelines

Imagine you are given the task of implementing a social network in the style of X (formerly Twitter), in which users can post messages and follow other users. This will be a huge simplification of how such a service actually works [[1](#), [2](#), [3](#)], but it will help illustrate some of the issues that arise in large-scale systems.

Let's assume that users make 500 million posts per day, or 5,700 posts per second on average. Occasionally, the rate can spike as high as 150,000 posts/second [4]. Let's also assume that the average user follows 200 people and has 200 followers (although there is a very wide range: most people have only a handful of followers, and a few celebrities such as Barack Obama have over 100 million followers).

Representing Users, Posts, and Follows

Imagine we keep all of the data in a relational database as shown in [Figure 2-1](#). We have one table for users, one table for posts, and one table for follow relationships.

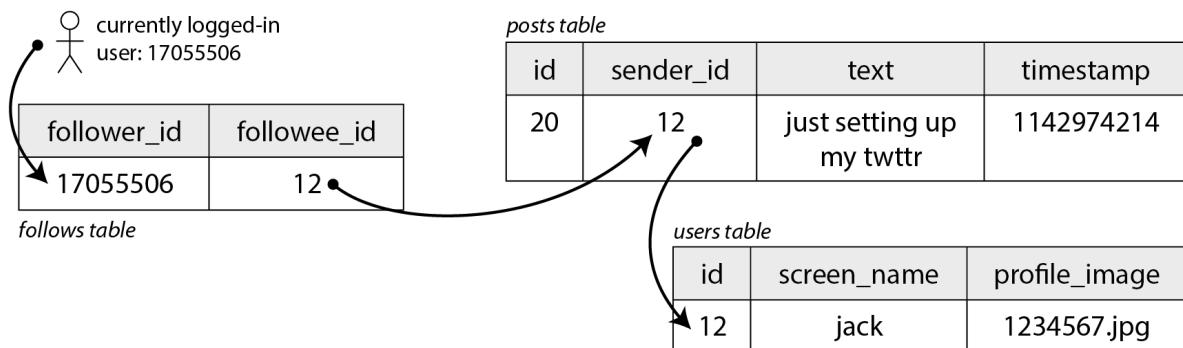


Figure 2-1. Simple relational schema for a social network in which users can follow each other.

Let's say the main read operation that our social network must support is the *home timeline*, which displays recent posts by people you are following (for simplicity we will ignore ads,

suggested posts from people you are not following, and other extensions). We could write the following SQL query to get the home timeline for a particular user:

```
SELECT posts.*, users.* FROM posts
    JOIN follows ON posts.sender_id = follows.follower_id
    JOIN users ON posts.sender_id = users.id
    WHERE follows.follower_id = current_user
    ORDER BY posts.timestamp DESC
    LIMIT 1000
```

To execute this query, the database will use the `follows` table to find everybody who `current_user` is following, look up recent posts by those users, and sort them by timestamp to get the most recent 1,000 posts by any of the followed users.

Posts are supposed to be timely, so let's assume that after somebody makes a post, we want their followers to be able to see it within 5 seconds. One way of doing that would be for the user's client to repeat the query above every 5 seconds while the user is online (this is known as *polling*). If we assume that 10 million users are online and logged in at the same time, that would mean running the query 2 million times per second. Even if you increase the polling interval, this is a lot.

Moreover, the query above is quite expensive: if you are following 200 people, it needs to fetch a list of recent posts by each of those 200 people, and merge those lists. 2 million timeline queries per second then means that the database needs to look up the recent posts from some sender 400 million times per second—a huge number. And that is the average case. Some users follow tens of thousands of accounts; for them, this query is very expensive to execute, and difficult to make fast.

Materializing and Updating Timelines

How can we do better? Firstly, instead of polling, it would be better if the server actively pushed new posts to any followers who are currently online. Secondly, we should precompute the results of the query above so that a user’s request for their home timeline can be served from a cache.

Imagine that for each user we store a data structure containing their home timeline, i.e., the recent posts by people they are following. Every time a user makes a post, we look up all of their followers, and insert that post into the home timeline of each follower—like delivering a message to a mailbox. Now when a user logs in, we can simply give them this home timeline that we precomputed. Moreover, to receive a notification about any new posts on their timeline, the user’s

client simply needs to subscribe to the stream of posts being added to their home timeline.

The downside of this approach is that we now need to do more work every time a user makes a post, because the home timelines are derived data that needs to be updated. The process is illustrated in [Figure 2-2](#). When one initial request results in several downstream requests being carried out, we use the term *fan-out* to describe the factor by which the number of requests increases.

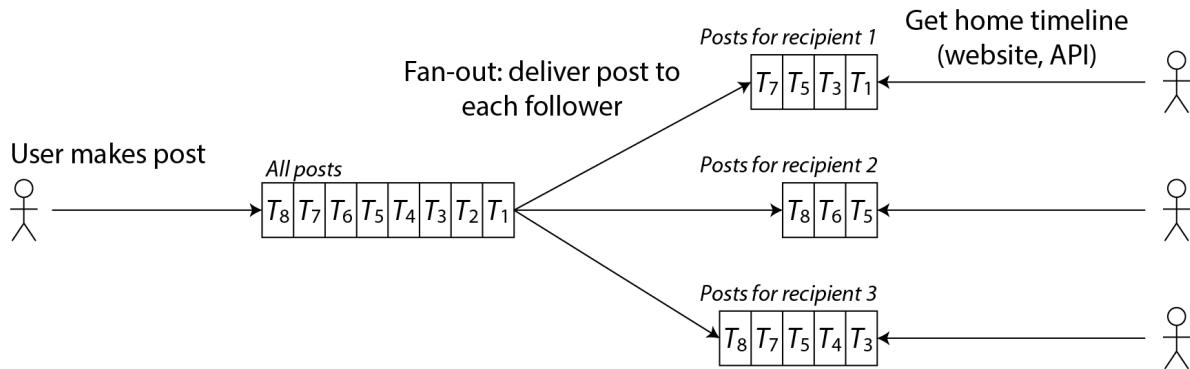


Figure 2-2. Fan-out: delivering new posts to every follower of the user who made the post.

At a rate of 5,700 posts posted per second, if the average post reaches 200 followers (i.e., a fan-out factor of 200), we will need to do just over 1 million home timeline writes per second. This is a lot, but it's still a significant saving compared to the 400

million per-sender post lookups per second that we would otherwise have to do.

If the rate of posts spikes due to some special event, we don't have to do the timeline deliveries immediately—we can enqueue them and accept that it will temporarily take a bit longer for posts to show up in followers' timelines. Even during such load spikes, timelines remain fast to load, since we simply serve them from a cache.

This process of precomputing and updating the results of a query is called *materialization*, and the timeline cache is an example of a *materialized view* (a concept we will discuss further in [Link to Come]). The downside of materialization is that every time a celebrity makes a post, we now have to do a large amount of work to insert that post into the home timelines of each of their millions of followers.

One way of solving this problem is to handle celebrity posts separately from everyone else's posts: we can save ourselves the effort of adding them to millions of timelines by storing the celebrity posts separately and merging them with the materialized timeline when it is read. Despite such optimizations, handling celebrities on a social network can require a lot of infrastructure [5].

Describing Performance

Most discussions of software performance consider two main types of metric:

Response time

The elapsed time from the moment when a user makes a request until they receive the requested answer. The unit of measurement is seconds (or milliseconds, or microseconds).

Throughput

The number of requests per second, or the data volume per second, that the system is processing. For a given allocation of hardware resources, there is a *maximum throughput* that can be handled. The unit of measurement is “somethings per second”.

In the social network case study, “posts per second” and “timeline writes per second” are throughput metrics, whereas the “time it takes to load the home timeline” or the “time until a post is delivered to followers” are response time metrics.

There is often a connection between throughput and response time; an example of such a relationship for an online service is sketched in [Figure 2-3](#). The service has a low response time when request throughput is low, but response time increases as load increases. This is because of *queueing*: when a request arrives on a highly loaded system, it's likely that the CPU is already in the process of handling an earlier request, and therefore the incoming request needs to wait until the earlier request has been completed. As throughput approaches the maximum that the hardware can handle, queueing delays increase sharply.

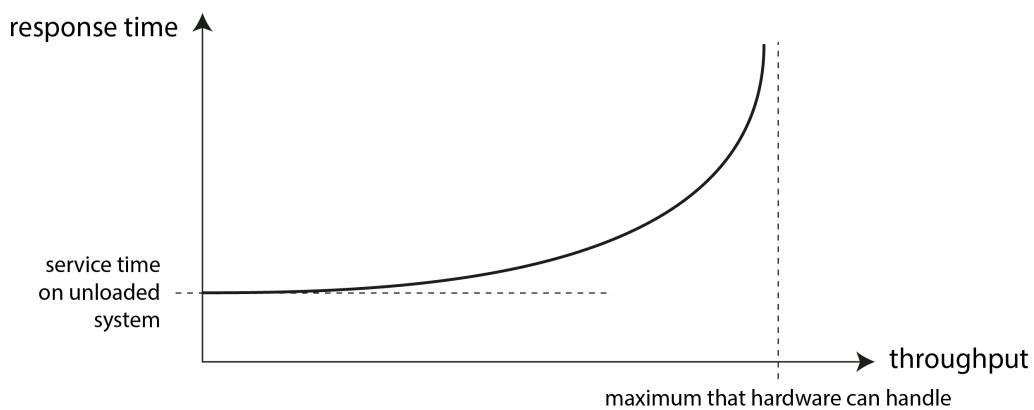


Figure 2-3. As the throughput of a service approaches its capacity, the response time increases dramatically due to queueing.

WHEN AN OVERLOADED SYSTEM WON'T RECOVER

If a system is close to overload, with throughput pushed close to the limit, it can sometimes enter a vicious cycle where it becomes less efficient and hence even more overloaded. For example, if there is a long queue of requests waiting to be handled, response times may increase so much that clients time out and resend their request. This causes the rate of requests to increase even further, making the problem worse—a *retry storm*. Even when the load is reduced again, such a system may remain in an overloaded state until it is rebooted or otherwise reset. This phenomenon is called a *metastable failure*, and it can cause serious outages in production systems [6, 7].

To avoid retries overloading a service, you can increase and randomize the time between successive retries on the client side (*exponential backoff* [8, 9]), and temporarily stop sending requests to a service that has returned errors or timed out recently (using a *circuit breaker* [10, 11] or *token bucket* algorithm [12]). The server can also detect when it is approaching overload and start proactively rejecting requests (*load shedding* [13]), and send back responses asking clients to slow down (*backpressure* [1, 14]). The choice of queueing and load-balancing algorithms can also make a difference [15].

In terms of performance metrics, the response time is usually what users care about the most, whereas the throughput determines the required computing resources (e.g., how many servers you need), and hence the cost of serving a particular workload. If throughput is likely to increase beyond what the current hardware can handle, the capacity needs to be expanded; a system is said to be *scalable* if its maximum throughput can be significantly increased by adding computing resources.

In this section we will focus primarily on response times, and we will return to throughput and scalability in “[Scalability](#)”.

Latency and Response Time

“Latency” and “response time” are sometimes used interchangeably, but in this book we will use the terms in a specific way (illustrated in [Figure 2-4](#)):

- The *response time* is what the client sees; it includes all delays incurred anywhere in the system.
- The *service time* is the duration for which the service is actively processing the user request.
- *Queueing delays* can occur at several points in the flow: for example, after a request is received, it might need to wait

until a CPU is available before it can be processed; a response packet might need to be buffered before it is sent over the network if other tasks on the same machine are sending a lot of data via the outbound network interface.

- *Latency* is a catch-all term for time during which a request is not being actively processed, i.e., during which it is *latent*. In particular, *network latency* or *network delay* refers to the time that request and response spend traveling through the network.

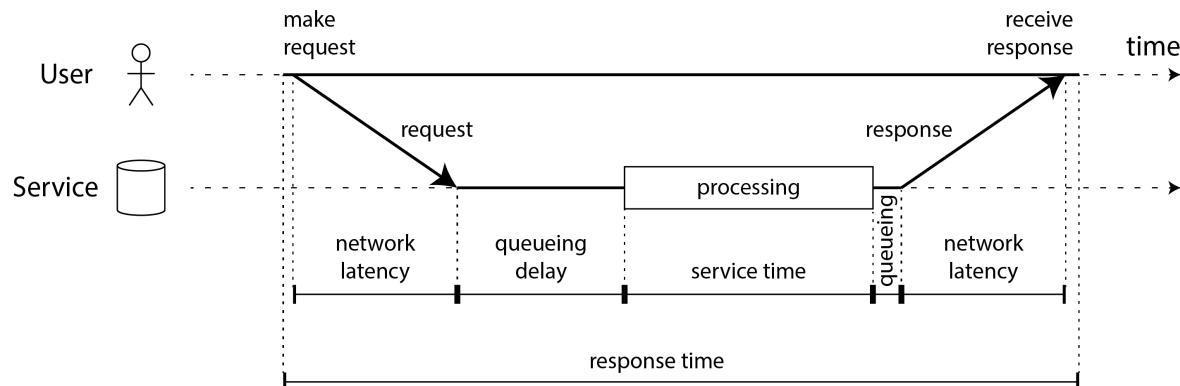


Figure 2-4. Response time, service time, network latency, and queueing delay.

In [Figure 2-4](#), time flows from left to right, each communicating node is shown as a horizontal line, and a request or response message is shown as a thick diagonal arrow from one node to another. You will encounter this style of diagram frequently over the course of this book.

The response time can vary significantly from one request to the next, even if you keep making the same request over and over again. Many factors can add random delays: for example, 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 [16], or many other causes. We will discuss this topic in more detail in [Link to Come].

Queueing delays often account for a large part of the variability in response times. 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 known as *head-of-line blocking*. Even if those subsequent requests have fast service times, the client will see a slow overall response time due to the time waiting for the prior request to complete. The queueing delay is not part of the service time, and for this reason it is important to measure response times on the client side.

Average, Median, and Percentiles

Because the response time varies from one request to the next, we need to think of it not as a single number, but as a

distribution of values that you can measure. In [Figure 2-5](#), 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.

Variation in network delay is also known as *jitter*.

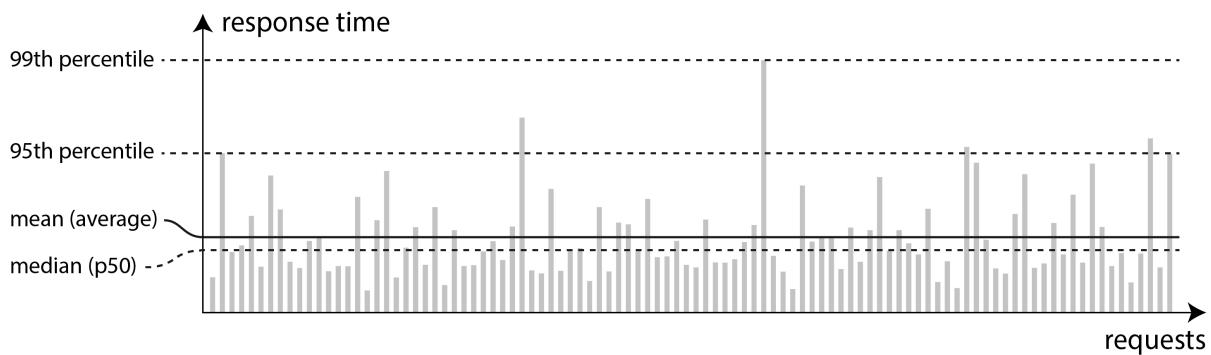


Figure 2-5. Illustrating mean and percentiles: response times for a sample of 100 requests to a service.

It's common to report the *average* response time of a service (technically, the *arithmetic mean*: that is, sum all the response times, and divide by the number of requests). The mean response time is useful for estimating throughput limits [\[17\]](#). However, the mean is not a very good metric if you want to know your “typical” response time, because it doesn’t tell you how many users actually experienced that delay.

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

response time is 200 ms, that means half your requests return in less than 200 ms, and half your requests take longer than that. This makes the median a good metric if you want to know how long users typically have to wait. The median is also known as the *50th percentile*, and sometimes abbreviated as *p50*.

In order to figure out how bad your outliers are, you can look at higher percentiles: the *95th*, *99th*, and *99.9th* percentiles are common (abbreviated *p95*, *p99*, and *p999*). They are the response time thresholds at which 95%, 99%, or 99.9% of requests are faster than that particular threshold. For example, if the 95th percentile response time is 1.5 seconds, that means 95 out of 100 requests take less than 1.5 seconds, and 5 out of 100 requests take 1.5 seconds or more. This is illustrated in [Figure 2-5](#).

High percentiles of response times, also known as *tail latencies*, are important because they directly affect users' experience of the service. For example, Amazon describes response time requirements for internal services in terms of the 99.9th percentile, even though it only affects 1 in 1,000 requests. This is because the customers with the slowest requests are often those who have the most data on their accounts because they have made many purchases—that is, they're the most valuable

customers [18]. It's important to keep those customers happy by ensuring the website is fast for them.

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

THE USER IMPACT OF RESPONSE TIMES

It seems intuitively obvious that a fast service is better for users than a slow service [19]. However, it is surprisingly difficult to get hold of reliable data to quantify the effect that latency has on user behavior.

Some often-cited statistics are unreliable. In 2006 Google reported that a slowdown in search results from 400 ms to 900 ms was associated with a 20% drop in traffic and revenue [20]. However, another Google study from 2009 reported that a 400 ms increase in latency resulted in only 0.6% fewer searches per day [21], and in the same year Bing found that a two-second increase in load time reduced ad revenue by 4.3% [22]. Newer data from these companies appears not to be publicly available.

A more recent Akamai study [23] claims that a 100 ms increase in response time reduced the conversion rate of e-commerce sites by up to 7%; however, on closer inspection, the same study reveals that very *fast* page load times are also correlated with lower conversion rates! This seemingly paradoxical result is explained by the fact that the pages that load fastest are often those that have no useful content (e.g., 404 error pages). However, since the study makes no effort to separate the effects of page content from the effects of load time, its results are probably not meaningful.

A study by Yahoo [24] compares click-through rates on fast-loading versus slow-loading search results, controlling for quality of search results. It finds 20–30% more clicks on fast searches when the difference between fast and slow responses is 1.25 seconds or more.

Use of Response Time Metrics

High percentiles are 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 2-6](#). Even if only a small percentage of backend calls are slow, the chance of getting a slow call increases if an end-user request requires multiple backend calls, and so a higher proportion of end-user requests end up being slow (an effect known as *tail latency amplification* [25]).

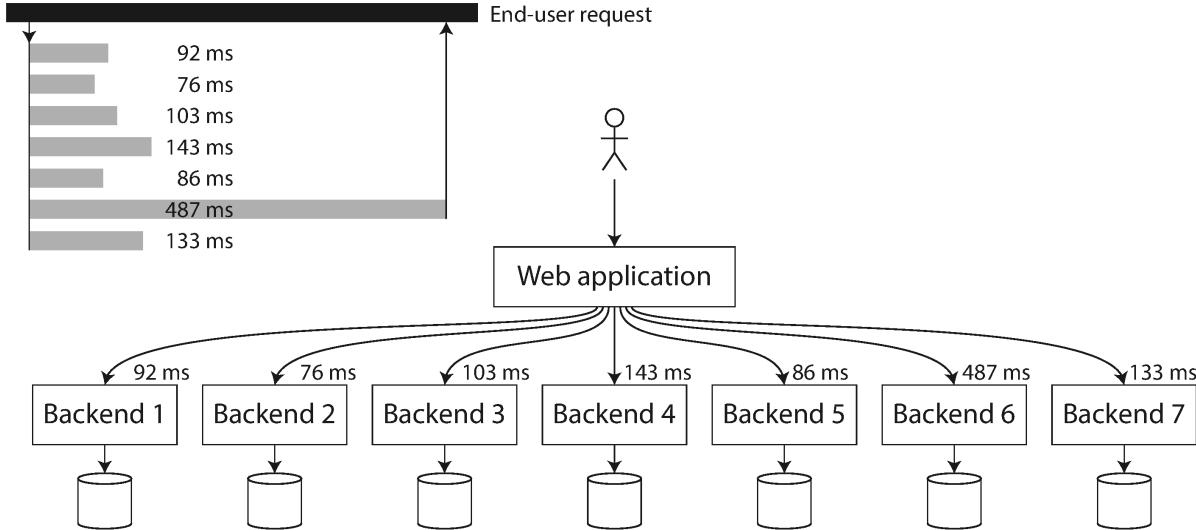


Figure 2-6. When several backend calls are needed to serve a request, it takes just a single slow backend request to slow down the entire end-user request.

Percentiles are often used in *service level objectives* (SLOs) and *service level agreements* (SLAs) as ways of defining the expected performance and availability of a service [26]. For example, an SLO may set a target for a service to have a median response time of less than 200 ms and a 99th percentile under 1 s, and a target that at least 99.9% of valid requests result in non-error responses. An SLA is a contract that specifies what happens if the SLO is not met (for example, customers may be entitled to a refund). That is the basic idea, at least; in practice, defining good availability metrics for SLOs and SLAs is not straightforward [27, 28].

COMPUTING PERCENTILES

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

The simplest implementation is to keep a list of response times for all requests within the time window and to sort that list every minute. If that is too inefficient for you, there are algorithms that can calculate a good approximation of percentiles at minimal CPU and memory cost. Open source percentile estimation libraries include HdrHistogram, t-digest [29, 30], OpenHistogram [31], and DDSketch [32].

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 [33].

Reliability and Fault Tolerance

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

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

If all those things together mean “working correctly,” then we can understand *reliability* as meaning, roughly, “continuing to work correctly, even when things go wrong.” To be more precise about things going wrong, we will distinguish between *faults* and *failures* [34, 35, 36]:

Fault

A fault is when a particular *part* of a system stops working correctly: for example, if a single hard drive malfunctions, or a single machine crashes, or an external service (that the system depends on) has an outage.

Failure

A failure is when the system *as a whole* stops providing the required service to the user; in other words, when it does not meet the service level objective (SLO).

The distinction between fault and failure can be confusing because they are the same thing, just at different levels. For example, if a hard drive stops working, we say that the hard drive has failed: if the system consists only of that one hard drive, it has stopped providing the required service. However, if the system you're talking about contains many hard drives, then the failure of a single hard drive is only a fault from the point of view of the bigger system, and the bigger system might be able to tolerate that fault by having a copy of the data on another hard drive.

Fault Tolerance

We call a system *fault-tolerant* if it continues providing the required service to the user in spite of certain faults occurring. If a system cannot tolerate a certain part becoming faulty, we call that part a *single point of failure* (SPOF), because a fault in that part escalates to cause the failure of the whole system.

For example, in the social network case study, a fault that might happen is that during the fan-out process, a machine involved in updating the materialized timelines crashes or becomes unavailable. To make this process fault-tolerant, we would need to ensure that another machine can take over this task without missing any posts that should have been delivered, and without duplicating any posts. (This idea is known as *exactly-once semantics*, and we will examine it in detail in [Link to Come].)

Fault tolerance is always limited to a certain number of certain types of faults. For example, a system might be able to tolerate a maximum of two hard drives failing at the same time, or a maximum of one out of three nodes crashing. It would not make sense to tolerate any number of faults: if all nodes crash, there is nothing that can be done. 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.

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. This is called *fault injection*. Many critical bugs are actually due to poor error handling [37]; 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. *Chaos engineering* is a discipline that aims to improve confidence in fault-tolerance mechanisms through experiments such as deliberately injecting faults [38].

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

Hardware and Software Faults

When we think of causes of system failure, hardware faults quickly come to mind:

- Approximately 2–5% of magnetic hard drives fail per year [39, 40]; in a storage cluster with 10,000 disks, we should therefore expect on average one disk failure per day. Recent data suggests that disks are getting more reliable, but failure rates remain significant [41].

- Approximately 0.5–1% of solid state drives (SSDs) fail per year [42]. Small numbers of bit errors are corrected automatically [43], but uncorrectable errors occur approximately once per year per drive, even in drives that are fairly new (i.e., that have experienced little wear); this error rate is higher than that of magnetic hard drives [44, 45].
- Other hardware components such as power supplies, RAID controllers, and memory modules also fail, although less frequently than hard drives [46, 47].
- Approximately one in 1,000 machines has a CPU core that occasionally computes the wrong result, likely due to manufacturing defects [48, 49, 50]. In some cases, an erroneous computation leads to a crash, but in other cases it leads to a program simply returning the wrong result.
- Data in RAM can also be corrupted, either due to random events such as cosmic rays, or due to permanent physical defects. Even when memory with error-correcting codes (ECC) is used, more than 1% of machines encounter an uncorrectable error in a given year, which typically leads to a crash of the machine and the affected memory module needing to be replaced [51]. Moreover, certain pathological memory access patterns can flip bits with high probability [52].

- An entire datacenter might become unavailable (for example, due to power outage or network misconfiguration) or even be permanently destroyed (for example by fire or flood). Although such large-scale failures are rare, their impact can be catastrophic if a service cannot tolerate the loss of a datacenter [53].

These events are rare enough that you often don't need to worry about them when working on a small system, as long as you can easily replace hardware that becomes faulty. However, in a large-scale system, hardware faults happen often enough that they become part of the normal system operation.

Tolerating hardware faults through redundancy

Our first response to unreliable hardware 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 (spreading data across multiple disks in the same machine so that a failed disk does not cause data loss), servers may have dual power supplies and hot-swappable CPUs, and datacenters may have batteries and diesel generators for backup power. Such redundancy can often keep a machine running uninterrupted for years.

Redundancy is most effective when component faults are independent, that is, the occurrence of one fault does not change how likely it is that another fault will occur. However, experience has shown that there are often significant correlations between component failures [40, 54, 55]; unavailability of an entire server rack or an entire datacenter still happens more often than we would like.

Hardware redundancy increases the uptime of a single machine; however, as discussed in “[Distributed versus Single-Node Systems](#)”, there are advantages to using a distributed system, such as being able to tolerate a complete outage of one datacenter. For this reason, cloud systems tend to focus less on the reliability of individual machines, and instead aim to make services highly available by tolerating faulty nodes at the software level. Cloud providers use *availability zones* to identify which resources are physically co-located; resources in the same place are more likely to fail at the same time than geographically separated resources.

The fault-tolerance techniques we discuss in this book are designed to tolerate the loss of entire machines, racks, or availability zones. They generally work by allowing a machine in one datacenter to take over when a machine in another datacenter fails or becomes unreachable. We will discuss such

techniques for fault tolerance in [Chapter 6](#), [Link to Come], and at various other points in this book.

Systems that can tolerate the loss of entire machines 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 multi-node fault-tolerant system can be patched by restarting one node at a time, without affecting the service for users. This is called a *rolling upgrade*, and we will discuss it further in [Chapter 5](#).

Software faults

Although hardware failures can be weakly correlated, they are still mostly independent: for example, if one disk fails, it's likely that other disks in the same machine will be fine for another while. On the other hand, software faults are often very highly correlated, because it is common for many nodes to run the same software and thus have the same bugs [\[56, 57\]](#). Such faults are harder to anticipate, and they tend to cause many more system failures than uncorrelated hardware faults [\[46\]](#). For example:

- A software bug that causes every node to fail at the same time in particular circumstances. For example, on June 30, 2012, a leap second caused many Java applications to hang simultaneously due to a bug in the Linux kernel, bringing down many Internet services [58]. Due to a firmware bug, all SSDs of certain models suddenly fail after precisely 32,768 hours of operation (less than 4 years), rendering the data on them unrecoverable [59].
- A runaway process that uses up some shared, limited resource, such as CPU time, memory, disk space, network bandwidth, or threads [60]. For example, a process that consumes too much memory while processing a large request may be killed by the operating system. A bug in a client library could cause a much higher request volume than anticipated [61].
- A service that the system depends on slows down, becomes unresponsive, or starts returning corrupted responses.
- An interaction between different systems results in emergent behavior that does not occur when each system was tested in isolation [62].
- Cascading failures, where a problem in one component causes another component to become overloaded and slow down, which in turn brings down another component [63, 64].

The bugs that cause these kinds of software faults often lie dormant for a long time until they are triggered by an unusual set of circumstances. In those circumstances, it is revealed that the software is making some kind of assumption about its environment—and while that assumption is usually true, it eventually stops being true for some reason [65, 66].

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; avoiding feedback loops such as retry storms (see “[When an overloaded system won’t recover](#)”); measuring, monitoring, and analyzing system behavior in production.

Humans and Reliability

Humans design and build software systems, and the operators who keep the systems running are also human. Unlike machines, humans don’t just follow rules; their strength is being creative and adaptive in getting their job done. However, this characteristic also leads to unpredictability, and sometimes mistakes that can lead to failures, despite best intentions. For example, one study of large internet services found that configuration changes by operators were the leading cause of

outages, whereas hardware faults (servers or network) played a role in only 10–25% of outages [67].

It is tempting to label such problems as “human error” and to wish that they could be solved by better controlling human behavior through tighter procedures and compliance with rules. However, blaming people for mistakes is counterproductive. What we call “human error” is not really the cause of an incident, but rather a symptom of a problem with the sociotechnical system in which people are trying their best to do their jobs [68]. Often complex systems have emergent behavior, in which unexpected interactions between components may also lead to failures [69].

Various technical measures can help minimize the impact of human mistakes, including thorough testing (both hand-written tests and *property testing* on lots of random inputs) [37], rollback mechanisms for quickly reverting configuration changes, gradual roll-outs of new code, detailed and clear monitoring, observability tools for diagnosing production issues (see [“Problems with Distributed Systems”](#)), and well-designed interfaces that encourage “the right thing” and discourage “the wrong thing”.

However, these things require an investment of time and money, and in the pragmatic reality of everyday business, organizations often prioritize revenue-generating activities over measures that increase their resilience against mistakes. If there is a choice between more features and more testing, many organizations understandably choose features. Given this choice, when a preventable mistake inevitably occurs, it does not make sense to blame the person who made the mistake—the problem is the organization’s priorities.

Increasingly, organizations are adopting a culture of *blameless postmortems*: after an incident, the people involved are encouraged to share full details about what happened, without fear of punishment, since this allows others in the organization to learn how to prevent similar problems in the future [70].

This process may uncover a need to change business priorities, a need to invest in areas that have been neglected, a need to change the incentives for the people involved, or some other systemic issue that needs to be brought to the management’s attention.

As a general principle, when investigating an incident, you should be suspicious of simplistic answers. “Bob should have been more careful when deploying that change” is not productive, but neither is “We must rewrite the backend in

Haskell.” Instead, management should take the opportunity to learn the details of how the sociotechnical system works from the point of view of the people who work with it every day, and take steps to improve it based on this feedback [68].

HOW IMPORTANT IS RELIABILITY?

Reliability is not just for nuclear power stations and air traffic control—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 damage to reputation.

In many applications, a temporary outage of a few minutes or even a few hours is tolerable [71], but permanent data loss or corruption would be catastrophic. Consider a parent who stores all their pictures and videos of their children in your photo application [72]. How would they feel if that database was suddenly corrupted? Would they know how to restore it from a backup?

As another example of how unreliable software can harm people, consider the Post Office Horizon scandal. Between 1999 and 2019, hundreds of people managing Post Office branches in Britain were convicted of theft or fraud because the accounting software showed a shortfall in their accounts. Eventually it became clear that many of these shortfalls were due to bugs in the software, and many convictions have since been overturned [73]. What led to this, probably the largest miscarriage of justice in British history, is the fact that English law assumes that

computers operate correctly (and hence, evidence produced by computers is reliable) unless there is evidence to the contrary [74]. Software engineers may laugh at the idea that software could ever be bug-free, but this is little solace to the people who were wrongfully imprisoned, declared bankrupt, or even committed suicide as a result of a wrongful conviction due to an unreliable computer system.

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)—but we should be very conscious of when we are cutting corners and keep in mind the potential consequences.

Scalability

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

Scalability is the term we use to describe a system's ability to cope with increased load. Sometimes, when discussing scalability, people make comments along the lines of, “You’re not Google or Amazon. Stop worrying about scale and just use a relational database.” Whether this maxim applies to you depends on the type of application you are building.

If you are building a new product that currently only has a small number of users, perhaps at a startup, the overriding engineering goal is usually to keep the system as simple and flexible as possible, so that you can easily modify and adapt the features of your product as you learn more about customers' needs [75]. In such an environment, it is counterproductive to worry about hypothetical scale that might be needed in the future: in the best case, investments in scalability are wasted effort and premature optimization; in the worst case, they lock you into an inflexible design and make it harder to evolve your application.

The reason is that scalability is not a one-dimensional label: it is meaningless to say “X is scalable” or “Y doesn’t scale.” Rather, discussing scalability means considering questions like:

- “If the system grows in a particular way, what are our options for coping with the growth?”

- “How can we add computing resources to handle the additional load?”
- “Based on current growth projections, when will we hit the limits of our current architecture?”

If you succeed in making your application popular, and therefore handling a growing amount of load, you will learn where your performance bottlenecks lie, and therefore you will know along which dimensions you need to scale. At that point it's time to start worrying about techniques for scalability.

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?). Often this will be a measure of throughput: for example, the number of requests per second to a service, how many gigabytes of new data arrive per day, or the number of shopping cart checkouts per hour. Sometimes you care about the peak of some variable quantity, such as the number of simultaneously online users in [“Case Study: Social Network Home Timelines”](#).

Often there are other statistical characteristics of the load that also affect the access patterns and hence the scalability

requirements. For example, you may need to know the ratio of reads to writes in a database, the hit rate on a cache, or the number of data items per user (for example, the number of followers in the social network case study). Perhaps the average case is what matters for you, or perhaps your bottleneck is dominated by a small number of extreme cases. It all depends on the details of your particular application.

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

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

Usually our goal is to keep the performance of the system within the requirements of the SLA (see [“Use of Response Time Metrics”](#)) while also minimizing the cost of running the system. The greater the required computing resources, the higher the cost. It might be that some types of hardware are more cost-

effective than others, and these factors may change over time as new types of hardware become available.

If you can double the resources in order to handle twice the load, while keeping performance the same, we say that you have *linear scalability*, and this is considered a good thing. Occasionally it is possible to handle twice the load with less than double the resources, due to economies of scale or a better distribution of peak load [76, 77]. Much more likely is that the cost grows faster than linearly, and there may be many reasons for the inefficiency. For example, if you have a lot of data, then processing a single write request may involve more work than if you have a small amount of data, even if the size of the request is the same.

Shared-Memory, Shared-Disk, and Shared-Nothing Architecture

The simplest way of increasing the hardware resources of a service is to move it to a more powerful machine. Individual CPU cores are no longer getting significantly faster, but you can buy a machine (or rent a cloud instance) with more CPU cores, more RAM, and more disk space. This approach is called *vertical scaling* or *scaling up*.

You can get parallelism on a single machine by using multiple processes or threads. All the threads belonging to the same process can access the same RAM, and hence this approach is also called a *shared-memory architecture*. The problem with a shared-memory approach is that the cost grows faster than linearly: a high-end machine with twice the hardware resources typically costs significantly more than twice as much. And due to bottlenecks, a machine twice the size can often handle less than twice the load.

Another approach is the *shared-disk architecture*, which uses several machines with independent CPUs and RAM, but which stores data on an array of disks that is shared between the machines, which are connected via a fast network: *Network-Attached Storage* (NAS) or *Storage Area Network* (SAN). This architecture has traditionally been used for on-premises data warehousing workloads, but contention and the overhead of locking limit the scalability of the shared-disk approach [78].

By contrast, the *shared-nothing architecture* [79] (also called *horizontal scaling* or *scaling out*) has gained a lot of popularity. In this approach, we use a distributed system with multiple nodes, each of which has its own CPUs, RAM, and disks. Any coordination between nodes is done at the software level, via a conventional network.

The advantages of shared-nothing are that it has the potential to scale linearly, it can use whatever hardware offers the best price/performance ratio (especially in the cloud), it can more easily adjust its hardware resources as load increases or decreases, and it can achieve greater fault tolerance by distributing the system across multiple data centers and regions. The downsides are that it requires explicit sharding (see [Chapter 7](#)), and it incurs all the complexity of distributed systems ([Link to Come]).

Some cloud-native database systems use separate services for storage and transaction execution (see [“Separation of storage and compute”](#)), with multiple compute nodes sharing access to the same storage service. This model has some similarity to a shared-disk architecture, but it avoids the scalability problems of older systems: instead of providing a filesystem (NAS) or block device (SAN) abstraction, the storage service offers a specialized API that is designed for the specific needs of the database [\[80\]](#).

Principles for Scalability

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

Moreover, an architecture that is appropriate for one level of load is unlikely to cope with 10 times that load. If you are working on a fast-growing service, it is therefore likely that you will need to rethink your architecture on every order of magnitude load increase. As the needs of the application are likely to evolve, it is usually not worth planning future scaling needs more than one order of magnitude in advance.

A good general principle for scalability is to break a system down into smaller components that can operate largely independently from each other. This is the underlying principle behind microservices (see [“Microservices and Serverless”](#)), sharding ([Chapter 7](#)), stream processing ([Link to Come]), and shared-nothing architectures. However, the challenge is in knowing where to draw the line between things that should be together, and things that should be apart. Design guidelines for microservices can be found in other books [\[81\]](#), and we discuss sharding of shared-nothing systems in [Chapter 7](#).

Another good principle is not to make things more complicated than necessary. If a single-machine database will do the job, it's probably preferable to a complicated distributed setup. Auto-scaling systems (which automatically add or remove resources in response to demand) are cool, but if your load is fairly predictable, a manually scaled system may have fewer operational surprises (see "[Operations: Automatic or Manual Rebalancing](#)"). A system with five services is simpler than one with fifty. Good architectures usually involve a pragmatic mixture of approaches.

Maintainability

Software does not wear out or suffer material fatigue, so it does not break in the same ways as mechanical objects do. But the requirements for an application frequently change, the environment that the software runs in changes (such as its dependencies and the underlying platform), and it has bugs that need fixing.

It is widely recognized 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 [[82](#), [83](#)].

However, maintenance is also difficult. If a system has been successfully running for a long time, it may well use outdated technologies that not many engineers understand today (such as mainframes and COBOL code); institutional knowledge of how and why a system was designed in a certain way may have been lost as people have left the organization; it might be necessary to fix other people's mistakes. Moreover, the computer system is often intertwined with the human organization that it supports, which means that maintenance of such *legacy* systems is as much a people problem as a technical one [[84](#)].

Every system we create today will one day become a legacy system if it is valuable enough to survive for a long time. In order to minimize the pain for future generations who need to maintain our software, we should design it with maintenance concerns in mind. Although we cannot always predict which decisions might create maintenance headaches in the future, in this book we will pay attention to several principles that are widely applicable:

Operability

Make it easy for the organization to keep the system running smoothly.

Simplicity

Make it easy for new engineers to understand the system, by implementing it using well-understood, consistent patterns and structures, and avoiding unnecessary complexity.

Evolvability

Make it easy for engineers to make changes to the system in the future, adapting it and extending it for unanticipated use cases as requirements change.

Operability: Making Life Easy for Operations

We previously discussed the role of operations in “[Operations in the Cloud Era](#)”, and we saw that human processes are at least as important for reliable operations as software tools. In fact, 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” [57].

In large-scale systems consisting of many thousands of machines, manual maintenance would be unreasonably expensive, and automation is essential. However, automation can be a two-edged sword: there will always be edge cases (such as rare failure scenarios) that require manual intervention from the operations team. Since the cases that cannot be handled automatically are the most complex issues, greater automation requires a *more* skilled operations team that can resolve those issues [85].

Moreover, if an automated system goes wrong, it is often harder to troubleshoot than a system that relies on an operator to perform some actions manually. For that reason, it is not the case that more automation is always better for operability. However, some amount of automation is important, and the sweet spot will depend on the specifics of your particular application and organization.

Good operability means making routine tasks easy, allowing the operations team to focus their efforts on high-value activities. Data systems can do various things to make routine tasks easy, including [86]:

- Allowing monitoring tools to check the system's key metrics, and supporting observability tools (see “[Problems with](#)

[Distributed Systems](#)”) to give insights into the system’s runtime behavior. A variety of commercial and open source tools can help here [87].

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

Simplicity: Managing Complexity

Small software projects can have delightfully simple and expressive code, but as projects get larger, they often become very complex and difficult to understand. This complexity slows down everyone who needs to work on the system, further increasing the cost of maintenance. A software project mired in complexity is sometimes described as a *big ball of mud* [88].

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 [66].

Conversely, reducing complexity greatly improves the maintainability of software, and thus simplicity should be a key goal for the systems we build.

Simple systems are easier to understand, and therefore we should try to solve a given problem in the simplest way possible. Unfortunately, this is easier said than done. Whether something is simple or not is often a subjective matter of taste, as there is no objective standard of simplicity [89]. For example, one system may hide a complex implementation behind a simple interface, whereas another may have a simple implementation that exposes more internal detail to its users—which one is simpler?

One attempt at reasoning about complexity has been to break it down into two categories, *essential* and *accidental* complexity [90]. The idea is that essential complexity is inherent in the problem domain of the application, while accidental complexity arises only because of limitations of our tooling. Unfortunately,

this distinction is also flawed, because boundaries between the essential and the accidental shift as our tooling evolves [91].

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

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

Abstractions for application code, which aim to reduce its complexity, can be created using methodologies such as *design patterns* [92] and *domain-driven design* (DDD) [93]. This book is

not about such application-specific abstractions, but rather about general-purpose abstractions on top of which you can build your applications, such as database transactions, indexes, and event logs. If you want to use techniques such as DDD, you can implement them on top of the foundations described in this book.

Evolvability: Making Change Easy

It's extremely unlikely that your system's requirements will remain unchanged forever. They are much more likely to be in constant flux: you learn new facts, previously unanticipated use cases emerge, business priorities change, users request new features, new platforms replace old platforms, legal or regulatory requirements change, growth of the system forces architectural changes, etc.

In terms of organizational processes, *Agile* working patterns provide a framework for adapting to change. The Agile community has also developed technical tools and processes that are helpful when developing software in a frequently changing environment, such as test-driven development (TDD) and refactoring. In this book, we search for ways of increasing agility at the level of a system consisting of several different applications or services with different characteristics.

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: loosely-coupled, simple systems are usually easier to modify than tightly-coupled, complex ones. Since this is such an important idea, we will use a different word to refer to agility on a data system level: *evolvability* [94].

One major factor that makes change difficult in large systems is when some action is irreversible, and therefore that action needs to be taken very carefully [95]. For example, say you are migrating from one database to another: if you cannot switch back to the old system in case of problems with the new one, the stakes are much higher than if you can easily go back. Minimizing irreversibility improves flexibility.

Summary

In this chapter we examined several examples of nonfunctional requirements: performance, reliability, scalability, and maintainability. Through these topics we have also encountered principles and terminology that we will need throughout the rest of the book. We started with a case study of how one might implement home timelines in a social network, which illustrated some of the challenges that arise at scale.

We discussed how to measure performance (e.g., using response time percentiles), the load on a system (e.g., using throughput metrics), and how they are used in SLAs. Scalability is a closely related concept: that is, ensuring performance stays the same when the load grows. We saw some general principles for scalability, such as breaking a task down into smaller parts that can operate independently, and we will dive into deep technical detail on scalability techniques in the following chapters.

To achieve reliability, you can use fault tolerance techniques, which allow a system to continue providing its service even if some component (e.g., a disk, a machine, or another service) is faulty. We saw examples of hardware faults that can occur, and distinguished them from software faults, which can be harder to deal with because they are often strongly correlated. Another aspect of achieving reliability is to build resilience against humans making mistakes, and we saw blameless postmortems as a technique for learning from incidents.

Finally, we examined several facets of maintainability, including supporting the work of operations teams, managing complexity, and making it easy to evolve an application's functionality over time. There are no easy answers on how to achieve these things, but one thing that can help is to build

applications using well-understood building blocks that provide useful abstractions. The rest of this book will cover a selection of building blocks that have proved to be valuable in practice.

FOOTNOTES

REFERENCES

Mike Cvet. [How We Learned to Stop Worrying and Love Fan-In at Twitter](#). At *QCon San Francisco*, December 2016.

Raffi Krikorian. [Timelines at Scale](#). At *QCon San Francisco*, November 2012. Archived at perma.cc/V9G5-KLYK

Twitter. [Twitter's Recommendation Algorithm](#). *blog.twitter.com*, March 2023. Archived at perma.cc/L5GT-229T

Raffi Krikorian. [New Tweets per second record, and how!](#) *blog.twitter.com*, August 2013. Archived at perma.cc/6JZN-XJYN

Samuel Axon. [3% of Twitter's Servers Dedicated to Justin Bieber](#). *mashable.com*, September 2010. Archived at perma.cc/F35N-CGVX

Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. [Metastable Failures in Distributed Systems](#). At *Workshop on Hot Topics in Operating Systems* (HotOS), May 2021. [doi:10.1145/3458336.3465286](https://doi.org/10.1145/3458336.3465286)

Marc Brooker. [Metastability and Distributed Systems](#). *brooker.co.za*, May 2021. Archived at perma.cc/7FGJ-7XRK

Marc Brooker. [Exponential Backoff And Jitter](#). *aws.amazon.com*, March 2015. Archived at [perma.cc/R6MS-AZKH](#)

Marc Brooker. [What is Backoff For?](#) *brooker.co.za*, August 2022. Archived at [perma.cc/PW9N-55Q5](#)

Michael T. Nygard. [Release It!](#), 2nd Edition. Pragmatic Bookshelf, January 2018. ISBN: 9781680502398

Frank Chen. [Slowing Down to Speed Up – Circuit Breakers for Slack's CI/CD](#). *slack.engineering*, August 2022. Archived at [perma.cc/5FGS-ZPH3](#)

Marc Brooker. [Fixing retries with token buckets and circuit breakers](#). *brooker.co.za*, February 2022. Archived at [perma.cc/MD6N-GW26](#)

David Yanacek. [Using load shedding to avoid overload](#). Amazon Builders' Library, *aws.amazon.com*. Archived at [perma.cc/9SAW-68MP](#)

Matthew Sackman. [Pushing Back](#). *wellquite.org*, May 2016. Archived at [perma.cc/3KCZ-RUFY](#)

Dmitry Kopytkov and Patrick Lee. [Meet Bandaid, the Dropbox service proxy](#). *dropbox.tech*, March 2018. Archived at [perma.cc/KUU6-YG4S](#)

Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. [Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems](#). At *16th USENIX Conference on File and Storage Technologies*, February 2018.

Marc Brooker. [Is the Mean Really Useless?](#) *brooker.co.za*, December 2017. Archived at [perma.cc/U5AE-CVEM](#)

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. [Dynamo: Amazon's Highly Available Key-Value Store](#). At *21st ACM Symposium on Operating Systems Principles* (SOSP), October 2007.
[doi:10.1145/1294261.1294281](#)

Kathryn Whitenton. [The Need for Speed, 23 Years Later](#). *nngroup.com*, May 2020. Archived at [perma.cc/C4ER-LZYA](#)

Greg Linden. [Marissa Mayer at Web 2.0](#). *glinden.blogspot.com*, November 2005. Archived at [perma.cc/V7EA-3VXB](#)

Jake Brutlag. [Speed Matters for Google Web Search](#). *services.google.com*, June 2009. Archived at [perma.cc/BK7R-X7M2](#)

Eric Schurman and Jake Brutlag. [Performance Related Changes and their User Impact](#). Talk at *Velocity 2009*.

Akamai Technologies, Inc. [The State of Online Retail Performance](#). *akamai.com*, April 2017. Archived at [perma.cc/UEK2-HYCS](#)

Xiao Bai, Ioannis Arapakis, B. Barla Cambazoglu, and Ana Freire. [Understanding and Leveraging the Impact of Response Latency on User Behaviour in Web Search](#). *ACM Transactions on Information Systems*, volume 36, issue 2, article 21, April 2018.
[doi:10.1145/3106372](#)

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

Alex Hidalgo. [Implementing Service Level Objectives: A Practical Guide to SLIs, SLOs, and Error Budgets](#). O'Reilly Media, September 2020. ISBN: 1492076813

Jeffrey C. Mogul and John Wilkes. [Nines are Not Enough: Meaningful Metrics for Clouds](#). At *17th Workshop on Hot Topics in Operating Systems* (HotOS), May 2019. [doi:10.1145/3317550.3321432](https://doi.org/10.1145/3317550.3321432)

Tamás Hauer, Philipp Hoffmann, John Lunney, Dan Ardelean, and Amer Diwan. [Meaningful Availability](#). At *17th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), February 2020.

Ted Dunning. [The t-digest: Efficient estimates of distributions](#). *Software Impacts*, volume 7, article 100049, February 2021. [doi:10.1016/j.simpa.2020.100049](https://doi.org/10.1016/j.simpa.2020.100049)

David Kohn. [How percentile approximation works \(and why it's more useful than averages\)](#). *timescale.com*, September 2021. Archived at perma.cc/3PDP-NR8B

Heinrich Hartmann and Theo Schlossnagle. [Circllhist — A Log-Linear Histogram Data Structure for IT Infrastructure Monitoring](#). *arxiv.org*, January 2020.

Charles Masson, Jee E. Rim, and Homin K. Lee. [DDSketch: A Fast and Fully-Mergeable Quantile Sketch with Relative-Error Guarantees](#). *Proceedings of the VLDB Endowment*, volume 12, issue 12, pages 2195–2205, August 2019. [doi:10.14778/3352063.3352135](https://doi.org/10.14778/3352063.3352135)

Baron Schwartz. [Why Percentiles Don't Work the Way You Think](#). *solarwinds.com*, November 2016. Archived at perma.cc/469T-6UGB

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. Archived at perma.cc/GD2V-DMJW

Felix C. Gärtner. [Fundamentals of fault-tolerant distributed computing in asynchronous environments](#). *ACM Computing Surveys*, volume 31, issue 1, pages 1–26, March 1999. [doi:10.1145/311531.311532](#)

Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. [Basic Concepts and Taxonomy of Dependable and Secure Computing](#). *IEEE Transactions on Dependable and Secure Computing*, volume 1, issue 1, January 2004. [doi:10.1109/TDSC.2004.2](#)

Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. [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.

Casey Rosenthal and Nora Jones. [Chaos Engineering](#). O'Reilly Media, April 2020. ISBN: 9781492043867

Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. [Failure Trends in a Large Disk Drive Population](#). At *5th USENIX Conference on File and Storage Technologies* (FAST), February 2007.

Bianca Schroeder and Garth A. Gibson. [Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?](#) At *5th USENIX Conference on File and Storage Technologies* (FAST), February 2007.

Andy Klein. [Backblaze Drive Stats for Q2 2021](#). *backblaze.com*, August 2021. Archived at [perma.cc/2943-UD5E](#)

Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. [SSD Failures in Datacenters: What? When? and Why?](#) At *9th ACM International on Systems and Storage Conference* (SYSTOR), June 2016. [doi:10.1145/2928275.2928278](#)

Alibaba Cloud Storage Team. [Storage System Design Analysis: Factors Affecting NVMe SSD Performance \(1\)](#). *alibabacloud.com*, January 2019. Archived at [archive.org](#)

Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. [Flash Reliability in Production: The Expected and the Unexpected](#). At *14th USENIX Conference on File and Storage Technologies* (FAST), February 2016.

Jacob Alter, Ji Xue, Alma Dimnaku, and Evgenia Smirni. [SSD failures in the field: symptoms, causes, and prediction models](#). At *International Conference for High Performance Computing, Networking, Storage and Analysis* (SC), November 2019. [doi:10.1145/3295500.3356172](#)

Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. [Availability in Globally Distributed Storage Systems](#). At *9th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2010.

Kashi Venkatesh Vishwanath and Nachiappan Nagappan. [Characterizing Cloud Computing Hardware Reliability](#). At *1st ACM Symposium on Cloud Computing* (SoCC), June 2010. [doi:10.1145/1807128.1807161](#)

Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. [Cores that don't count](#). At *Workshop on Hot Topics in Operating Systems* (HotOS), June 2021. [doi:10.1145/3458336.3465297](#)

Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. [Silent Data Corruptions at Scale](#). *arXiv:2102.11245*, February 2021.

Diogo Behrens, Marco Serafini, Sergei Arnautov, Flavio P. Junqueira, and Christof Fetzer. [Scalable Error Isolation for Distributed Systems](#). At *12th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), May 2015.

Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. [DRAM Errors in the Wild: A Large-Scale Field Study](#). At *11th International Joint Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS), June 2009.
[doi:10.1145/1555349.1555372](#)

Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. [Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors](#). At *41st Annual International Symposium on Computer Architecture* (ISCA), June 2014.
[doi:10.5555/2665671.2665726](#)

Adrian Cockcroft. [Failure Modes and Continuous Resilience](#). *adrianco.medium.com*, November 2019. Archived at [perma.cc/7SYS-BVJP](#)

Shujie Han, Patrick P. C. Lee, Fan Xu, Yi Liu, Cheng He, and Jiongzhou Liu. [An In-Depth Study of Correlated Failures in Production SSD-Based Data Centers](#). At *19th USENIX Conference on File and Storage Technologies* (FAST), February 2021.

Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. [Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs](#). At *6th European Conference on Computer Systems* (EuroSys), April 2011.
[doi:10.1145/1966445.1966477](#)

Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Elazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. [What Bugs Live in the Cloud?](#) At *5th ACM Symposium on Cloud Computing* (SoCC), November 2014.
[doi:10.1145/2670979.2670986](#)

Jay Kreps. [Getting Real About Distributed System Reliability](#). *blog.empathybox.com*, March 2012. Archived at [perma.cc/9B5Q-AEBW](#)

Nelson Minar. [Leap Second Crashes Half the Internet](#). *somebits.com*, July 2012.

Archived at perma.cc/2WB8-D6EU

Hewlett Packard Enterprise. [Support Alerts – Customer Bulletin a00092491en_us](#). *support.hpe.com*, November 2019. Archived at perma.cc/S5F6-7ZAC

Lorin Hochstein. [awesome limits](#). *github.com*, November 2020. Archived at perma.cc/3R5M-E5Q4

Caitie McCaffrey. [Clients Are Jerks: AKA How Halo 4 DoSed the Services at Launch & How We Survived](#). *caitiem.com*, June 2015. Archived at perma.cc/MXX4-W373

Lilia Tang, Chaitanya Bhandari, Yongle Zhang, Anna Karanika, Shuyang Ji, Indranil Gupta, and Tianyin Xu. [Fail through the Cracks: Cross-System Interaction Failures in Modern Cloud Systems](#). At *18th European Conference on Computer Systems* (EuroSys), May 2023. [doi:10.1145/3552326.3587448](https://doi.org/10.1145/3552326.3587448)

Mike Ulrich. [Addressing Cascading Failures](#). In Betsy Beyer, Jennifer Petoff, Chris Jones, and Niall Richard Murphy (ed). [Site Reliability Engineering: How Google Runs Production Systems](#). O'Reilly Media, 2016. ISBN: 9781491929124

Harri Faßbender. [Cascading failures in large-scale distributed systems](#). *blog.mi.hdm-stuttgart.de*, March 2022. Archived at perma.cc/K7VY-YJRX

Richard I. Cook. [How Complex Systems Fail](#). Cognitive Technologies Laboratory, April 2000. Archived at perma.cc/RDS6-2YVA

David D. Woods. [STELLA: Report from the SNAFUcatchers Workshop on Coping With Complexity](#). *snafcatchers.github.io*, March 2017. Archived at archive.org

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.

Sidney Dekker. [The Field Guide to Understanding 'Human Error', 3rd Edition](#). CRC Press, November 2017. ISBN: 9781472439055

Sidney Dekker. [Drift into Failure: From Hunting Broken Components to Understanding Complex Systems](#). CRC Press, 2011. ISBN: 9781315257396

John Allspaw. [Blameless PostMortems and a Just Culture](#). etsy.com, May 2012.
Archived at [perma.cc/YMJ7-NTAP](#)

Itzy Sabo. [Uptime Guarantees — A Pragmatic Perspective](#). world.hey.com, March 2023.
Archived at [perma.cc/F7TU-78JB](#)

Michael Jurewitz. [The Human Impact of Bugs](#). jury.me, March 2013. Archived at [perma.cc/5KQ4-VDYL](#)

Mark Halper. [How Software Bugs led to 'One of the Greatest Miscarriages of Justice' in British History](#). Communications of the ACM, January 2025. [doi:10.1145/3703779](#)

Nicholas Bohm, James Christie, Peter Bernard Ladkin, Bev Littlewood, Paul Marshall, Stephen Mason, Martin Newby, Steven J. Murdoch, Harold Thimbleby, and Martyn Thomas. [The legal rule that computers are presumed to be operating correctly – unforeseen and unjust consequences](#). Briefing note, benthamsgaze.org, June 2022.
Archived at [perma.cc/WQ6X-TMW4](#)

Dan McKinley. [Choose Boring Technology](#). mcfunley.com, March 2015. Archived at [perma.cc/7QW7-J4YP](#)

Andy Warfield. [Building and operating a pretty big storage system called S3](#). allthingsdistributed.com, July 2023. Archived at [perma.cc/7LPK-TP7V](#)

Marc Brooker. [Surprising Scalability of Multitenancy](#). brooker.co.za, March 2023.
Archived at [perma.cc/ZZD9-VV8T](#)

Ben Stopford. [Shared Nothing vs. Shared Disk Architectures: An Independent View](#). benstopford.com, November 2009. Archived at perma.cc/7BXH-EDUR

Michael Stonebraker. [The Case for Shared Nothing](#). *IEEE Database Engineering Bulletin*, volume 9, issue 1, pages 4–9, March 1986.

Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade.

[Socrates: The New SQL Server in the Cloud](#). At *ACM International Conference on Management of Data* (SIGMOD), pages 1743–1756, June 2019.

[doi:10.1145/3299869.3314047](https://doi.org/10.1145/3299869.3314047)

Sam Newman. [Building Microservices, second edition](#). O'Reilly Media, 2021. ISBN: 9781492034025

Nathan Ensmenger. [When Good Software Goes Bad: The Surprising Durability of an Ephemeral Technology](#). At *The Maintainers Conference*, April 2016. Archived at perma.cc/ZXT4-HGZB

Robert L. Glass. [Facts and Fallacies of Software Engineering](#). Addison-Wesley Professional, October 2002. ISBN: 9780321117427

Marianne Bellotti. [Kill It with Fire](#). No Starch Press, April 2021. ISBN: 9781718501188

Lisanne Bainbridge. [Ironies of automation](#). *Automatica*, volume 19, issue 6, pages 775–779, November 1983. [doi:10.1016/0005-1098\(83\)90046-8](https://doi.org/10.1016/0005-1098(83)90046-8)

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

Dotan Horovits. [Open Source for Better Observability](#). *horovits.medium.com*, October 2021. Archived at perma.cc/R2HD-U2ZT

Brian Foote and Joseph Yoder. [Big Ball of Mud](#). At *4th Conference on Pattern Languages of Programs* (PLoP), September 1997. Archived at perma.cc/4GUP-2PBV

Marc Brooker. [What is a simple system?](#) *brooker.co.za*, May 2022. Archived at perma.cc/U72T-BFVE

Frederick P. Brooks. [No Silver Bullet – Essence and Accident in Software Engineering](#). In [The Mythical Man-Month](#), Anniversary edition, Addison-Wesley, 1995. ISBN: 9780201835953

Dan Luu. [Against essential and accidental complexity](#). *danluu.com*, December 2020. Archived at perma.cc/H5ES-69KC

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. [Design Patterns: Elements of Reusable Object-Oriented Software](#). Addison-Wesley Professional, October 1994. ISBN: 9780201633610

Eric Evans. [Domain-Driven Design: Tackling Complexity in the Heart of Software](#). Addison-Wesley Professional, August 2003. ISBN: 9780321125217

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

Enrico Zaninotto. [From X programming to the X organisation](#). At *XP Conference*, May 2002. Archived at perma.cc/R9AR-QCKZ

Chapter 3. Data Models and Query Languages

The limits of my language mean the limits of my world.

—Ludwig Wittgenstein, *Tractatus Logico-Philosophicus* (1922)

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. The GitHub repo for this book is <https://github.com/ept/ddia2-feedback>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out on GitHub.

Data models are perhaps the most important part of developing software, because they have such a profound effect: not only on

how the software is written, but also on how we *think about the problem* that we are solving.

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

1. As an application developer, you look at the real world (in which there are people, organizations, goods, actions, money flows, sensors, etc.) and model it in terms of objects or data structures, and APIs that manipulate those data structures. Those structures are often specific to your application.
2. When you want to store those data structures, you express them in terms of a general-purpose data model, such as JSON or XML documents, tables in a relational database, or vertices and edges in a graph. Those data models are the topic of this chapter.
3. The engineers who built your database software decided on a way of representing that document/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. We will discuss these storage engine designs in [Chapter 4](#).
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.

Several different data models are widely used in practice, often for different purposes. Some types of data and some queries are easy to express in one model, and awkward in another. In this chapter we will explore those trade-offs by comparing the relational model, the document model, graph-based data models, event sourcing, and dataframes. We will also briefly look at query languages that allow you to work with these models. This comparison will help you decide when to use which model.

TERMINOLOGY: DECLARATIVE QUERY LANGUAGES

Many of the query languages in this chapter (such as SQL, Cypher, SPARQL, or Datalog) are *declarative*, which means that you specify the pattern of the data you want—what conditions the results must meet, and how you want the data to be transformed (e.g., sorted, grouped, and aggregated)—but not *how* to achieve that goal. The database system’s query optimizer can decide which indexes and which join algorithms to use, and in which order to execute various parts of the query.

In contrast, with most programming languages you would have to write an *algorithm*—i.e., telling the computer which operations to perform in which order. A declarative query language is attractive because it is typically more concise and easier to write than an explicit algorithm. But more importantly, it also hides implementation details of the query engine, which makes it possible for the database system to introduce performance improvements without requiring any changes to queries. [1].

For example, a database might be able to execute a declarative query in parallel across multiple CPU cores and machines, without you having to worry about how to implement that

parallelism [2]. In a hand-coded algorithm it would be a lot of work to implement such parallel execution yourself.

Relational Model versus Document Model

The best-known data model today is probably that of SQL, based on the relational model proposed by Edgar Codd in 1970 [3]: data is organized into *relations* (called *tables* in SQL), where each relation is an unordered collection of *tuples* (*rows* in SQL).

The relational model was originally 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 tools of choice for most people who needed to store and query data with some kind of regular structure. Many data management use cases are still dominated by relational data decades later—for example, business analytics (see [“Stars and Snowflakes: Schemas for Analytics”](#)).

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 [4]. Instead, SQL has grown to incorporate other data types besides its relational core—for example, adding support for XML, JSON, and graph data [5].

In the 2010s, *NoSQL* was the latest buzzword that tried to overthrow the dominance of relational databases. NoSQL refers not to a single technology, but a loose set of ideas around new data models, schema flexibility, scalability, and a move towards open source licensing models. Some databases branded themselves as *NewSQL*, as they aim to provide the scalability of NoSQL systems along with the data model and transactional guarantees of traditional relational databases. The NoSQL and NewSQL ideas have been very influential in the design of data systems, but as the principles have become widely adopted, use of those terms has faded.

One lasting effect of the NoSQL movement is the popularity of the *document model*, which usually represents data as JSON. This model was originally popularized by specialized document databases such as MongoDB and Couchbase, although most

relational databases have now also added JSON support. Compared to relational tables, which are often seen as having a rigid and inflexible schema, JSON documents are thought to be more flexible.

The pros and cons of document and relational data have been debated extensively; let's examine some of the key points of that debate.

The Object-Relational Mismatch

Much 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*.

NOTE

The term *impedance mismatch* is 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.

Object-relational mapping (ORM)

Object-relational mapping (ORM) frameworks like ActiveRecord and Hibernate reduce the amount of boilerplate code required for this translation layer, but they are often criticized [6]. Some commonly cited problems are:

- ORMs are complex and can't completely hide the differences between the two models, so developers still end up having to think about both the relational and the object representations of the data.
- ORMs are generally only used for OLTP app development (see “[Characterizing Transaction Processing and Analytics](#)”); data engineers making the data available for analytics purposes still need to work with the underlying relational representation, so the design of the relational schema still matters when using an ORM.
- Many ORMs work only with relational OLTP databases. Organizations with diverse data systems such as search engines, graph databases, and NoSQL systems might find ORM support lacking.
- Some ORMs generate relational schemas automatically, but these might be awkward for the users who are accessing the relational data directly, and they might be inefficient on the underlying database. Customizing the ORM's schema and

query generation can be complex and negate the benefit of using the ORM in the first place.

- ORMs make it easy to accidentally write inefficient queries, such as the *N+1 query problem* [7]. For example, say you want to display a list of user comments on a page, so you perform one query that returns N comments, each containing the ID of its author. To show the name of the comment author you need to look up the ID in the users table. In hand-written SQL you would probably perform this join in the query and return the author name along with each comment, but with an ORM you might end up making a separate query on the users table for each of the N comments to look up its author, resulting in $N+1$ database queries in total, which is slower than performing the join in the database. To avoid this problem, you may need to tell the ORM to fetch the author information at the same time as fetching the comments.

Nevertheless, ORMs also have advantages:

- For data that is well suited to a relational model, some kind of translation between the persistent relational and the in-memory object representation is inevitable, and ORMs reduce the amount of boilerplate code required for this translation. Complicated queries may still need to be handled

outside of the ORM, but the ORM can help with the simple and repetitive cases.

- Some ORMs help with caching the results of database queries, which can help reduce the load on the database.
- ORMs can also help with managing schema migrations and other administrative activities.

The document data model for one-to-many relationships

Not all data lends itself well to a relational representation; let's look at an example to explore a limitation of the relational model. [Figure 3-1](#) 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.

Most people have had more than one job in their career (positions), and people may have varying numbers of periods of education and any number of pieces of contact information. One way of representing such *one-to-many relationships* is to put positions, education, and contact information in separate tables, with a foreign key reference to the `users` table, as in [Figure 3-1](#).

<https://www.linkedin.com/in/barackobama/>



Barack Obama
Washington, DC, United States

Former President of the United States of America

Experience

President • United States of America
2009 – 2017

US Senator (D-IL) • United States Senate
2005 – 2008

Education

Juris Doctor, Law • Harvard University
1988 – 1991

Bachelor of Arts • Columbia University
1981 – 1983

Contact Info

Website: barackobama.com
Twitter: @barackobama

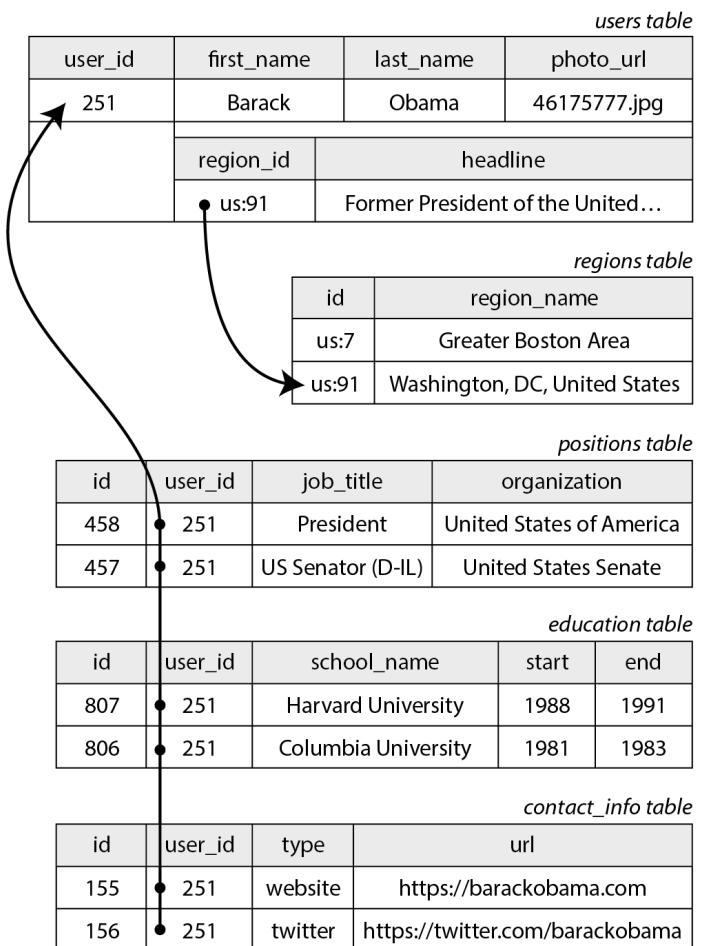


Figure 3-1. Representing a LinkedIn profile using a relational schema.

Another way of representing the same information, which is perhaps more natural and maps more closely to an object structure in application code, is as a JSON document as shown in [Example 3-1](#).

Example 3-1. Representing a LinkedIn profile as a JSON document

```
{  
    "user_id": 251,  
    "first_name": "Barack",  
    "last_name": "Obama",  
    "headline": "Former President of the United  
    "region_id": "us:91",  
    "photo_url": "/p/7/000/253/05b/308dd6e.jpg",  
    "positions": [  
        {"job_title": "President", "organization": "The White House", "start_date": "2009-01-20", "end_date": null},  
        {"job_title": "US Senator (D-IL)", "organization": "Illinois State Senate", "start_date": "2005-01-01", "end_date": "2009-01-20"},  
    ],  
    "education": [  
        {"school_name": "Harvard University", "start_date": "1983-09-01", "end_date": "1985-05-01"},  
        {"school_name": "Columbia University", "start_date": "1980-09-01", "end_date": "1983-05-01"},  
    ],  
    "contact_info": {  
        "website": "https://barackobama.com",  
        "twitter": "https://twitter.com/barackobama"  
    }  
}
```

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 5](#), 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”](#).

The JSON representation has better *locality* than the multi-table schema in [Figure 3-1](#) (see [“Data locality for reads and writes”](#)). 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 [8]. In the JSON representation, all the relevant information is in one place, making the query both faster and simpler.

The one-to-many relationships from the user profile to the user’s positions, educational history, and contact information imply a tree structure in the data, and the JSON representation makes this tree structure explicit (see [Figure 3-2](#)).

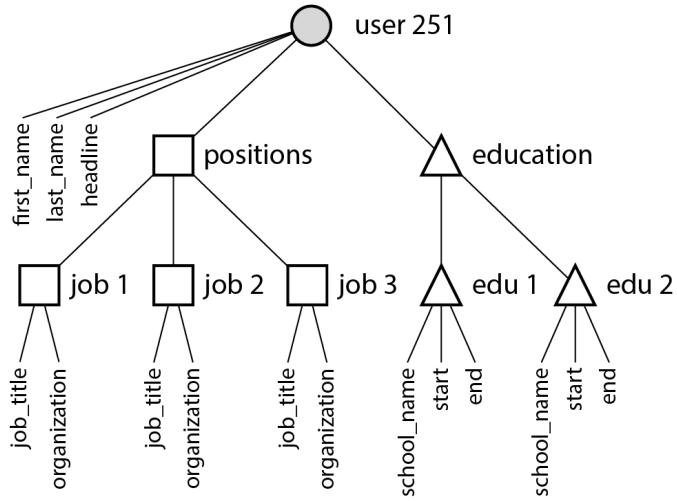


Figure 3-2. One-to-many relationships forming a tree structure.

NOTE

This type of relationship is sometimes called *one-to-few* rather than *one-to-many*, since a résumé typically has a small number of positions [9, 10]. In situations where there may be a genuinely large number of related items—say, comments on a celebrity’s social media post, of which there could be many thousands—embedding them all in the same document may be too unwieldy, so the relational approach in [Figure 3-1](#) is preferable.

Normalization, Denormalization, and Joins

In [Example 3-1](#) in the preceding section, `region_id` is given as an ID, not as the plain-text string "Washington, DC, United States". Why?

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

- Consistent style and spelling across profiles
- Avoiding ambiguity if there are several places with the same name (if the string were just “Washington”, would it refer to DC or to the state?)
- Ease of updating—the name is stored in only one place, so it is easy to update across the board if it ever needs to be changed (e.g., change of a city name due to political events)
- Localization support—when the site is translated into other languages, the standardized lists can be localized, so the region can be displayed in the viewer’s language
- Better search—e.g., a search for people on the US East Coast can match this profile, because the list of regions can encode the fact that Washington is located on the East Coast (which is not apparent from the string “Washington, DC”)

Whether you store an ID or a text string is a question of *normalization*. When you use an ID, your data is more normalized: the information that is meaningful to humans (such as the text *Washington, DC*) 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; this representation is *denormalized*.

The advantage of using an ID is that because it has no meaning to humans, it never needs to change: the ID can remain the same, even if the information it identifies changes. Anything that is meaningful to humans may need to change sometime in the future—and if that information is duplicated, all the redundant copies need to be updated. That requires more code, more write operations, more disk space, and risks inconsistencies (where some copies of the information are updated but others aren't).

The downside of a normalized representation is that every time you want to display a record containing an ID, you have to do an additional lookup to resolve the ID into something human-readable. In a relational data model, this is done using a *join*, for example:

```
SELECT users.*, regions.region_name  
FROM users
```

```
JOIN regions ON users.region_id = regions.id  
WHERE users.id = 251;
```

Document databases can store both normalized and denormalized data, but they are often associated with denormalization—partly because the JSON data model makes it easy to store additional, denormalized fields, and partly because the weak support for joins in many document databases makes normalization inconvenient. Some document databases don't support joins at all, so you have to perform them in application code—that is, you first fetch a document containing an ID, and then perform a second query to resolve that ID into another document. In MongoDB, it is also possible to perform a join using the `$lookup` operator in an aggregation pipeline:

```
db.users.aggregate([  
  { $match: { _id: 251 } },  
  { $lookup: {  
    from: "regions",  
    localField: "region_id",  
    foreignField: "_id",  
    as: "region"
```

```
    }  }  
])
```

Trade-offs of normalization

In the résumé example, while the `region_id` field is a reference into a standardized set of regions, the name of the organization (the company or government where the person worked) and `school_name` (where they studied) are just strings. This representation is denormalized: many people may have worked at the same company, but there is no ID linking them.

Perhaps the organization and school should be entities instead, and the profile should reference their IDs instead of their names? The same arguments for referencing the ID of a region also apply here. For example, say we wanted to include the logo of the school or company in addition to their name:

- In a denormalized representation, we would include the image URL of the logo on every individual person's profile; this makes the JSON document self-contained, but it creates a headache if we ever need to change the logo, because we now need to find all of the occurrences of the old URL and update them [9].

- In a normalized representation, we would create an entity representing an organization or school, and store its name, logo URL, and perhaps other attributes (description, news feed, etc.) once on that entity. Every résumé that mentions the organization would then simply reference its ID, and updating the logo is easy.

As a general principle, normalized data is usually faster to write (since there is only one copy), but slower to query (since it requires joins); denormalized data is usually faster to read (fewer joins), but more expensive to write (more copies to update, more disk space used). You might find it helpful to view denormalization as a form of derived data ([“Systems of Record and Derived Data”](#)), since you need to set up a process for updating the redundant copies of the data.

Besides the cost of performing all these updates, you also need to consider the consistency of the database if a process crashes halfway through making its updates. Databases that offer atomic transactions (see [“Atomicity”](#)) make it easier to remain consistent, but not all databases offer atomicity across multiple documents. It is also possible to ensure consistency through stream processing, which we discuss in [Link to Come].

Normalization tends to be better for OLTP systems, where both reads and updates need to be fast; analytics systems often fare better with denormalized data, since they perform updates in bulk, and the performance of read-only queries is the dominant concern. Moreover, in systems of small to moderate scale, a normalized data model is often best, because you don't have to worry about keeping multiple copies of the data consistent with each other, and the cost of performing joins is acceptable. However, in very large-scale systems, the cost of joins can become problematic.

Denormalization in the social networking case study

In [“Case Study: Social Network Home Timelines”](#) we compared a normalized representation ([Figure 2-1](#)) and a denormalized one (precomputed, materialized timelines): here, the join between `posts` and `follows` was too expensive, and the materialized timeline is a cache of the result of that join. The fan-out process that inserts a new post into followers' timelines was our way of keeping the denormalized representation consistent.

However, the implementation of materialized timelines at X (formerly Twitter) does not store the actual text of each post: each entry actually only stores the post ID, the ID of the user

who posted it, and a little bit of extra information to identify reposts and replies [11]. In other words, it is a precomputed result of (approximately) the following query:

```
SELECT posts.id, posts.sender_id FROM posts
  JOIN follows ON posts.sender_id = follows.follower_id
 WHERE follows.follower_id = current_user
 ORDER BY posts.timestamp DESC
 LIMIT 1000
```

This means that whenever the timeline is read, the service still needs to perform two joins: look up the post ID to fetch the actual post content (as well as statistics such as the number of likes and replies), and look up the sender's profile by ID (to get their username, profile picture, and other details). This process of looking up the human-readable information by ID is called *hydrating* the IDs, and it is essentially a join performed in application code [11].

The reason for storing only IDs in the precomputed timeline is that the data they refer to is fast-changing: the number of likes and replies may change multiple times per second on a popular post, and some users regularly change their username or profile photo. Since the timeline should show the latest like count and profile picture when it is viewed, it would not make

sense to denormalize this information into the materialized timeline. Moreover, the storage cost would be increased significantly by such denormalization.

This example shows that having to perform joins when reading data is not, as sometimes claimed, an impediment to creating high-performance, scalable services. Hydrating post ID and user ID is actually a fairly easy operation to scale, since it parallelizes well, and the cost doesn't depend on the number of accounts you are following or the number of followers you have.

If you need to decide whether to denormalize something in your application, the social network case study shows that the choice is not immediately obvious: the most scalable approach may involve denormalizing some things and leaving other things normalized. You will have to carefully consider how often the information changes, and the cost of reads and writes (which might be dominated by outliers, such as users with many follows/followers in the case of a typical social network). Normalization and denormalization are not inherently good or bad—they are just a trade-off in terms of performance of reads and writes, as well as the amount of effort to implement.

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

While `positions` and `education` in [Figure 3-1](#) are examples of one-to-many or one-to-few relationships (one résumé has several positions, but each position belongs only to one résumé), the `region_id` field is an example of a *many-to-one* relationship (many people live in the same region, but we assume that each person lives in only one region at any one time).

If we introduce entities for organizations and schools, and reference them by ID from the résumé, then we also have *many-to-many* relationships (one person has worked for several organizations, and an organization has several past or present employees). In a relational model, such a relationship is usually represented as an *associative table* or *join table*, as shown in [Figure 3-3](#): each position associates one user ID with one organization ID.

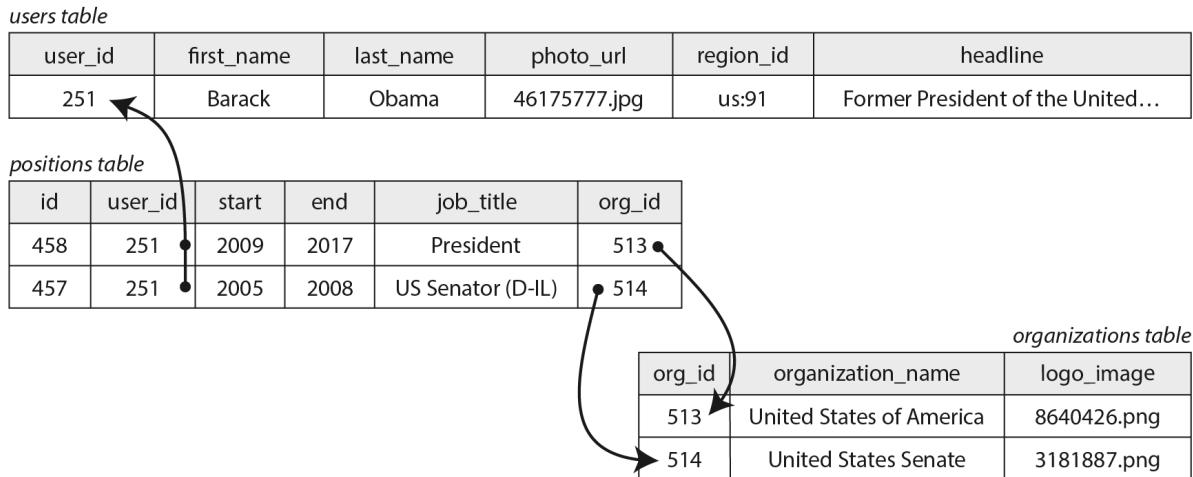


Figure 3-3. Many-to-many relationships in the relational model.

Many-to-one and many-to-many relationships do not easily fit within one self-contained JSON document; they lend themselves more to a normalized representation. In a document model, one possible representation is given in [Example 3-2](#) and illustrated in [Figure 3-4](#): the data within each dotted rectangle can be grouped into one document, but the links to organizations and schools are best represented as references to other documents.

Example 3-2. A résumé that references organizations by ID.

```
{
  "user_id": 251,
  "first_name": "Barack",
  "last_name": "Obama",
  "positions": [
    {"start": 2009, "end": 2017, "job_title": "P}
}
```

```

        {"start": 2005, "end": 2008, "job_title": "US
    ],
    ...
}
```

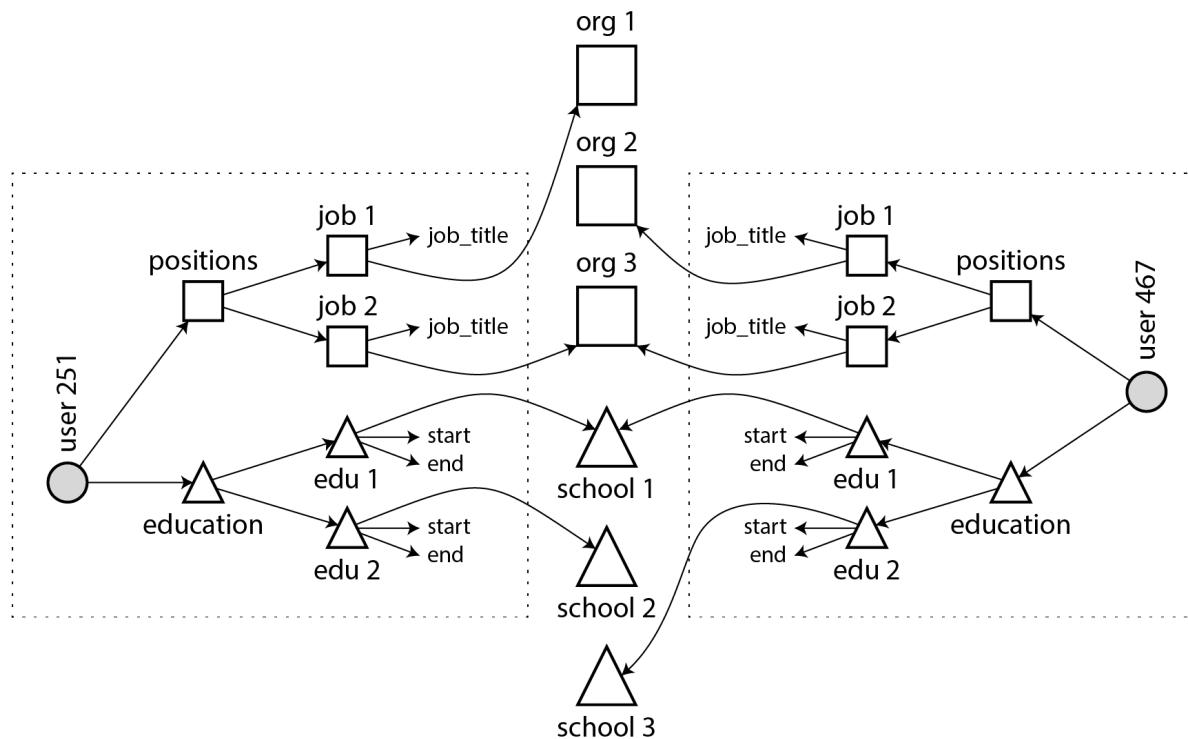


Figure 3-4. Many-to-many relationships in the document model: the data within each dotted box can be grouped into one document.

Many-to-many relationships often need to be queried in “both directions”: for example, finding all of the organizations that a particular person has worked for, and finding all of the people who have worked at a particular organization. One way of enabling such queries is to store ID references on both sides, i.e., a résumé includes the ID of each organization where the

person has worked, and the organization document includes the IDs of the résumés that mention that organization. This representation is denormalized, since the relationship is stored in two places, which could become inconsistent with each other.

A normalized representation stores the relationship in only one place, and relies on *secondary indexes* (which we discuss in [Chapter 4](#)) to allow the relationship to be efficiently queried in both directions. In the relational schema of [Figure 3-3](#), we would tell the database to create indexes on both the `user_id` and the `org_id` columns of the `positions` table.

In the document model of [Example 3-2](#), the database needs to index the `org_id` field of objects inside the `positions` array. Many document databases and relational databases with JSON support are able to create such indexes on values inside a document.

Stars and Snowflakes: Schemas for Analytics

Data warehouses (see [“Data Warehousing”](#)) are usually relational, and there are a few widely-used conventions for the structure of tables in a data warehouse: a *star schema*, *snowflake schema*, *dimensional modeling* [[12](#)], and *one big table*

(OBT). These structures are optimized for the needs of business analysts. ETL processes translate data from operational systems into this schema.

[Figure 3-5](#) shows an example of a star schema that might be found in the data warehouse of a grocery retailer. At the center of the schema is a so-called *fact table* (in this example, it is called `fact_sales`). Each row of the fact table represents an event that occurred at a particular time (here, each row represents a customer's purchase of a product). If we were analyzing website traffic rather than retail sales, each row might represent a page view or a click by a user.

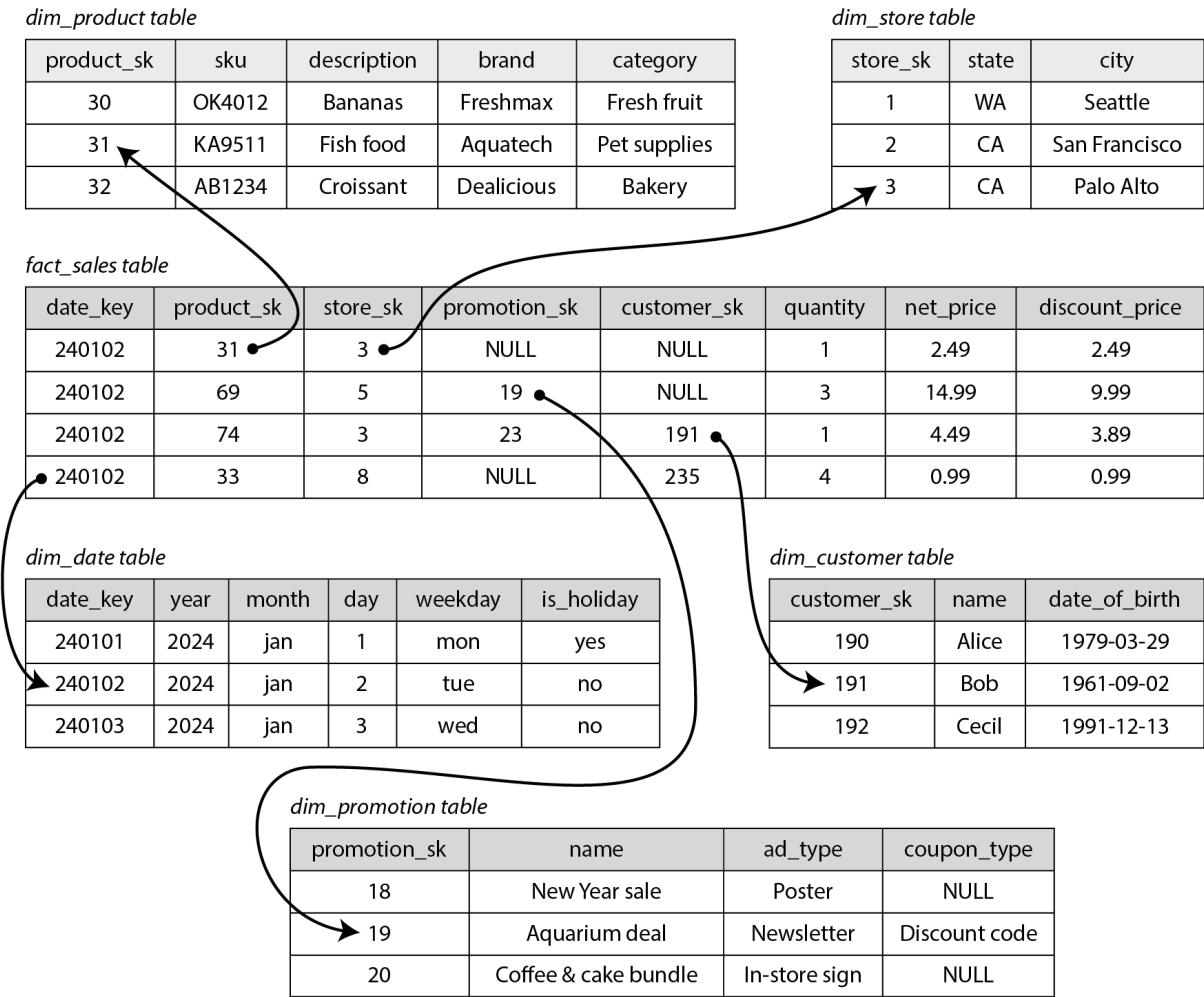


Figure 3-5. Example of a star schema for use in a data warehouse.

Usually, facts are captured as individual events, because this allows maximum flexibility of analysis later. However, this means that the fact table can become extremely large. A big enterprise may have many petabytes of transaction history in its data warehouse, mostly represented as fact tables.

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 3-5](#), 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. Queries often involve multiple joins to multiple dimension tables.

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.

[Figure 3-5](#) is an example of a star schema. The name comes from the fact that when the table relationships are visualized, the fact table is in the middle, surrounded by its dimension tables; the connections to these tables are like the rays of a star.

A variation of this template is known as the *snowflake schema*, where dimensions are further broken down into subdimensions. For example, there could be separate tables for brands and product categories, and each row in the

`dim_product` table could reference the brand and category as foreign keys, rather than storing them as strings in the

`dim_product` table. Snowflake schemas are more normalized than star schemas, but star schemas are often preferred because they are simpler for analysts to work with [12].

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

A star or snowflake schema consists mostly of many-to-one relationships (e.g., many sales occur for one particular product, in one particular store), represented as the fact table having foreign keys into dimension tables, or dimensions into sub-dimensions. In principle, other types of relationship could exist,

but they are often denormalized in order to simplify queries. For example, if a customer buys several different products at once, that multi-item transaction is not represented explicitly; instead, there is a separate row in the fact table for each product purchased, and those facts all just happen to have the same customer ID, store ID, and timestamp.

Some data warehouse schemas take denormalization even further and leave out the dimension tables entirely, folding the information in the dimensions into denormalized columns on the fact table instead (essentially, precomputing the join between the fact table and the dimension tables). This approach is known as *one big table* (OBT), and while it requires more storage space, it sometimes enables faster queries [13].

In the context of analytics, such denormalization is unproblematic, since the data typically represents a log of historical data that is not going to change (except maybe for occasionally correcting an error). The issues of data consistency and write overheads that occur with denormalization in OLTP systems are not as pressing in analytics.

When to Use Which Model

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

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 3-1](#))—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”. If you do need to reference nested items, a relational approach works better, since you can refer to any item directly by its ID.

Some applications allow the user to choose the order of items: for example, imagine a to-do list or issue tracker where the user can drag and drop tasks to reorder them. The document model supports such applications well, because the items (or their IDs) can simply be stored in a JSON array to determine their order. In relational databases there isn't a standard way of representing such reorderable lists, and various tricks are used: sorting by an integer column (requiring renumbering when you insert into the middle), a linked list of IDs, or fractional indexing [14, 15, 16].

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 [17]. 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 when the data is written) [18].

Schema-on-read is similar to dynamic (runtime) type checking in programming languages, whereas schema-on-write is similar to static (compile-time) type checking. Just as the advocates of static and dynamic type checking have big debates about their relative merits [19], enforcement of schemas in database is a contentious topic, and in general there's no right or wrong answer.

The difference between the approaches is particularly noticeable in situations where an application wants to change the format of its data. For example, say you are currently storing each user's full name in one field, and you instead want to store the first name and last name separately [20]. In a document database, you would just start writing new documents with the new fields and have code in the application that handles the case when old documents are read. For example:

```
if (user && user.name && !user.first_name) {  
    // Documents written before Dec 8, 2023 don't  
    user.first_name = user.name.split(" ")[0];  
}
```

The downside of this approach is that every part of your application that reads from the database now needs to deal with documents in old formats that may have been written a long time in the past. On the other hand, in a schema-on-write database, you would typically perform a *migration* along the lines of:

```
ALTER TABLE users ADD COLUMN first_name text DEF...  
UPDATE users SET first_name = split_part(name, '  
UPDATE users SET first_name = substring_index(nar...
```

In most relational databases, adding a column with a default value is fast and unproblematic, even on large tables. However, running the `UPDATE` statement is likely to be slow on a large table, since every row needs to be rewritten, and other schema operations (such as changing the data type of a column) also typically require the entire table to be copied.

Various tools exist to allow this type of schema changes to be performed in the background without downtime [[21](#), [22](#), [23](#), [24](#)], but performing such migrations on large databases remains operationally challenging. Complicated migrations can be avoided by only adding the `first_name` column with a default value of `NULL` (which is fast), and filling it in at read time, like you would with a document database.

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

- There are many different types of objects, and it is not practicable to put each type of object in its own table.
- The structure of the data is determined by external systems over which you have no control and which may change at any time.

In situations like these, a schema may hurt more than it helps, and schemaless documents can be a much more natural data model. But in cases where all records are expected to have the same structure, schemas are a useful mechanism for documenting and enforcing that structure. We will discuss schemas and schema evolution in more detail in [Chapter 5](#).

Data locality for reads and writes

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 3-1](#), 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, which can be wasteful if you only need to access a small part of a large document. On updates to a document, the entire document usually needs to be rewritten. For these reasons, it is generally recommended that you keep documents fairly small and avoid frequent small updates to a document.

However, the idea of storing 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 [25]. Oracle allows the same, using a feature called *multi-table index cluster tables* [26]. The *wide-column* data model popularized by Google's Bigtable, and used e.g. in HBase and Accumulo, has a concept of *column families*, which have a similar purpose of managing locality [27].

Query languages for documents

Another difference between a relational and a document database is the language or API that you use to query it. Most relational databases are queried using SQL, but document databases are more varied. Some allow only key-value access by primary key, while others also offer secondary indexes to query for values inside documents, and some provide rich query languages.

XML databases are often queried using XQuery and XPath, which are designed to allow complex queries, including joins across multiple documents, and also format their results as XML [28]. JSON Pointer [29] and JSONPath [30] provide an equivalent to XPath for JSON. MongoDB's aggregation pipeline, whose `$lookup` operator for joins we saw in “[Normalization, Denormalization, and Joins](#)”, is an example of a query language for collections of JSON documents.

Let's look at another example to get a feel for this language—this time an aggregation, which is especially needed for analytics. Imagine you are a marine biologist, and you add an observation record to your database every time you see animals in the ocean. Now you want to generate a report saying how many sharks you have sighted per month. In PostgreSQL you might express that query like this:

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

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

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

same query can be expressed using MongoDB's aggregation pipeline as follows:

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

The aggregation pipeline language is similar in expressiveness to a subset of SQL, but it uses a JSON-based syntax rather than SQL's English-sentence-style syntax; the difference is perhaps a matter of taste.

Convergence of document and relational databases

Document databases and relational databases started out as very different approaches to data management, but they have grown more similar over time [31]. Relational databases added support for JSON types and query operators, and the ability to

index properties inside documents. Some document databases (such as MongoDB, Couchbase, and RethinkDB) added support for joins, secondary indexes, and declarative query languages.

This convergence of the models is good news for application developers, because the relational model and the document model work best when you can combine both in the same database. Many document databases need relational-style references to other documents, and many relational databases have sections where schema flexibility is beneficial. Relational-document hybrids are a powerful combination.

NOTE

Codd's original description of the relational model [3] actually allowed something similar to JSON within a relational schema. He called it *nonsimple domains*. The idea was that a value in a row doesn't have to just be a primitive datatype like a number or a string, but it could also be a nested relation (table)—so you can have an arbitrarily nested tree structure as a value, much like the JSON or XML support that was added to SQL over 30 years later.

Graph-Like Data Models

We saw earlier that the type of relationships is an important distinguishing feature between different data models. If your application has mostly one-to-many relationships (tree-

structured data) and few other relationships between records, the document model is appropriate.

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

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

Social graphs

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

The web graph

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

Road or rail networks

Vertices are junctions, and edges represent the roads or railway lines between them.

Well-known algorithms can operate on these graphs: for example, map navigation apps search for the shortest path between two points in a road network, and PageRank can be used on the web graph to determine the popularity of a web page and thus its ranking in search results [32].

Graphs can be represented in several different ways. In the *adjacency list* model, each vertex stores the IDs of its neighbor vertices that are one edge away. Alternatively, you can use an *adjacency matrix*, a two-dimensional array where each row and each column corresponds to a vertex, where the value is zero when there is no edge between the row vertex and the column vertex, and where the value is one if there is an edge. The adjacency list is good for graph traversals, and the matrix is good for machine learning (see “[Dataframes, Matrices, and Arrays](#)”).

In the examples just given, all the vertices in a graph represent the same kind of thing (people, web pages, or road junctions, respectively). However, graphs are not limited to such *homogeneous* data: an equally powerful use of graphs is to provide a consistent way of storing completely different types of objects in a single database. For example:

- Facebook maintains a single graph with many different types of vertices and edges: vertices represent people, locations, events, checkins, and comments made by users; edges indicate which people are friends with each other, which checkin happened in which location, who commented on which post, who attended which event, and so on [33].
- Knowledge graphs are used by search engines to record facts about entities that often occur in search queries, such as organizations, people, and places [34]. This information is obtained by crawling and analyzing the text on websites; some websites, such as Wikidata, also publish graph data in a structured form.

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, Memgraph, KùzuDB [35], and others [36]) and the *triple-store* model (implemented by Datomic, AllegroGraph, Blazegraph, and others). These models are fairly similar in what they can express, and some graph databases (such as Amazon Neptune) support both models.

We will also look at four query languages for graphs (Cypher, SPARQL, Datalog, and GraphQL), as well as SQL support for

querying graphs. Other graph query languages exist, such as Gremlin [37], but these will give us a representative overview.

To illustrate these different languages and models, this section uses the graph shown in [Figure 3-6](#) as running example. It could be taken from a social network or a genealogical database: it shows two people, Lucy from Idaho and Alain from Saint-Lô, France. They are married and living in London. Each person and each location is represented as a vertex, and the relationships between them as edges. This example will help demonstrate some queries that are easy in graph databases, but difficult in other models.

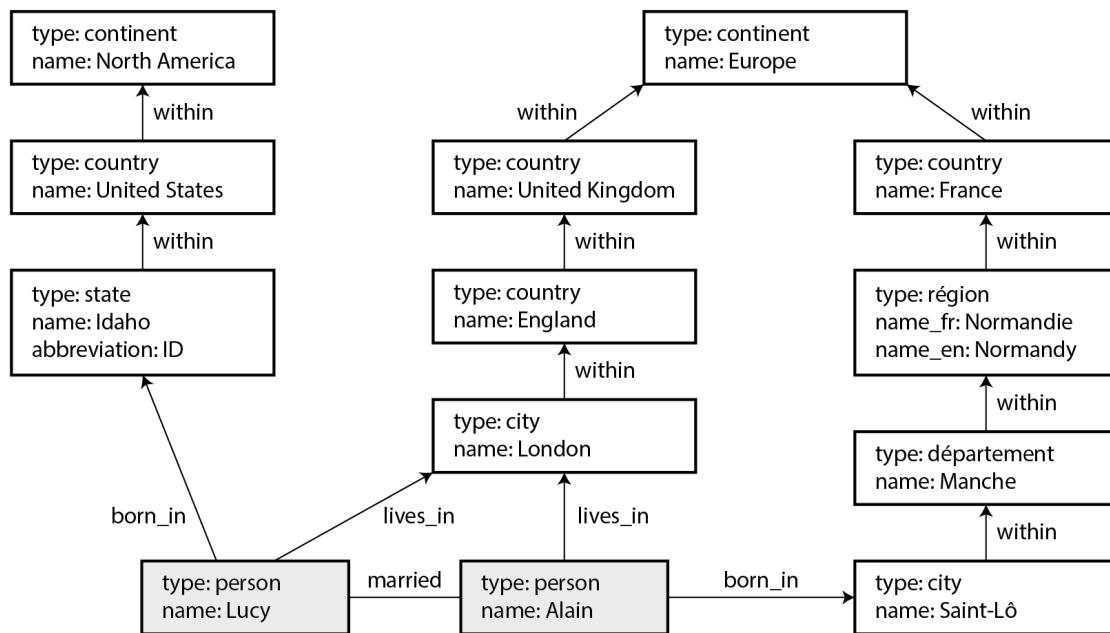


Figure 3-6. Example of graph-structured data (boxes represent vertices, arrows represent edges).

Property Graphs

In the *property graph* (also known as *labeled property graph*) model, each vertex consists of:

- A unique identifier
- A label (string) to describe what type of object this vertex represents
- A set of outgoing edges
- A set of incoming edges
- A collection of properties (key-value pairs)

Each edge consists of:

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

You can think of a graph store as consisting of two relational tables, one for vertices and one for edges, as shown in

[Example 3-3](#) (this schema uses the PostgreSQL `jsonb` datatype to store the properties of each vertex or edge). The head and tail vertex are stored for each edge; if you want the set of incoming

or outgoing edges for a vertex, you can query the `edges` table by `head_vertex` or `tail_vertex`, respectively.

Example 3-3. Representing a property graph using a relational schema

```
CREATE TABLE vertices (
    vertex_id    integer PRIMARY KEY,
    label        text,
    properties   jsonb
);

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   jsonb
);

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

Some important aspects of this model are:

1. Any vertex can have an edge connecting it with any other vertex. There is no schema that restricts which kinds of

things can or cannot be associated.

2. Given any vertex, you can efficiently find both its incoming and its outgoing edges, and thus *traverse* the graph—i.e., follow a path through a chain of vertices—both forward and backward. (That's why [Example 3-3](#) has indexes on both the `tail_vertex` and `head_vertex` columns.)
3. By using different labels for different kinds of vertices and relationships, you can store several different kinds of information in a single graph, while still maintaining a clean data model.

The edges table is like the many-to-many associative table/join table we saw in [“Many-to-One and Many-to-Many Relationships”](#), generalized to allow many different types of relationship to be stored in the same table. There may also be indexes on the labels and the properties, allowing vertices or edges with certain properties to be found efficiently.

NOTE

A limitation of graph models is that an edge can only associate two vertices with each other, whereas a relational join table can represent three-way or even higher-degree relationships by having multiple foreign key references on a single row. Such relationships can be represented in a graph by creating an additional vertex corresponding to each row of the join table, and edges to/from that vertex, or by using a *hypergraph*.

Those features give graphs a great deal of flexibility for data modeling, as illustrated in [Figure 3-6](#). The figure shows a few things that would be difficult to express in a traditional relational schema, such as different kinds of regional structures in different countries (France has *départements* and *régions*, whereas the US has *counties* and *states*), quirks of history such as a country within a country (ignoring for now the intricacies of sovereign states and nations), and varying granularity of data (Lucy's current residence is specified as a city, whereas her place of birth is specified only at the level of a state).

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

The Cypher Query Language

Cypher is a query language for property graphs, originally created for the Neo4j graph database, and later developed into an open standard as *openCypher* [38]. Besides Neo4j, Cypher is supported by Memgraph, KùzuDB [35], Amazon Neptune, Apache AGE (with storage in PostgreSQL), and others. It is named after a character in the movie *The Matrix* and is not related to ciphers in cryptography [39].

[Example 3-4](#) shows the Cypher query to insert the lefthand portion of [Figure 3-6](#) into a graph database. The rest of the graph can be added similarly. Each vertex is given a symbolic name like `usa` or `idaho`. That name is not stored in the database, but only used internally within the query to create edges between the vertices, using an arrow notation: `(idaho) -[:WITHIN] -> (usa)` creates an edge labeled `WITHIN`, with `idaho` as the tail node and `usa` as the head node.

Example 3-4. A subset of the data in [Figure 3-6](#), represented as a Cypher query

```
CREATE
  (namerica :Location {name:'North America', typ
  (usa       :Location {name:'United States', typ
  (idaho     :Location {name:'Idaho', typ
```

```
(lucy      :Person {name:'Lucy' }),  
(idaho) - [:WITHIN ] -> (usa)   - [:WITHIN] -> (name)  
(lucy)   - [:BORN_IN] -> (idaho)
```

When all the vertices and edges of [Figure 3-6](#) 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*. That is, find all the vertices that have a `BORN_IN` edge to a location within the US, and also a `LIVING_IN` edge to a location within Europe, and return the `name` property of each of those vertices.

[Example 3-5](#) shows how to express that query in Cypher. The same arrow notation is used in a `MATCH` clause to find patterns in the graph: `(person) - [:BORN_IN] -> ()` matches any two vertices that are related by an edge labeled `BORN_IN`. The tail vertex of that edge is bound to the variable `person`, and the head vertex is left unnamed.

Example 3-5. Cypher query to find people who emigrated from the US to Europe

```
MATCH  
(person) - [:BORN_IN] -> () - [:WITHIN*0..] -> (:)
```

```
(person) -[:LIVES_IN]-> () -[:WITHIN*0..]-> (:)
RETURN person.name
```

The query can be read as follows:

Find any vertex (call it `person`) that meets both of the following conditions:

1. *`person` has an outgoing `BORN_IN` edge to some vertex.*

From that vertex, you can follow a chain of outgoing `WITHIN` edges until eventually you reach a vertex of type `Location`, whose `name` property is equal to "United States".

2. *That same `person` vertex also has an outgoing `LIVES_IN` edge. Following that edge, and then a chain of outgoing `WITHIN` edges, you eventually reach a vertex of type `Location`, whose `name` property is equal to "Europe".*

For each such `person` vertex, return the `name` property.

There are several possible ways of executing the query. The description given here suggests that you start by scanning all the people in the database, examine each person's birthplace

and residence, and return only those people who meet the criteria.

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

Graph Queries in SQL

Example 3-3 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. Every edge that you traverse in a graph query is effectively a join with the `edges` table. In a relational database, you usually know in advance which joins you need in your query. On the other hand, in a graph query, you may need to traverse a variable number of edges before you find the vertex you're looking for—that is, the number of joins is not fixed in advance.

In our example, that happens in the `(-) [:WITHIN*0..] -> (-)` pattern 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 away in the location hierarchy.

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

Since SQL:1999, this idea of variable-length traversal paths in a query can be expressed using something called *recursive common table expressions* (the `WITH RECURSIVE` syntax).

[Example 3-6](#) shows the same query—finding the names of people who emigrated from the US to Europe—expressed in SQL using this technique. However, the syntax is very clumsy in comparison to Cypher.

Example 3-6. The same query as [Example 3-5](#), written in SQL using recursive common table expressions

```
WITH RECURSIVE
```

```
-- in_usa is the set of vertex IDs of all locat
```

```

in_usa(vertex_id) AS (
    SELECT vertex_id FROM vertices
        WHERE label = 'Location' AND properties->
UNION
    SELECT edges.tail_vertex FROM edges ②
        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 in Europe
in_europe(vertex_id) AS (
    SELECT vertex_id FROM vertices
        WHERE label = 'location' AND properties->
UNION
    SELECT edges.tail_vertex FROM edges
        JOIN in_europe ON edges.head_vertex = in_usa.vertex_id
        WHERE edges.label = 'within'
) ,


-- born_in_usa is the set of vertex IDs of all people born in USA
born_in_usa(vertex_id) AS (④)
    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 (⑤)
    SELECT edges.tail_vertex FROM edges
        JOIN in_europe ON edges.head_vertex = in_usa.vertex_id
        WHERE edges.label = 'lives_in'
)

```

```

SELECT edges.tail_vertex FROM edges
    JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
    WHERE edges.label = 'lives_in'
)

SELECT vertices.properties->>'name'
FROM vertices
-- join to find those people who were both born in the US
JOIN born_in_usa      ON vertices.vertex_id = born_in_usa.vertex_id
JOIN lives_in_europe  ON vertices.vertex_id = lives_in_europe.vertex_id

```

- ❶ First find the vertex whose `name` property has the value "United States", and make it the first element of the set of vertices `in_usa`.
- ❷ Follow all incoming `within` edges from vertices in the set `in_usa`, and add them to the same set, until all incoming `within` edges have been visited.
- ❸ Do the same starting with the vertex whose `name` property has the value "Europe", and build up the set of vertices `in_europe`.
- ❹ For each of the vertices in the set `in_usa`, follow incoming `born_in` edges to find people who were born in some place within the United States.

- ⑤ Similarly, for each of the vertices in the set `in_europe`, follow incoming `lives_in` edges to find people who live in Europe.
- ⑥ Finally, intersect the set of people born in the USA with the set of people living in Europe, by joining them.

The fact that a 4-line Cypher query requires 31 lines in SQL shows how much of a difference the right choice of data model and query language can make. And this is just the beginning; there are more details to consider, e.g., around handling cycles, and choosing between breadth-first or depth-first traversal [40]. Oracle has a different SQL extension for recursive queries, which it calls *hierarchical* [41].

However, the situation may be improving: at the time of writing, there are plans to add a graph query language called GQL to the SQL standard [42, 43], which will provide a syntax inspired by Cypher, GSQL [44], and PGQL [45].

Triple-Stores and SPARQL

The triple-store model is mostly equivalent to the property graph model, using different words to describe the same ideas. It is nevertheless worth discussing, because there are various

tools and languages for triple-stores that can be valuable additions to your toolbox for building applications.

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

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

1. A value of 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. Using the example from [Figure 3-6](#), (*lucy*, *birthYear*, 1989) is like a vertex `lucy` with properties `{"birthYear": 1989}`.
2. Another vertex in the graph. In that case, the predicate is an edge in the graph, the subject is the tail vertex, and the object is the head vertex. For example, in (*lucy*, *marriedTo*, *alain*) the subject and object *lucy* and *alain* are both vertices, and the predicate *marriedTo* is the label of the edge that connects them.

NOTE

To be precise, databases that offer a triple-like data model often need to store some additional metadata on each tuple. For example, AWS Neptune uses quads (4-tuples) by adding a graph ID to each triple [46]; Datomic uses 5-tuples, extending each triple with a transaction ID and a boolean to indicate deletion [47]. Since these databases retain the basic *subject-predicate-object* structure explained above, this book nevertheless calls them triple-stores.

[Example 3-7](#) shows the same data as in [Example 3-4](#), written as triples in a format called *Turtle*, a subset of *Notation3 (N3)* [48].

Example 3-7. A subset of the data in [Figure 3-6](#), 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 readable: see [Example 3-8](#).

Example 3-8. A more concise way of writing the data in [Example 3-7](#)

```
@prefix : <urn:example:>.  
_:lucy a :Person; :name "Lucy"; :k  
_:idaho a :Location; :name "Idaho"; :t  
_:usa a :Location; :name "United States"; :t  
_:namerica a :Location; :name "North America"; :t
```

THE SEMANTIC WEB

Some of the research and development effort on triple stores was motivated by the *Semantic Web*, an early-2000s effort to facilitate internet-wide data exchange by publishing data not only as human-readable web pages, but also in a standardized, machine-readable format. Although the Semantic Web as originally envisioned did not succeed [49, 50], the legacy of the Semantic Web project lives on in a couple of specific technologies: *linked data* standards such as JSON-LD [51], *ontologies* used in biomedical science [52], Facebook's Open Graph protocol [53] (which is used for link unfurling [54]), knowledge graphs such as Wikidata, and standardized vocabularies for structured data maintained by schema.org.

Triple-stores are another Semantic Web technology that has found use outside of its original use case: even if you have no interest in the Semantic Web, triples can be a good internal data model for applications.

The RDF data model

The Turtle language we used in [Example 3-8](#) is actually a way of encoding data in the *Resource Description Framework* (RDF) [55], a data model that was designed for the Semantic Web. RDF

data can also be encoded in other ways, for example (more verbosely) in XML, as shown in [Example 3-9](#). Tools like Apache Jena can automatically convert between different RDF encodings.

Example 3-9. The data of [Example 3-8](#), expressed using RDF/XML syntax

```
<rdf:RDF xmlns="urn:example:"  
          xmlns:rdf="http://www.w3.org/1999/02/22-rdf-s  
  
          <Location rdf:nodeID="idaho">  
              <name>Idaho</name>  
              <type>state</type>  
              <within>  
                  <Location rdf:nodeID="usa">  
                      <name>United States</name>  
                      <type>country</type>  
                      <within>  
                          <Location rdf:nodeID="namerica">  
                              <name>North America</name>  
                              <type>continent</type>  
                          </Location>  
                      </within>  
                  </Location>  
              </within>  
          </Location>  
      </within>  
  </Location>
```

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

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

```
<http://other.org/foo#within> and
<http://other.org/foo#lives_in>.
```

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

you can just specify this prefix once at the top of the file, and then forget about it.

The SPARQL query language

SPARQL is a query language for triple-stores using the RDF data model [56]. (It is an acronym for *SPARQL Protocol and RDF Query Language*, pronounced “sparkle.”) It predates Cypher, and since Cypher’s pattern matching is borrowed from SPARQL, they look quite similar.

The same query as before—finding people who have moved from the US to Europe—is similarly concise in SPARQL as it is in Cypher (see [Example 3-10](#)).

Example 3-10. The same query as [Example 3-5](#), 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)  
?person :bornIn / :within* ?location.
```

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

```
(usa {name:'United States'})    # Cypher  
  
?usa :name "United States".      # SPARQL
```

SPARQL is supported by Amazon Neptune, AllegroGraph, Blazegraph, OpenLink Virtuoso, Apache Jena, and various other triple stores [36].

Datalog: Recursive Relational Queries

Datalog is a much older language than SPARQL or Cypher: it arose from academic research in the 1980s [57, 58, 59]. It is less well known among software engineers and not widely supported in mainstream databases, but it ought to be better-known since it is a very expressive language that is particularly powerful for complex queries. Several niche databases, including Datomic, LogicBlox, CozoDB, and LinkedIn's LiQuid [60] use Datalog as their query language.

Datalog is actually based on a relational data model, not a graph, but it appears in the graph databases section of this book because recursive queries on graphs are a particular strength of Datalog.

The contents of a Datalog database consists of *facts*, and each fact corresponds to a row in a relational table. For example, say we have a table *location* containing locations, and it has three columns: *ID*, *name*, and *type*. The fact that the US is a country could then be written as `location(2, "United States", "country")`, where `2` is the ID of the US. In general, the statement `table(val1, val2, ...)` means that `table` contains a row where the first column contains `val1`, the second column contains `val2`, and so on.

[Example 3-11](#) shows how to write the data from the left-hand side of [Figure 3-6](#) in Datalog. The edges of the graph (`within`, `born_in`, and `lives_in`) are represented as two-column join tables. For example, Lucy has the ID 100 and Idaho has the ID 3, so the relationship “Lucy was born in Idaho” is represented as `born_in(100, 3)`.

Example 3-11. A subset of the data in [Figure 3-6](#), represented as Datalog facts

```
location(1, "North America", "continent").  
location(2, "United States", "country").  
location(3, "Idaho", "state").  
  
within(2, 1).      /* US is in North America */  
within(3, 2).      /* Idaho is in the US */  
  
person(100, "Lucy").  
born_in(100, 3). /* Lucy was born in Idaho */
```

Now that we have defined the data, we can write the same query as before, as shown in [Example 3-12](#). 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, a programming language that you might have seen before if you’ve studied computer science.

Example 3-12. The same query as [Example 3-5](#), expressed in Datalog

```
within_recursive(LocID, PlaceName) :- location(LocID, PlaceName), !.  
within_recursive(LocID, PlaceName) :- within(LocID, PlaceName),  
        within_recursive(PlaceName).  
  
migrated(PName, BornIn, LivingIn) :- person(Person),  
        born_in(Person, BornIn),  
        within_recursive(BornIn, PlaceName),  
        lives_in(Person, PlaceName),  
        within_recursive(PlaceName, LivingIn).  
  
us_to_europe(Person) :- migrated(Person, "United States").  
/* us_to_europe contains the row "Lucy". */
```

Cypher and SPARQL jump in right away with `SELECT`, but Datalog takes a small step at a time. We define *rules* that derive new virtual tables from the underlying facts. These derived tables are like (virtual) SQL views: they are not stored in the database, but you can query them in the same way as a table containing stored facts.

In [Example 3-12](#) we define three derived tables:

`within_recursive`, `migrated`, and `us_to_europe`. The

name and columns of the virtual tables are defined by what appears before the `:-` symbol of each rule. For example, `migrated(PName, BornIn, LivingIn)` is a virtual table with three columns: the name of a person, the name of the place where they were born, and the name of the place where they are living.

The content of a virtual table is defined by the part of the rule after the `:-` symbol, where we try to find rows that match a certain pattern in the tables. For example, `person(PersonID, PName)` matches the row `person(100, "Lucy")`, with the variable `PersonID` bound to the value `100` and the variable `PName` bound to the value `"Lucy"`. A rule applies if the system can find a match for *all* patterns on the righthand side of the `:-` operator. When the rule applies, it's as though the lefthand side of the `:-` was added to the database (with variables replaced by the values they matched).

One possible way of applying the rules is thus (and as illustrated in [Figure 3-7](#)):

1. `location(1, "North America", "continent")` exists in the database, so rule 1 applies. It generates
`within_recursive(1, "North America")`.

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

By repeated application of rules 1 and 2, the `within_recursive` virtual table can tell us all the locations in North America (or any other location) contained in our database.

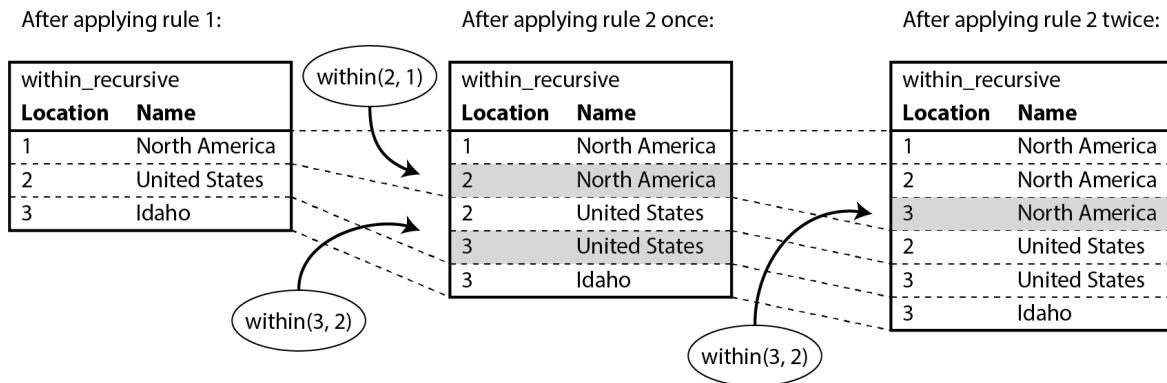


Figure 3-7. Determining that Idaho is in North America, using the Datalog rules from [Example 3-12](#).

Now rule 3 can find people who were born in some location `BornIn` and live in some location `LivingIn`. Rule 4 invokes

rule 3 with `BornIn = 'United States'` and `LivingIn = 'Europe'`, and returns only the names of the people who match the search. By querying the contents of the virtual `us_to_europe` table, the Datalog system finally gets the same answer as in the earlier Cypher and SPARQL queries.

The Datalog approach requires a different kind of thinking compared to the other query languages discussed in this chapter. It allows complex queries to be built up rule by rule, with one rule referring to other rules, similarly to the way that you break down code into functions that call each other. Just like functions can be recursive, Datalog rules can also invoke themselves, like rule 2 in [Example 3-12](#), which enables graph traversals in Datalog queries.

GraphQL

GraphQL is a query language that, by design, is much more restrictive than the other query languages we have seen in this chapter. The purpose of GraphQL is to allow client software running on a user's device (such as a mobile app or a JavaScript web app frontend) to request a JSON document with a particular structure, containing the fields necessary for rendering its user interface. GraphQL interfaces allow

developers to rapidly change queries in client code without changing server-side APIs.

GraphQL's flexibility comes at a cost. Organizations that adopt GraphQL often need tooling to convert GraphQL queries into requests to internal services, which often use REST or gRPC (see [Chapter 5](#)). Authorization, rate limiting, and performance challenges are additional concerns [\[61\]](#). GraphQL's query language is also limited since GraphQL come from an untrusted source. The language does not allow anything that could be expensive to execute, since otherwise users could perform denial-of-service attacks on a server by running lots of expensive queries. In particular, GraphQL does not allow recursive queries (unlike Cypher, SPARQL, SQL, or Datalog), and it does not allow arbitrary search conditions such as “find people who were born in the US and are now living in Europe” (unless the service owners specifically choose to offer such search functionality).

Nevertheless, GraphQL is useful. [Example 3-13](#) shows how you might implement a group chat application such as Discord or Slack using GraphQL. The query requests all the channels that the user has access to, including the channel name and the 50 most recent messages in each channel. For each message it requests the timestamp, the message content, and the name and

profile picture URL for the sender of the message. Moreover, if a message is a reply to another message, the query also requests the sender name and the content of the message it is replying to (which might be rendered in a smaller font above the reply, in order to provide some context).

Example 3-13. Example GraphQL query for a group chat application

```
query ChatApp {  
  channels {  
    name  
    recentMessages(latest: 50) {  
      timestamp  
      content  
      sender {  
        fullName  
        imageUrl  
      }  
      replyTo {  
        content  
        sender {  
          fullName  
        }  
      }  
    }  
  }  
}
```

```
    }  
}
```

[Example 3-14](#) shows what a response to the query in [Example 3-13](#) might look like. The response is a JSON document that mirrors the structure of the query: it contains exactly those attributes that were requested, no more and no less. This approach has the advantage that the server does not need to know which attributes the client requires in order to render the user interface; instead, the client can simply request what it needs. For example, this query does not request a profile picture URL for the sender of the `replyTo` message, but if the user interface were changed to add that profile picture, it would be easy for the client to add the required `imageUrl` attribute to the query without changing the server.

Example 3-14. A possible response to the query in [Example 3-13](#)

```
{  
  "data": {  
    "channels": [  
      {  
        "name": "#general",  
        "recentMessages": [  
          {  
            "text": "Hello everyone!",  
            "sender": "Alice",  
            "timestamp": "2023-10-01T12:00:00Z"  
          },  
          {  
            "text": "How are you?",  
            "sender": "Bob",  
            "timestamp": "2023-10-01T12:05:00Z"  
          },  
          {  
            "text": "I'm good, thanks!",  
            "sender": "Alice",  
            "timestamp": "2023-10-01T12:10:00Z"  
          },  
          {  
            "text": "Great to hear that!",  
            "sender": "Bob",  
            "timestamp": "2023-10-01T12:15:00Z"  
          }  
        ]  
      }  
    ]  
  }  
}
```

```
"timestamp": 1693143014,  
  "content": "Hey! How are y'all doing?",  
  "sender": {"fullName": "Aaliyah", "image": "https://..."},  
  "replyTo": null  
},  
{  
  "timestamp": 1693143024,  
  "content": "Great! And you?",  
  "sender": {"fullName": "Caleb", "image": "https://..."},  
  "replyTo": {  
    "content": "Hey! How are y'all doing?",  
    "sender": {"fullName": "Aaliyah"}  
  },  
  ...
```

In [Example 3-14](#) the name and image URL of a message sender is embedded directly in the message object. If the same user sends multiple messages, this information is repeated on each message. In principle, it would be possible to reduce this duplication, but GraphQL makes the design choice to accept a larger response size in order to make it simpler to render the user interface based on the data.

The `replyTo` field is similar: in [Example 3-14](#), the second message is a reply to the first, and the content (“Hey!...”) and

sender Aaliyah are duplicated under `replyTo`. It would be possible to instead return the ID of the message being replied to, but then the client would have to make an additional request to the server if that ID is not among the 50 most recent messages returned. Duplicating the content makes it much simpler to work with the data.

The server's database can store the data in a more normalized form, and perform the necessary joins to process a query. For example, the server might store a message along with the user ID of the sender and the ID of the message it is replying to; when it receives a query like the one above, the server would then resolve those IDs to find the records they refer to. However, the client can only ask the server to perform joins that are explicitly offered in the GraphQL schema.

Even though the response to a GraphQL query looks similar to a response from a document database, and even though it has “graph” in the name, GraphQL can be implemented on top of any type of database—relational, document, or graph.

Event Sourcing and CQRS

In all the data models we have discussed so far, the data is queried in the same form as it is written—be it JSON documents, rows in tables, or vertices and edges in a graph. However, in complex applications it can sometimes be difficult to find a single data representation that is able to satisfy all the different ways that the data needs to be queried and presented. In such situations, it can be beneficial to write data in one form, and then to derive from it several representations that are optimized for different types of reads.

We previously saw this idea in [“Systems of Record and Derived Data”](#), and ETL (see [“Data Warehousing”](#)) is one example of such a derivation process. Now we will take the idea further. If we are going to derive one data representation from another anyway, we can choose different representations that are optimized for writing and for reading, respectively. How would you model your data if you only wanted to optimize it for writing, and if efficient queries were of no concern?

Perhaps the simplest, fastest, and most expressive way of writing data is an *event log*: every time you want to write some data, you encode it as a self-contained string (perhaps as JSON),

including a timestamp, and then append it to a sequence of events. Events in this log are *immutable*: you never change or delete them, you only ever append more events to the log (which may supersede earlier events). An event can contain arbitrary properties.

[Figure 3-8](#) shows an example that could be taken from a conference management system. A conference can be a complex business domain: not only can individual attendees register and pay by card, but companies can also order seats in bulk, pay by invoice, and then later assign the seats to individual people. Some number of seats may be reserved for speakers, sponsors, volunteer helpers, and so on. Reservations may also be cancelled, and meanwhile, the conference organizer might change the capacity of the event by moving it to a different room. With all of this going on, simply calculating the number of available seats becomes a challenging query.

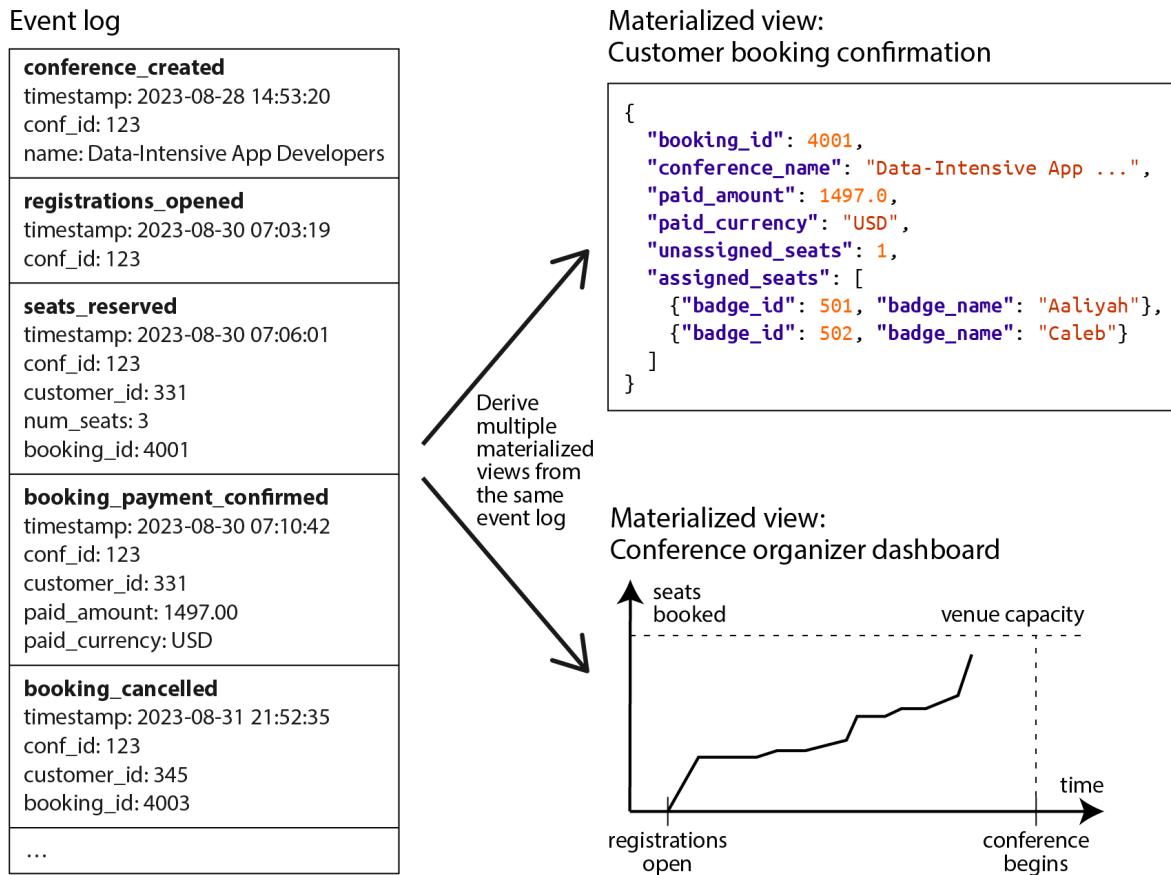


Figure 3-8. Using a log of immutable events as source of truth, and deriving materialized views from it.

In [Figure 3-8](#), every change to the state of the conference (such as the organizer opening registrations, or attendees making and cancelling registrations) is first stored as an event. Whenever an event is appended to the log, several *materialized views* (also known as *projections* or *read models*) are also updated to reflect the effect of that event. In the conference example, there might be one materialized view that collects all information related to the status of each booking, another that computes charts for the

conference organizer's dashboard, and a third that generates files for the printer that produces the attendees' badges.

The idea of using events as the source of truth, and expressing every state change as an event, is known as *event sourcing* [62, 63]. The principle of maintaining separate read-optimized representations and deriving them from the write-optimized representation is called *command query responsibility segregation (CQRS)* [64]. These terms originated in the domain-driven design (DDD) community, although similar ideas have been around for a long time, for example in *state machine replication* (see [Link to Come]).

When a request from a user comes in, it is called a *command*, and it first needs to be validated. Only once the command has been executed and it has been determined to be valid (e.g., there were enough available seats for a requested reservation), it becomes a fact, and the corresponding event is added to the log. Consequently, the event log should contain only valid events, and a consumer of the event log that builds a materialized view is not allowed to reject an event.

When modelling your data in an event sourcing style, it is recommended that you name your events in the past tense (e.g., "the seats were booked"), because an event is a record of the

fact that something has happened in the past. Even if the user later decides to change or cancel, the fact remains true that they formerly held a booking, and the change or cancellation is a separate event that is added later.

A similarity between event sourcing and a star schema fact table, as discussed in [“Stars and Snowflakes: Schemas for Analytics”](#), is that both are collections of events that happened in the past. However, rows in a fact table all have the same set of columns, whereas in event sourcing there may be many different event types, each with different properties. Moreover, a fact table is an unordered collection, while in event sourcing the order of events is important: if a booking is first made and then cancelled, processing those events in the wrong order would not make sense.

Event sourcing and CQRS have several advantages:

- For the people developing the system, events better communicate the intent of *why* something happened. For example, it's easier to understand the event “the booking was cancelled” than “the `active` column on row 4001 of the `bookings` table was set to `false`, three rows associated with that booking were deleted from the `seat_assignments` table, and a row representing the

refund was inserted into the `payments` table". Those row modifications may still happen when a materialized view processes the cancellation event, but when they are driven by an event, the reason for the updates becomes much clearer.

- A key principle of event sourcing is that the materialized views are derived from the event log in a reproducible way: you should always be able to delete the materialized views and recompute them by processing the same events in the same order, using the same code. If there was a bug in the view maintenance code, you can just delete the view and recompute it with the new code. It's also easier to find the bug because you can re-run the view maintenance code as often as you like and inspect its behavior.
- You can have multiple materialized views that are optimized for the particular queries that your application requires. They can be stored either in the same database as the events or a different one, depending on your needs. They can use any data model, and they can be denormalized for fast reads. You can even keep a view only in memory and avoid persisting it, as long as it's okay to recompute the view from the event log whenever the service restarts.
- If you decide you want to present the existing information in a new way, it is easy to build a new materialized view from

the existing event log. You can also evolve the system to support new features by adding new types of events, or new properties to existing event types (any older events remain unmodified). You can also chain new behaviors off existing events (for example, when a conference attendee cancels, their seat could be offered to the next person on the waiting list).

- If an event was written in error you can delete it again, and then you can rebuild the views without the deleted event. On the other hand, in a database where you update and delete data directly, a committed transaction is often difficult to reverse. Event sourcing can therefore reduce the number of irreversible actions in the system, making it easier to change (see [“Evolvability: Making Change Easy”](#)).
- The event log can also serve as an audit log of everything that happened in the system, which is valuable in regulated industries that require such auditability.

However, event sourcing and CQRS also have downsides:

- You need to be careful if external information is involved. For example, say an event contains a price given in one currency, and for one of the views it needs to be converted into another currency. Since the exchange rate may fluctuate, it would be problematic to fetch the exchange rate from an

external source when processing the event, since you would get a different result if you recompute the materialized view on another date. To make the event processing logic deterministic, you either need to include the exchange rate in the event itself, or have a way of querying the historical exchange rate at the timestamp indicated in the event, ensuring that this query always returns the same result for the same timestamp.

- The requirement that events are immutable creates problems if events contain personal data from users, since users may exercise their right (e.g., under the GDPR) to request deletion of their data. If the event log is on a per-user basis, you can just delete the whole log for that user, but that doesn't work if your event log contains events relating to multiple users. You can try storing the personal data outside of the actual event, or encrypting it with a key that you can later choose to delete, but that also makes it harder to recompute derived state when needed.
- Reprocessing events requires care if there are externally visible side-effects—for example, you probably don't want to resend confirmation emails every time you rebuild a materialized view.

You can implement event sourcing on top of any database, but there are also some systems that are specifically designed to

support this pattern, such as EventStoreDB, MartenDB (based on PostgreSQL), and Axon Framework. You can also use message brokers such as Apache Kafka to store the event log, and stream processors can keep the materialized views up-to-date; we will return to these topics in [Link to Come].

The only important requirement is that the event storage system must guarantee that all materialized views process the events in exactly the same order as they appear in the log; as we shall see in [Link to Come], this is not always easy to achieve in a distributed system.

Dataframes, Matrices, and Arrays

The data models we have seen so far in this chapter are generally used for both transaction processing and analytics purposes (see [“Analytical versus Operational Systems”](#)). There are also some data models that you are likely to encounter in an analytical or scientific context, but that rarely feature in OLTP systems: dataframes and multidimensional arrays of numbers such as matrices.

Dataframes are a data model supported by the R language, the Pandas library for Python, Apache Spark, ArcticDB, Dask, and

other systems. They are a popular tool for data scientists preparing data for training machine learning models, but they are also widely used for data exploration, statistical data analysis, data visualization, and similar purposes.

At first glance, a dataframe is similar to a table in a relational database or a spreadsheet. It supports relational-like operators that perform bulk operations on the contents of the dataframe: for example, applying a function to all of the rows, filtering the rows based on some condition, grouping rows by some columns and aggregating other columns, and joining the rows in one dataframe with another dataframe based on some key (what a relational database calls *join* is typically called *merge* on dataframes).

Instead of a declarative query such as SQL, a dataframe is typically manipulated through a series of commands that modify its structure and content. This matches the typical workflow of data scientists, who incrementally “wrangle” the data into a form that allows them to find answers to the questions they are asking. These manipulations usually take place on the data scientist’s private copy of the dataset, often on their local machine, although the end result may be shared with other users.

Dataframe APIs also offer a wide variety of operations that go far beyond what relational databases offer, and the data model is often used in ways that are very different from typical relational data modelling [65]. For example, a common use of dataframes is to transform data from a relational-like representation into a matrix or multidimensional array representation, which is the form that many machine learning algorithms expect of their input.

A simple example of such a transformation is shown in [Figure 3-9](#). On the left we have a relational table of how different users have rated various movies (on a scale of 1 to 5), and on the right the data has been transformed into a matrix where each column is a movie and each row is a user (similarly to a *pivot table* in a spreadsheet). The matrix is *sparse*, which means there is no data for many user-movie combinations, but this is fine. This matrix may have many thousands of columns and would therefore not fit well in a relational database, but dataframes and libraries that offer sparse arrays (such as NumPy for Python) can handle such data easily.

The diagram illustrates the transformation of a relational database table into a matrix representation. On the left, a table named 'movie_ratings' is shown with columns: user_id, movie_id, rating, and date. The data consists of 12 rows of user ratings for various movies. An arrow points from this table to a matrix on the right. The matrix has 'users' listed on the vertical axis and 'movies' listed on the horizontal axis. The columns are labeled 10, 11, 12, 13, 14, and 15. The matrix values represent the rating given by each user to each movie.

		movies					
		10	11	12	13	14	15
users	100			4		5	
	101	1	3		2		
	102					4	
	103				3		
	104				3	5	
	105		4				
	106					5	

Figure 3-9. Transforming a relational database of movie ratings into a matrix representation.

A matrix can only contain numbers, and various techniques are used to transform non-numerical data into numbers in the matrix. For example:

- Dates (which are omitted from the example matrix in [Figure 3-9](#)) could be scaled to be floating-point numbers within some suitable range.
- For columns that can only take one of a small, fixed set of values (for example, the genre of a movie in a database of movies), a *one-hot encoding* is often used: we create a column for each possible value (one for “comedy”, one for “drama”, one for “horror”, etc.), and for each row representing a movie, we put a 1 in the column corresponding to the genre of that movie, and a 0 in all the other columns. This

representation also easily generalizes to movies that fit within several genres.

Once the data is in the form of a matrix of numbers, it is amenable to linear algebra operations, which form the basis of many machine learning algorithms. For example, the data in [Figure 3-9](#) could be a part of a system for recommending movies that the user may like. Dataframes are flexible enough to allow data to be gradually evolved from a relational form into a matrix representation, while giving the data scientist control over the representation that is most suitable for achieving the goals of the data analysis or model training process.

There are also databases such as TileDB [\[66\]](#) that specialize in storing large multidimensional arrays of numbers; they are called *array databases* and are most commonly used for scientific datasets such as geospatial measurements (raster data on a regularly spaced grid), medical imaging, or observations from astronomical telescopes [\[67\]](#). Dataframes are also used in the financial industry for representing *time series data*, such as the prices of assets and trades over time [\[68\]](#).

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.

The *relational model*, despite being more than half a century old, remains an important data model for many applications—especially in data warehousing and business analytics, where relational star or snowflake schemas and SQL queries are ubiquitous. However, several alternatives to relational data have also become popular in other domains:

- The *document model* targets use cases where data comes in self-contained JSON documents, and where relationships between one document and another are rare.
- *Graph data models* go in the opposite direction, targeting use cases where anything is potentially related to everything, and where queries potentially need to traverse multiple hops to find the data of interest (which can be expressed using recursive queries in Cypher, SPARQL, or Datalog).

- *Dataframes* generalize relational data to large numbers of columns, and thereby provide a bridge between databases and the multidimensional arrays that form the basis of much machine learning, statistical data analysis, and scientific computing.

To some degree, one model can be emulated in terms of another model—for example, graph data can be represented in a relational database—but the result can be awkward, as we saw with the support for recursive queries in SQL.

Various specialist databases have therefore been developed for each data model, providing query languages and storage engines that are optimized for a particular model. However, there is also a trend for databases to expand into neighboring niches by adding support for other data models: for example, relational databases have added support for document data in the form of JSON columns, document databases have added relational-like joins, and support for graph data within SQL is gradually improving.

Another model we discussed is *event sourcing*, which represents data as an append-only log of immutable events, and which can be advantageous for modeling activities in complex business domains. An append-only log is good for writing data (as we

shall see in [Chapter 4](#)); in order to support efficient queries, the event log is translated into read-optimized materialized views through CQRS.

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

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

- Researchers working with genome data often need to perform *sequence-similarity searches*, which means taking one very long string (representing a DNA molecule) and matching it against a large database of strings that are similar, but not identical. None of the databases described here can handle this kind of usage, which is why researchers have written specialized genome database software like GenBank [\[69\]](#).
- Many financial systems use *ledgers* with double-entry accounting as their data model. This type of data can be

represented in relational databases, but there are also databases such as TigerBeetle that specialize in this data model. Cryptocurrencies and blockchains are typically based on distributed ledgers, which also have value transfer built into their data model.

- *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 and vector search in [“Full-Text Search”](#).

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.

FOOTNOTES

REFERENCES

amie Brandon. [Unexplanations: query optimization works because sql is declarative](#). *scattered-thoughts.net*, February 2024. Archived at [perma.cc/P6W2-WMFZ](#)

oseph M. Hellerstein. [The Declarative Imperative: Experiences and Conjectures in Distributed Logic](#). Tech report UCB/EECS-2010-90, Electrical Engineering and Computer Sciences, University of California at Berkeley, June 2010. Archived at [perma.cc/K56R-VVQM](#)

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

Michael Stonebraker and Joseph M. Hellerstein. [What Goes Around Comes Around](#). In *Readings in Database Systems*, 4th edition, MIT Press, pages 2–41, 2005. ISBN: 9780262693141

Markus Winand. [Modern SQL: Beyond Relational](#). *modern-sql.com*, 2015. Archived at [perma.cc/D63V-WAPN](#)

Martin Fowler. [OrmHate](#). *martinfowler.com*, May 2012. Archived at [perma.cc/VCM8-PKNG](#)

Vlad Mihalcea. [N+1 query problem with JPA and Hibernate](#). *vladmihalcea.com*, January 2023. Archived at [perma.cc/79EV-TZKB](#)

Jens Schauder. [This is the Beginning of the End of the N+1 Problem: Introducing Single Query Loading](#). *spring.io*, August 2023. Archived at [perma.cc/6V96-R333](#)

William Zola. [6 Rules of Thumb for MongoDB Schema Design](#). *mongodb.com*, June 2014. Archived at [perma.cc/T2BZ-PPJB](#)

Sidney Andrews and Christopher McClister. [Data modeling in Azure Cosmos DB](#). *learn.microsoft.com*, February 2023. Archived at [archive.org](#)

Raffi Krikorian. [Timelines at Scale](#). At *QCon San Francisco*, November 2012. Archived at [perma.cc/V9G5-KLYK](#)

Ralph Kimball and Margy Ross. [The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling](#), 3rd edition. John Wiley & Sons, July 2013. ISBN: 9781118530801

Michael Kaminsky. [Data warehouse modeling: Star schema vs. OBT](#). *fivetran.com*, August 2022. Archived at perma.cc/2PZK-BFFP

Joe Nelson. [User-defined Order in SQL](#). *begriffs.com*, March 2018. Archived at perma.cc/GS3W-F7AD

Evan Wallace. [Realtime Editing of Ordered Sequences](#). *figma.com*, March 2017. Archived at perma.cc/K6ER-CQZW

David Greenspan. [Implementing Fractional Indexing](#). *observablehq.com*, October 2020. Archived at perma.cc/5N4R-MREN

Martin Fowler. [Schemaless Data Structures](#). *martinfowler.com*, January 2013.

Amr Awadallah. [Schema-on-Read vs. Schema-on-Write](#). At *Berkeley EECS RAD Lab Retreat*, Santa Cruz, CA, May 2009. Archived at perma.cc/DTB2-JCFR

Martin Odersky. [The Trouble with Types](#). At *Strange Loop*, September 2013. Archived at perma.cc/85QE-PVEP

Conrad Irwin. [MongoDB—Confessions of a PostgreSQL Lover](#). At *HTML5DevConf*, October 2013. Archived at perma.cc/C2J6-3AL5

[Percona Toolkit Documentation: pt-online-schema-change](#). *docs.percona.com*, 2023. Archived at perma.cc/9K8R-E5UH

Shlomi Noach. [gh-ost: GitHub's Online Schema Migration Tool for MySQL](#). *github.blog*, August 2016. Archived at perma.cc/7XAG-XB72

Shayon Mukherjee. [pg-osc: Zero downtime schema changes in PostgreSQL](#). *shayon.dev*, February 2022. Archived at perma.cc/35WN-7WMY

Carlos Pérez-Aradros Herce. [Introducing pgroll: zero-downtime, reversible, schema migrations for Postgres](#). [xata.io](#), October 2023. Archived at [archive.org](#)

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. [Spanner: Google's Globally-Distributed Database](#). At *10th USENIX Symposium on Operating System Design and Implementation* (OSDI), October 2012.

Donald K. Burleson. [Reduce I/O with Oracle Cluster Tables](#). [dba-oracle.com](#). Archived at [perma.cc/7LBJ-9X2C](#)

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. [Bigtable: A Distributed Storage System for Structured Data](#). At *7th USENIX Symposium on Operating System Design and Implementation* (OSDI), November 2006.

Priscilla Walmsley. [XQuery, 2nd Edition](#). O'Reilly Media, December 2015. ISBN: 9781491915080

Paul C. Bryan, Kris Zyp, and Mark Nottingham. [JavaScript Object Notation \(JSON\) Pointer](#). RFC 6901, IETF, April 2013.

Stefan Gössner, Glyn Normington, and Carsten Bormann. [JSONPath: Query Expressions for JSON](#). RFC 9535, IETF, February 2024.

Michael Stonebraker and Andrew Pavlo. [What Goes Around Comes Around... And Around.... ACM SIGMOD Record](#), volume 53, issue 2, pages 21–37.
[doi:10.1145/3685980.3685984](#)

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. [The PageRank Citation Ranking: Bringing Order to the Web](#). Technical Report 1999-66, Stanford University InfoLab, November 1999. Archived at perma.cc/UML9-UZHW

Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. [TAO: Facebook's Distributed Data Store for the Social Graph](#). At *USENIX Annual Technical Conference (ATC)*, June 2013.

Natasha Noy, Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, and Jamie Taylor. [Industry-Scale Knowledge Graphs: Lessons and Challenges](#). *Communications of the ACM*, volume 62, issue 8, pages 36–43, August 2019. [doi:10.1145/3331166](https://doi.org/10.1145/3331166)

Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoglu. [KÙZU Graph Database Management System](#). At *3th Annual Conference on Innovative Data Systems Research (CIDR 2023)*, January 2023.

Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, Torsten Hoefler. [Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries](#). arxiv.org, October 2019.

[Apache TinkerPop 3.6.3 Documentation](#). tinkerpop.apache.org, May 2023. Archived at perma.cc/KM7W-7PAT

Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. [Cypher: An Evolving Query Language for Property Graphs](#). At *International Conference on Management of Data (SIGMOD)*, pages 1433–1445, May 2018. [doi:10.1145/3183713.3190657](https://doi.org/10.1145/3183713.3190657)

Emil Eifrem. [Twitter correspondence](#), January 2014. Archived at [perma.cc/WM4S-BW64](#)

Francesco Tisiot. [Explore the new SEARCH and CYCLE features in PostgreSQL® 14](#). [aiven.io](#), December 2021. Archived at [perma.cc/J6BT-83UZ](#)

Gaurav Goel. [Understanding Hierarchies in Oracle](#). [towardsdatascience.com](#), May 2020. Archived at [perma.cc/5ZLR-Q7EW](#)

Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. [Graph Pattern Matching in GQL and SQL/PGQ](#). At *International Conference on Management of Data* (SIGMOD), pages 2246–2258, June 2022.
[doi:10.1145/3514221.3526057](#)

Alastair Green. [SQL... and now GQL](#). [openCypher.org](#), September 2019. Archived at [perma.cc/AFB2-3SY7](#)

Alin Deutsch, Yu Xu, and Mingxi Wu. [Seamless Syntactic and Semantic Integration of Query Primitives over Relational and Graph Data in GSQL](#). [tigergraph.com](#), November 2018. Archived at [perma.cc/JG7J-Y35X](#)

Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. [PGQL: a property graph query language](#). At *4th International Workshop on Graph Data Management Experiences and Systems* (GRADES), June 2016.
[doi:10.1145/2960414.2960421](#)

Amazon Web Services. [Neptune Graph Data Model](#). Amazon Neptune User Guide, [docs.aws.amazon.com](#). Archived at [perma.cc/CX3T-EZU9](#)

Cognitect. [Datomic Data Model](#). Datomic Cloud Documentation, [docs.datomic.com](#). Archived at [perma.cc/LGM9-LEUT](#)

David Beckett and Tim Berners-Lee. [Turtle – Terse RDF Triple Language](#). W3C Team Submission, March 2011.

Sinclair Target. [Whatever Happened to the Semantic Web?](#) *twobithistory.org*, May 2018. Archived at [perma.cc/M8GL-9KHS](#)

Gavin Mendel-Gleason. [The Semantic Web is Dead – Long Live the Semantic Web!](#) *terminusdb.com*, August 2022. Archived at [perma.cc/G2MZ-DSS3](#)

Manu Sporny. [JSON-LD and Why I Hate the Semantic Web](#). *manu.sporny.org*, January 2014. Archived at [perma.cc/7PT4-PJKF](#)

University of Michigan Library. [Biomedical Ontologies and Controlled Vocabularies](#), *guides.lib.umich.edu/ontology*. Archived at [perma.cc/Q5GA-F2N8](#)

Facebook. [The Open Graph protocol](#), *ogp.me*. Archived at [perma.cc/C49A-GUSY](#)

Matt Haughey. [Everything you ever wanted to know about unfurling but were afraid to ask /or/ How to make your site previews look amazing in Slack](#). *medium.com*, November 2015. Archived at [perma.cc/C7S8-4PZN](#)

W3C RDF Working Group. [Resource Description Framework \(RDF\)](#). *w3.org*, February 2004.

Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. [SPARQL 1.1 Query Language](#). W3C Recommendation, March 2013.

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

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, issue 1, pages 146–166, March 1989. [doi:10.1109/69.43410](https://doi.org/10.1109/69.43410)

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN: 9780201537710, available online at webdam.inria.fr/Alice

Scott Meyer, Andrew Carter, and Andrew Rodriguez. [LiQuid: The soul of a new graph database, Part 2](#). *engineering.linkedin.com*, September 2020. Archived at perma.cc/K9M4-PD6Q

Matt Bessey. [Why, after 6 years, I'm over GraphQL](#). *bessey.dev*, May 2024. Archived at perma.cc/2PAU-JYRA

Dominic Betts, Julián Domínguez, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. *Exploring CQRS and Event Sourcing*. Microsoft Patterns & Practices, July 2012. ISBN: 1621140164, archived at perma.cc/7A39-3NM8

Greg Young. [CQRS and Event Sourcing](#). At *Code on the Beach*, August 2014.

Greg Young. [CQRS Documents](#). *cqrs.wordpress.com*, November 2010. Archived at perma.cc/X5R6-R47F

Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. [Towards Scalable Dataframe Systems](#). *Proceedings of the VLDB Endowment*, volume 13, issue 11, pages 2033–2046. [doi:10.14778/3407790.3407807](https://doi.org/10.14778/3407790.3407807)

Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. [The TileDB Array Data Storage Manager](#). *Proceedings of the VLDB Endowment*, volume 10, issue 4, pages 349–360, November 2016. [doi:10.14778/3025111.3025117](https://doi.org/10.14778/3025111.3025117)

Florin Rusu. [Multidimensional Array Data Management](#). *Foundations and Trends in Databases*, volume 12, numbers 2–3, pages 69–220, February 2023.
[doi:10.1561/1900000069](https://doi.org/10.1561/1900000069)

Ed Targett. [Bloomberg, Man Group team up to develop open source “ArcticDB” database](#). *thestack.technology*, March 2023. Archived at perma.cc/M5YD-QQYV

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](https://doi.org/10.1093/nar/gkm929)

Chapter 4. Storage and Retrieval

One of the miseries of life is that everybody names things a little bit wrong. And so it makes everything a little harder to understand in the world than it would be if it were named differently. A computer does not primarily compute in the sense of doing arithmetic. [...] They primarily are filing systems.

—[Richard Feynman](#), *Idiosyncratic Thinking*
seminar (1985)

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. The GitHub repo for this book is <https://github.com/ept/ddia2-feedback>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out on GitHub.

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 3](#) we discussed data models and query languages—i.e., the format in which you give the database your data, and the interface through which you can ask for it again later. In this chapter we discuss the same from the database’s point of view: how the database can store the data that you give it, and how it can find the data again when you ask for it.

Why should you, as an application developer, care how the database handles storage and retrieval internally? You’re probably not going to implement your own storage engine from scratch, but you *do* need to select a storage engine that is appropriate for your application, from the many that are available. In order to configure 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 (OLTP) and those that are optimized for analytics (we introduced this distinction in [“Analytical versus Operational Systems”](#)). This chapter starts by examining two families of storage engines for

OLTP: *log-structured* storage engines that write out immutable data files, and storage engines such as *B-trees* that update data in-place. These structures are used for both key-value storage as well as secondary indexes.

Later in “[Data Storage for Analytics](#)” we’ll discuss a family of storage engines that is optimized for analytics, and in “[Multidimensional and Full-Text Indexes](#)” we’ll briefly look at indexes for more advanced queries, such as text retrieval.

Storage and Indexing for OLTP

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 -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 12 '{"name": "London", "attractions": ["Big Ben", "Trafalgar Square", "Westminster Abbey"]}'  
$ db_set 42 '{"name": "San Francisco", "attractions": ["Golden Gate Bridge", "Fisherman's Wharf", "Alcatraz Island"]}'  
$ db_get 42  
{ "name": "San Francisco", "attractions": ["Golden Gate Bridge", "Fisherman's Wharf", "Alcatraz Island"]}
```

The 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. If you update a key several times, 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", "Golden Gate Bridge", "Fisherman's Wharf"]}'  
$ db_get 42  
{ "name": "San Francisco", "attractions": ["Exploratorium", "Golden Gate Bridge", "Fisherman's Wharf"] }  
  
$ cat database  
12, {"name": "London", "attractions": ["Big Ben", "London Eye", "Buckingham Palace"]}  
42, {"name": "San Francisco", "attractions": ["Exploratorium", "Golden Gate Bridge", "Fisherman's Wharf"]}  
42, {"name": "San Francisco", "attractions": ["Exploratorium", "Golden Gate Bridge", "Fisherman's Wharf"]}
```

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

NOTE

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 on disk. It doesn't have to be human-readable; it might be binary and intended only for internal use by the database system.

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

In order to efficiently find the value for a particular key in the database, we need a different data structure: an *index*. In this chapter we will look at a range of indexing structures and see how they compare; the general idea is to structure the data in a particular way (e.g., sorted by some key) that makes it faster to locate the data you want. If you want to search the same data in several different ways, you may need several different indexes on different parts of the data.

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

This is an important trade-off in storage systems: well-chosen indexes speed up read queries, but every index consumes additional disk space and slows down writes, sometimes substantially [1]. For this reason, databases don't usually index everything by default, but require you—the person writing the application or administering the database—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 on writes than necessary.

Log-Structured Storage

To start, let's assume that you want to continue storing data in the append-only file written by `db_set`, and you just want to

speed up reads. One way you could do this is by keeping a hash map in memory, in which every key is mapped to the byte offset in the file at which the most recent value for that key can be found, as illustrated in [Figure 4-1](#).

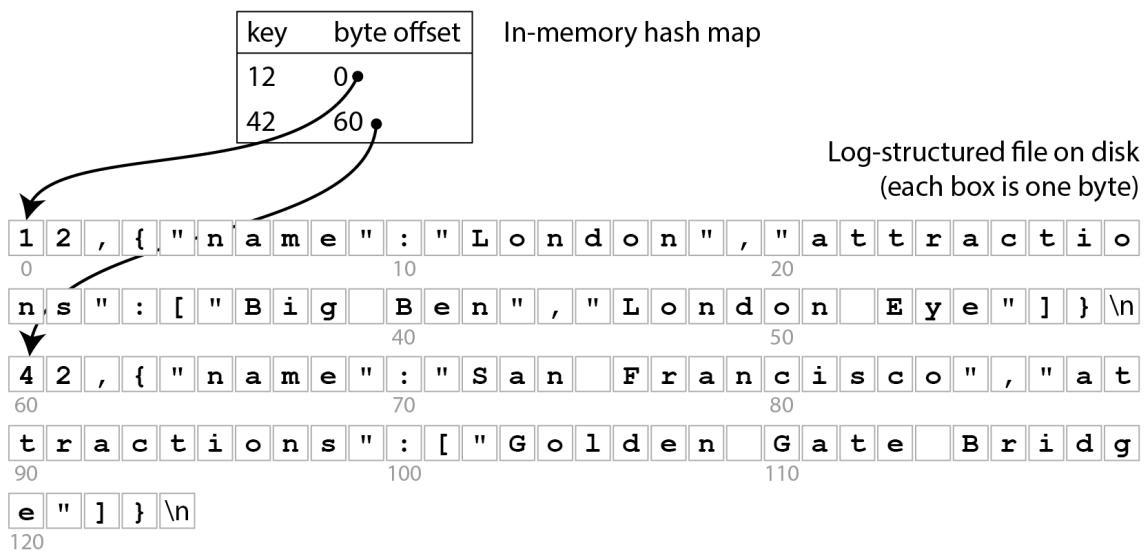


Figure 4-1. Storing a log of key-value pairs in a CSV-like format, indexed with an in-memory hash map.

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. When you want to look up a value, you use the hash map to find the offset in the log file, seek to that location, and read the value. If that part of the data file is already in the filesystem cache, a read doesn't require any disk I/O at all.

This approach is much faster, but it still suffers from several problems:

- You never free up disk space occupied by old log entries that have been overwritten; if you keep writing to the database you might run out of disk space.
- The hash map is not persisted, so you have to rebuild it when you restart the database—for example, by scanning the whole log file to find the latest byte offset for each key. This makes restarts slow if you have a lot of data.
- The hash table must fit in memory. In principle, you could maintain a hash table 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 [2].
- Range queries are not efficient. For example, you cannot easily scan over all keys between 10000 and 19999—you’d have to look up each key individually in the hash map.

The SSTable file format

In practice, hash tables are not used very often for database indexes, and instead it is much more common to keep data in a structure that is *sorted by key* [3]. One example of such a structure is a *Sorted String Table*, or *SSTable* for short, as shown in [Figure 4-2](#). This file format also stores key-value pairs, but it ensures that they are sorted by key, and each key only appears once in the file.

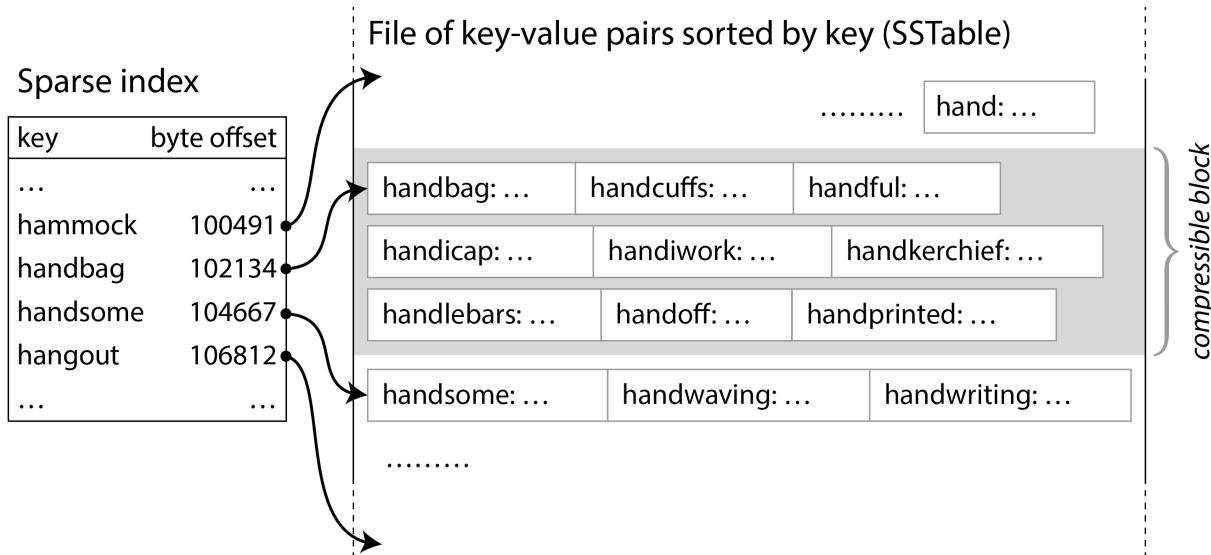


Figure 4-2. An SSTable with a sparse index, allowing queries to jump to the right block.

Now you do not need to keep all the keys in memory: you can group the key-value pairs within an SSTable into *blocks* of a few kilobytes, and then store the first key of each block in the index. This kind of index, which stores only some of the keys, is called *sparse*. This index is stored in a separate part of the SSTable, for example using an immutable B-tree, a trie, or another data structure that allows queries to quickly look up a particular key [4].

For example, in [Figure 4-2](#), the first key of one block is `handbag`, and the first key of the next block is `handsome`. Now say you're looking for the key `handiwork`, which doesn't appear in the sparse index. Because of the sorting you know that `handiwork` must appear between `handbag` and

`handsome`. This means you can seek to the offset for `handbag` and scan the file from there until you find `handiwork` (or not, if the key is not present in the file). A block of a few kilobytes can be scanned very quickly.

Moreover, each block of records can be compressed (indicated by the shaded area in [Figure 4-2](#)). Besides saving disk space, compression also reduces the I/O bandwidth use, at the cost of using a bit more CPU time.

Constructing and merging SSTables

The SSTable file format is better for reading than an append-only log, but it makes writes more difficult. We can't simply append at the end, because then the file would no longer be sorted (unless the keys happen to be written in ascending order). If we had to rewrite the whole SSTable every time a key is inserted somewhere in the middle, writes would become far too expensive.

We can solve this problem with a *log-structured* approach, which is a hybrid between an append-only log and a sorted file:

1. When a write comes in, add it to an in-memory ordered map data structure, such as a red-black tree, skip list [5], or trie [6]. With these data structures, you can insert keys in any

order, look them up efficiently, and read them back in sorted order. This in-memory data structure is called the *memtable*.

2. When the memtable gets bigger than some threshold—typically a few megabytes—write it out to disk in sorted order as an SSTable file. We call this new SSTable file the most recent *segment* of the database, and it is stored as a separate file alongside the older segments. Each segment has a separate index of its contents. While the new segment is being written out to disk, the database can continue writing to a new memtable instance, and the old memtable's memory is freed when the writing of the SSTable is complete.
3. In order to read the value for some key, first try to find the key in the memtable and the most recent on-disk segment. If it's not there, look in the next-older segment, etc. until you either find the key or reach the oldest segment. If the key does not appear in any of the segments, it does not exist in the database.
4. From time to time, run a merging and compaction process in the background to combine segment files and to discard overwritten or deleted values.

Merging segments works similarly to the *mergesort* algorithm [5]. The process is illustrated in [Figure 4-3](#): 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. If the same key appears in more than one input file, keep only the more recent value. This produces a new merged segment file, also sorted by key, with one value per key, and it uses minimal memory because we can iterate over the SSTables one key at a time.



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

To ensure that the data in the memtable is not lost if the database crashes, the storage engine keeps a separate log on disk to which every write is immediately appended. This log is not sorted by key, but that doesn't matter, because its only

purpose is to restore the memtable after a crash. Every time the memtable has been written out to an SSTable, the corresponding part of the log can be discarded.

If you want to delete a key and its associated value, you have to append a special deletion record called a *tombstone* to the data file. When log segments are merged, the tombstone tells the merging process to discard any previous values for the deleted key. Once the tombstone is merged into the oldest segment, it can be dropped.

The algorithm described here is essentially what is used in RocksDB [7], Cassandra, Scylla, and HBase [8], all of which were inspired by Google's Bigtable paper [9] (which introduced the terms *SSTable* and *memtable*).

The algorithm was originally published in 1996 under the name *Log-Structured Merge-Tree* or *LSM-Tree* [10], building on earlier work on log-structured filesystems [11]. For this reason, storage engines that are based on the principle of merging and compacting sorted files are often called *LSM storage engines*.

In LSM storage engines, a segment file is written in one pass (either by writing out the memtable or by merging some existing segments), and thereafter it is immutable. The merging

and compaction of segments can be done in a background thread, and while it is going on, we can still continue to serve reads using the old segment files. When 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 be deleted.

The segment files don't necessarily have to be stored on local disk: they are also well suited for writing to object storage. SlateDB and Delta Lake [12]. take this approach, for example.

Having immutable segment files also simplifies crash recovery: if a crash happens while writing out the memtable or while merging segments, the database can just delete the unfinished SSTable and start afresh. The log that persists writes to the memtable could contain incomplete records if there was a crash halfway through writing a record, or if the disk was full; these are typically detected by including checksums in the log, and discarding corrupted or incomplete log entries. We will talk more about durability and crash recovery in [Chapter 8](#).

Bloom filters

With LSM storage it can be slow to read a key that was last updated a long time ago, or that does not exist, since the storage

engine needs to check several segment files. In order to speed up such reads, LSM storage engines often include a *Bloom filter* [13] in each segment, which provides a fast but approximate way of checking whether a particular key appears in a particular SSTable.

[Figure 4-4](#) shows an example of a Bloom filter containing two keys and 16 bits (in reality, it would contain more keys and more bits). For every key in the SSTable we compute a hash function, producing a set of numbers that are then interpreted as indexes into the array of bits [14]. We set the bits corresponding to those indexes to 1, and leave the rest as 0. For example, the key `handbag` hashes to the numbers (2, 9, 4), so we set the 2nd, 9th, and 4th bits to 1. The bitmap is then stored as part of the SSTable, along with the sparse index of keys. This takes a bit of extra space, but the Bloom filter is generally small compared to the rest of the SSTable.

Keys that exist in the SSTable:

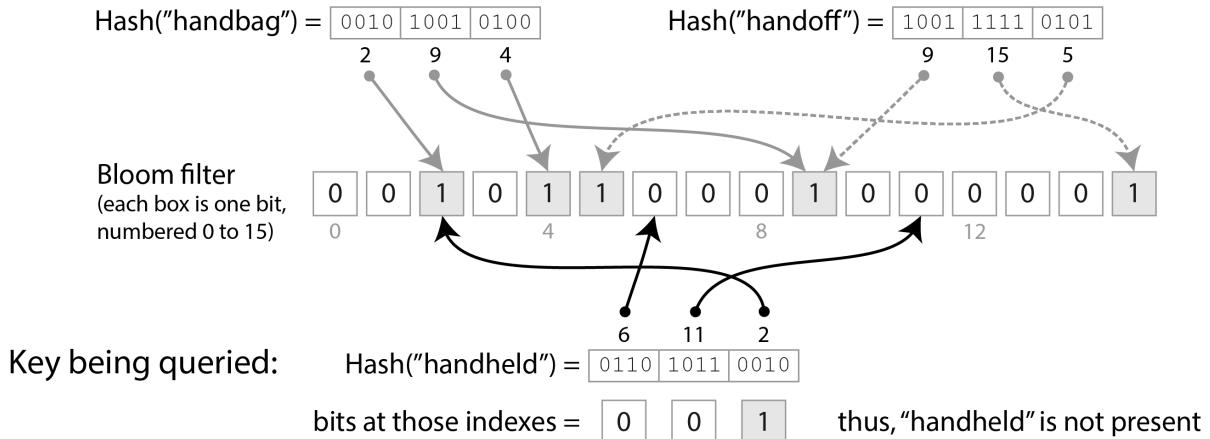


Figure 4-4. A Bloom filter provides a fast, probabilistic check whether a particular key exists in a particular SSTable.

When we want to know whether a key appears in the SSTable, we compute the same hash of that key as before, and check the bits at those indexes. For example, in [Figure 4-4](#), we're querying the key `handheld`, which hashes to (6, 11, 2). One of those bits is 1 (namely, bit number 2), while the other two are 0. These checks can be made extremely fast using the bitwise operations that all CPUs support.

If at least one of the bits is 0, we know that the key definitely does not appear in the SSTable. If the bits in the query are all 1, it's likely that the key is in the SSTable, but it's also possible that by coincidence all of those bits were set to 1 by other keys. This case when it looks as if a key is present, even though it isn't, is called a *false positive*.

The probability of false positives depends on the number of keys, the number of bits set per key, and the total number of bits in the Bloom filter. You can use an online calculator tool to work out the right parameters for your application [15]. As a rule of thumb, you need to allocate 10 bits of Bloom filter space for every key in the SSTable to get a false positive probability of 1%, and the probability is reduced tenfold for every 5 additional bits you allocate per key.

In the context of an LSM storage engines, false positives are no problem:

- If the Bloom filter says that a key *is not* present, we can safely skip that SSTable, since we can be sure that it doesn't contain the key.
- If the Bloom filter says the key *is* present, we have to consult the sparse index and decode the block of key-value pairs to check whether the key really is there. If it was a false positive, we have done a bit of unnecessary work, but otherwise no harm is done—we just continue the search with the next-oldest segment.

Compaction strategies

An important detail is how the LSM storage chooses when to perform compaction, and which SSTables to include in a compaction. Many LSM-based storage systems allow you to configure which compaction strategy to use, and some of the common choices are [16]:

Size-tiered compaction

Newer and smaller SSTables are successively merged into older and larger SSTables. The SSTables containing older data can get very large, and merging them requires a lot of temporary disk space. The advantage of this strategy is that it can handle very high write throughput.

Leveled compaction

The key range is split up into smaller SSTables and older data is moved into separate “levels,” which allows the compaction to proceed more incrementally and use less disk space than the size-tiered strategy. This strategy is more efficient for reads than size-tiered compaction because the storage engine needs to read fewer SSTables to check whether they contain the key.

As a rule of thumb, size-tiered compaction performs better if you have mostly writes and few reads, whereas leveled compaction performs better if your workload is dominated by reads. If you write a small number of keys frequently and a large number of keys rarely, then leveled compaction can also be advantageous [17].

Even though there are many subtleties, the basic idea of LSM-trees—keeping a cascade of SSTables that are merged in the background—is simple and effective. We discuss their performance characteristics in more detail in “[Comparing B-Trees and LSM-Trees](#)”.

EMBEDDED STORAGE ENGINES

Many databases run as a service that accepts queries over a network, but there are also *embedded* databases that don't expose a network API. Instead, they are libraries that run in the same process as your application code, typically reading and writing files on the local disk, and you interact with them through normal function calls. Examples of embedded storage engines include RocksDB, SQLite, LMDB, DuckDB, and KùzuDB [18].

Embedded databases are very commonly used in mobile apps to store the local user's data. On the backend, they can be an appropriate choice if the data is small enough to fit on a single machine, and if there are not many concurrent transactions. For example, in a multitenant system in which each tenant is small enough and completely separate from others (i.e., you do not need to run queries that combine data from multiple tenants), you can potentially use a separate embedded database instance per tenant [19].

The storage and retrieval methods we discuss in this chapter are used in both embedded and in client-server databases. In [Chapter 6](#) and [Chapter 7](#) we will discuss techniques for scaling a database across multiple machines.

B-Trees

The log-structured approach is popular, but it is not the only form of key-value storage. The most widely used structure for reading and writing database records by key is the *B-tree*.

Introduced in 1970 [20] and called “ubiquitous” less than 10 years later [21], B-trees have stood the test of time very well. They remain the standard index implementation in almost all relational databases, and many nonrelational databases use them too.

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

The log-structured indexes we saw earlier break the database down into variable-size *segments*, typically several megabytes or more in size, that are written once and are then immutable. By contrast, B-trees break the database down into fixed-size *blocks* or *pages*, and may overwrite a page in-place. A page is traditionally 4 KiB in size, but PostgreSQL now uses 8 KiB and MySQL uses 16 KiB by default.

Each page can be identified using a page number, which allows one page to refer to another—similar to a pointer, but on disk instead of in memory. If all the pages are stored in the same file, multiplying the page number by the page size gives us the byte offset in the file where the page is located. We can use these page references to construct a tree of pages, as illustrated in [Figure 4-5](#).

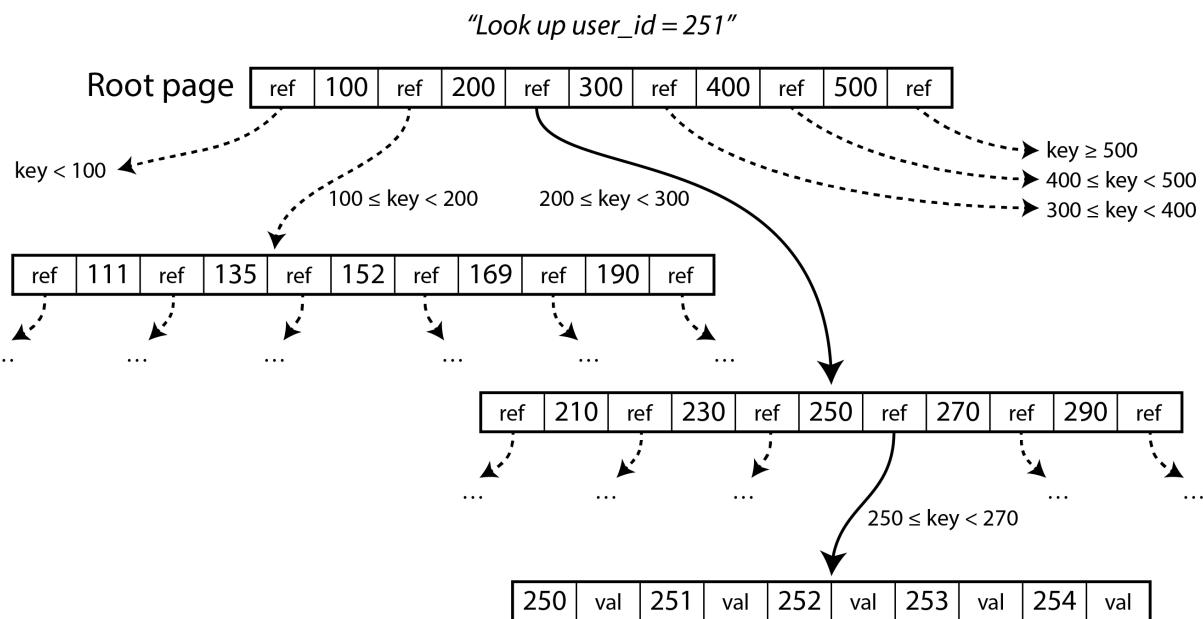


Figure 4-5. Looking up the key 251 using a B-tree index. From the root page we first follow the reference to the page for keys 200–300, then the page for keys 250–270.

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

between the references indicate where the boundaries between those ranges lie. (This structure is sometimes called a B^+ tree, but we don't need to distinguish it from other B-tree variants.)

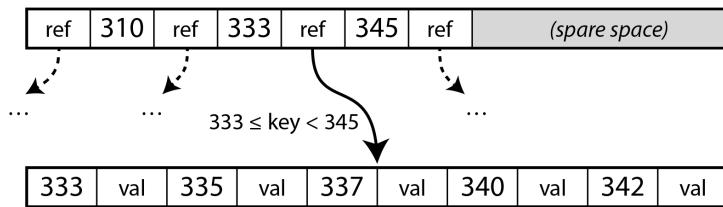
In the example in [Figure 4-5](#), we are looking for the key 251, so we know that we need to follow the page reference between the boundaries 200 and 300. That takes us to a similar-looking page that further breaks down the 200–300 range into subranges. Eventually we get down to a page containing individual keys (a *leaf page*), which either contains the value for each key inline or contains references to the pages where the values can be found.

The number of references to child pages in one page of the B-tree is called the *branching factor*. For example, in [Figure 4-5](#) the branching factor is six. In practice, the branching factor depends on the amount of space required to store the page references and the range boundaries, but typically it is several hundred.

If you want to update the value for an existing key in a B-tree, you search for the leaf page containing that key, and overwrite that page on disk with a version that contains the new value. 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, the page is split into two half-full pages, and the parent page is updated to account for the new subdivision of key ranges.

Before:



After adding key 334:

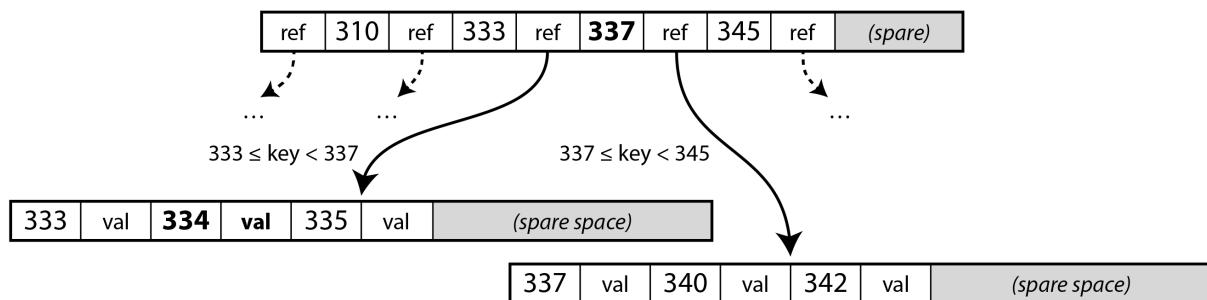


Figure 4-6. Growing a B-tree by splitting a page on the boundary key 337. The parent page is updated to reference both children.

In the example of [Figure 4-6](#), we want to insert the key 334, but the page for the range 333–345 is already full. We therefore split it into a page for the range 333–337 (including the new key), and a page for 337–344. We also have to update the parent page to have references to both children, with a boundary value of 337 between them. If the parent page doesn't have enough space for the new reference, it may also need to be split, and the

splits can continue all the way to the root of the tree. When the root is split, we make a new root above it. Deleting keys (which may require nodes to be merged) is more complex [5].

This algorithm ensures that the tree remains *balanced*: a B-tree with n keys always has a depth of $O(\log n)$. Most databases can fit into a B-tree that is three or four levels deep, so you don't need to follow many page references to find the page you are looking for. (A four-level tree of 4 KiB pages with a branching factor of 500 can store up to 250 TB.)

Making B-trees reliable

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

Overwriting several pages at once, like in a page split, is a dangerous operation: if the database crashes after only some of the pages have been written, you end up with a corrupted tree

(e.g., there may be an *orphan* page that is not a child of any parent).

In order to make the database resilient to crashes, it is common for B-tree implementations to include an additional data structure on disk: a *write-ahead log* (WAL). 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 [2, 22]. In filesystems, the equivalent mechanism is known as *journaling*.

To improve performance, B-tree implementations typically don't immediately write every modified page to disk, but buffer the B-tree pages in memory for a while first. The write-ahead log then also ensures that data is not lost in the case of a crash: as long as data has been written to the WAL, and flushed to disk using the `fsync()` system call, the data will be durable as the database will be able to recover it after a crash [23].

B-tree variants

As B-trees have been around for so long, many variants 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 [24]. A modified page is written to a different location, and a new version of the parent pages in the tree is created, pointing at the new location. This approach is also useful for concurrency control, as we shall see in [“Snapshot Isolation and Repeatable Read”](#).
- 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.
- To speed up scans over the key range in sorted order, some B-tree implementations try to lay out the tree so that leaf pages appear in sequential order on disk, reducing the number of disk seeks. However, it's difficult to maintain that order as the tree grows.
- 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.

Comparing B-Trees and LSM-Trees

As a rule of thumb, LSM-trees are better suited for write-heavy applications, whereas B-trees are faster for reads [25, 26].

However, benchmarks are often sensitive to details of the workload. You need to test systems with your particular workload in order to make a valid comparison. Moreover, it's not a strict either/or choice between LSM and B-trees: storage engines sometimes blend characteristics of both approaches, for example by having multiple B-trees and merging them LSM-style. In this section we will briefly discuss a few things that are worth considering when measuring the performance of a storage engine.

Read performance

In a B-tree, looking up a key involves reading one page at each level of the B-tree. Since the number of levels is usually quite small, this means that reads from a B-tree are generally fast and have predictable performance. In an LSM storage engine, reads often have to check several different SSTables at different stages of compaction, but Bloom filters help reduce the number of actual disk I/O operations required. Both approaches can perform well, and which is faster depends on the details of the storage engine and the workload.

Range queries are simple and fast on B-trees, as they can use the sorted structure of the tree. On LSM storage, range queries can also take advantage of the SSTable sorting, but they need to scan all the segments in parallel and combine the results. Bloom filters don't help for range queries (since you would need to compute the hash of every possible key within the range, which is impractical), making range queries more expensive than point queries in the LSM approach [27].

High write throughput can cause latency spikes in a log-structured storage engine if the memtable fills up. This happens if data can't be written out to disk fast enough, perhaps because the compaction process cannot keep up with incoming writes. Many storage engines, including RocksDB, perform *backpressure* in this situation: they suspend all reads and writes until the memtable has been written out to disk [28, 29].

Regarding read throughput, modern SSDs (and especially NVMe) can perform many independent read requests in parallel. Both LSM-trees and B-trees are able to provide high read throughput, but storage engines need to be carefully designed to take advantage of this parallelism [30].

Sequential vs. random writes

With a B-tree, if the application writes keys that are scattered all over the key space, the resulting disk operations are also scattered randomly, since the pages that the storage engine needs to overwrite could be located anywhere on disk. On the other hand, a log-structured storage engine writes entire segment files at a time (either writing out the memtable or while compacting existing segments), which are much bigger than a page in a B-tree.

The pattern of many small, scattered writes (as found in B-trees) is called *random writes*, while the pattern of fewer large writes (as found in LSM-trees) is called *sequential writes*. Disks generally have higher sequential write throughput than random write throughput, which means that a log-structured storage engine can generally handle higher write throughput on the same hardware than a B-tree. This difference is particularly big on spinning-disk hard drives (HDDs); on the solid state drives (SSDs) that most databases use today, the difference is smaller, but still noticeable (see [“Sequential vs. Random Writes on SSDs”](#)).

SEQUENTIAL VS. RANDOM WRITES ON SSDS

On spinning-disk hard drives (HDDs), sequential writes are much faster than random writes: a random write has to mechanically move the disk head to a new position and wait for the right part of the platter to pass underneath the disk head, which takes several milliseconds—an eternity in computing timescales. However, SSDs (solid-state drives) including NVMe (Non-Volatile Memory Express, i.e. flash memory attached to the PCI Express bus) have now overtaken HDDs for many use cases, and they are not subject to such mechanical limitations.

Nevertheless, SSDs also have higher throughput for sequential writes than for random writes. The reason is that flash memory can be read or written one page (typically 4 KiB) at a time, but it can only be erased one block (typically 512 KiB) at a time. Some of the pages in a block may contain valid data, whereas others may contain data that is no longer needed. Before erasing a block, the controller must first move pages containing valid data into other blocks; this process is called *garbage collection* (GC) [31].

A sequential write workload writes larger chunks of data at a time, so it is likely that a whole 512 KiB block belongs to a single file; when that file is later deleted again, the whole block can be erased without having to perform any GC. On the other hand,

with a random write workload, it is more likely that a block contains a mixture of pages with valid and invalid data, so the GC has to perform more work before a block can be erased [32, 33, 34].

The write bandwidth consumed by GC is then not available for the application. Moreover, the additional writes performed by GC contribute to wear on the flash memory; therefore, random writes wear out the drive faster than sequential writes.

Write amplification

With any type of storage engine, one write request from the application turns into multiple I/O operations on the underlying disk. With LSM-trees, a value is first written to the log for durability, then again when the memtable is written to disk, and again every time the key-value pair is part of a compaction. (If the values are significantly larger than the keys, this overhead can be reduced by storing values separately from keys, and performing compaction only on SSTables containing keys and references to values [35].)

A B-tree index must write every piece of data at least twice: once to the write-ahead log, and once to the tree page itself. In addition, they sometimes need to write out an entire page, even

if only a few bytes in that page changed, to ensure the B-tree can be correctly recovered after a crash or power failure [[36](#), [37](#)].

If you take the total number of bytes written to disk in some workload, and divide by the number of bytes you would have to write if you simply wrote an append-only log with no index, you get the *write amplification*. (Sometimes write amplification is defined in terms of I/O operations rather than bytes.) In write-heavy applications, the bottleneck might be the rate at which the database can write to disk. In this case, the higher the write amplification, the fewer writes per second it can handle within the available disk bandwidth.

Write amplification is a problem in both LSM-trees and B-trees. Which one is better depends on various factors, such as the length of your keys and values, and how often you overwrite existing keys versus insert new ones. For typical workloads, LSM-trees tend to have lower write amplification because they don't have to write entire pages and they can compress chunks of the SSTable [[38](#)]. This is another factor that makes LSM storage engines well suited for write-heavy workloads.

Besides affecting throughput, write amplification is also relevant for the wear on SSDs: a storage engine with lower

write amplification will wear out the SSD less quickly.

When measuring the write throughput of a storage engine, it is important to run the experiment for long enough that the effects of write amplification become clear. When writing to an empty LSM-tree, there are no compactions going on yet, so all of the disk bandwidth is available for new writes. As the database grows, new writes need to share the disk bandwidth with compaction.

Disk space usage

B-trees can become *fragmented* over time: for example, if a large number of keys are deleted, the database file may contain a lot of pages that are no longer used by the B-tree. Subsequent additions to the B-tree can use those free pages, but they can't easily be returned to the operating system because they are in the middle of the file, so they still take up space on the filesystem. Databases therefore need a background process that moves pages around to place them better, such as the vacuum process in PostgreSQL [23].

Fragmentation is less of a problem in LSM-trees, since the compaction process periodically rewrites the data files anyway, and SSTables don't have pages with unused space. Moreover,

blocks of key-value pairs can better be compressed in SSTables, and thus often produce smaller files on disk than B-trees. Keys and values that have been overwritten continue to consume space until they are removed by a compaction, but this overhead is quite low when using leveled compaction [38, 39]. Size-tiered compaction (see “[Compaction strategies](#)”) uses more disk space, especially temporarily during compaction.

Having multiple copies of some data on disk can also be a problem when you need to delete some data, and be confident that it really has been deleted (perhaps to comply with data protection regulations). For example, in most LSM storage engines a deleted record may still exist in the higher levels until the tombstone representing the deletion has been propagated through all of the compaction levels, which may take a long time. Specialist storage engine designs can propagate deletions faster [40].

On the other hand, the immutable nature of SSTable segment files is useful if you want to take a snapshot of a database at some point in time (e.g. for a backup or to create a copy of the database for testing): you can write out the memtable and record which segment files existed at that point in time. As long as you don’t delete the files that are part of the snapshot, you

don't need to actually copy them. In a B-tree whose pages are overwritten, taking such a snapshot efficiently is more difficult.

Multi-Column and Secondary Indexes

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, allowing you to search by columns other than the primary key. For example, in [Figure 3-1](#) in [Chapter 3](#) 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 in a secondary index, the indexed values are not necessarily unique; that is, there might be many rows (documents, vertices) under the same index

entry. This can be solved in two ways: either by making each value in the index a list of matching row identifiers (like a postings list in a full-text index) or by making each entry unique by appending a row identifier to it. Storage engines with in-place updates, like B-trees, and log-structured storage can both be used to implement an index.

Storing values within the index

The key in an index is the thing that queries search by, but the value can be one of several things:

- If the actual data (row, document, vertex) is stored directly within the index structure, it is called a *clustered index*. For example, in MySQL's InnoDB storage engine, the primary key of a table is always a clustered index, and in SQL Server, you can specify one clustered index per table.
- Alternatively, the value can be a reference to the actual data: either the primary key of the row in question (InnoDB does this for secondary indexes), or a direct reference to a location on disk. 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). For example, Postgres uses the heap file approach [41].

- A middle ground between the two is a *covering index* or *index with included columns*, which stores *some* of a table's columns within the index, in addition to storing the full row on the heap or in the primary key clustered index [42]. This allows some queries to be answered by using the index alone, without having to resolve the primary key or look in the heap file (in which case, the index is said to *cover* the query). This can make some queries faster, but the duplication of data means the index uses more disk space and slows down writes.

The indexes discussed so far only map a single key to a value. If you need to query multiple columns of a table (or multiple fields in a document) simultaneously, see ["Multidimensional and Full-Text Indexes"](#).

When updating a value without changing the key, the heap file approach can allow the record to 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 [2].

Keeping everything in memory

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

As RAM becomes cheaper, the cost-per-gigabyte argument is eroded. Many datasets are simply not that big, so it's quite feasible to keep them entirely in memory, 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), by writing a log of changes to disk, by writing periodic snapshots to disk, or by replicating the in-memory state to other machines.

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

Products such as VoltDB, SingleStore, and Oracle TimesTen are in-memory databases with a relational model, and the vendors claim that they can offer big performance improvements by removing all the overheads associated with managing on-disk data structures [43, 44]. 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) [45]. Redis and Couchbase provide weak durability by writing to disk asynchronously.

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

overheads of encoding in-memory data structures in a form that can be written to disk [46].

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

Data Storage for Analytics

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

On the surface, a data warehouse and a relational OLTP database look similar, because they both have a SQL query interface. However, the internals of the systems can look quite different, because they are optimized for very different query patterns. Many database vendors now focus on supporting

either transaction processing or analytics workloads, but not both.

Some databases, such as Microsoft SQL Server, SAP HANA, and SingleStore, have support for transaction processing and data warehousing in the same product. However, these hybrid transactional and analytical processing (HTAP) databases (introduced in [“Data Warehousing”](#)) are increasingly becoming two separate storage and query engines, which happen to be accessible through a common SQL interface [[47](#), [48](#), [49](#), [50](#)].

Cloud Data Warehouses

Data warehouse vendors such as Teradata, Vertica, and SAP HANA sell both on-premises warehouses under commercial licenses and cloud-based solutions. But as many of their customers move to the cloud, new cloud data warehouses such as Google Cloud BigQuery, Amazon Redshift, and Snowflake have also become widely adopted. Unlike traditional data warehouses, cloud data warehouses take advantage of scalable cloud infrastructure like object storage and serverless computation platforms.

Cloud data warehouses tend to integrate better with other cloud services and to be more elastic. For example, many cloud

warehouses support automatic log ingestion, and offer easy integration with data processing frameworks such as Google Cloud's Dataflow or Amazon Web Services' Kinesis. These warehouses are also more elastic because they decouple query computation from the storage layer [51]. Data is persisted on object storage rather than local disks, which makes it easy to adjust storage capacity and compute resources for queries independently, as we previously saw in ["Cloud-Native System Architecture"](#).

Open source data warehouses such as Apache Hive, Trino, and Apache Spark have also evolved with the cloud. As data storage for analytics has moved to data lakes on object storage, open source warehouses have begun to break apart [52]. The following components, which were previously integrated in a single system such as Apache Hive, are now often implemented as separate components:

Query engine

Query engines such as Trino, Apache DataFusion, and Presto parse SQL queries, optimize them into execution plans, and execute them against the data. Execution usually requires parallel, distributed data processing tasks. Some query engines provide built-in task execution,

while others choose to use third party execution frameworks such as Apache Spark or Apache Flink.

Storage format

The storage format determines how the rows of a table are encoded as bytes in a file, which is then typically stored in object storage or a distributed filesystem [12]. This data can then be accessed by the query engine, but also by other applications using the data lake. Examples of such storage formats are Parquet, ORC, Lance, or Nimble, and we will see more about them in the next section.

Table format

Files written in Apache Parquet and similar storage formats are typically immutable once written. To support row inserts and deletions, a table format such as Apache Iceberg or Databricks's Delta format are used. Table formats specify a file format that defines which files constitute a table along with the table's schema. Such formats also offer advanced features such as time travel (the ability to query a table as it was at a previous point in time), garbage collection, and even transactions.

Data catalog

Much like a table format defines which files make up a table, a data catalog defines which tables comprise a database. Catalogs are used to create, rename, and drop tables. Unlike storage and table formats, data catalogs such as Snowflake's Polaris and Databricks's Unity Catalog usually run as a standalone service that can be queried using a REST interface. Apache Iceberg also offers a catalog, which can be run inside a client or as a separate process. Query engines use catalog information when reading and writing tables. Traditionally, catalogs and query engines have been integrated, but decoupling them has enabled data discovery and data governance systems (discussed in "[Data Systems, Law, and Society](#)") to access a catalog's metadata as well.

Column-Oriented Storage

As discussed in "[Stars and Snowflakes: Schemas for Analytics](#)", data warehouses by convention often use a relational schema with a big fact table that contains foreign key references into dimension tables. 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 focus 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) [49].

Take the query in [Example 4-1](#): it accesses a large number of rows (every occurrence of someone buying fruit or candy during the 2024 calendar year), but it only needs to access three columns of the `fact_sales` table: `date_key`, `product_sk`, and `quantity`. The query ignores all other columns.

Example 4-1. 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 = 2024 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 4-1](#).

In order to process a query like [Example 4-1](#), you may have indexes on `fact_sales.date_key` and/or `fact_sales.product_sk` that tell the storage engine where to find all the sales for a particular date or for a particular product. But then, a row-oriented storage engine still needs to load all of those rows (each consisting of over 100 attributes) from disk into memory, parse them, and filter out those that don't meet the required conditions. That can take a long time.

The idea behind *column-oriented* (or *columnar*) storage is simple: don't store all the values from one row together, but store all the values from each *column* together instead [53]. If each column is stored separately, a query only needs to read and parse those columns that are used in that query, which can save a lot of work. [Figure 4-7](#) shows this principle using an expanded version of the fact table from [Figure 3-5](#).

NOTE

Column storage is easiest to understand in a relational data model, but it applies equally to nonrelational data. For example, Parquet [54] is a columnar storage format that supports a document data model, based on Google's Dremel [55], using a technique known as *shredding* or *striping* [56].

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
240102	69	4	NULL	NULL	1	13.99	13.99
240102	69	5	19	NULL	3	14.99	9.99
240102	69	5	NULL	191	1	14.99	14.99
240102	74	3	23	202	5	0.99	0.89
240103	30	2	NULL	NULL	1	2.49	2.49
240103	30	3	NULL	NULL	3	14.99	9.99
240103	30	3	21	123	1	49.99	39.99
240103	30	8	NULL	233	1	0.99	0.99

Columnar storage layout

date_key: 240102, 240102, 240102, 240102, 240103, 240103, 240103, 240103
product_sk: 69, 69, 69, 74, 30, 30, 30, 30
store_sk: 4, 5, 5, 3, 2, 3, 3, 8
promotion_sk: NULL, 19, NULL, 23, NULL, NULL, 21, NULL
customer_sk: NULL, NULL, 191, 202, NULL, NULL, 123, 233
quantity: 1, 3, 1, 5, 1, 3, 1, 1
net_price: 13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99
discount_price: 13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99

Figure 4-7. Storing relational data by column, rather than by row.

The column-oriented storage layout relies on each column storing 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 columns and put them together to form the 23rd row of the table.

In fact, columnar storage engines don't actually store an entire column (containing perhaps trillions of rows) in one go.

Instead, they break the table into blocks of thousands or millions of rows, and within each block they store the values from each column separately [57]. Since many queries are restricted to a particular date range, it is common to make each block contain the rows for a particular timestamp range. A query then only needs to load the columns it needs in those blocks that overlap with the required date range.

Columnar storage is used in almost all analytic databases nowadays [57], ranging from large-scale cloud data warehouses such as Snowflake [58] to single-node embedded databases such as DuckDB [59], and product analytics systems such as Pinot [60] and Druid [61]. It is used in storage formats such as Parquet, ORC [62, 63], Lance [64], and Nimble [65], and in-memory analytics formats like Apache Arrow [62, 66] and Pandas/NumPy [67]. Some time-series databases, such as InfluxDB IOx [68] and TimescaleDB [69], are also based on column-oriented storage.

Column Compression

Besides only loading those columns from disk that are required for a query, we can further reduce the demands on disk

throughput and network bandwidth by compressing data.

Fortunately, column-oriented storage often lends itself very well to compression.

Take a look at the sequences of values for each column in

[Figure 4-7](#): they often look quite repetitive, which is a good sign for compression. Depending on the data in the column, different compression techniques can be used. One technique that is particularly effective in data warehouses is *bitmap encoding*, illustrated in [Figure 4-8](#).

Column values:

product_sk: 

Bitmap for each possible value:

product_sk = 29: 

product_sk = 30: 

product_sk = 31: 

product_sk = 68: 

product_sk = 69: 

product_sk = 74: 

Run-length encoding:

product_sk = 29: 9, 1 (9 zeros, 1 one, rest zeros)

product_sk = 30: 5, 4, 3, 3 (5 zeros, 4 ones, 3 zeros, 3 ones, rest zeros)

product_sk = 31: 10, 2 (10 zeros, 2 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 4-8. 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.

One option is to store those bitmaps using one bit per row. However, these bitmaps typically contain a lot of zeros (we say that they are *sparse*). In that case, the bitmaps can additionally be run-length encoded: counting the number of consecutive zeros or ones and storing that number, as shown at the bottom of [Figure 4-8](#). Techniques such as *roaring bitmaps* switch between the two bitmap representations, using whichever is the most compact [70]. This can make the encoding of a column remarkably efficient.

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

```
WHERE product_sk IN (31, 68, 69) :
```

Load the three bitmaps for `product_sk = 31`,
`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 = 30 AND store_sk = 3;
```

Load the bitmaps for `product_sk = 30` 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.

Bitmaps can also be used to answer graph queries, such as finding all users of a social network who are followed by user X and who also follow user Y [71]. There are also various other compression schemes for columnar databases, which you can find in the references [72].

NOTE

Don't confuse column-oriented databases with the *wide-column* (also known as *column-family*) data model, in which a row can have thousands of columns, and there is no need for all the rows to have the same columns [9]. Despite the similarity in name, wide-column databases are row-oriented, since they store all values from a row together. Google's Bigtable, Apache Accumulo, and HBase are examples of the wide-column model.

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 columns. 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 can scan only the rows from the last month, which will be much faster than scanning all rows.

A second column can determine the sort order of any rows that have the same value in the first column. For example, if

`date_key` is the first sort key in [Figure 4-7](#), it might make sense for `product_sk` to be the second sort key so that all sales for the same product on the same day are grouped together in storage. That will help queries that need to group or filter sales by product within a certain date range.

Another advantage of sorted order is that it can help with compression of columns. If the primary sort column does not have many distinct values, then after sorting, it will have long sequences where the same value is repeated many times in a row. A simple run-length encoding, like we used for the bitmaps in [Figure 4-8](#), could compress that column down to a few kilobytes—even if the table has billions of rows.

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

Writing to Column-Oriented Storage

We saw in [“Characterizing Transaction Processing and Analytics”](#) that reads in data warehouses tend to consist of aggregations over a large number of rows; column-oriented storage, compression, and sorting all help to make those read queries faster. Writes in a data warehouse tend to be a bulk import of data, often via an ETL process.

With columnar storage, writing an individual row somewhere in the middle of a sorted table would be very inefficient, as you would have to rewrite all the compressed columns from the insertion position onwards. However, a bulk write of many rows at once amortizes the cost of rewriting those columns, making it efficient.

A log-structured approach is often used to perform writes in batches. All writes first go to a row-oriented, sorted, in-memory store. When enough writes have accumulated, they are merged with the column-encoded files on disk and written to new files in bulk. As old files remain immutable, and new files are written in one go, object storage is well suited for storing these files.

Queries need to examine both the column data on disk and the recent writes in memory, and combine the two. The query execution engine 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. Snowflake, Vertica, Apache Pinot, Apache Druid, and many others do this [58, 60, 61, 73].

Query Execution: Compilation and Vectorization

A complex SQL query for analytics is broken down into a *query plan* consisting of multiple stages, called *operators*, which may be distributed across multiple machines for parallel execution. Query planners can perform a lot of optimizations by choosing which operators to use, in which order to perform them, and where to run each operator.

Within each operator, the query engine needs to do various things with the values in a column, such as finding all the rows where the value is among a particular set of values (perhaps as part of a join), or checking whether the value is greater than 15. It also needs to look at several columns for the same row, for

example to find all sales transactions where the product is bananas and the store is a particular store of interest.

For data warehouse queries that need to scan over millions of rows, we need to worry not only about the amount of data they need to read off disk, but also the CPU time required to execute complex operators. The simplest kind of operator is like an interpreter for a programming language: while iterating over each row, it checks a data structure representing the query to find out which comparisons or calculations it needs to perform on which columns. Unfortunately, this is too slow for many analytics purposes. Two alternative approaches for efficient query execution have emerged [74]:

Query compilation

The query engine takes the SQL query and generates code for executing it. The code iterates over the rows one by one, looks at the values in the columns of interest, performs whatever comparisons or calculations are needed, and copies the necessary values to an output buffer if the required conditions are satisfied. The query engine compiles the generated code to machine code (often using an existing compiler such as LLVM), and then runs it on the column-encoded data that has been loaded into memory. This approach to code generation is similar

to the just-in-time (JIT) compilation approach that is used in the Java Virtual Machine (JVM) and similar runtimes.

Vectorized processing

The query is interpreted, not compiled, but it is made fast by processing many values from a column in a batch, instead of iterating over rows one by one. A fixed set of predefined operators are built into the database; we can pass arguments to them and get back a batch of results [47, 72].

For example, we could pass the `product_sk` column and the ID of “bananas” to an equality operator, and get back a bitmap (one bit per value in the input column, which is 1 if it’s a banana); we could then pass the `store_sk` column and the ID of the store of interest to the same equality operator, and get back another bitmap; and then we could pass the two bitmaps to a “bitwise AND” operator, as shown in [Figure 4-9](#). The result would be a bitmap containing a 1 for all sales of bananas in a particular store.

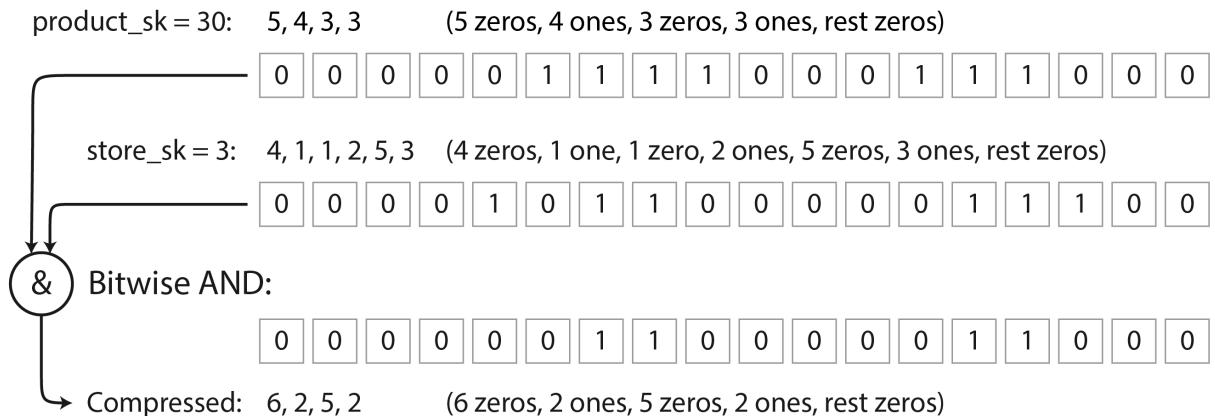


Figure 4-9. A bitwise AND between two bitmaps lends itself to vectorization.

The two approaches are very different in terms of their implementation, but both are used in practice [74]. Both can achieve very good performance by taking advantages of the characteristics of modern CPUs:

- preferring sequential memory access over random access to reduce cache misses [75],
- doing most of the work in tight inner loops (that is, with a small number of instructions and no function calls) to keep the CPU instruction processing pipeline busy and avoid branch mispredictions,
- making use of parallelism such as multiple threads and single-instruction-multi-data (SIMD) instructions [76, 77], and
- operating directly on compressed data without decoding it into a separate in-memory representation, which saves memory allocation and copying costs.

Materialized Views and Data Cubes

We previously encountered *materialized views* in “[Materializing and Updating Timelines](#)”: in a relational data model, they are 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 accordingly. Some databases can do that automatically, and there are also systems such as Materialize that specialize in materialized view maintenance [[78](#)].

Performing such updates means more work on writes, but materialized views can improve read performance in workloads that repeatedly need to perform the same queries.

Materialized aggregates are a type of materialized views that can be useful in data warehouses. As discussed earlier, data warehouse queries often involve an aggregate function, such as `COUNT`, `SUM`, `AVG`, `MIN`, or `MAX` in SQL. If the same aggregates are used by many different queries, it can be wasteful to crunch through the raw data every time. Why not

cache some of the counts or sums that queries use most often? A *data cube* or *OLAP cube* does this by creating a grid of aggregates grouped by different dimensions [79]. [Figure 4-10](#) shows an example.

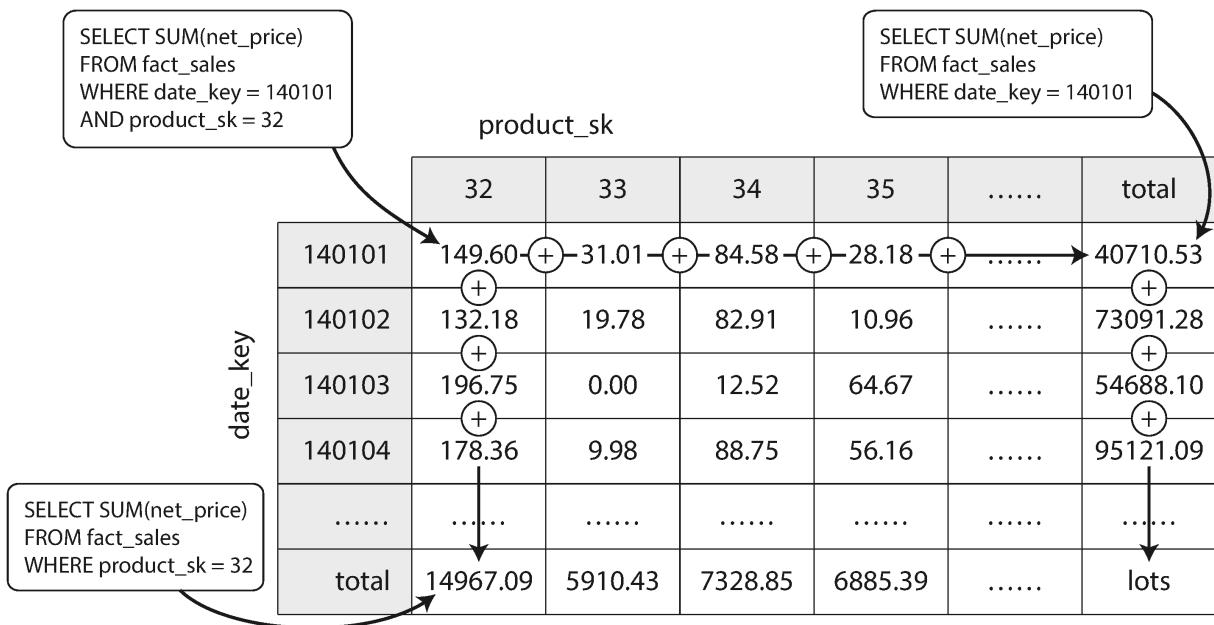


Figure 4-10. Two dimensions of a data cube, aggregating data by summing.

Imagine for now that each fact has foreign keys to only two dimension tables—in [Figure 4-10](#), these are `date_key` and `product_sk`. 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 3-5](#) there are five dimensions: date, product, store, promotion, and customer. It's a lot harder to imagine what a five-dimensional hypercube would look like, but the principle remains the same: each cell contains the sales for a particular date-product-store-promotion-customer combination. These values can then repeatedly be summarized along each of the dimensions.

The advantage of a materialized data cube is that certain queries become very fast because they have effectively been precomputed. For example, if you want to know the total sales per store yesterday, you just need to look at the totals along the appropriate dimension—no need to scan millions of rows.

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

Multidimensional and Full-Text Indexes

The B-trees and LSM-trees we saw in the first half of this chapter allow range queries over a single attribute: for example, if the key is a username, you can use them as an index to efficiently find all names starting with an L. But sometimes, searching by a single attribute is not enough.

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.

On the other hand, *multi-dimensional indexes* allow you to query several columns at once. One case where this is particularly important is 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 > 51.49  
AND longitude > -0.11
```

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

One option is to translate a two-dimensional location into a single number using a space-filling curve, and then to use a regular B-tree index [80]. More commonly, specialized spatial indexes such as R-trees or Bkd-trees [81] are used; they divide up the space so that nearby data points tend to be grouped in the same subtree. For example, PostGIS implements geospatial indexes as R-trees using PostgreSQL's Generalized Search Tree

indexing facility [82]. It is also possible to use regularly spaced grids of triangles, squares, or hexagons [83].

Multi-dimensional indexes are not just for geographic locations. For example, on an ecommerce website you could use a three-dimensional index on the dimensions (*red, green, blue*) to search for products in a certain range of colors, or in a database of weather observations you could have a two-dimensional index on (*date, temperature*) in order to efficiently search for all the observations during the year 2013 where the temperature was between 25 and 30°C. 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 [84].

Full-Text Search

Full-text search allows you to search a collection of text documents (web pages, product descriptions, etc.) by keywords that might appear anywhere in the text [85]. Information retrieval is a big, specialist topic that often involves language-specific processing: for example, several Asian languages are written without spaces or punctuation between words, and therefore splitting text into words requires a model that

indicates which character sequences constitute a word. Full-text search also often involves matching words that are similar but not identical (such as typos or different grammatical forms of words) and synonyms. Those problems go beyond the scope of this book.

However, at its core, you can think of full-text search as another kind of multidimensional query: in this case, each word that might appear in a text (a *term*) is a dimension. A document that contains term x has a value of 1 in dimension x , and a document that doesn't contain x has a value of 0. Searching for documents mentioning "red apples" means a query that looks for a 1 in the *red* dimension, and simultaneously a 1 in the *apples* dimension. The number of dimensions may thus be very large.

The data structure that many search engines use to answer such queries is called an *inverted index*. This is a key-value structure where the key is a term, and the value is the list of IDs of all the documents that contain the term (the *postings list*). If the document IDs are sequential numbers, the postings list can also be represented as a sparse bitmap, like in [Figure 4-8](#): the n th bit in the bitmap for term x is a 1 if the document with ID n contains the term x [\[86\]](#).

Finding all the documents that contain both terms x and y is now similar to a vectorized data warehouse query that searches for rows matching two conditions ([Figure 4-9](#)): load the two bitmaps for terms x and y and compute their bitwise AND. Even if the bitmaps are run-length encoded, this can be done very efficiently.

For example, Lucene, the full-text indexing engine used by Elasticsearch and Solr, works like this [87]. It stores the mapping from term to postings list in SSTable-like sorted files, which are merged in the background using the same log-structured approach we saw earlier in this chapter [88]. PostgreSQL's GIN index type also uses postings lists to support full-text search and indexing inside JSON documents [89, 90].

Instead of breaking text into words, an alternative is to find all the substrings of length n , which are called n -grams. For example, the trigrams ($n = 3$) of the string "hello" are "hel", "ell", and "llo". If we build an inverted index of all trigrams, we can search the documents for arbitrary substrings that are at least three characters long. Trigram indexes even allows regular expressions in search queries; the downside is that they are quite large [91].

To cope with typos in documents or queries, Lucene is able to search text for words within a certain edit distance (an edit distance of 1 means that one letter has been added, removed, or replaced) [92]. It does this by storing the set of terms as a finite state automaton over the characters in the keys, similar to a *trie* [93], and transforming it into a *Levenshtein automaton*, which supports efficient search for words within a given edit distance [94].

Vector Embeddings

Semantic search goes beyond synonyms and typos to try and understand document concepts and user intentions. For example, if your help pages contain a page titled “cancelling your subscription”, users should still be able to find that page when searching for “how to close my account” or “terminate contract”, which are close in terms of meaning even though they use completely different words.

To understand a document’s semantics—its meaning—semantic search indexes use embedding models to translate a document into a vector of floating-point values, called a *vector embedding*. The vector represents a point in a multi-dimensional space, and each floating-point value represents the document’s location along one dimension’s axis. Embedding models generate vector

embeddings that are near each other (in this multi-dimensional space) when the embedding's input documents are semantically similar.

NOTE

We saw the term *vectorized processing* in “[Query Execution: Compilation and Vectorization](#)”. Vectors in semantic search have a different meaning. In vectorized processing, the vector refers to a batch of bits that can be processed with specially optimized code. In embedding models, vectors are a list of floating point numbers that represent a location in multi-dimensional space.

For example, a three-dimensional vector embedding for a Wikipedia page about agriculture might be [0.1, 0.22, 0.11]. A Wikipedia page about vegetables would be quite near, perhaps with an embedding of [0.13, 0.19, 0.24]. A page about star schemas might have an embedding of [0.82, 0.39, -0.74], comparatively far away. We can tell by looking that the first two vectors are closer than the third.

Embedding models use much larger vectors (often over 1,000 numbers), but the principles are the same. We don't try to understand what the individual numbers mean; they're simply a way for embedding models to point to a location in an abstract multi-dimensional space. Search engines use distance functions such as cosine similarity or Euclidean distance to

measure the distance between vectors. Cosine similarity measures the cosine of the angle of two vectors to determine how close they are, while Euclidean distance measures the straight-line distance between two points in space.

Many early embedding models such as Word2Vec [95], BERT [96], and GPT [97] worked with text data. Such models are usually implemented as neural networks. Researchers went on to create embedding models for video, audio, and images as well. More recently, model architecture has become *multimodal*: a single model can generate vector embeddings for multiple modalities such as text and images.

Semantic search engines use an embedding model to generate a vector embedding when a user enters a query. The user's query and related context (such as a user's location) are fed into the embedding model. After the embedding model generates the query's vector embedding, the search engine must find documents with similar vector embeddings using a vector index.

Vector indexes store the vector embeddings of a collection of documents. To query the index, you pass in the vector embedding of the query, and the index returns the documents whose vectors are closest to the query vector. Since the R-trees

we saw previously don't work well for vectors with many dimensions, specialized vector indexes are used, such as:

Flat indexes

Vectors are stored in the index as they are. A query must read every vector and measure its distance to the query vector. Flat indexes are accurate, but measuring the distance between the query and each vector is slow.

Inverted file (IVF) indexes

The vector space is clustered into partitions (called *centroids*) of vectors to reduce the number of vectors that must be compared. IVF indexes are faster than flat indexes, but can give only approximate results: the query and a document may fall into different partitions, even though they are close to each other. A query on an IVF index first defines *probes*, which are simply the number of partitions to check. Queries that use more probes will be more accurate, but will be slower, as more vectors must be compared.

Hierarchical Navigable Small World (HNSW)

HNSW indexes maintain multiple layers of the vector space, as illustrated in [Figure 4-11](#). Each layer is

represented as a graph, where nodes represent vectors, and edges represent proximity to nearby vectors. A query starts by locating the nearest vector in the topmost layer, which has a small number of nodes. The query then moves to the same node in the layer below and follows the edges in that layer, which is more densely connected, looking for a vector that is closer to the query vector. The process continues until the last layer is reached. As with IVF indexes, HNSW indexes are approximate.

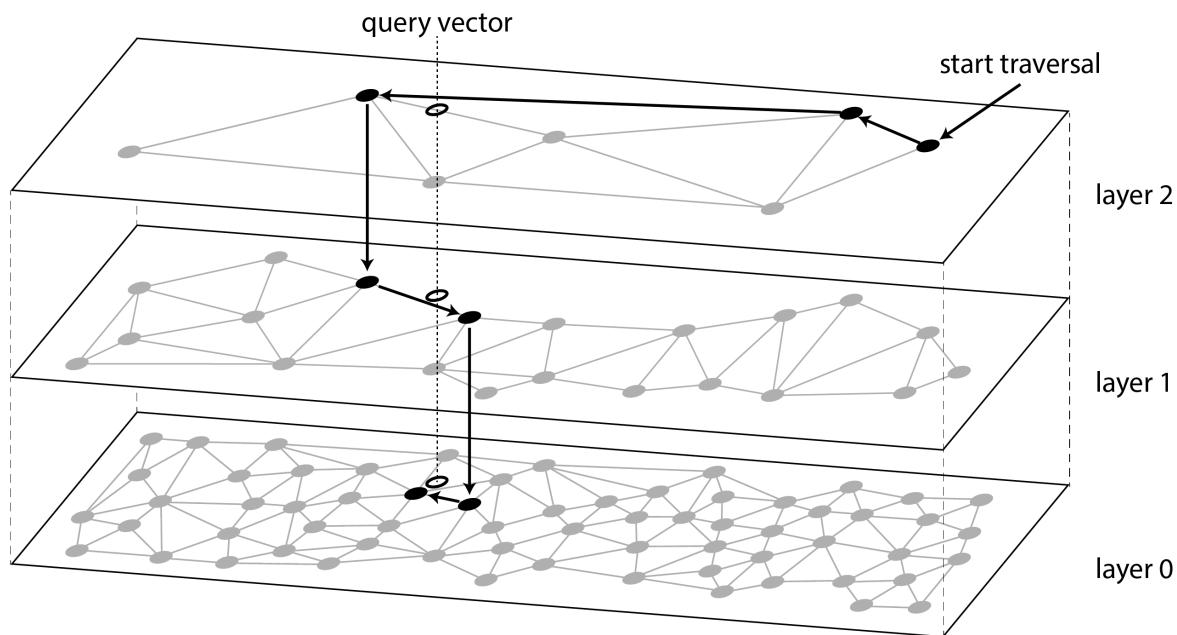


Figure 4-11. Searching for the database entry that is closest to a given query vector in a HNSW index.

Many popular vector databases implement IVF and HNSW indexes. Facebook's Faiss library has many variations of each

[[98](#)], and PostgreSQL's pgvector supports both as well [[99](#)]. The full details of the IVF and HNSW algorithms are beyond the scope of this book, but their papers are an excellent resource [[100](#), [101](#)].

Summary

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

[“Analytical versus Operational Systems”](#) introduced the distinction between transaction processing (OLTP) and analytics (OLAP). In this chapter we saw that storage engines optimized for OLTP look very different from those optimized for analytics:

- OLTP systems are optimized for a high volume of requests, each of which reads and writes a small number of records, and which need fast responses. The records are typically accessed via a primary key or a secondary index, and these indexes are typically ordered mappings from key to record, which also support range queries.

- Data warehouses and similar analytic systems are optimized for complex read queries that scan over a large number of records. They generally use a column-oriented storage layout with compression that minimizes the amount of data that such a query needs to read off disk, and just-in-time compilation of queries or vectorization to minimize the amount of CPU time spent processing the data.

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

- The log-structured approach, which only permits appending to files and deleting obsolete files, but never updates a file that has been written. SSTables, LSM-trees, RocksDB, Cassandra, HBase, Scylla, Lucene, and others belong to this group. In general, log-structured storage engines tend to provide high write throughput.
- The update-in-place approach, which treats the disk as a set of fixed-size pages that can be overwritten. B-trees, the biggest example of this philosophy, are used in all major relational OLTP databases and also many nonrelational ones. As a rule of thumb, B-trees tend to be better for reads, providing higher read throughput and lower response times than log-structured storage.

We then looked at indexes that can search for multiple conditions at the same time: multidimensional indexes such as R-trees that can search for points on a map by latitude and longitude at the same time, and full-text search indexes that can search for multiple keywords appearing in the same text. Finally, vector databases are used for semantic search on text documents and other media; they use vectors with a larger number of dimensions and find similar documents by comparing vector similarity.

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.

FOOTNOTES

REFERENCES

Nikolay Samokhvalov. [How partial, covering, and multicolumn indexes may slow down UPDATEs in PostgreSQL](#). *postgres.ai*, October 2021. Archived at [perma.cc/PBK3-F4G9](#)

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

Evan Jones. [Why databases use ordered indexes but programming uses hash tables](#). *evanjones.ca*, December 2019. Archived at [perma.cc/NJX8-3ZZD](#)

Branimir Lambov. [CEP-25: Trie-indexed SSTable format](#). *cwiki.apache.org*, November 2022. Archived at [perma.cc/HD7W-PW8U](#). Linked Google Doc archived at [perma.cc/UL6C-AAAE](#)

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

Branimir Lambov. [Trie Memtables in Cassandra](#). *Proceedings of the VLDB Endowment*, volume 15, issue 12, pages 3359–3371, August 2022. [doi:10.14778/3554821.3554828](#)

Dhruba Borthakur. [The History of RocksDB](#). *rocksdb.blogspot.com*, November 2013. Archived at [perma.cc/Z7C5-JPSP](#)

Matteo Bertozzi. [Apache HBase I/O – HFile](#). *blog.cloudera.com*, June 2012. Archived at [perma.cc/U9XH-L2KL](#)

Sey Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. [Bigtable: A](#)

[Distributed Storage System for Structured Data](#). At *7th USENIX Symposium on Operating System Design and Implementation* (OSDI), November 2006.

Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. [The Log-Structured Merge-Tree \(LSM-Tree\)](#). *Acta Informatica*, volume 33, issue 4, pages 351–385, June 1996. [doi:10.1007/s002360050048](https://doi.org/10.1007/s002360050048)

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

Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał Świtakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. [Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores](#).

Proceedings of the VLDB Endowment, volume 13, issue 12, pages 3411–3424, August 2020. [doi:10.14778/3415478.3415560](https://doi.org/10.14778/3415478.3415560)

Burton H. Bloom. [Space/Time Trade-offs in Hash Coding with Allowable Errors](#). *Communications of the ACM*, volume 13, issue 7, pages 422–426, July 1970. [doi:10.1145/362686.362692](https://doi.org/10.1145/362686.362692)

Adam Kirsch and Michael Mitzenmacher. [Less Hashing, Same Performance: Building a Better Bloom Filter](#). *Random Structures & Algorithms*, volume 33, issue 2, pages 187–218, September 2008. [doi:10.1002/rsa.20208](https://doi.org/10.1002/rsa.20208)

Thomas Hurst. [Bloom Filter Calculator](#). *hur:st*, September 2023. Archived at perma.cc/L3AV-6VC2

Chen Luo and Michael J. Carey. [LSM-based storage techniques: a survey](#). *The VLDB Journal*, volume 29, pages 393–418, July 2019. [doi:10.1007/s00778-019-00555-y](https://doi.org/10.1007/s00778-019-00555-y)

Mark Callaghan. [Name that compaction algorithm](#). *smalldatum.blogspot.com*, August 2018. Archived at [perma.cc/CN4M-82DY](#)

Prashanth Rao. [Embedded databases \(1\): The harmony of DuckDB, KùzuDB and LanceDB](#). *thedataquarry.com*, August 2023. Archived at [perma.cc/PA28-2R35](#)

Hacker News discussion. [Bluesky migrates to single-tenant SQLite](#). *news.ycombinator.com*, October 2023. Archived at [perma.cc/69LM-5P6X](#)

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.
[doi:10.1145/1734663.1734671](#)

Douglas Comer. [The Ubiquitous B-Tree](#). *ACM Computing Surveys*, volume 11, issue 2, pages 121–137, June 1979. [doi:10.1145/356770.356776](#)

C. Mohan and Frank Levine. [ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging](#). At *ACM International Conference on Management of Data* (SIGMOD), June 1992. [doi:10.1145/130283.130338](#)

Hironobu Suzuki. [The Internals of PostgreSQL](#). *interdb.jp*, 2017.

Howard Chu. [LDAP at Lightning Speed](#). At *Build Stuff '14*, November 2014. Archived at [perma.cc/GB6Z-P8YH](#)

Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. [Designing Access Methods: The RUM Conjecture](#). At *19th International Conference on Extending Database Technology* (EDBT), March 2016. [doi:10.5441/002/edbt.2016.42](#)

Ben Stopford. [Log Structured Merge Trees](#). *benstopford.com*, February 2015. Archived at [perma.cc/E5BV-KUJ6](#)

Mark Callaghan. [The Advantages of an LSM vs a B-Tree](#). *smalldatum.blogspot.co.uk*, January 2016. Archived at [perma.cc/3TYZ-EFUD](#)

Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. [SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores](#). At *USENIX Annual Technical Conference*, July 2019.

Igor Canadi, Siying Dong, Mark Callaghan, et al. [RocksDB Tuning Guide](#). *github.com*, 2023. Archived at [perma.cc/UNY4-MK6C](#)

Gabriel Haas and Viktor Leis. [What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines](#). *Proceedings of the VLDB Endowment*, volume 16, issue 9, pages 2090-2102.
[doi:10.14778/3598581.3598584](#)

Emmanuel Goossaert. [Coding for SSDs](#). *codecapsule.com*, February 2014.

Jack Vanlightly. [Is sequential IO dead in the era of the NVMe drive?](#) *jack-vanlightly.com*, May 2023. Archived at [perma.cc/7TMZ-TAPU](#)

Alibaba Cloud Storage Team. [Storage System Design Analysis: Factors Affecting NVMe SSD Performance \(2\)](#). *alibabacloud.com*, January 2019. Archived at [archive.org](#)

Xiao-Yu Hu and Robert Haas. [The Fundamental Limit of Flash Random Write Performance: Understanding, Analysis and Performance Modelling](#). *dominoweb.draco.res.ibm.com*, March 2010. Archived at [perma.cc/8JUL-4ZDS](#)

Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. [WiscKey: Separating Keys from Values in SSD-conscious Storage](#). At *4th USENIX Conference on File and Storage Technologies (FAST)*, February 2016.

Peter Zaitsev. [Innodb Double Write](#). *percona.com*, August 2006. Archived at [perma.cc/NT4S-DK7T](#)

Tomas Vondra. [On the Impact of Full-Page Writes](#). *2ndquadrant.com*, November 2016. Archived at [perma.cc/7N6B-CVL3](#)

Mark Callaghan. [Read, write & space amplification - B-Tree vs LSM](#). *smalldatum.blogspot.com*, November 2015. Archived at [perma.cc/S487-WK5P](#)

Mark Callaghan. [Choosing Between Efficiency and Performance with RocksDB](#). At *Code Mesh*, November 2016. Video at youtube.com/watch?v=tgzkgZVXKB4

Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. [Enabling Timely and Persistent Deletion in LSM-Engines](#). *ACM Transactions on Database Systems*, volume 48, issue 3, article no. 8, August 2023. [doi:10.1145/3599724](https://doi.org/10.1145/3599724)

Drew Silcock. [How Postgres stores data on disk – this one's a page turner](#). *drew.silcock.dev*, August 2024. Archived at [perma.cc/8K7K-7VJ2](#)

Joe Webb. [Using Covering Indexes to Improve Query Performance](#). *simple-talk.com*, September 2008. Archived at [perma.cc/6MEZ-R5VR](#)

Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. [The End of an Architectural Era \(It's Time for a Complete Rewrite\)](#). At *33rd International Conference on Very Large Data Bases (VLDB)*, September 2007.

[VoltDB Technical Overview White Paper](#). VoltDB, 2017. Archived at [perma.cc/B9SF-SK5G](#)

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.

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](#)

Per-Åke Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. [Enhancements to SQL Server Column Stores](#). At *ACM International Conference on Management of Data* (SIGMOD), June 2013.
[doi:10.1145/2463676.2463708](#)

Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. [The SAP HANA Database – An Architecture Overview](#). *IEEE Data Engineering Bulletin*, volume 35, issue 1, pages 28–33, March 2012.

Michael Stonebraker. [The Traditional RDBMS Wisdom Is \(Almost Certainly\) All Wrong](#). Presentation at EPFL, May 2013.

Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, and Nikita Shamgunov. [Cloud-Native Transactions and Analytics in SingleStore](#). At *ACM International Conference on Management of Data* (SIGMOD), June 2022. [doi:10.1145/3514221.3526055](#)

Tino Tereshko and Jordan Tigani. [BigQuery under the hood](#). *cloud.google.com*, January 2016. Archived at [perma.cc/WP2Y-FUCF](#)

Wes McKinney. [The Road to Composable Data Systems: Thoughts on the Last 15 Years and the Future](#). *wesmckinney.com*, September 2023. Archived at [perma.cc/6L2M-GTJX](#)

Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O’Neil, Pat

O’Neil, Alex Rasin, Nga Tran, and Stan Zdonik. [C-Store: A Column-oriented DBMS](#). At *31st International Conference on Very Large Data Bases* (VLDB), pages 553–564, September 2005.

Julien Le Dem. [Dremel Made Simple with Parquet](#). *blog.twitter.com*, September 2013.

Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. [Dremel: Interactive Analysis of Web-Scale Datasets](#). At *36th International Conference on Very Large Data Bases* (VLDB), pages 330–339, September 2010. [doi:10.14778/1920841.1920886](https://doi.org/10.14778/1920841.1920886)

Joe Kearney. [Understanding Record Shredding: storing nested data in columns](#). *joekearney.co.uk*, December 2016. Archived at perma.cc/ZD5N-AX5D

Jamie Brandon. [A shallow survey of OLAP and HTAP query engines](#). *scattered-thoughts.net*, September 2023. Archived at perma.cc/L3KH-J4JF

Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. [The Snowflake Elastic Data Warehouse](#). At *ACM International Conference on Management of Data* (SIGMOD), pages 215–226, June 2016. [doi:10.1145/2882903.2903741](https://doi.org/10.1145/2882903.2903741)

Mark Raasveldt and Hannes Mühleisen. [Data Management for Data Science Towards Embedded Analytics](#). At *10th Conference on Innovative Data Systems Research* (CIDR), January 2020.

Jean-François Im, Kishore Gopalakrishna, Subbu Subramaniam, Mayank Srivastava, Adwait Tumbde, Xiaotian Jiang, Jennifer Dai, Seunghyun Lee, Neha Pawar, Jialiang Li, and Ravi Aringunram. [Pinot: Realtime OLAP for 530 Million Users](#). At *ACM International Conference on Management of Data* (SIGMOD), pages 583–594, May 2018. [doi:10.1145/3183713.3190661](https://doi.org/10.1145/3183713.3190661)

Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. [Druid: A Real-time Analytical Data Store](#). At *ACM International Conference on Management of Data* (SIGMOD), June 2014. [doi:10.1145/2588555.2595631](https://doi.org/10.1145/2588555.2595631)

Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. [Deep Dive into Common Open Formats for Analytical DBMSs](#). *Proceedings of the VLDB Endowment*, volume 16, issue 11, pages 3044–3056, July 2023. [doi:10.14778/3611479.3611507](https://doi.org/10.14778/3611479.3611507)

Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. [An Empirical Evaluation of Columnar Storage Formats](#). *Proceedings of the VLDB Endowment*, volume 17, issue 2, pages 148–161. [doi:10.14778/3626292.3626298](https://doi.org/10.14778/3626292.3626298)

Weston Pace. [Lance v2: A columnar container format for modern data](#). blog.lancedb.com, April 2024. Archived at perma.cc/ZK3Q-S9VJ

Yoav Helfman. [Nimble, A New Columnar File Format](#). At *VeloxCon*, April 2024.

Wes McKinney. [Apache Arrow: High-Performance Columnar Data Framework](#). At *CMU Database Group – Vaccination Database Tech Talks*, December 2021.

Wes McKinney. [Python for Data Analysis, 3rd Edition](#). O'Reilly Media, August 2022.
ISBN: 9781098104023

Paul Dix. [The Design of InfluxDB IOx: An In-Memory Columnar Database Written in Rust with Apache Arrow](#). At *CMU Database Group – Vaccination Database Tech Talks*, May 2021.

Carlota Soto and Mike Freedman. [Building Columnar Compression for Large PostgreSQL Databases](#). timescale.com, March 2024. Archived at perma.cc/7KTF-V3EH

Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. [Consistently faster and smaller compressed bitmaps with Roaring](#). *Software: Practice and Experience*, volume 46, issue 11, pages 1547–1569, November 2016. [doi:10.1002/spe.2402](https://doi.org/10.1002/spe.2402)

Jaz Volpert. [An entire Social Network in 1.6GB \(GraphD Part 2\)](#). *jazco.dev*, April 2024.
Archived at perma.cc/L27Z-QVMG

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, issue 3, pages 197–280, December 2013. [doi:10.1561/1900000024](https://doi.org/10.1561/1900000024)

Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. [The Vertica Analytic Database: C-Store 7 Years Later](#). *Proceedings of the VLDB Endowment*, volume 5, issue 12, pages 1790–1801, August 2012. [doi:10.14778/2367502.2367518](https://doi.org/10.14778/2367502.2367518)

Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. [Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask](#). *Proceedings of the VLDB Endowment*, volume 11, issue 13, pages 2209–2222, September 2018. [doi:10.14778/3275366.3284966](https://doi.org/10.14778/3275366.3284966)

Forrest Smith. [Memory Bandwidth Napkin Math](#). *forrestthewoods.com*, February 2020. Archived at perma.cc/Y8U4-PS7N

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.

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](https://doi.org/10.1145/564691.564709)

Kevin Bartley. [OLTP Queries: Transfer Expensive Workloads to Materialize](#). *materialize.com*, August 2024. Archived at perma.cc/4TYM-TYD8

Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. [Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals](#). *Data Mining and Knowledge Discovery*, volume 1, issue 1, pages 29–53, March 2007.
[doi:10.1023/A:1009726021843](#)

Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. [Integrating the UB-Tree into a Database System Kernel](#). At *26th International Conference on Very Large Data Bases* (VLDB), September 2000.

Octavian Procopiuc, Pankaj K. Agarwal, Lars Arge, and Jeffrey Scott Vitter. [Bkd-Tree: A Dynamic Scalable kd-Tree](#). At *8th International Symposium on Spatial and Temporal Databases* (SSTD), pages 46–65, July 2003. [doi:10.1007/978-3-540-45072-6_4](#)

Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. [Generalized Search Trees for Database Systems](#). At *21st International Conference on Very Large Data Bases* (VLDB), September 1995.

Isaac Brodsky. [H3: Uber’s Hexagonal Hierarchical Spatial Index](#). *eng.uber.com*, June 2018. Archived at [archive.org](#)

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](#)

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. [Introduction to Information Retrieval](#). Cambridge University Press, 2008. ISBN: 978-0-521-86571-5, available online at [nlp.stanford.edu/IR-book](#)

Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. [An Experimental Study of Bitmap Compression vs. Inverted List Compression](#). At *ACM International Conference on Management of Data* (SIGMOD), pages 993–1008, May 2017. [doi:10.1145/3035918.3064007](#)

Adrien Grand. [What is in a Lucene Index?](#) At *Lucene/Solr Revolution*, November 2013.
Archived at [perma.cc/Z7QN-GBYY](#)

Michael McCandless. [Visualizing Lucene's Segment Merges](#). *blog.mikemccandless.com*, February 2011. Archived at [perma.cc/3ZV8-72W6](#)

Lukas Fittl. [Understanding Postgres GIN Indexes: The Good and the Bad](#). *pganalyze.com*, December 2021. Archived at [perma.cc/V3MW-26H6](#)

Jimmy Angelakos. [The State of \(Full\) Text Search in PostgreSQL 12](#). At *FOSDEM*, February 2020. Archived at [perma.cc/J6US-3WZS](#)

Alexander Korotkov. [Index support for regular expression search](#). At *PGConf.EU Prague*, October 2012. Archived at [perma.cc/5RFZ-ZKDQ](#)

Michael McCandless. [Lucene's FuzzyQuery Is 100 Times Faster in 4.0](#). *blog.mikemccandless.com*, March 2011. Archived at [perma.cc/E2WC-GHTW](#)

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, issue 2, pages 192–223, April 2002. [doi:10.1145/506309.506312](https://doi.org/10.1145/506309.506312)

Klaus U. Schulz and Stoyan Mihov. [Fast String Correction with Levenshtein Automata](#). *International Journal on Document Analysis and Recognition*, volume 5, issue 1, pages 67–85, November 2002. [doi:10.1007/s10032-002-0082-8](https://doi.org/10.1007/s10032-002-0082-8)

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. [Efficient Estimation of Word Representations in Vector Space](#). At *International Conference on Learning Representations* (ICLR), May 2013. [doi:10.48550/arXiv.1301.3781](https://doi.org/10.48550/arXiv.1301.3781)

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. [BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding](#). At *Conference of the North American Chapter of the Association for Computational*

Linguistics: Human Language Technologies, volume 1, pages 4171–4186, June 2019.
[doi:10.18653/v1/N19-1423](https://doi.org/10.18653/v1/N19-1423)

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. [Improving Language Understanding by Generative Pre-Training](#). *openai.com*, June 2018.
Archived at perma.cc/5N3C-DJ4C

Matthijs Douze, Maria Lomeli, and Lucas Hosseini. [Faiss indexes](#). *github.com*, August 2024. Archived at perma.cc/2EWG-FPBS

Varik Matevosyan. [Understanding pgvector's HNSW Index Storage in Postgres](#). *lantern.dev*, August 2024. Archived at perma.cc/B2YB-JB59

] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. [Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors](#). At *European Conference on Computer Vision (ECCV)*, pages 202–216, September 2018. [doi:10.1007/978-3-030-01258-8_13](https://doi.org/10.1007/978-3-030-01258-8_13)

] Yury A. Malkov and Dmitry A. Yashunin. [Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs](#). *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 42, issue 4, pages 824–836, April 2020. [doi:10.1109/TPAMI.2018.2889473](https://doi.org/10.1109/TPAMI.2018.2889473)

Chapter 5. Encoding and Evolution

Everything changes and nothing stands still.

—Heraclitus of Ephesus, as quoted by Plato in
Cratylus (360 BCE)

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. The GitHub repo for this book is <https://github.com/ept/ddia2-feedback>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out on GitHub.

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 2](#) 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”](#)).

In most cases, a change to an application’s features also requires a change to data that it stores: perhaps a new field or record type needs to be captured, or perhaps existing data needs to be presented in a new way.

The data models we discussed in [Chapter 3](#) 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”](#)).

When a data format or schema changes, a corresponding change to application code often needs to happen (for example, you add a new field to a record, and the application code starts reading and writing that field). However, in a large application, code changes often cannot happen instantaneously:

- With server-side applications you may want to perform a *rolling upgrade* (also known as a *staged rollout*), deploying the new version to a few nodes at a time, checking whether the new version is running smoothly, and gradually working your way through all the nodes. This allows new versions to be deployed without service downtime, and thus encourages more frequent releases and better evolvability.
- With client-side applications you're at the mercy of the user, who may not install the update for some time.

This means that old and new versions of the code, and old and new data formats, may potentially all coexist in the system at the same time. In order for the system to continue running smoothly, we need to maintain compatibility in both directions:

Backward compatibility

Newer code can read data that was written by older code.

Forward compatibility

Older code can read data that was written by newer code.

Backward compatibility is normally not hard to achieve: as author of the newer code, you know the format of data written by older code, and so you can explicitly handle it (if necessary by simply keeping the old code to read the old data). Forward

compatibility can be trickier, because it requires older code to ignore additions made by a newer version of the code.

Another challenge with forward compatibility is illustrated in [Figure 5-1](#). Say you add a field to a record schema, and the newer code creates a record containing that new field and stores it in a 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. But if the record is decoded into a model object that does not explicitly preserve unknown fields, data can be lost, like in [Figure 5-1](#).

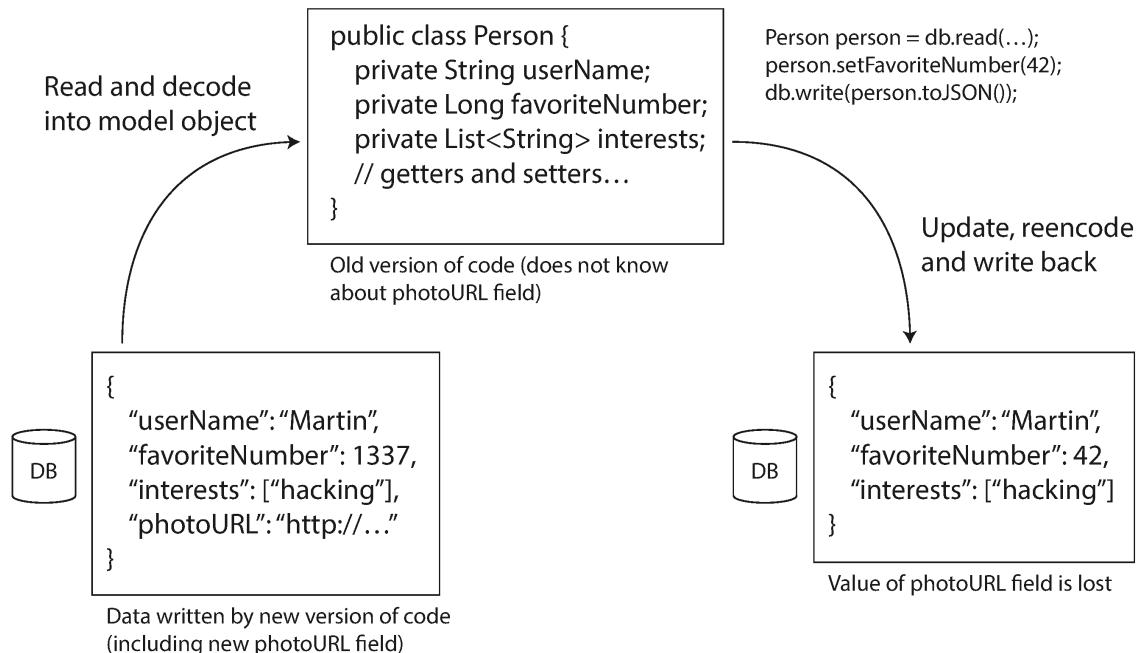


Figure 5-1. When an older version of the application updates data previously written by a newer version of the application, data may be lost if you're not careful.

In this chapter we will look at several formats for encoding data, including JSON, XML, Protocol Buffers, 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 databases, web services, REST APIs, remote procedure calls (RPC), workflow engines, and event-driven 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 often looks quite different from the data structures that are normally used in memory.

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*).

TERMINOLOGY CLASH

Serialization is unfortunately also used in the context of transactions (see [Chapter 8](#)), 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.

There are exceptions in which encoding/decoding is not needed—for example, when a database operates directly on compressed data loaded from disk, as discussed in [“Query Execution: Compilation and Vectorization”](#). There are also *zero-copy* data formats that are designed to be used both at runtime and on disk/on the network, without an explicit conversion step, such as Cap'n Proto and FlatBuffers.

However, most systems need to convert between in-memory objects and flat byte sequences. 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`, Python has `pickle`, Ruby has `Marshal`, and so on. Many third-party libraries also exist, such as Kryo for Java.

These encoding libraries are very convenient, because they allow in-memory objects to be saved and restored with minimal additional code. However, they also have a number of deep problems:

- The encoding is often tied to a particular programming language, and reading the data in another language is very difficult. If you store or transmit data in such an encoding, you are committing yourself to your current programming language for potentially a very long time, and precluding integrating your systems with those of other organizations (which may use different languages).
- In order to restore data in the same object types, the decoding process needs to be able to instantiate arbitrary

classes. This is frequently a source of security problems [1]: 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 [2, 3].

- 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 [4].
- 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 [5].

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

When moving to standardized encodings that can be written and read by many programming languages, JSON and XML are the obvious contenders. They are widely known, widely supported, and almost as widely disliked. XML is often criticized for being too verbose and unnecessarily complicated

[6]. JSON's popularity is mainly due to its built-in support in web browsers and simplicity relative to XML. CSV is another popular language-independent format, but it only supports tabular data without nesting.

JSON, XML, and CSV are textual formats, and thus somewhat human-readable (although the syntax is a popular topic of debate). Besides the superficial syntactic issues, they also have some subtle problems:

- There is a lot of ambiguity around the encoding of numbers. In XML and CSV, you cannot distinguish between a number and a string that happens to consist of digits (except by referring to an external schema). JSON distinguishes strings and numbers, but it doesn't distinguish integers and floating-point numbers, and it doesn't specify a precision. This is a problem when dealing with large numbers; for example, integers greater than 2^{53} cannot be exactly represented in an IEEE 754 double-precision floating-point number, so such numbers become inaccurate when parsed in a language that uses floating-point numbers, such as JavaScript [7]. An example of numbers larger than 2^{53} occurs on X (formerly Twitter), which uses a 64-bit number to identify each post. The JSON returned by the API includes post 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 [8].

- JSON and XML have good support for Unicode character strings (i.e., human-readable text), but they don't support binary strings (sequences of bytes without a character encoding). Binary strings are a useful feature, so people get around this limitation by encoding the binary data as text using Base64. The schema is then used to indicate that the value should be interpreted as Base64-encoded. This works, but it's somewhat hacky and increases the data size by 33%.
- XML Schema and JSON Schema are powerful, and thus quite complicated to learn and implement. Since the correct interpretation of data (such as numbers and binary strings) depends on information in the schema, applications that don't use XML/JSON schemas need to potentially hard-code the appropriate encoding/decoding logic instead.
- CSV does not have any schema, so it is up to the application to define the meaning of each row and column. If an application change adds a new row or column, you have to handle that change manually. CSV is also a quite vague format (what happens if a value contains a comma or a newline character?). Although its escaping rules have been formally specified [9], not all parsers implement them correctly.

Despite these flaws, JSON, XML, and CSV are good enough for many purposes. It's likely that they will remain popular, especially as data interchange formats (i.e., for sending data from one organization to another). In these situations, as long as people agree on what the format is, it often doesn't matter how pretty or efficient the format is. The difficulty of getting different organizations to agree on *anything* outweighs most other concerns.

JSON Schema

JSON Schema has become widely adopted as a way to model data whenever it's exchanged between systems or written to storage. You'll find JSON schemas in web services (see [“Web services”](#)) as part of the OpenAPI web service specification, schema registries such as Confluent's Schema Registry and Red Hat's Apicurio Registry, and in databases such as PostgreSQL's pg_jsonschema validator extension and MongoDB's `$jsonSchema` validator syntax.

The JSON Schema specification offers a number of features. Schemas include standard primitive types including strings, numbers, integers, objects, arrays, booleans, or nulls. But JSON Schema also offers a separate validation specification that allows developers to overlay constraints on fields. For example,

a `port` field might have a minimum of 1 and a maximum of 65535.

JSON Schemas can have either open or closed content models. An open content model permits any field not defined in the schema to exist with any data type, whereas a closed content model only allows fields that are explicitly defined. The open content model in JSON Schema is enabled when

`additionalProperties` is set to `true`, which is the default. Thus, JSON Schemas are usually a definition of what *isn't* permitted (namely, invalid values on any of the defined fields), rather than what *is* permitted in a schema.

Open content models are powerful, but can be complex. For example, say you want to define a map from integers (such as IDs) to strings. JSON does not have a map or dictionary type, only an “object” type that can contain string keys, and values of any type. You can then constrain this type with JSON Schema so that keys may only contain digits, and values can only be strings, using `patternProperties` and `additionalProperties` as shown in [Example 5-1](#).

Example 5-1. Example JSON Schema with integer keys and string values. Integer keys are represented as strings

containing only integers since JSON Schema requires all keys to be strings.

```
{  
  "$schema": "http://json-schema.org/draft-07/schema#",  
  "type": "object",  
  "patternProperties": {  
    "^[0-9]+$": {  
      "type": "string"  
    }  
  },  
  "additionalProperties": false  
}
```

In addition to open and closed content models and validators, JSON Schema supports conditional if/else schema logic, named types, references to remote schemas, and much more. All of this makes for a very powerful schema language. Such features also make for unwieldy definitions. It can be challenging to resolve remote schemas, reason about conditional rules, or evolve schemas in a forwards or backwards compatible way [10]. Similar concerns apply to XML Schema [11].

Binary encoding

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, CBOR, BSON, BJSON, UBJSON, BISON, Hessian, and Smile, to name a few) and for XML (WBXML and Fast Infoset, for example). These formats have been adopted in various niches, as they are more compact and sometimes faster to parse, but none of them are as widely adopted as the textual versions of JSON and XML [12].

Some of these formats extend the set of datatypes (e.g., distinguishing integers and floating-point numbers, or adding support for binary strings), but otherwise they keep the JSON/XML data model unchanged. In particular, since they don't prescribe a schema, they need to include all the object field names within the encoded data. That is, in a binary encoding of the JSON document in [Example 5-2](#), they will need to include the strings `userName`, `favoriteNumber`, and `interests` somewhere.

Example 5-2. 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 5-2](#) shows the byte sequence that you get if you encode the JSON document in [Example 5-2](#) with MessagePack. The first few bytes are as follows:

1. The first byte, `0x83`, indicates that what follows is an object (top four bits = `0x80`) with three fields (bottom four bits = `0x03`). (In case you're wondering what happens if an object has more than 15 fields, so that the number of fields doesn't fit in four bits, it then gets a different type indicator, and the number of fields is encoded in two or four bytes.)
2. The second byte, `0xa8`, indicates that what follows is a string (top four bits = `0xa0`) that is eight bytes long (bottom four bits = `0x08`).
3. The next eight bytes are the field name `userName` in ASCII. Since the length was indicated previously, there's no need for any marker to tell us where the string ends (or any escaping).

4. The next seven bytes encode the six-letter string value

Martin with a prefix 0xa6, and so on.

The binary encoding is 66 bytes long, which is only a little less than the 81 bytes taken by the textual JSON encoding (with whitespace removed). All the binary encodings of JSON are similar in this regard. It's not clear whether such a small space reduction (and perhaps a speedup in parsing) is worth the loss of human-readability.

In the following sections we will see how we can do much better, and encode the same record in just 32 bytes.

MessagePack

Byte sequence (66 bytes):

83	a8	75	73	65	72	4e	61	6d	65	a6	4d	61	72	74	69	6e	ae	66	61
76	6f	72	69	74	65	4e	75	6d	62	65	72	cd	05	39	a9	69	6e	74	65
72	65	73	74	73	92	ab	64	61	79	64	72	65	61	6d	69	6e	67	a7	68
61	63	6b	69	6e	67														

Breakdown:

object (3 entries)	string (length 8)	u s e r N a m e	string (length 6)	M a r t i n
83	a8	75 73 65 72 4e 61 6d 65	a6	4d 61 72 74 69 6e
	string (length 14)	f a v o r i t e N u m b e r		
	ae	66 61 76 6f 72 69 74 65 4e 75 6d 62 65 72		
	uint16	1337	string (length 9)	i n t e r e s t s
	cd	05 39	a9	69 6e 74 65 72 65 73 74 73
array (2 entries)	string (length 11)	d a y d r e a m i n g		
92	ab	64 61 79 64 72 65 61 6d 69 6e 67		
	string (length 7)	h a c k i n g		
	a7	68 61 63 6b 69 6e 67		

Figure 5-2. Example record ([Example 5-2](#)) encoded using MessagePack.

Protocol Buffers

Protocol Buffers (protobuf) is a binary encoding library developed at Google. It is similar to Apache Thrift, which was originally developed by Facebook [13]; most of what this section says about Protocol Buffers applies also to Thrift.

Protocol Buffers requires a schema for any data that is encoded. To encode the data in [Example 5-2](#) in Protocol Buffers, you would describe the schema in the Protocol Buffers interface definition language (IDL) like this:

```
syntax = "proto3";

message Person {
    string user_name = 1;
    int64 favorite_number = 2;
    repeated string interests = 3;
}
```

Protocol Buffers comes with a code generation tool that takes a schema definition like the one shown here, and produces classes that implement the schema in various programming languages. Your application code can call this generated code to encode or decode records of the schema. The schema language is very simple compared to JSON Schema: it only defines the fields of records and their types, but it does not support other restrictions on the possible values of fields.

Encoding [Example 5-2](#) using a Protocol Buffers encoder requires 33 bytes, as shown in [Figure 5-3 \[14\]](#).

Protocol Buffers

Byte sequence (33 bytes):

0a	06	4d	61	72	74	69	6e	10	b9	0a	1a	0b	64	61	79	64	72	65	61
6d	69	6e	67	1a	07	68	61	63	6b	69	6e	67							

Breakdown:

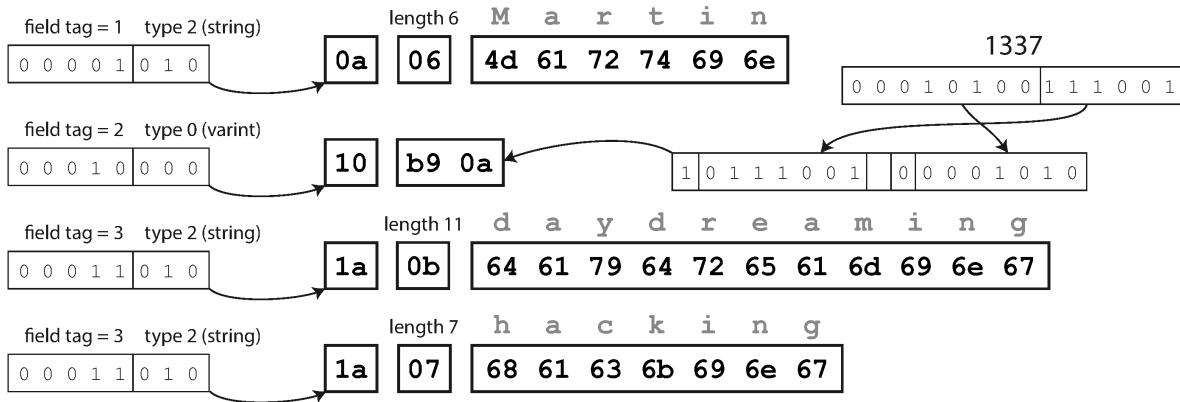


Figure 5-3. Example record encoded using Protocol Buffers.

Similarly to [Figure 5-2](#), each field has a type annotation (to indicate whether it is a string, integer, etc.) and, where required, a length indication (such as the length of a string). The strings that appear in the data (“Martin”, “daydreaming”, “hacking”) are also encoded as ASCII (to be precise, UTF-8), similar to before.

The big difference compared to [Figure 5-2](#) is that there are no field names (`userName`, `favoriteNumber`, `interests`). Instead, the encoded data contains *field tags*, which are numbers (1, 2, and 3). Those are the numbers that appear in the schema definition. Field tags are like aliases for fields—they

are a compact way of saying what field we're talking about, without having to spell out the field name.

As you can see, Protocol Buffers saves even more space by packing the field type and tag number into a single byte. It uses variable-length integers: the number 1337 is encoded in two bytes, with the top bit of each byte used to indicate whether there are still more bytes to come. This means numbers between -64 and 63 are encoded in one byte, numbers between -8192 and 8191 are encoded in two bytes, etc. Bigger numbers use more bytes.

Protocol Buffers doesn't have an explicit list or array datatype. Instead, the `repeated` modifier on the `interests` field indicates that the field contains a list of values, rather than a single value. In the binary encoding, the list elements are represented simply as repeated occurrences of the same field tag within the same record.

Field tags and schema evolution

We said previously that schemas inevitably need to change over time. We call this *schema evolution*. How does Protocol Buffers handle schema changes while keeping backward and forward compatibility?

As you can see from the examples, an encoded record is just the concatenation of its encoded fields. Each field is identified by its tag number (the numbers 1, 2, 3 in the sample schema) and annotated with a datatype (e.g., string or integer). If a field value is not set, it is simply omitted from the encoded record. From this you can see that field tags are critical to the meaning of the encoded data. You can change the name of a field in the schema, since the encoded data never refers to field names, but you cannot change a field's tag, since that would make all existing encoded data invalid.

You can add new fields to the schema, provided that you give each field a new tag number. If old code (which doesn't know about the new tag numbers you added) tries to read data written by new code, including a new field with a tag number it doesn't recognize, it can simply ignore that field. The datatype annotation allows the parser to determine how many bytes it needs to skip, and preserve the unknown fields to avoid the problem in [Figure 5-1](#). 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. If a field was added in the new schema, and you read old data that does not

yet contain that field, it is filled in with a default value (for example, the empty string if the field type is string, or zero if it's a number).

Removing a field is just like adding a field, with backward and forward compatibility concerns reversed. 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. Tag numbers used in the past can be reserved in the schema definition to ensure they are not forgotten.

What about changing the datatype of a field? That is possible with some types—check the documentation for details—but there is a risk that values will get truncated. For example, say you change a 32-bit integer into a 64-bit integer. New code can easily read data written by old code, because the parser can fill in any missing bits with zeros. However, if old code reads data written by new code, the old code is still using a 32-bit variable to hold the value. If the decoded 64-bit value won't fit in 32 bits, it will be truncated.

Avro

Apache Avro is another binary encoding format that is interestingly different from Protocol Buffers. It was started in 2009 as a subproject of Hadoop, as a result of Protocol Buffers not being a good fit for Hadoop's use cases [15].

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. Like Protocol Buffers, this schema language specifies only fields and their types, and not complex validation rules like in JSON Schema.

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", "integer"]},  
        { "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 5-2](#)) 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 5-4](#).

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.

Avro

Byte sequence (32 bytes):

0c	4d	61	72	74	69	6e	02	f2	14	04	16	64	61	79	64	72	65	61	6d
69	6e	67	0e	68	61	63	6b	69	6e	67	00								

Breakdown:

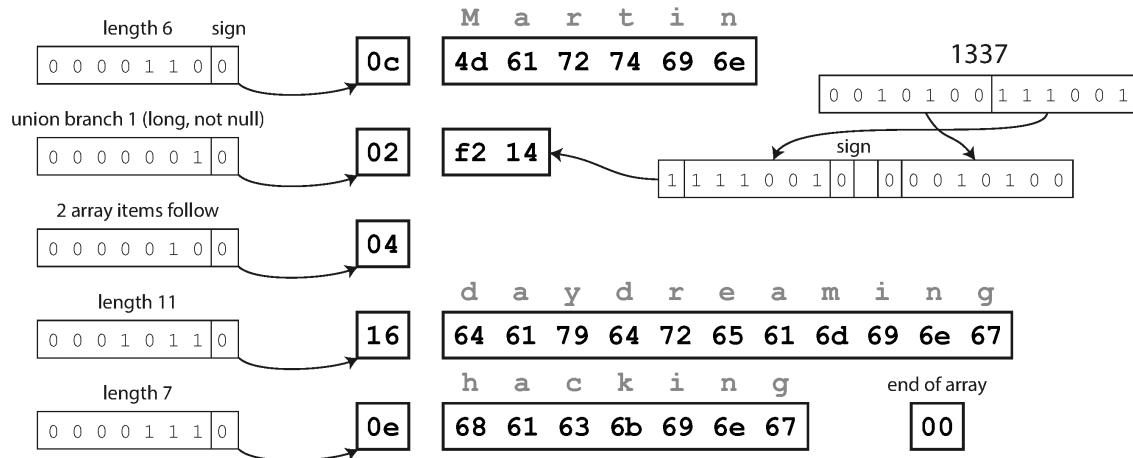


Figure 5-4. Example record encoded using Avro.

To parse the binary data, you go through the fields in the order that they appear in the schema and use the schema to tell you the datatype of each field. This means that the binary data can only be decoded correctly if the code reading the data is using the *exact same schema* as the code that wrote the data. Any mismatch in the schema between the reader and the writer would mean incorrectly decoded data.

So, how does Avro support schema evolution?

The writer's schema and the reader's schema

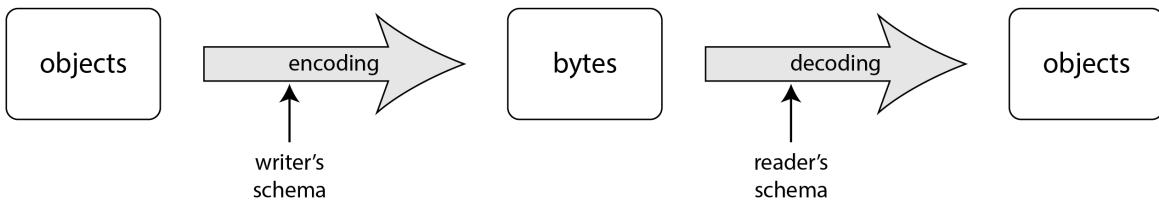
When an application wants to encode some data (to write it to a file or database, to send it over the network, etc.), it encodes the data using whatever version of the schema it knows about—for example, that schema may be compiled into the application.

This is known as the *writer's schema*.

When an application wants to decode some data (read it from a file or database, receive it from the network, etc.), it uses two schemas: the writer's schema that is identical to the one used for encoding, and the *reader's schema*, which may be different.

This is illustrated in [Figure 5-5](#). The reader's schema defines the fields of each record that the application code is expecting, and their types.

Protocol Buffers



Avro

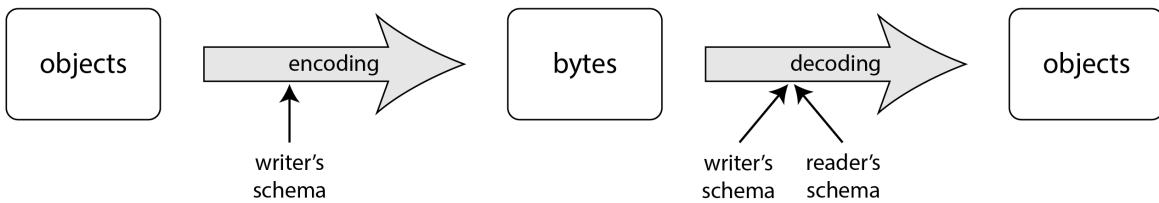


Figure 5-5. In Protocol Buffers, encoding and decoding can use different versions of a schema. In Avro, decoding uses two schemas: the writer's schema must be identical to the one used for encoding, but the reader's schema can be an older or newer version.

If the reader's and writer's schema are the same, decoding is easy. If they are different, Avro 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 [16, 17] defines exactly how this resolution works, and it is illustrated in [Figure 5-6](#).

For example, it's no problem if the writer's schema and the reader's schema have their fields in a different order, because the schema resolution matches up the fields by field name. If the code reading the data encounters a field that appears in the writer's schema but not in the reader's schema, it is ignored. If the code reading the data expects some field, but the writer's

schema does not contain a field of that name, it is filled in with a default value declared in the reader's schema.

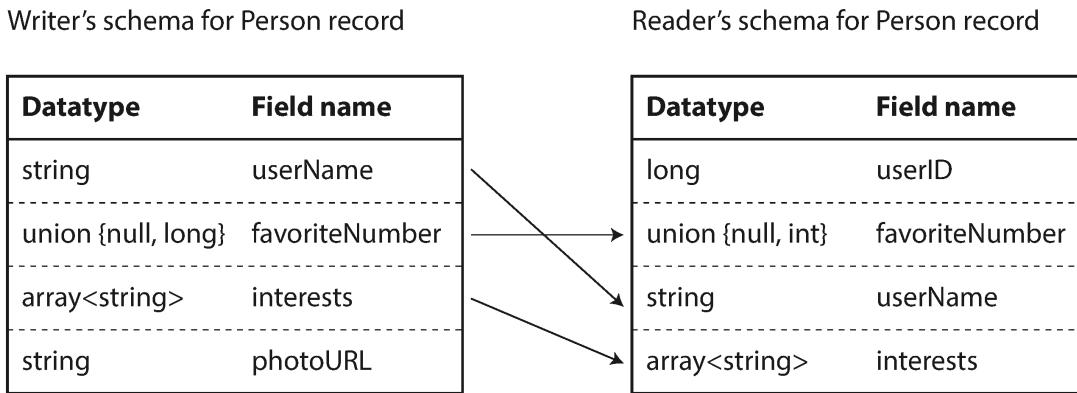


Figure 5-6. An Avro reader resolves differences between the writer's schema and the reader's schema.

Schema evolution rules

With Avro, forward compatibility means that you can have a new version of the schema as writer and an old version of the schema as reader. Conversely, backward compatibility means that you can have a new version of the schema as reader and an old version as writer.

To maintain compatibility, you may only add or remove a field that has a default value. (The field `favoriteNumber` in our Avro schema has a default value of `null`.) For example, say you add a field with a default value, so this new field exists in the new schema but not the old one. When a reader using the

new schema reads a record written with the old schema, the default value is filled in for the missing field.

If you were to add a field that has no default value, new readers wouldn't be able to read data written by old writers, so you would break backward compatibility. If you were to remove a field that has no default value, old readers wouldn't be able to read data written by new writers, so you would break forward compatibility.

In some programming languages, `null` is an acceptable default for any variable, but this is not the case in Avro: if you want to allow a field to be null, you have to use a *union type*. For example, `union { null, long, string } field;` indicates that `field` can be a number, or a string, or null. You can only use `null` as a default value if it is the first branch of the union. 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 [18].

Changing the datatype of a field is possible, provided that Avro can convert the type. Changing the name of a field is possible but a little tricky: the reader's schema can contain aliases for field names, so it can match an old writer's schema field names against the aliases. This means that changing a field name is

backward compatible but not forward compatible. Similarly, adding a branch to a union type is backward compatible but not forward compatible.

But what is the writer's schema?

There is an important question that we've glossed over so far: how does the reader know the writer's schema with which a particular piece of data was encoded? We can't just include the entire schema with every record, because the schema would likely be much bigger than the encoded data, making all the space savings from the binary encoding futile.

The answer depends on the context in which Avro is being used. To give a few examples:

Large file with lots of records

A common use for Avro is for storing a large file containing millions of records, all encoded with the same schema. (We will discuss this kind of situation in [Link to Come].) In this case, the writer of that file can just include the writer's schema once at the beginning of the file. Avro specifies a file format (object container files) to do this.

Database with individually written records

In a database, different records may be written at different points in time using different writer's schemas—you cannot assume that all the records will have the same schema. The simplest solution is to include a version number at the beginning of every encoded record, and to keep a list of schema versions in your 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. Confluent's schema registry for Apache Kafka [19] and LinkedIn's Espresso [20] work this way, for example.

Sending records over a network connection

When two processes are communicating over a bidirectional network connection, they can negotiate the schema version on connection setup and then use that schema for the lifetime of the connection. The Avro RPC protocol (see [“Dataflow Through Services: REST and RPC”](#)) works like this.

A database of schema versions is a useful thing to have in any case, since it acts as documentation and gives you a chance to check schema compatibility [21]. As the version number, you

could use a simple incrementing integer, or you could use a hash of the schema.

Dynamically generated schemas

One advantage of Avro's approach, compared to Protocol Buffers, is that the schema doesn't contain any tag numbers. But why is this important? What's the problem with keeping a couple of numbers in the schema?

The difference is that Avro is friendlier to *dynamically generated* schemas. For example, say you have a relational database whose contents you want to dump to a file, and you want to use a binary format to avoid the aforementioned problems with textual formats (JSON, CSV, XML). If you use Avro, you can fairly easily generate an Avro schema (in the JSON representation we saw earlier) from the relational schema and encode the database contents using that schema, dumping it all to an Avro object container file [22]. You can generate a record schema for each database table, and each column becomes a field in that record. The column name in the database maps to the field name in Avro.

Now, if the database schema changes (for example, a table has one column added and one column removed), you can just

generate a new Avro schema from the updated database schema and export data in the new Avro schema. The data export process does not need to pay any attention to the schema change—it can simply do the schema conversion every time it runs. Anyone who reads the new data files will see that the fields of the record have changed, but since the fields are identified by name, the updated writer's schema can still be matched up with the old reader's schema.

By contrast, if you were using 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 Protocol Buffers, whereas it was for Avro.

The Merits of Schemas

As we saw, Protocol Buffers and Avro both 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 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 [[23](#), [24](#)]. It was used to define various network protocols, and its binary encoding (DER) is still used to encode SSL certificates (X.509), for example [[25](#)]. ASN.1 supports schema evolution using tag numbers, similar to Protocol Buffers [[26](#)]. However, it’s also very complex and badly documented, so ASN.1 is probably not a good choice for new applications.

Many data systems also implement some kind of proprietary binary encoding for their data. For example, most relational databases have a network protocol over which you can send queries to the database and get back responses. Those protocols are generally specific to a particular database, and the database vendor provides a driver (e.g., using the ODBC or JDBC APIs) that decodes responses from the database’s network protocol into in-memory data structures.

So, we can see that although textual data formats such as JSON, XML, and CSV are widespread, binary encodings based on schemas are also a viable option. They have a number of nice properties:

- They can be much more compact than the various “binary JSON” variants, since they can omit field names from the encoded data.
- The schema is a valuable form of documentation, and because the schema is required for decoding, you can be sure that it is up to date (whereas manually maintained documentation may easily diverge from reality).
- Keeping a database of schemas allows you to check forward and backward compatibility of schema changes, before anything is deployed.
- For users of statically typed programming languages, the ability to generate code from the schema is useful, since it enables type-checking at compile time.

In summary, schema evolution allows the same kind of flexibility as schemaless/schema-on-read JSON databases provide (see [“Schema flexibility in the document model”](#)), while also providing better guarantees about your data and better tooling.

Modes of Dataflow

At the beginning of this chapter we said that whenever you want to send some data to another process with which you don't share memory—for example, whenever you want to send data over the network or write it to a file—you need to encode it as a sequence of bytes. We then discussed a variety of different encodings for doing this.

We talked about forward and backward compatibility, which are important for evolvability (making change easy by allowing you to upgrade different parts of your system independently, and not having to change everything at once). Compatibility is a relationship between one process that encodes the data, and another process that decodes it.

That's a fairly abstract idea—there are many ways data can flow from one process to another. Who encodes the data, and who decodes it? In the rest of this chapter we will explore some of the most common ways how data flows between processes:

- Via databases (see “[Dataflow Through Databases](#)”)
- Via service calls (see “[Dataflow Through Services: REST and RPC](#)”)

- Via workflow engines (see “[Durable Execution and Workflows](#)”)
- Via asynchronous messages (see “[Event-Driven Architectures](#)”)

Dataflow Through Databases

In a database, the process that writes to the database encodes the data, and the process that reads from the database decodes it. There may just be a single process accessing the database, in which case the reader is simply a later version of the same process—in that case you can think of storing something in the database as *sending a message to your future self*.

Backward compatibility is clearly necessary here; otherwise your future self won’t be able to decode what you previously wrote.

In general, it’s common for several different processes to be accessing a database at the same time. Those processes might be several different applications or services, or they may simply be several instances of the same service (running in parallel for scalability or fault tolerance). Either way, in an environment where the application is changing, it is likely that some processes accessing the database will be running newer code

and some will be running older code—for example because a new version is currently being deployed in a rolling upgrade, so some instances have been updated while others haven't yet.

This means that a value in the database may be written by a *newer* version of the code, and subsequently read by an *older* version of the code that is still running. Thus, forward compatibility is also often required for databases.

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. When an old row is read, the database fills in `null`s for any columns that are missing from the encoded data on disk. 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.

More complex schema changes—for example, changing a single-valued attribute to be multi-valued, or moving some data into a separate table—still require data to be rewritten, often at the application level [27]. Maintaining forward and backward compatibility across such migrations is still a research problem [28].

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](#)”). 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”](#)).

In [Link to Come] we will talk more about using data in archival storage.

Dataflow Through Services: REST and RPC

When you have processes that need to communicate over a network, there are a few different ways of arranging that communication. The most common arrangement is to have two roles: *clients* and *servers*. The servers expose an API over the network, and the clients can connect to the servers to make requests to that API. The API exposed by the server is known as a *service*.

The web works this way: clients (web browsers) make requests to web servers, making `GET` requests to download HTML, CSS, JavaScript, images, etc., and making `POST` requests to submit

data to the server. The API consists of a standardized set of protocols and data formats (HTTP, URLs, SSL/TLS, HTML, etc.). Because web browsers, web servers, and website authors mostly agree on these standards, you can use any web browser to access any website (at least in theory!).

Web browsers are not the only type of client. For example, native apps running on mobile devices and desktop computers often talk to servers, and client-side JavaScript applications running inside web browsers can also make HTTP requests. 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 (most often 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.

In some ways, services are similar to databases: they typically allow clients to submit and query data. However, while databases allow arbitrary queries using the query languages we discussed in [Chapter 3](#), services expose an application-specific API that only allows inputs and outputs that are predetermined by the business logic (application code) of the service [\[29\]](#). This restriction provides a degree of encapsulation: services can

impose fine-grained restrictions on what clients can and cannot do.

A key design goal of a service-oriented/microservices architecture is to make the application easier to change and maintain by making services independently deployable and evolvable. A common principle is that 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. We should therefore 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.

Web services

When HTTP is used as the underlying protocol for talking to the service, it is called a *web service*. Web services are commonly used when building a service oriented or microservices architecture (discussed earlier in [“Microservices and Serverless”](#)). The term “web service” is perhaps a slight misnomer, because web services are not only used on the web, but in several different contexts. For example:

1. A client application running on a user's device (e.g., a native app on a mobile device, or a JavaScript web app in a browser) making requests to a service over HTTP. These requests typically go over the public internet.
2. One service making requests to another service owned by the same organization, often located within the same datacenter, as part of a service-oriented/microservices architecture.
3. One service making requests to a service owned by a different organization, usually via the internet. This is used for data exchange between different organizations' backend systems. This category includes public APIs provided by online services, such as credit card processing systems, or OAuth for shared access to user data.

The most popular service design philosophy is REST, which builds upon the principles of HTTP [30, 31]. It emphasizes simple data formats, using URLs for identifying resources and using HTTP features for cache control, authentication, and content type negotiation. An API designed according to the principles of REST is called *RESTful*.

Code that needs to invoke a web service API must know which HTTP endpoint to query, and what data format to send and expect in response. Even if a service adopts RESTful design principles, clients need to somehow find out these details.

Service developers often use an interface definition language (IDL) to define and document their service's API endpoints and data models, and to evolve them over time. Other developers can then use the service definition to determine how to query the service. The two most popular service IDLs are OpenAPI (also known as Swagger [32]) and gRPC. OpenAPI is used for web services that send and receive JSON data, while gRPC services send and receive Protocol Buffers.

Developers typically write OpenAPI service definitions in JSON or YAML; see [Example 5-3](#). The service definition allows developers to define service endpoints, documentation, versions, data models, and much more. gRPC definitions look similar, but are defined using Protocol Buffers service definitions.

Example 5-3. Example OpenAPI service definition in YAML

```
openapi: 3.0.0
info:
  title: Ping, Pong
  version: 1.0.0
servers:
  - url: http://localhost:8080
paths:
  /ping:
```

```
get:  
    summary: Given a ping, returns a pong message  
    responses:  
        '200':  
            description: A pong  
            content:  
                application/json:  
                    schema:  
                        type: object  
                        properties:  
                            message:  
                                type: string  
                            example: Pong!
```

Even if a design philosophy and IDL are adopted, developers must still write the code that implements their service's API calls. A service framework is often adopted to simplify this effort. Service frameworks such as Spring Boot, FastAPI, and gRPC allow developers to write the business logic for each API endpoint while the framework code handles routing, metrics, caching, authentication, and so on. [Example 5-4](#) shows an example Python implementation of the service defined in [Example 5-3](#).

Example 5-4. Example FastAPI service implementing the

definition from Example 5-3

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI(title="Ping, Pong", version="1.0.0"

class PongResponse(BaseModel):
    message: str = "Pong!"

@app.get("/ping", response_model=PongResponse,
         summary="Given a ping, returns a pong me
async def ping():
    return PongResponse()
```

Many frameworks couple service definitions and server code together. In some cases, such as with the popular Python FastAPI framework, servers are written in code and an IDL is generated automatically. In other cases, such as with gRPC, the service definition is written first, and server code scaffolding is generated. Both approaches allow developers to generate client libraries and SDKs in a variety of languages from the service definition. In addition to code generation, IDL tools such as Swagger's can generate documentation, verify schema change compatibility, and provide a graphical user interfaces for developers to query and test services.

The problems with remote procedure calls (RPCs)

Web services are merely the latest incarnation of a long line of technologies for making API requests over a network, many of which received a lot of hype but have serious problems.

Enterprise JavaBeans (EJB) and Java's Remote Method Invocation (RMI) are limited to Java. The Distributed Component Object Model (DCOM) is limited to Microsoft platforms. The Common Object Request Broker Architecture (CORBA) is excessively complex, and does not provide backward or forward compatibility [33]. SOAP and the WS-* web services framework aim to provide interoperability across vendors, but are also plagued by complexity and compatibility problems [34, 35, 36].

All of these are based on the idea of a *remote procedure call* (RPC), which has been around since the 1970s [37]. The RPC model tries to make a request to a remote network service look the same as calling a function or method in your programming language, within the same process (this abstraction is called *location transparency*). Although RPC seems convenient at first, the approach is fundamentally flawed [38, 39]. A network request is very different from a local function call:

- A local function call is predictable and either succeeds or fails, depending only on parameters that are under your control. A network request is unpredictable: the request or response may be lost due to a network problem, or the remote machine may be slow or unavailable, and such problems are entirely outside of your control. Network problems are common, so you have to anticipate them, for example by retrying a failed request.
- A local function call either returns a result, or throws an exception, or never returns (because it goes into an infinite loop or the process crashes). A network request has another possible outcome: it may return without a result, due to a *timeout*. In that case, you simply don't know what happened: if you don't get a response from the remote service, you have no way of knowing whether the request got through or not. (We discuss this issue in more detail in [Link to Come].)
- If you retry a failed network request, it could happen that the previous request actually got through, and only the response was 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 [40]. Local function calls don't have this problem. (We discuss idempotence in more detail in [Link to Come].)

- Every time you call a local function, it normally takes about the same time to execute. A network request is much slower than a function call, and its latency is also wildly variable: at good times it may complete in less than a millisecond, but when the network is congested or the remote service is overloaded it may take many seconds to do exactly the same thing.
- When you call a local function, you can efficiently pass it references (pointers) to objects in local memory. When you make a network request, all those parameters need to be encoded into a sequence of bytes that can be sent over the network. That's okay if the parameters are immutable primitives like numbers or short strings, but it quickly becomes problematic with larger amounts of data and mutable 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} ,⁵³ for example (see “[JSON, XML, and Binary Variants](#)”). 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 treats state transfer over a network as a process that is distinct from a function call.

Load balancers, service discovery, and service meshes

All services communicate over the network. For this reason, a client must know the address of the service it's connecting to—a problem known as *service discovery*. The simplest approach is to configure a client to connect to the IP address and port where the service is running. This configuration will work, but if the server goes offline, is transferred to a new machine, or becomes overloaded, the client has to be manually reconfigured.

To provide higher availability and scalability, there are usually multiple instances of a service running on different machines, any of which can handle an incoming request. Spreading requests across these instances is called *load balancing* [41]. There are many load balancing and service discovery solutions available:

- *Hardware load balancers* are specialized pieces of equipment that are installed in data centers. They allow clients to connect to a single host and port, and incoming connections are routed to one of the servers running the service. Such load balancers detect network failures when connecting to a downstream server and shift the traffic to other servers.
- *Software load balancers* behave in much the same way as hardware load balancers. But rather than requiring a special appliance, software load balancers such as Nginx and HAProxy are applications that can be installed on a standard machine.
- The *domain name service (DNS)* is how domain names are resolved on the Internet when you open a webpage. It supports load balancing by allowing multiple IP addresses to be associated with a single domain name. Clients can then be configured to connect to a service using a domain name rather than IP address, and the client's network layer picks which IP address to use when making a connection. One drawback of this approach is that DNS is designed to propagate changes over longer periods of time, and to cache DNS entries. If servers are started, stopped, or moved frequently, clients might see stale IP addresses that no longer have a server running on them.

- *Service discovery systems* use a centralized registry rather than DNS to track which service endpoints are available. When a new service instance starts up, it registers itself with the service discovery system by declaring the host and port it's listening on, along with relevant metadata such as shard ownership information (see [Chapter 7](#)), data center location, and more. The service then periodically sends a heartbeat signal to the discovery system to signal that the service is still available.
When a client wishes to connect to a service, it first queries the discovery system to get a list of available endpoints, and then connects directly to the endpoint. Compared to DNS, service discovery supports a much more dynamic environment where service instances change frequently. Discovery systems also give clients more metadata about the service they're connecting to, which enables clients to make smarter load balancing decisions.
- *Service meshes* are a sophisticated form of load balancing that combine software load balancers and service discovery. Unlike traditional software load balancers, which run on a separate machine, service mesh load balancers are typically deployed as an in-process client library or as a process or “sidecar” container on both the client and server. Client applications connect to their own local service load balancer,

which connects to the server's load balancer. From there, the connection is routed to the local server process.

Though complicated, this topology offers a number of advantages. Because the clients and servers are routed entirely through local connections, connection encryption can be handled entirely at the load balancer level. This shields clients and servers from having to deal with the complexities of SSL certificates and TLS. Mesh systems also provide sophisticated observability. They can track which services are calling each other in realtime, detect failures, track traffic load, and more.

Which solution is appropriate depends on an organization's needs. Those running in a very dynamic service environment with an orchestrator such as Kubernetes often choose to run a service mesh such as Istio or Linkerd. Specialized infrastructure such as databases or messaging systems might require their own purpose-built load balancer. Simpler deployments are best served with software load balancers.

Data encoding and evolution for RPC

For evolvability, it is important that RPC clients and servers can be changed and deployed independently. Compared to data flowing through databases (as described in the last section), we

can make a simplifying assumption in the case of dataflow through services: it is reasonable to assume that all the servers will be updated first, and all the clients second. Thus, you only need backward compatibility on requests, and forward compatibility on responses.

The backward and forward compatibility properties of an RPC scheme are inherited from whatever encoding it uses:

- gRPC (Protocol Buffers) and Avro RPC can be evolved according to the compatibility rules of the respective encoding format.
- RESTful APIs most commonly use JSON for responses, and JSON or URI-encoded/form-encoded request parameters for requests. Adding optional request parameters and adding new fields to response objects are usually considered changes that maintain compatibility.

Service compatibility is made harder by the fact that RPC is often used for communication across organizational boundaries, so the provider of a service often has no control over its clients and cannot force them to upgrade. Thus, compatibility needs to be maintained for a long time, perhaps indefinitely. If a compatibility-breaking change is required, the

service provider often ends up maintaining multiple versions of the service API side by side.

There is no agreement on how API versioning should work (i.e., how a client can indicate which version of the API it wants to use [42]). 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 [43].

Durable Execution and Workflows

By definition, service-based architectures have multiple services that are all responsible for different portions of an application. Consider a payment processing application that charges a credit card and deposits the funds into a bank account. This system would likely have different services responsible for fraud detection, credit card integration, bank integration, and so on.

Processing a single payment in our example requires many service calls. A payment processor service might invoke the fraud detection service to check for fraud, call the credit card

service to debit the credit card, and call the banking service to deposit debited funds, as shown in [Figure 5-7](#). We call this sequence of steps a *workflow*, and each step a *task*. Workflows are typically defined as a graph of tasks. Workflow definitions may be written in a general-purpose programming language, a domain specific language (DSL), or a markup language such as Business Process Execution Language (BPEL) [44].

TASKS, ACTIVITIES, AND FUNCTIONS

Different workflow engines use different names for tasks. Temporal, for example, uses the term *activity*. Others refer to tasks as *durable functions*. Though the names differ, the concepts are the same.

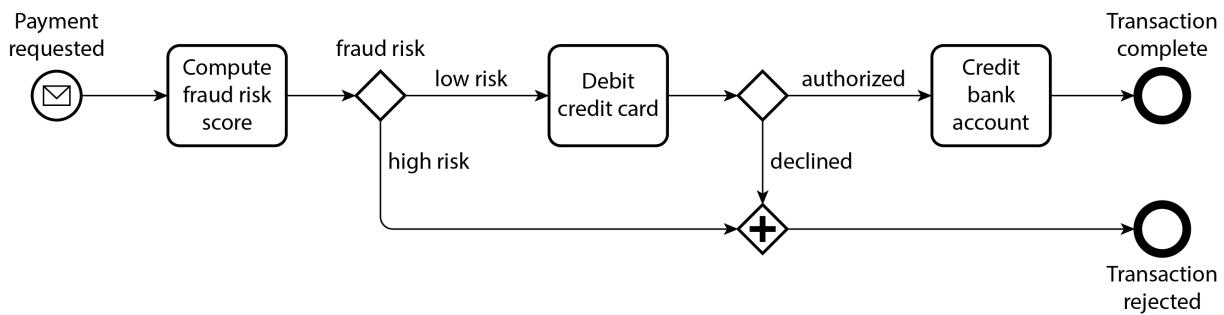


Figure 5-7. Example of a workflow expressed using Business Process Model and Notation (BPMN), a graphical notation.

Workflows are run, or executed, by a *workflow engine*. Workflow engines determine when to run each task, on which machine a task must be run, what to do if a task fails (e.g., if the

machine crashes while the task is running), how many tasks are allowed to execute in parallel, and more.

Workflow engines are typically composed of an orchestrator and an executor. The orchestrator is responsible for scheduling tasks to be executed and the executor is responsible for executing tasks. Execution begins when a workflow is triggered. The orchestrator triggers the workflow itself if users define a time-based schedule, such as hourly execution. External sources such as a web service or even a human can also trigger workflow executions. Once triggered, executors are invoked to run tasks.

There are many kinds of workflow engines that address a diverse set of use cases. Some, such as Airflow, Dagster, and Prefect, integrate with data systems and orchestrate ETL tasks. Others, such as Camunda and Orkes, provide a graphical notation for workflows (such as BPMN, used in [Figure 5-7](#)) so that non-engineers can more easily define and execute workflows. Still others, such as Temporal and Restate provide *durable execution*.

Durable execution

Durable execution frameworks have become a popular way to build service-based architectures that require transactionality. In our payment example, we would like to process each payment exactly once. A failure while the workflow is executing could result in a credit card charge, but no corresponding bank account deposit. In a service-based architecture, we can't simply wrap the two tasks in a database transaction. Moreover, we might be interacting with third-party payment gateways that we have limited control over.

Durable execution frameworks are a way to provide *exactly-once semantics* for workflows. If a task fails, the framework will re-execute the task, but will skip any RPC calls or state changes that the task made successfully before failing. Instead, the framework will pretend to make the call, but will instead return the results from the previous call. This is possible because durable execution frameworks log all RPCs and state changes to durable storage like a write-ahead log [45, 46]. [Example 5-5](#) shows an example of a workflow definition that supports durable execution using Temporal.

Example 5-5. A Temporal workflow definition fragment for

the payment workflow in [Figure 5-7](#).

```
@workflow.defn
class PaymentWorkflow:
    @workflow.run
    async def run(self, payment: PaymentRequest):
        is_fraud = await workflow.execute_activity(
            check_fraud,
            payment,
            start_to_close_timeout=timedelta(seconds=10)
        )
        if is_fraud:
            return PaymentResultFraudulent
        credit_card_response = await workflow.execute_activity(
            debit_credit_card,
            payment,
            start_to_close_timeout=timedelta(seconds=10)
        )
        # ...
```

Frameworks like Temporal are not without their challenges. External services, such as the third-party payment gateway in our example, must still provide an idempotent API. Developers must remember to use unique IDs for these APIs to prevent duplicate execution [47]. And because durable execution frameworks log each RPC call in order, it expects a subsequent

execution to make the same RPC calls in the same order. This makes code changes brittle. You might introduce undefined behavior simply by re-ordering function calls [48].

Similarly, because durable execution frameworks expect to replay all code deterministically (the same inputs produce the same outputs), nondeterministic code such as random number generators or system clocks are problematic [48]. Frameworks often provide their own, deterministic implementations of such library functions, but you have to remember to use them. In some cases, such as with Temporal's workflowcheck tool, frameworks provide static analysis tools to determine if nondeterministic behavior has been introduced.

NOTE

Making code deterministic is a powerful idea, but tricky to do robustly. In [Link to Come] we will return to this topic.

Event-Driven Architectures

In this final section, we will briefly look at *event-driven architectures*, which are another way how encoded data can flow from one process to another. A request is called an *event* or *message*; unlike RPC, the sender usually does not wait for the

recipient to process the event. Moreover, events are typically not sent to the recipient via a direct network connection, but go via an intermediary called a *message broker* (also called an *event broker*, *message queue*, or *message-oriented middleware*), which stores the message temporarily. [49].

Using a message broker has several advantages compared to direct RPC:

- It can act as a buffer if the recipient is unavailable or overloaded, and thus improve system reliability.
- It can automatically redeliver messages to a process that has crashed, and thus prevent messages from being lost.
- It avoids the need for service discovery, since senders do not need to directly connect to the IP address of the recipient.
- It allows the same 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).

The communication via a message broker is *asynchronous*: the sender doesn't wait for the message to be delivered, but simply sends it and then forgets about it. It's possible to implement a synchronous RPC-like model by having the sender wait for a response on a separate channel.

Message brokers

In the past, the landscape of message brokers was dominated by commercial enterprise software from companies such as TIBCO, IBM WebSphere, and webMethods, before open source implementations such as RabbitMQ, ActiveMQ, HornetQ, NATS, and Apache Kafka became popular. More recently, cloud services such as Amazon Kinesis, Azure Service Bus, and Google Cloud Pub/Sub have gained adoption. We will compare them in more detail in [Link to Come].

The detailed delivery semantics vary by implementation and configuration, but in general, two message distribution patterns are most often used:

- One process adds a message to a named *queue*, and the broker delivers that message to a *consumer* of that queue. If there are multiple consumers, one of them receives the message.
- One process publishes a message to a named *topic*, and the broker delivers that message to all *subscribers* of that topic. If there are multiple subscribers, they all receive the message.

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. A common approach is to use Protocol Buffers, Avro, or JSON, and to deploy a schema registry alongside the message broker to store all the valid schema versions and check their compatibility [19, 21]. AsyncAPI, a messaging-based equivalent of OpenAPI, can also be used to specify the schema of messages.

Message brokers differ in terms of how durable their messages are. Many write messages to disk, so that they are not lost in case the message broker crashes or needs to be restarted. Unlike databases, many message brokers automatically delete messages again after they have been consumed. Some brokers can be configured to store messages indefinitely, which you would require if you want to use event sourcing (see [“Event Sourcing and CQRS”](#)).

If a consumer republishes messages to another topic, you may need to be careful to preserve unknown fields, to prevent the issue described previously in the context of databases ([Figure 5-1](#)).

Distributed actor frameworks

The *actor model* is a programming model for concurrency in a single process. Rather than dealing directly with threads (and

the associated problems of race conditions, locking, and deadlock), logic is encapsulated in *actors*. Each actor typically represents one client or entity, it may have some local state (which is not shared with any other actor), and it communicates with other actors by sending and receiving asynchronous messages. Message delivery is not guaranteed: in certain error scenarios, messages will be lost. Since each actor processes only one message at a time, it doesn't need to worry about threads, and each actor can be scheduled independently by the framework.

In *distributed actor frameworks* such as Akka, Orleans [50], and Erlang/OTP, this programming model is used to scale an application across multiple nodes. The same message-passing mechanism is used, no matter whether the sender and recipient are on the same node or different nodes. If they are on different nodes, the message is transparently encoded into a byte sequence, sent over the network, and decoded on the other side.

Location transparency works better in the actor model than in RPC, because the actor model already assumes that messages may be lost, even within a single process. Although latency over the network is likely higher than within the same process, there is less of a fundamental mismatch between local and remote communication when using the actor model.

A distributed actor framework essentially integrates a message broker and the actor programming model into a single framework. However, if you want to perform rolling upgrades of your actor-based application, you still have to worry about forward and backward compatibility, as messages may be sent from a node running the new version to a node running the old version, and vice versa. This can be achieved by using one of the encodings discussed in this chapter.

Summary

In this chapter we looked at several ways of turning data structures into bytes on the network or bytes on disk. We saw how the details of these encodings affect not only their efficiency, but more importantly also the architecture of applications and your options for evolving them.

In particular, many services need to support rolling upgrades, where a new version of a service is gradually deployed to a few nodes at a time, rather than deploying to all nodes simultaneously. Rolling upgrades allow new versions of a service to be released without downtime (thus encouraging frequent small releases over rare big releases) and make deployments less risky (allowing faulty releases to be detected

and rolled back before they affect a large number of users). These properties are hugely beneficial for *evolvability*, the ease of making changes to an application.

During rolling upgrades, or for various other reasons, we must assume that different nodes are running the different versions of our application's code. Thus, it is important that all data flowing around the system is encoded in a way that provides backward compatibility (new code can read old data) and forward compatibility (old code can read new data).

We discussed several data encoding formats and their compatibility properties:

- Programming language-specific encodings are restricted to a single programming language and often fail to provide forward and backward compatibility.
- Textual formats like JSON, XML, and CSV are widespread, and their compatibility depends on how you use them. They have optional schema languages, which are sometimes helpful and sometimes a hindrance. These formats are somewhat vague about datatypes, so you have to be careful with things like numbers and binary strings.
- Binary schema-driven formats like Protocol Buffers and Avro allow compact, efficient encoding with clearly defined

forward and backward compatibility semantics. The schemas can be useful for documentation and code generation in statically typed languages. However, these formats have the downside that data needs to be decoded before it is human-readable.

We also discussed several modes of dataflow, illustrating different scenarios in which data encodings are important:

- Databases, where the process writing to the database encodes the data and the process reading from the database decodes it
- RPC and REST APIs, where the client encodes a request, the server decodes the request and encodes a response, and the client finally decodes the response
- Event-driven architectures (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.

FOOTNOTES

REFERENCES

[CWE-502: Deserialization of Untrusted Data](#). Common Weakness Enumeration, cwe.mitre.org, July 2006. Archived at perma.cc/26EU-UK9Y

Steve Breen. [What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability](#). foxglovesecurity.com, November 2015. Archived at perma.cc/9U97-UVVD

Patrick McKenzie. [What the Rails Security Issue Means for Your Startup](#). kalzumeus.com, January 2013. Archived at perma.cc/2MBJ-7PZ6

Brian Goetz. [Towards Better Serialization](#). openjdk.org, June 2019. Archived at perma.cc/UK6U-GQDE

Eishay Smith. [jvm-serializers wiki](#). github.com, October 2023. Archived at perma.cc/PJP7-WCNG

[XML Is a Poor Copy of S-Expressions](#). wiki.c2.com, May 2013. Archived at perma.cc/7FAN-YBKL

Julia Evans. [Examples of floating point problems](#). jvns.ca, January 2023. Archived at perma.cc/M57L-QKKW

Matt Harris. [Snowflake: An Update and Some Very Important Information](#). Email to *Twitter Development Talk* mailing list, October 2010. Archived at perma.cc/8UBV-MZ3D

Iakov Shafranovich. [RFC 4180: Common Format and MIME Type for Comma-Separated Values \(CSV\) Files](#). IETF, October 2005.

Andy Coates. [Evolving JSON Schemas - Part I](#) and [Part II](#). *creekservice.org*, January 2024. Archived at [perma.cc/MZW3-UA54](#) and [perma.cc/GT5H-WKZ5](#)

Pierre Genevès, Nabil Layaïda, and Vincent Quint. [Ensuring Query Compatibility with Evolving XML Schemas](#). INRIA Technical Report 6711, November 2008.

Tim Bray. [Bits On the Wire](#). *tbray.org*, November 2019. Archived at [perma.cc/3BT3-BQU3](#)

Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. [Thrift: Scalable Cross-Language Services Implementation](#). Facebook technical report, April 2007. Archived at [perma.cc/22BS-TUFB](#)

Martin Kleppmann. [Schema Evolution in Avro, Protocol Buffers and Thrift](#). *martin.kleppmann.com*, December 2012. Archived at [perma.cc/E4R2-9RJT](#)

Doug Cutting, Chad Walters, Jim Kellerman, et al. [\[PROPOSAL\] New Subproject: Avro](#). Email thread on *hadoop-general* mailing list, *lists.apache.org*, April 2009. Archived at [perma.cc/4A79-BMEB](#)

Apache Software Foundation. [Apache Avro 1.12.0 Specification](#). *avro.apache.org*, August 2024. Archived at [perma.cc/C36P-5EBQ](#)

Apache Software Foundation. [Avro schemas as LL\(1\) CFG definitions](#). *avro.apache.org*, August 2024. Archived at [perma.cc/JB44-EM9Q](#)

Tony Hoare. [Null References: The Billion Dollar Mistake](#). Talk at *QCon London*, March 2009.

Confluent, Inc. [Schema Registry Overview](#). *docs.confluent.io*, 2024. Archived at [perma.cc/92C3-A9JA](#)

Aditya Auradkar and Tom Quiggle. [Introducing Espresso—LinkedIn’s Hot New Distributed Document Store](#). *engineering.linkedin.com*, January 2015. Archived at perma.cc/FX4P-VW9T

Jay Kreps. [Putting Apache Kafka to Use: A Practical Guide to Building a Stream Data Platform \(Part 2\)](#). *confluent.io*, February 2015. Archived at perma.cc/8UA4-ZS5S

Gwen Shapira. [The Problem of Managing Schemas](#). *oreilly.com*, November 2014. Archived at perma.cc/BY8Q-RYV3

John Larmouth. [ASN.1 Complete](#). Morgan Kaufmann, 1999. ISBN: 978-0-122-33435-1. Archived at perma.cc/GB7Y-XSXQ

Burton S. Kaliski Jr. [A Layman’s Guide to a Subset of ASN.1, BER, and DER](#). Technical Note, RSA Data Security, Inc., November 1993. Archived at perma.cc/2LMN-W9U8

Jacob Hoffman-Andrews. [A Warm Welcome to ASN.1 and DER](#). *letsencrypt.org*, April 2020. Archived at perma.cc/CYT2-GPQ8

Lev Walkin. [Question: Extensibility and Dropping Fields](#). *lionet.info*, September 2010. Archived at perma.cc/VX8E-NLH3

Jacqueline Xu. [Online migrations at scale](#). *stripe.com*, February 2017. Archived at perma.cc/X59W-DK7Y

Geoffrey Litt, Peter van Hardenberg, and Orion Henry. [Project Cambria: Translate your data with lenses](#). Technical Report, *Ink & Switch*, October 2020. Archived at perma.cc/WA4V-VKDB

Pat Helland. [Data on the Outside Versus Data on the Inside](#). At *2nd Biennial Conference on Innovative Data Systems Research* (CIDR), January 2005.

Roy Thomas Fielding. [Architectural Styles and the Design of Network-Based Software Architectures](#). PhD Thesis, University of California, Irvine, 2000. Archived at [perma.cc/LWY9-7BPE](#)

Roy Thomas Fielding. [REST APIs must be hypertext-driven.](#) "roy.gbiv.com, October 2008. Archived at [perma.cc/M2ZW-8ATG](#)

[OpenAPI Specification Version 3.1.0](#). swagger.io, February 2021. Archived at [perma.cc/3S6S-K5M4](#)

Michi Henning. [The Rise and Fall of CORBA](#). *Communications of the ACM*, volume 51, issue 8, pages 52–57, August 2008. [doi:10.1145/1378704.1378718](#)

Pete Lacey. [The S Stands for Simple](#). harmful.cat-v.org, November 2006. Archived at [perma.cc/4PMK-Z9X7](#)

Stefan Tilkov. [Interview: Pete Lacey Criticizes Web Services](#). infoq.com, December 2006. Archived at [perma.cc/JWF4-XY3P](#)

Tim Bray. [The Loyal WS-Opposition](#). tbray.org, September 2004. Archived at [perma.cc/J5Q8-69Q2](#)

Andrew D. Birrell and Bruce Jay Nelson. [Implementing Remote Procedure Calls](#). *ACM Transactions on Computer Systems* (TOCS), volume 2, issue 1, pages 39–59, February 1984. [doi:10.1145/2080.357392](#)

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. Archived at [perma.cc/8LRZ-BSZR](#)

Steve Vinoski. [Convenience over Correctness](#). *IEEE Internet Computing*, volume 12, issue 4, pages 89–92, July 2008. [doi:10.1109/MIC.2008.75](#)

Brandur Leach. [Designing robust and predictable APIs with idempotency](#). *stripe.com*, February 2017. Archived at perma.cc/JD22-XZQT

Sam Rose. [Load Balancing](#). *samwho.dev*, April 2023. Archived at perma.cc/Q7BA-9AE2

Troy Hunt. [Your API versioning is wrong, which is why I decided to do it 3 different wrong ways](#). *troyhunt.com*, February 2014. Archived at perma.cc/9DSW-DGR5

Brandur Leach. [APIs as infrastructure: future-proofing Stripe with versioning](#). *stripe.com*, August 2017. Archived at perma.cc/L63K-USFW

Alexandre Alves, Assaf Arkin, Sid Askary, et al. [Web Services Business Process Execution Language Version 2.0](#). *docs.oasis-open.org*, April 2007.

[What is a Temporal Service?](#) *docs.temporal.io*, 2024. Archived at perma.cc/32P3-CJ9V

Stephan Ewen. [Why we built Restate](#). *restate.dev*, August 2023. Archived at perma.cc/BJJ2-X75K

Keith Tenzer and Joshua Smith. [Idempotency and Durable Execution](#). *temporal.io*, February 2024. Archived at perma.cc/9LGW-PCLU

[What is a Temporal Workflow?](#) *docs.temporal.io*, 2024. Archived at perma.cc/B5C5-Y396

Srinath Perera. [Exploring Event-Driven Architecture: A Beginner's Guide for Cloud Native Developers](#). *wso2.com*, August 2023. Archived at archive.org

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. Archived at perma.cc/PD3U-WDMF

Chapter 6. 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)

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. The GitHub repo for this book is <https://github.com/ept/ddia2-feedback>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out on GitHub.

Replication means keeping a copy of the same data on multiple machines that are connected via a network. As discussed in

[“Distributed versus Single-Node Systems”](#), there are several reasons why you might want to replicate data:

- To keep data geographically close to your users (and thus reduce access latency)
- To allow the system to continue working even if some of its parts have failed (and thus increase availability)
- To scale out the number of machines that can serve read queries (and thus increase read throughput)

In this chapter we will assume that your dataset is small enough that each machine can hold a copy of the entire dataset.

In [Chapter 7](#) we will relax that assumption and discuss *sharding* (*partitioning*) of datasets that are too big for a single machine.

In later chapters we will discuss various kinds of faults that can occur in a replicated data system, and how to deal with them.

If the data that you’re replicating does not change over time, then replication is easy: you just need to copy the data to every node once, and you’re done. All of the difficulty in replication lies in handling *changes* to replicated data, and that’s what this chapter is about. We will discuss three families of 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. Despite being so old, concepts such as *eventual consistency* still cause confusion. In “[Problems with Replication Lag](#)” we will get more precise about eventual consistency and discuss things like the *read-your-writes* and *monotonic reads* guarantees.

BACKUPS AND REPLICATION

You might be wondering whether you still need backups if you have replication. The answer is yes, because they have different purposes: replicas quickly reflect writes from one node on other nodes, but backups store old snapshots of the data so that you can go back in time. If you accidentally delete some data, replication doesn't help since the deletion will have also been propagated to the replicas, so you need a backup if you want to restore the deleted data.

In fact, replication and backups are often complementary to each other. Backups are sometimes part of the process of setting up replication, as we shall see in [“Setting Up New Followers”](#). Conversely, archiving replication logs can be part of a backup process.

Some databases internally maintain immutable snapshots of past states, which serve as a kind of internal backup. However, this means keeping old versions of the data on the same storage media as the current state. If you have a large amount of data, it can be cheaper to keep the backups of old data in an object store that is optimized for infrequently-accessed data, and to store only the current state of the database in primary storage.

Single-Leader Replication

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 is called *leader-based replication*, *primary-backup*, or *active/passive*. It works as follows (see [Figure 6-1](#)):

1. One of the replicas is designated the *leader* (also known as *primary* or *source* [2]). When clients want to write to the database, they must send their requests to the leader, which first writes the new data to its local storage.
2. The other replicas are known as *followers* (*read replicas*, *secondaries*, or *hot standbys*). Whenever the leader writes new data to its local storage, it also sends the data change to all of its followers as part of a *replication log* or *change stream*. Each follower takes the log from the leader and updates its local copy of the database accordingly, by applying all writes in the same order as they were processed on the leader.

3. When a client wants to read from the database, it can query either the leader or any of the followers. However, writes are only accepted on the leader (the followers are read-only from the client's point of view).

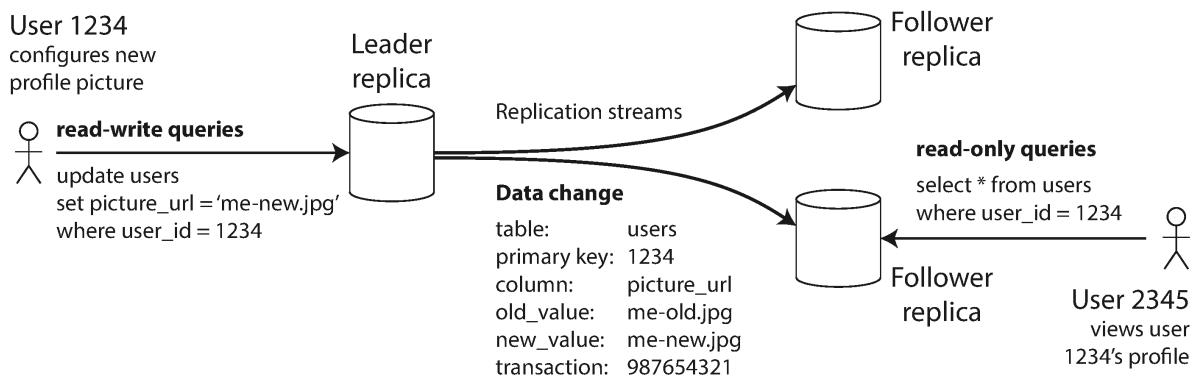


Figure 6-1. Single-leader replication directs all writes to a designated leader, which sends a stream of changes to the follower replicas.

If the database is sharded (see [Chapter 7](#)), each shard has one leader. Different shards may have their leaders on different nodes, but each shard must nevertheless have one leader node. In [**“Multi-Leader Replication”**](#) we will discuss an alternative model in which a system may have multiple leaders for the same shard at the same time.

Single-leader replication is very widely used. It's a built-in feature of many relational databases, such as PostgreSQL, MySQL, Oracle Data Guard [3], and SQL Server's Always On Availability Groups [4]. It is also used in some document databases such as MongoDB and DynamoDB [5], message

brokers such as Kafka, replicated block devices such as DRBD, and some network filesystems. Many consensus algorithms such as Raft, which is used for replication in CockroachDB [6], TiDB [7], etcd, and RabbitMQ quorum queues (among others), are also based on a single leader, and automatically elect a new leader if the old one fails (we will discuss consensus in more detail in [Link to Come]).

NOTE

In older documents you may see the term *master–slave replication*. It means the same as leader-based replication, but the term should be avoided as it is widely considered offensive [8].

Synchronous Versus Asynchronous Replication

An important detail of a replicated system is whether the replication happens *synchronously* or *asynchronously*. (In relational databases, this is often a configurable option; other systems are often hardcoded to be either one or the other.)

Think about what happens in [Figure 6-1](#), where the user of a website updates their profile image. At some point in time, the client sends the update request to the leader; shortly afterward,

it is received by the leader. At some point, the leader forwards the data change to the followers. Eventually, the leader notifies the client that the update was successful. [Figure 6-2](#) shows one possible way how the timings could work out.

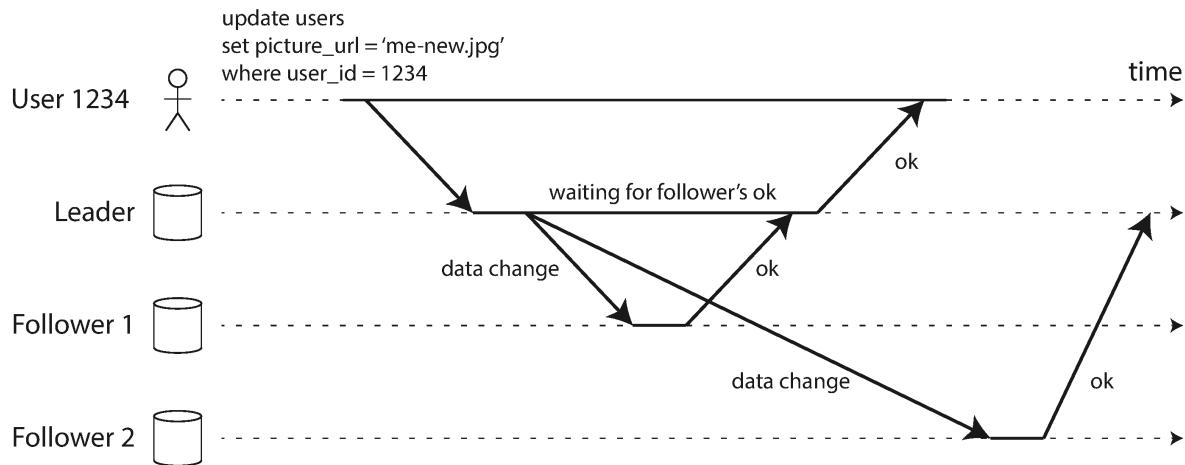


Figure 6-2. Leader-based replication with one synchronous and one asynchronous follower.

In the example of [Figure 6-2](#), the replication to follower 1 is *synchronous*: the leader waits until follower 1 has confirmed that it received the write before reporting success to the user, and before making the write visible to other clients. The replication to follower 2 is *asynchronous*: the leader sends the message, but doesn't wait for a response from the follower.

The diagram shows that there is a substantial delay before follower 2 processes the message. Normally, replication is quite fast: most database systems apply changes to followers in less

than a second. However, there is no guarantee of how long it might take. There are circumstances when followers might fall behind the leader by several minutes or more; for example, if a follower is recovering from a failure, if the system is operating near maximum capacity, or if there are network problems between the nodes.

The advantage of synchronous replication is that the follower is guaranteed to have an up-to-date copy of the data that is consistent with the leader. If the leader suddenly fails, we can be sure that the data is still available on the follower. The disadvantage is that if the synchronous follower doesn't respond (because it has crashed, or there is a network fault, or for any other reason), the write cannot be processed. The leader must block all writes and wait until the synchronous replica is available again.

For that reason, it is impracticable for all followers to be synchronous: any one node outage would cause the whole system to grind to a halt. In practice, if a database offers synchronous replication, it often means that *one* of the followers is synchronous, and the others are asynchronous. If the synchronous follower becomes unavailable or slow, one of the asynchronous followers is made synchronous. This guarantees that you have an up-to-date copy of the data on at

least two nodes: the leader and one synchronous follower. This configuration is sometimes also called *semi-synchronous*.

In some systems, a *majority* (e.g., 3 out of 5 replicas, including the leader) of replicas is updated synchronously, and the remaining minority is asynchronous. This is an example of a *quorum*, which we will discuss further in “[Quorums for reading and writing](#)”. Majority quorums are often used in systems that use a consensus protocol for automatic leader election, which we will return to in [Link to Come].

Sometimes, leader-based replication is configured to be completely asynchronous. In this case, if the leader fails and is not recoverable, any writes that have not yet been replicated to followers are lost. This means that a write is not guaranteed to be durable, even if it has been confirmed to the client. However, a fully asynchronous configuration has the advantage that the leader can continue processing writes, even if all of its followers have fallen behind.

Weakening durability may sound like a bad trade-off, but asynchronous replication is nevertheless widely used, especially if there are many followers or if they are geographically distributed [9]. We will return to this issue in “[Problems with Replication Lag](#)”.

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 Percona XtraBackup for MySQL.
2. Copy the snapshot to the new follower node.

3. The follower connects to the leader and requests all the data changes that have happened since the snapshot was taken. This requires that the snapshot is associated with an exact position in the leader's replication log. That position has various names: for example, PostgreSQL calls it the *log sequence number*; MySQL has two mechanisms, *binlog coordinates* and *global transaction identifiers* (GTIDs).
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.

You can also archive the replication log to an object store; along with periodic snapshots of the whole database in the object store this is a good way of implementing database backups and disaster recovery. You can also perform steps 1 and 2 of setting up a new follower by downloading those files from the object store. For example, WAL-G does this for PostgreSQL, MySQL, and SQL Server, and Litestream does the equivalent for SQLite.

DATABASES BACKED BY OBJECT STORAGE

Object storage can be used for more than archiving data. Many databases are beginning to use object stores such as Amazon Web Services S3, Google Cloud Storage, and Azure Blob Storage to serve data for live queries. Storing database data in object storage has many benefits:

- Object storage is inexpensive compared to other cloud storage options, which allow cloud databases to store less-often queried data on cheaper, higher-latency storage while serving the working set from memory, SSDs, and NVMe.
- Object stores also provide multi-zone, dual-region, or multi-region replication with very high durability guarantees. This also allows databases to bypass inter-zone network fees.
- Databases can use an object store's *conditional write* feature—essentially, a *compare-and-swap* (CAS) operation—to implement transactions and leadership election [10].
- Storing data from multiple databases in the same object store can simplify data integration, particularly when open formats such as Apache Parquet and Apache Iceberg are used.

These benefits dramatically simplify the database architecture by shifting the responsibility of transactions, leadership election, and replication to object storage.

Systems that adopt object storage for replication must grapple with some tradeoffs. Notably, object stores have much higher read and write latencies than local disks or virtual block devices such as EBS. Many cloud providers also charge a per-API call fee, which forces systems to batch reads and writes to reduce cost. Such batching further increases latency. Moreover, many object stores do not offer standard filesystem interfaces. This prevents systems that lack object storage integration from leveraging object storage. Interfaces such as *filesystem in userspace* (FUSE) allow operators to mount object store buckets as filesystems that applications can use without knowing their data is stored on object storage. Still, many FUSE interfaces to object stores lack POSIX features such as non-sequential writes or symlinks, which systems might depend on.

Different systems deal with these trade-offs in various ways. Some introduce a *tiered storage* architecture that places less frequently accessed data on object storage while new or frequently accessed data is kept on faster storage devices such as SSDs, NVMe, or even in memory. Other systems use object storage as their primary storage tier, but use a separate low-latency storage system such as Amazon's EBS or Neon's Safekeepers [11]) to store their WAL. Recently, some systems have gone even farther by adopting a *zero-disk architecture* (ZDA). ZDA-based systems persist all data to object storage and

use disks and memory strictly for caching. This allows nodes to have no persistent state, which dramatically simplifies operations. WarpStream, Confluent Freight, Buf's Bufstream, and Redpanda Serverless are all Kafka-compatible systems built using a zero-disk architecture. Nearly every modern cloud data warehouse also adopts such an architecture, as does Turbopuffer (a vector search engine), and SlateDB (a cloud-native LSM storage engine).

Handling Node Outages

Any node in the system can go down, perhaps unexpectedly due to a fault, but just as likely due to planned maintenance (for example, rebooting a machine to install a kernel security patch). Being able to reboot individual nodes without downtime is a big advantage for operations and maintenance. Thus, our goal is to keep the system as a whole running despite individual node failures, and to keep the impact of a node outage as small as possible.

How do you achieve high availability with leader-based replication?

Follower failure: Catch-up recovery

On its local disk, each follower keeps a log of the data changes it has received from the leader. If a follower crashes and is restarted, or if the network between the leader and the follower is temporarily interrupted, the follower can recover quite easily: from its log, it knows the last transaction that was processed before the fault occurred. Thus, the follower can connect to the leader and request all the data changes that occurred during the time when the follower was disconnected. When it has applied these changes, it has caught up to the leader and can continue receiving a stream of data changes as before.

Although follower recovery is conceptually simple, it can be challenging in terms of performance: if the database has a high write throughput or if the follower has been offline for a long time, there might be a lot of writes to catch up on. There will be high load on both the recovering follower and the leader (which needs to send the backlog of writes to the follower) while this catch-up is ongoing.

The leader can delete its log of writes once all followers have confirmed that they have processed it, but if a follower is unavailable for a long time, the leader faces a choice: either it

retains the log until the follower recovers and catches up (at the risk of running out of disk space on the leader), or it deletes the log that the unavailable follower has not yet acknowledged (in which case the follower won't be able to recover from the log, and will have to be restored from a backup when it comes back).

Leader failure: Failover

Handling a failure of the leader is trickier: one of the followers needs to be promoted to be the new leader, clients need to be reconfigured to send their writes to the new leader, and the other followers need to start consuming data changes from the new leader. This process is called *failover*.

Failover can happen manually (an administrator is notified that the leader has failed and takes the necessary steps to make a new leader) or automatically. An automatic failover process usually consists of the following steps:

1. *Determining that the leader has failed.* There are many things that could potentially go wrong: crashes, power outages, network issues, and more. There is no foolproof way of detecting what has gone wrong, so most systems simply use a timeout: nodes frequently bounce messages back and forth

between each other, and if a node doesn't respond for some period of time—say, 30 seconds—it is assumed to be dead. (If the leader is deliberately taken down for planned maintenance, this doesn't apply.)

2. *Choosing a new leader.* This could be done through an election process (where the leader is chosen by a majority of the remaining replicas), or a new leader could be appointed by a previously established *controller node* [12]. 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 [Link to Come].
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](#)”). If the old leader comes back, it might still believe that it is the leader, not realizing that the other replicas have forced it to step down. The system needs to ensure that the old leader becomes a follower and recognizes the new leader.

Failover is fraught with things that can go wrong:

- If asynchronous replication is used, the new leader may not have received all the writes from the old leader before it failed. If the former leader rejoins the cluster after a new

leader has been chosen, what should happen to those writes? The new leader may have received conflicting writes in the meantime. The most common solution is for the old leader's unreplicated writes to simply be discarded, which means that writes you believed to be committed actually weren't durable after all.

- Discarding writes is especially dangerous if other storage systems outside of the database need to be coordinated with the database contents. For example, in one incident at GitHub [13], an out-of-date MySQL follower was promoted to leader. The database used an autoincrementing counter to assign primary keys to new rows, but because the new leader's counter lagged behind the old leader's, it reused some primary keys that were previously assigned by the old leader. These primary keys were also used in a Redis store, so the reuse of primary keys resulted in inconsistency between MySQL and Redis, which caused some private data to be disclosed to the wrong users.
- In certain fault scenarios (see [Link to Come]), 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](#)”), 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. However, if this mechanism is not carefully designed, you can end up with both nodes being shut down [14]. Moreover, there is a risk that by the time the split brain is detected and the old node is shut down, it is already too late and data has already been corrupted.

- What is the right timeout before the leader is declared dead? A longer timeout means a longer time to recovery in the case where the leader fails. However, if the timeout is too short, there could be unnecessary failovers. For example, a temporary load spike could cause a node's response time to increase above the timeout, or a network glitch could cause delayed packets. If the system is already struggling with high load or network problems, an unnecessary failover is likely to make the situation worse, not better.

NOTE

Guarding against split brain by limiting or shutting down old leaders is known as *fencing* or, more emphatically, *Shoot The Other Node In The Head* (STONITH). We will discuss fencing in more detail in [Link to Come].

There are no easy solutions to these problems. For this reason, some operations teams prefer to perform failovers manually, even if the software supports automatic failover.

The most important thing with failover is to pick an up-to-date follower as the new leader—if synchronous or semi-synchronous replication is used, this would be the follower that the old leader waited for before acknowledging writes. With asynchronous replication, you can pick the follower with the greatest log sequence number. This minimizes the amount of data that is lost during failover: losing a fraction of a second of writes may be tolerable, but picking a follower that is behind by several days could be catastrophic.

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 [Link to Come] and [Link to Come] 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 nondeterministic function, such as `NOW()` to get the current date and time or `RAND()` to get a random number, is likely to generate a different value on each replica.
- If statements use an autoincrementing column, or if they depend on the existing data in the database (e.g., `UPDATE ... WHERE <some condition>`), they must be executed in exactly the same order on each replica, or else they may have a different effect. This can be limiting when there are multiple concurrently executing transactions.
- Statements that have side effects (e.g., triggers, stored procedures, user-defined functions) may result in different

side effects occurring on each replica, unless the side effects are absolutely deterministic.

It is possible to work around those issues—for example, the leader can replace any nondeterministic function calls with a fixed return value when the statement is logged so that the followers all get the same value. The idea of executing deterministic statements in a fixed order is similar to the event sourcing model that we previously discussed in [“Event Sourcing and CQRS”](#). This approach is also known as *state machine replication*, and we will discuss the theory behind it in [Link to Come].

Statement-based replication was used in MySQL before version 5.1. It is still sometimes used today, as it is quite compact, but by default MySQL now switches to row-based replication (discussed shortly) if there is any nondeterminism in a statement. VoltDB uses statement-based replication, and makes it safe by requiring transactions to be deterministic [15]. However, determinism can be hard to guarantee in practice, so many databases prefer other replication methods.

Write-ahead log (WAL) shipping

In [Chapter 4](#) we saw that a write-ahead log is needed to make B-tree storage engines robust: every modification is first written to the WAL so that the tree can be restored to a consistent state after a crash. Since the WAL contains all the information necessary to restore the indexes and heap into a consistent state, 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 files as found on the leader.

This method of replication is used in PostgreSQL and Oracle, among others [[16](#), [17](#)]. The main disadvantage is that the log describes the data on a very low level: a WAL contains details of which bytes were changed in which disk blocks. This makes replication tightly coupled to the storage engine. If the database changes its storage format from one version to another, it is typically not possible to run different versions of the database software on the leader and the followers.

That may seem like a minor implementation detail, but it can have a big operational impact. If the replication protocol allows the follower to use a newer software version than the leader,

you can perform a zero-downtime upgrade of the database software by first upgrading the followers and then performing a failover to make one of the upgraded nodes the new leader. If the replication protocol does not allow this version mismatch, as is often the case with WAL shipping, such upgrades require downtime.

Logical (row-based) log replication

An alternative is to use different log formats for replication and for the storage engine, which allows the replication log to be decoupled from the storage engine internals. This kind of replication log is called a *logical log*, to distinguish it from the storage engine's (*physical*) data representation.

A logical log for a relational database is usually a sequence of records describing writes to database tables at the granularity of a row:

- For an inserted row, the log contains the new values of all columns.
- For a deleted row, the log contains enough information to uniquely identify the row that was deleted. Typically this would be the primary key, but if there is no primary key on the table, the old values of all columns need to be logged.

- For an updated row, the log contains enough information to uniquely identify the updated row, and the new values of all columns (or at least the new values of all columns that changed).

A transaction that modifies several rows generates several such log records, followed by a record indicating that the transaction was committed. MySQL keeps a separate logical replication log, called the *binlog*, in addition to the WAL (when configured to use row-based replication). PostgreSQL implements logical replication by decoding the physical WAL into row insertion/update/delete events [18].

Since a logical log is decoupled from the storage engine internals, it can more easily be kept backward compatible, allowing the leader and the follower to run different versions of the database software. This in turn enables upgrading to a new version with minimal downtime [19].

A logical log format is also easier for external applications to parse. This aspect is useful if you want to send the contents of a database to an external system, such as a data warehouse for offline analysis, or for building custom indexes and caches [20]. This technique is called *change data capture*, and we will return to it in [Link to Come].

Problems with Replication Lag

Being able to tolerate node failures is just one reason for wanting replication. As mentioned in [“Distributed versus Single-Node Systems”](#), other reasons are scalability (processing more requests than a single machine can handle) and latency (placing replicas geographically closer to users).

Leader-based replication requires all writes to go through a single node, but read-only queries can go to any replica. For workloads that consist of mostly reads and only a small percentage of writes (which is often the case with online services), there is an attractive option: create many followers, and distribute the read requests across those followers. This removes load from the leader and allows read requests to be served by nearby replicas.

In this *read-scaling* architecture, you can increase the capacity for serving read-only requests simply by adding more followers. However, this approach only realistically works with asynchronous replication—if you tried to synchronously replicate to all followers, a single node failure or network outage would make the entire system unavailable for writing. And the more nodes you have, the likelier it is that one will be

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

Unfortunately, if an application reads from an *asynchronous* follower, it may see outdated information if the follower has fallen behind. This leads to apparent inconsistencies in the database: if you run the same query on the leader and a follower at the same time, you may get different results, because not all writes have been reflected in the follower. This inconsistency is just a temporary state—if you stop writing to the database and wait a while, the followers will eventually catch up and become consistent with the leader. For that reason, this effect is known as *eventual consistency* [[21](#)].

NOTE

The term *eventual consistency* was coined by Douglas Terry et al. [[22](#)], popularized by Werner Vogels [[23](#)], 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.

The term “eventually” is deliberately vague: in general, there is no limit to how far a replica can fall behind. In normal operation, the delay between a write happening on the leader and being reflected on a follower—the *replication lag*—may be

only a fraction of a second, and not noticeable in practice. However, if the system is operating near capacity or if there is a problem in the network, the lag can easily increase to several seconds or even minutes.

When the lag is so large, the inconsistencies it introduces are not just a theoretical issue but a real problem for applications. In this section we will highlight three examples of problems that are likely to occur when there is replication lag. We'll also outline some approaches to solving them.

Reading Your Own Writes

Many applications let the user submit some data and then view what they have submitted. This might be a record in a customer database, or a comment on a discussion thread, or something else of that sort. When new data is submitted, it must be sent to the leader, but when the user views the data, it can be read from a follower. This is especially appropriate if data is frequently viewed but only occasionally written.

With asynchronous replication, there is a problem, illustrated in [Figure 6-3](#): if the user views the data shortly after making a write, the new data may not yet have reached the replica. To

the user, it looks as though the data they submitted was lost, so they will be understandably unhappy.

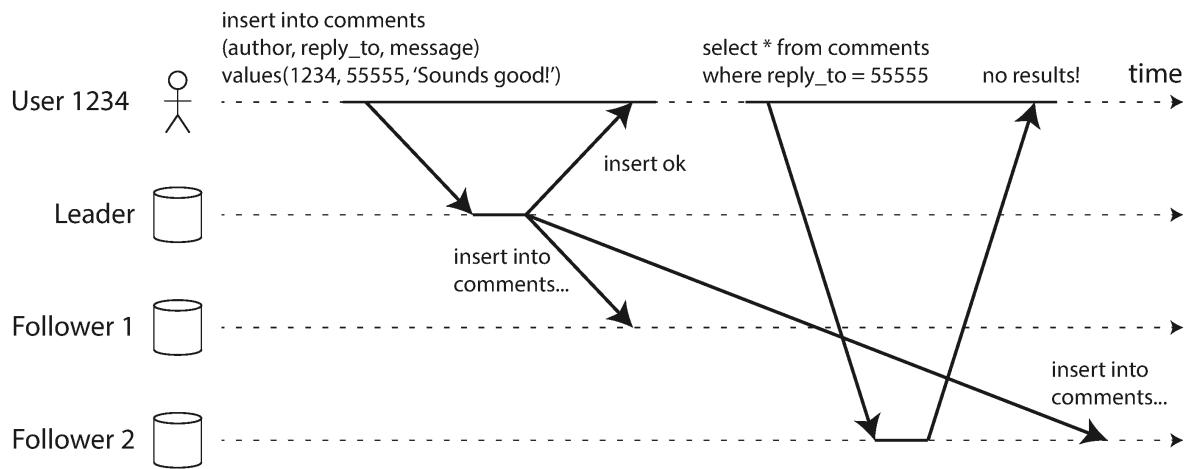


Figure 6-3. A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need read-after-write consistency.

In this situation, we need *read-after-write consistency*, also known as *read-your-writes consistency* [22]. 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 or a synchronously updated follower; otherwise, read it from an asynchronously updated follower. This requires that you have some way of knowing whether something might have been modified, without actually querying it. For example, user profile information on a social network is normally only editable by the owner of the profile, not by anybody else. Thus, a simple rule is: always read the user's own profile from the leader, and any other users' profiles from a follower.
- If most things in the application are potentially editable by the user, that approach won't be effective, as most things would have to be read from the leader (negating the benefit of read scaling). In that case, other criteria may be used to decide whether to read from the leader. For example, you could track the time of the last update and, for one minute after the last update, make all reads from the leader [24]. You could also monitor the replication lag on followers and prevent queries on any follower that is more than one minute behind the leader.
- The client can remember the timestamp of its most recent write—then the system can ensure that the replica serving any reads for that user reflects updates at least until that timestamp. If a replica is not sufficiently up to date, either the

read can be handled by another replica or the query can wait until the replica has caught up [25]. 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 [Link to Come]).

- If your replicas are distributed across regions (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 region that contains the leader.

Another complication arises when the same user is accessing your service from multiple devices, for example a desktop web browser and a mobile app. In this case you may want to provide *cross-device* read-after-write consistency: if the user enters some information on one device and then views it on another device, they should see the information they just entered.

In this case, there are some additional issues to consider:

- Approaches that require remembering the timestamp of the user's last update become more difficult, because the code running on one device doesn't know what updates have

happened on the other device. This metadata will need to be centralized.

- If your replicas are distributed across different regions, there is no guarantee that connections from different devices will be routed to the same region. (For example, if the user's desktop computer uses the home broadband connection and their mobile device uses the cellular data network, the devices' network routes may be completely different.) If your approach requires reading from the leader, you may first need to route requests from all of a user's devices to the same region.

REGIONS AND AVAILABILITY ZONES

We use the term *region* to refer to one or more datacenters in a single geographic location. Cloud providers locate multiple datacenters in the same geographic region. Each datacenter is referred to as an *availability zone* or simply *zone*. Thus, a single cloud region is made up of multiple zones. Each zone is a separate datacenter located in separate physical facility with its own power, cooling, and so on.

Zones in the same region are connected by very high speed network connections. Latency is low enough that most distributed systems can run with nodes spread across multiple zones in the same region as though they were in a single zone. Multi-zone configurations allow distributed systems to survive zonal outages where one zone goes offline, but they do not protect against regional outages where all zones in a region are unavailable. To survive a regional outage, a distributed system must be deployed across multiple regions, which can result in higher latencies, lower throughput, and increased cloud networking bills. We will discuss these tradeoffs more in “[Multi-leader replication topologies](#)”. For now, just know that when we say region, we mean a collection of zones/datacenters in a single geographic location.

Monotonic Reads

Our second example of an anomaly that can occur when reading from asynchronous followers is that it's possible for a user to see things *moving backward in time*.

This can happen if a user makes several reads from different replicas. For example, [Figure 6-4](#) 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 observes the system state at an earlier point in time than the first query. This wouldn't be so bad if the first query hadn't returned anything, because user 2345 probably wouldn't know that user 1234 had recently added a comment. However, it's very confusing for user 2345 if they first see user 1234's comment appear, and then see it disappear again.

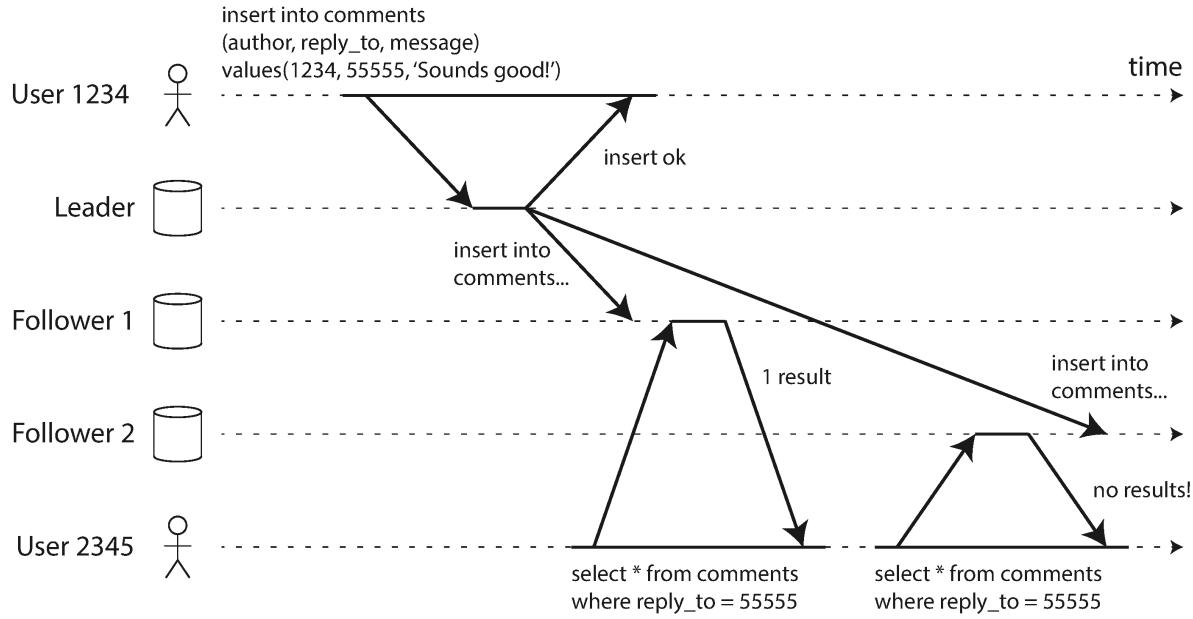


Figure 6-4. A user first reads from a fresh replica, then from a stale replica. Time appears to go backward. To prevent this anomaly, we need monotonic reads.

Monotonic reads [21] is a guarantee that this kind of anomaly does not happen. It's a lesser guarantee than strong consistency, but a stronger guarantee than eventual consistency. When you read data, you may see an old value; monotonic reads only means that if one user makes several reads in sequence, they will not see time go backward—i.e., they will not read older data after having previously read newer data.

One way of achieving monotonic reads is to make sure that each user always makes their reads from the same replica (different users can read from different replicas). For example, the replica can be chosen based on a hash of the user ID, rather

than randomly. However, if that replica fails, the user's queries will need to be rerouted to another replica.

Consistent Prefix Reads

Our third example of replication lag anomalies concerns violation of causality. Imagine the following short dialog between Mr. Poons and Mrs. Cake:

Mr. Poons

How far into the future can you see, Mrs. Cake?

Mrs. Cake

About ten seconds usually, Mr. Poons.

There is a causal dependency between those two sentences: Mrs. Cake heard Mr. Poons's question and answered it.

Now, imagine a third person is listening to this conversation through followers. The things said by Mrs. Cake go through a follower with little lag, but the things said by Mr. Poons have a longer replication lag (see [Figure 6-5](#)). This observer would hear the following:

Mrs. Cake

About ten seconds usually, Mr. Poons.

Mr. Poons

How far into the future can you see, Mrs. Cake?

To the observer it looks as though Mrs. Cake is answering the question before Mr. Poons has even asked it. Such psychic powers are impressive, but very confusing [26].

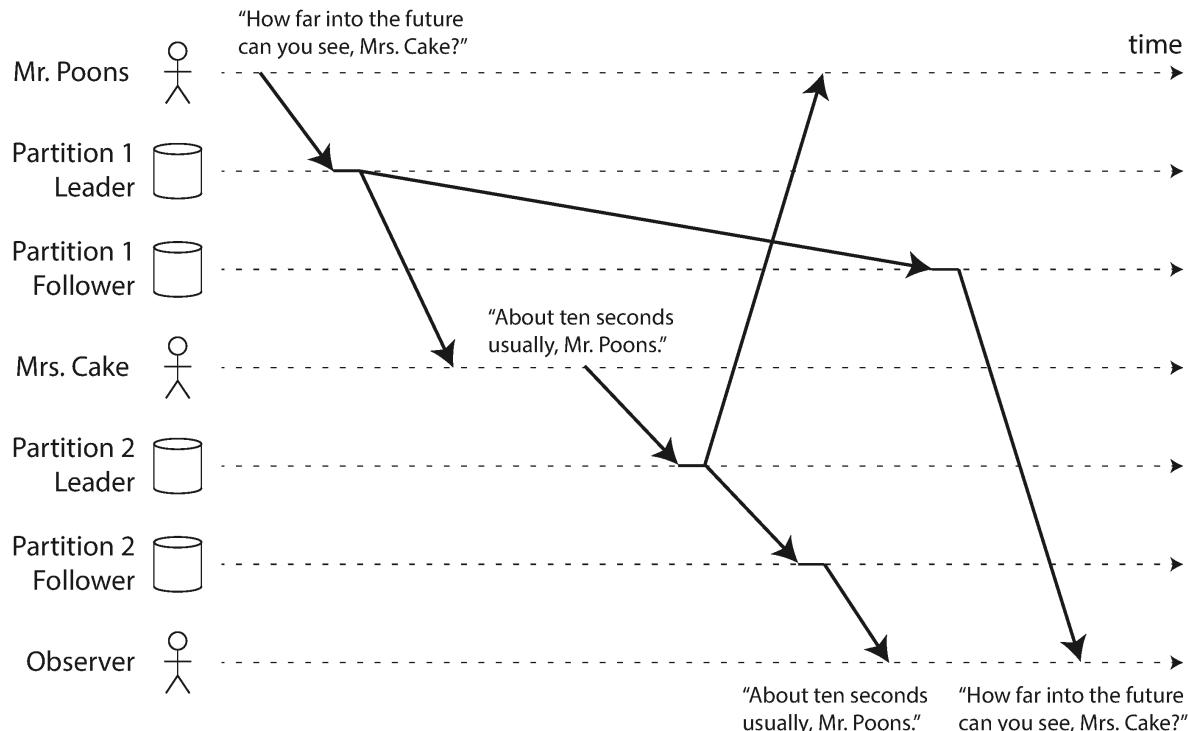


Figure 6-5. If some shards are replicated slower than others, an observer may see the answer before they see the question.

Preventing this kind of anomaly requires another type of guarantee: *consistent prefix reads* [21]. 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 sharded (partitioned) databases, which we will discuss in [Chapter 7](#). 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 shards operate independently, so there is no global ordering of writes: when a user reads from the database, they may see some parts of the database in an older state and some in a newer state.

One solution is to make sure that any writes that are causally related to each other are written to the same shard—but in some applications that cannot be done efficiently. There are also algorithms that explicitly keep track of causal dependencies, a topic that we will return to in [“The ‘happens-before’ relation and concurrency”](#).

Solutions for Replication Lag

When working with an eventually consistent system, it is worth thinking about how the application behaves if the replication lag increases to several minutes or even hours. If the answer is “no problem,” that’s great. However, if the result is a bad

experience for users, it's important to design the system to provide a stronger guarantee, such as read-after-write.

Pretending that replication is synchronous when in fact it is asynchronous is a recipe for problems down the line.

As discussed earlier, there are ways in which an application can provide a stronger guarantee than the underlying database—for example, by performing certain kinds of reads on the leader or a synchronously updated follower. However, dealing with these issues in application code is complex and easy to get wrong.

The simplest programming model for application developers is to choose a database that provides a strong consistency guarantee for replicas such as linearizability (see [Link to Come]), and ACID transactions (see [Chapter 8](#)). This allows you to mostly ignore the challenges that arise from replication, and treat the database as if it had just a single node. In the early 2010s the *NoSQL* movement promoted the view that these features limited scalability, and that large-scale systems would have to embrace eventual consistency.

However, since then, a number of databases started providing strong consistency and transactions while also offering the fault tolerance, high availability, and scalability advantages of a

distributed database. As mentioned in “[Relational Model versus Document Model](#)”, this trend is known as *NewSQL* to contrast with NoSQL (although it’s less about SQL specifically, and more about new approaches to scalable transaction management).

Even though scalable, strongly consistent distributed databases are now available, there are still good reasons why some applications choose to use different forms of replication that offer weaker consistency guarantees: they can offer stronger resilience in the face of network interruptions, and have lower overheads compared to transactional systems. We will explore such approaches in the rest of this chapter.

Multi-Leader Replication

So far in this chapter we have only considered replication architectures using a single leader. Although that is a common approach, there are interesting alternatives.

Single-leader replication has one major downside: all writes must go through the one leader. 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 single-leader 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 *active/active* or *bidirectional* replication). In this setup, each leader simultaneously acts as a follower to the other leaders.

As with single-leader replication, there is a choice between making it synchronous or asynchronous. Let's say you have two leaders, *A* and *B*, and you're trying to write to *A*. If writes are synchronously replicated from *A* to *B*, and the network between the two nodes is interrupted, you can't write to *A* until the network comes back. Synchronous multi-leader replication thus gives you a model that is very similar to single-leader replication, i.e. if you had made *B* the leader and *A* simply forwards any write requests to *B* to be executed.

For that reason, we won't go further into synchronous multi-leader replication, and simply treat it as equivalent to single-leader replication. The rest of this section focusses on asynchronous multi-leader replication, in which any leader can process writes even when its connection to the other leaders is interrupted.

Geographically Distributed Operation

It rarely makes sense to use a multi-leader setup within a single region, because the benefits rarely outweigh the added complexity. However, there are some situations in which this configuration is reasonable.

Imagine you have a database with replicas in several different regions (perhaps so that you can tolerate the failure of an entire region, or perhaps in order to be closer to your users). This is known as a *geographically distributed*, *geo-distributed* or *geo-replicated* setup. With single-leader replication, the leader has to be in *one* of the regions, and all writes must go through that region.

In a multi-leader configuration, you can have a leader in *each* region. [Figure 6-6](#) shows what this architecture might look like. Within each region, regular leader–follower replication is used (with followers maybe in a different availability zone from the leader); between regions, each region’s leader replicates its changes to the leaders in other regions.

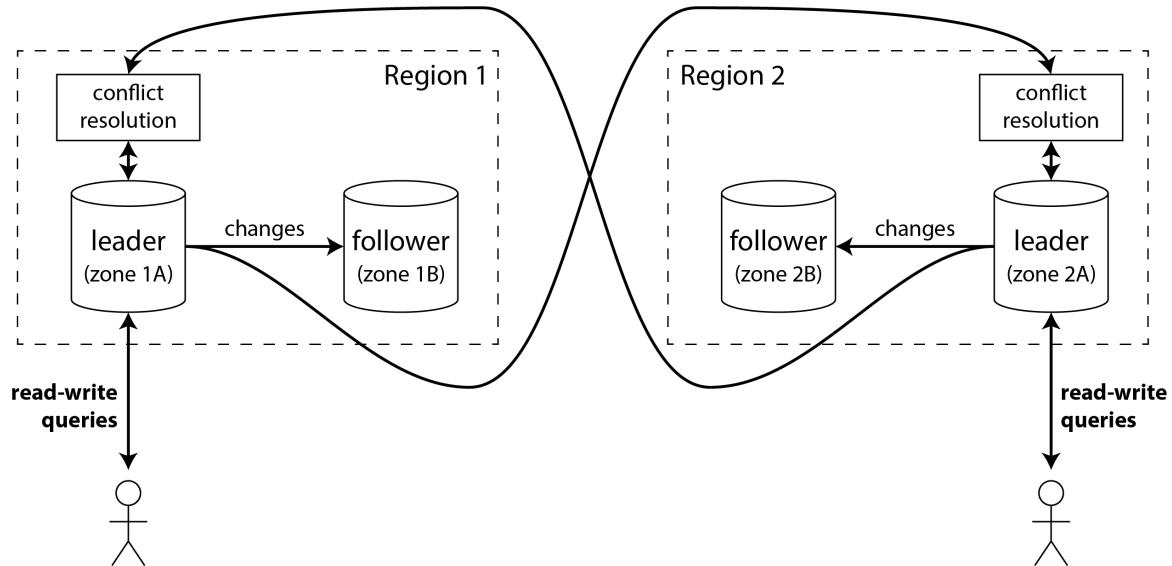


Figure 6-6. Multi-leader replication across multiple regions.

Let's compare how the single-leader and multi-leader configurations fare in a multi-region deployment:

Performance

In a single-leader configuration, every write must go over the internet to the region with the leader. This can add significant latency to writes and might contravene the purpose of having multiple regions in the first place. In a multi-leader configuration, every write can be processed in the local region and is replicated asynchronously to the other regions. Thus, the inter-region network delay is hidden from users, which means the perceived performance may be better.

Tolerance of regional outages

In a single-leader configuration, if the region with the leader becomes unavailable, failover can promote a follower in another region to be leader. In a multi-leader configuration, each region can continue operating independently of the others, and replication catches up when the offline region comes back online.

Tolerance of network problems

Even with dedicated connections, traffic between regions can be less reliable than traffic between zones in the same region or within a single zone. A single-leader configuration is very sensitive to problems in this inter-region link, because when a client in one region wants to write to a leader in another region, it has to send its request over that link and wait for the response before it can complete.

A multi-leader configuration with asynchronous replication can tolerate network problems better: during a temporary network interruption, each region's leader can continue independently processing writes.

Consistency

A single-leader system can provide strong consistency guarantees, such as serializable transactions, which we will discuss in [Chapter 8](#). The biggest downside of multi-leader systems is that the consistency they can achieve is much weaker. For example, you can't guarantee that a bank account won't go negative or that a username is unique: it's always possible for different leaders to process writes that are individually fine (paying out some of the money in an account, registering a particular username), but which violate the constraint when taken together with another write on another leader.

This is simply a fundamental limitation of distributed systems [\[27\]](#). If you need to enforce such constraints, you're therefore better off with a single-leader system. However, as we will see in [“Dealing with Conflicting Writes”](#), multi-leader systems can still achieve consistency properties that are useful in a wide range of apps that don't need such constraints.

Multi-leader replication is less common than single-leader replication, but it is still supported by many databases, including MySQL, Oracle, SQL Server, and YugabyteDB. In some cases it is an external add-on feature, for example in Redis Enterprise, EDB Postgres Distributed, and pglogical [\[28\]](#).

As multi-leader replication is a somewhat retrofitted feature in many databases, there are often subtle configuration pitfalls and surprising interactions with other database features. For example, autoincrementing keys, triggers, and integrity constraints can be problematic. For this reason, multi-leader replication is often considered dangerous territory that should be avoided if possible [29].

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 6-9](#), 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 6-7](#).

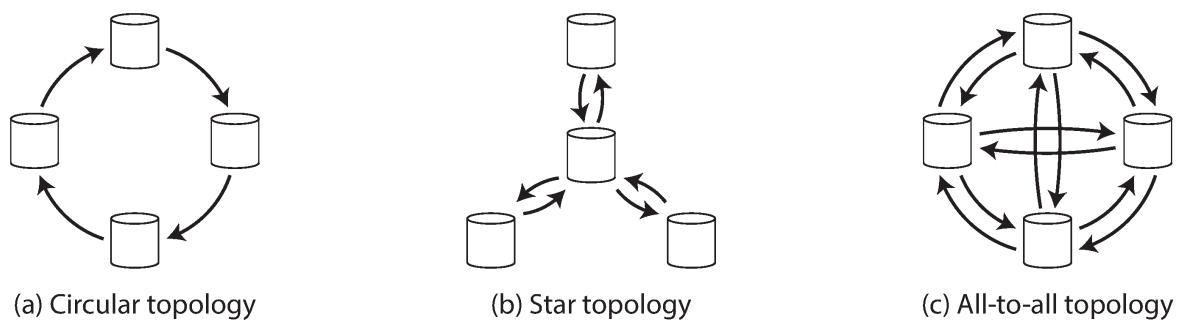


Figure 6-7. Three example topologies in which multi-leader replication can be set up.

The most general topology is *all-to-all*, shown in [Figure 6-7\(c\)](#), in which every leader sends its writes to every other leader.

However, more restricted topologies are also used: for example a *circular topology* 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*: one designated root node forwards writes to all of the other nodes. The star topology can be generalized to a tree.

NOTE

Don't confuse a star-shaped network topology with a *star schema* (see [“Stars and Snowflakes: Schemas for Analytics”](#)), which describes the structure of a data model.

In circular and star topologies, a write may need to pass through several nodes before it reaches all replicas. Therefore, nodes need to forward data changes they receive from other nodes. To prevent infinite replication loops, each node is given a unique identifier, and in the replication log, each write is tagged with the identifiers of all the nodes it has passed through [\[30\]](#). 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.

Problems with different topologies

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, leaving them 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 6-8](#).

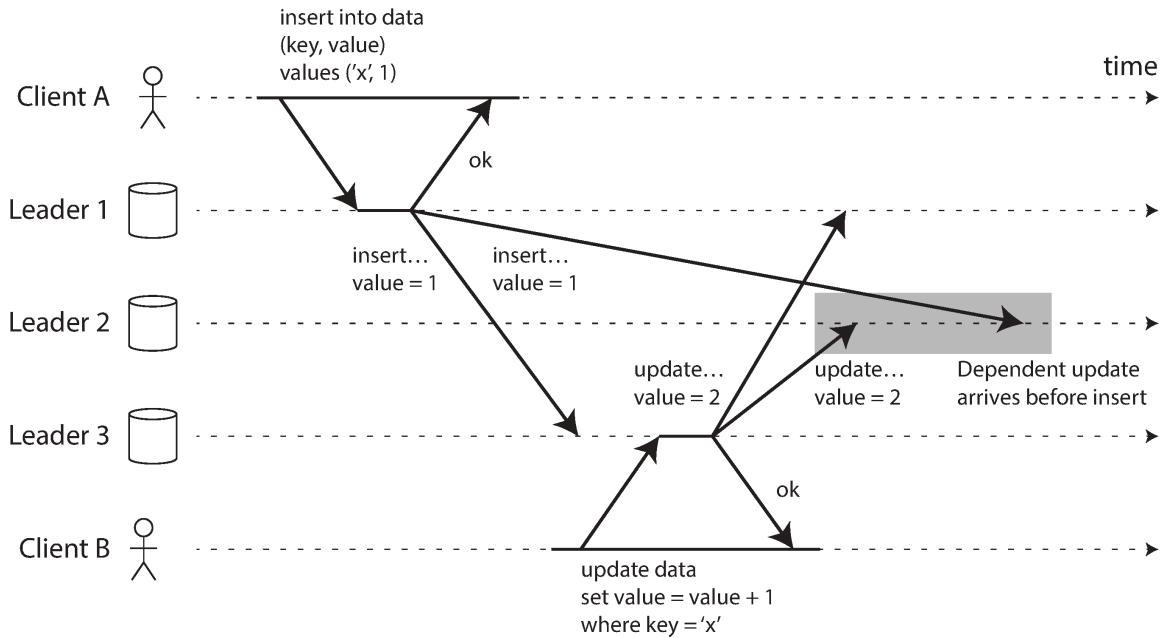


Figure 6-8. With multi-leader replication, writes may arrive in the wrong order at some replicas.

In [Figure 6-8](#), 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”](#): 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 [Link to Come]).

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](#)”). However, many multi-leader replication systems don’t use good techniques for ordering updates, leaving them vulnerable to issues like the one in [Figure 6-8](#). If you are using 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.

Sync Engines and Local-First Software

Another situation in which multi-leader replication is appropriate is if you have an application that needs to continue to work while it is disconnected from the internet.

For example, consider the calendar apps on your mobile phone, your laptop, and other devices. You need to be able to see your meetings (make read requests) and enter new meetings (make write requests) at any time, regardless of whether your device currently has an internet connection. If you make any changes

while you are offline, they need to be synced with a server and your other devices when the device is next online.

In this case, every device has a local database replica that acts as a leader (it accepts write requests), and there is an asynchronous multi-leader replication process (sync) between the replicas of your calendar on all of your devices. The replication lag may be hours or even days, depending on when you have internet access available.

From an architectural point of view, this setup is very similar to multi-leader replication between regions, taken to the extreme: each device is a “region,” and the network connection between them is extremely unreliable.

Real-time collaboration, offline-first, and local-first apps

Moreover, many modern web apps offer *real-time collaboration* features, such as Google Docs and Sheets for text documents and spreadsheets, Figma for graphics, and Linear for project management. What makes these apps so responsive is that user input is immediately reflected in the user interface, without waiting for a network round-trip to the server, and edits by one user are shown to their collaborators with low latency [[31](#), [32](#), [33](#)].

This again results in a multi-leader architecture: each web browser tab that has opened the shared file is a replica, and any updates that you make to the file are asynchronously replicated to the devices of the other users who have opened the same file. Even if the app does not allow you to continue editing a file while offline, the fact that multiple users can make edits without waiting for a response from the server already makes it multi-leader.

Both offline editing and real-time collaboration require a similar replication infrastructure: the application needs to capture any changes that the user makes to a file, and either send them to collaborators immediately (if online), or store them locally for sending later (if offline). Additionally, the application needs to receive changes from collaborators, merge them into the user's local copy of the file, and update the user interface to reflect the latest version. If multiple users have changed the file concurrently, conflict resolution logic may be needed to merge those changes.

A software library that supports this process is called a *sync engine*. Although the idea has existed for a long time, the term has recently gained attention [34, 35, 36]. An application that allows a user to continue editing a file while offline (which may be implemented using a sync engine) is called *offline-first* [37].

The term *local-first software* refers to collaborative apps that are not only offline-first, but are also designed to continue working even if the developer who made the software shuts down all of their online services [38]. This can be achieved by using a sync engine with an open standard sync protocol for which multiple service providers are available [39]. For example, Git is a local-first collaboration system (albeit one that doesn't support real-time collaboration) since you can sync via GitHub, GitLab, or any other repository hosting service.

Pros and cons of sync engines

The dominant way of building web apps today is to keep very little persistent state on the client, and to rely on making requests to a server whenever a new piece of data needs to be displayed or some data needs to be updated. In contrast, when using a sync engine, you have persistent state on the client, and communication with the server is moved into a background process. The sync engine approach has a number of advantages:

- Having the data locally means the user interface can be much faster to respond than if it had to wait for a service call to fetch some data. Some apps aim to respond to user input in the *next frame* of the graphics system, which means

rendering within 16 ms on a display with a 60 Hz refresh rate.

- Allowing users to continue working while offline is valuable, especially on mobile devices with intermittent connectivity. With a sync engine, an app doesn't need a separate offline mode: being offline is the same as having very large network delay.
- A sync engine simplifies the programming model for frontend apps, compared to performing explicit service calls in application code. Every service call requires error handling, as discussed in [“The problems with remote procedure calls \(RPCs\)”](#): for example, if a request to update data on a server fails, the user interface needs to somehow reflect that error. A sync engine allows the app to perform reads and writes on local data, which almost never fails, leading to a more declarative programming style [40].
- In order to display edits from other users in real-time, you need to receive notifications of those edits and efficiently update the user interface accordingly. A sync engine combined with a *reactive programming* model is a good way of implementing this [41].

Sync engines work best when all the data that the user may need is downloaded in advance and stored persistently on the client. This means that the data is available for offline access

when needed, but it also means that sync engines are not suitable if the user has access to a very large amount of data. For example, downloading all the files that the user themselves created is probably fine (one user generally doesn't generate that much data), but downloading the entire catalog of an e-commerce website probably doesn't make sense.

The sync engine was pioneered by Lotus Notes in the 1980s [42] (without using that term), and sync for specific apps such as calendars has also existed for a long time. Today there are a number of general-purpose sync engines, some of which use a proprietary backend service (e.g., Google Firestore, Realm, or Ditto), and some have an open source backend, making them suitable for creating local-first software (e.g., PouchDB/CouchDB, Automerge, or Yjs).

Multiplayer video games have a similar need to respond immediately to the user's local actions, and reconcile them with other players' actions received asynchronously over the network. In game development jargon the equivalent of a sync engine is called *netcode*. The techniques used in netcode are quite specific to the requirements of games [43], and don't directly carry over to other types of software, so we won't consider them further in this book.

Dealing with Conflicting Writes

The biggest problem with multi-leader replication—both in a geo-distributed server-side database and a local-first sync engine on end user devices—is that concurrent writes on different leaders can lead to conflicts that need to be resolved.

For example, consider a wiki page that is simultaneously being edited by two users, as shown in [Figure 6-9](#). User 1 changes the title of the page from A to B, and user 2 independently changes the title from A to C. Each user's change is successfully applied to their local leader. However, when the changes are asynchronously replicated, a conflict is detected. This problem does not occur in a single-leader database.

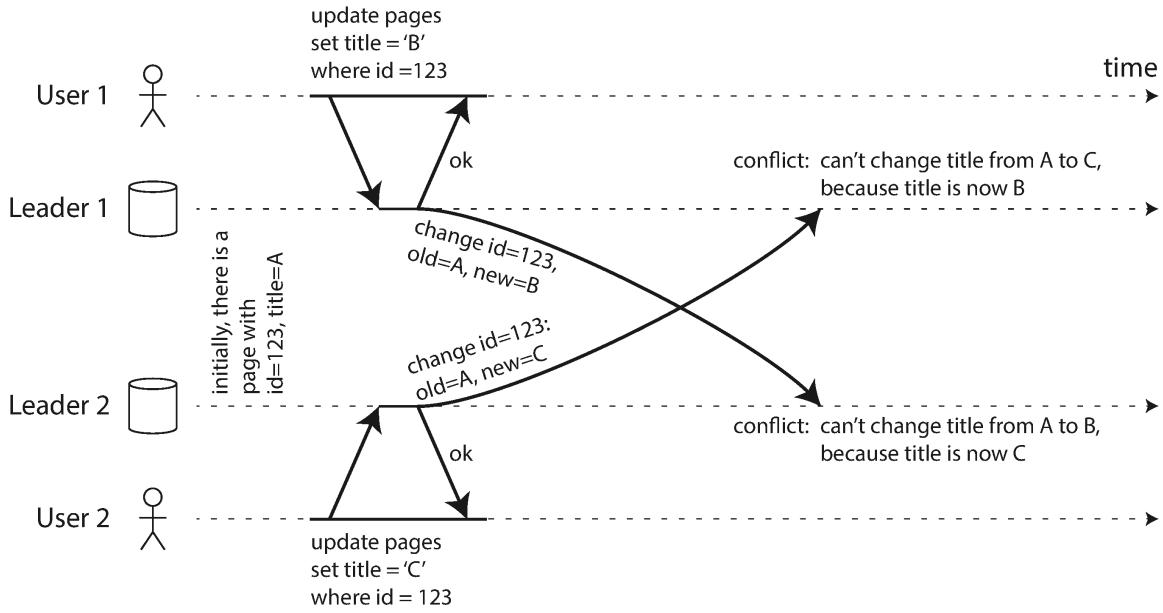


Figure 6-9. A write conflict caused by two leaders concurrently updating the same record.

NOTE

We say that the two writes in [Figure 6-9](#) are *concurrent* because neither was “aware” of the other at the time the write was originally made. It doesn’t matter whether the writes literally happened at the same time; indeed, if the writes were made while offline, they might have actually happened some time apart. What matters is whether one write occurred in a state where the other write has already taken effect.

In [Detecting Concurrent Writes](#) we will tackle the question of how a database can determine whether two writes are concurrent. For now we will assume that we can detect conflicts, and we want to figure out the best way of resolving them.

Conflict avoidance

One strategy for conflicts is to avoid them occurring in the first place. For example, if the application can ensure that all writes for a particular record go through the same leader, then conflicts cannot occur, even if the database as a whole is multi-leader. This approach is not possible in the case of a sync engine client being updated offline, but it is sometimes possible in geo-replicated server systems [29].

For example, in an application where a user can only edit their own data, you can ensure that requests from a particular user are always routed to the same region and use the leader in that region for reading and writing. Different users may have different “home” regions (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 region is unavailable and you need to reroute traffic to another region, or perhaps because a user has moved to a different location and is now closer to a different region. There is now a risk that the user performs a write while the change of designated leader is in progress, leading to a conflict that would have to be resolved

using one of the methods below. Thus, conflict avoidance breaks down if you allow the leader to be changed.

Another example of conflict avoidance: imagine you want to insert new records and generate unique IDs for them based on an auto-incrementing counter. If you have two leaders, you could set them up so that one leader only generates odd numbers and the other only generates even numbers. That way you can be sure that the two leaders won't concurrently assign the same ID to different records. Alternatively, you could use *universally unique identifiers* (UUIDs) or a distributed ID generation scheme such as Snowflake [44].

Last write wins (discarding concurrent writes)

If conflicts can't be avoided, the simplest way of resolving them is to attach a timestamp to each write, and to always use the value with the greatest timestamp. For example, in [Figure 6-9](#), let's say that the timestamp of user 1's write is greater than the timestamp of user 2's write. In that case, both leaders will determine that the new title of the page should be B, and they discard the write that sets it to C. If the writes coincidentally have the same timestamp, the winner can be chosen by comparing the values (e.g., in the case of strings, taking the one that's earlier in the alphabet).

This approach is called *last write wins* (LWW) because the write with the greatest timestamp can be considered the “last” one. The term is misleading though, because when two writes are concurrent like in [Figure 6-9](#), which one is older and which is later is undefined, and so the timestamp order of concurrent writes is essentially random.

Therefore the real meaning of LWW is: when the same record is concurrently written on different leaders, one of those writes is randomly chosen to be the winner, and the other writes are silently discarded, even though they were successfully processed at their respective leaders. This achieves the goal that eventually all replicas end up in a consistent state, but at the cost of data loss.

If you can avoid conflicts—for example, by only inserting records with a unique key such as a UUID or a Snowflake ID, and never updating them—then LWW is no problem. But if you update existing records, or if different leaders may insert records with the same key, then you have to decide whether lost updates are a problem for your application. If lost updates are not acceptable, you need to use one of the conflict resolution approaches described below.

Another problem with LWW is that if a real-time clock (e.g. a Unix timestamp) is used as timestamp for the writes, the system becomes very sensitive to clock synchronization. If one node has a clock that is ahead of the others, and you try to overwrite a value written by that node, your write may be ignored as it may have a lower timestamp, even though it clearly occurred later. This problem can be solved by using a *logical clock*, which we will discuss in [Link to Come].

Manual conflict resolution

If randomly discarding some of your writes is not desirable, the next option is to resolve the conflict manually. You may be familiar with manual conflict resolution from Git and other version control systems: if commits on two different branches edit the same lines of the same file, and you try to merge those branches, you will get a merge conflict that needs to be resolved before the merge is complete.

In a database, it would be impractical for a conflict to stop the entire replication process until a human has resolved it. Instead, databases typically store all the concurrently written values for a given record—for example, both B and C in [Figure 6-9](#). These values are sometimes called *siblings*. The next time you query that record, the database returns *all* those

values, rather than just the latest one. You can then resolve those values in whatever way you want, either automatically in application code (for example, you could concatenate B and C into “B/C”), or by asking the user. You then write back a new value to the database to resolve the conflict.

This approach to conflict resolution is used in some systems, such as CouchDB. However, it also suffers from a number of problems:

- The API of the database changes: for example, where previously the title of the wiki page was just a string, it now becomes a set of strings that usually contains one element, but may sometimes contain multiple elements if there is a conflict. This can make the data awkward to work with in application code.
- Asking the user to manually merge the siblings is a lot of work, both for the app developer (who needs to build the user interface for conflict resolution) and for the user (who may be confused about what they are being asked to do, and why). In many cases, it’s better to merge automatically than to bother the user.
- Merging siblings automatically can lead to surprising behavior if it is not done carefully. For example, the shopping cart on Amazon used to allow concurrent updates, which

were then merged by keeping all the shopping cart items that appeared in any of the siblings (i.e., taking the set union of the carts). This meant that if the customer had removed an item from their cart in one sibling, but another sibling still contained that old item, the removed item would unexpectedly reappear in the customer's cart [45]. [Figure 6-10](#) shows an example where Device 1 removes Book from the shopping cart and concurrently Device 2 removes DVD, but after merging the conflict both items reappear.

- If multiple nodes observe the conflict and concurrently resolve it, the conflict resolution process can itself introduce a new conflict. Those resolutions could even be inconsistent: for example, one node may merge B and C into “B/C” and another may merge them into “C/B” if you are not careful to order them consistently. When the conflict between “B/C” and “C/B” is merged, it may result in “B/C/C/B” or something similarly surprising.

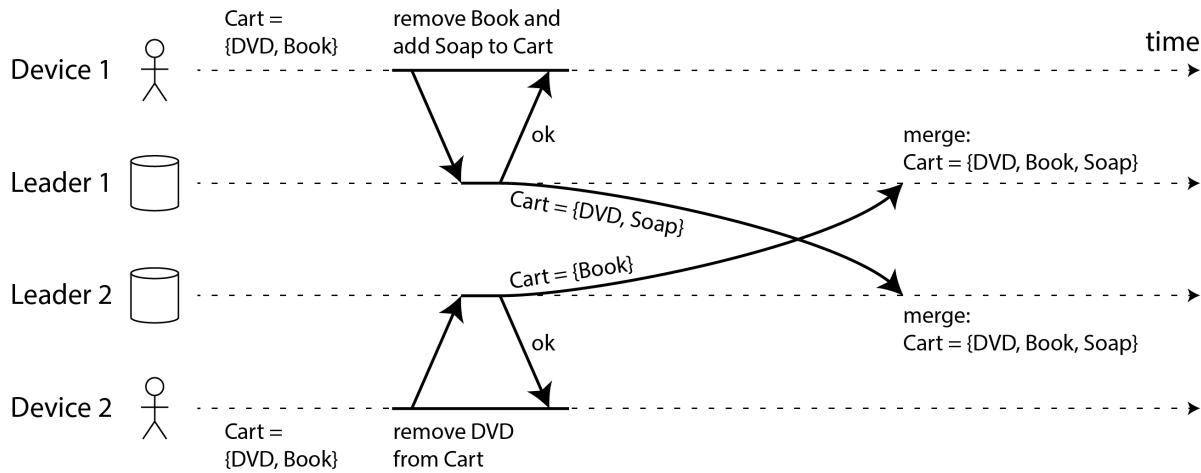


Figure 6-10. Example of Amazon's shopping cart anomaly: if conflicts on a shopping cart are merged by taking the union, deleted items may reappear.

Automatic conflict resolution

For many applications, the best way of handling conflicts is to use an algorithm that automatically merges concurrent writes into a consistent state. Automatic conflict resolution ensures that all replicas *converge* to the same state—i.e., all replicas that have processed the same set of writes have the same state, regardless of the order in which the writes arrived.

LWW is a simple example of a conflict resolution algorithm. More sophisticated merge algorithms have been developed for different types of data, with the goal of preserving the intended effect of all updates as much as possible, and hence avoiding data loss:

- If the data is text (e.g., the title or body of a wiki page), we can detect which characters have been inserted or deleted from one version to the next. The merged result then preserves all the insertions and deletions made in any of the siblings. If users concurrently insert text at the same position, it can be ordered deterministically so that all nodes get the same merged outcome.
- If the data is a collection of items (ordered like a to-do list, or unordered like a shopping cart), we can merge it similarly to text by tracking insertions and deletions. To avoid the shopping cart issue in [Figure 6-10](#), the algorithms track the fact that Book and DVD were deleted, so the merged result is Cart = {Soap}.
- If the data is an integer representing a counter that can be incremented or decremented (e.g., the number of likes on a social media post), the merge algorithm can tell how many increments and decrements happened on each sibling, and add them together correctly so that the result does not double-count and does not drop updates.
- If the data is a key-value mapping, we can merge updates to the same key by applying one of the other conflict resolution algorithms to the values under that key. Updates to different keys can be handled independently from each other.

There are limits to what is possible with conflict resolution. For example, if you want to enforce that a list contains no more than five items, and multiple users concurrently add items to the list so that there are more than five in total, your only option is to drop some of the items. Nevertheless, automatic conflict resolution is sufficient to build many useful apps. And if you start from the requirement of wanting to build a collaborative offline-first or local-first app, then conflict resolution is inevitable, and automating it is often the best approach.

CRDTs and Operational Transformation

Two families of algorithms are commonly used to implement automatic conflict resolution: *Conflict-free replicated datatypes* (CRDTs) [46] and *Operational Transformation* (OT) [47]. They have different design philosophies and performance characteristics, but both are able to perform automatic merges for all the aforementioned types of data.

[Figure 6-11](#) shows an example of how OT and a CRDT merge concurrent updates to a text. Assume you have two replicas that both start off with the text “ice”. One replica prepends the letter “n” to make “nice”, while concurrently the other replica appends an exclamation mark to make “ice!”.

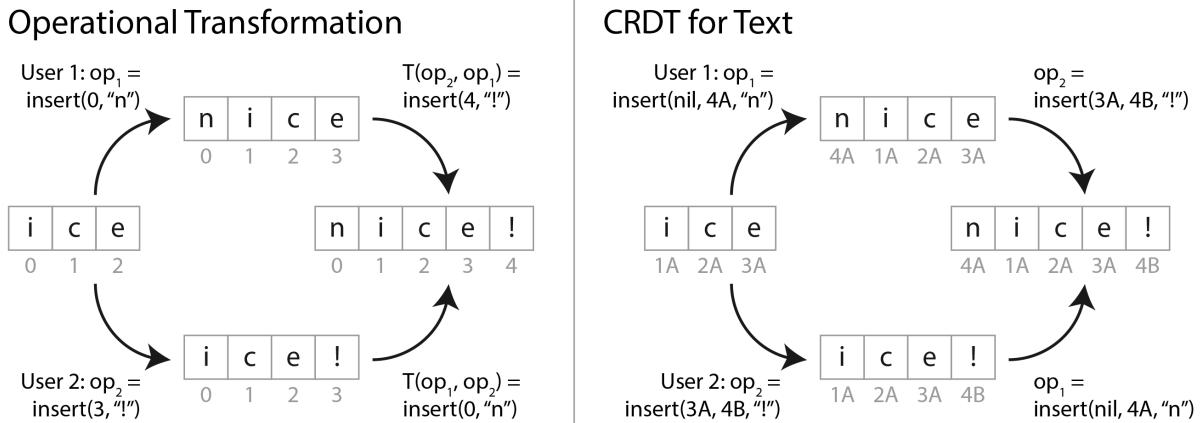


Figure 6-11. How two concurrent insertions into a string are merged by OT and a CRDT respectively.

The merged result “nice!” is achieved differently by both types of algorithms:

OT

We record the index at which characters are inserted or deleted: “n” is inserted at index 0, and “!” at index 3. Next, the replicas exchange their operations. The insertion of “n” at 0 can be applied as-is, but if the insertion of “!” at 3 were applied to the state “nice” we would get “nic!e”, which is incorrect. We therefore need to transform the index of each operation to account for concurrent operations that have already been applied; in this case, the insertion of “!” is transformed to index 4 to account for the insertion of “n” at an earlier index.

CRDT

Most CRDTs give each character a unique, immutable ID and use those to determine the positions of insertions/deletions, instead of indexes. For example, in [Figure 6-11](#) we assign the ID 1A to “i”, the ID 2A to “c”, etc. When inserting the exclamation mark, we generate an operation containing the ID of the new character (4B) and the ID of the existing character after which we want to insert (3A). To insert at the beginning of the string we give “nil” as the preceding character ID. Concurrent insertions at the same position are ordered by the IDs of the characters. This ensures that replicas converge without performing any transformation.

There are many algorithms based on variations of these ideas. Lists/arrays can be supported similarly, using list elements instead of characters, and other datatypes such as key-value maps can be added quite easily. There are some performance and functionality trade-offs between OT and CRDTs, but it’s possible to combine the advantages of CRDTs and OT in one algorithm [\[48\]](#).

OT is most often used for real-time collaborative editing of text, e.g. in Google Docs [\[31\]](#), whereas CRDTs can be found in distributed databases such as Redis Enterprise, Riak, and Azure Cosmos DB [\[49\]](#). Sync engines for JSON data can be

implemented both with CRDTs (e.g., Automerge or Yjs) and with OT (e.g., ShareDB).

What is a conflict?

Some kinds of conflict are obvious. In the example in [Figure 6-9](#), two writes concurrently modified the same field in the same record, setting it to two different values. There is little doubt that this is a conflict.

Other kinds of conflict can be more subtle to detect. For example, consider a meeting room booking system: it tracks which room is booked by which group of people at which time. This application needs to ensure that each room is only booked by one group of people at any one time (i.e., there must not be any overlapping bookings for the same room). In this case, a conflict may arise if two different bookings are created for the same room at the same time. Even if the application checks availability before allowing a user to make a booking, there can be a conflict if the two bookings are made on two different leaders.

There isn't a quick ready-made answer, but in the following chapters we will trace a path toward a good understanding of this problem. We will see some more examples of conflicts in

[Chapter 8](#), and in [Link to Come] we will discuss scalable approaches for detecting and resolving conflicts in a replicated system.

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, 50], but the idea was mostly forgotten during the era of dominance of relational databases. It once again became a fashionable architecture for databases after Amazon used it for its in-house *Dynamo* system in 2007 [45]. Riak, Cassandra, and ScyllaDB are open source datastores with leaderless replication models inspired by

Dynamo, so this kind of database is also known as *Dynamo-style*.

NOTE

The original *Dynamo* system was only described in a paper [45], but never released outside of Amazon. The similarly-named *DynamoDB* is a more recent cloud database from AWS, but it has a completely different architecture: it uses single-leader replication based on the Multi-Paxos consensus algorithm [5].

In some leaderless implementations, the client directly sends its writes to several replicas, while in others, a coordinator node does this on behalf of the client. However, unlike a leader database, that coordinator does not enforce a particular ordering of writes. As we shall see, this difference in design has profound consequences for the way the database is used.

Writing to the Database When a Node Is Down

Imagine you have a database with three replicas, and one of the replicas is currently unavailable—perhaps it is being rebooted to install a system update. In a single-leader configuration, if you want to continue processing writes, you may need to perform a failover (see “[Handling Node Outages](#)”).

On the other hand, in a leaderless configuration, failover does not exist. [Figure 6-12](#) shows what happens: the client (user 1234) sends the write to all three replicas in parallel, and the two available replicas accept the write but the unavailable replica misses it. Let's say that it's sufficient for two out of three replicas to acknowledge the write: after user 1234 has received two *ok* responses, we consider the write to be successful. The client simply ignores the fact that one of the replicas missed the write.

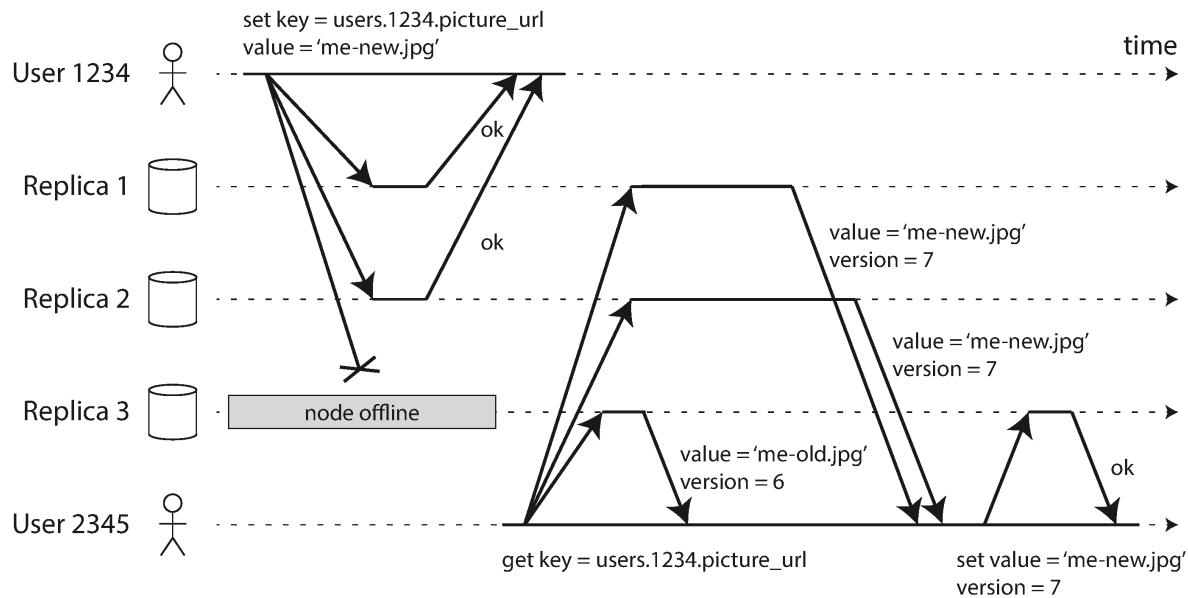


Figure 6-12. A quorum write, quorum read, and read repair after a node outage.

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; for example, the up-to-date value from one node and a stale value from another.

In order to tell which responses are up-to-date and which are outdated, every value that is written needs to be tagged with a version number or timestamp, similarly to what we saw in [“Last write wins \(discarding concurrent writes\)”](#). When a client receives multiple values in response to a read, it uses the one with the greatest timestamp (even if that value was only returned by one replica, and several other replicas returned older values). See [“Detecting Concurrent Writes”](#) for more details.

Catching up on missed writes

The replication system 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? Several mechanisms are 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 6-12](#), user 2345 gets a version 6 value from replica 3 and a version 7 value from replicas 1 and 2. The client sees that replica 3 has a stale value and writes the newer value back to that replica. This approach works well for values that are frequently read.

Hinted handoff

If one replica is unavailable, another replica may store writes on its behalf in the form of *hints*. When the replica that was supposed to receive those writes comes back, the replica storing the hints sends them to the recovered replica, and then deletes the hints. This *handoff* process helps bring replicas up-to-date even for values that are never read, and therefore not handled by read repair.

Anti-entropy

In addition, there is a background process that periodically 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.

Quorums for reading and writing

In the example of [Figure 6-12](#), 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 [\[50\]](#).

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.

NOTE

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 sharded, supporting datasets that are larger than you can fit on one node. We will return to sharding in [Chapter 7](#).

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, like in [Figure 6-12](#).

- With $n = 5$, $w = 3$, $r = 3$ we can tolerate two unavailable nodes. This case is illustrated in [Figure 6-13](#).

Normally, reads and writes are always sent to all n replicas in parallel. The parameters w and r determine how many nodes we wait for—i.e., how many of the n nodes need to report success before we consider the read or write to be successful.

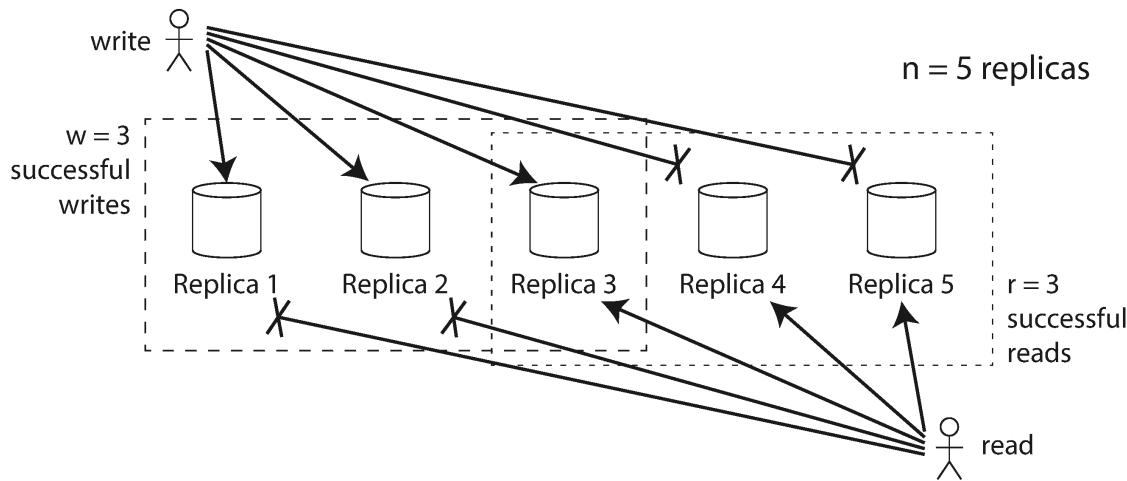


Figure 6-13. If $w + r > n$, at least one of the r replicas you read from must have seen the most recent successful write.

If fewer than the required w or r nodes are available, writes or reads return an error. A node could be unavailable for many reasons: because the node is down (crashed, powered down), due to an error executing the operation (can't write because the disk is full), due to a network interruption between the client and the node, or for any number of other reasons. We only care

whether the node returned a successful response and don't need to distinguish between different kinds of fault.

Limitations of Quorum Consistency

If you have n replicas, and you choose w and r such that $w + r > n$, you can generally expect every read to return the most recent value written for a key. This is the case because the set of nodes to which you've written and the set of nodes from which you've read must overlap. That is, among the nodes you read there must be at least one node with the latest value (illustrated in [Figure 6-13](#)).

Often, r and w are chosen to be a majority (more than $n/2$) of nodes, because that ensures $w + r > n$ while still tolerating up to $n/2$ (rounded down) node failures. But quorums are not necessarily majorities—it only matters that the sets of nodes used by the read and write operations overlap in at least one node. Other quorum assignments are possible, which allows some flexibility in the design of distributed algorithms [51].

You may also set w and r to smaller numbers, so that $w + r \leq n$ (i.e., the quorum condition is not satisfied). In this case, reads and writes will still be sent to n nodes, but a smaller number of successful responses is required for the operation to succeed.

With a smaller w and r you are more likely to read stale values, because it's more likely that your read didn't include the node with the latest value. On the upside, this configuration allows lower latency and higher availability: if there is a network interruption and many replicas become unreachable, there's a higher chance that you can continue processing reads and writes. Only after the number of reachable replicas falls below w or r does the database become unavailable for writing or reading, respectively.

However, even with $w + r > n$, there are edge cases in which the consistency properties can be confusing. Some scenarios include:

- 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.
- While a rebalancing is in progress, where some data is moved from one node to another (see [Chapter 7](#)), nodes may have inconsistent views of which nodes should be holding the n replicas for a particular value. This can result in the read and write quorums no longer overlapping.
- If a read is concurrent with a write operation, the read may or may not see the concurrently written value. In particular,

it's possible for one read to see the new value, and a subsequent read to see the old value, as we shall see in [Link to Come].

- 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 [52].
- If the database uses timestamps from a real-time clock to determine which write is newer (as Cassandra and ScyllaDB do, for example), writes might be silently dropped if another node with a faster clock has written to the same key—an issue we previously saw in “[Last write wins \(discarding concurrent writes\)](#)”. We will discuss this in more detail in [Link to Come].
- If two writes occur concurrently, one of them might be processed first on one replica, and the other might be processed first on another replica. This leads to a conflict, similarly to what we saw for multi-leader replication (see “[Dealing with Conflicting Writes](#)”). We will return to this topic in “[Detecting Concurrent Writes](#)”.

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 [53], but it's wise to not take them as absolute guarantees.

Monitoring staleness

From an operational perspective, it's important to monitor whether your databases are returning up-to-date results. Even if your application can tolerate stale reads, you need to be aware of the health of your replication. If it falls behind significantly, it should alert you so that you can investigate the cause (for example, a problem in the network or an overloaded node).

For leader-based replication, the database typically exposes metrics for the replication lag, which you can feed into a monitoring system. This is possible because writes are applied to the leader and to followers in the same order, and each node has a position in the replication log (the number of writes it has applied locally). By subtracting a follower's current position from the leader's current position, you can measure the amount of replication lag.

However, in systems with leaderless replication, there is no fixed order in which writes are applied, which makes monitoring more difficult. The number of hints that a replica stores for handoff can be one measure of system health, but it's difficult to interpret usefully [54]. Eventual consistency is a deliberately vague guarantee, but for operability it's important to be able to quantify "eventual."

Single-Leader vs. Leaderless Replication Performance

A replication system based on a single leader can provide strong consistency guarantees that are difficult or impossible to achieve in a leaderless system. However, as we have seen in "[Problems with Replication Lag](#)", reads in a leader-based replicated system can also return stale values if you make them on an asynchronously updated follower.

Reading from the leader ensures up-to-date responses, but it suffers from performance problems:

- Read throughput is limited by the leader's capacity to handle requests (in contrast with read scaling, which distributes reads across asynchronously updated replicas that may return stale values).

- If the leader fails, you have to wait for the fault to be detected, and for the failover to complete before you can continue handling requests. Even if the failover process is very quick, users will notice it because of the temporarily increased response times; if failover takes a long time, the system is unavailable for its duration.
- The system is very sensitive to performance problems on the leader: if the leader is slow to respond, e.g. due to overload or some resource contention, the increased response times immediately affect users as well.

A big advantage of a leaderless architecture is that it is more resilient against such issues. Because there is no failover, and requests go to multiple replicas in parallel anyway, one replica becoming slow or unavailable has very little impact on response times: the client simply uses the responses from the other replicas that are faster to respond. Using the fastest responses is called *request hedging*, and it can significantly reduce tail latency [55]).

At its core, the resilience of a leaderless system comes from the fact that it doesn't distinguish between the normal case and the failure case. This is especially helpful when handling so-called *gray failures*, in which a node isn't completely down, but running in a degraded state where it is unusually slow to

handle requests [56], or when a node is simply overloaded (for example, if a node has been offline for a while, recovery via hinted handoff can cause a lot of additional load). A leader-based system has to decide whether the situation is bad enough to warrant a failover (which can itself cause further disruption), whereas in a leaderless system that question doesn't even arise.

That said, leaderless systems can have performance problems as well:

- Even though the system doesn't need to perform failover, one replica does need to detect when another replica is unavailable so that it can store hints about writes that the unavailable replica missed. When the unavailable replica comes back, the handoff process needs to send it those hints. This puts additional load on the replicas at a time when the system is already under strain [54].
- The more replicas you have, the bigger the size of your quorums, and the more responses you have to wait for before a request can complete. Even if you wait only for the fastest r or w replicas to respond, and even if you make the requests in parallel, a bigger r or w increases the chance that you hit a slow replica, increasing the overall response time (see [“Use of Response Time Metrics”](#)).

- A large-scale network interruption that disconnects a client from a large number of replicas can make it impossible to form a quorum. Some leaderless databases offer a configuration option that allows any reachable replica to accept writes, even if it's not one of the usual replicas for that key (Riak and Dynamo call this a *sloppy quorum* [45]; Cassandra and ScyllaDB call it *consistency level ANY*). There is no guarantee that subsequent reads will see the written value, but depending on the application it may still be better than having the write fail.

Multi-leader replication can offer even greater resilience against network interruptions than leaderless replication, since reads and writes only require communication with one leader, which can be co-located with the client. However, since a write on one leader is propagated asynchronously to the others, reads can be arbitrarily out-of-date. Quorum reads and writes provide a compromise: good fault tolerance while also having a high likelihood of reading up-to-date data.

Multi-region operation

We previously discussed cross-region replication as a use case for multi-leader replication (see “[Multi-Leader Replication](#)”). Leaderless replication is also suitable for multi-region

operation, since it is designed to tolerate conflicting concurrent writes, network interruptions, and latency spikes.

Cassandra and ScyllaDB implement their multi-region support within the normal leaderless model: the client sends its writes directly to the replicas in all regions, and you can choose from a variety of consistency levels that determine how many responses are required for a request to be successful. For example, you can request a quorum across the replicas in all the regions, a separate quorum in each of the regions, or a quorum only in the client's local region. A local quorum avoids having to wait for slow requests to other regions, but it is also more likely to return stale results.

Riak keeps all communication between clients and database nodes local to one region, so n describes the number of replicas within one region. Cross-region replication between database clusters happens asynchronously in the background, in a style that is similar to multi-leader replication.

Detecting Concurrent Writes

Like with multi-leader replication, leaderless databases allow concurrent writes to the same key, resulting in conflicts that need to be resolved. Such conflicts may occur as the writes

happen, but not always: they could also be detected later during read repair, hinted handoff, or anti-entropy.

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 6-14](#) 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.

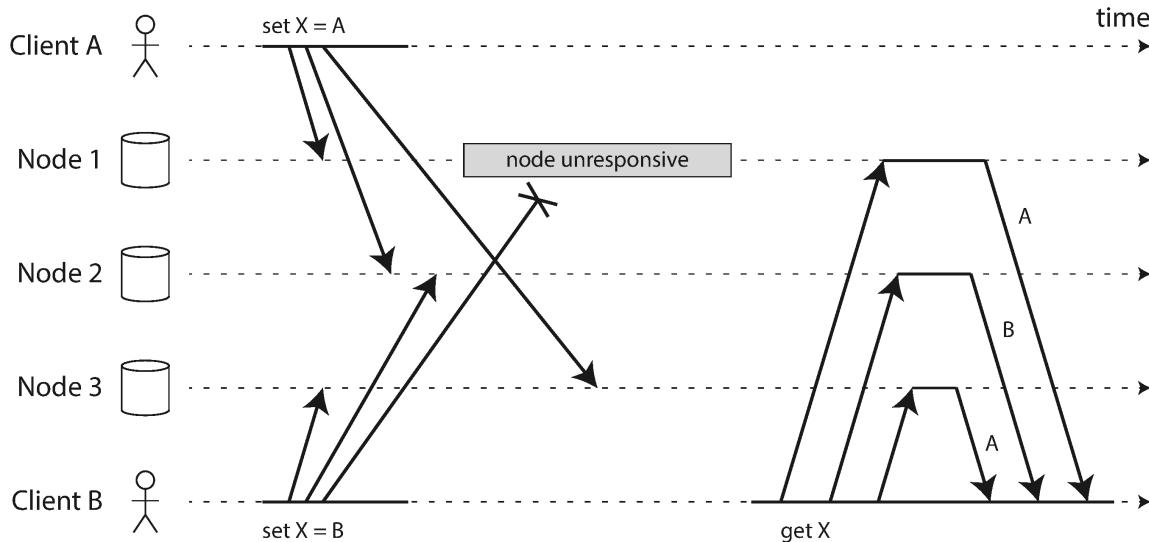


Figure 6-14. Concurrent writes in a Dynamo-style datastore: there is no well-defined ordering.

If each node simply overwrote the value for a key whenever it received a write request from a client, the nodes would become permanently inconsistent, as shown by the final *get* request in [Figure 6-14](#): node 2 thinks that the final value of X is B, whereas the other nodes think that the value is A.

In order to become eventually consistent, the replicas should converge toward the same value. For this, we can use any of the conflict resolution mechanisms we previously discussed in [“Dealing with Conflicting Writes”](#), such as last-write-wins (used by Cassandra and ScyllaDB), manual resolution, or CRDTs (described in [“CRDTs and Operational Transformation”](#), and used by Riak).

Last-write-wins is easy to implement: each write is tagged with a timestamp, and a value with a higher timestamp always overwrites a value with a lower timestamp. However, a timestamp doesn't tell you whether two values are actually conflicting (i.e., they were written concurrently) or not (they were written one after another). If you want to resolve conflicts explicitly, the system needs to take more care to detect concurrent writes.

The “happens-before” relation 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 6-8](#), 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 6-14](#) are concurrent: when each client starts the operation, it does not know that another client is also performing an operation on the same key. Thus, there is no causal dependency between the operations.

An operation A *happens before* another operation B if B knows about A, or depends on A, or builds upon A in some way.

Whether one operation happens before another operation is the key to defining what concurrency means. In fact, we can simply say that two operations are *concurrent* if neither happens before the other (i.e., neither knows about the other) [\[57\]](#).

Thus, whenever you have two operations A and B, there are three possibilities: either A happened before B, or B happened before A, or A and B are concurrent. What we need is an algorithm to tell us whether two operations are concurrent or not. If one operation happened before another, the later operation should overwrite the earlier operation, but if the operations are concurrent, we have a conflict that needs to be resolved.

CONCURRENCY, TIME, AND RELATIVITY

It may seem that two operations should be called concurrent if they occur “at the same time”—but in fact, it is not important whether they literally overlap in time. Because of problems with clocks in distributed systems, it is actually quite difficult to tell whether two things happened at exactly the same time—an issue we will discuss in more detail in [Link to Come].

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 [57], which introduced the idea that information cannot travel faster than the speed of light. Consequently, two events that occur some distance apart cannot possibly affect each other if the time between the events is shorter than the time it takes light to travel the distance between them.

In computer systems, two operations might be concurrent even though the speed of light would in principle have allowed one operation to affect the other. For example, if the network was slow or interrupted at the time, two operations can occur some time apart and still be concurrent, because the network

problems prevented one operation from being able to know about the other.

Capturing the happens-before relationship

Let's look at an algorithm that determines whether two operations are concurrent, or whether one happened before another. To keep things simple, let's start with a 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 6-15](#) shows two clients concurrently adding items to the same shopping cart. (If that example strikes you as too inane, imagine instead two air traffic controllers concurrently adding aircraft to the sector they are tracking.) Initially, the cart is empty. Between them, the clients make five writes to the database:

1. Client 1 adds `milk` to the cart. This is the first write to that key, so the server successfully stores it and assigns it version 1. The server also echoes the value back to the client, along with the version number.
2. Client 2 adds `eggs` to the cart, not knowing that client 1 concurrently added `milk` (client 2 thought that its `eggs`

were the only item in the cart). The server assigns version 2 to this write, and stores `eggs` and `milk` as two separate values (siblings). It then returns *both* values to the client, along with the version number of 2.

3. Client 1, oblivious to client 2's write, wants to add `flour` to the cart, so it thinks the current cart contents should be

`[milk, flour]`. It sends this value to the server, along with the version number 1 that the server gave client 1 previously. The server can tell from the version number that the write of `[milk, flour]` supersedes the prior value of `[milk]` but that it is concurrent with `[eggs]`. Thus, the server assigns version 3 to `[milk, flour]`, overwrites the version 1 value `[milk]`, but keeps the version 2 value `[eggs]` and returns both remaining values to the client.

4. Meanwhile, client 2 wants to add `ham` to the cart, unaware that client 1 just added `flour`. Client 2 received the two values `[milk]` and `[eggs]` from the server in the last response, so the client now merges those values and adds `ham` to form a new value, `[eggs, milk, ham]`. It sends that value to the server, along with the previous version number 2. The server detects that version 2 overwrites `[eggs]` but is concurrent with `[milk, flour]`, so the two remaining values are `[milk, flour]` with version 3, and `[eggs, milk, ham]` with version 4.

5. Finally, client 1 wants to add bacon. It previously received [milk, flour] and [eggs] from the server at version 3, so it merges those, adds bacon, and sends the final value [milk, flour, eggs, bacon] to the server, along with the version number 3. This overwrites [milk, flour] (note that [eggs] was already overwritten in the last step) but is concurrent with [eggs, milk, ham], so the server keeps those two concurrent values.

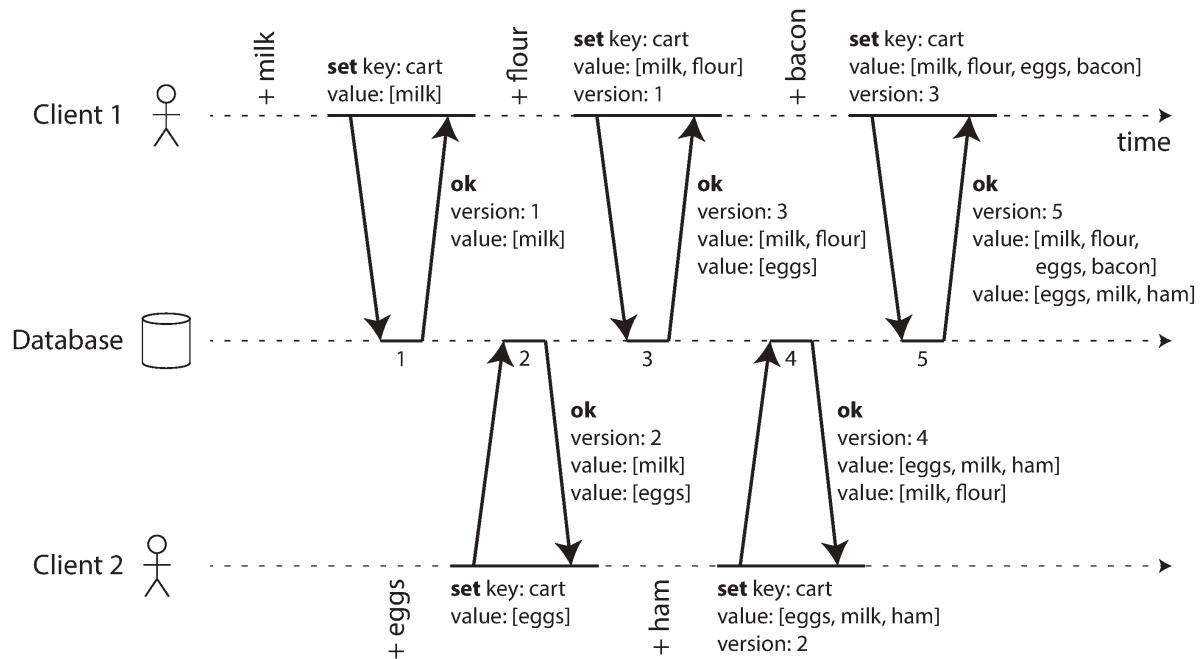


Figure 6-15. Capturing causal dependencies between two clients concurrently editing a shopping cart.

The dataflow between the operations in [Figure 6-15](#) is illustrated graphically in [Figure 6-16](#). The arrows indicate which operation *happened before* which other operation, in the

sense that the later operation *knew about* or *depended on* the earlier one. In this example, the clients are never fully up to date with the data on the server, since there is always another operation going on concurrently. But old versions of the value do get overwritten eventually, and no writes are lost.

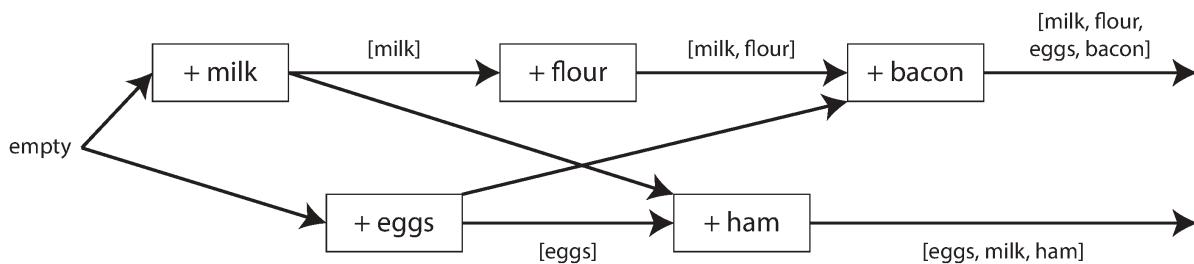


Figure 6-16. Graph of causal dependencies in [Figure 6-15](#).

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 siblings, i.e., 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, e.g. using a CRDT or by asking the user. The response from a write request is like a read, returning all siblings, which allows us to chain several writes like in the shopping cart example.
- When the server receives a write with a particular version number, it can overwrite all values with that version number or below (since it knows that they have been merged into the new value), but it must keep all values with a higher version number (because those values are concurrent with the incoming write).

When a write includes the version number from a prior read, that tells us which previous state the write is based on. If you make a write without including a version number, it is concurrent with all other writes, so it will not overwrite anything—it will just be returned as one of the values on subsequent reads.

Version vectors

The example in [Figure 6-15](#) used only a single replica. How does the algorithm change when there are multiple replicas, but no leader?

[Figure 6-15](#) uses a single version number to capture dependencies between operations, but that is not sufficient when there are multiple replicas accepting writes concurrently. Instead, we need to use a version number *per replica* as well as per key. Each replica increments its own version number when processing a write, and also keeps track of the version numbers it has seen from each of the other replicas. This information indicates which values to overwrite and which values to keep as siblings.

The collection of version numbers from all the replicas is called a *version vector* [58]. A few variants of this idea are in use, but the most interesting is probably the *dotted version vector* [59, 60], which is used in Riak 2.0 [61, 62]. We won't go into the details, but the way it works is quite similar to what we saw in our cart example.

Like the version numbers in [Figure 6-15](#), version vectors are sent from the database replicas to clients when values are read, and need to be sent back to the database when a value is subsequently written. (Riak encodes the version vector as a string that it calls *causal context*.) The version vector allows the database to distinguish between overwrites and concurrent writes.

The version vector also ensures that it is safe to read from one replica and subsequently write back to another replica. Doing so may result in siblings being created, but no data is lost as long as siblings are merged correctly.

VERSION VECTORS AND VECTOR CLOCKS

A *version vector* is sometimes also called a *vector clock*, even though they are not quite the same. The difference is subtle—please see the references for details [[60](#), [63](#), [64](#)]. In brief, when comparing the state of replicas, version vectors are the right data structure to use.

Summary

In this chapter we looked at the issue of replication. Replication can serve several purposes:

High availability

Keeping the system running, even when one machine (or several machines, a zone, or even an entire region) goes down

Disconnected operation

Allowing an application to continue working when there is a network interruption

Latency

Placing data geographically close to users, so that users can interact with it faster

Scalability

Being able to handle a higher volume of reads than a single machine could handle, by performing reads on replicas

Despite being a simple goal—keeping a copy of the same data on several machines—replication turns out to be a remarkably tricky problem. It requires carefully thinking about concurrency and about all the things that can go wrong, and dealing with the consequences of those faults. At a minimum, we need to deal with unavailable nodes and network interruptions (and that's not even considering the more insidious kinds of fault, such as silent data corruption due to software bugs or hardware errors).

We discussed three main approaches to replication:

Single-leader replication

Clients send all writes to a single node (the leader), which sends a stream of data change events to the other replicas

(followers). Reads can be performed on any replica, but reads from followers might be stale.

Multi-leader replication

Clients send each write to one of several leader nodes, any of which can accept writes. The leaders send streams of data change events to each other and to any follower nodes.

Leaderless replication

Clients send each write to several nodes, and read from several nodes in parallel in order to detect and correct nodes with stale data.

Each approach has advantages and disadvantages. Single-leader replication is popular because it is fairly easy to understand and it offers strong consistency. Multi-leader and leaderless replication can be more robust in the presence of faulty nodes, network interruptions, and latency spikes—at the cost of requiring conflict resolution and providing weaker consistency guarantees.

Replication can be synchronous or asynchronous, which has a profound effect on the system behavior when there is a fault. Although asynchronous replication can be fast when the system

is running smoothly, it's important to figure out what happens when replication lag increases and servers fail. If a leader fails and you promote an asynchronously updated follower to be the new leader, recently committed data may be lost.

We looked at some strange effects that can be caused by replication lag, and we discussed a few consistency models which are helpful for deciding how an application should behave under replication lag:

Read-after-write consistency

Users should always see data that they submitted themselves.

Monotonic reads

After users have seen the data at one point in time, they shouldn't later see the data from some earlier point in time.

Consistent prefix reads

Users should see the data in a state that makes causal sense: for example, seeing a question and its reply in the correct order.

Finally, we discussed how multi-leader and leaderless replication ensure that all replicas eventually converge to a consistent state: by using a version vector or similar algorithm to detect which writes are concurrent, and by using a conflict resolution algorithm such as a CRDT to merge the concurrently written values. Last-write-wins and manual conflict resolution are also possible.

This chapter has assumed that every replica stores a full copy of the whole database, which is unrealistic for large datasets. In the next chapter we will look at *sharding*, which allows each machine to store only a subset of the data.

FOOTNOTES

REFERENCES

- Bruce G. Lindsay, Patricia Griffiths Selinger, C. Galtieri, J.N. Gray, R.A. Lorie, T.G. Price, F. Putzolu, I.L. Traiger, and B.W. Wade. [Notes on Distributed Databases](#). IBM Research, Research Report RJ2571(33471), July 1979. Archived at perma.cc/EPZ3-MHDD
- Kenny Gryp. [MySQL Terminology Updates](#). *dev.mysql.com*, July 2020. Archived at perma.cc/S62G-6RJ2
- Oracle Corporation. [Oracle \(Active\) Data Guard 19c: Real-Time Data Protection and Availability](#). White Paper, *oracle.com*, March 2019. Archived at perma.cc/P5ST-RPKE

Microsoft. [What is an Always On availability group?](#) *learn.microsoft.com*, September 2024. Archived at perma.cc/ABH6-3MXF

Mostafa Elhemali, Niall Gallagher, Nicholas Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somu Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. [Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service](#). At *USENIX Annual Technical Conference (ATC)*, July 2022.

Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. [CockroachDB: The Resilient Geo-Distributed SQL Database](#). At *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1493–1509, June 2020. [doi:10.1145/3318464.3386134](https://doi.org/10.1145/3318464.3386134)

Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. [TiDB: a Raft-based HTAP database](#). *Proceedings of the VLDB Endowment*, volume 13, issue 12, pages 3072–3084. [doi:10.14778/3415478.3415535](https://doi.org/10.14778/3415478.3415535)

Mallory Knodel and Niels ten Oever. [Terminology, Power, and Inclusive Language in Internet-Drafts and RFCs](#). *IETF Internet-Draft*, August 2023. Archived at perma.cc/5ZY9-725E

Buck Hodges. [Postmortem: VSTS 4 September 2018](#). *devblogs.microsoft.com*, September 2018. Archived at perma.cc/ZF5R-DYZS

Gunnar Morling. [Leader Election With S3 Conditional Writes](#). *www.morling.dev*, August 2024. Archived at perma.cc/7V2N-J78Y

Stas Kelvich. [Why does Neon use Paxos instead of Raft, and what's the difference?](#). [neon.tech](#), August 2022. Archived at [perma.cc/SEZ4-2GXU](#)

Dimitri Fontaine. [An introduction to the pg_auto_failover project](#). [tapoueh.org](#), November 2021. Archived at [perma.cc/3WH5-6BAF](#)

Jesse Newland. [GitHub availability this week](#). [github.blog](#), September 2012. Archived at [perma.cc/3YRF-FTFJ](#)

Mark Imbriaco. [Downtime last Saturday](#). [github.blog](#), December 2012. Archived at [perma.cc/M7X5-E8SQ](#)

John Hugg. [‘All In’ with Determinism for Performance and Testing in Distributed Systems](#). At *Strange Loop*, September 2015.

Hironobu Suzuki. [The Internals of PostgreSQL](#). [interdb.jp](#), 2017.

Amit Kapila. [WAL Internals of PostgreSQL](#). At *PostgreSQL Conference (PGCon)*, May 2012. Archived at [perma.cc/6225-3SUX](#)

Amit Kapila. [Evolution of Logical Replication](#). [amitkapila16.blogspot.com](#), September 2023. Archived at [perma.cc/F9VX-JLER](#)

Aru Patchimuthu. [Upgrade your Amazon RDS for PostgreSQL or Amazon Aurora PostgreSQL database, Part 2: Using the pglogical extension](#). [aws.amazon.com](#), August 2021. Archived at [perma.cc/RXT8-FS2T](#)

Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie. [Wormhole: Reliable Pub-](#)

[Sub to Support Geo-Replicated Internet Services](#). At *12th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), May 2015.

Douglas B. Terry. [Replicated Data Consistency Explained Through Baseball](#). Microsoft Research, Technical Report MSR-TR-2011-137, October 2011. Archived at perma.cc/F4KZ-AR38

Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theher, and Brent B. Welch. [Session Guarantees for Weakly Consistent Replicated Data](#). At *3rd International Conference on Parallel and Distributed Information Systems* (PDIS), September 1994. [doi:10.1109/PDIS.1994.331722](https://doi.org/10.1109/PDIS.1994.331722)

Werner Vogels. [Eventually Consistent](#). *ACM Queue*, volume 6, issue 6, pages 14–19, October 2008. [doi:10.1145/1466443.1466448](https://doi.org/10.1145/1466443.1466448)

Simon Willison. [Reply to: “My thoughts about Fly.io \(so far\) and other newish technology I’m getting into”](#). *news.ycombinator.com*, May 2022. Archived at perma.cc/ZRV4-WWV8

Nithin Tharakan. [Scaling Bitbucket’s Database](#). *atlassian.com*, October 2020. Archived at perma.cc/JAB7-9FGX

Terry Pratchett. *Reaper Man: A Discworld Novel*. Victor Gollancz, 1991. ISBN: 978-0-575-04979-6

Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. [Coordination Avoidance in Database Systems](#). *Proceedings of the VLDB Endowment*, volume 8, issue 3, pages 185–196, November 2014. [doi:10.14778/2735508.2735509](https://doi.org/10.14778/2735508.2735509)

Yaser Raja and Peter Celentano. [PostgreSQL bi-directional replication using pglogical](#). *aws.amazon.com*, January 2022. Archived at <https://perma.cc/BUQ2-5QWN>

Robert Hodges. [If You *Must* Deploy Multi-Master Replication, Read This First](#). *scale-out-blog.blogspot.com*, April 2012. Archived at [perma.cc/C2JN-F6Y8](#)

Lars Hofhansl. [HBASE-7709: Infinite Loop Possible in Master/Master Replication](#). *issues.apache.org*, January 2013. Archived at [perma.cc/24G2-8NLC](#)

John Day-Richter. [What's Different About the New Google Docs: Making Collaboration Fast](#). *drive.googleblog.com*, September 2010. Archived at [perma.cc/5TL8-TSJ2](#)

Evan Wallace. [How Figma's multiplayer technology works](#). *figma.com*, October 2019. Archived at [perma.cc/L49H-LY4D](#)

Tuomas Artman. [Scaling the Linear Sync Engine](#). *linear.app*, June 2023.

Amr Saafan. [Why Sync Engines Might Be the Future of Web Applications](#). *nilebits.com*, September 2024. Archived at [perma.cc/5N73-5M3V](#)

Isaac Hagoel. [Are Sync Engines The Future of Web Applications?](#) *dev.to*, July 2024. Archived at [perma.cc/R9HF-BKKL](#)

Sujay Jayakar. [A Map of Sync](#). *stack.convex.dev*, October 2024. Archived at [perma.cc/82R3-H42A](#)

Alex Feyerke. [Designing Offline-First Web Apps](#). *alistapart.com*, December 2013. Archived at [perma.cc/WH7R-S2DS](#)

Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. [Local-first software: You own your data, in spite of the cloud](#). At *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Onward!), October 2019, pages 154–178.
[doi:10.1145/3359591.3359737](#)

Martin Kleppmann. [The past, present, and future of local-first](#). At *Local-First Conference*, May 2024.

Conrad Hofmeyr. [API Calling is to Sync Engines as jQuery is to React](#). *powersync.com*, November 2024. Archived at perma.cc/2FP9-7WJJ.

Peter van Hardenberg and Martin Kleppmann. [PushPin: Towards Production-Quality Peer-to-Peer Collaboration](#). At *7th Workshop on Principles and Practice of Consistency for Distributed Data* (PaPoC), April 2020. [doi:10.1145/3380787.3393683](https://doi.org/10.1145/3380787.3393683)

Leonard Kawell, Jr., Steven Beckhardt, Timothy Halvorsen, Raymond Ozzie, and Irene Greif. [Replicated document management in a group communication system](#). At *ACM Conference on Computer-Supported Cooperative Work* (CSCW), September 1988. [doi:10.1145/62266.1024798](https://doi.org/10.1145/62266.1024798)

Ricky Pusch. [Explaining how fighting games use delay-based and rollback netcode](#). *words.infil.net* and *arstechnica.com*, October 2019. Archived at perma.cc/DE7W-RDJ8

Ryan King. [Announcing Snowflake](#). *blog.x.com*, June 2010. Archived at archive.org

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. [Dynamo: Amazon's Highly Available Key-Value Store](#). At *21st ACM Symposium on Operating Systems Principles* (SOSP), October 2007. [doi:10.1145/1323293.1294281](https://doi.org/10.1145/1323293.1294281)

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.

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. [doi:10.1145/289444.289469](https://doi.org/10.1145/289444.289469)

Joseph Gentle and Martin Kleppmann. [Collaborative Text Editing with Eg-walker: Better, Faster, Smaller](#). At *20th European Conference on Computer Systems* (EuroSys), March 2025. [doi:10.1145/3689031.3696076](#)

Dharma Shukla. [Azure Cosmos DB: Pushing the frontier of globally distributed databases](#). azure.microsoft.com, September 2018. Archived at [perma.cc/UT3B-HH6R](#)

David K. Gifford. [Weighted Voting for Replicated Data](#). At *7th ACM Symposium on Operating Systems Principles* (SOSP), December 1979. [doi:10.1145/800215.806583](#)

Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. [Flexible Paxos: Quorum Intersection Revisited](#). At *20th International Conference on Principles of Distributed Systems* (OPODIS), December 2016. [doi:10.4230/LIPIcs.OPODIS.2016.25](#)

Joseph Blomstedt. [Bringing Consistency to Riak](#). At *RICON West*, October 2012.

Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. [Quantifying eventual consistency with PBS](#). *The VLDB Journal*, volume 23, pages 279–302, April 2014. [doi:10.1007/s00778-013-0330-1](#)

Colin Breck. [Shared-Nothing Architectures for Server Replication and Synchronization](#). blog.colinbreck.com, December 2019. Archived at [perma.cc/48P3-J6CJ](#)

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

Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. [Gray Failure: The Achilles' Heel of Cloud-Scale Systems](#). At *16th Workshop on Hot Topics in Operating Systems* (HotOS), May 2017. [doi:10.1145/3102980.3103005](#)

Leslie Lamport. [Time, Clocks, and the Ordering of Events in a Distributed System](#). *Communications of the ACM*, volume 21, issue 7, pages 558–565, July 1978.
[doi:10.1145/359545.359563](#)

D. Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. [Detection of Mutual Inconsistency in Distributed Systems](#). *IEEE Transactions on Software Engineering*, volume SE-9, issue 3, pages 240–247, May 1983.
[doi:10.1109/TSE.1983.236733](#)

Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. [Dotted Version Vectors: Logical Clocks for Optimistic Replication](#). arXiv:1011.5808, November 2010.

Giridhar Manepalli. [Clocks and Causality - Ordering Events in Distributed Systems](#). *exhypothesi.com*, November 2022. Archived at [perma.cc/8REU-KVLQ](#)

Sean Cribbs. [A Brief History of Time in Riak](#). At RICON, October 2014. Archived at [perma.cc/7U9P-6JFX](#)

Russell Brown. [Vector Clocks Revisited Part 2: Dotted Version Vectors](#). *riak.com*, November 2015. Archived at [perma.cc/96QP-W98R](#)

Carlos Baquero. [Version Vectors Are Not Vector Clocks](#). *haslab.wordpress.com*, July 2011. Archived at [perma.cc/7PNU-4AMG](#)

Reinhard Schwarz and Friedemann Mattern. [Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail](#). *Distributed Computing*, volume 7, issue 3, pages 149–174, March 1994. [doi:10.1007/BF02277859](#)

Chapter 7. Sharding

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)

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. The GitHub repo for this book is <https://github.com/ept/ddia2-feedback>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out on GitHub.

A distributed database typically distributes data across nodes in two ways:

1. Having a copy of the same data on multiple nodes: this is *replication*, which we discussed in [Chapter 6](#).
2. If we don't want every node to store all the data, we can split up a large amount of data into smaller *shards* or *partitions*, and store different shards on different nodes. We'll discuss sharding in this chapter.

Normally, shards are defined in such a way that each piece of data (each record, row, or document) belongs to exactly one shard. There are various ways of achieving this, which we discuss in depth in this chapter. In effect, each shard is a small database of its own, although some database systems support operations that touch multiple shards at the same time.

Sharding is usually combined with replication so that copies of each shard are stored on multiple nodes. This means that, even though each record belongs to exactly one shard, it may still be stored on several different nodes for fault tolerance.

A node may store more than one shard. If a single-leader replication model is used, the combination of sharding and replication can look like [Figure 7-1](#), for example. Each shard's

leader is assigned to one node, and its followers are assigned to other nodes. Each node may be the leader for some shards and a follower for other shards, but each shard still only has one leader.

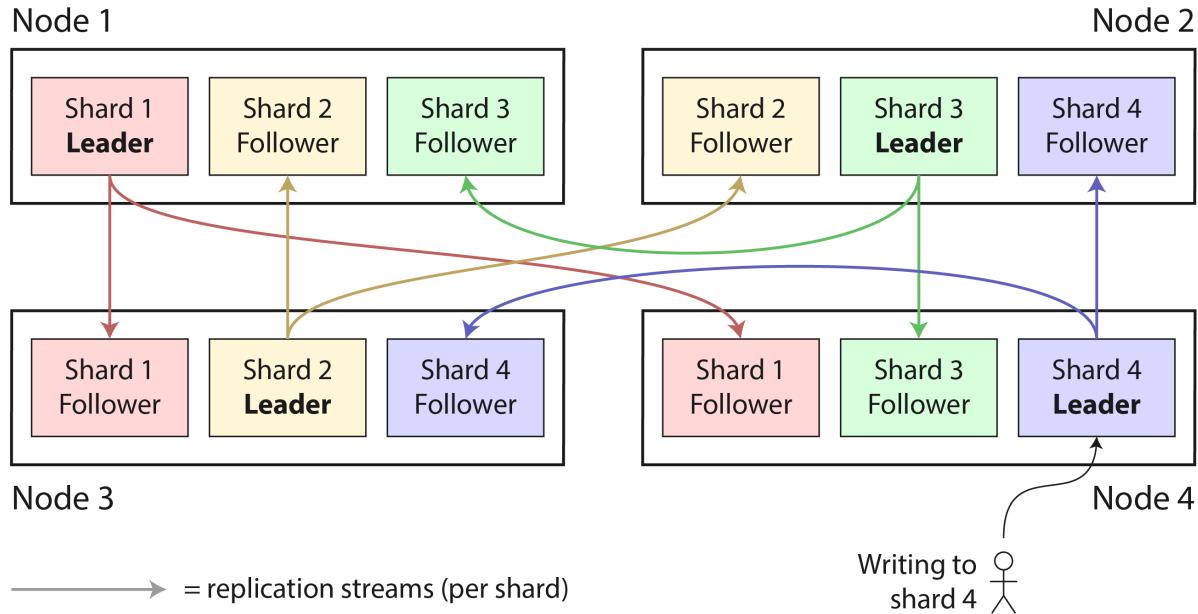


Figure 7-1. Combining replication and sharding: each node acts as leader for some shards and follower for other shards.

Everything we discussed in [Chapter 6](#) about replication of databases applies equally to replication of shards. Since the choice of sharding scheme is mostly independent of the choice of replication scheme, we will ignore replication in this chapter for the sake of simplicity.

SHARDING AND PARTITIONING

What we call a *shard* in this chapter has many different names depending on which software you’re using: it’s called a *partition* in Kafka, a *range* in CockroachDB, a *region* in HBase and TiDB, a *tablet* in Bigtable and YugabyteDB, a *vnode* in Cassandra, ScyllaDB, and Riak, and a *vBucket* in Couchbase, to name just a few.

Some databases treat partitions and shards as two distinct concepts. For example, in PostgreSQL, partitioning is a way of splitting a large table into several files that are stored on the same machine (which has several advantages, such as making it very fast to delete an entire partition), whereas sharding splits a dataset across multiple machines [1, 2]. In many other systems, partitioning is just another word for sharding.

While *partitioning* is quite descriptive, the term *sharding* is perhaps surprising. According to one theory, the term arose from the online role-play game *Ultima Online*, in which a magic crystal was shattered into pieces, and each of those shards refracted a copy of the game world [3]. The term *shard* thus came to mean one of a set of parallel game servers, and later was carried over to databases. Another theory is that *shard* was originally an acronym of *System for Highly Available Replicated*

Data—reportedly a 1980s database, details of which are lost to history.

By the way, partitioning has nothing to do with *network partitions* (netsplits), a type of fault in the network between nodes. We will discuss such faults in [Link to Come].

Pros and Cons of Sharding

The primary reason for sharding a database is *scalability*: it's a solution if the volume of data or the write throughput has become too great for a single node to handle, as it allows you to spread that data and those writes across multiple nodes. (If read throughput is the problem, you don't necessarily need sharding—you can use *read scaling* as discussed in [Chapter 6](#).)

In fact, sharding is one of the main tools we have for achieving *horizontal scaling* (a *scale-out* architecture), as discussed in [“Shared-Memory, Shared-Disk, and Shared-Nothing Architecture”](#): that is, allowing a system to grow its capacity not by moving to a bigger machine, but by adding more (smaller) machines. If you can divide the workload such that each shard handles a roughly equal share, you can then assign those

shards to different machines in order to process their data and queries in parallel.

While replication is useful at both small and large scale, because it enables fault tolerance and offline operation, sharding is a heavyweight solution that is mostly relevant at large scale. If your data volume and write throughput are such that you can process them on a single machine (and a single machine can do a lot nowadays!), it's often better to avoid sharding and stick with a single-shard database.

The reason for this recommendation is that sharding often adds complexity: you typically have to decide which records to put in which shard by choosing a *partition key*; all records with the same partition key are placed in the same shard [4]. This choice matters because accessing a record is fast if you know which shard it's in, but if you don't know the shard you have to do an inefficient search across all shards, and the sharding scheme is difficult to change.

Thus, sharding often works well for key-value data, where you can easily shard by key, but it's harder with relational data where you may want to search by a secondary index, or join records that may be distributed across different shards. We will discuss this further in [“Sharding and Secondary Indexes”](#).

Another problem with sharding is that a write may need to update related records in several different shards. While transactions on a single node are quite common (see [Chapter 8](#)), ensuring consistency across multiple shards requires a *distributed transaction*. As we shall see in [Link to Come], distributed transactions are available in some databases, but they are usually much slower than single-node transactions, may become a bottleneck for the system as a whole, and some systems don't support them at all.

Some systems use sharding even on a single machine, typically running one single-threaded process per CPU core to make use of the parallelism in the CPU, or to take advantage of a *nonuniform memory access* (NUMA) architecture in which some banks of memory are closer to one CPU than to others [5]. For example, Redis, VoltDB, and FoundationDB use one process per core, and rely on sharding to spread load across CPU cores in the same machine [6].

Sharding for Multitenancy

Software as a Service (SaaS) products and cloud services are often *multitenant*, where each tenant is a customer. Multiple users may have logins on the same tenant, but each tenant has a self-contained dataset that is separate from other tenants. For

example, in an email marketing service, each business that signs up is typically a separate tenant, since one business's newsletter signups, delivery data etc. are separate from those of other businesses.

Sometimes sharding is used to implement multitenant systems: either each tenant is given a separate shard, or multiple small tenants may be grouped together into a larger shard. These shards might be physically separate databases (which we previously touched on in "[Embedded storage engines](#)"), or separately manageable portions of a larger logical database [7]. Using sharding for multitenancy has several advantages:

Resource isolation

If one tenant performs a computationally expensive operation, it is less likely that other tenants' performance will be affected if they are running on different shards.

Permission isolation

If there is a bug in your access control logic, it's less likely that you will accidentally give one tenant access to another tenant's data if those tenants' datasets are stored physically separately from each other.

Cell-based architecture

You can apply sharding not only at the data storage level, but also for the services running your application code. In a *cell-based architecture*, the services and storage for a particular set of tenants are grouped into a self-contained *cell*, and different cells are set up such that they can run largely independently from each other. This approach provides *fault isolation*: that is, a fault in one cell remains limited to that cell, and tenants in other cells are not affected [8].

Per-tenant backup and restore

Backing up each tenant's shard separately makes it possible to restore a tenant's state from a backup without affecting other tenants, which can be useful in case the tenant accidentally deletes or overwrites important data [9].

Regulatory compliance

Data privacy regulation such as the GDPR gives individuals the right to access and delete all data stored about them. If each person's data is stored in a separate shard, this translates into simple data export and deletion operations on their shard [10].

Data residence

If a particular tenant's data needs to be stored in a particular jurisdiction in order to comply with data residency laws, a region-aware database can allow you to assign that tenant's shard to a particular region.

Gradual schema rollout

Schema migrations (previously discussed in “[Schema flexibility in the document model](#)”) can be rolled out gradually, one tenant at a time. This reduces risk, as you can detect problems before they affect all tenants, but it can be difficult to do transactionally [11].

The main challenges around using sharding for multitenancy are:

- It assumes that each individual tenant is small enough to fit on a single node. If that is not the case, and you have a single tenant that's too big for one machine, you would need to additionally perform sharding within a single tenant, which brings us back to the topic of sharding for scalability [12].
- If you have many small tenants, then creating a separate shard for each one may incur too much overhead. You could group several small tenants together into a bigger shard, but then you have the problem of how you move tenants from one shard to another as they grow.

- If you ever need to support features that connect data across multiple tenants, these become harder to implement if you need to join data across multiple shards.

Sharding of Key-Value Data

Say you have a large amount of data, and you want to shard it. How do you decide which records to store on which nodes?

Our goal with sharding is to spread the data and the query load evenly across nodes. If every node takes a fair share, then—in theory—10 nodes should be able to handle 10 times as much data and 10 times the read and write throughput of a single node (ignoring replication). Moreover, if we add or remove a node, we want to be able to *rebalance* the load so that it is evenly distributed across the 11 (when adding) or the remaining 9 (when removing) nodes.

If the sharding is unfair, so that some shards have more data or queries than others, we call it *skewed*. The presence of skew makes sharding much less effective. In an extreme case, all the load could end up on one shard, so 9 out of 10 nodes are idle and your bottleneck is the single busy node. A shard with disproportionately high load is called a *hot shard* or *hot spot*. If

there's one key with a particularly high load (e.g., a celebrity in a social network), we call it a *hot key*.

Therefore we need an algorithm that takes as input the partition key of a record, and tells us which shard that record is in. In a key-value store the partition key is usually the key, or the first part of the key. In a relational model the partition key might be some column of a table (not necessarily its primary key). That algorithm needs to be amenable to rebalancing in order to relieve hot spots.

Sharding by Key Range

One way of sharding is to assign a contiguous range of partition keys (from some minimum to some maximum) to each shard, like the volumes of a paper encyclopedia, as illustrated in [Figure 7-2](#). In this example, an entry's partition key is its title. If you want to look up the entry for a particular title, you can easily determine which shard contains that entry by finding the volume whose key range contains the title you're looking for, and thus pick the correct book off the shelf.

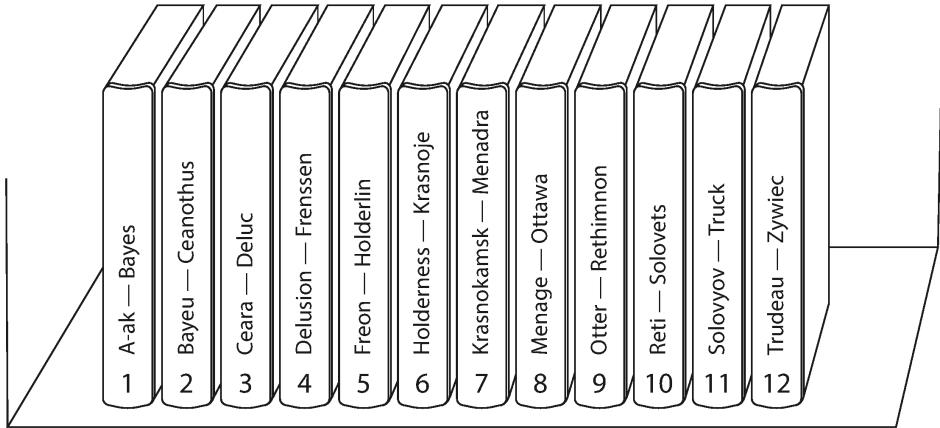


Figure 7-2. A print encyclopedia is sharded by key range.

The ranges of keys are not necessarily evenly spaced, because your data may not be evenly distributed. For example, in [Figure 7-2](#), volume 1 contains words starting with A and B, but volume 12 contains words starting with T, U, V, W, 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 shard boundaries need to adapt to the data.

The shard boundaries might be chosen manually by an administrator, or the database can choose them automatically. Manual key-range sharding is used by Vitess (a sharding layer for MySQL), for example; the automatic variant is used by Bigtable, its open source equivalent HBase, the range-based sharding option in MongoDB, CockroachDB, RethinkDB, and

FoundationDB [6]. YugabyteDB offers both manual and automatic tablet splitting.

Within each shard, keys are stored in sorted order (e.g., in a B-tree or SSTables, as discussed in [Chapter 4](#)). 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 [“Multidimensional and Full-Text Indexes”](#)). For example, consider an application that stores data from a network of sensors, where the key is the timestamp of the measurement. Range scans are very useful in this case, because they let you easily fetch, say, all the readings from a particular month.

A downside of key range sharding is that you can easily get a hot shard if there are a lot of writes to nearby keys. For example, if the key is a timestamp, then the shards correspond to ranges of time—e.g., one shard per month. Unfortunately, if you write data from the sensors to the database as the measurements happen, all the writes end up going to the same shard (the one for this month), so that shard can be overloaded with writes while others sit idle [13].

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 ID so that the key ordering is first by sensor ID and then by timestamp. Assuming you have many sensors active at the same time, the write load will end up more evenly spread across the shards. The downside is that when you want to fetch the values of multiple sensors within a time range, you now need to perform a separate range query for each sensor.

Rebalancing key-range sharded data

When you first set up your database, there are no key ranges to split into shards. Some databases, such as HBase and MongoDB, allow you to configure an initial set of shards on an empty database, which is called *pre-splitting*. This requires that you already have some idea of what the key distribution is going to look like, so that you can choose appropriate key range boundaries [14].

Later on, as your data volume and write throughput grow, a system with key-range sharding grows by splitting an existing shard into two or more smaller shards, each of which holds a contiguous sub-range of the original shard's key range. The resulting smaller shards can then be distributed across multiple nodes. If large amounts of data are deleted, you may also need to merge several adjacent shards that have become small into

one bigger one. This process is similar to what happens at the top level of a B-tree (see “[B-Trees](#)”).

With databases that manage shard boundaries automatically, a shard split is typically triggered by:

- the shard reaching a configured size (for example, on HBase, the default is 10 GB), or
- in some systems, the write throughput being persistently above some threshold. Thus, a hot shard may be split even if it is not storing a lot of data, so that its write load can be distributed more uniformly.

An advantage of key-range sharding is that the number of shards adapts to the data volume. If there is only a small amount of data, a small number of shards is sufficient, so overheads are small; if there is a huge amount of data, the size of each individual shard is limited to a configurable maximum [\[15\]](#).

A downside of this approach is that splitting a shard is an expensive operation, since it requires all of its data to be rewritten into new files, similarly to a compaction in a log-structured storage engine. A shard that needs splitting is often

also one that is under high load, and the cost of splitting can exacerbate that load, risking it becoming overloaded.

Sharding by Hash of Key

Key-range sharding is useful if you want records with nearby (but different) partition keys to be grouped into the same shard; for example, this might be the case with timestamps. If you don't care whether partition keys are near each other (e.g., if they are tenant IDs in a multitenant application), a common approach is to first hash the partition key before mapping it to a shard.

A good hash function takes skewed data and makes it uniformly distributed. Say you have a 32-bit hash function that takes a string. Whenever you give it a new string, it returns a seemingly random number between 0 and $2^{32} - 1$. Even if the input strings are very similar, their hashes are evenly distributed across that range of numbers (but the same input always produces the same output).

For sharding purposes, the hash function need not be cryptographically strong: for example, MongoDB uses MD5, whereas Cassandra and ScyllaDB use Murmur3. Many programming languages have simple hash functions built in (as

they are used for hash tables), but they may not be suitable for sharding: for example, in Java's `Object.hashCode()` and Ruby's `Object#hash`, the same key may have a different hash value in different processes, making them unsuitable for sharding [16].

Hash modulo number of nodes

Once you have hashed the key, how do you choose which shard to store it in? Maybe your first thought is to take the hash value *modulo* the number of nodes in the system (using the `%` operator in many programming languages). For example, `hash(key) % 10` would return a number between 0 and 9 (if we write the hash as a decimal number, the hash `% 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 have to be moved from one node to another. [Figure 7-3](#) shows what happens when you have three nodes and add a fourth. Before the rebalancing, node 0 stored the keys whose hashes are 0, 3, 6, 9, and so on. After adding the fourth node, the key with hash 3 has moved to node 3, the key with hash 6 has moved to node 2, the key with hash 9 has moved to node 1, and so on.

Before rebalancing (3 nodes):

Node 0 hash(key) % 3 = 0	Node 1 hash(key) % 3 = 1	Node 2 hash(key) % 3 = 2
0 3 6 9 12 15 18 21	1 4 7 10 13 16 19 22	2 5 8 11 14 17 20 23

After rebalancing (4 nodes):

Node 0 hash(key) % 4 = 0	Node 1 hash(key) % 4 = 1	Node 2 hash(key) % 4 = 2	Node 3 hash(key) % 4 = 3
0 4 8 12 16 20	1 5 9 13 17 21	2 6 10 14 18 22	3 7 11 15 19 23

Figure 7-3. Assigning keys to nodes by hashing the key and taking it modulo the number of nodes. Changing the number of nodes results in many keys moving from one node to another.

The *mod N* function is easy to compute, but it leads to very inefficient rebalancing because there is a lot of unnecessary movement of records from one node to another. We need an approach that doesn't move data around more than necessary.

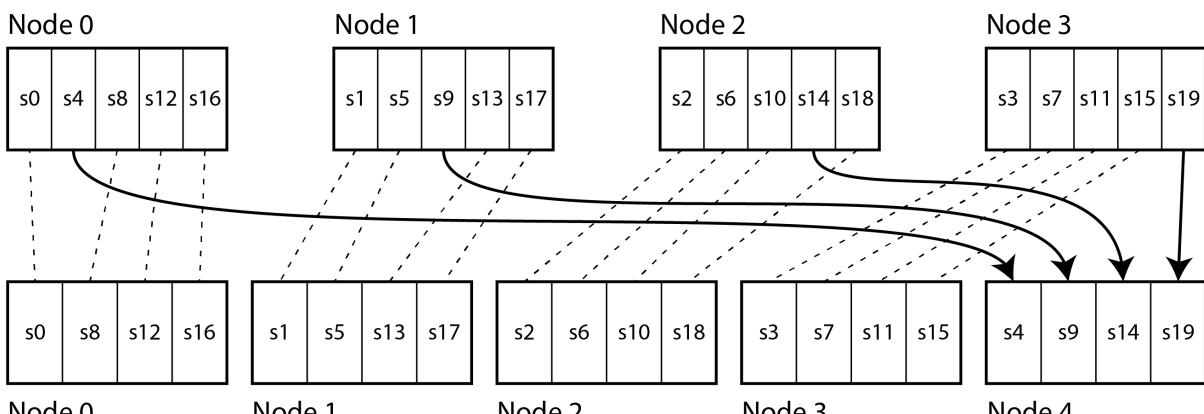
Fixed number of shards

One simple but widely-used solution is to create many more shards than there are nodes, and to assign several shards to each node. For example, a database running on a cluster of 10 nodes may be split into 1,000 shards from the outset so that 100 shards are assigned to each node. A key is then stored in shard

number $hash(key) \% 1,000$, and the system separately keeps track of which shard is stored on which node.

Now, if a node is added to the cluster, the system can reassign some of the shards from existing nodes to the new node until they are fairly distributed once again. This process is illustrated in [Figure 7-4](#). If a node is removed from the cluster, the same happens in reverse.

Before rebalancing (4 nodes in cluster)



After rebalancing (5 nodes in cluster)

Legend:

- - - - - shard remains on the same node
- shard migrated to another node

Figure 7-4. Adding a new node to a database cluster with multiple shards per node.

In this model, only entire shards are moved between nodes, which is cheaper than splitting shards. The number of shards does not change, nor does the assignment of keys to shards. The only thing that changes is the assignment of shards 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 shards is used for any reads and writes that happen while the transfer is in progress.

It's common to choose the number of shards to be a number that is divisible by many factors, so that the dataset can be evenly split across various different numbers of nodes—not requiring the number of nodes to be a power of 2, for example [4]. You can even account for mismatched hardware in your cluster: by assigning more shards to nodes that are more powerful, you can make those nodes take a greater share of the load.

This approach to sharding is used in Citus (a sharding layer for PostgreSQL), Riak, Elasticsearch, and Couchbase, among others. It works well as long as you have a good estimate of how many shards you will need when you first create the database. You can then add or remove nodes easily, subject to the limitation that you can't have more nodes than you have shards.

If you find the originally configured number of shards to be wrong—for example, if you have reached a scale where you need more nodes than you have shards—then an expensive resharding operation is required. It needs to split each shard

and write it out to new files, using a lot of additional disk space in the process. Some systems don't allow resharding while concurrently writing to the database, which makes it difficult to change the number of shards without downtime.

Choosing the right number of shards is difficult if the total size of the dataset is highly variable (for example, if it starts small but may grow much larger over time). Since each shard contains a fixed fraction of the total data, the size of each shard grows proportionally to the total amount of data in the cluster. If shards are very large, rebalancing and recovery from node failures become expensive. But if shards are too small, they incur too much overhead. The best performance is achieved when the size of shards is "just right," neither too big nor too small, which can be hard to achieve if the number of shards is fixed but the dataset size varies.

Sharding by hash range

If the required number of shards can't be predicted in advance, it's better to use a scheme in which the number of shards can adapt easily to the workload. The aforementioned key-range sharding scheme has this property, but it has a risk of hot spots when there are a lot of writes to nearby keys. One solution is to combine key-range sharding with a hash function so that each

shard contains a range of *hash values* rather than a range of *keys*.

[Figure 7-5](#) shows an example using a 16-bit hash function that returns a number between 0 and $65,535 = 2^{16} - 1$ (in reality, the hash is usually 32 bits or more). Even if the input keys are very similar (e.g., consecutive timestamps), their hashes are uniformly distributed across that range. We can then assign a range of hash values to each shard: for example, values between 0 and 16,383 to shard 0, values between 16,384 and 32,767 to shard 1, and so on.

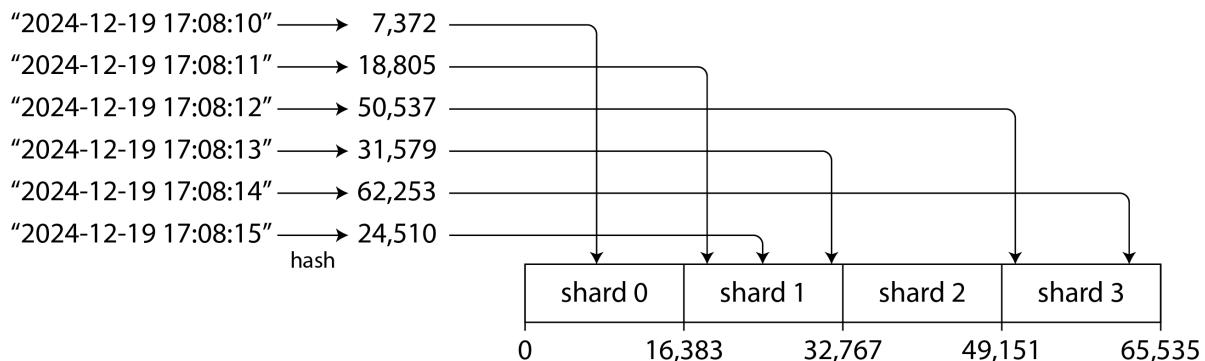


Figure 7-5. Assigning a contiguous range of hash values to each shard.

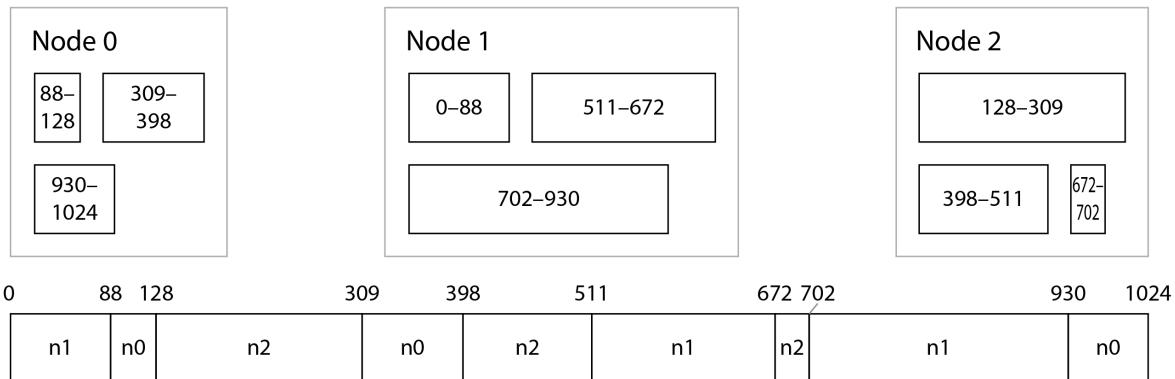
Like with key-range sharding, a shard in hash-range sharding can be split when it becomes too big or too heavily loaded. This is still an expensive operation, but it can happen as needed, so the number of shards adapts to the volume of data rather than being fixed in advance.

The downside compared to key-range sharding is that range queries over the partition key are not efficient, as keys in the range are now scattered across all the shards. However, if keys consist of two or more columns, and the partition key is only the first of these columns, you can still perform efficient range queries over the second and later columns: as long as all records in the range query have the same partition key, they will be in the same shard.

PARTITIONING AND RANGE QUERIES IN DATA WAREHOUSES

Data warehouses such as BigQuery, Snowflake, and Delta Lake support a similar indexing approach, though the terminology differs. In BigQuery, for example, the partition key determines which partition a record resides in while “cluster columns” determine how records are sorted within the partition. Snowflake assigns records to “micro-partitions” automatically, but allows users to define cluster keys for a table. Delta Lake supports both manual and automatic partition assignment, and supports cluster keys. Clustering data not only improves range scan performance, but can improve compression and filtering performance as well.

Hash-range sharding is used in YugabyteDB and DynamoDB [17], and is an option in MongoDB. Cassandra and ScyllaDB use a variant of this approach that is illustrated in [Figure 7-6](#): the space of hash values is split into a number of ranges proportional to the number of nodes (3 ranges per node in [Figure 7-6](#), but actual numbers are 8 per node in Cassandra by default, and 256 per node in ScyllaDB), with random boundaries between those ranges. This means some ranges are bigger than others, but by having multiple ranges per node those imbalances tend to even out [15, 18].



After adding Node 3 (with hash range boundaries 60, 276, and 551):

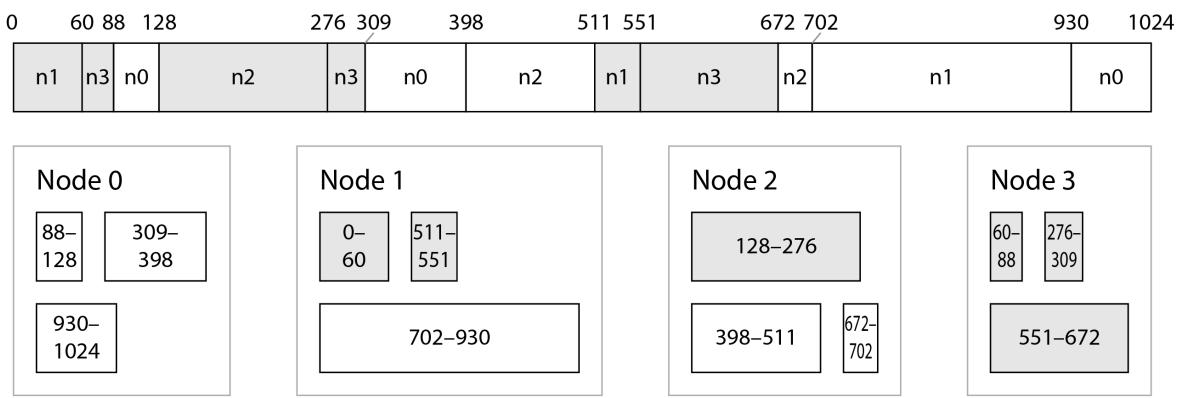


Figure 7-6. Cassandra and ScyllaDB split the range of possible hash values (here 0–1023) into contiguous ranges with random boundaries, and assign several ranges to each node.

When nodes are added or removed, range boundaries are added and removed, and shards are split or merged accordingly [19]. In the example of [Figure 7-6](#), when node 3 is added, node 1 transfers parts of two of its ranges to node 3, and node 2 transfers part of one of its ranges to node 3. This has the effect of giving the new node an approximately fair share of the dataset, without transferring more data than necessary from one node to another.

Consistent hashing

A *consistent hashing* algorithm is a hash function that maps keys to a specified number of shards in a way that satisfies two properties:

1. the number of keys mapped to each shard is roughly equal, and
2. when the number of shards changes, as few keys as possible are moved from one shard to another.

Note that *consistent* here has nothing to do with replica consistency (see [Chapter 6](#)) or ACID consistency (see [Chapter 8](#)), but rather describes the tendency of a key to stay in the same shard as much as possible.

The sharding algorithm used by Cassandra and ScyllaDB is similar to the original definition of consistent hashing [20], but several other consistent hashing algorithms have also been proposed [21], such as *highest random weight*, also known as *rendezvous hashing* [22], and *jump consistent hash* [23]. With Cassandra's algorithm, if one node is added, a small number of existing shards are split into sub-ranges; on the other hand, with rendezvous and jump consistent hashes, the new node is assigned individual keys that were previously scattered across

all of the other nodes. Which one is preferable depends on the application.

Skewed Workloads and Relieving Hot Spots

Consistent hashing ensures that keys are uniformly distributed across nodes, but that doesn't mean that the actual load is uniformly distributed. If the workload is highly skewed—that is, the amount of data under some partition keys is much greater than other keys, or if the rate of requests to some keys is much higher than to others—you can still end up with some servers being overloaded while others sit almost idle.

For example, on a social media site, a celebrity user with millions of followers may cause a storm of activity when they do something [24]. This event can result in a large volume of reads and writes to the same key (where the partition key is perhaps the user ID of the celebrity, or the ID of the action that people are commenting on).

In such situations, a more flexible sharding policy is required [25, 26]. A system that defines shards based on ranges of keys (or ranges of hashes) makes it possible to put an individual hot

key in a shard by its own, and perhaps even assigning it a dedicated machine [27].

It's also possible to compensate for skew at the application level. 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 shards.

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. The volume of reads to each shard of the hot key is not reduced; only the write load is split. This technique also requires additional bookkeeping: it only makes sense to append the random number for the small number of hot keys; for the vast majority of keys with low write throughput this would be unnecessary overhead. Thus, you also need some way of keeping track of which keys are being split, and a process for converting a regular key into a specially-managed hot key.

The problem is further compounded by change of load over time: for example, a particular social media post that has gone viral may experience high load for a couple of days, but

thereafter it's likely to calm down again. Moreover, some keys may be hot for writes while others are hot for reads, necessitating different strategies for handling them.

Some systems (especially cloud services designed for large scale) have automated approaches for dealing with hot shards; for example, Amazon calls it *heat management* [28] or *adaptive capacity* [17]. The details of how these systems work go beyond the scope of this book.

Operations: Automatic or Manual Rebalancing

There is one important question with regard to rebalancing that we have glossed over: does the splitting of shards and rebalancing happen automatically or manually?

Some systems automatically decide when to split shards and when to move them from one node to another, without any human interaction, while others leave sharding to be explicitly configured by an administrator. There is also a middle ground: for example, Couchbase and Riak generate a suggested shard 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, and such systems can even auto-scale to adapt to changes in workload. Cloud databases such as DynamoDB are promoted as being able to automatically add and remove shards to adapt to big increases or decreases of load within a matter of minutes [17, 29].

However, automatic shard management can also be unpredictable. Rebalancing is an expensive operation, because it requires rerouting requests and moving a large amount of data from one node to another. If it is not done carefully, this process can overload the network or the nodes, and it might harm the performance of other requests. The system must continue processing writes while the rebalancing is in progress; if a system is near its maximum write throughput, the shard-splitting process might not even be able to keep up with the rate of incoming writes [29].

Such automation can be dangerous in combination with automatic failure detection. For example, say one node is overloaded and is temporarily slow to respond to requests. The other nodes conclude that the overloaded node is dead, and automatically rebalance the cluster to move load away from it. This puts additional load on other nodes and the network,

making the situation worse. There is a risk of causing a cascading failure where other nodes become overloaded and are also falsely suspected of being down.

For that reason, it can be a good thing to have a human in the loop for rebalancing. It's slower than a fully automatic process, but it can help prevent operational surprises.

Request Routing

We have discussed how to shard a dataset across multiple nodes, and how to rebalance those shards as nodes are added or removed. Now let's move on to the question: if you want to read or write a particular key, how do you know which node—i.e., which IP address and port number—you need to connect to?

We call this problem *request routing*, and it's very similar to *service discovery*, which we previously discussed in [“Load balancers, service discovery, and service meshes”](#). The biggest difference between the two is that with services running application code, each instance is usually stateless, and a load balancer can send a request to any of the instances. With

sharded databases, a request for a key can only be handled by a node that is a replica for the shard containing that key.

This means that request routing has to be aware of the assignment from keys to shards, and from shards to nodes. On a high level, there are a few different approaches to this problem (illustrated in [Figure 7-7](#)):

1. Allow clients to contact any node (e.g., via a round-robin load balancer). If that node coincidentally owns the shard to which the request applies, it can handle the request directly; otherwise, it forwards the request to the appropriate node, receives the reply, and passes the reply along to the client.
2. Send all requests from clients to a routing tier first, which determines the node that should handle each request and forwards it accordingly. This routing tier does not itself handle any requests; it only acts as a shard-aware load balancer.
3. Require that clients be aware of the sharding and the assignment of shards to nodes. In this case, a client can connect directly to the appropriate node, without any intermediary.

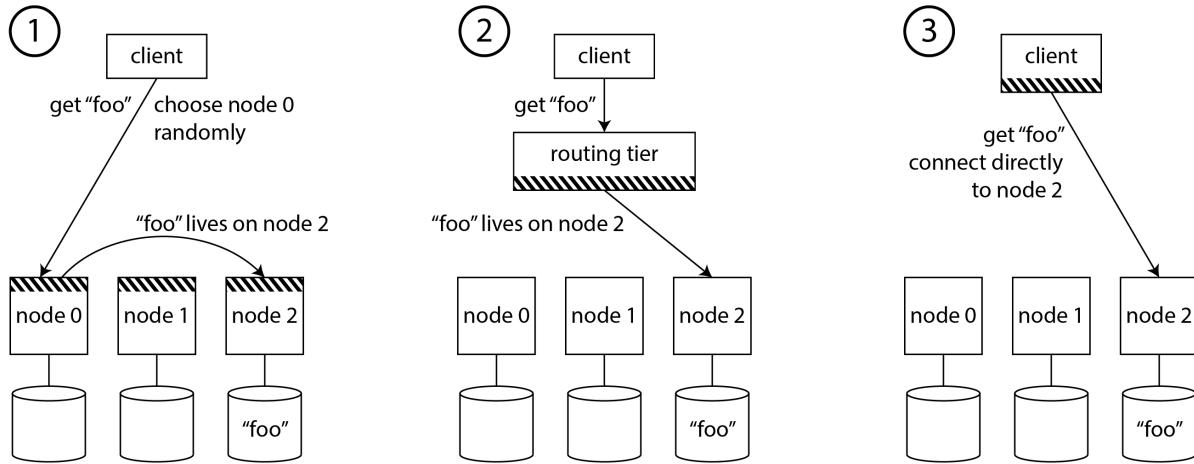


Figure 7-7. Three different ways of routing a request to the right node.

In all cases, there are some key problems:

- Who decides which shard should live on which node? It's simplest to have a single coordinator making that decision, but in that case how do you make it fault-tolerant in case the node running the coordinator goes down? And if the coordinator role can failover to another node, how do you prevent a split-brain situation (see [“Handling Node Outages”](#)) where two different coordinators make contradictory shard assignments?
- How does the component performing the routing (which may be one of the nodes, or the routing tier, or the client) learn about changes in the assignment of shards to nodes?
- While a shard is being moved from one node to another, there is a cutover period during which the new node has

taken over, but requests to the old node may still be in flight. How do you handle those?

Many distributed data systems rely on a separate coordination service such as ZooKeeper or etcd to keep track of shard assignments, as illustrated in [Figure 7-8](#). They use consensus algorithms (see [Link to Come]) to provide fault tolerance and protection against split-brain. Each node registers itself in ZooKeeper, and ZooKeeper maintains the authoritative mapping of shards to nodes. Other actors, such as the routing tier or the sharding-aware client, can subscribe to this information in ZooKeeper. Whenever a shard changes ownership, or a node is added or removed, ZooKeeper notifies the routing tier so that it can keep its routing information up to date.

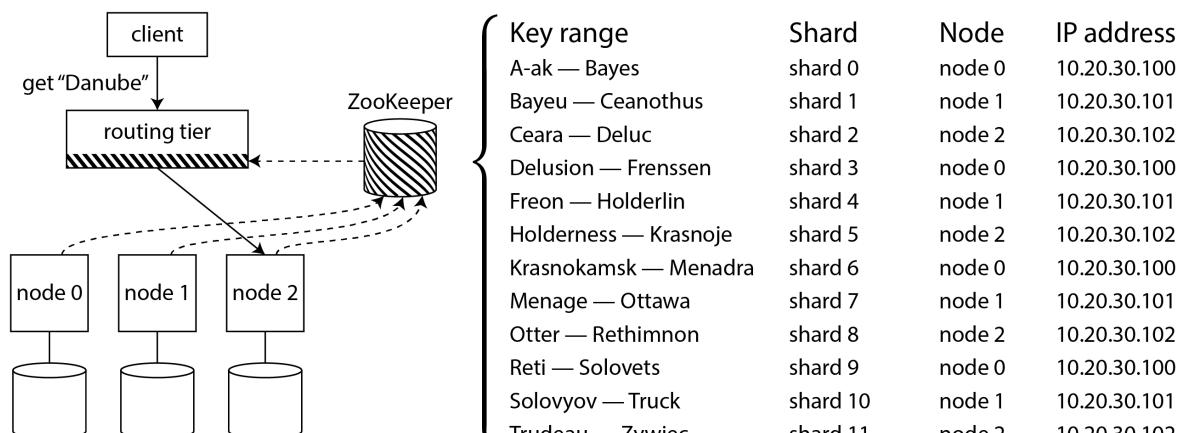


Figure 7-8. Using ZooKeeper to keep track of assignment of shards to nodes.

For example, HBase and SolrCloud use ZooKeeper to manage shard assignment, and Kubernetes uses etcd to keep track of which service instance is running where. MongoDB has a similar architecture, but it relies on its own *config server* implementation and *mongos* daemons as the routing tier. Kafka, YugabyteDB, and TiDB use built-in implementations of the Raft consensus protocol to perform this coordination function.

Cassandra, ScyllaDB, and Riak take a different approach: they use a *gossip protocol* among the nodes to disseminate any changes in cluster state. This provides much weaker consistency than a consensus protocol; it is possible to have split brain, in which different parts of the cluster have different node assignments for the same shard. Leaderless databases can tolerate this because they generally make weak consistency guarantees anyway (see [“Limitations of Quorum Consistency”](#)).

When using a routing tier or when sending requests to a random node, clients still need to find the IP addresses to connect to. These are not as fast-changing as the assignment of shards to nodes, so it is often sufficient to use DNS for this purpose.

This discussion of request routing has focused on finding the shard for an individual key, which is most relevant for sharded

OLTP databases. Analytic databases often use sharding as well, but they typically have a very different kind of query execution: rather than executing in a single shard, a query typically needs to aggregate and join data from many different shards in parallel. We will discuss techniques for such parallel query execution in [Link to Come].

Sharding and Secondary Indexes

The sharding schemes we have discussed so far rely on the client knowing the partition key for any record it wants to access. This is most easily done in a key-value data model, where the partition key is the first part of the primary key (or the entire primary key), and so we can use the partition key to determine the shard, and thus route reads and writes to the node that is responsible for that key.

The situation becomes more complicated if secondary indexes are involved (see also [“Multi-Column and Secondary Indexes”](#)). A secondary index usually doesn’t identify a record uniquely but rather is a way of searching for occurrences of a particular value: find all actions by user `123`, find all articles containing the word `hogwash`, find all cars whose color is `red`, and so on.

Key-value stores often don't have secondary indexes, but they are the bread and butter of relational databases, they are common in document databases too, and they are the *raison d'être* of full-text search engines such as Solr and Elasticsearch. The problem with secondary indexes is that they don't map neatly to shards. There are two main approaches to sharding a database with secondary indexes: local and global indexes.

Local Secondary Indexes

For example, imagine you are operating a website for selling used cars (illustrated in [Figure 7-9](#)). Each listing has a unique ID, and you use that ID as partition key for sharding (for example, IDs 0 to 499 in shard 0, IDs 500 to 999 in shard 1, etc.).

If you want to let users search for cars, allowing them to filter by color and by make, 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. For example, whenever a red car is added to the database, the database shard automatically adds its ID to the list of IDs for the index entry `color:red`. As discussed in [Chapter 4](#), that list of IDs is also called a *postings list*.

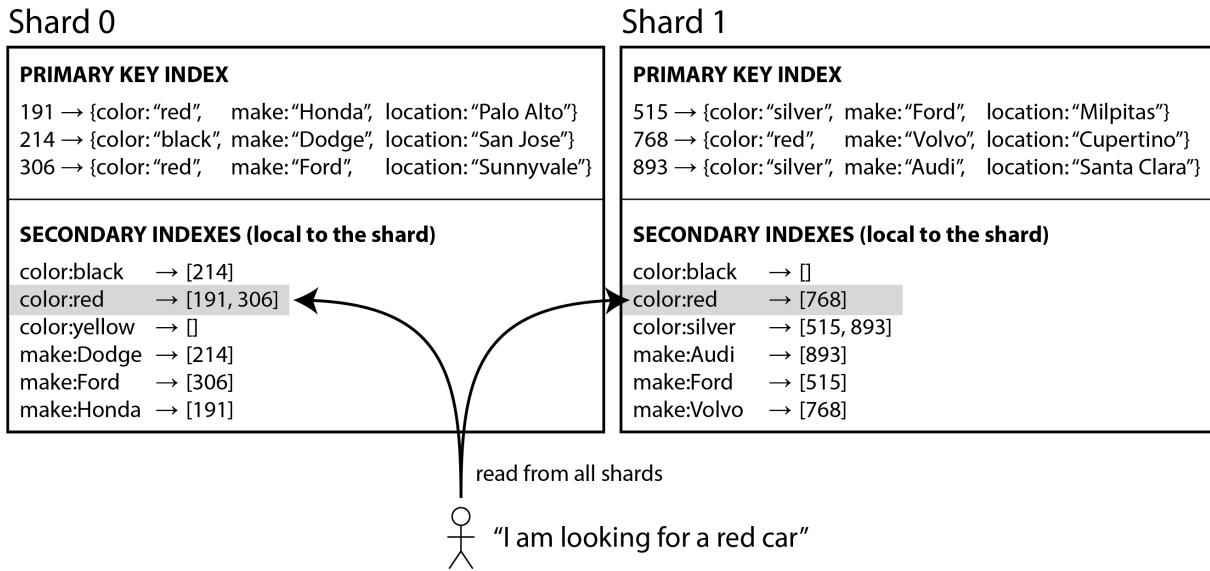


Figure 7-9. Local secondary indexes: each shard indexes only the records within its own shard.

WARNING

If your database only supports a key-value model, you might be tempted to implement a secondary index yourself by creating a mapping from values to IDs in application code. If you go down this route, you need to take great care to ensure your indexes remain consistent with the underlying data. Race conditions and intermittent write failures (where some changes were saved but others weren't) can very easily cause the data to go out of sync—see [“The need for multi-object transactions”](#).

In this indexing approach, each shard is completely separate: each shard maintains its own secondary indexes, covering only the records in that shard. It doesn't care what data is stored in other shards. Whenever you write to the database—to add, remove, or update a records—you only need to deal with the

shard that contains the record that you are writing. For that reason, this type of secondary index is known as a *local index*. In an information retrieval context it is also known as a *document-partitioned index* [30].

When reading from a local secondary index, if you already know the partition key of the record you're looking for, you can just perform the search on the appropriate shard. Moreover, if you only want *some* results, and you don't need all, you can send the request to any shard.

However, if you want all the results and don't know their partition key in advance, you need to send the query to all shards, and combine the results you get back, because the matching records might be scattered across all the shards. In [Figure 7-9](#), red cars appear in both shard 0 and shard 1.

This approach to querying a sharded database can make read queries on secondary indexes quite expensive. Even if you query the shards in parallel, it is prone to tail latency amplification (see [“Use of Response Time Metrics”](#)). It also limits the scalability of your application: adding more shards lets you store more data, but it doesn't increase your query throughput if every shard has to process every query anyway.

Nevertheless, local secondary indexes are widely used [31]: for example, MongoDB, Riak, Cassandra [32], Elasticsearch [33], SolrCloud, and VoltDB [34] all use local secondary indexes.

Global Secondary Indexes

Rather than each shard having its own, local secondary index, we can construct a *global index* that covers data in all shards. However, we can't just store that index on one node, since it would likely become a bottleneck and defeat the purpose of sharding. A global index must also be sharded, but it can be sharded differently from the primary key index.

[Figure 7-10](#) illustrates what this could look like: the IDs of red cars from all shards appear under `color:red` in the index, but the index is sharded so that colors starting with the letters *a* to *r* appear in shard 0 and colors starting with *s* to *z* appear in shard 1. The index on the make of car is partitioned similarly (with the shard boundary being between *f* and *h*).

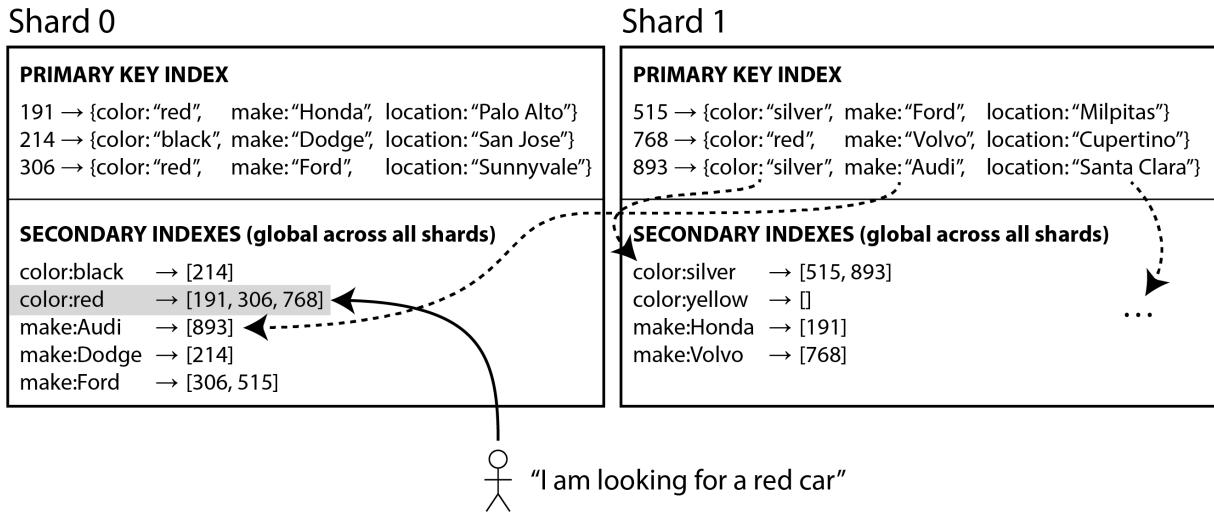


Figure 7-10. A global secondary index reflects data from all shards, and is itself sharded by the indexed value.

This kind of index is also called *term-partitioned* [30]: recall from “[Full-Text Search](#)” that in full-text search, a *term* is a keyword in a text that you can search for. Here we generalise it to mean any value that you can search for in the secondary index.

The global index uses the term as partition key, so that when you’re looking for a particular term or value, you can figure out which shard you need to query. As before, a shard can contain a contiguous range of terms (as in [Figure 7-10](#)), or you can assign terms to shards based on a hash of the term.

Global indexes have the advantage that a query with a single condition (such as *color = red*) only needs to read from a single shard to fetch the postings list. However, if you want to fetch

records and not just IDs, you still have to read from all the shards that are responsible for those IDs.

If you have multiple search conditions or terms (e.g., searching for cars of a certain color and a certain make, or searching for multiple words occurring in the same text), it's likely that those terms will be assigned to different shards. To compute the logical AND of the two conditions, the system needs to find all the IDs that occur in both of the postings lists. That's no problem if the postings lists are short, but if they are long, it can be slow to send them over the network to compute their intersection [\[30\]](#).

Another challenge with global secondary indexes is that writes are more complicated than with local indexes, because writing a single record might affect multiple shards of the index (every term in the document might be on a different shard). This makes it harder to keep the secondary index in sync with the underlying data. One option is to use a distributed transaction to atomically update the shards storing the primary record and its secondary indexes (see [Chapter 8](#) and [\[Link to Come\]](#)).

Global secondary indexes are used by CockroachDB, TiDB, and YugabyteDB; DynamoDB supports both local and global secondary indexes. In the case of DynamoDB, writes are

asynchronously reflected in global indexes, so reads from a global index may be stale (similarly to replication lag, as in “[Problems with Replication Lag](#)”). Nevertheless, global indexes are useful if read throughput is higher than write throughput, and if the postings lists are not too long.

Summary

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

The goal of sharding is to spread the data and query load evenly across multiple machines, avoiding hot spots (nodes with disproportionately high load). This requires choosing a sharding scheme that is appropriate to your data, and rebalancing the shards when nodes are added to or removed from the cluster.

We discussed two main approaches to sharding:

- *Key range sharding*, where keys are sorted, and a shard owns all the keys from some minimum up to some maximum. Sorting has the advantage that efficient range queries are

possible, but there is a risk of hot spots if the application often accesses keys that are close together in the sorted order. In this approach, shards are typically rebalanced by splitting the range into two subranges when a shard gets too big.

- *Hash sharding*, where a hash function is applied to each key, and a shard owns a range of hash values (or another consistent hashing algorithm may be used to map hashes to shards). This method destroys the ordering of keys, making range queries inefficient, but it may distribute load more evenly.

When sharding by hash, it is common to create a fixed number of shards in advance, to assign several shards to each node, and to move entire shards from one node to another when nodes are added or removed. Splitting shards, like with key ranges, is also possible.

It is common to use the first part of the key as the partition key (i.e., to identify the shard), and to sort records within that shard by the rest of the key. That way you can still have efficient range queries among the records with the same partition key.

We also discussed the interaction between sharding and secondary indexes. A secondary index also needs to be sharded, and there are two methods:

- *Local secondary indexes*, where the secondary indexes are stored in the same shard as the primary key and value. This means that only a single shard needs to be updated on write, but a lookup of the secondary index requires reading from all shards.
- *Global secondary indexes*, which are sharded separately based on the indexed values. An entry in the secondary index may refer to records from all shards of the primary key. When a record is written, several secondary index shards may need to be updated; however, a read of the postings list can be served from a single shard (fetching the actual records still requires reading from multiple shards).

Finally, we discussed techniques for routing queries to the appropriate shard, and how a coordination service is often used to keep track of the assignment of shards to nodes.

By design, every shard operates mostly independently—that’s what allows a sharded database to scale to multiple machines. However, operations that need to write to several shards can be problematic: for example, what happens if the write to one shard succeeds, but another fails? We will address that question in the following chapters.

FOOTNOTES

REFERENCES

Claire Giordano. [Understanding partitioning and sharding in Postgres and Citus](#). *citusdata.com*, August 2023. Archived at [perma.cc/8BTK-8959](#)

Brandur Leach. [Partitioning in Postgres, 2022 edition](#). *brandur.org*, October 2022. Archived at [perma.cc/Z5LE-6AKX](#)

Raph Koster. [Database “sharding” came from UO?](#) *raphkoster.com*, January 2009. Archived at [perma.cc/4N9U-5KYF](#)

Garrett Fidalgo. [Herding elephants: Lessons learned from sharding Postgres at Notion](#). *notion.com*, October 2021. Archived at [perma.cc/5J5V-W2VX](#)

Jlrich Drepper. [What Every Programmer Should Know About Memory](#). *akkadia.org*, November 2007. Archived at [perma.cc/NU6Q-DRXZ](#)

ingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. [FoundationDB: A Distributed Unbundled Transactional Key Value Store](#). At *ACM International Conference on Management of Data* (SIGMOD), June 2021.
[doi:10.1145/3448016.3457559](#)

Marco Slot. [Citus 12: Schema-based sharding for PostgreSQL](#). *citusdata.com*, July 2023. Archived at [perma.cc/R874-EC9W](#)

Robisson Oliveira. [Reducing the Scope of Impact with Cell-Based Architecture](#). AWS Well-Architected white paper, Amazon Web Services, September 2023. Archived at [perma.cc/4KWW-47NR](#)

Gwen Shapira. [Things DBs Don't Do - But Should](#). *thenile.dev*, February 2023. Archived at perma.cc/C3J4-JSFW

Malte Schwarzkopf, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. [Position: GDPR Compliance by Construction](#). At *Towards Polystores that manage multiple Databases, Privacy, Security and/or Policy Issues for Heterogenous Data* (Poly), August 2019. [doi:10.1007/978-3-030-33752-0_3](https://doi.org/10.1007/978-3-030-33752-0_3)

Gwen Shapira. [Introducing pg_karnak: Transactional schema migration across tenant databases](#). *thenile.dev*, November 2024. Archived at perma.cc/R5RD-8HR9

Arka Ganguli, Guido Iaquinti, Maggie Zhou, and Rafael Chacón. [Scaling Datastores at Slack with Vitess](#). *slack.engineering*, December 2020. Archived at perma.cc/UW8F-ALJK

Ikai Lan. [App Engine Datastore Tip: Monotonically Increasing Values Are Bad](#). *ikaisays.com*, January 2011. Archived at perma.cc/BPX8-RPJB

Enis Soztutar. [Apache HBase Region Splitting and Merging](#). *cloudera.com*, February 2013. Archived at perma.cc/S9HS-2X2C

Eric Evans. [Rethinking Topology in Cassandra](#). At *Cassandra Summit*, June 2013. Archived at perma.cc/2DKM-F438

Martin Kleppmann. [Java's hashCode Is Not Safe for Distributed Systems](#). *martin.kleppmann.com*, June 2012. Archived at perma.cc/LK5U-VZSN

Mostafa Elhemali, Niall Gallagher, Nicholas Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somu Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. [Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service](#). At *USENIX Annual Technical Conference (ATC)*, July 2022.

Brandon Williams. [Virtual Nodes in Cassandra 1.2](#). *datastax.com*, December 2012.
Archived at [perma.cc/N385-EQXV](#)

Branimir Lambov. [New Token Allocation Algorithm in Cassandra 3.0](#). *datastax.com*, January 2016. Archived at [perma.cc/2BG7-LDWY](#)

David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. [Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web](#). In *29th Annual ACM Symposium on Theory of Computing* (STOC), May 1997. [doi:10.1145/258533.258660](#)

Damian Gryski. [Consistent Hashing: Algorithmic Tradeoffs](#). *dgryski.medium.com*, April 2018. Archived at [perma.cc/B2WF-TYQ8](#)

David G. Thaler and Chinya V. Ravishankar. [Using name-based mappings to increase hit rates](#). *IEEE/ACM Transactions on Networking*, volume 6, issue 1, pages 1–14, February 1998. [doi:10.1109/90.663936](#)

John Lamping and Eric Veach. [A Fast, Minimal Memory, Consistent Hash Algorithm](#). *arxiv.org*, June 2014.

Samuel Axon. [3% of Twitter's Servers Dedicated to Justin Bieber](#). *mashable.com*, September 2010. Archived at [perma.cc/F35N-CGVX](#)

Gerald Guo and Thawan Kooburat. [Scaling services with Shard Manager](#). *engineering.fb.com*, August 2020. Archived at [perma.cc/EFS3-XQYT](#)

Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. [Shard Manager: A Generic Shard Management Framework for Geo-distributed Applications](#). *28th ACM SIGOPS Symposium on Operating Systems Principles* (SOSP), pages 553–569, October 2021. [doi:10.1145/3477132.3483546](#)

Scott Lystig Fritchie. [A Critique of Resizable Hash Tables: Riak Core & Random Slicing](#). [infoq.com](#), August 2018. Archived at [perma.cc/RPX7-7BLN](#)

Andy Warfield. [Building and operating a pretty big storage system called S3](#). [allthingsdistributed.com](#), July 2023. Archived at [perma.cc/6S7P-GLM4](#)

Rich Houlihan. [DynamoDB adaptive capacity: smooth performance for chaotic workloads \(DAT327\)](#). At AWS *re:Invent*, November 2017.

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. [Introduction to Information Retrieval](#). Cambridge University Press, 2008. ISBN: 978-0-521-86571-5, available online at [nlp.stanford.edu/IR-book](#)

Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and Jimmy Lin. [Earlybird: Real-Time Search at Twitter](#). At *28th IEEE International Conference on Data Engineering* (ICDE), April 2012. [doi:10.1109/ICDE.2012.149](#)

Nadav Har'El. [Indexing in Cassandra 3](#). [github.com](#), April 2017. Archived at [perma.cc/3ENV-8T9P](#)

Zachary Tong. [Customizing Your Document Routing](#). [elastic.co](#), June 2013. Archived at [perma.cc/97VM-MREN](#)

Andrew Pavlo. [H-Store Frequently Asked Questions](#). [hstore.cs.brown.edu](#), October 2013. Archived at [perma.cc/X3ZA-DW6Z](#)

Chapter 8. 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)

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book. The GitHub repo for this book is <https://github.com/ept/ddia2-feedback>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out on GitHub.

In the harsh reality of data systems, many things can go wrong:

- The database software or hardware may fail at any time (including in the middle of a write operation).
- The application may crash at any time (including halfway through a series of operations).
- Interruptions in the network can unexpectedly cut off the application from the database, or one database node from another.
- Several clients may write to the database at the same time, overwriting each other’s changes.

- A client may read data that doesn't make sense because it has only partially been updated.
- Race conditions between clients can cause surprising bugs.

In order to be reliable, a system has to deal with these faults and ensure that they don't cause catastrophic failure of the entire system. However, implementing fault-tolerance mechanisms is a lot of work. It requires a lot of careful thinking about all the things that can go wrong, and a lot of testing to ensure that the solution actually works.

For decades, *transactions* have been the mechanism of choice for simplifying these issues. A transaction is a way for an application to group several reads and writes together into a logical unit. Conceptually, all the reads and writes in a transaction are executed as one operation: either the entire transaction succeeds (*commit*) or it fails (*abort*, *rollback*). If it fails, the application can safely retry. With transactions, error handling becomes much simpler for an application, because it doesn't need to worry about partial failure—i.e., the case where some operations succeed and some fail (for whatever reason).

If you have spent years working with transactions, they may seem obvious, but we shouldn't take them for granted.

Transactions are not a law of nature; they were created with a

purpose, namely to *simplify the programming model* for applications accessing a database. By using transactions, the application is free to ignore certain potential error scenarios and concurrency issues, because the database takes care of them instead (we call these *safety guarantees*).

Not every application needs transactions, and sometimes there are advantages to weakening transactional guarantees or abandoning them entirely (for example, to achieve higher performance or higher availability). Some safety properties can be achieved without transactions. On the other hand, transactions can prevent a lot of grief: for example, the technical cause behind the Post Office Horizon scandal (see “[How Important Is Reliability?](#)”) was probably a lack of ACID transactions in the underlying accounting system [1].

How do you figure out whether you need transactions? In order to answer that question, we first need to understand exactly what safety guarantees transactions can provide, and what costs are associated with them. Although transactions seem straightforward at first glance, there are actually many subtle but important details that come into play.

In this chapter, we will examine many examples of things that can go wrong, and explore the algorithms that databases use to

guard against those issues. We will go especially deep in the area of concurrency control, discussing various kinds of race conditions that can occur and how databases implement isolation levels such as *read committed*, *snapshot isolation*, and *serializability*.

This chapter applies to both single-node and distributed databases; in [Link to Come] we will focus the discussion on the particular challenges that arise only in distributed systems.

What Exactly Is a Transaction?

Almost all relational databases today, and some nonrelational databases, support transactions. Most of them follow the style that was introduced in 1975 by IBM System R, the first SQL database [2, 3, 4]. Although some implementation details have changed, the general idea has remained virtually the same for 50 years: the transaction support in MySQL, PostgreSQL, Oracle, SQL Server, etc., is uncannily similar to that of System R.

In the late 2000s, nonrelational (NoSQL) databases started gaining popularity. They aimed to improve upon the relational status quo by offering a choice of new data models (see [Chapter 3](#)), and by including replication ([Chapter 6](#)) and sharding ([Chapter 7](#)) by default. Transactions were the main

casualty of this movement: many of this generation of databases abandoned transactions entirely, or redefined the word to describe a much weaker set of guarantees than had previously been understood.

The hype around NoSQL distributed databases led to a popular belief that transactions were fundamentally unscalable, and that any large-scale system would have to abandon transactions in order to maintain good performance and high availability. More recently, that belief has turned out to be wrong. So-called “NewSQL” databases such as CockroachDB [5], TiDB [6], Spanner [7], FoundationDB [8], and Yugabyte have shown that transactional systems can scale to large data volumes and high throughput. These systems combine sharding with consensus protocols ([Link to Come]) to provide strong ACID guarantees at scale.

However, that doesn’t mean that every system must be transactional either: like every other technical design choice, transactions have advantages and limitations. In order to understand those trade-offs, let’s go into the details of the guarantees that transactions can provide—both in normal operation and in various extreme (but realistic) circumstances.

The Meaning of ACID

The safety guarantees provided by transactions are often described by the well-known acronym *ACID*, which stands for *Atomicity*, *Consistency*, *Isolation*, and *Durability*. It was coined in 1983 by Theo Härder and Andreas Reuter [9] 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* [10]. 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* [11]. 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 multi-threaded programming, if one thread executes an atomic operation, that means there is no way that another thread could see the half-finished result of the operation. The system can only be in the state it was before the operation or after the operation, not something in between.

By contrast, in the context of ACID, atomicity is *not* about concurrency. It does not describe what happens if several processes try to access the same data at the same time, because that is covered under the letter *I*, for *isolation* (see “[Isolation](#)”).

Rather, ACID atomicity describes what happens if a client wants to make several writes, but a fault occurs after some of the writes have been processed—for example, a process crashes, a network connection is interrupted, a disk becomes full, or some integrity constraint is violated. If the writes are grouped together into an atomic transaction, and the transaction cannot be completed (*committed*) due to a fault, then the transaction is *aborted* and the database must discard or undo any writes it has made so far in that transaction.

Without atomicity, if an error occurs partway through making multiple changes, it's difficult to know which changes have taken effect and which haven't. The application could try again, but that risks making the same change twice, leading to duplicate or incorrect data. Atomicity simplifies this problem: if a transaction was aborted, the application can be sure that it didn't change anything, so it can safely be retried.

The ability to abort a transaction on error and have all writes from that transaction discarded is the defining feature of ACID atomicity. Perhaps *abortability* would have been a better term than *atomicity*, but we will stick with *atomicity* since that's the usual word.

Consistency

The word *consistency* is terribly overloaded:

- In [Chapter 6](#) we discussed *replica consistency* and the issue of *eventual consistency* that arises in asynchronously replicated systems (see [“Problems with Replication Lag”](#)).
- A *consistent snapshot* of a database, e.g. for a backup, is a snapshot of the entire database as it existed at one moment in time. More precisely, it is consistent with the happens-before relation (see [“The “happens-before” relation and](#)

concurrency): that is, if the snapshot contains a value that was written at a particular time, then it also reflects all the writes that happened before that value was written.

- *Consistent hashing* is an approach to sharding that some systems use for rebalancing (see “Consistent hashing”).
- In the CAP theorem (see [Link to Come]), the word *consistency* is used to mean *linearizability* (see [Link to Come]).
- 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 five 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. (An invariant may be temporarily violated during transaction execution, but it should be satisfied again at transaction commit.)

If you want the database to enforce your invariants, you need to declare them as *constraints* as part of the schema. For example, foreign key constraints, uniqueness constraints, or check constraints (which restrict the values that can appear in an individual row) are often used to model specific types of invariants. More complex consistency requirements can sometimes be modeled using triggers or materialized views [12].

However, complex invariants can be difficult or impossible to model using the constraints that databases usually provide. In that case, it's the application's responsibility to define its transactions correctly so that they preserve consistency. If you write bad data that violates your invariants, but you haven't declared those invariants, the database can't stop you. As such, the C in ACID often depends on how the application uses the database, and it's not a property of the database alone.

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 8-1](#) 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 8-1](#) the counter should have increased from 42 to 44, because two increments happened, but it actually only went to 43 because of the race condition.

Isolation in the sense of ACID means that concurrently executing transactions are isolated from each other: they cannot step on each other's toes. The classic database textbooks formalize isolation as *serializability*, which means that each transaction can pretend that it is the only transaction running on the entire database. The database ensures that when the transactions have committed, the result is the same as if they had run *serially* (one after another), even though in reality they may have run concurrently [\[13\]](#).

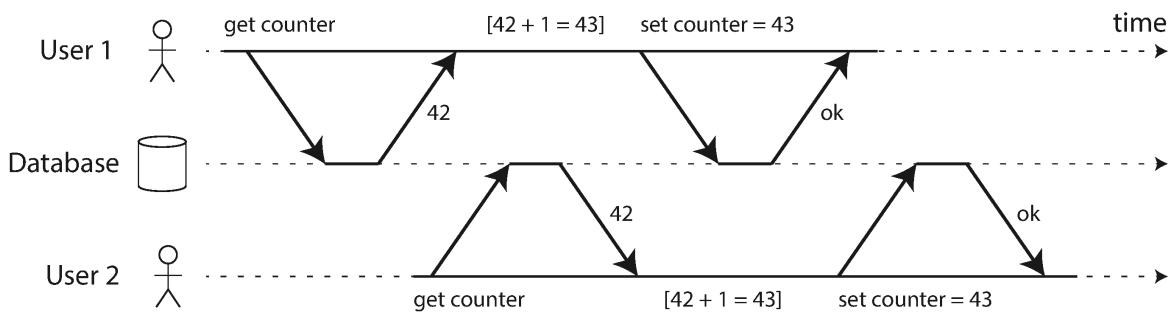


Figure 8-1. A race condition between two clients concurrently incrementing a counter.

However, serializability has a performance cost. In practice, many databases use forms of isolation that are weaker than serializability: that is, they allow concurrent transactions to interfere with each other in limited ways. Some popular databases, such as Oracle, don't even implement it (Oracle has an isolation level called "serializable," but it actually implements *snapshot isolation*, which is a weaker guarantee than serializability [10, 14]). This means that some kinds of race conditions can still occur. We will explore snapshot isolation and other forms of isolation in "["Weak Isolation Levels"](#)".

Durability

The purpose of a database system is to provide a safe place where data can be stored without fear of losing it. *Durability* is the promise that once a transaction has committed successfully,

any data it has written will not be forgotten, even if there is a hardware fault or the database crashes.

In a single-node database, durability typically means that the data has been written to nonvolatile storage such as a hard drive or SSD. Regular file writes are usually buffered in memory before being sent to the disk sometime later, which means they would be lost if there is a sudden power failure; many databases therefore use the `fsync()` system call to ensure the data really has been written to disk. Databases usually also have a write-ahead log or similar (see “[Making B-trees reliable](#)”), which allows them to recover in the event that a crash occurs part way through a write.

In a replicated database, durability may mean that the data has been successfully copied to some number of nodes. In order to provide a durability guarantee, a database must wait until these writes or replications are complete before reporting a transaction as successfully committed. However, as discussed in “[Reliability and Fault Tolerance](#)”, perfect durability does not exist: if all your hard disks and all your backups are destroyed at the same time, there’s obviously nothing your database can do to save you.

REPLICATION AND DURABILITY

Historically, durability meant writing to an archive tape. Then it was understood as writing to a disk or SSD. More recently, it has been adapted to mean replication. Which implementation is better?

The truth is, nothing is perfect:

- If you write to disk and the machine dies, even though your data isn't lost, it is inaccessible until you either fix the machine or transfer the disk to another machine. Replicated systems can remain available.
- A correlated fault—a power outage or a bug that crashes every node on a particular input—can knock out all replicas at once (see “[Reliability and Fault Tolerance](#)”), losing any data that is only in memory. Writing to disk is therefore still relevant for replicated databases.
- In an asynchronously replicated system, recent writes may be lost when the leader becomes unavailable (see “[Handling Node Outages](#)”).
- 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 [15]. Disk firmware can have bugs, just like any other kind of software [16, 17], e.g. causing drives to fail after

exactly 32,768 hours of operation [18]. And `fsync` is hard to use; even PostgreSQL used it incorrectly for over 20 years [19, 20, 21].

- 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 [22, 23]. Filesystem errors on one replica can sometimes spread to other replicas as well [24].
- Data on disk can gradually become corrupted without this being detected [25]. If data has been corrupted for some time, replicas and recent backups may also be corrupted. In this case, you will need to try to restore the data from a historical backup.
- One study of SSDs found that between 30% and 80% of drives develop at least one bad block during the first four years of operation, and only some of these can be corrected by the firmware [26]. Magnetic hard drives have a lower rate of bad sectors, but a higher rate of complete failure than SSDs.
- When a worn-out SSD (that has gone through many write/erase cycles) is disconnected from power, it can start losing data within a timescale of weeks to months, depending on the temperature [27]. This is less of a problem for drives with lower wear levels [28].

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 8-2](#) 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 =
```

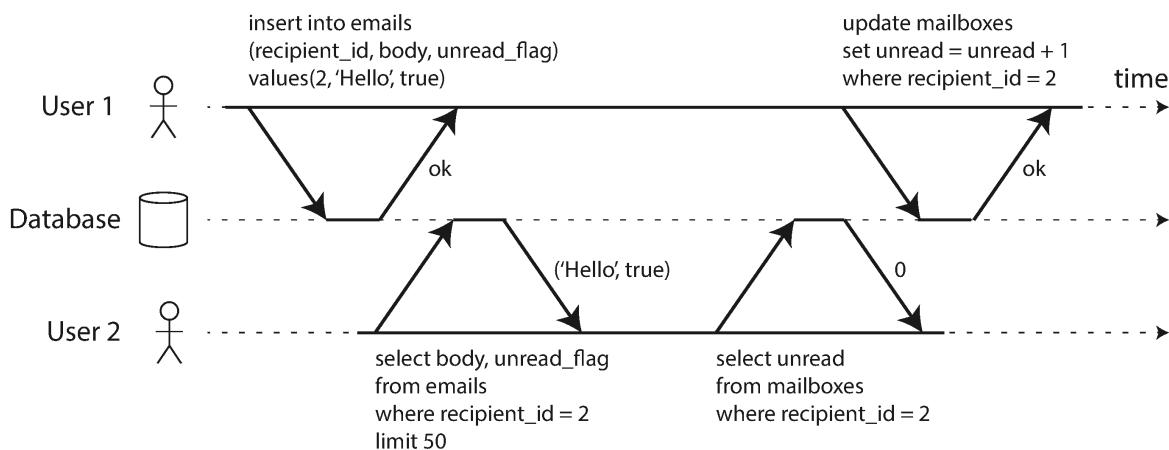


Figure 8-2. Violating isolation: one transaction reads another transaction's uncommitted writes (a “dirty read”).

However, you might find this query to be too slow if there are many emails, and decide to store the number of unread

messages in a separate field (a kind of denormalization, which we discuss in “[Normalization, Denormalization, and Joins](#)”).

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 8-2](#), 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. (If an incorrect counter in an email application seems too insignificant, think of a customer account balance instead of an unread counter, and a payment transaction instead of an email.) Isolation would have prevented this issue by ensuring that user 2 sees either both the inserted email and the updated counter, or neither, but not an inconsistent halfway point.

[Figure 8-3](#) illustrates the need for atomicity: if an error occurs somewhere over the course of the transaction, the contents of the mailbox and the unread counter might become out of sync. In an atomic transaction, if the update to the counter fails, the transaction is aborted and the inserted email is rolled back.

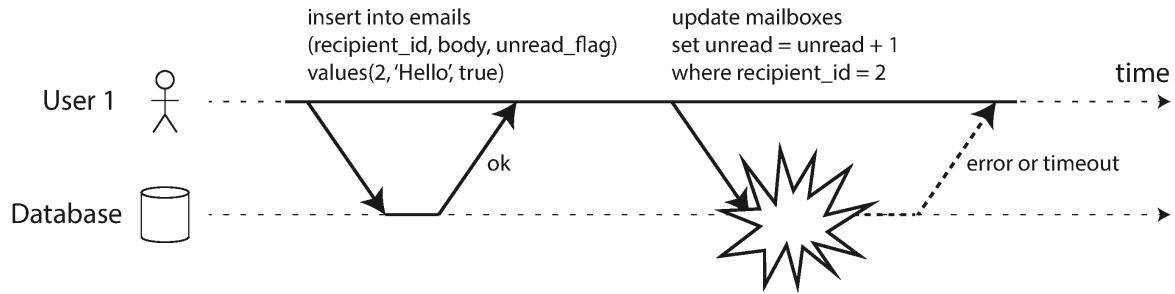


Figure 8-3. Atomicity ensures that if an error occurs any prior writes from that transaction are undone, to avoid an inconsistent state.

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. If the TCP connection is interrupted, the transaction must be aborted.

On the other hand, many nonrelational databases don't have such a way of grouping operations together. Even if there is a multi-object API (for example, a key-value store may have a *multi-put* operation that updates several keys in one operation), that doesn't necessarily mean it has transaction semantics: the command may succeed for some keys and fail for others, leaving the database in a partially updated state.

Single-object writes

Atomicity and isolation also apply when a single object is being changed. For example, imagine you are writing a 20 KB JSON document to a database:

- If the network connection is interrupted after the first 10 KB have been sent, does the database store that unparsable 10 KB fragment of JSON?
- If the power fails while the database is in the middle of overwriting the previous value on disk, do you end up with the old and new values spliced together?
- If another client reads that document while the write is in progress, will it see a partially updated value?

Those issues would be incredibly confusing, so storage engines almost universally aim to provide atomicity and isolation on the level of a single object (such as a key-value pair) on one node. Atomicity can be implemented using a log for crash recovery (see [“Making B-trees reliable”](#)), and isolation can be implemented using a lock on each object (allowing only one thread to access an object at any one time).

Some databases also provide more complex atomic operations, such as an increment operation, which removes the need for a

read-modify-write cycle like that in [Figure 8-1](#). Similarly popular is a *conditional write* operation, which allows a write to happen only if the value has not been concurrently changed by someone else (see [“Conditional writes \(compare-and-swap\)”](#)), similarly to a compare-and-swap (CAS) operation in shared-memory concurrency.

NOTE

Strictly speaking, the term *atomic increment* uses the word *atomic* in the sense of multi-threaded programming. In the context of ACID, it should actually be called an *isolated* or *serializable* increment, but that’s not the usual term.

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”](#)). However, they are not transactions in the usual sense of the word. For example, the “lightweight transactions” feature of Cassandra and ScyllaDB, and Aerospike’s “strong consistency” mode offer linearizable (see [Link to Come]) reads and conditional writes on a single object, but no guarantees across multiple objects.

The need for multi-object transactions

Do we need multi-object transactions at all? Would it be possible to implement any application with only a key-value data model and single-object operations?

There are some use cases in which single-object inserts, updates, and deletes are sufficient. However, in many other cases writes to several different objects need to be coordinated:

- In a relational data model, a row in one table often has a foreign key reference to a row in another table. Similarly, in a graph-like data model, a vertex has edges to other vertices. Multi-object transactions allow you to ensure that these references remain valid: when inserting several records that refer to one another, the foreign keys have to be correct and up to date, or the data becomes nonsensical.
- In a document data model, the fields that need to be updated together are often within the same document, which is treated as a single object—no multi-object transactions are needed when updating a single document. However, document databases lacking join functionality also encourage denormalization (see [“When to Use Which Model”](#)). When denormalized information needs to be updated, like in the example of [Figure 8-2](#), you need to

update several documents in one go. Transactions are very useful in this situation to prevent denormalized data from going out of sync.

- In databases with secondary indexes (almost everything except pure key-value stores), the indexes also need to be updated every time you change a value. These indexes are different database objects from a transaction point of view: for example, without transaction isolation, it's possible for a record to appear in one index but not another, because the update to the second index hasn't happened yet (see [“Sharding and Secondary Indexes”](#)).

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”](#), and explore alternative approaches in [Link to Come].

Handling errors and aborts

A key feature of a transaction is that it can be aborted and safely retried if an error occurred. ACID databases are based on this philosophy: if the database is in danger of violating its guarantee of atomicity, isolation, or durability, it would rather

abandon the transaction entirely than allow it to remain half-finished.

Not all systems follow that philosophy, though. In particular, datastores with leaderless replication (see [“Leaderless Replication”](#)) work much more on a “best effort” basis, which could be summarized as “the database will do as much as it can, and if it runs into an error, it won’t undo something it has already done”—so it’s the application’s responsibility to recover from errors.

Errors will inevitably happen, but many software developers prefer to think only about the happy path rather than the intricacies of error handling. For example, popular object-relational mapping (ORM) frameworks such as Rails’s ActiveRecord and Django don’t retry aborted transactions—the error usually results in an exception bubbling up the stack, so any user input is thrown away and the user gets an error message. This is a shame, because the whole point of aborts is to enable safe retries.

Although retrying an aborted transaction is a simple and effective error handling mechanism, it isn’t perfect:

- If the transaction actually succeeded, but the network was interrupted while the server tried to acknowledge the successful commit to the client (so it timed out from the client's point of view), 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 or high contention between concurrent transactions, retrying the transaction will make the problem worse, not better. To avoid such feedback cycles, you can limit the number of retries, use exponential backoff, and handle overload-related errors differently from other errors (see [“When an overloaded system won’t recover”](#)).
- It is only worth retrying after transient errors (for example due to deadlock, isolation violation, temporary network interruptions, and failover); after a permanent error (e.g., constraint violation) a retry would be pointless.
- If the transaction also has side effects outside of the database, those side effects may happen even if the transaction is aborted. For example, if you're sending an email, you wouldn't want to send the email again every time you retry the transaction. If you want to make sure that several different systems either commit or abort together, two-phase commit can help (we will discuss this in [Link to Come]).

- If the client process crashes while retrying, any data it was trying to write to the database is lost.

Weak Isolation Levels

If two transactions don't access the same data, or if both are read-only, 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 the two transactions try to modify the same data.

Concurrency bugs are hard to find by testing, because such bugs are only triggered when you get unlucky with the timing. Such timing issues might occur very rarely, and are usually difficult to reproduce. Concurrency is also very difficult to reason about, especially in a large application where you don't necessarily know which other pieces of code are accessing the database. Application development is difficult enough if you just have one user at a time; having many concurrent users makes it much harder still, because any piece of data could unexpectedly change at any time.

For that reason, databases have long tried to hide concurrency issues from application developers by providing *transaction isolation*. In theory, isolation should make your life easier by letting you pretend that no concurrency is happening: *serializable* isolation means that the database guarantees that transactions have the same effect as if they ran *serially* (i.e., one at a time, without any concurrency).

In practice, isolation is unfortunately not that simple. Serializable isolation has a performance cost, and many databases don't want to pay that price [10]. 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 [29].

Concurrency bugs caused by weak transaction isolation are not just a theoretical problem. They have caused substantial loss of money [30, 31, 32], led to investigation by financial auditors [33], and caused customer data to be corrupted [34]. A popular comment on revelations of such problems is “Use an ACID database if you’re handling financial data!”—but that misses the point. Even many popular relational database systems (which are usually considered “ACID”) use weak isolation, so

they wouldn't necessarily have prevented these bugs from occurring.

NOTE

Incidentally, much of the banking system relies on text files that are exchanged via secure FTP [35]. In this context, having an audit trail and some human-level fraud prevention measures is actually more important than ACID properties.

Those examples also highlight an important point: even if concurrency issues are rare in normal operation, you have to consider the possibility that an attacker deliberately sends a burst of highly concurrent requests to your API in an attempt to deliberately exploit concurrency bugs [30]. Therefore, in order to build applications that are reliable and secure, you have to ensure that such bugs are systematically prevented.

In this section we will look at several weak (nonserializable) isolation levels that are used in practice, and discuss in detail what kinds of race conditions can and cannot occur, so that you can decide what level is appropriate to your application. Once we've done that, we will discuss serializability in detail (see [“Serializability”](#)). Our discussion of isolation levels will be informal, using examples. If you want rigorous definitions and

analyses of their properties, you can find them in the academic literature [36, 37, 38, 39].

Read Committed

The most basic level of transaction isolation is *read committed*. 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*).

Some databases support an even weaker isolation level called *read uncommitted*. It prevents dirty writes, but does not prevent dirty reads. Let's discuss these two guarantees in more detail.

No dirty reads

Imagine a transaction has written some data to the database, but the transaction has not yet committed or aborted. Can another transaction see that uncommitted data? If yes, that is called a *dirty read* [3].

Transactions running at the read committed isolation level must prevent dirty reads. This means that any writes by a

transaction only become visible to others when that transaction commits (and then all of its writes become visible at once). This is illustrated in [Figure 8-4](#), where user 1 has set $x = 3$, but user 2's $get\ x$ still returns the old value, 2, while user 1 has not yet committed.

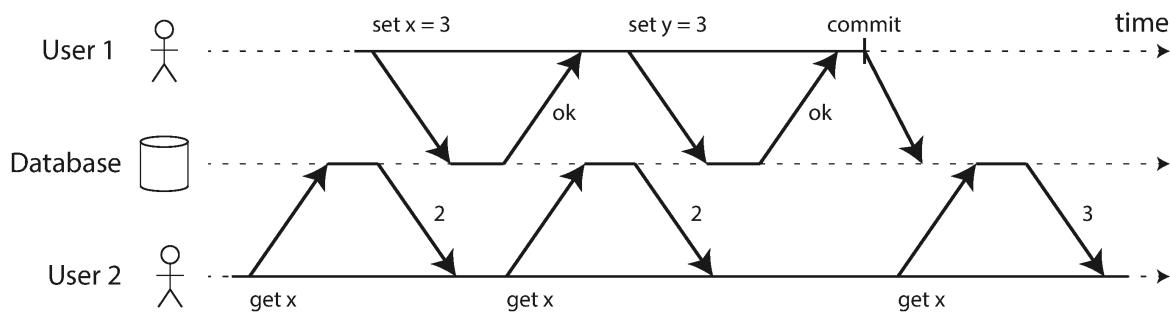


Figure 8-4. No dirty reads: user 2 sees the new value for x only after user 1's transaction has committed.

There are a few reasons why it's useful to prevent dirty reads:

- If a transaction needs to update several rows, a dirty read means that another transaction may see some of the updates but not others. For example, in [Figure 8-2](#), 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 8-3](#)). If the database allows dirty

reads, that means a transaction may see data that is later rolled back—i.e., which is never actually committed to the database. Any transaction that read uncommitted data would also need to be aborted, leading to a problem called *cascading aborts*.

No dirty writes

What happens if two transactions concurrently try to update the same row 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* [36]. Transactions running at the read committed isolation level must prevent dirty writes, usually by delaying the second write until the first write's transaction has committed or aborted.

By preventing dirty writes, this isolation level avoids some kinds of concurrency problems:

- If transactions update multiple rows, dirty writes can lead to a bad outcome. For example, consider [Figure 8-5](#), 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 8-5](#), 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 8-1](#). 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](#)” we will discuss how to make such counter increments safe.

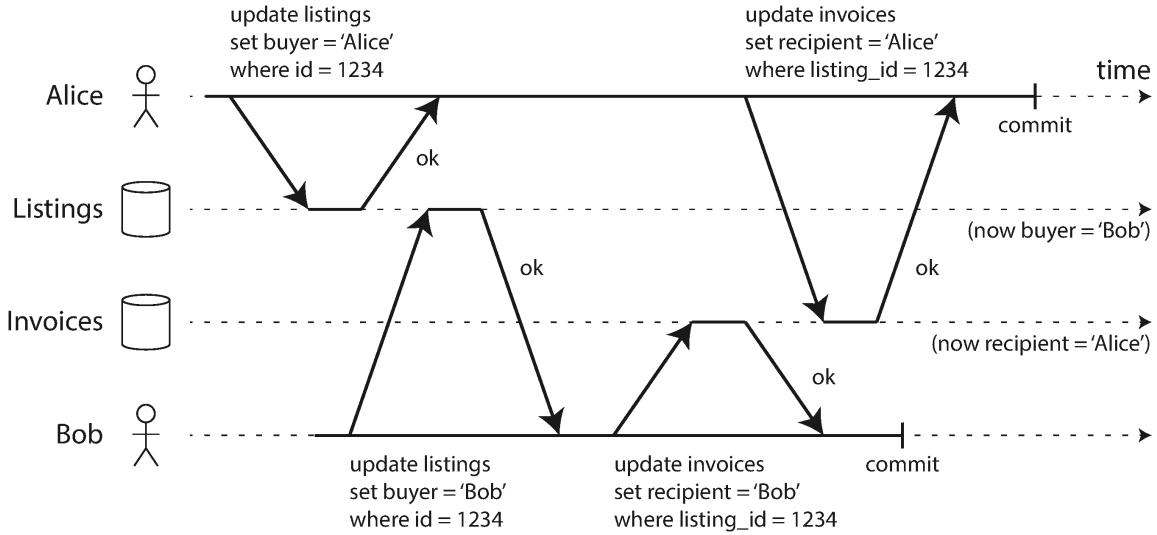


Figure 8-5. With dirty writes, conflicting writes from different transactions can be mixed up.

Implementing read committed

Read committed is a very popular isolation level. It is the default setting in Oracle Database, PostgreSQL, SQL Server, and many other databases [10].

Most commonly, databases prevent dirty writes by using row-level locks: when a transaction wants to modify a particular row (or document or some other object), it must first acquire a lock on that row. It must then hold that lock until the transaction is committed or aborted. Only one transaction can hold the lock for any given row; if another transaction wants to write to the same row, 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 a row to briefly acquire the lock and then release it again immediately after reading. This would ensure that a read couldn't happen while a row 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 other transactions to wait until the long-running transaction has completed, even if the other transactions only read and do not write anything to the database. 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.

Nevertheless, locks are used to prevent dirty reads in some databases, such as IBM Db2 and Microsoft SQL Server in the `read_committed_snapshot=off` setting [29].

A more commonly used approach to preventing dirty reads is the one illustrated in [Figure 8-4](#): for every row that is written, the database remembers both the old committed value and the new value set by the transaction that currently holds the write lock. While the transaction is ongoing, any other transactions that read the row are simply given the old value. Only when the new value is committed do transactions switch over to reading the new value (see [“Multi-version concurrency control \(MVCC\)”](#) for more detail).

Snapshot Isolation and Repeatable Read

If you look superficially at read committed isolation, you could be forgiven for thinking that it does everything that a transaction needs to do: it allows aborts (required for atomicity), it prevents reading the incomplete results of transactions, and it prevents concurrent writes from getting intermingled. Indeed, those are useful features, and much stronger guarantees than you can get from a system that has no transactions.

However, there are still plenty of ways in which you can have concurrency bugs when using this isolation level. For example, [Figure 8-6](#) illustrates a problem that can occur with read committed.

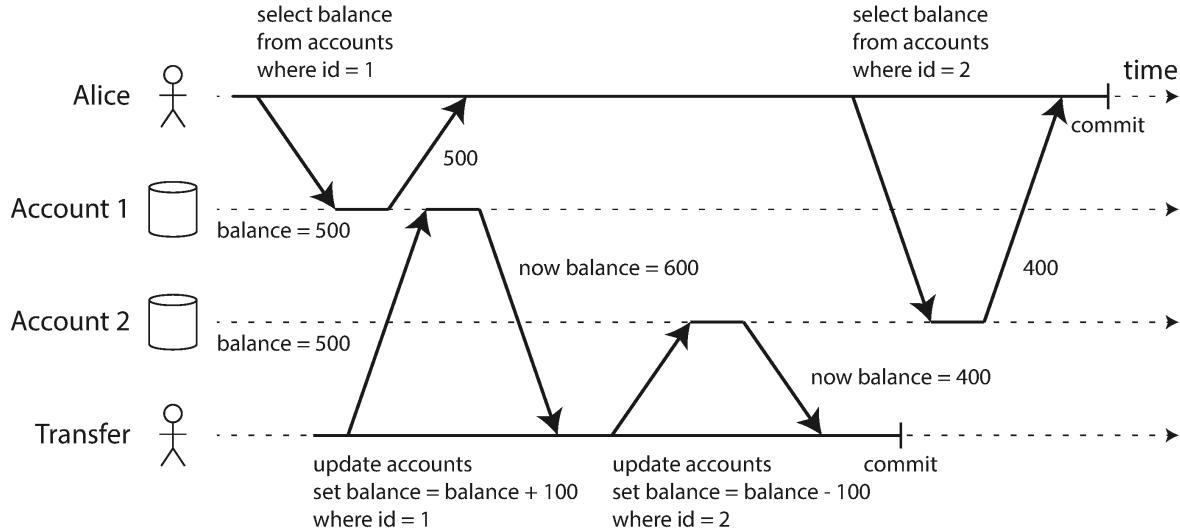


Figure 8-6. Read skew: Alice observes the database in an inconsistent state.

Say Alice has \$1,000 of savings at a bank, split across two accounts with \$500 each. Now a transaction transfers \$100 from one of her accounts to the other. If she is unlucky enough to look at her list of account balances in the same moment as that transaction is being processed, she may see one account balance at a time before the incoming payment has arrived (with a balance of \$500), and the other account after the outgoing transfer has been made (the new balance being \$400). To Alice it now appears as though she only has a total of \$900 in her accounts—it seems that \$100 has vanished into thin air.

This anomaly is called *read skew*, and it is an example of a *nonrepeatable read*: 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.

NOTE

The term *skew* is unfortunately overloaded: we previously used it in the sense of an *unbalanced workload with hot spots* (see [“Skewed Workloads and Relieving Hot Spots”](#)), whereas here it means *timing anomaly*.

In Alice’s case, this is not a lasting problem, because she will most likely see consistent account balances if she reloads the online banking website a few seconds later. However, some situations cannot tolerate such temporary inconsistency:

Backups

Taking a backup requires making a copy of the entire database, which may take hours on a large database.

During the time that the backup process is running, writes will continue to be made to the database. Thus, you could end up with some parts of the backup containing an older version of the data, and other parts containing a newer version. If you need to restore from such a backup, the inconsistencies (such as disappearing money) become permanent.

Analytic queries and integrity checks

Sometimes, you may want to run a query that scans over large parts of the database. Such queries are common in analytics (see “[Analytical versus Operational Systems](#)”), or may be part of a periodic integrity check that everything is in order (monitoring for data corruption). These queries are likely to return nonsensical results if they observe parts of the database at different points in time.

Snapshot isolation [36] is the most common solution to this problem. The idea is that each transaction reads from a *consistent snapshot* of the database—that is, the transaction sees all the data that was committed in the database at the start of the transaction. Even if the data is subsequently changed by another transaction, each transaction sees only the old data from that particular point in time.

Snapshot isolation is a boon for long-running, read-only queries such as backups and analytics. It is very hard to reason about the meaning of a query if the data on which it operates is changing at the same time as the query is executing. When a transaction can see a consistent snapshot of the database, frozen at a particular point in time, it is much easier to understand.

Snapshot isolation is a popular feature: variants of it are supported by PostgreSQL, MySQL with the InnoDB storage engine, Oracle, SQL Server, and others, although the detailed behavior varies from one system to the next [29, 40, 41]. Some databases, such as Oracle, TiDB, and Aurora DSQL, even choose snapshot isolation as their highest isolation level.

Multi-version concurrency control (MVCC)

Like read committed isolation, implementations of snapshot isolation typically use write locks to prevent dirty writes (see “[Implementing read committed](#)”), which means that a transaction that makes a write can block the progress of another transaction that writes to the same row. However, reads do not require any locks. From a performance point of view, a key principle of snapshot isolation is *readers never block writers, and writers never block readers*. This allows a database to handle long-running read queries on a consistent snapshot at the same time as processing writes normally, without any lock contention between the two.

To implement snapshot isolation, databases use a generalization of the mechanism we saw for preventing dirty reads in [Figure 8-4](#). Instead of two versions of each row (the committed version and the overwritten-but-not-yet-committed

version), the database must potentially keep several different committed versions of a row, because various in-progress transactions may need to see the state of the database at different points in time. Because it maintains several versions of a row side by side, this technique is known as *multi-version concurrency control* (MVCC).

[Figure 8-7](#) illustrates how MVCC-based snapshot isolation is implemented in PostgreSQL [40, 42, 43] (other implementations are similar). When a transaction is started, it is given a unique, always-increasing transaction ID (`txid`). Whenever a transaction writes anything to the database, the data it writes is tagged with the transaction ID of the writer. (To be precise, transaction IDs in PostgreSQL are 32-bit integers, so they overflow after approximately 4 billion transactions. The vacuum process performs cleanup to ensure that overflow does not affect the data.)

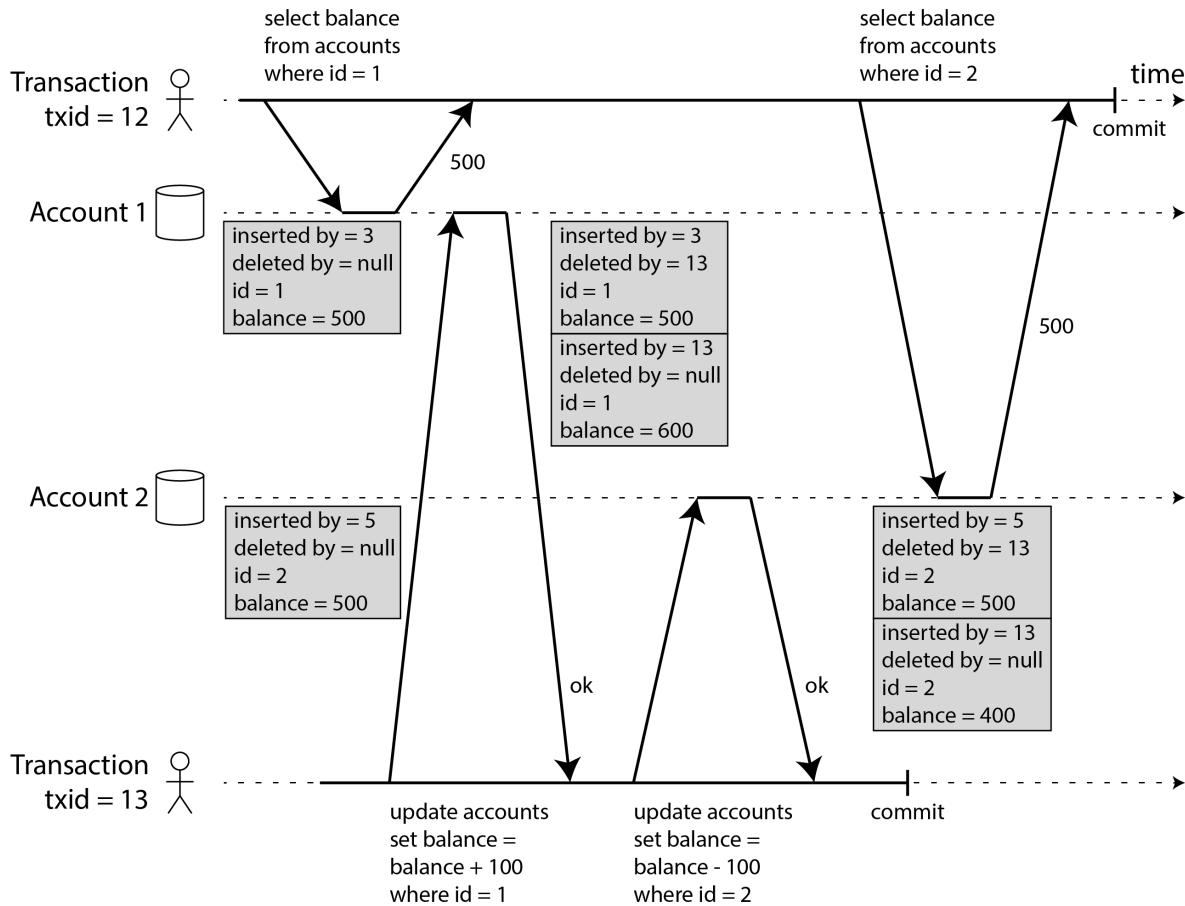


Figure 8-7. Implementing snapshot isolation using multi-version concurrency control.

Each row in a table has a `inserted_by` field, containing the ID of the transaction that inserted this row into the table.

Moreover, each row has a `deleted_by` field, which is initially empty. If a transaction deletes a row, the row isn't actually removed 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 insert. For example, in [Figure 8-7](#), 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 inserted by transaction 13.

All of the versions of a row are stored within the same database heap (see [“Storing values within the index”](#)), regardless of whether the transactions that wrote them have committed or not. The versions of the same row form a linked list, going either from newest version to oldest version or the other way round, so that queries can internally iterate over all versions of a row [44, 45].

Visibility rules for observing a consistent snapshot

When a transaction reads from the database, transaction IDs are used to decide which row versions it can see and which are invisible. By carefully defining visibility rules, the database can

present a consistent snapshot of the database to the application. This works roughly as follows [43]:

1. At the start of each transaction, the database makes a list of all the other transactions that are in progress (not yet committed or aborted) at that time. Any writes that those transactions have made are ignored, even if the transactions subsequently commit. This ensures that we see a consistent snapshot that is not affected by another transaction committing.
2. Any writes made by transactions with a later transaction ID (i.e., which started after the current transaction started, and which are therefore not included in the list of in-progress transactions) are ignored, regardless of whether those transactions have committed.
3. Any writes made by aborted transactions are ignored, regardless of when that abort happened. This has the advantage that when a transaction aborts, we don't need to immediately remove the rows it wrote from storage, since the visibility rule filters them out. The garbage collection process can remove them later.
4. All other writes are visible to the application's queries.

These rules apply to both insertion and deletion of rows. In [Figure 8-7](#), 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 2, transaction 12 cannot see a deletion made by transaction 13), and the insertion of the \$400 balance is not yet visible (by the same rule).

Put another way, a row is visible if both of the following conditions are true:

- At the time when the reader's transaction started, the transaction that inserted the row had already committed.
- The row is not marked for deletion, or if it is, the transaction that requested deletion had not yet committed at the time when the reader's transaction started.

A long-running transaction may continue using a snapshot for a long time, continuing to read values that (from other transactions' point of view) have long been overwritten or deleted. By never updating values in place but instead inserting a new version every time a value is changed, the database can provide a consistent snapshot while incurring only a small overhead.

Indexes and snapshot isolation

How do indexes work in a multi-version database? The most common approach is that each index entry points at one of the

versions of a row that matches the entry (either the oldest or the newest version). Each row version may contain a reference to the next-oldest or next-newest version. A query that uses the index must then iterate over the rows to find one that is visible, and where the value matches what the query is looking for.

When garbage collection removes old row versions that are no longer visible to any transaction, the corresponding index entries can also be removed.

Many implementation details affect the performance of multi-version concurrency control [44, 45]. For example, PostgreSQL has optimizations for avoiding index updates if different versions of the same row can fit on the same page [40]. Some other databases avoid storing full copies of modified rows, and only store differences between versions to save space.

Another approach is used in CouchDB, Datomic, and LMDB. Although they also use B-trees (see “[B-Trees](#)”), they use an *immutable* (copy-on-write) variant that does not overwrite pages of the tree when they are updated, but instead creates a new copy of each modified page. Parent pages, up to the root of the tree, are copied and updated to point to the new versions of their child pages. Any pages that are not affected by a write do not need to be copied, and can be shared with the new tree [46].

With immutable 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 rows based on transaction IDs because subsequent writes cannot modify an existing B-tree; they can only create new tree roots. This approach also requires a background process for compaction and garbage collection.

Snapshot isolation, repeatable read, and naming confusion

MVCC is a commonly used implementation technique for databases, and often it is used to implement snapshot isolation. However, different databases sometimes use different terms to refer to the same thing: for example, snapshot isolation is called “repeatable read” in PostgreSQL, and “serializable” in Oracle [29]. Sometimes different systems use the same term to mean different things: for example, while in PostgreSQL “repeatable read” means snapshot isolation, in MySQL it means an implementation of MVCC with weaker consistency than snapshot isolation [41].

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 [3] and snapshot isolation hadn’t yet been invented then. Instead, it defines repeatable read, which looks superficially similar to snapshot isolation. PostgreSQL calls its 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 [36]. Even though several databases implement repeatable read, there are big differences in the guarantees they actually provide, despite being ostensibly standardized [29]. There has been a formal definition of repeatable read in the research literature [37, 38], but most implementations don’t satisfy that formal definition. And to top it off, IBM Db2 uses “repeatable read” to refer to serializability [10].

As a result, nobody really knows what repeatable read means.

Preventing Lost Updates

The read committed and snapshot isolation levels we’ve discussed so far have been primarily about the guarantees of

what a read-only transaction can see in the presence of concurrent writes. We have mostly ignored the issue of two transactions writing concurrently—we have only discussed dirty writes (see “[No dirty writes](#)”), one particular type of write-write conflict that can occur.

There are several other interesting kinds of conflicts that can occur between concurrently writing transactions. The best known of these is the *lost update* problem, illustrated in [Figure 8-1](#) with the example of two concurrent counter increments.

The lost update problem can occur if an application reads some value from the database, modifies it, and writes back the modified value (a *read-modify-write cycle*). If two transactions do this concurrently, one of the modifications can be lost, because the second write does not include the first modification. (We sometimes say that the later write *clobbers* the earlier write.) This pattern occurs in various different scenarios:

- Incrementing a counter or updating an account balance (requires reading the current value, calculating the new value, and writing back the updated value)

- Making a local change to a complex value, e.g., adding an element to a list within a JSON document (requires parsing the document, making the change, and writing back the modified document)
- Two users editing a wiki page at the same time, where each user saves their changes by sending the entire page contents to the server, overwriting whatever is currently in the database

Because this is such a common problem, a variety of solutions have been developed.

Atomic write operations

Many databases provide atomic update operations, which remove the need to implement read-modify-write cycles in application code. They are usually the best solution if your code can be expressed in terms of those operations. For example, the following instruction is concurrency-safe in most relational databases:

```
UPDATE counters SET value = value + 1 WHERE key =
```

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, which can be handled using algorithms discussed in “[CRDTs and Operational Transformation](#)”—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 applied. Another option is to simply force all atomic operations to be executed on a single thread.

Unfortunately, object-relational mapping (ORM) frameworks make it easy to accidentally write code that performs unsafe read-modify-write cycles instead of using atomic operations provided by the database [47, 48, 49]. This can be 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 update or lock the same object, it is forced to wait until the first read-modify-write cycle has completed.

For example, consider a multiplayer game in which several players can move the same figure concurrently. In this case, an atomic operation may not be sufficient, because the application also needs to ensure that a player's move abides by the rules of the game, which involves some logic that you cannot sensibly implement as a database query. Instead, you may use a lock to prevent two players from concurrently moving the same piece, as illustrated in [Example 8-1](#).

Example 8-1. 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 p
-- of the piece that was returned by the previous
UPDATE figures SET position = 'c4' WHERE id = 123
```

```
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.

Moreover, if you lock multiple objects there is a risk of deadlock, where two or more transactions are waiting for each other to release their locks. Many databases automatically detect deadlocks, and abort one of the involved transactions so that the system can make progress. You can handle this situation at the application level by retrying the aborted transaction.

Automatically detecting lost updates

Atomic operations and locks are ways of preventing lost updates by forcing the read-modify-write cycles to happen sequentially. An alternative is to allow them to execute in parallel and, if the transaction manager detects a lost update,

abort the transaction and force it to retry its read-modify-write cycle.

An advantage of this approach is that databases can perform this check efficiently in conjunction with snapshot isolation. Indeed, PostgreSQL’s repeatable read, Oracle’s serializable, and SQL Server’s snapshot isolation levels automatically detect when a lost update has occurred and abort the offending transaction. However, MySQL/InnoDB’s repeatable read does not detect lost updates [29, 41]. Some authors [36, 38] 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. However, you also have to retry aborted transactions at the application level.

Conditional writes (compare-and-swap)

In databases that don’t provide transactions, you sometimes find a *conditional write* operation that can prevent lost updates

by allowing an update to happen only if the value has not changed since you last read it (previously mentioned in “[Single-object writes](#)”). 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. It is the database equivalent of an atomic *compare-and-swap (CAS)* instruction that is supported by many CPUs.

For example, to prevent two users concurrently updating the same wiki page, you might try something like this, expecting the update to occur only if the content of the page hasn’t changed since the user started editing it:

```
-- This may or may not be safe, depending on the
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. Instead of comparing the full content, you could also use a version number column that you increment on every update, and apply the update only if the current version number hasn’t changed. This approach is sometimes called *optimistic locking* [50].

Note that if another transaction has concurrently modified content, the new content may not be visible under the MVCC visibility rules (see “[Visibility rules for observing a consistent snapshot](#)”). Many implementations of MVCC have an exception to the visibility rules for this scenario, where values written by other transactions are visible to the evaluation of the WHERE clause of UPDATE and DELETE queries, even though those writes are not otherwise visible in the snapshot.

Conflict resolution and replication

In replicated databases (see [Chapter 6](#)), preventing lost updates takes on another dimension: since they have copies of the data on multiple nodes, and the data can potentially be modified concurrently on different nodes, some additional steps need to be taken to prevent lost updates.

Locks and conditional write 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 conditional writes do not apply in this context. (We will revisit this issue in more detail in [Link to Come].)

Instead, as discussed in “[Dealing with Conflicting Writes](#)”, 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.

Merging conflicting values can prevent lost updates if the updates 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 CRDTs, which we encountered in “[CRDTs and Operational Transformation](#)”. However, some operations such as conditional writes cannot be made commutative.

On the other hand, the *last write wins* (LWW) conflict resolution method is prone to lost updates, as discussed in “[Last write wins \(discarding concurrent writes\)](#)”. Unfortunately, LWW is the default in many replicated databases.

Write Skew and Phantoms

In the previous sections we saw *dirty writes* and *lost updates*, two kinds of race conditions that can occur when different

transactions concurrently try to write to the same objects. In order to avoid data corruption, those race conditions need to be prevented—either automatically by the database, or by manual safeguards such as using locks or atomic write operations.

However, that is not the end of the list of potential race conditions that can occur between concurrent writes. In this section we will see some subtler examples of conflicts.

To begin, imagine this example: you are writing an application for doctors to manage their on-call shifts at a hospital. The hospital usually tries to have several doctors on call at any one time, but it absolutely must have at least one doctor on call. Doctors can give up their shifts (e.g., if they are sick themselves), provided that at least one colleague remains on call in that shift [[51](#), [52](#)].

Now imagine that Alice and Bob are the two on-call doctors for a particular shift. Both are feeling unwell, so they both decide to request leave. Unfortunately, they happen to click the button to go off call at approximately the same time. What happens next is illustrated in [Figure 8-8](#).

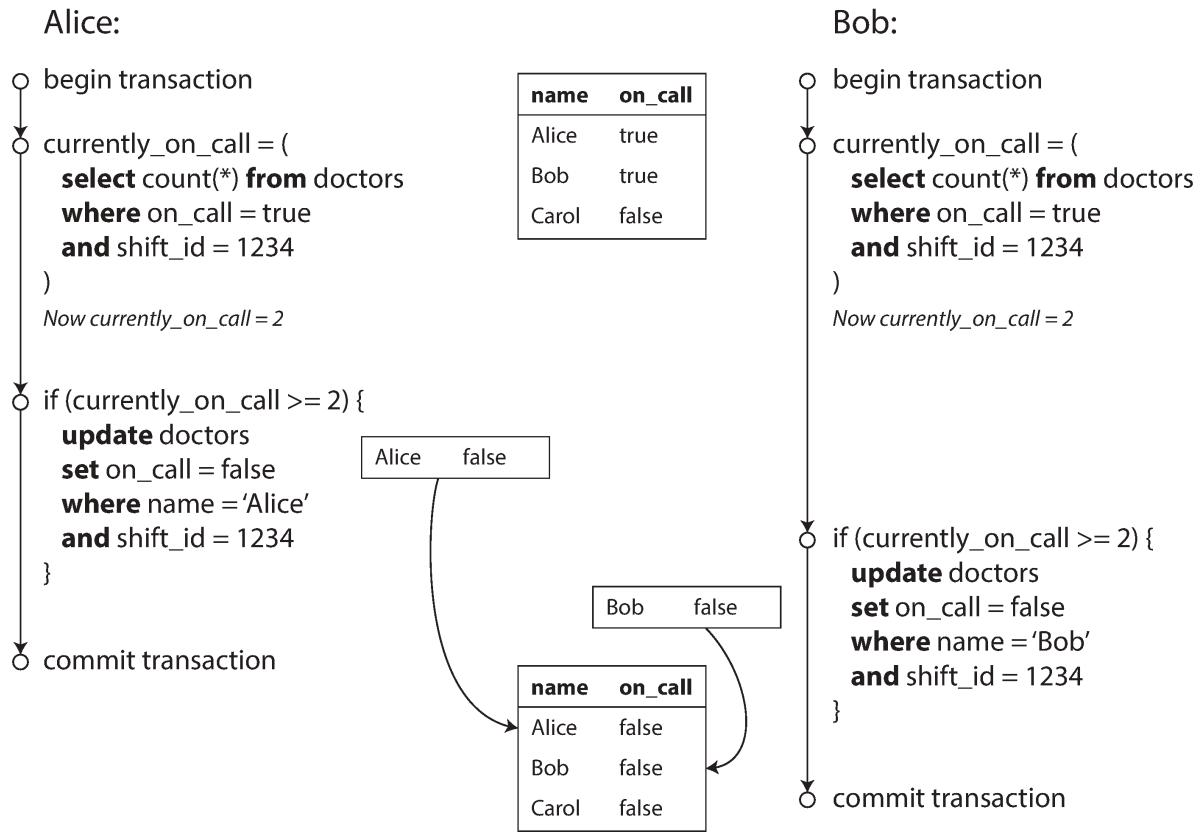


Figure 8-8. Example of write skew causing an application bug.

In each transaction, your application first checks that two or more doctors are currently on call; if yes, it assumes it's safe for one doctor to go off call. Since the database is using snapshot isolation, both checks return 2, so both transactions proceed to the next stage. Alice updates her own record to take herself off call, and Bob updates his own record likewise. Both transactions commit, and now no doctor is on call. Your requirement of having at least one doctor on call has been violated.

Characterizing write skew

This anomaly is called *write skew* [36]. It is neither a dirty write nor a lost update, because the two transactions are updating two different objects (Alice's and Bob's on-call records, respectively). It is less obvious that a conflict occurred here, but it's definitely a race condition: if the two transactions had run one after another, the second doctor would have been prevented from going off call. The anomalous behavior was only possible because the transactions ran concurrently.

You can think of write skew as a generalization of the lost update problem. Write skew can occur if two transactions read the same objects, and then update some of those objects (different transactions may update different objects). In the special case where different transactions update the same object, you get a dirty write or lost update anomaly (depending on the timing).

We saw that there are various different ways of preventing lost updates. With write skew, our options are more restricted:

- Atomic single-object operations don't help, as multiple objects are involved.

- The automatic detection of lost updates that you find in some implementations of snapshot isolation unfortunately doesn't help either: write skew is not automatically detected in PostgreSQL's repeatable read, MySQL/InnoDB's repeatable read, Oracle's serializable, or SQL Server's snapshot isolation level [29]. Automatically preventing write skew requires true serializable isolation (see "["Serializability"](#)").
- Some databases allow you to configure constraints, which are then enforced by the database (e.g., uniqueness, foreign key constraints, or restrictions on a particular value). However, in order to specify that at least one doctor must be on call, you would need a constraint that involves multiple objects. Most databases do not have built-in support for such constraints, but you may be able to implement them with triggers or materialized views, as discussed in "["Consistency"](#)" [12].
- If you can't use a serializable isolation level, the second-best option in this case is probably to explicitly lock the rows that the transaction depends on. In the doctors example, you could write something like the following:

```
BEGIN TRANSACTION;

SELECT * FROM doctors
WHERE on_call = true
```

```
AND shift_id = 1234 FOR UPDATE; ①

UPDATE doctors
    SET on_call = false
    WHERE name = 'Alice'
    AND shift_id = 1234;

COMMIT;
```

- ① 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 [53]. When someone wants to make a booking, you first check for any conflicting bookings (i.e., bookings for the same room with an overlapping time range), and if none are found, you create the meeting (see [Example 8-2](#)).

Example 8-2. A meeting room booking system tries to avoid double-booking (not safe under snapshot isolation)

```
BEGIN TRANSACTION;

-- Check for any existing bookings that over
SELECT COUNT(*) FROM bookings
WHERE room_id = 123 AND
    end_time > '2025-01-01 12:00' AND start_-

-- If the previous query returned zero:
INSERT INTO bookings
(room_id, start_time, end_time, user_id)
VALUES (123, '2025-01-01 12:00', '2025-01-01 13:00');

COMMIT;
```

Unfortunately, snapshot isolation does not prevent another user from concurrently inserting a conflicting meeting. In order to guarantee you won't get scheduling conflicts, you once again need serializable isolation.

Multiplayer game

In [Example 8-1](#), we used a lock to prevent lost updates (that is, making sure that two players can't move the same

figure at the same time). However, the lock doesn't prevent players from moving two different figures to the same position on the board or potentially making some other move that violates the rules of the game. Depending on the kind of rule you are enforcing, you might be able to use a unique constraint, but otherwise you're vulnerable to write skew.

Claiming a username

On a website where each user has a unique username, two users may try to create accounts with the same username at the same time. You may use a transaction to check whether a name is taken and, if not, create an account with that name. However, like in the previous examples, that is not safe under snapshot isolation. Fortunately, a unique constraint is a simple solution here (the second transaction that tries to register the username will be aborted due to violating the constraint).

Preventing double-spending

A service that allows users to spend money or points needs to check that a user doesn't spend more than they have. You might implement this by inserting a tentative spending item into a user's account, listing all the items in the account, and checking that the sum is positive. With

write skew, it could happen that two spending items are inserted concurrently that together cause the balance to go negative, but that neither transaction notices the other.

Phantoms causing write skew

All of these examples follow a similar pattern:

1. A `SELECT` query checks whether some requirement is satisfied by searching for rows that match some search condition (there are at least two doctors on call, there are no existing bookings for that room at that time, the position on the board doesn't already have another figure on it, the username isn't already taken, there is still money in the account).
2. Depending on the result of the first query, the application code decides how to continue (perhaps to go ahead with the operation, or perhaps to report an error to the user and abort).
3. If the application decides to go ahead, it makes a write (`INSERT`, `UPDATE`, or `DELETE`) to the database and commits the transaction.

The effect of this write changes the precondition of the decision of step 2. In other words, if you were to repeat the `SELECT` query from step 1 after committing the write, you

would get a different result, because the write changed the set of rows matching the search condition (there is now one fewer doctor on call, the meeting room is now booked for that time, the position on the board is now taken by the figure that was moved, the username is now taken, there is now less money in the account).

The steps may occur in a different order. For example, you could first make the write, then the `SELECT` query, and finally decide whether to abort or commit based on the result of the query.

In the case of the doctor on call example, the row being modified in step 3 was one of the rows returned in step 1, so we could make the transaction safe and avoid write skew by locking the rows in step 1 (`SELECT FOR UPDATE`). However, the other four examples are different: they check for the *absence* of rows matching some search condition, and the write *adds* a row matching the same condition. If the query in step 1 doesn't return any rows, `SELECT FOR UPDATE` can't attach locks to anything [54].

This effect, where a write in one transaction changes the result of a search query in another transaction, is called a *phantom* [4]. 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. The SQL generated by ORMs is also prone to write skew [[48](#), [49](#)].

Materializing conflicts

If the problem of phantoms is that there is no object to which we can attach the locks, perhaps we can artificially introduce a lock object into the database?

For example, in the meeting room booking case you could imagine creating a table of time slots and rooms. Each row in this table corresponds to a particular room for a particular time period (say, 15 minutes). You create rows for all possible combinations of rooms and time periods ahead of time, e.g. for the next six months.

Now a transaction that wants to create a booking can lock (`SELECT FOR UPDATE`) the rows in the table that correspond to the desired room and time period. After it has acquired the locks, it can check for overlapping bookings and insert a new booking as before. Note that the additional table isn't used to store information about the booking—it's purely a collection of locks which is used to prevent bookings on the same room and time range from being modified concurrently.

This approach is called *materializing conflicts*, because it takes a phantom and turns it into a lock conflict on a concrete set of rows that exist in the database [14]. Unfortunately, it can be hard and error-prone to figure out how to materialize conflicts, and it's ugly to let a concurrency control mechanism leak into the application data model. For those reasons, materializing conflicts should be considered a last resort if no alternative is possible. A serializable isolation level is much preferable in most cases.

Serializability

In this chapter we have seen several examples of transactions that are prone to race conditions. Some race conditions are prevented by the read committed and snapshot isolation levels, but others are not. We encountered some particularly tricky examples with write skew and phantoms. It's a sad situation:

- Isolation levels are hard to understand, and inconsistently implemented in different databases (e.g., the meaning of “repeatable read” varies significantly).
- If you look at your application code, it’s difficult to tell whether it is safe to run at a particular isolation level—

especially in a large application, where you might not be aware of all the things that may be happening concurrently.

- There are no good tools to help us detect race conditions. In principle, static analysis may help [33], but research techniques have not yet found their way into practical use. Testing for concurrency issues is hard, because they are usually nondeterministic—problems only occur if you get unlucky with the timing.

This is not a new problem—it has been like this since the 1970s, when weak isolation levels were first introduced [3]. All along, the answer from researchers has been simple: use *Serializable* isolation!

Serializable isolation is the strongest isolation level. It guarantees that even though transactions may execute in parallel, the end result is the same as if they had executed one at a time, *serially*, without any concurrency. Thus, the database guarantees that if the transactions behave correctly when run individually, they continue to be correct when run concurrently—in other words, the database prevents *all* possible race conditions.

But if serializable isolation is so much better than the mess of weak isolation levels, then why isn't everyone using it? To

answer this question, we need to look at the options for implementing serializability, and how they perform. Most databases that provide serializability today use one of three techniques, which we will explore in the rest of this chapter:

- Literally executing transactions in a serial order (see “[Actual Serial Execution](#)”)
- Two-phase locking (see “[Two-Phase Locking \(2PL\)](#)”), which for several decades was the only viable option
- Optimistic concurrency control techniques such as serializable snapshot isolation (see “[Serializable Snapshot Isolation \(SSI\)](#)”)

For now, we will discuss these techniques primarily in the context of single-node databases; in [Link to Come] we will examine how they can be generalized to transactions that involve multiple nodes in a distributed system.

Actual Serial Execution

The simplest way of avoiding concurrency problems is to remove the concurrency entirely: to execute only one transaction at a time, in serial order, on a single thread. By doing so, we completely sidestep the problem of detecting and

preventing conflicts between transactions: the resulting isolation is by definition serializable.

Even though this seems like an obvious idea, it was only in the 2000s that database designers decided that a single-threaded loop for executing transactions was feasible [55]. If multi-threaded concurrency was considered essential for getting good performance during the previous 30 years, what changed to make single-threaded execution possible?

Two developments caused this rethink:

- RAM became cheap enough that for many use cases it is now feasible to keep the entire active dataset in memory (see “[Keeping everything in memory](#)”). 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 “[Analytical versus Operational Systems](#)”). By contrast, long-running analytic queries are typically read-only, so they can be run on a consistent snapshot (using snapshot isolation) outside of the serial execution loop.

The approach of executing transactions serially is implemented in VoltDB/H-Store, Redis, and Datomic, for example [56, 57, 58]. A system designed for single-threaded execution can sometimes perform better than a system that supports concurrency, because it can avoid the coordination overhead of locking. However, its throughput is limited to that of a single CPU core. In order to make the most of that single thread, transactions need to be structured differently from their traditional form.

Encapsulating transactions in stored procedures

In the early days of databases, the intention was that a database transaction could encompass an entire flow of user activity. For example, booking an airline ticket is a multi-stage process (searching for routes, fares, and available seats; deciding on an itinerary; booking seats on each of the flights of the itinerary; entering passenger details; making payment). Database designers thought that it would be neat if that entire process was one transaction so that it could be committed atomically.

Unfortunately, humans are very slow to make up their minds and respond. If a database transaction needs to wait for input from a user, the database needs to support a potentially huge number of concurrent transactions, most of them idle. Most databases cannot do that efficiently, and so almost all OLTP

applications keep transactions short by avoiding interactively waiting for a user within a transaction. On the web, this means that a transaction is committed within the same HTTP request—a transaction does not span multiple requests. A new HTTP request starts a new transaction.

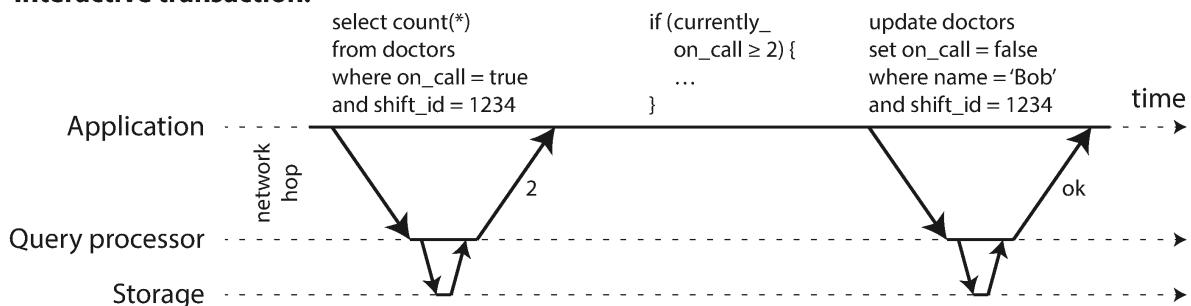
Even though the human has been taken out of the critical path, transactions have continued to be executed in an interactive client/server style, one statement at a time. An application makes a query, reads the result, perhaps makes another query depending on the result of the first query, and so on. The queries and results are sent back and forth between the application code (running on one machine) and the database server (on another machine).

In this interactive style of transaction, a lot of time is spent in network communication between the application and the database. If you were to disallow concurrency in the database and only process one transaction at a time, the throughput would be dreadful because the database would spend most of its time waiting for the application to issue the next query for the current transaction. In this kind of database, it's necessary to process multiple transactions concurrently in order to get reasonable performance.

For this reason, systems with single-threaded serial transaction processing don't allow interactive multi-statement transactions. Instead, the application must either limit itself to transactions containing a single statement, or submit the entire transaction code to the database ahead of time, as a *stored procedure* [59].

The differences between interactive transactions and stored procedures is illustrated in [Figure 8-9](#). Provided that all data required by a transaction is in memory, the stored procedure can execute very quickly, without waiting for any network or disk I/O.

Interactive transaction:



Stored procedure:

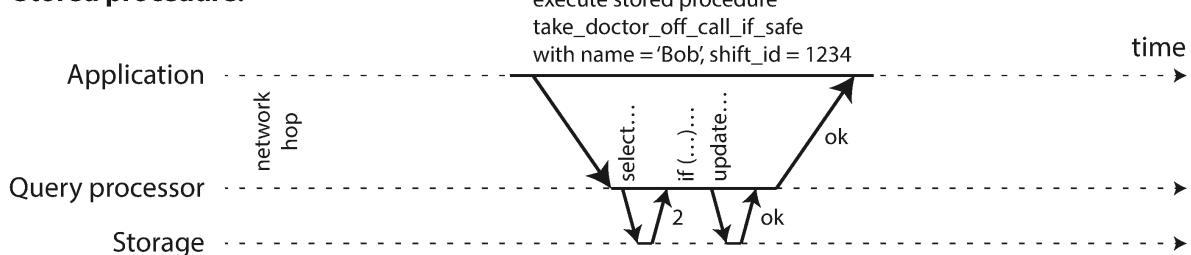


Figure 8-9. The difference between an interactive transaction and a stored procedure (using the example transaction of [Figure 8-8](#)).

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:

- Traditionally, each database vendor had its own language for stored procedures (Oracle has PL/SQL, SQL Server has T-SQL, PostgreSQL has PL/pgSQL, etc.). These languages haven't kept up with developments in general-purpose programming languages, so they look quite ugly and archaic from today's point of view, and they lack the ecosystem of libraries that you find with most programming languages.
- Code running in a database is difficult to manage: compared to an application server, it's harder to debug, more awkward to keep in version control and deploy, trickier to test, and difficult to integrate with a metrics collection system for monitoring.
- A database is often much more performance-sensitive than an application server, because a single database instance is often shared by many application servers. A badly written stored procedure (e.g., using a lot of memory or CPU time) in a database can cause much more trouble than equivalent badly written code in an application server.

- In a multitenant system that allows tenants to write their own stored procedures, it's a security risk to execute untrusted code in the same process as the database kernel [60].

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, Redis uses Lua, and MongoDB uses Javascript.

Stored procedures are also useful in cases where application logic can't easily be embedded elsewhere. Applications that use GraphQL, for example, might directly expose their database through a GraphQL proxy. If the proxy doesn't support complex validation logic, you can embed such logic directly in the database using a stored procedure. If the database doesn't support stored procedures, you would have to deploy a validation service between the proxy and the database to do validation.

With stored procedures and in-memory data, executing all transactions on a single thread becomes feasible. When stored procedures 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 (see [Durable Execution and Workflows](#) for more details on deterministic operations). This approach is called *state machine replication*, and we will return to it in [Link to Come].

Sharding

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 shard your data (see [Chapter 7](#)), which is supported in VoltDB. If you can find a way of sharding your dataset so that each transaction only needs to read and write data within a single shard, then each shard can have its own transaction processing thread running independently from the others. In this case, you can give each CPU core its own shard, which allows your transaction throughput to scale linearly with the number of CPU cores [[57](#)].

However, for any transaction that needs to access multiple shards, the database must coordinate the transaction across all the shards that it touches. The stored procedure needs to be performed in lock-step across all shards to ensure serializability across the whole system.

Since cross-shard transactions have additional coordination overhead, they are vastly slower than single-shard transactions. VoltDB reports a throughput of about 1,000 cross-shard writes per second, which is orders of magnitude below its single-shard throughput and cannot be increased by adding more machines [[59](#)]. More recent research has explored ways of making multi-shard transactions more scalable [[61](#)].

Whether transactions can be single-shard depends very much on the structure of the data used by the application. Simple key-value data can often be sharded very easily, but data with multiple secondary indexes is likely to require a lot of cross-shard coordination (see [“Sharding and Secondary Indexes”](#)).

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 most appropriate in situations 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.
- Write throughput must be low enough to be handled on a single CPU core, or else transactions need to be sharded without requiring cross-shard coordination.
- Cross-shard transactions are possible, but their throughput is hard to scale.

Two-Phase Locking (2PL)

For around 30 years, there was only one widely used algorithm for serializability in databases: *two-phase locking* (2PL), sometimes called *strong strict two-phase locking* (SS2PL) to distinguish it from other variants of 2PL.

2PL IS NOT 2PC

Note that while two-phase *locking* (2PL) sounds very similar to two-phase *commit* (2PC), they are completely different things. We will discuss 2PC in [Link to Come].

We saw previously that locks are often used to prevent dirty writes (see “[No dirty writes](#)”): 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 8-4](#), is not acceptable under 2PL.)

In 2PL, writers don't just block other writers; they also block readers and vice versa. Snapshot isolation has the mantra *readers never block writers, and writers never block readers* (see [“Multi-version concurrency control \(MVCC\)”](#)), which captures this key difference between snapshot isolation and two-phase locking. On the other hand, because 2PL provides serializability, it protects against all the race conditions discussed earlier, including lost updates and write skew.

Implementation of two-phase locking

2PL is used by the serializable isolation level in MySQL (InnoDB) and SQL Server, and the repeatable read isolation level in Db2 [\[29\]](#).

The blocking of readers and writers is implemented by having a lock on each object in the database. The lock can either be in *shared mode* or in *exclusive mode* (also known as a *multi-reader single-writer* lock). The lock is used as follows:

- If a transaction wants to read an object, it must first acquire the lock in shared mode. Several transactions are allowed to hold the lock in shared mode simultaneously, but if another transaction already has an exclusive lock on the object, these transactions must wait.
- If a transaction wants to write to an object, it must first acquire the lock in exclusive mode. No other transaction may hold the lock at the same time (either in shared or in exclusive mode), so if there is any existing lock on the object, the transaction must wait.
- If a transaction first reads and then writes an object, it may upgrade its shared lock to an exclusive lock. The upgrade works the same as getting an exclusive lock directly.
- After a transaction has acquired the lock, it must continue to hold the lock until the end of the transaction (commit or abort). This is where the name “two-phase” comes from: the first phase (while the transaction is executing) is when the locks are acquired, and the second phase (at the end of the transaction) is when all the locks are released.

Since so many locks are in use, it can happen quite easily that transaction A is stuck waiting for transaction B to release its lock, and vice versa. This situation is called *deadlock*. The database automatically detects deadlocks between transactions and aborts one of them so that the others can make progress. The aborted transaction needs to be retried by the application.

Performance of two-phase locking

The big downside of two-phase locking, and the reason why it hasn't been used by everybody since the 1970s, is performance: transaction throughput and response times of queries are significantly worse under two-phase locking than under weak isolation.

This is partly due to the overhead of acquiring and releasing all those locks, but more importantly due to reduced concurrency. By design, if two concurrent transactions try to do anything that may in any way result in a race condition, one has to wait for the other to complete.

For example, if you have a transaction that needs to read an entire table (e.g. a backup, analytics query, or integrity check, as discussed in [“Snapshot Isolation and Repeatable Read”](#)), that transaction has to take a shared lock on the entire table.

Therefore, the reading transaction first has to wait until all in-progress transactions writing to that table have completed; then, while the whole table is being read (which may take a long time on a large table), all other transactions that want to write to that table are blocked until the big read-only transaction commits. In effect, the database becomes unavailable for writes for an extended time.

For this reason, databases running 2PL can have quite unstable latencies, and they can be very slow at high percentiles (see [“Describing Performance”](#)) 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.

Although deadlocks can happen with the lock-based read committed isolation level, they occur much more frequently under 2PL serializable isolation (depending on the access patterns of your transaction). This can be an additional performance problem: when a transaction is aborted due to deadlock and is retried, it needs to do its work all over again. If deadlocks are frequent, this can mean significant wasted effort.

Predicate locks

In the preceding description of locks, we glossed over a subtle but important detail. In “[Phantoms causing write skew](#)” we discussed the problem of *phantoms*—that is, one transaction changing the results of another transaction’s search query. A database with serializable isolation must prevent phantoms.

In the meeting room booking example this means that if one transaction has searched for existing bookings for a room within a certain time window (see [Example 8-2](#)), another transaction is not allowed to concurrently insert or update another booking for the same room and time range. (It’s okay to concurrently insert bookings for other rooms, or for the same room at a different time that doesn’t affect the proposed booking.)

How do we implement this? Conceptually, we need a *predicate lock* [4]. It works similarly to the shared/exclusive lock described earlier, but rather than belonging to a particular object (e.g., one row in a table), it belongs to all objects that match some search condition, such as:

```
SELECT * FROM bookings  
WHERE room_id = 123 AND
```

```
end_time    > '2025-01-01 12:00' AND  
start_time < '2025-01-01 13:00';
```

A predicate lock restricts access as follows:

- If transaction A wants to read objects matching some condition, like in that `SELECT` query, it must acquire a shared-mode predicate lock on the conditions of the query. If another transaction B currently has an exclusive lock on any object matching those conditions, A must wait until B releases its lock before it is allowed to make its query.
- If transaction A wants to insert, update, or delete any object, it must first check whether either the old or the new value matches any existing predicate lock. If there is a matching predicate lock held by transaction B, then A must wait until B has committed or aborted before it can continue.

The key idea here is that a predicate lock applies even to objects that do not yet exist in the database, but which might be added in the future (phantoms). If two-phase locking includes predicate locks, the database prevents all forms of write skew and other race conditions, and so its isolation becomes serializable.

Index-range locks

Unfortunately, predicate locks do not perform well: if there are many locks by active transactions, checking for matching locks becomes time-consuming. For that reason, most databases with 2PL actually implement *index-range locking* (also known as *next-key locking*), which is a simplified approximation of predicate locking [52, 62].

It's safe to simplify a predicate by making it match a greater set of objects. For example, if you have a predicate lock for bookings of room 123 between noon and 1 p.m., you can approximate it by locking bookings for room 123 at any time, or you can approximate it by locking all rooms (not just room 123) between noon and 1 p.m. This is safe because any write that matches the original predicate will definitely also match the approximations.

In the room bookings database you would probably have an index on the `room_id` column, and/or indexes on `start_time` and `end_time` (otherwise the preceding query would be very slow on a large database):

- Say your index is on `room_id`, and the database uses this index to find existing bookings for room 123. Now the

database can simply attach a shared lock to this index entry, indicating that a transaction has searched for bookings of room 123.

- Alternatively, if the database uses a time-based index to find existing bookings, it can attach a shared lock to a range of values in that index, indicating that a transaction has searched for bookings that overlap with the time period of noon to 1 p.m. on January 1, 2025.

Either way, an approximation of the search condition is attached to one of the indexes. Now, if another transaction wants to insert, update, or delete a booking for the same room and/or an overlapping time period, it will have to update the same part of the index. In the process of doing so, it will encounter the shared lock, and it will be forced to wait until the lock is released.

This provides effective protection against phantoms and write skew. Index-range locks are not as precise as predicate locks would be (they may lock a bigger range of objects than is strictly necessary to maintain serializability), but since they have much lower overheads, they are a good compromise.

If there is no suitable index where a range lock can be attached, the database can fall back to a shared lock on the entire table.

This will not be good for performance, since it will stop all other transactions writing to the table, but it's a safe fallback position.

Serializable Snapshot Isolation (SSI)

This chapter has painted a bleak picture of concurrency control in databases. On the one hand, we have implementations of serializability that don't perform well (two-phase locking) or don't scale well (serial execution). On the other hand, we have weak isolation levels that have good performance, but are prone to various race conditions (lost updates, write skew, phantoms, etc.). Are serializable isolation and good performance fundamentally at odds with each other?

It seems not: an algorithm called *serializable snapshot isolation* (SSI) provides full serializability with only a small performance penalty compared to snapshot isolation. SSI is comparatively new: it was first described in 2008 [51, 63].

Today SSI and similar algorithms are used in single-node databases (the serializable isolation level in PostgreSQL [52], SQL Server's In-Memory OLTP/Hekaton [64], and HyPer [65]), distributed databases (CockroachDB [5] and FoundationDB [8]), and embedded storage engines such as BadgerDB.

Pessimistic versus optimistic concurrency control

Two-phase locking is a so-called *pessimistic* concurrency control mechanism: it is based on the principle that if anything might possibly go wrong (as indicated by a lock held by another transaction), it's better to wait until the situation is safe again before doing anything. It is like *mutual exclusion*, which is used to protect data structures in multi-threaded programming.

Serial execution is, in a sense, pessimistic to the extreme: it is essentially equivalent to each transaction having an exclusive lock on the entire database (or one shard of the database) for the duration of the transaction. We compensate for the pessimism by making each transaction very fast to execute, so it only needs to hold the “lock” for a short time.

By contrast, serializable snapshot isolation is an *optimistic* concurrency control technique. Optimistic in this context means that instead of blocking if something potentially dangerous happens, transactions continue anyway, in the hope that everything will turn out all right. When a transaction wants to commit, the database checks whether anything bad happened (i.e., whether isolation was violated); if so, the transaction is aborted and has to be retried. Only transactions that executed serializably are allowed to commit.

Optimistic concurrency control is an old idea [66], and its advantages and disadvantages have been debated for a long time [67]. 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 top of snapshot isolation, SSI adds an algorithm for detecting serialization conflicts among reads and writes, and determining which transactions to abort.

Decisions based on an outdated premise

When we previously discussed write skew in snapshot isolation (see [“Write Skew and Phantoms”](#)), we observed a recurring pattern: a transaction reads some data from the database, examines the result of the query, and decides to take some action (write to the database) based on the result that it saw. However, under snapshot isolation, the result from the original query may no longer be up-to-date by the time the transaction commits, because the data may have been modified in the meantime.

Put another way, the transaction is taking an action based on a *premise* (a fact that was true at the beginning of the transaction, e.g., “There are currently two doctors on call”). Later, when the transaction wants to commit, the original data may have changed—the premise may no longer be true.

When the application makes a query (e.g., “How many doctors are currently on call?”), the database doesn’t know how the application logic uses the result of that query. To be safe, the database needs to assume that any change in the query result (the premise) means that writes in that transaction may be invalid. In other words, there may be a causal dependency between the queries and the writes in the transaction. In order

to provide serializable isolation, the database must detect situations in which a transaction may have acted on an outdated premise and abort the transaction in that case.

How does the database know if a query result might have changed? There are two cases to consider:

- Detecting reads of a stale MVCC object version (uncommitted write occurred before the read)
- Detecting writes that affect prior reads (the write occurs after the read)

Detecting stale MVCC reads

Recall that snapshot isolation is usually implemented by multi-version concurrency control (MVCC; see [“Multi-version concurrency control \(MVCC\)”](#)). When a transaction reads from a consistent snapshot in an MVCC database, it ignores writes that were made by any other transactions that hadn’t yet committed at the time when the snapshot was taken.

In [Figure 8-10](#), transaction 43 sees Alice as having `on_call = true`, because transaction 42 (which modified Alice’s on-call status) is uncommitted. However, by the time transaction 43 wants to commit, transaction 42 has already committed. This means that the write that was ignored when reading from the

consistent snapshot has now taken effect, and transaction 43's premise is no longer true. Things get even more complicated when a writer inserts data that didn't exist before (see "[Phantoms causing write skew](#)"). We'll discuss detecting phantom writes for SSI in "[Detecting writes that affect prior reads](#)".

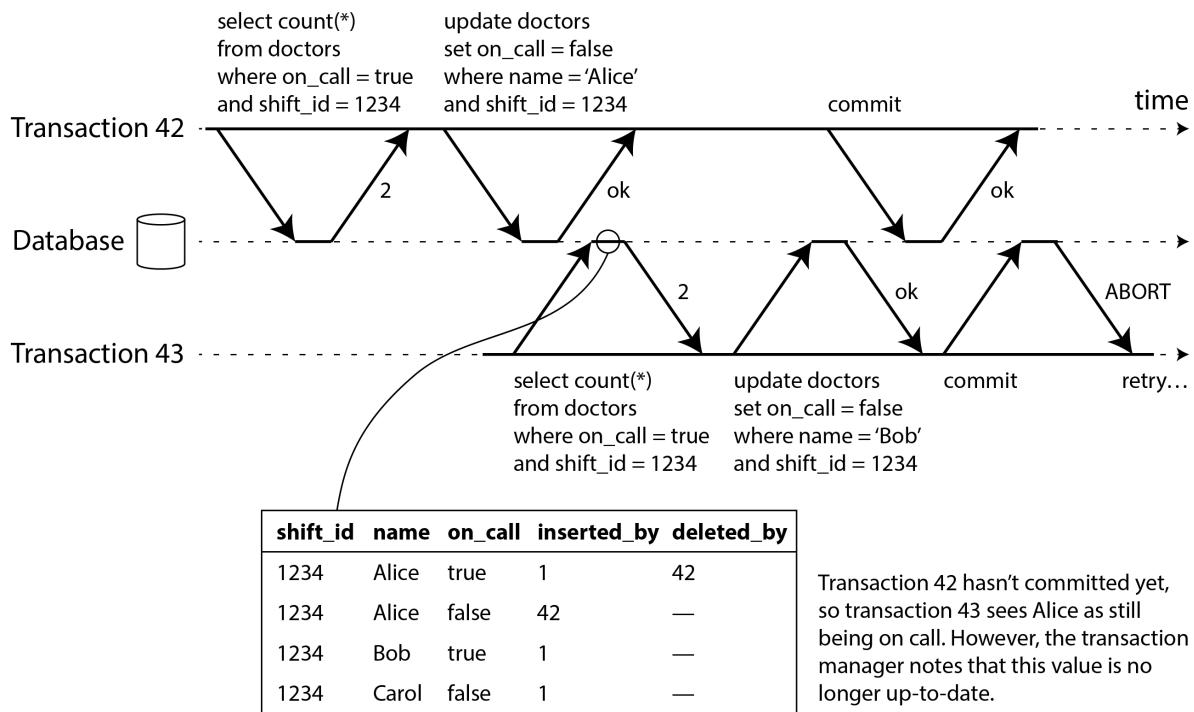


Figure 8-10. Detecting when a transaction reads outdated values from an MVCC snapshot.

In order to prevent this anomaly, the database needs to track when a transaction ignores another transaction's writes due to MVCC visibility rules. When the transaction wants to commit,

the database checks whether any of the ignored writes have now been committed. If so, the transaction must be aborted.

Why wait until committing? Why not abort transaction 43 immediately when the stale read is detected? Well, if transaction 43 was a read-only transaction, it wouldn't need to be aborted, because there is no risk of write skew. At the time when transaction 43 makes its read, the database doesn't yet know whether that transaction is going to later perform a write. Moreover, transaction 42 may yet abort or may still be uncommitted at the time when transaction 43 is committed, and so the read may turn out not to have been stale after all. By avoiding unnecessary aborts, SSI preserves snapshot isolation's support for long-running reads from a consistent snapshot.

Detecting writes that affect prior reads

The second case to consider is when another transaction modifies data after it has been read. This case is illustrated in [Figure 8-11](#).

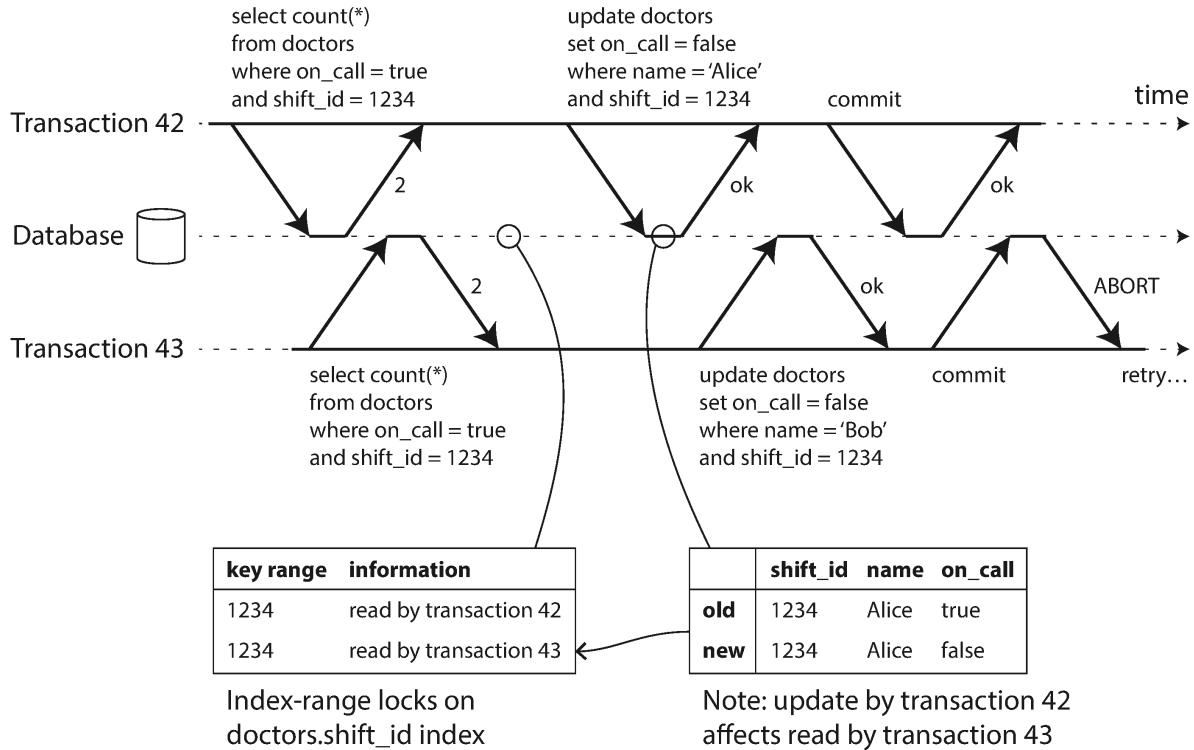


Figure 8-11. In serializable snapshot isolation, detecting when one transaction modifies another transaction's reads.

In the context of two-phase locking we discussed index-range locks (see [“Index-range locks”](#)), 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 8-11](#), transactions 42 and 43 both search for on-call doctors during shift 1234. If there is an index on `shift_id`, the database can use the index entry 1234 to record the fact that transactions 42 and 43 read this data. (If there is no index, this information can be tracked at the table level.) This information

only needs to be kept for a while: after a transaction has finished (committed or aborted), and all concurrent transactions have finished, the database can forget what data it read.

When a transaction writes to the database, it must look in the indexes for any other transactions that have recently read the affected data. This process is similar to acquiring a write lock on the affected key range, but rather than blocking until the readers have committed, the lock acts as a tripwire: it simply notifies the transactions that the data they read may no longer be up to date.

In [Figure 8-11](#), transaction 43 notifies transaction 42 that its prior read is outdated, and vice versa. Transaction 42 is first to commit, and it is successful: although transaction 43's write affected 42, 43 hasn't yet committed, so the write has not yet taken effect. However, when transaction 43 wants to commit, the conflicting write from 42 has already been committed, so 43 must abort.

Performance of serializable snapshot isolation

As always, many engineering details affect how well an algorithm works in practice. For example, one trade-off is the

granularity at which transactions' reads and writes are tracked. If the database keeps track of each transaction's activity in great detail, it can be precise about which transactions need to abort, but the bookkeeping overhead can become significant. Less detailed tracking is faster, but may lead to more transactions being aborted than strictly necessary.

In some cases, it's okay for a transaction to read information that was overwritten by another transaction: depending on what else happened, it's sometimes possible to prove that the result of the execution is nevertheless serializable. PostgreSQL uses this theory to reduce the number of unnecessary aborts [\[14, 52\]](#).

Compared to two-phase locking, the big advantage of serializable snapshot isolation is that one transaction doesn't need to block waiting for locks held by another transaction. Like under snapshot isolation, writers don't block readers, and vice versa. This design principle makes query latency much more predictable and less variable. In particular, read-only queries can run on a consistent snapshot without requiring any locks, which is very appealing for read-heavy workloads.

Compared to serial execution, serializable snapshot isolation is not limited to the throughput of a single CPU core: for example,

FoundationDB distributes the detection of serialization conflicts across multiple machines, allowing it to scale to very high throughput. Even though data may be sharded across multiple machines, transactions can read and write data in multiple shards while ensuring serializable isolation.

Compared to non-serializable snapshot isolation, the need to check for serializability violations introduces some performance overheads. How significant these overheads are is a matter of debate: some believe that serializability checking is not worth it [68], while others believe that the performance of serializability is now so good that there is no need to use the weaker snapshot isolation any more [65].

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 are okay). However, SSI is less sensitive to slow transactions than two-phase locking or serial execution.

Summary

Transactions are an abstraction layer that allows an application to pretend that certain concurrency problems and certain kinds of hardware and software faults don't exist. A large class of errors is reduced down to a simple *transaction abort*, and the application just needs to try again.

In this chapter we saw many examples of problems that transactions help prevent. Not all applications are susceptible to all those problems: an application with very simple access patterns, such as reading and writing only a single record, can probably manage without transactions. However, for more complex access patterns, transactions can hugely reduce the number of potential error cases you need to think about.

Without transactions, various error scenarios (processes crashing, network interruptions, power outages, disk full, unexpected concurrency, etc.) mean that data can become inconsistent in various ways. For example, denormalized data can easily go out of sync with the source data. Without transactions, it becomes very difficult to reason about the effects that complex interacting accesses can have on the database.

In this chapter, we went particularly deep into the topic of concurrency control. We discussed several widely used isolation levels, in particular *read committed*, *snapshot isolation* (sometimes called *repeatable read*), and *serializable*. We characterized those isolation levels by discussing various examples of race conditions, summarized in [Table 8-1](#):

Table 8-1. Summary of anomalies that can occur at various isolation levels

Isolation level	Dirty reads	Read skew	Phantom reads	Lost updates
Read uncommitted	✗ Possible	✗ Possible	✗ Possible	✗ If
Read committed	✓ Prevented	✗ Possible	✗ Possible	✗ If
Snapshot isolation	✓ Prevented	✓ Prevented	✓ Prevented	? Depends
Serializable	✓ Prevented	✓ Prevented	✓ Prevented	✓ Prevented

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

A client sees different parts of the database at different points in time. Some cases of read skew are also known as *nonrepeatable reads*. This issue is most commonly prevented with snapshot isolation, which allows a transaction to read from a consistent snapshot corresponding to one particular point in time. It is usually implemented with *multi-version concurrency control* (MVCC).

Lost updates

Two clients concurrently perform a read-modify-write cycle. One overwrites the other's write without incorporating its changes, so data is lost. Some implementations of snapshot isolation prevent this

anomaly automatically, while others require a manual lock (`SELECT FOR UPDATE`).

Write skew

A transaction reads something, makes a decision based on the value it saw, and writes the decision to the database. However, by the time the write is made, the premise of the decision is no longer true. Only serializable isolation prevents this anomaly.

Phantom reads

A transaction reads objects that match some search condition. Another client makes a write that affects the results of that search. Snapshot isolation prevents straightforward phantom reads, but phantoms in the context of write skew require special treatment, such as index-range locks.

Weak isolation levels protect against some of those anomalies but leave you, the application developer, to handle others manually (e.g., using explicit locking). Only serializable isolation protects against all of these issues. We discussed three different approaches to implementing serializable transactions:

Literally executing transactions in a serial order

If you can make each transaction very fast to execute (typically by using stored procedures), and the transaction throughput is low enough to process on a single CPU core or can be sharded, this is a simple and effective option.

Two-phase locking

For decades this has been the standard way of implementing serializability, but many applications avoid using it because of its poor performance.

Serializable snapshot isolation (SSI)

A comparatively new algorithm that avoids most of the downsides of the previous approaches. It uses an optimistic approach, allowing transactions to proceed without blocking. When a transaction wants to commit, it is checked, and it is aborted if the execution was not serializable.

The examples in this chapter used a relational data model. However, as discussed in [“The need for multi-object transactions”](#), transactions are a valuable database feature, no matter which data model is used.

In this chapter, we explored ideas and algorithms mostly in the context of a database running on a single machine.

Transactions in distributed databases open a new set of difficult challenges, which we'll discuss in the next two chapters.

FOOTNOTES

REFERENCES

Steven J. Murdoch. [What went wrong with Horizon: learning from the Post Office Trial.](#) *benthamsgaze.org*, July 2021. Archived at [perma.cc/CNM4-553F](#)

Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. [A History and Evaluation of System R.](#) *Communications of the ACM*, volume 24, issue 10, pages 632–646, October 1981. [doi:10.1145/358769.358784](#)

Jim N. Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. [Granularity of Locks and Degrees of Consistency in a Shared Data Base.](#) in *Modelling in Data Base Management Systems: Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems*, edited by G. M. Nijssen, pages 364–394, Elsevier/North Holland Publishing, 1976. Also in *Readings in Database Systems*, 4th edition, edited by Joseph M. Hellerstein and Michael Stonebraker, MIT Press, 2005. ISBN: 978-0-262-69314-1

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, issue 11, pages 624–633, November 1976. [doi:10.1145/360363.360369](#)

Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta

Ranade, Ben Darnell, Bram Grunen, Justin Jaffray, Lucy Zhang, and Peter Mattis. [CockroachDB: The Resilient Geo-Distributed SQL Database](#). At *ACM SIGMOD International Conference on Management of Data* (SIGMOD), pages 1493–1509, June 2020. [doi:10.1145/3318464.3386134](https://doi.org/10.1145/3318464.3386134)

Jongxu Huang, Qi Liu, Qiu Cui, Zuhu Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. [TiDB: a Raft-based HTAP database](#). *Proceedings of the VLDB Endowment*, volume 13, issue 12, pages 3072–3084. [doi:10.14778/3415478.3415535](https://doi.org/10.14778/3415478.3415535)

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. [Spanner: Google's Globally-Distributed Database](#). At *10th USENIX Symposium on Operating System Design and Implementation* (OSDI), October 2012.

Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. [FoundationDB: A Distributed Unbundled Transactional Key Value Store](#). At *ACM International Conference on Management of Data* (SIGMOD), June 2021. [doi:10.1145/3448016.3457559](https://doi.org/10.1145/3448016.3457559)

Theo Härdter and Andreas Reuter. [Principles of Transaction-Oriented Database Recovery](#). *ACM Computing Surveys*, volume 15, issue 4, pages 287–317, December 1983. [doi:10.1145/289.291](https://doi.org/10.1145/289.291)

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.

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. [doi:10.1145/268998.266662](https://doi.org/10.1145/268998.266662)

Tony Andrews. [Enforcing Complex Constraints in Oracle](#). tonyandrews.blogspot.co.uk, October 2004. Archived at archive.org

Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. [Concurrency Control and Recovery in Database Systems](#). Addison-Wesley, 1987. ISBN: 978-0-201-10715-9, available online at microsoft.com.

Alan Fekete, Dimitrios Liarokapis, Elizabeth O’Neil, Patrick O’Neil, and Dennis Shasha. [Making Snapshot Isolation Serializable](#). *ACM Transactions on Database Systems*, volume 30, issue 2, pages 492–528, June 2005. [doi:10.1145/1071610.1071615](https://doi.org/10.1145/1071610.1071615)

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.

Laurie Denness. [SSDs: A Gift and a Curse](#). laurie.ie, June 2015. Archived at perma.cc/6GLP-BX3T

Adam Surak. [When Solid State Drives Are Not That Solid](#). blog.algolia.com, June 2015. Archived at perma.cc/CBR9-QZEE

Hewlett Packard Enterprise. [Bulletin: \(Revision\) HPE SAS Solid State Drives - Critical Firmware Upgrade Required for Certain HPE SAS Solid State Drive Models to Prevent Drive Failure at 32,768 Hours of Operation](#). support.hpe.com, November 2019. Archived at perma.cc/CZR4-AQBS

Craig Ringer et al. [PostgreSQL's handling of fsync\(\) errors is unsafe and risks data loss at least on XFS](#). Email thread on pgsql-hackers mailing list, postgresql.org, March 2018. Archived at perma.cc/5RKU-57FL

Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. [Can Applications Recover from fsync Failures?](#) At *USENIX Annual Technical Conference* (ATC), July 2020.

Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. [Crash Consistency: Rethinking the Fundamental Abstractions of the File System](#). *ACM Queue*, volume 13, issue 7, pages 20–28, July 2015. [doi:10.1145/2800695.2801719](https://doi.org/10.1145/2800695.2801719)

Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. [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.

Chris Siebenmann. [Unix's File Durability Problem](#). utcc.utoronto.ca, April 2016. Archived at perma.cc/VSS8-5MC4

Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. [Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions](#). At *15th USENIX Conference on File and Storage Technologies* (FAST), February 2017.

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.

Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. [Flash Reliability in Production: The Expected and the Unexpected](#). At *14th USENIX Conference on File and Storage Technologies* (FAST), February 2016.

Don Allison. [SSD Storage – Ignorance of Technology Is No Excuse](#). *blog.korelogic.com*, March 2015. Archived at [perma.cc/9QN4-9SNJ](#)

Gordon Mah Ung. [Debunked: Your SSD won't lose data if left unplugged after all](#). *pcworld.com*, May 2015. Archived at [perma.cc/S46H-JUDU](#)

Martin Kleppmann. [Hermitage: Testing the 'I' in ACID](#). *martin.kleppmann.com*, November 2014. Archived at [perma.cc/KP2Y-AQGK](#)

Todd Warszawski and Peter Bailis. [ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications](#). At *ACM International Conference on Management of Data* (SIGMOD), May 2017. [doi:10.1145/3035918.3064037](#)

Tristan D'Agosta. [BTC Stolen from Poloniex](#). *bitcointalk.org*, March 2014. Archived at [perma.cc/YHA6-4C5D](#)

bitcointhief2. [How I Stole Roughly 100 BTC from an Exchange and How I Could Have Stolen More!](#) *reddit.com*, February 2014. Archived at [archive.org](#)

Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. [Automating the Detection of Snapshot Isolation Anomalies](#). At *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.

Michael Melanson. [Transactions: The Limits of Isolation](#). *michaelmelanson.net*, November 2014. Archived at [perma.cc/RG5R-KMYZ](#)

Edward Kim. [How ACH works: A developer perspective — Part 1](#). *engineering.gusto.com*, April 2014. Archived at [perma.cc/7B2H-PU94](#)

Hal Berenson, Philip A. Bernstein, Jim N. Gray, Jim Melton, Elizabeth O’Neil, and Patrick O’Neil. [A Critique of ANSI SQL Isolation Levels](#). At *ACM International Conference on Management of Data* (SIGMOD), May 1995. [doi:10.1145/568271.223785](#)

Atul Adya. [Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions](#). PhD Thesis, Massachusetts Institute of Technology, March 1999. Archived at [perma.cc/E97M-HW5Q](#)

Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. [Highly Available Transactions: Virtues and Limitations](#). At *40th International Conference on Very Large Data Bases* (VLDB), September 2014.

Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. [Seeing is Believing: A Client-Centric Specification of Database Isolation](#). At *ACM Symposium on Principles of Distributed Computing* (PODC), pages 73–82, July 2017. [doi:10.1145/3087801.3087802](#)

Bruce Momjian. [MVCC Unmasked](#). *momjian.us*, July 2014. Archived at [perma.cc/KQ47-9GYB](#)

Peter Alvaro and Kyle Kingsbury. [MySQL 8.0.34](#). *jepsen.io*, December 2023. Archived at [perma.cc/HGE2-Z878](#)

Egor Rogov. [PostgreSQL 14 Internals](#). *postgrespro.com*, April 2023. Archived at [perma.cc/FRK2-D7WB](#)

Hironobu Suzuki. [The Internals of PostgreSQL](#). *interdb.jp*, 2017.

Andy Pavlo and Bohan Zhang. [The Part of PostgreSQL We Hate the Most](#). *cs.cmu.edu*, April 2023. Archived at [perma.cc/XSP6-3JBN](#)

Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. [An empirical evaluation of in-memory multi-version concurrency control](#). *Proceedings of the VLDB*

Endowment, volume 10, issue 7, pages 781–792, March 2017.

[doi:10.14778/3067421.3067427](https://doi.org/10.14778/3067421.3067427)

Nikita Prokopov. [Unofficial Guide to Datomic Internals](#). *tonsky.me*, May 2014.

Nate Wiger. [An Atomic Rant](#). *nateware.com*, February 2010. Archived at perma.cc/5ZYB-PE44

James Coglan. [Reading and writing, part 3: web applications](#). *blog.jcoglan.com*, October 2020. Archived at perma.cc/A7EK-PJVS

Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. [Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity](#). At *ACM International Conference on Management of Data* (SIGMOD), June 2015. [doi:10.1145/2723372.2737784](https://doi.org/10.1145/2723372.2737784)

Jaana Dogan. [Things I Wished More Developers Knew About Databases](#). *rakyll.medium.com*, April 2020. Archived at perma.cc/6EFK-P2TD

Michael J. Cahill, Uwe Röhm, and Alan Fekete. [Serializable Isolation for Snapshot Databases](#). At *ACM International Conference on Management of Data* (SIGMOD), June 2008. [doi:10.1145/1376616.1376690](https://doi.org/10.1145/1376616.1376690)

Dan R. K. Ports and Kevin Grittner. [Serializable Snapshot Isolation in PostgreSQL](#). At *38th International Conference on Very Large Databases* (VLDB), August 2012.

Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer and Carl H. Hauser. [Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System](#). At *15th ACM Symposium on Operating Systems Principles* (SOSP), December 1995. [doi:10.1145/224056.224070](https://doi.org/10.1145/224056.224070)

Hans-Jürgen Schönig. [Constraints over multiple rows in PostgreSQL](#). *cybertec-postgresql.com*, June 2021. Archived at perma.cc/2TGH-XUPZ

Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. [The End of an Architectural Era \(It's Time for a Complete Rewrite\)](#). At *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.

John Hugg. [H-Store/VoltDB Architecture vs. CEP Systems and Newer Streaming Architectures](#). At *Data @Scale Boston*, November 2014.

Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. [H-Store: A High-Performance, Distributed Main Memory Transaction Processing System](#). *Proceedings of the VLDB Endowment*, volume 1, issue 2, pages 1496–1499, August 2008.

Rich Hickey. [The Architecture of Datomic](#). *infoq.com*, November 2012. Archived at perma.cc/5YWU-8XJK

John Hugg. [Debunking Myths About the VoltDB In-Memory Database](#). *dzone.com*, May 2014. Archived at perma.cc/2Z9N-HPKF

Xinjing Zhou, Viktor Leis, Xiangyao Yu, and Michael Stonebraker. [OLTP Through the Looking Glass 16 Years Later: Communication is the New Bottleneck](#). At *15th Annual Conference on Innovative Data Systems Research* (CIDR), January 2025.

Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. [Lotus: scalable multi-partition transactions on single-threaded partitioned databases](#). *Proceedings of the VLDB Endowment* (PVLDB), volume 15, issue 11, pages 2939–2952, July 2022.
[doi:10.14778/3551793.3551843](https://doi.org/10.14778/3551793.3551843)

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

Michael J. Cahill. [Serializable Isolation for Snapshot Databases](#). PhD Thesis, University of Sydney, July 2009. Archived at perma.cc/727J-NTMP

Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. [Hekaton: SQL Server's Memory-Optimized OLTP Engine](#). At *ACM SIGMOD International Conference on Management of Data* (SIGMOD), pages 1243–1254, June 2013. [doi:10.1145/2463676.2463710](https://doi.org/10.1145/2463676.2463710)

Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. [Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems](#). At *ACM SIGMOD International Conference on Management of Data* (SIGMOD), pages 677–689, May 2015. [doi:10.1145/2723372.2749436](https://doi.org/10.1145/2723372.2749436)

D. Z. Badal. [Correctness of Concurrency Control and Implications in Distributed Databases](#). At *3rd International IEEE Computer Software and Applications Conference* (COMPSAC), November 1979. [doi:10.1109/CMPSC.1979.762563](https://doi.org/10.1109/CMPSC.1979.762563)

Rakesh Agrawal, Michael J. Carey, and Miron Livny. [Concurrency Control Performance Modeling: Alternatives and Implications](#). *ACM Transactions on Database Systems* (TODS), volume 12, issue 4, pages 609–654, December 1987. [doi:10.1145/32204.32220](https://doi.org/10.1145/32204.32220)

Marc Brooker. [Snapshot Isolation vs Serializability](#). brooker.co.za, December 2024. Archived at perma.cc/5TRC-CR5G

About the Authors

Martin Kleppmann is a researcher in distributed systems at the University of Cambridge, UK. Previously he was a software engineer and entrepreneur at internet companies including LinkedIn and Rapportive, where he worked on large-scale data infrastructure. In the process he learned a few things the hard way, and he hopes this book will save you from repeating the same mistakes.

Martin is a regular conference speaker, blogger, and open source contributor. He believes that profound technical ideas should be accessible to everyone, and that deeper understanding will help us develop better software.

Chris Riccomini is a software engineer, startup investor, and author with 15+ years of experience at PayPal, LinkedIn, and WePay. He runs Materialized View Capital, where he invests in infrastructure startups. He is also the cocreator of Apache Samza and SlateDB, and coauthor of *The Missing README: A Guide for the New Software Engineer*.