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

Beijing • Boston • Farnham • Sebastopol • Tokyo

Designing Data-Intensive Applications

by Martin Kleppmann

Printed in the United States of America.

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

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

Acquisitions Editor: Aaron Black	
Development Editor: Virginia Wilson	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

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

978-1-098-11900-3

Dedication

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

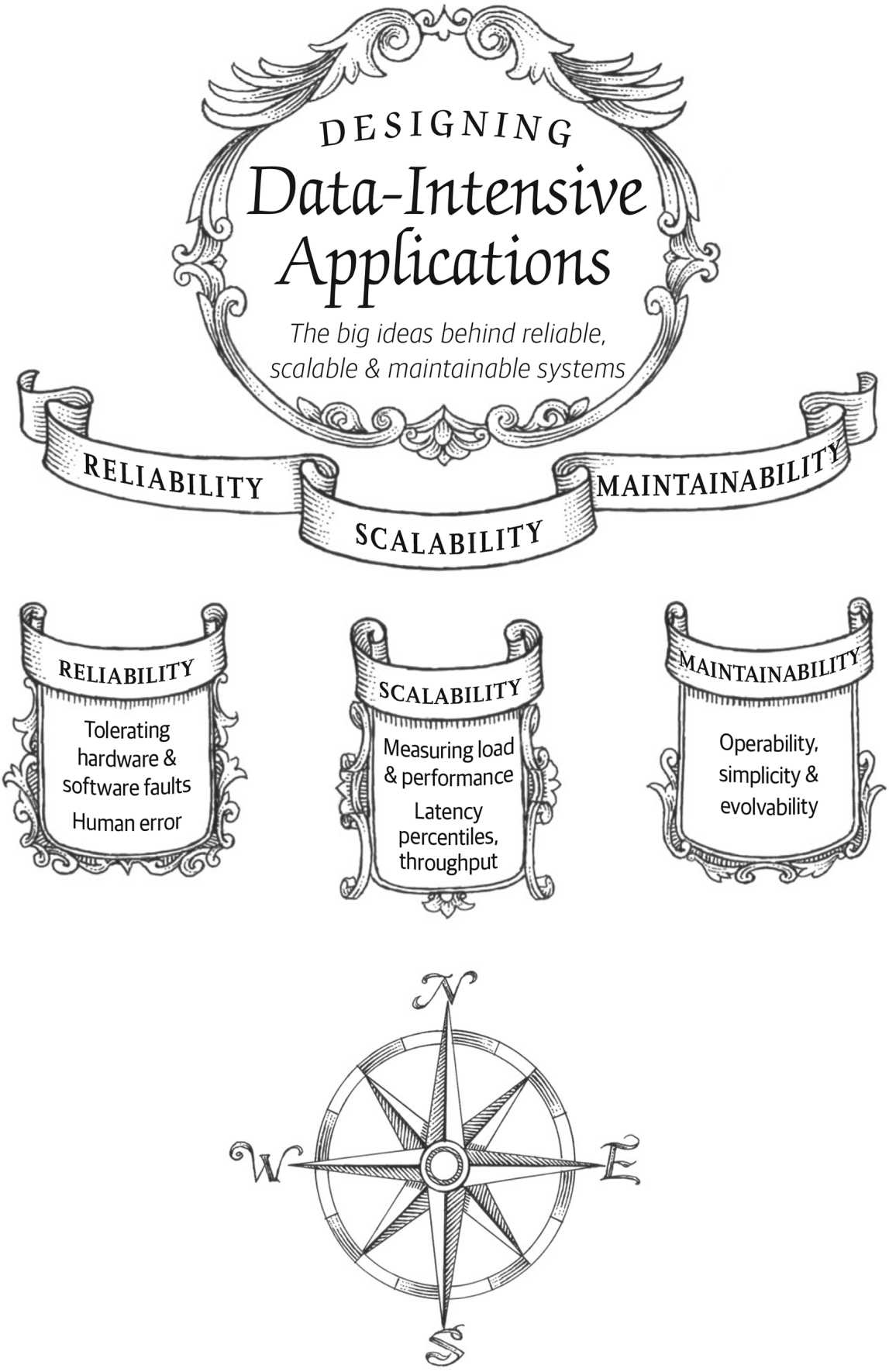
Computing is pop culture. […] Pop culture holds a disdain for history. Pop culture is all about identity and feeling like you're participating. It has nothing to do with cooperation, the past or the future—it's living in the present. I think the same is true of most people who write code for money. They 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. 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)


DESIGNING
Data-Intensive Applications
The big ideas behind reliable, scalable & maintainable systems
RELIABILITY
SCALABILITY
MAINTAINABILITY
RELIABILITY
Tolerating hardware & software faults
Human error
SCALABILITY
Measuring load & performance
Latency percentiles, throughput
MAINTAINABILITY
Operability, simplicity & evolvability
N
W
E
S

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, although you might even 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 transaction processing and analytics (“Transaction Processing versus Analytics”);
- 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; 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.

Transaction Processing versus Analytics

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 end users querying data 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 Analytical and Operational Systems

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

[Link to Come] 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)
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.

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 operational 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 [6]. As we shall see in [Link to Come], 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*.

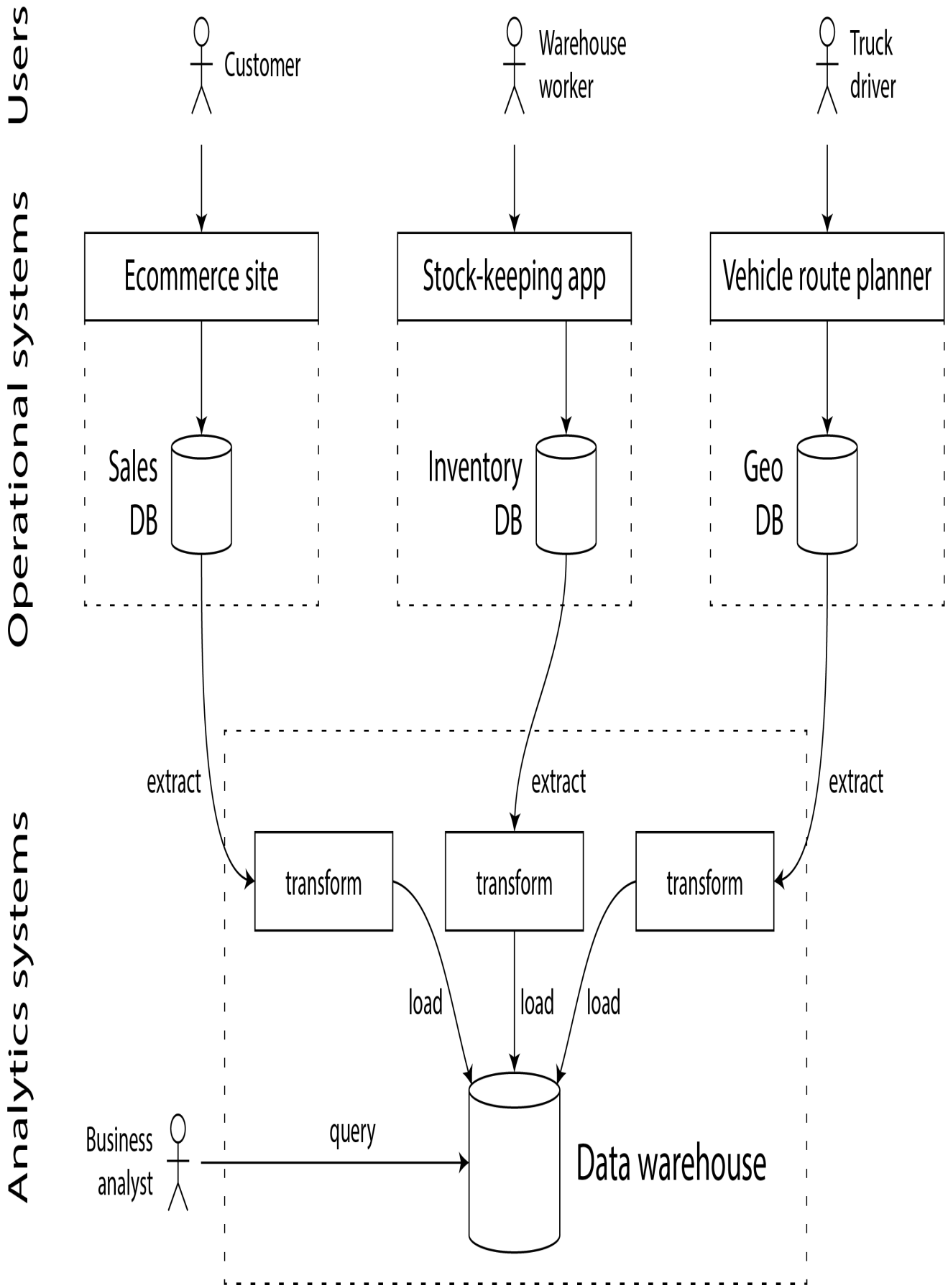

Users
Operational systems
Analytics systems
Customer
Warehouse worker
Truck driver
Ecommerce site
Stock-keeping app
Vehicle route planner
Sales DB
Inventory DB
Geo DB
extract
extract
extract
transform
transform
transform
load
load
load
Business analyst
query
Data warehouse

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 [7, 8]. However, many HTAP systems internally consist of an OLTP system coupled with a separate analytical system, hidden behind a common interface—so the distinction beween 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.

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 [9].

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 [10] and to build efficient machine learning systems on top of a relational foundation [11], 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 [12]. 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 [Link to Come]), but they can equally well contain text, images, videos, sensor readings, sparse matrices, feature vectors, genome sequences, or any other kind of data [13].

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" [14].

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 [15]. 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 [16]. 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* [17]). 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* [18]. 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 [19]. 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.

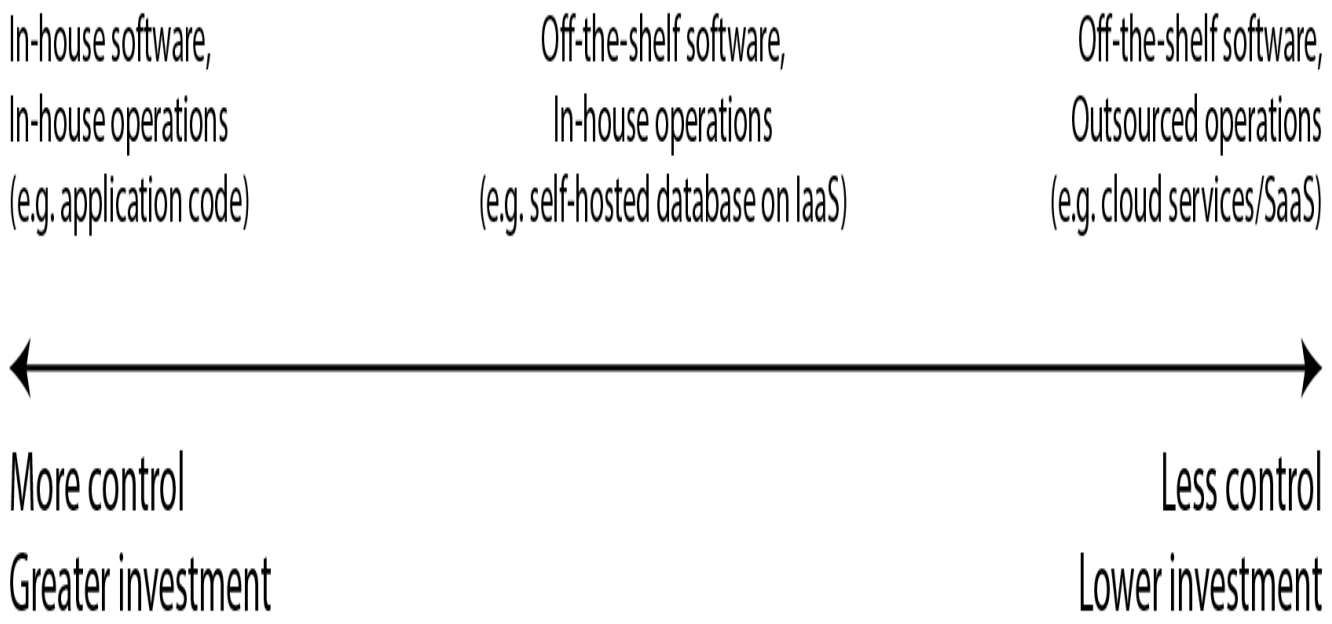


Figure 1-2. A spectrum of types of software and its operations.

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.

Seperately 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 [20, 21].

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 [22]. 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.

Despite all these risks, it has become more and more popular for organizations to build new applications on top of cloud services. 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 [23, 24, 25]. 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 [23], Azure SQL DB Hyperscale [24], Google Cloud Spanner
Analytical/OLAP	Teradata, ClickHouse, Spark	Snowflake [25], 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 [25], 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 it often suffers from poor performance and poor scalability [23].

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 [24].

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* [8, 25, 26, 27]: 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 [28]. 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 [29].

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 [30].

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 [31].

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 [32].

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 [33]. 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 [34].

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 [35, 36]. 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”.

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 at various locations worldwide so that each user can be served from a

datacenter that is geographically close to them. That avoids the users having to wait for network packets to travel halfway around the world 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 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 [37]. 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 [38]. 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 [39]. 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 [40].

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* [41, 42], 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 allow you to track which client called which server for which operation, and how long each call took [43].

Databases provide various mechanisms for ensuring data consistency, as we shall see in [Link to Come] and [Link to Come]. 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 [44].

For all these reasons, if you can do something on a single machine, this is often much simpler and easier compared to setting up a distributed system [21]. 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 [Link to Come].

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 [Link to Come]. 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 [45, 46]. 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 [Link to Come].

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 [45].

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 [29]. 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 [47]. The term “serverless” can misleading: each serverless function execution still runs on a server, but subsequent executions might run on a different one.

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, 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 [48, 49]. With cloud services, it is usually not desirable to stop the entire cluster, since the services need to continually serve users with minimal interruptions.
- Supercomputers are typically built from specialized hardware, where each node is quite reliable. Nodes in cloud services are usually built from commodity machines, which can provide equivalent performance at lower cost due to economies of scale, but which also have higher failure rates (see "Hardware and Software Faults").
- 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 [50]. 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 [48, 51]. Supercomputers often use specialized network topologies, such as multi-dimensional meshes and toruses [52], which yield better performance for HPC workloads with known communication patterns.

- Cloud computing allows nodes to be distributed across multiple geographic locations, 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 [53].

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 [54]. 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.

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 [55]. But it fits with the GDPR, which mandates that personal data many 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 [56].

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 [Link to Come] 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

[1] 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

[2] 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

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

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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

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

[12] 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

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

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

[15] 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.

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

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

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

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

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

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

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

[23] 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

[24] 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

[25] Midhul Vuppalapati, 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.

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

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

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

[29] 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.

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

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

[32] Charity Majors. The Future of Ops Jobs. *acloudguru.com*, August 2020. Archived at perma.cc/GRU2-CZG3

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

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

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

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

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

[38] Kousik Nath. These are the numbers every computer engineer should know. *freecodecamp.org*, September 2019. Archived at perma.cc/RW73-36RL

[39] 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.

[40] 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.

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

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

[43] 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

[44] 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

[45] Sam Newman. *Building Microservices*, second edition. O'Reilly Media, 2021. ISBN: 9781492034025

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

[47] 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.

[48] Luiz André Barroso, Urs Hölzle, 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

[49] 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

[50] 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.

[51] 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 Hölzle, 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

[52] Glenn K. Lockwood. Hadoop's Uncomfortable Fit in HPC. *glennklockwood.blogspot.co.uk*, May 2014. Archived at perma.cc/S8XX-Y67B

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

[54] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. 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

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

[56] 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.

Not all nonfunctional requirements fall within the scope of this book, but several do. In this chapter we will introduce several technical concepts that will help you articulate the nonfunctional requirements 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.

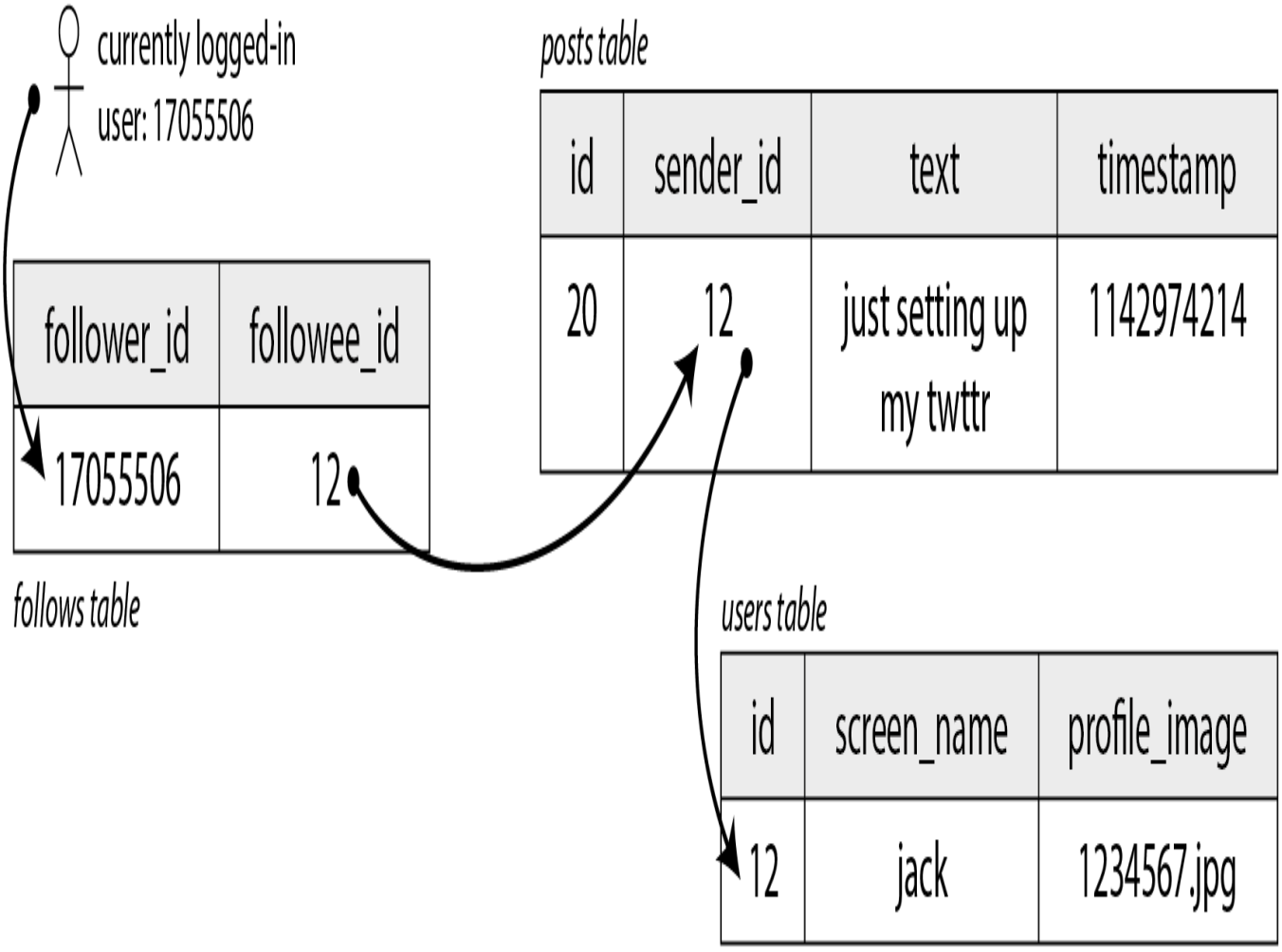


Figure 2-1. Simple relational schema for a social network in which users can follow each other.

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.followee_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.

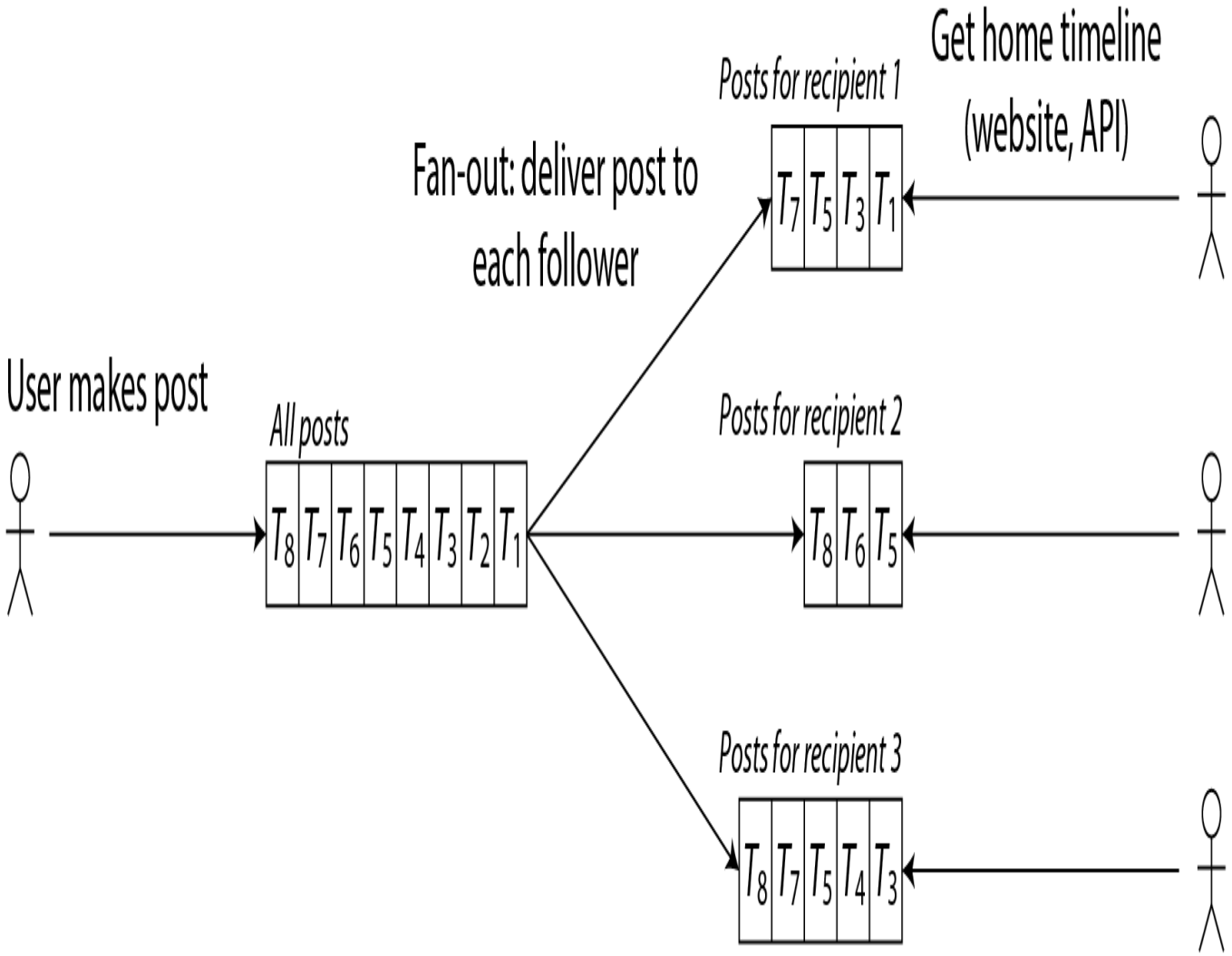


Figure 2-2. Fan-out: delivering new posts to every follower of the user who made the post.

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.

Throughput
The number of requests per second, or the data volume per second, that the system is processing. For a given a particular 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.

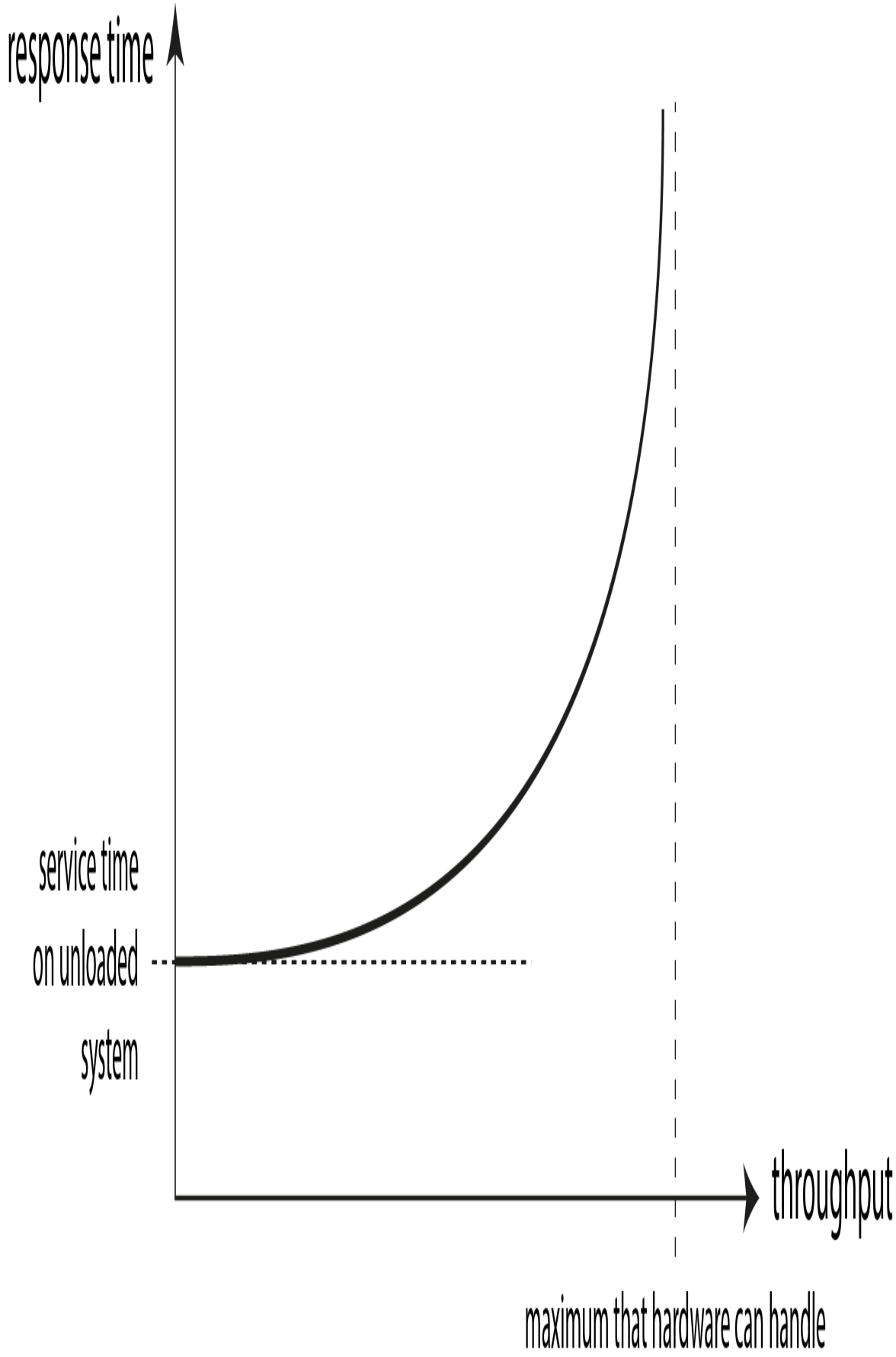

response time
service time
on unloaded
system
throughput
maximum that hardware can handle

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] or *token bucket* algorithm [11]). The server can also detect when it is approaching overload and start proactively rejecting requests (*load shedding* [12]), and send back responses asking clients to slow down (*backpressure* [1, 13]). The choice of queueing and load-balancing algorithms can also make a difference [14].

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.

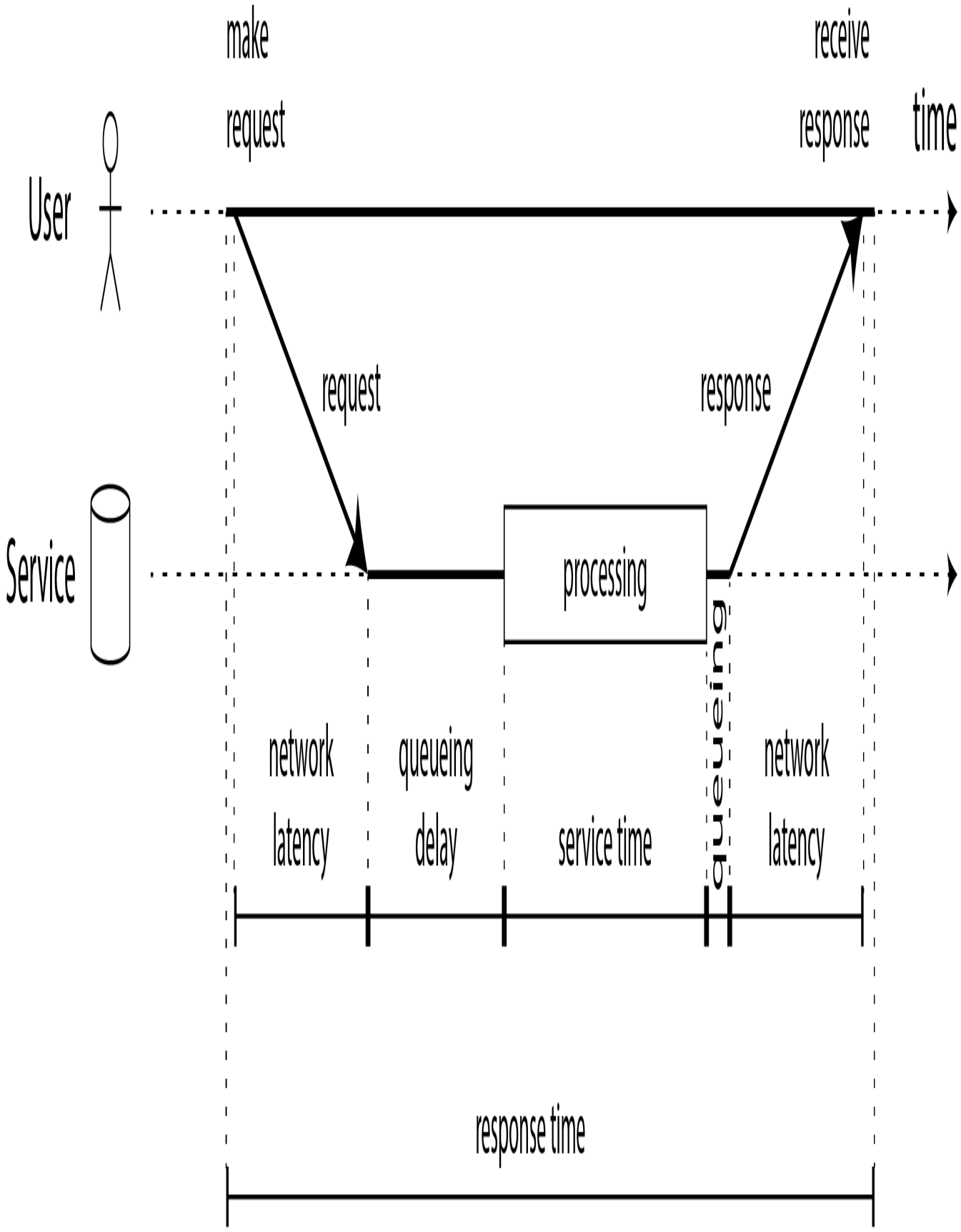

make
request
receive
response
time
User
request
response
Service
processing
queueing
network
latency
queueing
delay
service time
network
latency
response time

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

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 [15], 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*.

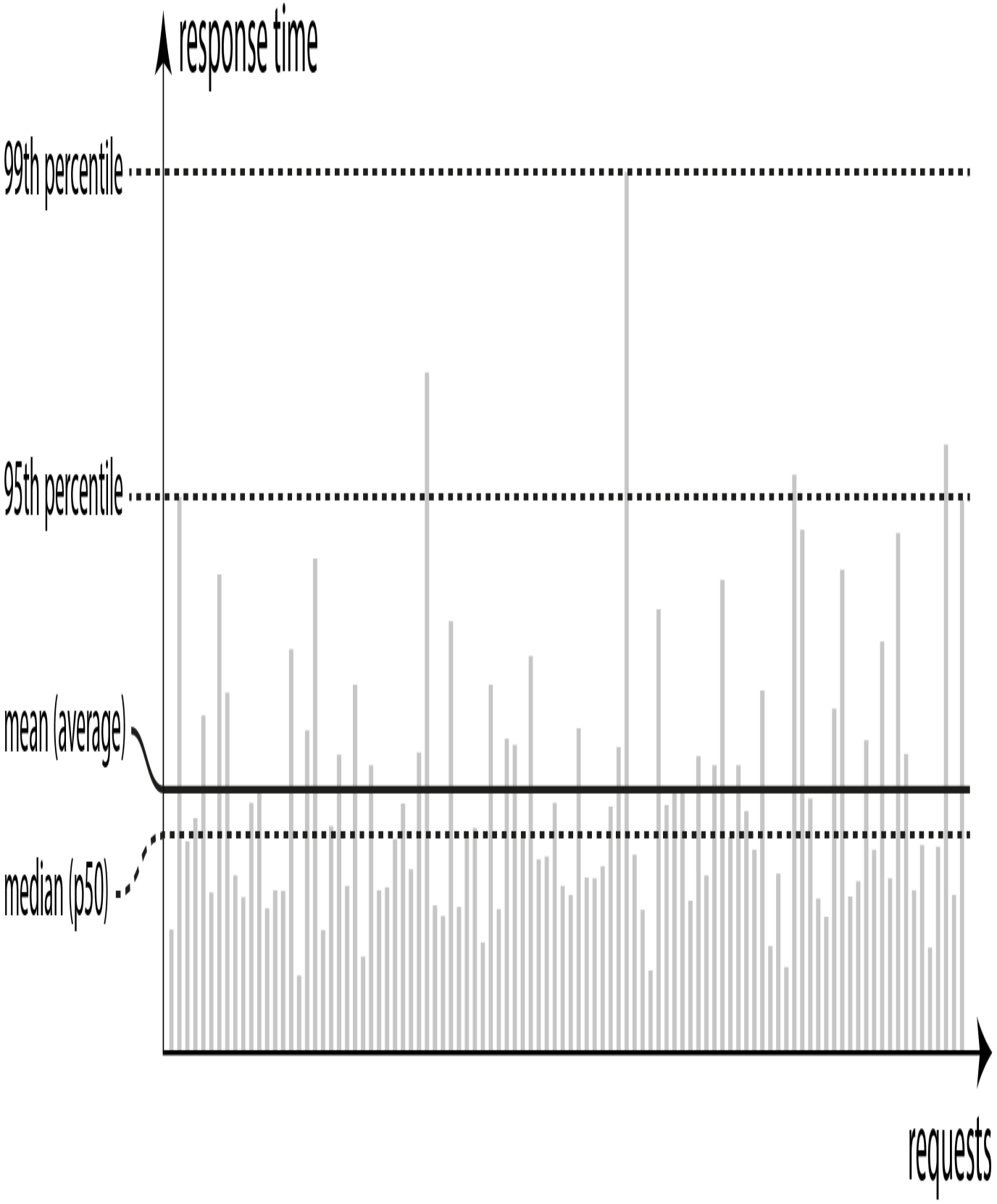

response time
99th percentile
95th percentile
mean (average)
median (p50)
requests

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). 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 [16]. 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 [17]. 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 [18]. However, another Google study from 2009 reported that a 400 ms increase in latency resulted in only 0.6% fewer searches per day [19], and in the same year Bing found that a two-second increase in load time reduced ad revenue by 4.3% [20]. Newer data from these companies appears not to be publicly available.

A more recent Akamai study [21] 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 [22] 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* [23]).

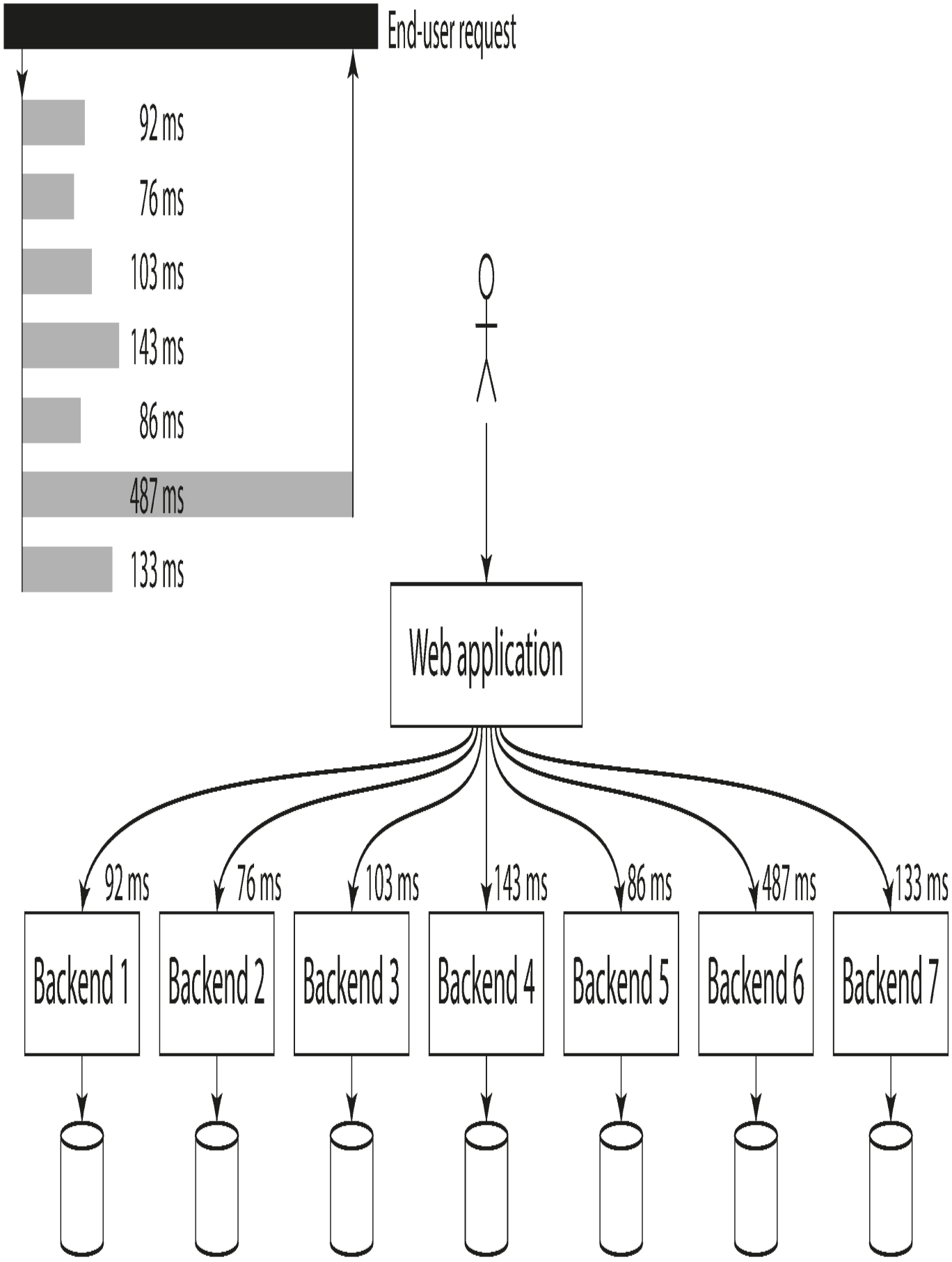

End-user request
92 ms
76 ms
103 ms
143 ms
86 ms
487 ms
133 ms
Web application
92 ms
76 ms
103 ms
143 ms
86 ms
487 ms
133 ms
Backend 1
Backend 2
Backend 3
Backend 4
Backend 5
Backend 6
Backend 7

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 [24]. 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 [25, 26].

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 [27, 28], OpenHistogram [29], and DDSketch [30].

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 [31].

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* [32, 33]:

> *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 become 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. Many critical bugs are actually due to poor error handling [34]; 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 [35].

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 [36, 37]; 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 [38].

- Approximately 0.5–1% of solid state drives (SSDs) fail per year [39]. Small numbers of bit errors are corrected automatically [40], 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 [41, 42].
- Other hardware components such as power supplies, RAID controllers, and memory modules also fail, although less frequently than hard drives [43, 44].
- Approximately one in 1,000 machines has a CPU core that occasionally computes the wrong result, likely due to manufacturing defects [45, 46, 47]. 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 [48]. Moreover, certain pathological memory access patterns can flip bits with high probability [49].
- 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 [50].

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 [37, 51, 52]; 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 [Link to Come], [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 [Link to Come].

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 [53, 54]. Such faults are harder to anticipate, and they tend to cause many more system failures than uncorrelated hardware faults [43]. 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 [55]. 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 [56].
- A runaway process that uses up some shared, limited resource, such as CPU time, memory, disk space, network bandwidth, or threads [57]. For example, a process that consumes too much memory while processing a large request may be killed by the operating system.
- 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 [58].
- Cascading failures, where a problem in one component causes another component to become overloaded and slow down, which in turn brings down another component [59, 60].

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 [61, 62].

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 [63].

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 [64].

Various technical measures can help minimize the impact of human mistakes, including thorough testing [34], 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 [65]. 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 [64].

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 [66], 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 [67]. 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 [68]. 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 [69]. 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 [70]. 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 [71, 72]. 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 [73].

By contrast, the *shared-nothing architecture* [74] (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 data partitioning (see [Link to Come]), 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 [75].

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"), partitioning ([Link to Come]), 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 [76], and we discuss partitioning of shared-nothing systems in [Link to Come].

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 [Link to Come]). 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 [77, 78].

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 [79].

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" [54].

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 [80].

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 [81]:

- 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 [82].
- 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* [83].

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 [62]. 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 [84]. 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 [85]. 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 [86].

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* [87] and *domain-driven design* (DDD) [88]. 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: simple and easy-to-understand systems are usually easier to modify than 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* [89].

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 [90]. For example, say you are migrating from one database to another: if you cannot switch back to the old system in case of problems wth 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 the most important such building blocks.

FOOTNOTES

REFERENCES

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

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

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

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

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

[6] 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

[7] Marc Brooker. Metastability and Distributed Systems. *brooker.co.za*, May 2021. Archived at archive.org

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

[9] Marc Brooker. What is Backoff For? *brooker.co.za*, August 2022. Archived at archive.org

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

[11] Marc Brooker. Fixing retries with token buckets and circuit breakers. *brooker.co.za*, February 2022. Archived at archive.org

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

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

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

[15] 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.

[16] 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

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

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

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

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

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

[22] 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

[23] 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

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

[25] 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

[26] 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.

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

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

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

[30] 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

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

[32] 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

[33] 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

[34] 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.

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

[36] 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.

[37] 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.

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

[39] 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

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

[41] 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.

[42] 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

[43] 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.

[44] 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

[45] 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

[46] 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.

[47] 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.

[48] 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

[49] 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

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

[51] 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.

[52] 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

[53] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, 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

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

[55] Nelson Minar. Leap Second Crashes Half the Internet. *somebits.com*, July 2012. Archived at perma.cc/2WB8-D6EU

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

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

[58] 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

[59] 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

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

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

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

[63] 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.

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

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

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

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

[68] Haroon Siddique and Ben Quinn. Court clears 39 post office operators convicted due to 'corrupt data'. *theguardian.com*, April 2021. Archived at archive.org

[69] 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

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

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

[72] Marc Brooker. Surprising Scalability of Multitenancy. *brooker.co.za*, March 2023. Archived at archive.org

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

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

[75] 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

[76] Sam Newman. *Building Microservices*, second edition. O'Reilly Media, 2021. ISBN: 9781492034025

[77] 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

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

[79] Marianne Bellotti. *Kill It with Fire*. No Starch Press, April 2021. ISBN: 9781718501188

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

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

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

[83] 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

[84] Marc Brooker. What is a simple system? *brooker.co.za*, May 2022. Archived at archive.org

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

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

[87] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-

Wesley Professional, October 1994. ISBN: 9780201633610

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

[89] 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

[90] 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)


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

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 JSON/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 [Link to Come].
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 Analytical and Operational Systems"); 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 often come with schema migration tools that update database schemas as model definitions change. Such tools are handy, but should be used with caution. Migrations on large or high-traffic tables can lock the entire table for an extended amount of time, resulting in downtime. Many operations teams prefer to run schema migrations manually, incrementally, during off peak hours, or with specialized tools. Safe schema migrations are discussed further in "Schema flexibility in the document model".
- 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

users table

user_id	first_name	last_name	photo_url
251	Barack	Obama	46175777.jpg

region_id	headline
us:91	Former President of the United…

regions table

id	region_name
us:7	Greater Boston Area
us:91	Washington, DC, United States

positions table

id	user_id	job_title	organization
458	251	President	United States of America
457	251	US Senator (D-IL)	United States Senate

education table

id	user_id	school_name	start	end
807	251	Harvard University	1988	1991
806	251	Columbia University	1981	1983

contact_info table

id	user_id	type	url
155	251	website	https://barackobama.com
156	251	twitter	https://twitter.com/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 States of Am
  "region_id":   "us:91",
  "photo_url":   "/p/7/000/253/05b/308dd6e.jpg",
  "positions": [
    {"job_title": "President", "organization": "United States
    {"job_title": "US Senator (D-IL)", "organization": "Unite
  ],
  "education": [
    {"school_name": "Harvard University",  "start": 1988, "er
    {"school_name": "Columbia University", "start": 1981, "er
  ],
  "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 [Link to Come], 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).

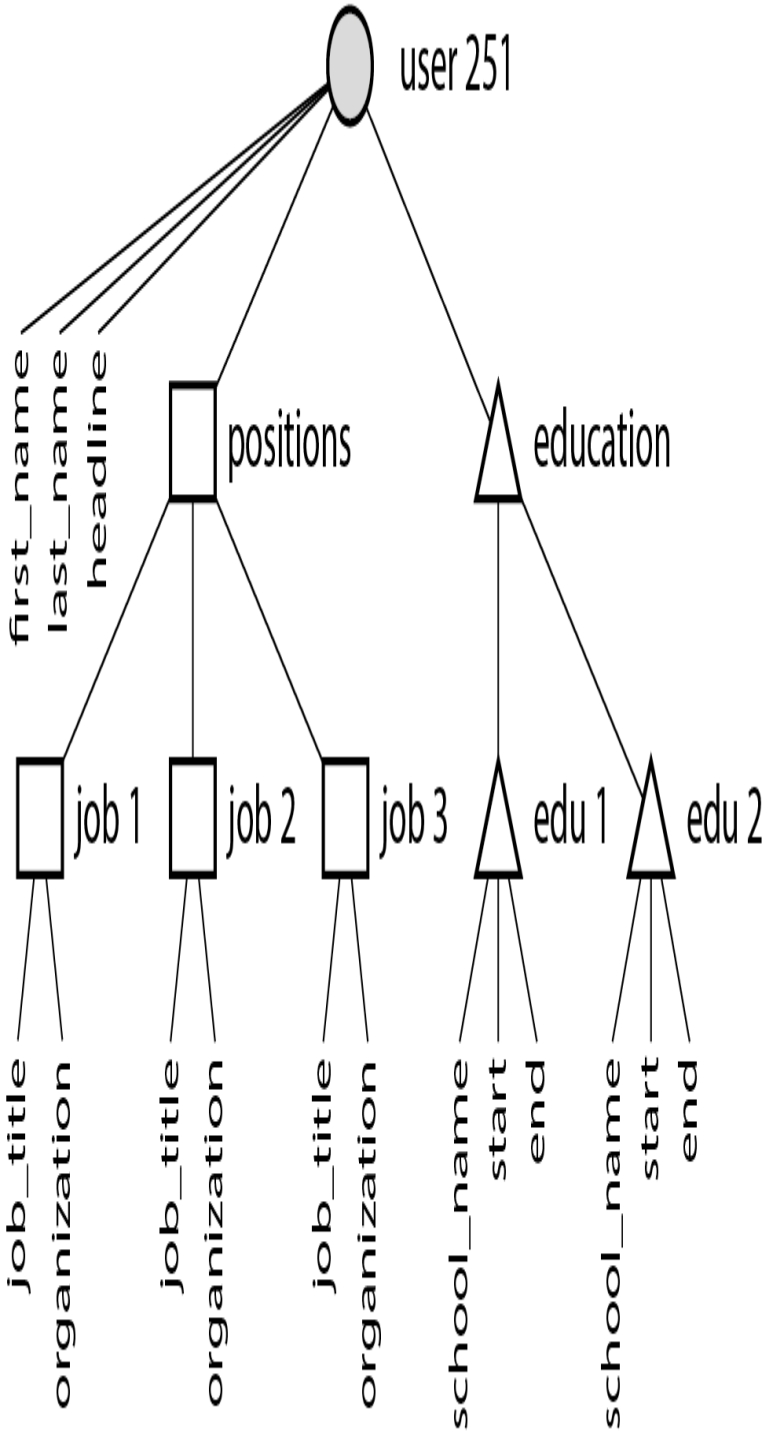


Figure 3-2. One-to-many relationships forming a tree structure.

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 sitations 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 autocompleter:

- 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, 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;
```

In a document database, it is more common to either use a denormalized representation that needs no join when reading, or to perform the join 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). 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 [Link to Come]) 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.followee_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.

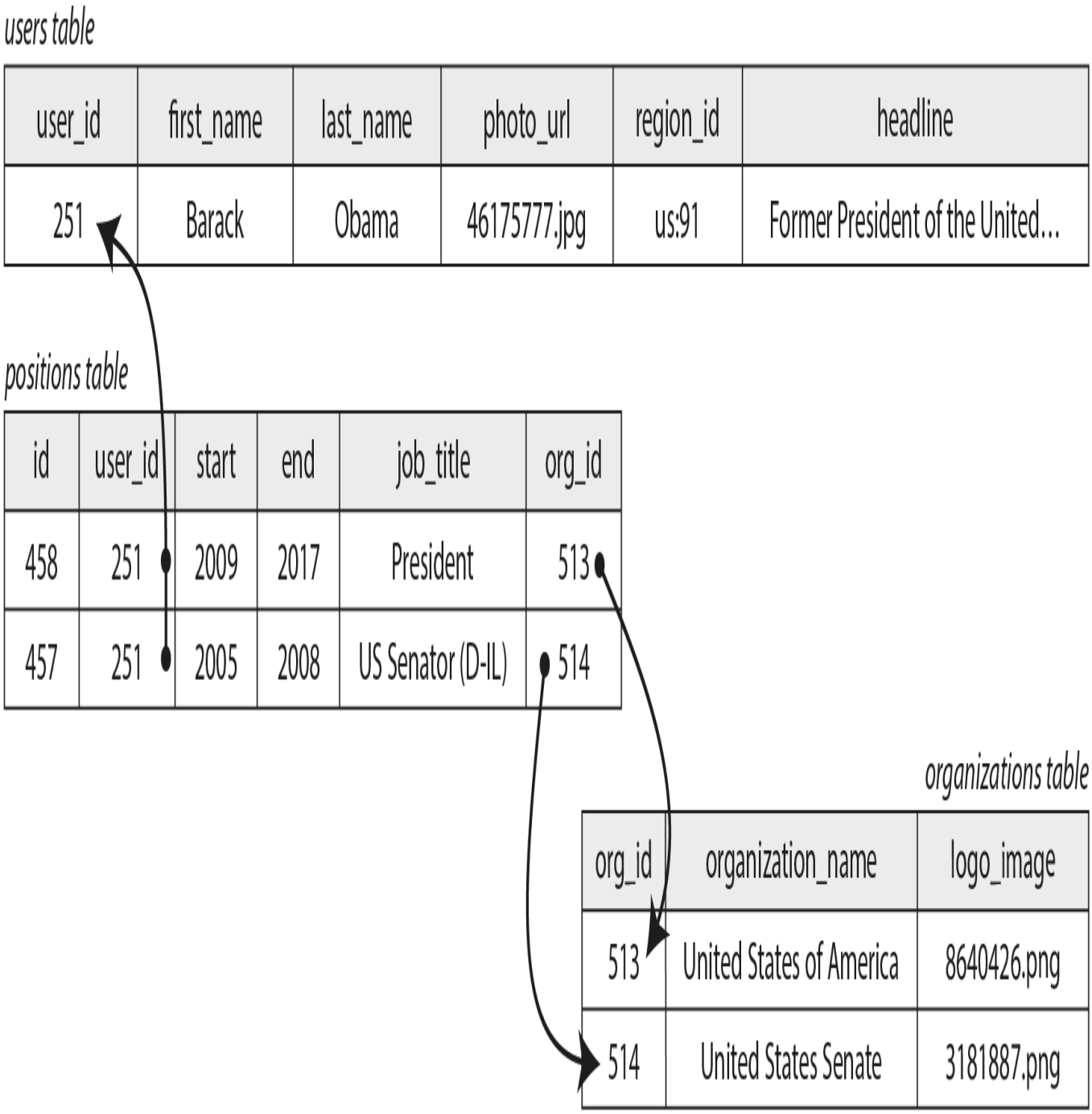


Figure 3-3. Many-to-many relationships in the relational model.

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": "President",
    {"start": 2005, "end": 2008, "job_title": "US Senator (D-
  ],
  ...
}
```

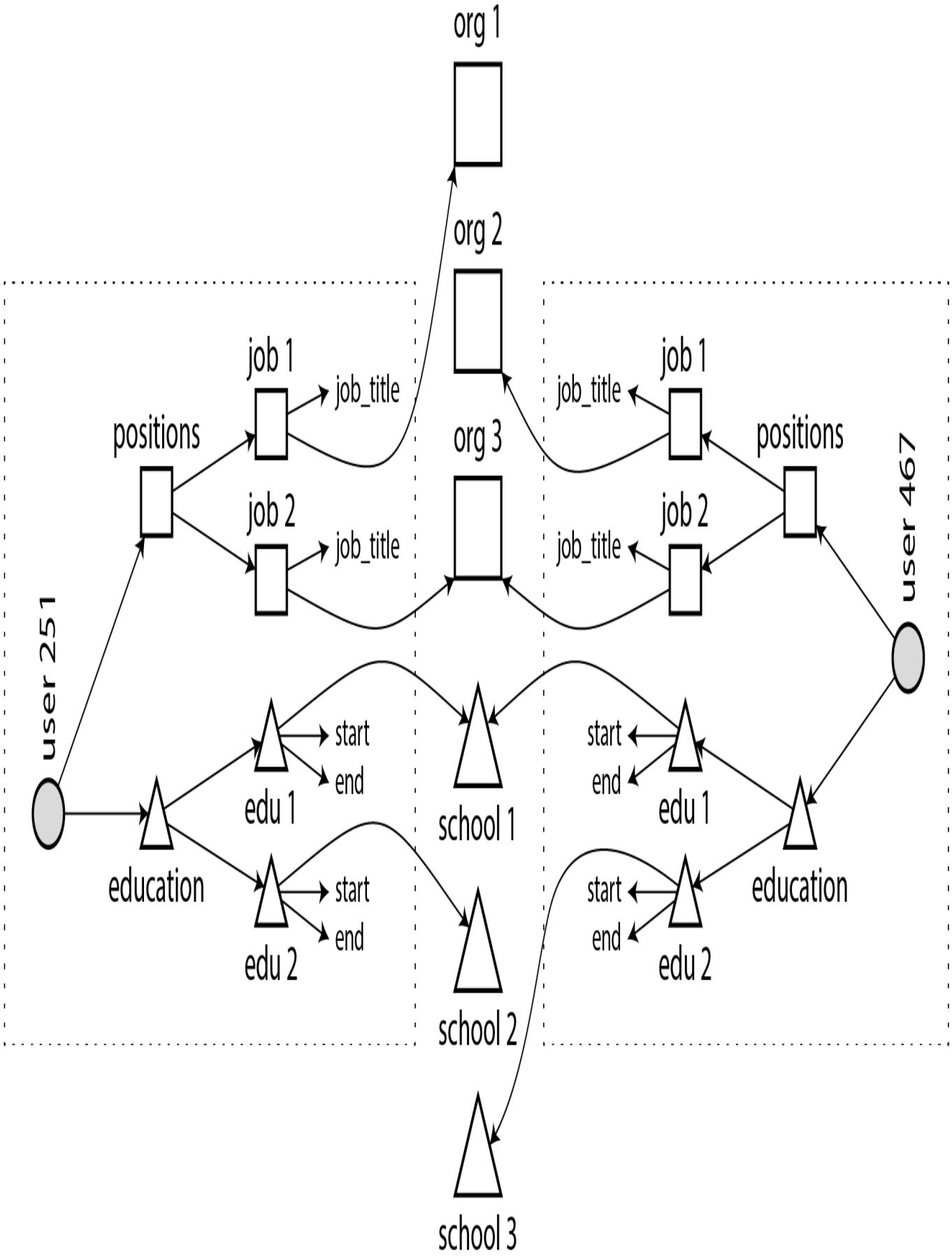

org 1
org 2
org 3
job 1
job_title
positions
job 2
job_title
user 251
start
end
edu 1
education
start
end
edu 2
school 1
school 2
school 3
job 1
job_title
positions
job 2
job_title
user 467
start
end
edu 1
education
start
end
edu 2

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 [Link to Come]) 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.

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

dim_product table

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

dim_store table

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

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
240102	31	3	NULL	NULL	1	2.49	2.49
240102	69	5	19	NULL	3	14.99	9.99
240102	74	3	23	191	1	4.49	3.89
240102	33	8	NULL	235	4	0.99	0.99

dim_date table

date_key	year	month	day	weekday	is_holiday
240101	2024	jan	1	mon	yes
240102	2024	jan	2	tue	no
240103	2024	jan	3	wed	no

dim_customer table

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

dim_promotion table

promotion_sk	name	ad_type	coupon_type
18	New Year sale	Poster	NULL
19	Aquarium deal	Newsletter	Discount code
20	Coffee & cake bundle	In-store sign	NULL

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 have first_
    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 DEFAULT NULL;
UPDATE users SET first_name = split_part(name, ' ', 1);
UPDATE users SET first_name = substring_index(name, ' ', 1);
```

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 [Link to Come].

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 *column-family* concept in the Bigtable data model (used in Cassandra, HBase, and ScyllaDB), also known as a *wide-column* model, has 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) AS observat
       sum(num_animals) AS total_animals
FROM observations
WHERE family = 'Sharks'
GROUP BY observation_month;
```

(1) 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. 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 [31].

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 [32].
- Knowledge graphs are used by search engines to record facts about entities that often occur in search queries, such as organizations, people, and places [33]. 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 [34], and others [35]) 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 [36], 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.

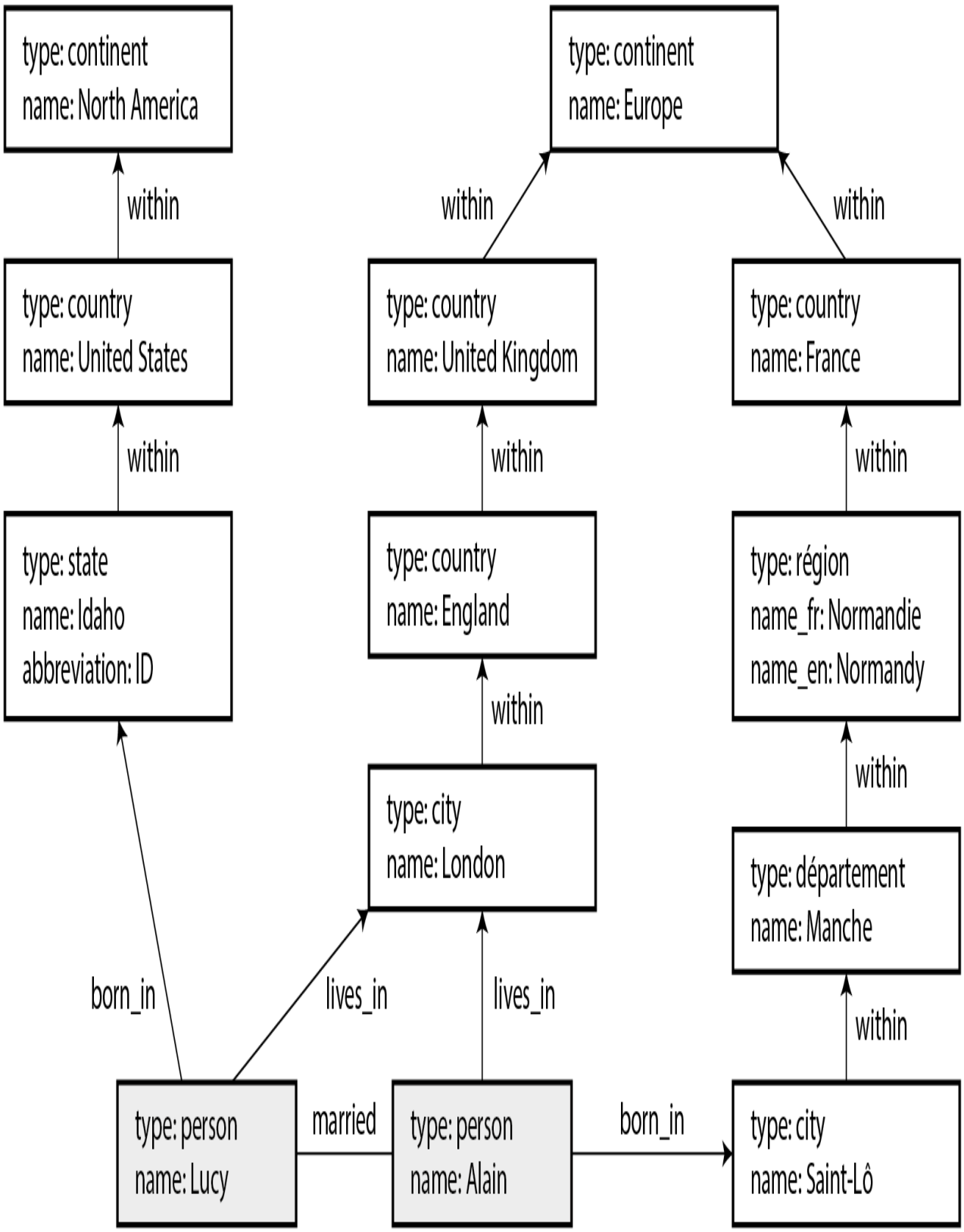

type: continent
name: North America
type: continent
name: Europe
within
within
within
type: country
name: United States
type: country
name: United Kingdom
type: country
name: France
within
within
within
type: state
name: Idaho
abbreviation: ID
type: country
name: England
type: région
name_fr: Normandie
name_en: Normandy
within
within
type: city
name: London
type: département
name: Manche
born_in
lives_in
lives_in
within
type: person
name: Lucy
married
type: person
name: Alain
born_in
type: city
name: Saint-Lô

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* [37]. Besides Neo4j, Cypher is supported by Memgraph, KùzuDB [34], 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 [38].

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',  type:'continent
  (usa      :Location {name:'United States',  type:'country'
  (idaho    :Location {name:'Idaho',          type:'state'
  (lucy     :Person   {name:'Lucy' }),
  (idaho) -[:WITHIN ]-> (usa)  -[:WITHIN]-> (namerica),
  (lucy)  -[:BORN_IN]-> (idaho)
```

When all the vertices and edges of Figure 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..]-> (:Location {nam
  (person) -[:LIVES_IN]-> () -[:WITHIN*0..]-> (:Location {nam
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 locations within
  in_usa(vertex_id) AS (
      SELECT vertex_id FROM vertices
        WHERE label = 'Location' AND properties->>'name' = 'U
    UNION
      SELECT edges.tail_vertex FROM edges ❷
        JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
        WHERE edges.label = 'within'
  ),

  -- in_europe is the set of vertex IDs of all locations with
  in_europe(vertex_id) AS (
      SELECT vertex_id FROM vertices
        WHERE label = 'location' AND properties->>'name' = 'E
```

```
    UNION
      SELECT edges.tail_vertex FROM edges
        JOIN in_europe ON edges.head_vertex = in_europe.verte
        WHERE edges.label = 'within'
  ),

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

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

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

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.

3. Do the same starting with the vertex whose `name` property has the value `"Europe"`, and build up the set of vertices `in_europe`.
4. 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 [39]. Oracle has a different SQL extension for recursive queries, which it calls *hierarchical* [40].

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 [41, 42], which will provide a syntax inspired by Cypher, GSQL [43], and PGQL [44].

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 [45]; Datomic uses 5-tuples, extending each triple with a transaction ID and a boolean to indicate deletion [46]. 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*) [47].

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";          :bornIn _:idah
_:idaho    a :Location; :name "Idaho";         :type "state";
_:usa      a :Location; :name "United States"; :type "country
_:namerica a :Location; :name "North America"; :type "contine
```

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 [48, 49], the legacy of the Semantic Web project lives on in a couple of specific technologies: *linked data* standards such as JSON-LD [50], *ontologies* used in biomedical science [51], Facebook's Open Graph protocol [52] (which is used for link unfurling [53]), 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) [54], 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-syntax-ns#">

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

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

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

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

The SPARQL query language

SPARQL is a query language for triple-stores using the RDF data model [55]. (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)   # Cyp

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

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 [35].

Datalog: Recursive Relational Queries

Datalog is a much older language than SPARQL or Cypher: it arose from academic research in the 1980s [56, 57, 58]. 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 [59] 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, PlaceNa
```

```
within_recursive(LocID, PlaceName) :- within(LocID, ViaID),
                                      within_recursive(ViaID,

migrated(PName, BornIn, LivingIn)  :- person(PersonID, PName)
                                      born_in(PersonID, BornI
                                      within_recursive(BornID
                                      lives_in(PersonID, Livi
                                      within_recursive(Living

us_to_europe(Person) :- migrated(Person, "United States", "Eu
/* 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.

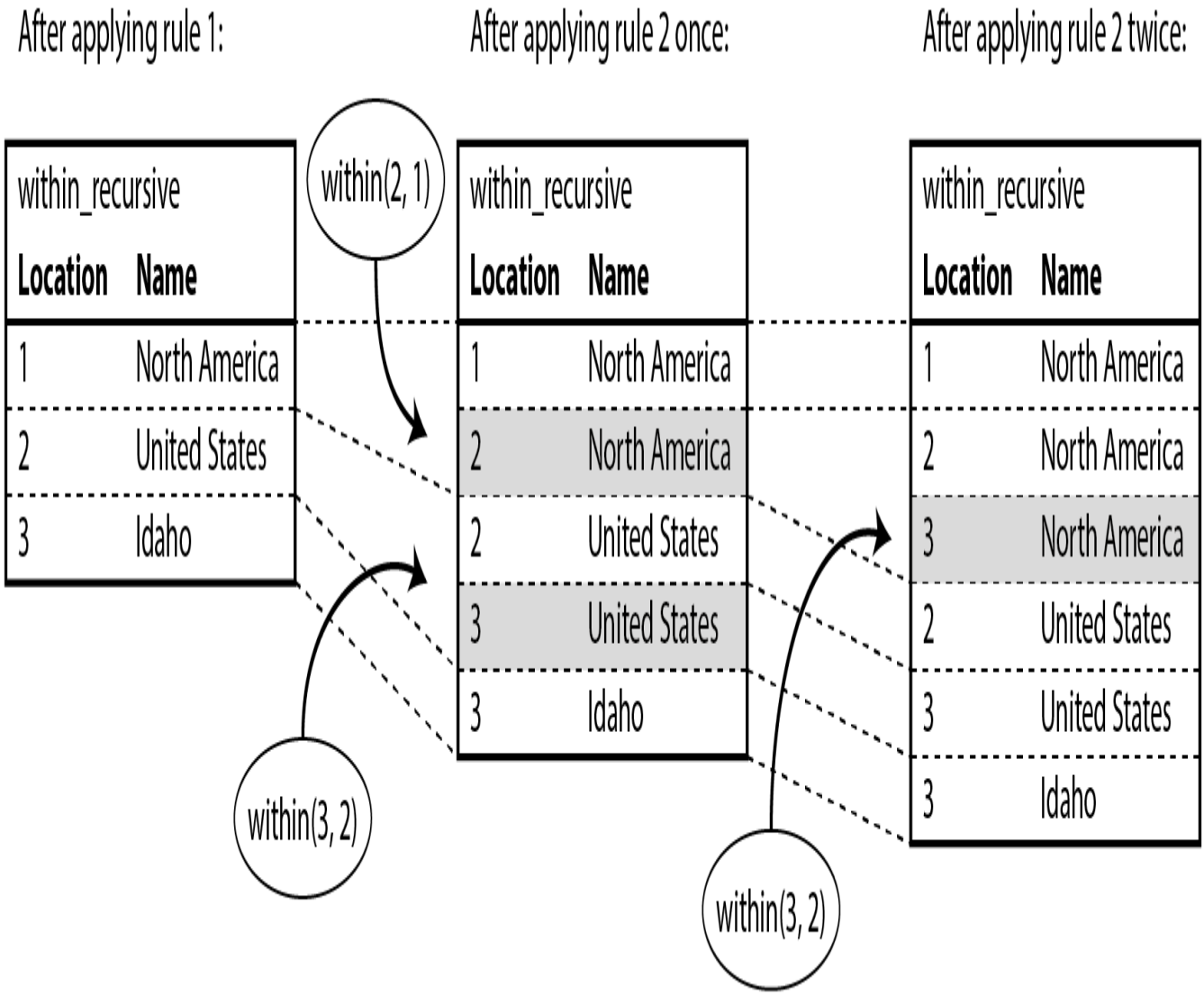


Figure 3-7. Determining that Idaho is in North America, using the Datalog rules from Example 3-12.

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 [Link to Come]). Authorization, rate limiting, and performance challenges are additional concerns [60]. 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": [
```

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

Event log

conference_created
timestamp: 2023-08-28 14:53:20
conf_id: 123
name: Data-Intensive App Developers

registrations_opened
timestamp: 2023-08-30 07:03:19
conf_id: 123

seats_reserved
timestamp: 2023-08-30 07:06:01
conf_id: 123
customer_id: 331
num_seats: 3
booking_id: 4001

booking_payment_confirmed
timestamp: 2023-08-30 07:10:42
conf_id: 123
customer_id: 331
paid_amount: 1497.00
paid_currency: USD

booking_cancelled
timestamp: 2023-08-31 21:52:35
conf_id: 123
customer_id: 345
booking_id: 4003

...

Derive multiple materialized views from the same event log

Materialized view:
Customer booking confirmation

```
{
  "booking_id": 4001,
  "conference_name": "Data-Intensive App ...",
  "paid_amount": 1497.0,
  "paid_currency": "USD",
  "unassigned_seats": 1,
  "assigned_seats": [
    {"badge_id": 501, "badge_name": "Aaliyah"},
    {"badge_id": 502, "badge_name": "Caleb"}
  ]
}
```

Materialized view:
Conference organizer dashboard

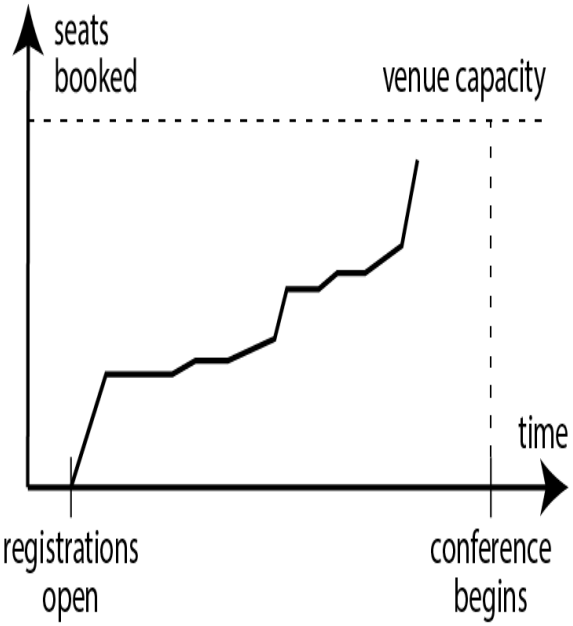


Figure 3-8. Using a log of immutable events as source of truth, and deriving materialized views from it.

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* [61, 62]. The principle of maintaining separate read-optimized representations and deriving them from the write-optimized representation is called *command query responsibility segregation (CQRS)* [63]. 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, wheras 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 "Transaction Processing versus Analytics"). 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.

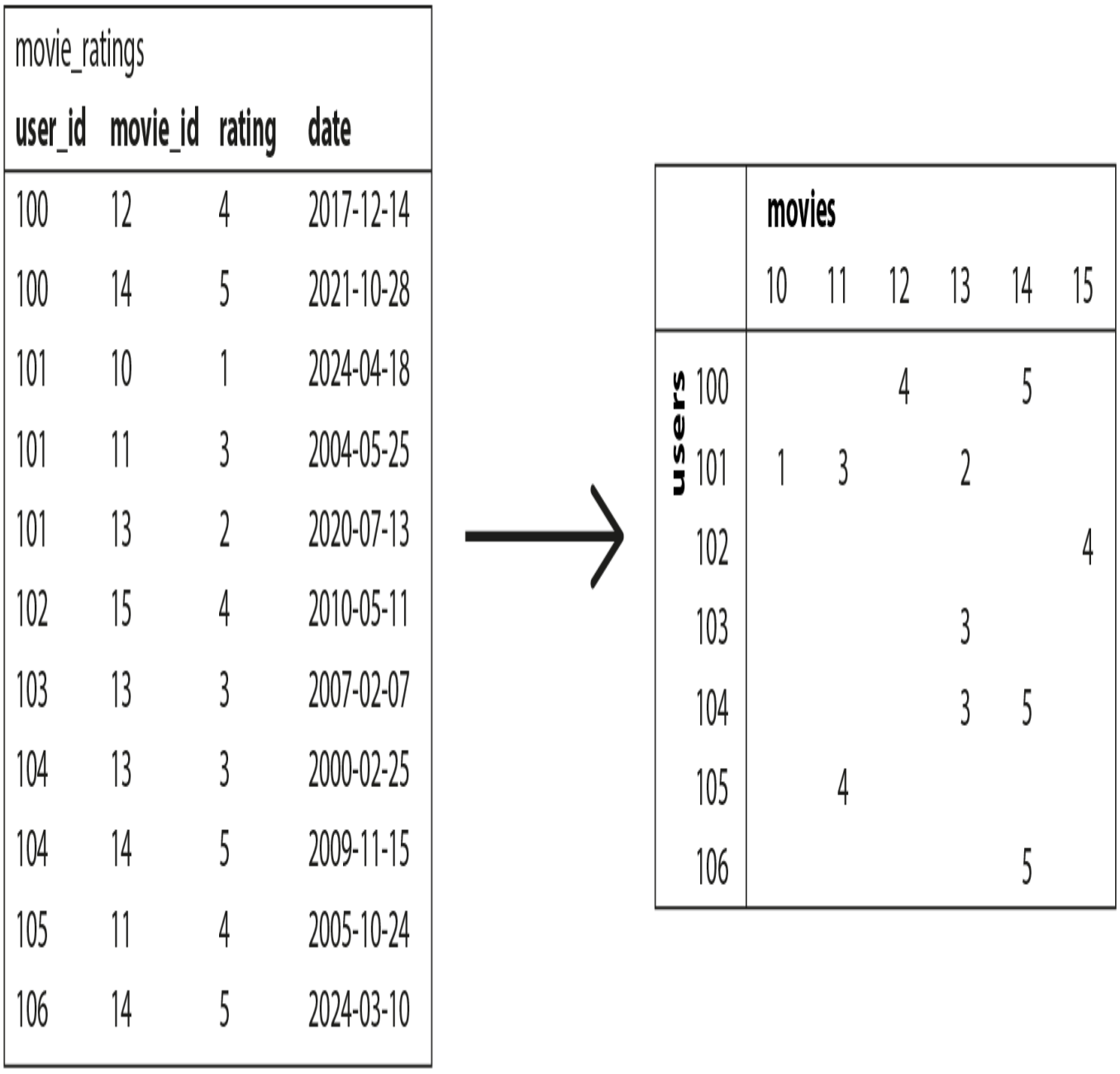
Dataframes are a data model supported by the R language, the Pandas library for Python, Apache Spark, ArcticDB, Dask, and other systems. They are a popular tool for data scientists preparing data for training machine learning models, but they are also widely used for data exploration, statistical data 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 [64]. 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.



movie_ratings

user_id	movie_id	rating	date
100	12	4	2017-12-14
100	14	5	2021-10-28
101	10	1	2024-04-18
101	11	3	2004-05-25
101	13	2	2020-07-13
102	15	4	2010-05-11
103	13	3	2007-02-07
104	13	3	2000-02-25
104	14	5	2009-11-15
105	11	4	2005-10-24
106	14	5	2024-03-10

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 [65] 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 [66]. Dataframes are also used in the financial industry for representing *time series data*, such as the prices of assets and trades over time [67].

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 [Link to Come]); 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 [68].
- 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 [Link to Come].

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

[1] Jamie Brandon. Unexplanations: query optimization works because sql is declarative. *scattered-thoughts.net*, February 2024. Archived at perma.cc/P6W2-WMFZ

[2] Joseph 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

[3] 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

[4] 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

[5] Markus Winand. Modern SQL: Beyond Relational. *modern-sql.com*, 2015. Archived at perma.cc/D63V-WAPN

[6] Martin Fowler. OrmHate. *martinfowler.com*, May 2012. Archived at perma.cc/VCM8-PKNG

[7] Vlad Mihalcea. N+1 query problem with JPA and Hibernate. *vladmihalcea.com*, January 2023. Archived at perma.cc/79EV-TZKB

[8] 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

[9] William Zola. 6 Rules of Thumb for MongoDB Schema Design. *mongodb.com*, June 2014. Archived at perma.cc/T2BZ-PPJB

[10] Sidney Andrews and Christopher McClister. Data modeling in Azure Cosmos DB. *learn.microsoft.com*, February 2023. Archived at archive.org

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

[12] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd edition. John Wiley & Sons, July 2013. ISBN: 9781118530801

[13] Michael Kaminsky. Data warehouse modeling: Star schema vs. OBT. *fivetran.com*, August 2022. Archived at perma.cc/2PZK-BFFP

[14] Joe Nelson. User-defined Order in SQL. *begriffs.com*, March 2018. Archived at perma.cc/GS3W-F7AD

[15] Evan Wallace. Realtime Editing of Ordered Sequences. *figma.com*, March 2017. Archived at perma.cc/K6ER-CQZW

[16] David Greenspan. Implementing Fractional Indexing. *observablehq.com*, October 2020. Archived at perma.cc/5N4R-MREN

[17] Martin Fowler. Schemaless Data Structures. *martinfowler.com*, January 2013.

[18] 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

[19] Martin Odersky. The Trouble with Types. At *Strange Loop*, September 2013. Archived at perma.cc/85QE-PVEP

[20] Conrad Irwin. MongoDB—Confessions of a PostgreSQL Lover. At *HTML5DevConf*, October 2013. Archived at perma.cc/C2J6-3AL5

[21] Percona Toolkit Documentation: pt-online-schema-change. *docs.percona.com*, 2023. Archived at perma.cc/9K8R-E5UH

[22] Shlomi Noach. gh-ost: GitHub's Online Schema Migration Tool for MySQL. *github.blog*, August 2016. Archived at perma.cc/7XAG-XB72

[23] Shayon Mukherjee. pg-osc: Zero downtime schema changes in PostgreSQL. *shayon.dev*, February 2022. Archived at perma.cc/35WN-7WMY

[24] Carlos Pérez-Aradros Herce. Introducing pgroll: zero-downtime, reversible, schema migrations for Postgres. *xata.io*, October 2023. Archived at archive.org

[25] 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.

[26] Donald K. Burleson. Reduce I/O with Oracle Cluster Tables. *dba-oracle.com*. Archived at perma.cc/7LBJ-9X2C

[27] 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.

[28] Priscilla Walmsley. *XQuery, 2nd Edition*. O'Reilly Media, December 2015. ISBN: 9781491915080

[29] Paul C. Bryan, Kris Zyp, and Mark Nottingham. JavaScript Object Notation (JSON) Pointer. RFC 6901, IETF, April 2013.

[30] Stefan Gössner, Glyn Normington, and Carsten Bormann. JSONPath: Query Expressions for JSON. RFC 9535, IETF, February 2024.

[31] 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

[32] 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.

[33] 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

[34] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. KÙZU Graph Database Management System. At *3th Annual Conference on Innovative Data Systems Research* (CIDR 2023), January 2023.

[35] 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.

[36] Apache TinkerPop 3.6.3 Documentation. *tinkerpop.apache.org*, May 2023. Archived at perma.cc/KM7W-7PAT

[37] 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

[38] Emil Eifrem. Twitter correspondence, January 2014. Archived at perma.cc/WM4S-BW64

[39] Francesco Tisiot. Explore the new SEARCH and CYCLE features in PostgreSQL® 14. *aiven.io*, December 2021. Archived at perma.cc/J6BT-83UZ

[40] Gaurav Goel. Understanding Hierarchies in Oracle. *towardsdatascience.com*, May 2020. Archived at perma.cc/5ZLR-Q7EW

[41] 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

[42] Alastair Green. SQL... and now GQL. *opencypher.org*, September 2019. Archived at perma.cc/AFB2-3SY7

[43] 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

[44] 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

[45] Amazon Web Services. Neptune Graph Data Model. Amazon Neptune User Guide, *docs.aws.amazon.com*. Archived at perma.cc/CX3T-EZU9

[46] Cognitect. Datomic Data Model. Datomic Cloud Documentation, *docs.datomic.com*. Archived at perma.cc/LGM9-LEUT

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

[48] Sinclair Target. Whatever Happened to the Semantic Web? *twobithistory.org*, May 2018. Archived at perma.cc/M8GL-9KHS

[49] Gavin Mendel-Gleason. The Semantic Web is Dead – Long Live the Semantic Web! *terminusdb.com*, August 2022. Archived at perma.cc/G2MZ-DSS3

[50] Manu Sporny. JSON-LD and Why I Hate the Semantic Web. *manu.sporny.org*, January 2014. Archived at perma.cc/7PT4-PJKF

[51] University of Michigan Library. Biomedical Ontologies and Controlled Vocabularies, *guides.lib.umich.edu/ontology*. Archived at perma.cc/Q5GA-F2N8

[52] Facebook. The Open Graph protocol, *ogp.me*. Archived at perma.cc/C49A-GUSY

[53] 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

[54] W3C RDF Working Group. Resource Description Framework (RDF). *w3.org*, February 2004.

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

[56] 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

[57] 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

[58] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN: 9780201537710, available online

at *webdam.inria.fr/Alice*

[59] 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

[60] Matt Bessey. Why, after 6 years, I'm over GraphQL. *bessey.dev*, May 2024. Archived at perma.cc/2PAU-JYRA

[61] 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

[62] Greg Young. CQRS and Event Sourcing. At *Code on the Beach*, August 2014.

[63] Greg Young. CQRS Documents. *cqrs.wordpress.com*, November 2010. Archived at perma.cc/X5R6-R47F

[64] 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

[65] 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

[66] 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

[67] Ed Targett. Bloomberg, Man Group team up to develop open source "ArcticDB" database. *thestack.technology*, March 2023. Archived at perma.cc/M5YD-QQYV

[68] 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

About the Author

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*.