O'REILLY®

Compliments of pentaho®

# Designing Data-Intensive Applications

THE BIG IDEAS BEHIND RELIABLE, SCALABLE, AND MAINTAINABLE SYSTEMS

PREVIEW EDITION

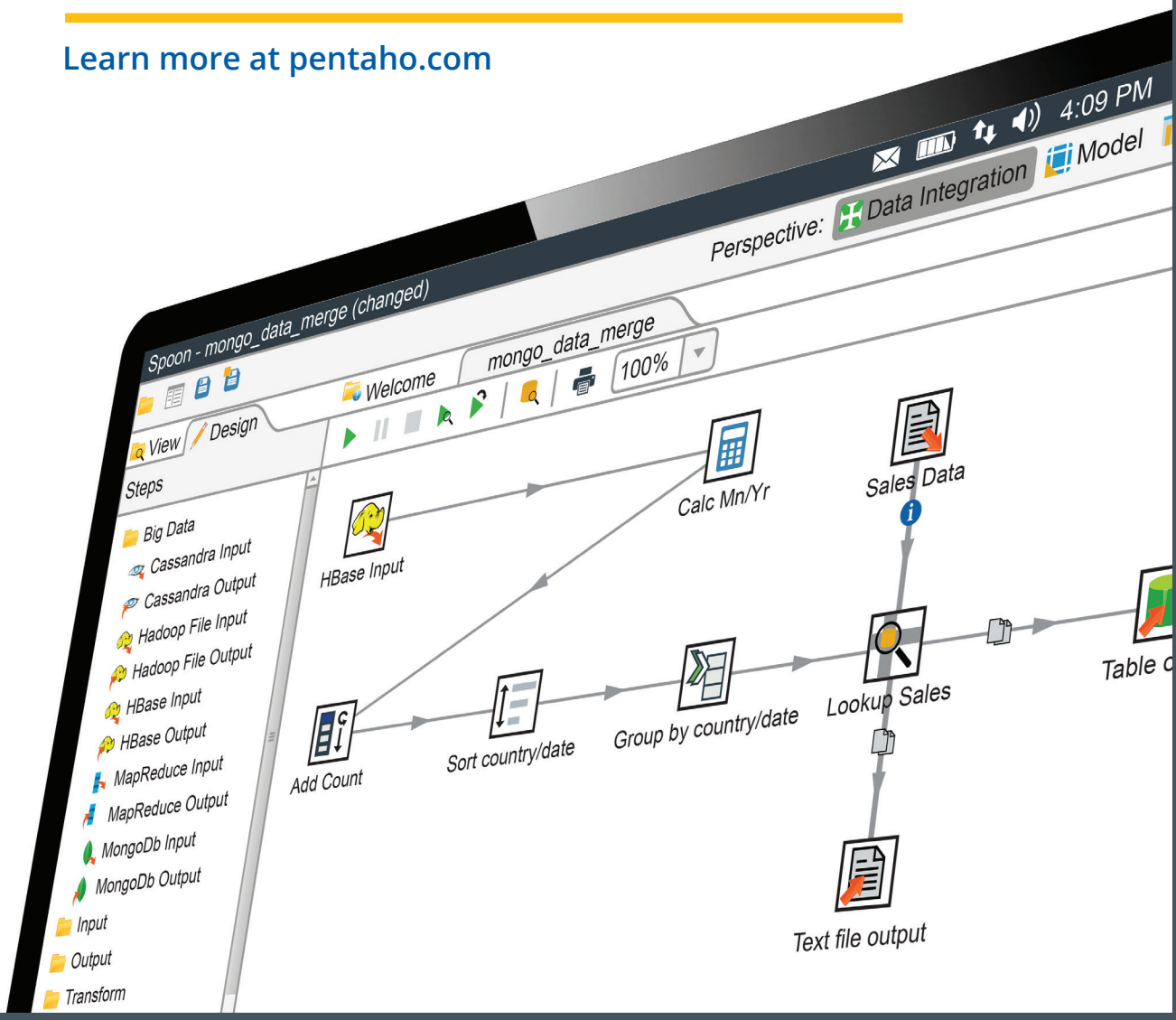
Martin Kleppmann

pentaho

# Big Data Integration with Zero Coding

Need native and flexible support for all big data sources?

**Learn more at pentaho.com**



This Preview Edition of *Designing Data-Intensive Applications*, *Chapters 1 and 2*, is a work in progress. The final book is currently scheduled for release in July 2015 and will be available at *oreilly.com* and other retailers once it is published.

# Designing Data-Intensive Applications

*Martin Kleppmann*

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

**Designing Data-Intensive Applications**

by Martin Kleppmann



Printed in the United States of America.

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

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

**Editors:** Ann Spencer and Marie Beaugureau
**Proofreader:** FIX ME!
**Cover Designer:** Karen Montgomery
**Interior Designer:** David Futato
**Illustrator:** Rebecca Demarest

July 2015: First Edition

**Revision History for the First Edition:**

2015-01-08: Pentaho release

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



ISBN: 978-1-449-37332-0

[?]

# Table of Contents

## Part I. Foundations of Data Systems

# PART I
# Foundations of Data Systems

This book is arranged into three parts:

1. In this Part I, we will discuss the fundamental ideas that we need in order to design data-intensive applications. We'll start in Chapter 1 by discussing what we're actually trying to achieve: reliability, scalability and maintainability — how we need to think about them, and how we can achieve them. In Chapter 2 we will compare several different data models and query languages, and see how they are appropriate to different situations. In (to come) we will talk about storage engines: how databases arrange data on disk so that you can find it again efficiently.
2. In (to come), we will move from data stored on one machine to data that is distributed across multiple machines. This is often necessary for scalability, but brings with it a variety of unique challenges. We'll discuss replication ((to come)), partitioning/sharding ((to come)), transactions ((to come)) and the problems of distributed consensus and consistency ((to come)).
3. In (to come), we move up another step and discuss building systems that consist of several different components. Each component may be distributed, e.g. a distributed database, a multi-node search index or a cluster of caches. However, now we have the additional challenge of integrating those various components.

# CHAPTER 1
# Reliable, Scalable and Maintainable Applications

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

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

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

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

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

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

In this Chapter 1, we will start by exploring the fundamentals of what we are trying to achieve: reliabile, scalabile and maintainable data systems. We'll clarify what those things mean, outline some ways of thinking about them, and go over the basics that we will need for later chapters. In the following chapters we will continue layer by layer, looking at different design decisions which need to be considered when working on a data-intensive application.

## Thinking About Data Systems

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

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

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

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

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

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

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

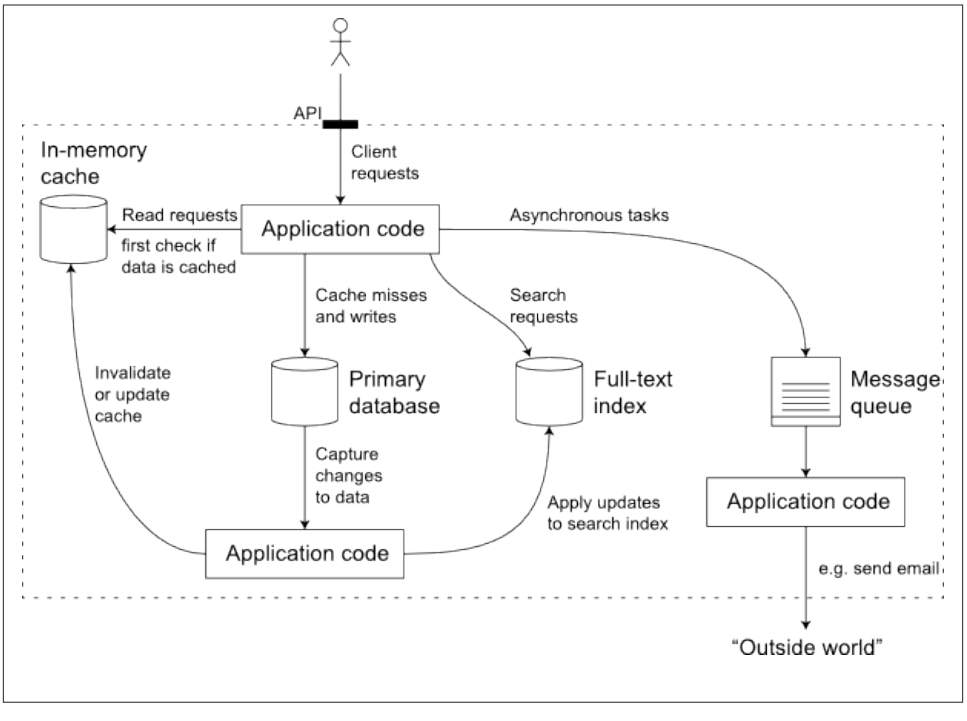



*Figure 1-1. One possible architecture for a data system that combines several components.*

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

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

*Reliability*

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

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

*Maintainability*
: Over time, many different people will work on the system (engineering and operations, both maintaining current behavior and adapting the system to new use cases), and they should all be able to work on it *productively*. See "Maintainability" on page 17.

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

# Reliability

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

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

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

The things that can go wrong are called *faults*, and systems that anticipate faults and can cope with them are called *fault-tolerant*. The term is slightly misleading: it suggests that we could make a system tolerant of every possible kind of fault, which in reality is not feasible. If the entire planet Earth (and all servers on it) were swallowed by a black hole, tolerance of that fault would require web hosting in space — good luck getting that budget item approved. So it only makes sense to talk about tolerance of certain *types of fault*.

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

Counter-intuitively, in such fault-tolerant systems, it can make sense to *increase* the rate of faults by triggering them deliberately. Software that deliberately causes faults — for example, randomly killing individual processes without warning — is known as a *chaos monkey* [3]. It ensures that the fault-tolerance machinery is continually exercised and tested, so that we can be confident that faults will be handled correctly when they occur naturally.

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

## Hardware faults

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

Hard disks are reported as having a mean time to failure (MTTF) of about 10 to 50 years [4, 5]. Thus, on a storage cluster with 10,000 disks, we should expect on average one disk to die per day.

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

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

However, as data volumes and applications' computing demands increase, more applications are using larger numbers of machines, which proportionally increases the rate of hardware faults. Moreover, in some "cloud" platforms such as Amazon Web Services it is fairly common for virtual machine instances to become unavailable without warn-

ing [6], as the platform is designed for flexibility and elasticity in favor of single-machine reliability.

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

## Software errors

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

Another class of fault is a systematic error within the system. Such faults are harder to anticipate, and because they are correlated across nodes, they tend to cause many more system failures than uncorrelated hardware faults [4]. Examples include:

- A software bug that causes every instance of an application server to crash when given a particular bad input. For example, consider the leap second on June 30, 2012 that caused many applications to hang simultaneously, due to a bug in the Linux kernel. [7]
- A runaway process uses up some shared resource — CPU time, memory, disk space or network bandwidth.
- A service that the system depends on slows down, becomes unresponsive or starts returning corrupted responses.
- Cascading failures, where a small fault in one component triggers a fault in another component, which in turn triggers further faults (see [8] for example).

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

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, measuring, monitoring and analyzing system behavior in production. If a system is expected to provide some guarantee (for example, in a message queue,

that the number of incoming messages equals the number of outgoing messages), it can constantly check itself while it is running, and raise an alert if a discrepancy is found [9].

## Human errors

Humans design and build software systems, and the operators who keep the system running are also human. Even when they have the best intentions, humans are known to be unreliable. How do we make our system reliable, in spite of unreliable humans?

The best systems combine several approaches:

- Design systems in a way that minimizes opportunities for error. For example, well-designed abstractions, APIs and admin interfaces make it easy to do "the right thing", and discourage "the wrong thing". However, if the interfaces are too restrictive, people will work around them, negating their benefit, so this is a tricky balance to get right.
- Decouple the places where people make the most mistakes from the places where they can cause failures. In particular, provide fully-featured non-production *sandbox* environments where people can explore and experiment safely, using real data, without affecting real users.
- Test thoroughly at all levels, from unit tests to whole-system integration tests and manual tests. Automated testing is widely used, well understood, and especially valuable for covering corner cases that rarely arise in normal operation.
- Allow quick and easy recovery from human errors, to minimize the impact in the case of a failure. For example, make it fast to roll back configuration changes, roll out new code gradually (so that any unexpected bugs affect only a small subset of users), and provide tools to recompute data (in case it turns out that the old computation was incorrect).
- Set up detailed and clear monitoring, such as performance metrics and error rates. In other engineering disciplines this is referred to as *telemetry*. (Once a rocket has left the ground, telemetry is essential for tracking what is happening, and for understanding failures. [10]) Monitoring can show us early warning signals, and allow us to check whether any assumptions or constraints are being violated. When a problem occurs, metrics can be invaluable in diagnosing the issue.
- Good management practices and training — a complex and important aspect, and beyond the scope of this book.

## How important is reliability?

Reliability is not just for nuclear power stations and air traffic control software — more mundane applications are also expected to work reliably. Bugs in business applications

cause lost productivity (and legal risks if figures are reported incorrectly), and outages of e-commerce sites can have huge costs in terms of lost revenue and reputation.

Even in "non-critical" applications we have a responsibility to our users. Consider a parent who stores all pictures and videos of their children in your photo application [11]. How would they feel if that database was suddenly corrupted? Would they know how to restore it from a backup?

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

# Scalability

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

*Scalability* is the term we use to describe a system's ability to cope with increased load. Note, however, that it is not a one-dimensional label that we can attach to a system: it is meaningless to say "X is scalable" or "Y doesn't scale". Rather, discussing scalability means to discuss the question: if the system grows in a particular way, what are our options for coping with the growth? How can we add computing resources to handle the additional load?

## Describing load

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

To make this idea more concrete, let's consider Twitter as an example, using data published in November 2012 [12]. Two of Twitter's main operations are:

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

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

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

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

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

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

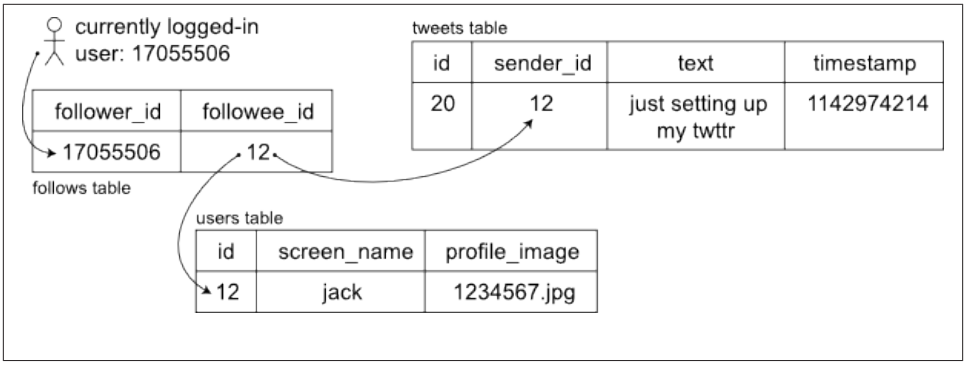



*Figure 1-2. Simple relational schema for implementing a Twitter home timeline*

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

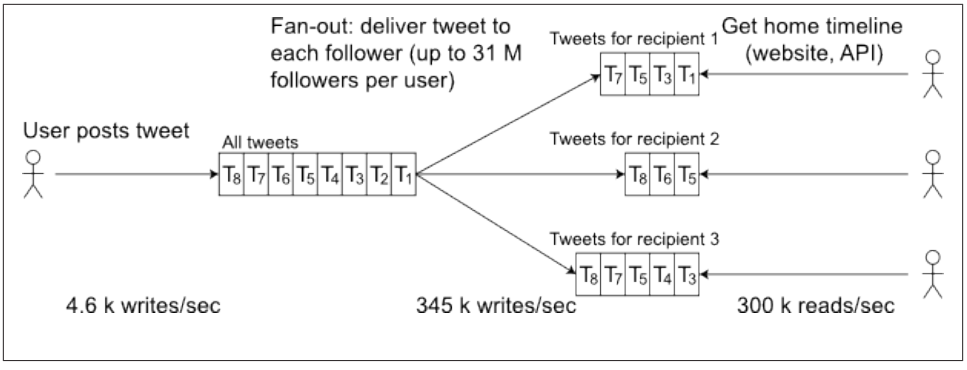



*Figure 1-3. Twitter's data pipeline for delivering tweets to followers, with load parameters as of November 2012 [12]*

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

However, the downside of approach 2 is that posting a tweet now requires a lot of extra work. On average, a tweet is delivered to about 75 followers, so 4.6 k tweets per second become 345 k writes per second to the home timeline caches. But this average hides the fact that the number of followers per user varies wildly, and some users have over 30 million followers. This means that a single tweet may result in over 30 million writes to home timelines! Doing this in a timely manner — Twitter tries to deliver tweets to followers within 5 seconds — is a significant challenge.

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

The final twist of the Twitter anecdote: now that approach 2 is robustly implemented, Twitter is moving to a hybrid of both approaches. Most users' tweets continue to be fanned out to home timelines at the time when they are posted, but a small number of users with a very large number of followers are excepted from this fan-out. Instead, when the home timeline is read, the tweets from celebrities followed by the user are fetched separately and merged with the home timeline when the timeline is read, like in approach 1. This hybrid approach is able to deliver consistently good performance.

## Describing performance

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

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

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

In a batch-processing system such as Hadoop, we usually care about *throughput* — the number of records we can process per second, or the total time it takes to run a job on a dataset of a certain size.[ii] In online systems, *latency* is usually more important — the time it takes to serve a request, also known as *response time*.

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

In Figure 1-4, each gray bar represents a request to a service, and its height shows how long that request took. Most requests are reasonably fast, but there are occasional *outliers* that take much longer. Perhaps the slow requests are intrinsically more expensive, e.g. because they process more data. But even in a scenario where you'd think all requests should take the same time, you get variation: random additional latency could be introduced by a context switch to a background process, the loss of a network packet and TCP retransmission, a garbage collection pause, a page fault forcing a read from disk, mechanical vibrations in the server rack [13], or many other things.

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

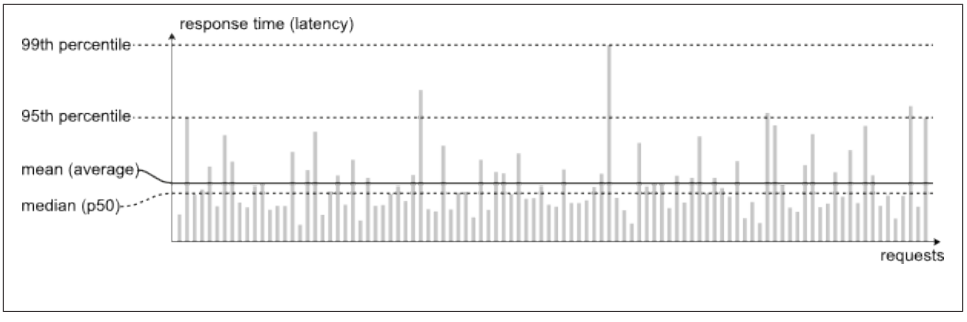



*Figure 1-4. Illustrating mean and percentiles: response times for a sample of 100 requests to a service*

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

Usually it is better to use *percentiles*. If you take your list of response times and sort it, from fastest to slowest, then the *median* is the half-way point: for example, if your median response time is 200 ms, that means half your requests return in less than 200 ms, and half your requests take longer than that.

This makes the median a good metric if you want to know how long users typically have to wait: half of user requests are served in less than the median latency, and the other half take longer than the median. The median is also known as *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* percentile are common (abbreviated *p95*, *p99* and *p999*). They are the response time thresholds at which 95%, 99% or 99.9% of requests are faster than that particular threshold. For example, if the 95th percentile response time is 1.5 seconds, that means 95 out of 100 requests take less than 1.5 seconds, and 5 out of 100 requests take 1.5 seconds or more. This is illustrated in Figure 1-4.

High percentiles are important because they directly affect users' experience of the service. For example, Amazon describes latency requirements for internal services in terms of the 99.9th percentile, even though it only affects 1 in 1,000 requests. This is because the customers with the slowest requests are often those who have made many purchases — i.e. the most valuable customers. [14] It's important to keep those customers happy by ensuring the website is fast for them: Amazon has also observed that a 100 ms increase in latency reduces sales by 1% [15] and others report that a 1-second slowdown reduces a customer satisfaction metry by 16%. [16]

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

In summary, when measuring performance, it's worth using percentiles rather than averages. The main advantage of the mean is that it's easy to calculate, but percentiles are much more meaningful.

### Percentiles in Practice

High percentiles become especially important in backend services that are called multiple times as part of serving a single end-user request. Even if you make the calls in parallel, the end-user request still needs to wait for the slowest of the parallel calls to complete. As it takes just one slow call to make the entire end-user request slow, rare slow calls to the backend become much more frequent at the end-user request level (Figure 1-5). See [17] for a discussion of approaches to solving this problem.

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

The naïve implementation is to keep a list of response times for all requests within the time window, and to sort that list every minute. If that is too inefficient for you, there are algorithms which give a good approximation of percentiles at minimal CPU and memory cost, such as forward decay [18], which has been implemented in Java [19] and Ruby [20].

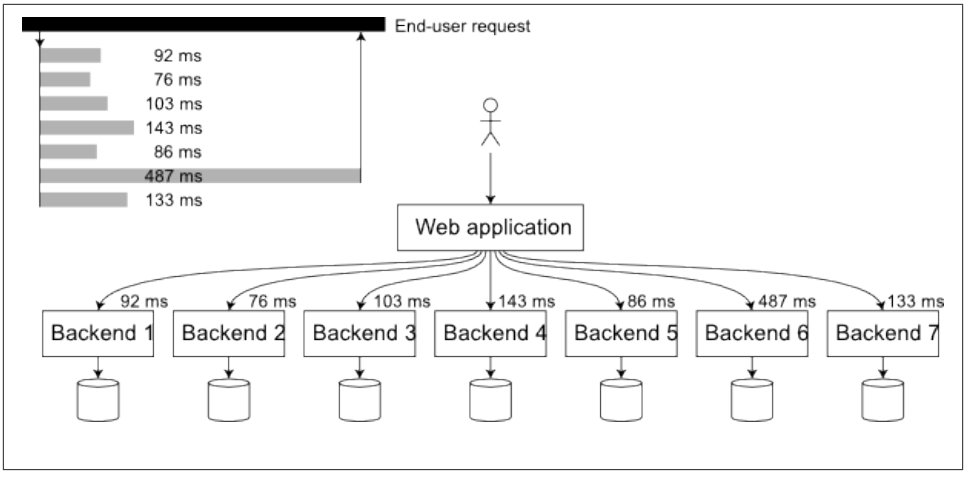



*Figure 1-5. When several backend calls are needed to serve a request, it takes just a single slow backend request to slow down the entire end-user request.*

## Approaches for coping with load

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

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

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

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

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

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

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

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

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

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

## Maintainability

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

Yet, unfortunately, many people working on software systems dislike maintenance of so-called *legacy* systems — perhaps it involves fixing other people's mistakes, or working with platforms that are now outdated, or systems that were forced to do things they

were never intended for. Every legacy system is unpleasant in its own way, and so it is difficult to give general recommendations for dealing with them.

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

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

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

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

As previously with reliability and scalability, there are no quick answers to achieving these goals. Rather, we will try to think about systems with operability, simplicity and plasticity in mind.

## Operability: making life easy for operations

It has been suggested that *"good operations can often work around the limitations of bad (or incomplete) software, but good software cannot run reliably with bad operations"*. [9] While some aspects of operations can and should be automated, it is still up to humans to set up that automation in the first place, and to make sure it's working correctly.

Operations teams are vital to keeping a software system running smoothly. A good operations team typically does the following, and more:

- monitoring the health of the system, and quickly restoring service if it goes into a bad state;
- tracking down the cause of problems, such as system failures or degraded performance;
- keeping software and platforms up-to-date, including security patches;
- keeping tabs on how different systems affect each other, so that a problematic change can be avoided before it causes damage;
- anticipating future problems and solving them before they occur, e.g. capacity planning;

- establishing good practices and tools for deployment, configuration management and more;
- performing complex maintenance tasks, such as moving an application from one platform to another;
- maintaining security of the system as configuration changes are made;
- defining processes that make operations predictable and help keep the production environment stable;
- preserving the organization's knowledge about the system, even as individual people come and go.

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

- provide visibility into the runtime behavior and internals of the system, with good monitoring;
- good support for automation and integration with standard tools;
- good documentation and an easy-to-understand operational model ("if I do X, Y will happen");
- good default behavior, but also giving administrators the freedom to override defaults when needed;
- self-healing where appropriate, but also giving administrators manual control over the system state when needed;
- predictable behavior, minimizing surprises.

## Simplicity: managing complexity

Small software projects can have delightfully simple and expressive code, but as projects get larger, they often become very complex and difficult to understand. This complexity slows down everyone who needs to work on the system, further increasing the cost of maintenance.

There are many possible symptoms of complexity: explosion of the state space, tight coupling of modules, tangled dependencies, inconsistent naming and terminology, hacks aimed at solving performance problems, special-casing to work around issues elsewhere, and many more.

Much has been written on this topic already — to mention just two articles, *No Silver Bullet* [21] is a classic, and its ideas are further developed in *Out of the Tar Pit* [22].

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

Making a system simpler does not necessarily mean reducing its functionality; it can also mean removing *accidental* complexity. Moseley and Marks [22] define complexity as *accidental* if it is not inherent in the problem that the software solves (as seen by the users), but arises only from the implementation.

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

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

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

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

## Plasticity: making change easy

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

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

Most discussions of these agile techniques focus on a fairly small, local scale (a couple of source code files within the same application). In this book, we search for ways of increasing agility on the level of a larger data system, perhaps consisting of several different applications or services with different characteristics. For example, how would you "refactor" Twitter's architecture for assembling home timelines ("Describing load" on page 10) from approach 1 to approach 2?

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

## Summary

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

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

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

*Scalability* means having strategies for keeping performance good, even when load increases. In order to discuss scalability, we first need ways of describing load and performance quantitatively. We briefly looked at Twitter's home timelines as an example of describing load, and latency percentiles as a way of measuring performance. In a scalable system, you can add processing capacity in order to remain reliable under high load.

*Maintainability* has many facets, but in essence it's about making life better for the engineering and operations teams who need to work with the system. Good abstractions can help reduce complexity and make the system easier to modify and adapt for new

iii. A term borrowed from materials science, where it refers to the ease with which an object's shape can be changed by pushing or pulling forces, without fracture. It is also used in neuroscience, referring to the brain's ability to adapt to environmental changes.

use cases. Good operability means having good visibility into the system's health, and having effective ways of managing it.

There is unfortunately no quick answer to making applications reliable, scalable or maintainable. However, there are certain patterns and techniques which keep reappearing in various different kinds of application. In the next few chapters we will take a look at some examples of data systems, and analyze how they work towards those goals.

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

## References


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

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

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

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

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

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

[7] Nelson Minar: "Leap Second crashes half the internet," somebits.com, 3 July 2012.

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

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

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

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

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

[13] Kelly Sommers: "After all that run around, what caused 500ms disk latency even when we replaced physical server? Slight unbalance in rack causing vibration.," twitter.com, 13 November 2014.

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

[15] Greg Linden: "Make Data Useful," at *Stanford University Data Mining Class* (CS345), December 2006.

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

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

[18] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu: "Forward Decay: A Practical Time Decay Model for Streaming Systems," at *25th IEEE International Conference on Data Engineering* (ICDE), pages 138–149, March 2009.

[19] Coda Hale: "Metrics," Apache License Version 2.0. http://metrics.codahale.com

[20] Eric Lindvall: "Metriks Client," MIT License. *https://github.com/eric/metriks*

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

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

# CHAPTER 2
# The Battle of the Data Models

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

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

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

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

In a complex application there may be more intermediary levels, such as APIs built upon APIs, but the basic idea is still the same: each layer hides the complexity of the layers

below it by providing a clean data model. These abstractions allow different groups of people — for example, the engineers at the database vendor and the application developers using their database — to work together effectively.

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

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

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

## Relational Model vs. Document Model

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

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

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

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

Over the years, there have been many competing approaches to data storage and querying. In the 1970s and early 1980s, CODASYL (the network model) and IMS (the hierarchical model) were the main alternatives, but the relational model came to dominate them. Object databases came and went again in the late 1980s and early 1990s. XML databases appeared in the early 2000s, but have only seen niche adoption. Each competitor to the relational model generated a lot of hype in its time, but it never lasted. [2]

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

## The birth of NoSQL

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

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

- A need for greater scalability than relational databases can easily achieve, including very large datasets or very high write throughput;
- Specialized query operations that are not well supported by the relational model;
- Frustration with the restrictiveness of relational schemas, and a desire for a more dynamic and expressive data model. [4]

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

## The object-relational mismatch

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

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

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

For example, Figure 2-1 illustrates how a résumé (a LinkedIn profile) could be expressed in a relational schema. The profile as a whole can be identified by a unique identifier, `user_id`. Fields like `first_name` and `last_name` appear exactly once per user, so they can be modeled as columns on the `users` table.

However, most people have had more than one job in their career (positions), varying numbers of periods of education, and any number of pieces of contact information. There is a one-to-many relationship from the user to these items, which can be represented in various ways:

- In the traditional SQL model (prior to SQL:1999), the most common normalized representation is to put `positions`, `education` and `contact_info` in separate tables, with a foreign key reference to the `users` table, as in Figure 2-1.
- Later versions of the SQL standard added support for structured datatypes and XML data, which allow multi-valued data to be stored within a single row, with support for querying and indexing inside those documents. These features are supported to varying degrees by Oracle, IBM DB2, MS SQL Server and PostgreSQL. [5, 6] PostgreSQL also has vendor-specific extensions for JSON, array datatypes, and more. [7]
- Encode jobs, education and contact info as a JSON or XML document, store it on a text column in the database, and to let the application interpret its structure and content. In this setup, you typically cannot use the database to query for values inside that encoded column.

For a data structure like a résumé, which is mostly a self-contained *document*, a JSON representation can be quite appropriate: see Example 2-1. JSON has the appeal of being much simpler than XML. Document-oriented databases like MongoDB [8], RethinkDB [9], CouchDB [10] and Espresso [11] support this data model, and many developers feel that the JSON model reduces the impedance mismatch between the application code and the storage layer.

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

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

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

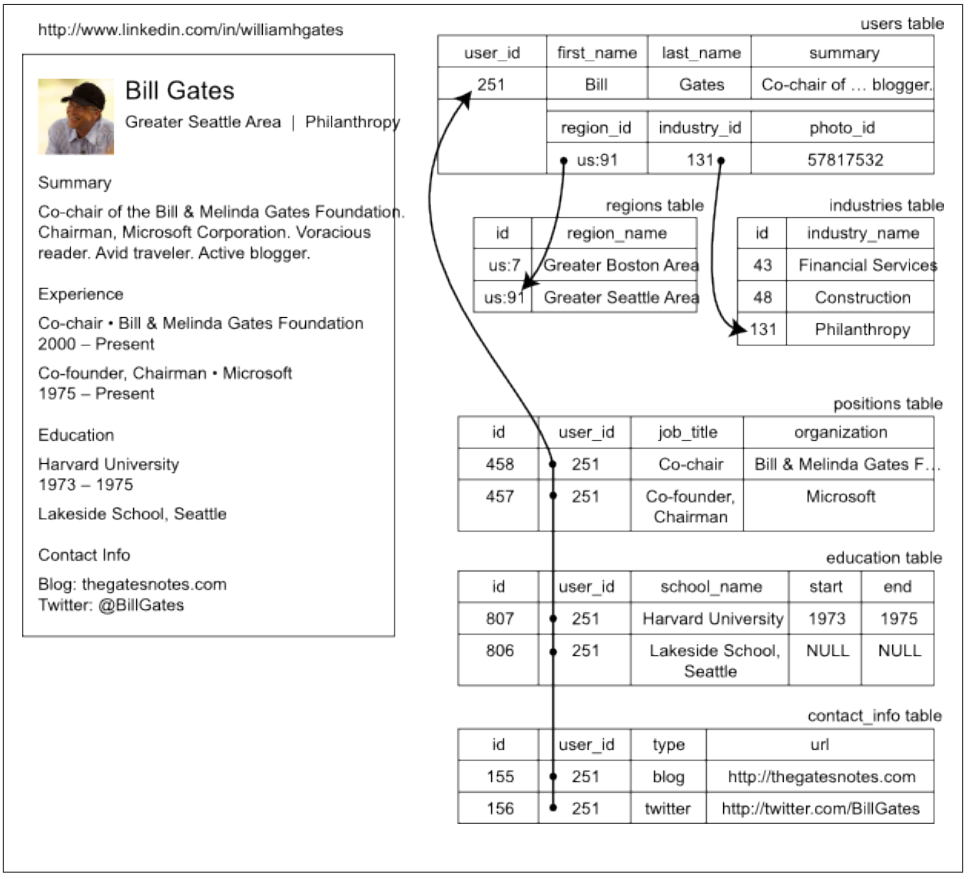



*Figure 2-1. Representing a LinkedIn profile using a relational schema.*

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

```
{
  "user_id":     251,
  "first_name":  "Bill",
```

```
  "last_name":   "Gates",
  "summary":     "Co-chair of the Bill & Melinda Gates... Active blogger.",
  "region_id":   "us:91",
  "industry_id": 131,
  "photo_url":   "/p/7/000/253/05b/308dd6e.jpg",
  "positions": [
    {"job_title": "Co-chair", "organization": "Bill & Melinda Gates Foundation"},
    {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}
  ],
  "education": [
    {"school_name": "Harvard University",       "start": 1973, "end": 1975},
    {"school_name": "Lakeside School, Seattle", "start": null, "end": null}
  ],
  "contact_info": {
    "blog":    "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}
```

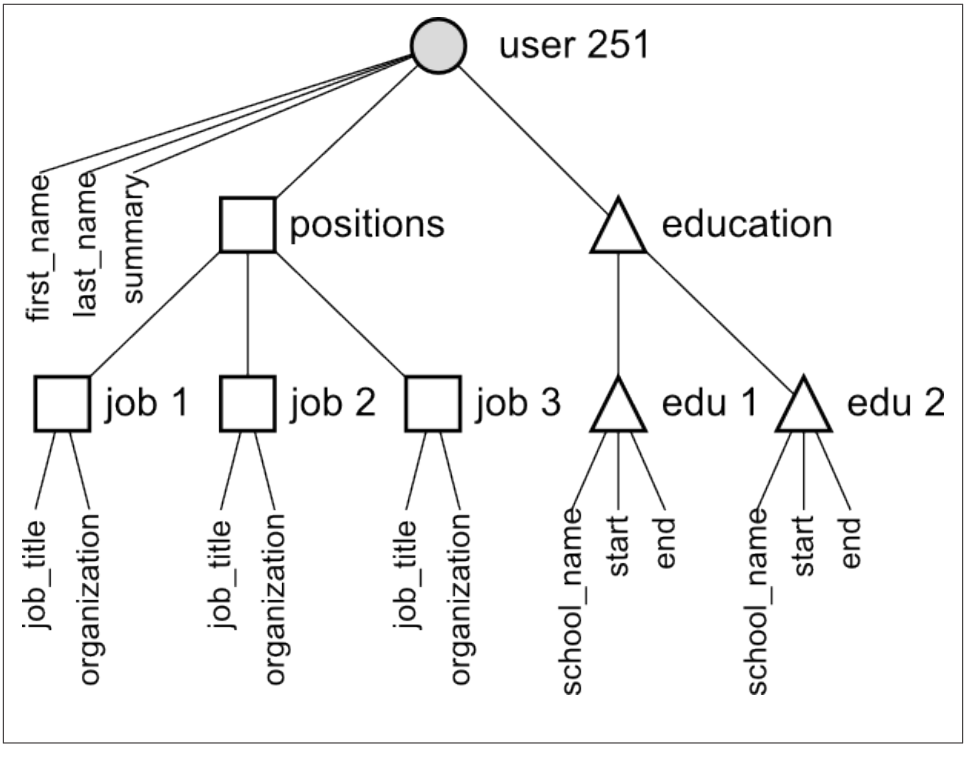



*Figure 2-2. One-to-many relationships forming a tree structure.*

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

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

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

- Consistent style and spelling across profiles,
- Avoiding ambiguity, e.g. if there are several cities with the same name,
- The name is stored only in one place, so it is easy to update across the board if it ever needs to be changed (for example, change of a city name due to political events),
- When the site is translated into other languages, the standardized lists can be localized, and so the region and industry can be displayed in the viewer's language,
- Better search — for example, a search for philanthropists in the state of Washington can match this profile, because the list of regions can include the fact that Seattle is in Washington.

A database in which entities like region and industry are referred to by ID is called *normalized*,[ii] whereas a database that duplicates the names and properties of entities on each document is *denormalized*. Normalization is a popular topic of debate among database administrators.

Duplication of data is appropriate in some situations and inappropriate in others, and it generally needs to be handled carefully. We discuss caching, denormalization and derived data in (to come) of this book.

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

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

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

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

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

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

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

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

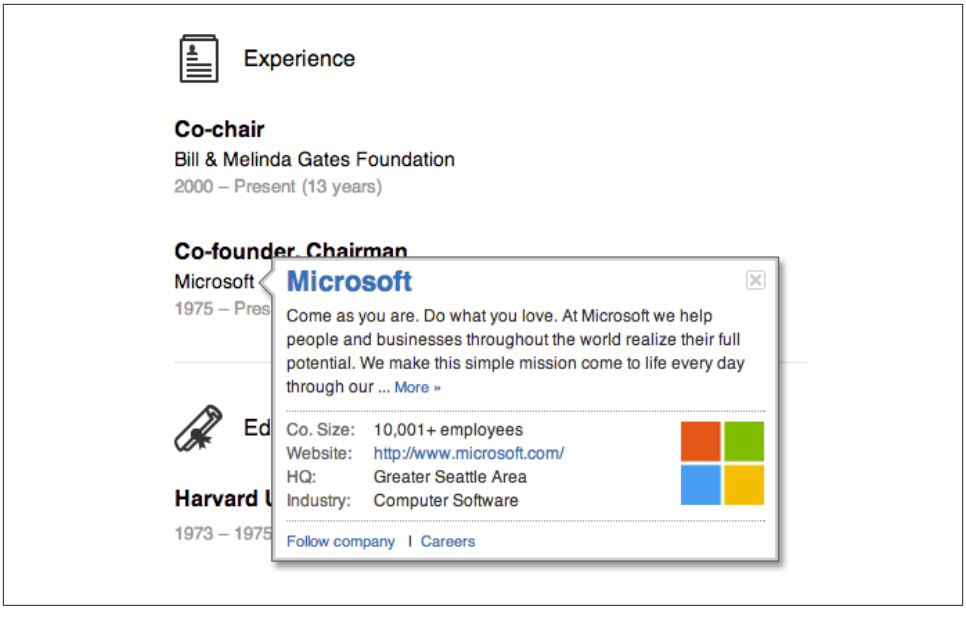



*Figure 2-3. The company name is not just a string, but a link to a company entity.*

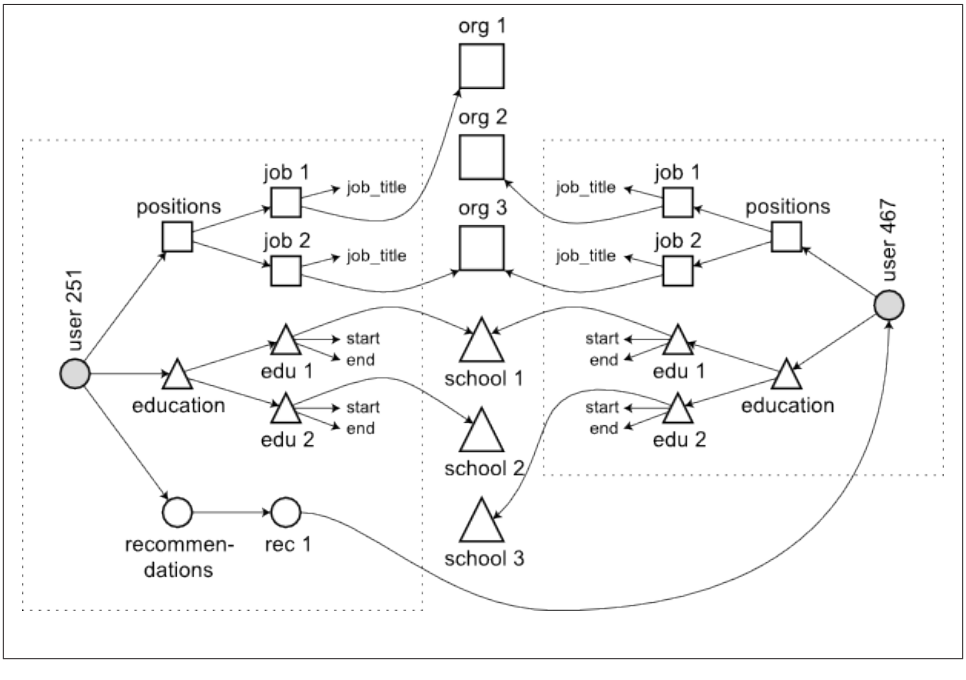



*Figure 2-4. Extending résumés with many-to-many relationships.*

## Are document databases repeating history?

While many-to-many relationships and joins are routinely used in relational databases without thinking twice, document databases and NoSQL reopened the debate on how best to represent such relationships in a database. This debate is much older than NoSQL — in fact, it goes back to the very earliest computerized database systems.

The most popular database for business data processing in the 1970s was IBM's *Information Management System* (IMS), a database system originally developed for stock-keeping in the Apollo space program, and first commercially released in 1968. [12] It is still in use and maintained today, running on OS/390 on IBM mainframes. [13]

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

Like document databases, IMS worked well for one-to-many relationships, but it made many-to-many relationships difficult and it didn't support joins. Developers had to decide whether to duplicate (denormalize) data, or to manually resolve references from one record to another. These problems of the 1960s were very much like the problems that developers are running into with document databases today. [14]

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

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

### The network model

The network model was standardized by a committee called *Conference on Data Systems Languages* (CODASYL), and implemented by several different database vendors, so it is also known as the *CODASYL model*. [15]

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

The links between records in the network model are not foreign keys, but more like pointers in a programming language (while still being stored on disk). The only way of

accessing a record was to follow a path from a root record along these chains of links. This was called an *access path*.

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

A query in CODASYL was performed by moving a cursor through the database by iterating over lists of records and following access paths. If a record had multiple parents, the application code had to keep track of all the various parent relationships. Even CODASYL committee members admitted that this was like navigating around an *n*-dimensional data space. [16]

The problem with access paths is that they make the code for querying and updating the database complicated and inflexible. With both the hierarchical and the network model, if you didn't have a path to the data you wanted, you were in a difficult situation. You could change the access paths, but then you had to go through a lot of hand-written database query code and rewrite it to handle the new access paths. It was difficult to make changes to an application's data model.

### The relational model

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

In a relational database, the query optimizer automatically decides which parts of the query to execute in which order, and which indexes to use. Those choices are effectively the "access path", but the big difference is that they are made automatically the query optimizer, not by the application developer, so we rarely need to think about them.

If you want to query your data in new ways, you can just declare a new index, and queries will automatically use whichever indexes are most appropriate. You don't need to change your queries to take advantage of a new index. (See also "Query Languages for Data" on

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

page 40.) The relational model thus made it much easier to add new features to applications.

Query optimizers for relational databases are complicated beasts, and they have consumed many years of research and development effort. [17] But a key insight of the relational model was this: you only need to build a query optimizer once, and then all applications that use the database can benefit from it. If you don't have a query optimizer, it's easier to hand-code the access paths for a particular query than to write a general-purpose optimizer — but the general-purpose solution wins in the long run.

#### Comparison to document databases

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

However, when it comes to representing many-to-one and many-to-many relationships, relational and document databases are not fundamentally different: in both cases, the related item is referenced by a unique identifier, which is called a *foreign key* in the relational model, and a *document reference* in the document model. [8] That identifier is resolved at read time, by using a join or follow-up queries. To date, document databases have not followed the path of CODASYL.

## Relational vs. document databases today

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

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

#### Which data model leads to simpler application code?

If the data in your application has a document-like structure (i.e. a tree of one-to-many relationships, where typically the entire tree is loaded at once), then it's probably a good idea to use a document model. The relational technique of *shredding* — splitting a document-like structure into multiple tables (like `positions`, `education` and `contact_info` in Figure 2-1) — can lead to cumbersome schemas and unnecessarily complicated application code.

The document model has limitations — for example, you cannot refer directly to a nested item within a document, but instead you need to say something like *"the second item*

*in the list of positions for user 251"* (much like an access path in the hierarchical model). However, as long as documents are not too deeply nested, that is not usually a problem.

The poor support for joins in document databases may or may not be a problem, depending on the application. For example, many-to-many relationships may never be needed in an analytics application that uses a document database to record which events occurred at which time. [18]

However, if your application does use many-to-many relationships, the document model becomes less appealing. It's possible to reduce the need for joins by denormalizing, but then the application code needs to do additional work to keep the denormalized data consistent. Joins can be emulated in application code by making multiple requests to the database, but that also moves complexity into the application, and is usually slower than a join performed by specialized code inside the database. In such cases, using a document model can lead to significantly more complex application code and worse performance. [14]

It's not possible to say in general which data model leads to simpler application code; it depends on the kinds of relationships that exist between data items. For highly interconnected data, the document model is very awkward, the relational model is acceptable, and graph models (see "Graph-like Data Models" on page 45) are the most natural.

### Schema flexibility in the document model

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

The question of whether databases should have schemas is a contentious issue, very much like the debate between advocates of static and dynamic type-checking in programming languages. [19] Many developers have strong opinions and good arguments for their respective viewpoints, and in general there's no right or wrong answer.

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

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

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

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

Schema changes have a bad reputation of being slow and requiring downtime. This reputation is not entirely deserved: most relational database systems execute the `ALTER TABLE` statement in a few milliseconds — with the exception of MySQL, which copies the entire table on `ALTER TABLE`, which can mean minutes or even hours of downtime when altering a large table. Various tools exist to work around this limitation of MySQL. [21, 22]

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

The schemaless approach is advantageous if the data is heterogeneous, i.e. the items in the collection don't all have the same structure for some reason, for example because:

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

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

### Data locality for queries

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

The locality advantage only applies if you need large parts of the document at the same time. The database typically needs to load the entire document, even if you access only a small portion of it, which can be wasteful on large documents. On updates to a document, the entire document usually needs to be re-written — only modifications that don't change the encoded size of a document can easily be performed in-place. [18] For

these reasons, it is generally recommended that you keep documents fairly small, and avoid writes that increase the size of a document. [8] These performance limitations significantly reduce the set of situations in which document databases are useful.

It's worth pointing out that the idea of grouping related data together for locality is not limited to the document model. For example, Google's Spanner database offers the same locality properties in a relational data model, by allowing the schema to declare that a table's rows should be interleaved (nested) within a parent table. [23] Oracle allows the same, using a feature called *multi-table index cluster tables*. [24] The *column-family* concept in the Bigtable data model (used in Cassandra and HBase) has a similar purpose in managing locality. [25]

We will also see more on locality in (to come).

### Convergence of document and relational databases

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

PostgreSQL since version 9.3 [7] and IBM DB2 since version 10.5 [26] also have a similar level of support for JSON documents. Given the popularity of JSON for web APIs, it is likely that other relational databases will follow their footsteps and add JSON support.

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

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

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

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

## Query Languages for Data

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

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

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

In the relational algebra, you would instead write:

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

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

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

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

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

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

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

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

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

Finally, declarative languages often lend themselves to parallel execution. Today, CPUs are getting faster by adding more cores, not by running at significantly higher clock speeds than before. Imperative code is very hard to parallelize across multiple cores and multiple machines, but declarative languages have a chance of getting faster. [27]

## Declarative queries on the web

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- If the `selected` class is removed (e.g. because the user clicked on a different page), the blue color won't be removed, even if the code is re-run — and so the item will remain highlighted until the page is reloaded. With CSS, the browser automatically detects when the `li.selected > p` rule no longer applies, and removes the blue background as soon as the `selected` class is removed.
- If you want to take advantage of a new API, such as `document.getElementsByClassName("selected")` or even `document.evaluate()` — which may improve

performance — you have to rewrite the code. On the other hand, browser vendors can improve the performance of CSS and XPath without breaking compatibility.

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

## MapReduce querying

*MapReduce* is a programming model for processing large amounts of data in bulk across many machines, popularized by Google. [28] A limited form of MapReduce is supported by some NoSQL data stores, including MongoDB and CouchDB, as a mechanism for performing read-only queries across many documents.

MapReduce in general is described in more detail in (to come). For now, we'll just briefly discuss MongoDB's use of the model.

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

The MapReduce model is best explained by example. Imagine you are a marine biologist, and you add an observation record to your database every time you see animals in the ocean. Now you want to generate a report saying how many sharks have been sighted per month.

In PostgreSQL you might express that query like this:

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

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

This query first filters the observations to only show species in the *Lamniformes* (mackerel 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.

vi. IMS and CODASYL both used imperative query APIs. Applications typically used COBOL code to iterate over records in the database, one record at a time. [2, 15]

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

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

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

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

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

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

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

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

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

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

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

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

Being able to use JavaScript code in the middle of a query is a great feature for advanced queries, and it's not limited to MapReduce — SQL databases can be extended with JavaScript functions too. [29] This means that MongoDB's MapReduce and SQL are roughly equivalent in terms of the kinds of queries you can express.

The difference is that with MapReduce, you have to write two carefully coordinated JavaScript functions, even for simple queries. This makes it harder to use than SQL, without significant advantages. So why was MapReduce chosen in the first place? Probably because it is easier to implement than a declarative query language, and perhaps because the term *MapReduce* sounds like high scalability, due to its association with Google.

The MongoDB team realized this too, and added a declarative query language called *aggregation pipeline* to MongoDB 2.2. [8] The kinds of queries you can write with it are very similar to SQL. Because it is declarative, it is able to perform automatic optimizations that are not possible with MapReduce. The same shark-counting query looks like this:

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

SQL is often criticized for its cumbersome syntax, but it's debatable whether this is any better.

# Graph-like Data Models

We saw earlier that many-to-many relationships are an important distinguishing feature between different data models. If your application has mostly one-to-many relation-

ships (tree-structured data) or no relationships between records, the document model is appropriate.

But what if many-to-many relationships are very common in your data? The relational model can handle simple cases of many-to-many relationships, but as the connections within your data become more complex, it becomes more natural to start modeling your data as a graph (Figure 2-5).

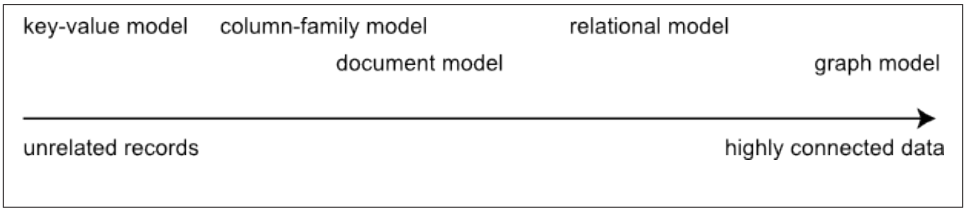



*Figure 2-5. Classifying data models by connectedness between items.*

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

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

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

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

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

In the examples above, all the vertices in a graph represent the same kind of thing (people, web pages or road junctions, respectively). However, graphs are not limited to such *homogeneous* data: an equally powerful use of graphs is to provide a consistent way of storing completely different types of object in a single data store. For example, Facebook maintains a single graph with many different types of vertex and edge: vertices represent people, locations, events, checkins and comments made by users; edges indicate which people are friends with each other, which checkin happened in which location, who commented on which post, who attended which event, etc. [30]

In this section we will use the example shown in Figure 2-6. It could be taken from a social network or a genealogical database: it shows two people, Lucy from Idaho and Alain from Beaune, France. They are married and living in London.

There are several different, but related, ways of structuring and querying data in graphs. In this section we will discuss the property graph model (implemented by Neo4j, Titan, InfiniteGraph) and the triple-store model (implemented by Datomic, AllegroGraph and others). We will look at three declarative query languages for graphs: Cypher, SPARQL, and Datalog. Imperative graph query languages such as Gremlin [31] are beyond the scope of this discussion, and graph processing frameworks like Pregel are covered in (to come).

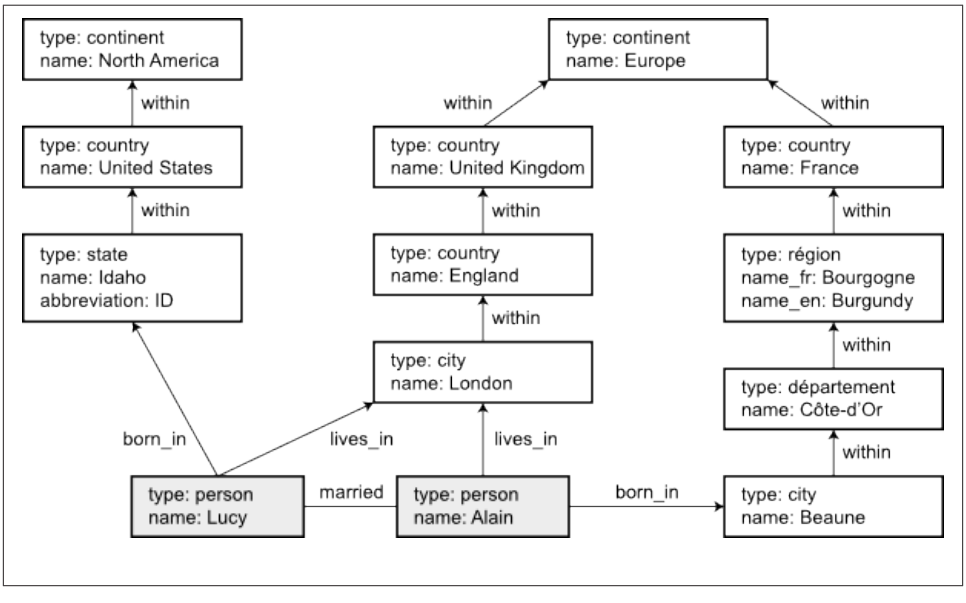



*Figure 2-6. Example of graph-structured data (boxes represent vertices, arrows represent edges).*

## Property graphs

In the property graph model, each vertex consists of:

- a unique identifier,
- a set of outgoing edges,
- a set of incoming edges, and
- a collection of properties (key-value pairs).

Each edge consists of:

- a unique identifier,
- the vertex at which the edge starts (the *tail vertex*),

- the vertex at which the edge ends (the *head vertex*),
- a label to describe the type of relationship between the two vertices, and
- a collection of properties (key-value pairs).

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

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

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

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

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

Some important aspects of this model are:

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

Those features give graphs a great deal of flexibility for data modeling, as illustrated in Figure 2-6. 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 cur-

rent residence is specified as a city, whereas her place of birth is specified only at the level of a state).

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

## The Cypher query language

*Cypher* is a declarative query language for property graphs, created for the Neo4j graph database. [32] (It is named after a character in the movie *The Matrix*, and is not related to ciphers in cryptography. [33])

Example 2-3 shows the Cypher query to insert the left-hand portion of Figure 2-6 into a graph database. The rest of the graph can be added similarly, and is omitted for readability. Each vertex is given a symbolic name like `USA` or `Idaho`, and other parts of the query can use those names to create edges between the vertices, using an arrow notation: `(Idaho) -[:WITHIN]-> (USA)` creates an edge of type `WITHIN`, with `Idaho` as tail node and `USA` as head node.

*Example 2-3. A subset of the data in Figure 2-6, 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 2-6 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*. To be precise: find all the vertices that have a `BORN_IN` edge to a location within the US, and also a `LIVING_IN` edge to a location within Europe, and return the `name` property of those vertices.

Example 2-4 shows how to express that query in Cypher. The same arrow notation is used in a `MATCH` clause to find patterns in the graph: `(person) -[:BORN_IN]-> ()` matches any two vertices that are related by an edge of type `BORN_IN`. The tail vertex of that edge is bound to the variable `person`, and the head vertex is left unnamed.

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

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

The query can be read as follows: "Find any vertex (call it *person*) that meets *both* of the following conditions:

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

For each such *person* vertex, return the *name* property."

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

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

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

## Graph queries in SQL

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

The answer is: yes, but with some difficulty. In a relational database, you usually know in advance which joins you need in your query. In a graph query, you may need to traverse a variable number of edges before you find the vertex you're looking for, i.e. the number of joins is not fixed in advance.

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

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

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

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

```
WITH RECURSIVE
  -- in_usa is the set of vertex IDs of all locations within the United States
  in_usa(vertex_id) AS (
      SELECT vertex_id FROM vertices WHERE properties->>'name' = 'United States' ❶
    UNION
      SELECT edges.tail_vertex FROM edges ❷
        JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
        WHERE edges.type = 'within'
  ),
  -- in_europe is the set of vertex IDs of all locations within Europe
  in_europe(vertex_id) AS (
      SELECT vertex_id FROM vertices WHERE properties->>'name' = 'Europe' ❸
    UNION
      SELECT edges.tail_vertex FROM edges
        JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
        WHERE edges.type = 'within'
  ),
  -- born_in_usa is the set of vertex IDs of all people born in the US
  born_in_usa(vertex_id) AS ( ❹
    SELECT edges.tail_vertex FROM edges
      JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
      WHERE edges.type = 'born_in'
  ),
  -- lives_in_europe is the set of vertex IDs of all people living in Europe
  lives_in_europe(vertex_id) AS ( ❺
    SELECT edges.tail_vertex FROM edges
      JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
      WHERE type = 'lives_in'
  )
SELECT vertices.properties->>'name'
FROM vertices
```

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

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

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

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

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

❺ Similarly, for each of the vertices in the set `in_europe`, follow incoming `lives_in` edges.

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

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

## Graph databases compared to the network model

In "Are document databases repeating history?" on page 34 we discussed how CODASYL and the relational model competed to solve the problem of many-to-many relationships in IMS. At first glance, CODASYL's network model looks similar to the graph model. Are graph databases the second coming of CODASYL in disguise?

No. They differ in several important ways:

- In CODASYL, a database had a schema that specified which record type could be nested within which other record type. In a graph database, there are is no such restriction: any vertex can have an edge to any other vertex. This gives much greater flexibility for applications to adapt to changing requirements.
- In CODASYL, the only way to reach a particular record was to traverse one of the access paths to it. In a graph database, you can refer directly to any vertex by its unique ID, or you can use a property index to find vertices with a particular value.
- In CODASYL, the children of a record were an ordered set, so the database had to maintain that ordering (which had consequences for the storage layout) and applications that inserted new records into the database had to worry about the position of the new record in these sets. In a graph database, there is no defined ordering of vertices or edges (you can only sort the results when making a query).

- In CODASYL, all queries were imperative, difficult to write and easily broken by changes in the schema. In a graph database, you can write your traversal in imperative code if you want to, but most graph databases also support high-level, declarative query languages such as Cypher or SPARQL.

## Triple-stores and SPARQL

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

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

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

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

For example, Example 2-6 shows the same data as in Example 2-3, written as triples in a format called *Turtle*, a subset of *Notation3* (*N3*). [34]

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

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

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

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

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

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

### The semantic web

If you read more about triple-stores, you may get sucked into a maelstrom of articles written about the *semantic web*. The triple-store data model is completely independent of the semantic web — for example, Datomic [35] is a triple-store that does not claim to have anything to do with the semantic web. But since the two are so closely linked in many people's minds, we should discuss them briefly.

The semantic web is fundamentally a simple and reasonable idea: websites already publish information as text and pictures for humans to read — why don't they also publish information as machine-readable data for computers to read? The *Resource Description Framework* (RDF) [36] was intended as a mechanism for different websites to publish data in a consistent format, allowing data from different websites to be automatically combined into a *web of data*, a kind of internet-wide 'database of everything'.

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

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

### The RDF data model

The Turtle language we used in Example 2-7 is a human-readable format for RDF data. Sometimes RDF is also written in an XML format, which does the same thing much more verbosely — see Example 2-8. Turtle/N3 is preferable as it is much easier on the

eyes, and tools like Apache Jena [37] can automatically convert between different RDF formats if necessary.

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

```
<rdf:RDF xmlns="urn:x-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: rather than a predicate being just `WITHIN` or `LIVES_IN`, it is actually something like `<http://my-company.com/namespace#within>` or `<http://my-company.com/namespace#lives_in>`. The idea behind this: you should be able to combine your data with someone else's data, and if they attach a different meaning to the word `within` or `lives_in`, you won't get a conflict because their predicates are actually `<http://other.org/foo#within>` and `<http://other.org/foo#lives_in>`.

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

### The SPARQL query language

*SPARQL* is a query language for triple-stores using the RDF data model. [38] (It is an acronym for *SPARQL Protocol and RDF Query Language*, and pronounced "sparkle".) It predates Cypher, and since Cypher's pattern-matching is borrowed from SPARQL, they look quite similar. [32]

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

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

```
PREFIX : <urn:x-example:>

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

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

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

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

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

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

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

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

## The foundation: Datalog

*Datalog* is a much older language than SPARQL or Cypher, having been studied extensively by academics in the 1980s. [39, 40] It is less well-known among software engineers, but it is nevertheless important, because it provides the foundation that later query languages build upon.

Every year we have more cores within each CPU, more CPUs in each machine, and more machines in a networked cluster. As programmers we still haven't figured out a good answer to the problem of how to best work with all that parallelism. But it has been suggested that declarative languages based on Datalog may be the future for parallel

programming, which has rekindled interest in Datalog recently. [27] There has also been a lot of research into evaluating Datalog queries efficiently. [39]

In practice, Datalog is used in a few data systems: for example, it is the query language of Datomic [35], and Cascalog [41] is a Datalog implementation for querying large datasets in Hadoop.[vii]

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

*Example 2-10. A subset of the data in Figure 2-6, represented as Datalog facts.*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

One possible way of applying the rules is thus:

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

By repeated application of rules 1 and 2, the `within_recursive` predicate can tell us all the locations in North America (or any other location name) contained in our database. This is illustrated in Figure 2-7.

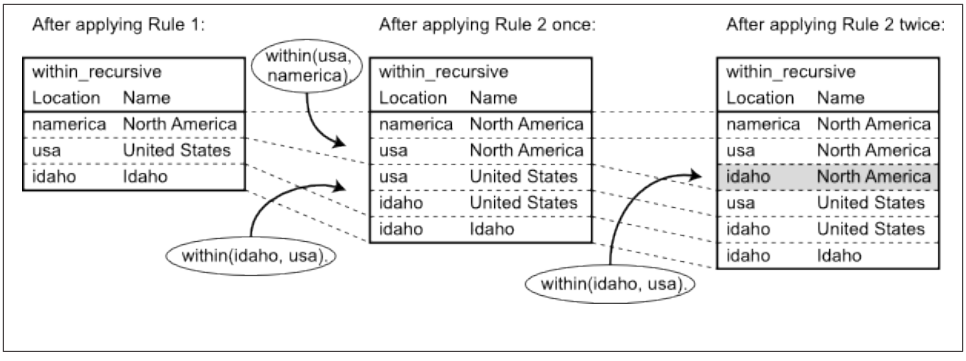



*Figure 2-7. Determining that Idaho is in North America, using the Datalog rules from Example 2-11.*

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

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

# Summary

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

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

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

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

One thing that document and graph databases have in common is that they typically don't enforce a schema for the data they store, which can make it easier to adapt applications to changing requirements.

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

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

- Researchers working with genome data often need to perform *sequence-similarity searches*, which means taking one very long string (representing a DNA molecule) and matching it against a large database of strings that are similar, but not identical. None of the databases described above can handle this kind of usage, which is why researchers have written specialized genome database software like GenBank. [42]
- Particle physicists have been doing *Big Data*-style large scale data analysis for decades, and projects like the Large Hadron Collider (LHC) now work with hundreds of petabytes! At such scale, custom solutions are required, to stop the hardware cost spiraling out of control. [43]

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

## References

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

[2] Michael Stonebraker and Joseph M Hellerstein: "What Goes Around Comes Around," in *Readings in Database Systems*, Fourth Edition, MIT Press, pages 2–41, 2005. ISBN: 9780262693141

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

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

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

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

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

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

[9] "RethinkDB 1.11 Documentation," www.rethinkdb.com, 2013.

[10] "Apache CouchDB 1.6 Documentation," docs.couchdb.org, 2013.

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

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

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

[14] Sarah Mei: "Why you should never use MongoDB," sarahmei.com, 11 November 2013.

[15] J S Knowles and D M R Bell: "The CODASYL Model," in *Databases - Role and Structure: an advanced course*, edited by P M Stocker, P M D Gray, and M P Atkinson, Cambridge University Press, pages 19–56, 1984. ISBN: 0521254302

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

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

[18] Sandeep Parikh and Kelly Stirman: "Schema design for time series data in MongoDB," blog.mongodb.org, 30 October 2013.

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

[20] Conrad Irwin: "MongoDB — confessions of a PostgreSQL lover," at *HTML5DevConf*, October 2013.

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

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

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

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

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

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

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

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

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

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

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

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

[33] Emil Eifrem: Twitter correspondence.

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

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

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

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

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

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

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

[41] Nathan Marz: "Cascalog," Apache License Version 2.0. cascalog.org

[42] 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

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