The Art of Immutable Architecture

Theory and Practice of Data Management
in Distributed Systems

Michael L. Perry

The Art of Immutable Architecture

Theory and Practice of Data Management in Distributed Systems

Michael L. Perry

The Art of Immutable Architecture: Theory and Practice of Data Management in Distributed Systems

Michael L. Perry
Allen, TX, USA

To Jenny. I wouldn't change a thing.

Table of Contents

About the Author

Michael L. Perry has built upon the works of mathematicians such as Bertrand Meyer, Leslie Lamport, and Donald Knuth to develop a mathematical system for software development. He has captured this system in a set of open source projects. Michael often presents on math and software at events and online. You can find out more at qedcode.com.

About the Technical Reviewer

Carsten Thomsen is a back-end developer primarily but working with smaller front-end bits as well. He has authored and reviewed a number of books and created numerous Microsoft Learning courses, all to do with software development. He works as freelancer/contractor in various countries in Europe, using Azure, Visual Studio, Azure DevOps, and GitHub as some of the tools he works with. Being an exceptional troubleshooter, asking the right questions, including the less logical ones, in a most logical to least logical fashion, he also enjoys working with architecture, research, analysis, development, testing, and bug fixing. He is a very good communicator with great mentoring and team lead skills and great skills in researching and presenting new material.

Acknowledgments

As Tolkien reminds us, it's dangerous business going out your door. The first step onto the road that has led to this book being in your hands was a shaky one. I was building my first distributed system and making all of the rookie mistakes. Fortunately, I had good friends to make those mistakes with.

Thank you, Russell Elledge and Jerry Feris, for fumbling alongside me. Together, we the Three Amigos learned all the wrong ways to use TCP/IP and SOAP. Who knew that the three-way handshake was not sufficient to guarantee delivery?

Although those first attempts were rough, we started to figure things out. Russell has been my constant co-conspirator, sounding board, and critic throughout this journey. I need to thank you also for introducing me to Chris Gould, who gave us both the freedom to apply what we had learned since that fateful first attempt. His support enabled us to build just the right solution on a mathematically sound foundation. It was the success of that project that gave me the final confirmation that these concepts *can* be taught.

A huge thank you goes out to Sean Whitesell for years of support, encouragement, and discussion. You always ask the best questions. Just as importantly, you are skilled at bringing people together. Thank you for building the community that helped me practice communicating the ideas that ended up in this book. And thank you especially for making the final connection to get this project started.

It was also Sean who introduced me to Floyd May. Floyd, you are such a deep thinker in technology, interpersonal relationships, and business. You have challenged me to become a better communicator. I cannot wait to see where your feet sweep you off to.

To all of my friends at Improving: Cori Drew, Harold Pulcher, Barry Forrest, Ben Kennedy, David Vibbert, David Belcher, David O'Hara...all the Daves. We have grown so much together. I remember the first time I met each of you, and all of the things we learned since then. Thanks especially to Tim Rayburn for helping me grow as a speaker, as an Improver, and as a leader. And please don't tell Devlin Liles that I think he's the most brilliant person in the company. I don't think I could stand him if he knew.

Introduction

It was 2001. I joined a team using J2EE version 1.3 to build a distributed gift card processor. The point of sale system was written in Microsoft Visual C++ 6.0. We were just learning about this new thing called SOAP, the Simple Object Access Protocol. The running joke was that it was too ill defined to be called a protocol, that it was not about accessing objects, and it was anything but simple. But it did hold some promise for making a C++ client talk to a Java server.

We all added three new books to our libraries. The first was on implementing a SOAP client in C++. The second was on JAXP, the Java API for XML Processing. And the third detailed the operation and limitations of TCP/IP. Armed with these tools, we began to build.

At first, the challenge was just to get the two platforms to talk to each other. When we finally settled on a subset of SOAP that both sides could handle, we thought we were over the hump. Little did we know that on the other side lay mountains.

There were reliability problems with the network. We set up a lab that continually ran transactions every night. We would check the card balances in the morning to find that some machines would have the wrong total. That led to a day of digging through logs, setting up the next test run, and then leaving it going until morning.

Over time, we evolved a message exchange protocol (over SOAP) based on confirmations and acknowledgments. One side sent a message. The next morning, we found messages missing. So next, the recipient confirmed that the message arrived. The next morning, we found duplicates. And so the sender acknowledged the confirmation. Fewer missing messages, but still not perfect.

It took many failed releases and many years of busy holiday seasons to work through all of the problems. We learned about the Two Generals' Problem (TGP) and realized why our message exchange protocol was flawed. Then we learned about eventual consistency and designed a working solution. This solution required that there be some uncertainty about how much money was left on a gift card. We tried to have that conversation with the product owner. Bankers get eventual consistency of money. Our product owner was not a banker.

The lessons we learned from gift cards were learned the hard way. "Guaranteed delivery" does not mean what you think it means. You need to first move data, then process it. Remote procedure calls (RPCs) aren't procedure calls. There is no line of code in a client–server system before which the transaction rolls back and after which it commits. I didn't want to learn those lessons over and over again.

And so I started putting those lessons together and defining a system that I called Historical Modeling. It was based on the idea that historical facts cannot be modified or destroyed. It relied upon the predecessor/successor relationships among facts. And it identified facts based only on their content, not on their location. I filled a notebook with examples of historical models. Eventually, I gained an intuitive feel for which kinds of solutions could be modeled historically and which could not. That's when I knew that I had to share it. Hopefully I could save someone else the pain of learning these lessons the hard way.

Since then, I have had countless conversations about immutable architectures. I broke the topic down into digestible chunks for conference and user group talks. I have created two open source frameworks—Correspondence and Jinaga. Yet none of that has truly empowered others to begin practicing immutability themselves. It can't just be adopted in pieces. Taking on only a subset of the ideas leaves gaps that can only be filled with the rest of the system.

Which lead to the book that you are now holding. This is a complete treatment of the system, the patterns, and the techniques. It anticipates the problems that Historical Modeling creates and provides the solutions that enable a cohesive implementation. Most importantly, it presents the mathematical foundation that makes the technique work.

If you have read this far into the introduction, you have probably faced some of these same problems. You might even have come up with similar solutions. This leaves only a few more questions you probably have about this book. Who should read it? What will I get out of it? How is it organized? And how do I go about reading it?

Glad you asked.

Who Should Read It

This book is intended primarily for three audiences: decision makers, system builders, and tool crafters. You are a decision maker if you identify the problems for which you want to create solutions. Your title might be CTO, product owner, or business systems

analyst. There are some problems that you can outsource, some that you can buy solutions for, and some that define your core business value. You need to find just the right team to build solutions to problems of this third kind. To find them, you need to be able to talk to them. And once you've brought them on board, you need to understand what they are doing. If your core business problem looks like the kind of thing that can be solved with an immutable architecture, this book will help you build that team and have those conversations.

Or perhaps you are a system builder. You are a member of the team brought in to deliver value against a core business domain. Your title might be developer, QA engineer, or user experience designer. You know how to solve problems. But it would be great to have some ready-made solutions to the most common problems of distributed computing. You want to know that all of the edge cases are accounted for. You desire a common language to talk about solutions with the people who are helping you build them. If your software development challenges require constructing eventually consistent distributed systems, then this book will give you those tools.

Finally, you—like me—might be a tool crafter. You are a force multiplier. The things that you build empower others to build solutions more quickly, more predictably, and more effectively. You might be a solutions architect or an open source maintainer. If you have a team, you want them focused on delivering business value while you take care of the plumbing. If you serve the community, you want consumers to be able to quickly learn and apply your framework to build robust systems. In either case, this book lays out the mathematics, algorithms, and patterns that assure the correctness of your solutions.

What You Will Get Out of It

I have a secret. This is a math book. Don't tell anybody who hasn't read this far into the introduction.

Mathematics is the greatest invention of humankind. It is surprising in its ability to describe the natural world. It is astonishingly applicable to a broad range of problems. And it is the only way that we can be sure of anything.

The way that we normally learn that we have gotten something right is to test it. We'll put our solution in one situation and see if we get the expected result. Then we'll try another scenario and see what it does. If we are really good, then we can imagine a few unexpected conditions and test for those. But the unexpected is really hard to anticipate.

Testing is all about gathering empirical evidence. It only gives you confidence that the system behaves as expected in certain cases. It does not give you any assurance that you haven't missed something.

Knowing requires mathematical deduction. If something is proven mathematically, then you can be sure that it will be true no matter what test case you try. Pythagoras is true for any right triangle. Euclid holds up for all figures on the plane. If your reasoning is sound, you can be sure that you haven't missed any edge cases.

It's not that mathematical truths are universal. It's that they come with known limitations. Division only works for nonzero divisors. Pythagoras only holds on the plane. The rules of deduction tell us how to carry those boundaries through to the solution, so that we know precisely where that solution applies and where it doesn't.

This book applies mathematical rigor to the problem of distributed computing. It is not the first to do so, but it does provide a complete and practical solution. If you follow the deductive reasoning over the problem and carry the limitations of distributed systems through your calculations, you will end up with an understanding of the boundaries of the solution. This book is your guide through that process.

How It Is Organized

The book is roughly divided into three parts, analogous to the three primary audiences. Decision makers need only read the first part, which includes the first three chapters. In this part, you first learn why immutability is so important. Then you explore the space of alternatives, eventually landing on Historical Modeling. Finally, you learn how to read a Historical Model so that you can communicate more effectively with your team. You can stop reading when we get into some deep math.

System builders will want to continue on to the second part. This includes Chapters 4 through 8. We get neck deep in the mathematical foundations of immutability, causality, and conflict-free replicated data types (CRDTs). Then we see how to apply this mathematical reasoning to analyzing systems, building state machines, and enforcing security rules. These are the tools that you will need to build robust distributed systems. We round out this section with a catalog of basic patterns to get you started building historical models.

My people, the tool crafters, will want to read right through to the end. In the third part, we take apart the components of a computer system and discover how to use them in an immutable architecture. We will update the user interface using query

inverters. We will store immutable records in a relational database. And we will exchange immutable messages securely and reliably over different kinds of networks. In the end, we pull it all together and describe an ecosystem made up of collaborative applications generating emergent behavior from shared specifications. It's something truly beautiful and inspiring, and I hope you follow me to the end.

How to Read It

Now that you know this is a math book, you might have some reservations about how you are going to read it. Perhaps you struggled through algebra or dropped out of calculus. You might think that math is not for you.

It is my belief that math is for everyone. And it is my goal with this book to prove it. Mathematics is nothing more than applying logical reasoning over symbolic representations of abstract concepts. Programming, on the other hand, is applying logical operations to a symbolic language describing generic rules. In other words, they are the same thing. If you are a programmer, then you are an applied mathematician.

One problem with mathematics is the jargon. In order to efficiently communicate with each other, mathematicians have to come up with words to represent ideas. Unfortunately, natural language is limited, and all of the good words are taken. And so mathematicians either make up new words or use terms that almost mean the right thing. In this book, we will be talking about the properties of a join semilattice. But I will try not to use those words if I can avoid them. And if I can't avoid them, I will clearly define them.

Another problem with mathematics is how it is written. Math papers have a predictable form. They start with an abstract. Then they fully define the problem. What follows is section after section of lemmas and propositions building an argument. Every statement is justified by the statements before, until finally, like an M. Night Shyamalan plot twist, one final assertion puts the whole argument into perspective and the result emerges.

While I really enjoy a good math paper, I don't read them the way that they are written. I skim the first few paragraphs for the motivation behind the problem. I scan the headings for the outline of the argument. I want to know why each statement is proven and how it will contribute to the whole. I want to know how the story is going to play out before I invest the time in understanding it.

I wrote this book the way that I read a math paper. In each section, you will understand the motivation behind a certain result. Then you will see a sketch of the basic reasoning. There will be no mystery why each of the steps is there. Then the section will justify each of those steps with the rigor they require.

I fully anticipate that this will impact the way you read the book. If you are after results, you can read just a paragraph or two past the section header. If you want to know why or how, then continue a bit further to understand the argument. And if you need to be convinced, then finish out the whole section. The important thing is that you can stop reading whenever it gets too deep and skip to the next section. You won't miss anything important to you.

If you have read this section without skipping anything, then I am truly pleased to have you. You are one of my people. With your help, we can build the software that the world needs. We will make it reliable, efficient, and correct. And it will give our users the autonomy they need to do their jobs with creativity and confidence, knowing that we have provided the mathematical rigor.

PART I

Definition

Why Immutable Architecture

Distributed systems are hard.

Most of us have used a website to buy a product. You might have seen a purchase page that contains a warning **do not click submit twice!** Maybe you've used a site that simply disables the buy button after you click it. The authors of that site have run up against one of the hard problems of distributed systems and did not know how to solve it. They abdicated the responsibility of preventing duplicate charges to the consumer.

Maybe you've used a mobile application on a train. The train enters a tunnel just as you save some data. The mobile app spins for a few seconds before you realize that you are in a race. Will the train leave the tunnel before the app gives up? Will the app correct itself once the connection is reestablished? Or will you lose your data and have to enter it again?

If you are involved in the creation of distributed systems, you are expected to find, fix, and prevent these kinds of bugs. If you are in QA, it is your job to imagine all of the possible scenarios and then replicate them in the lab. If you are in development, you need to code for all of the various exceptions and race conditions. And if you are in architecture, you are responsible for cutting the Gordian Knot of possible failures and mitigations. This is the fragile process by which we build the systems that run our society.

The Immutability Solution

Distributed systems are hard to write, test, and maintain. They are unreliable, unpredictable, and insecure. The process by which we build them is certain to miss defects that will adversely affect our users. But it is not your fault. As long as we depend upon individuals to find, fix, and mitigate these problems, defects will be missed.

This book explores a different process for building distributed systems. Rather than connecting programs together and testing away the defects, this approach starts with a fundamental representation of the business problem that *spans* machines. And this fundamental representation is *immutable*.

On its face, immutability is a simple concept. Write down some data, and ensure that it never changes. It can never be modified, updated, or deleted. It is indelible. Immutability solves the problem of distributed systems for one simple reason: every copy of an immutable object is just as good as any other copy. As long as things never change, keeping distant copies in sync is a trivial problem.

The Problems with Immutability

Unfortunately, immutability is counter to how computers actually work. A machine has a limited amount of memory. Machines work by modifying the contents memory locations over time to update their internal state. So the first problem of modeling immutable data on a computer is how to represent it in fixed mutable memory.

The second problem is that when we look out at the world of problems that we want to solve, we see change. People change their names, addresses, and phone numbers. Bank account balances go up and down. Property changes hands and ownership is transferred. How then are we to model a changing problem space with unchanging data?

Our initial instinct is to model the mutable world within the mutable space of the computer. This is the solution that has led us to build programs and databases based on mutation. Programs have assignment statements; databases have UPDATE statements. When we connect those programs and databases together to create distributed systems, crazy unpredictable behaviors emerge. And we are left with the unending task of testing until all of those anomalies are gone.

Begin a New Journey

What this book seeks to do is instead to model the business domain as one large immutable data structure. It would be impossible for a single machine or database to house that entire structure. Nor would that be desirable. And so the book also seeks to demonstrate how to implement subsets of that data structure within individual

databases, programs, and machines. These components communicate through well-crafted protocols that honor the idiosyncrasies of distributed systems to evolve that immutable data structure over time.

This solution is not new. Throughout this book, we will revisit research from the past in the form of math and computer science papers. Every claim is justified. None of the findings are original. I hope only to assemble this knowledge into a single consumable package that initiates your journey toward more reliable, resilient, and secure distributed systems. Let's begin that journey by understanding the problem of distributed computing.

The Fallacies of Distributed Computing

Between 1991 and 1997, engineers at Sun Microsystems collected a list of mistakes that programmers commonly make when writing software for networked computers. Bill Joy, Dave Lyon, Peter Deutsch, and James Gosling cataloged eight assumptions that developers commonly hold about distributed computing. These assumptions, while obviously incorrect when stated explicitly, nevertheless inform many of the decisions that the Sun engineers found in systems of the day.

The fallacies are these:

- The network is reliable.

- Latency is zero.

- Bandwidth is infinite.

- The network is secure.

- Topology doesn't change.

- There is one administrator.

- Transport cost is zero.

- The network is homogeneous.

Although it has been years since that list was written, many of these assumptions continue to be common. I can recall on several occasions being surprised that a program that worked flawlessly on *localhost* failed quickly when deployed to a test environment. The program contained hidden assumptions that the network was reliable, that latency was zero, and that the topology doesn't change. Here are examples of just these three.

The Network Is Not Reliable

One way in which these fallacies appear in modern systems is when a remote API is presented as if it were a function call. Several platform services have promoted this abstraction, including remote procedure calls, .NET Remoting, Distributed COM, SOAP, and SignalR. When a remote invocation is made to look like a local function call, it is easy for a developer to forget that the network is not reliable.

Any time you call a function, you can rest assured that execution will continue with its first line. And if the function makes it to the `return` statement, you can feel pretty confident that the next line to run will be the one following the function call. Remote procedure calls, however, make no such claims. They can fail on invocation or on return. The calling code will be unable to tell which.

An abstraction that hides the fact of a network hop does a disservice to its consumers. In an effort to make things easier and more familiar, it pretends that an inconvenient truth can be ignored. Such abstractions make it easier for developers to believe the fallacy that the network is somehow reliable.

Latency Is Not Zero

Modern web applications have moved away from the client proxy in favor of more explicit REST APIs. These APIs avoid the mistake of presenting the remote machine as if it were a library of functions that could be invoked reliably. They instead present the world as a web of interconnected resources, each responding to a small set of HTTP verbs. Unfortunately, this style of programming makes it easy to forget that latency is not zero.

Some of the HTTP verbs are guaranteed to be *idempotent*. If the client duplicates the request, the server promises not to duplicate the effect. There is no way for the protocol to enforce that guarantee, but server-side applications typically uphold the contract. Examples of HTTP verbs that are idempotent are PUT and PATCH. An HTTP verb that is *not* guaranteed to be idempotent is POST.

On the Web, HTTP POST is often used to submit a form. When a web application responds quickly, the lack of idempotency guarantee makes little difference. But as latency increases, the user starts to wonder if they actually clicked the submit button. And if that button triggered a purchase, they have to wonder if they will be charged twice if they try again. An end user has no good recourse during an extended latency after clicking a *Buy* button. Nor does a client-side application developer have a good response to a timeout on POST.

There is no correct use of an API that features non-idempotent network requests. Because latency is not zero, there will always be a time during which the client is unsure if the server has received the request. As latency exceeds the time that the client is willing to wait, they must make a choice: either abort the attempt or retry. If the client aborts, then they don't know whether the request has been processed. And if they retry, then the effect might be duplicated.

The POST verb is indeed part of the HTTP specification. And that specification makes no guarantee as to its idempotency. But any API that includes a non-idempotent POST is making the incorrect assumption that latency is zero. It forces the client to make an impossible choice when that assumption proves false.

Topology Doesn't Change

Most database management systems include a concept that leads developers to assume that topology doesn't change. These databases make it easy to set the identity of a record to an auto-incremented ID. Every time a record is inserted, the database generates the next number in the sequence. This number is used from then on to identify the record.

An auto-incremented ID requires that topology remain constant throughout a multistep process. Imagine a web application that inserts a user's form data into a database and then redirects them to a page representing that new data. To accomplish this with an auto-incremented ID, the browser must wait for the request to go all the way to the database and the response to come all the way back before it can learn the URL of the next page. The application assumes that the topology will not change in the meantime.

This may seem on the surface to be a valid assumption. It will usually be true. Changes to server topology are rare, and network requests are usually fast (latency is zero). However, for a heavily trafficked web application, there will never be a moment during which no requests are in flight. The assumption that topology does not change will be violated for some requests.

Topology may change during a system upgrade. It will certainly change during a disaster failover. And it will change again when reverting back after the disaster is resolved. When topology changes, the database that a request ends up on will not be the same as the one that generated the source page. That database will instead be a replica of the original. If the replica is just a little behind the original, then the change in topology will be noticeable. And it will be behind because, again, latency is not zero.

The use of auto-incremented IDs is ubiquitous. They are the default choice for most application database models. And yet their use belies an assumption that the topology will not change.

Changing Assumptions

The fallacies of distributed computing are easy assumptions to make. We make them because our tools, specifications, and training have led us to do so. The non-idempotent POST verb is a valid part of the HTTP specification. Auto-incrementing IDs are a valuable feature of most database management systems. Almost every tutorial on application development will teach a beginner to use these capabilities. The fact that by doing so they are making an incorrect assumption does not even occur to them.

The tools that we use and the patterns that we follow today all evolved from a time during which assumptions of high reliability, zero latency, and topological consistency were not fallacies. In-process procedure calls are perfectly reliable. Sequential program statements have very low, very predictable latency characteristics. And sequential counters in a for loop will never return to the top of the function to find the code's topology had changed. It's when we evolve these abstractions into RPCs, network requests, and auto-incremented IDs that problems arise. When we apply the languages and patterns of the past to the problems of modern distributed systems, it is no wonder that programmers will make incorrect assumptions.

All of the fallacies of distributed computing stem from one simple truth: distributed systems are built using tools designed to run in a single thread on a single computer. Developers imagine a fast, isolated, unchanging, sequential execution environment and then treat the idiosyncrasies of distributed systems as edge cases. A duplicate transaction due to a network timeout is not a bug. An ID collision caused by a database failover is not a defect. These are realities of distributed systems that we cannot code around or test away. They demand a new set of tools, patterns, and assumptions.

Immutability Changes Everything

In 2015, Pat Helland wrote *Immutability Changes Everything*,[1] an analysis of several computing solutions based on immutability. It demonstrates that immutability solves many problems in several layers of computational abstraction. At one end of the spectrum, low-level storage systems use copy-on-write semantics to mitigate against media wear. At the other end, applications accrete read-only facts and derive current state. This paper claims no new ideas, but only serves to point out the common thread of immutability in all of these solutions.

In the past, computers were slow, expensive, and limited machines that could only operate on small sets of data. Today, they are fast, cheap, and capable workhorses that store an embarrassment of data richness. Where application developers of the past had to optimize data storage by overwriting information when it was no longer needed, today we can afford to save everything. There is no economic need to update or destroy bits.

At the same time, computers of today are much more connected than they were in the past. Rather than co-locating a workload with the data on which it operates, we have moved to a world of microservices and mobile devices that share data far and wide. Many machines share the computational and storage burden of work that used to be performed by one. As a result, coordination has become more expensive, even as computing has become cheap.

And so while in the past it was expensive to keep immutable copies of data, current architectural constraints *require* that we do. Not only is data cheaper than it used to be, but making immutable copies actually *enables* the kinds of solutions that scale to multiple machines. When two machines share mutable data, they need to coordinate as that data changes. They may need to block one another to ensure that only one can change the data at any given time. But when that data cannot change, then no coordination or blocking is required. Cost reduction enables immutability, and immutability enables modern architecture.

Shared Mutable State

Many of the hard problems in computing are problems that we have created for ourselves. Take, for example, the problem of shared mutable state in a multi-threaded

---

[1]Helland, Pat. (2015). "Immutability Changes Everything." http://cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf

system. One thread writes source data into a shared memory location, and another thread performs calculations on it. These two threads must be carefully coordinated to ensure that one does not write to shared memory before the other is finished reading from it. If the first overwrites the data while the second is still calculating, the results would be complete nonsense. We typically solve this sort of problem with a lock, limiting the ability for the program to scale.
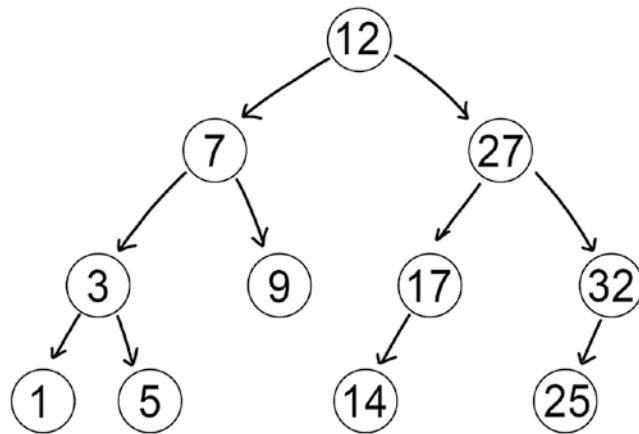
But there is a solution that does not impair scalability. Instead of a lock, we could use immutable data structures. Rather than overwriting memory with the next data set, the first thread would simply allocate new memory. When it is finished building the data structure, the first thread passes a pointer to the second. From that point on, no thread can modify the contents of that memory. It remains completely immutable.

On the surface, it appears that we have improved scalability at the cost of memory efficiency. Rather than modifying just one small part of a data structure, it would seem that we have to make an entire copy with every operation. If that were true, it would be hard to justify the trade-off, even with the decreased cost of storage. Fortunately, however, that is not a trade-off we have to make.

Structural Sharing

The fact that we intend for data structures to be immutable opens a new possibility. As we build new data structures, we can reuse existing pieces of old data structures. There is no need to copy those pieces, because we have already established that they will not change. We simply create new data elements to represent the ones that have "changed" and let them point to the ones that haven't.

This is a technique called *structural sharing*. It's a common optimization for immutable data structures that is *enabled* by immutable data structures. Take, for example, the binary tree shown in Figure 1-1. Each node in the tree contains a piece of data, in this case a number. It also contains two pointers, one to a number that is less than this node and one to a number that is greater. Finding a specific number in this data structure is fast, because you walk down a path asking "less than or greater than" at each stop.
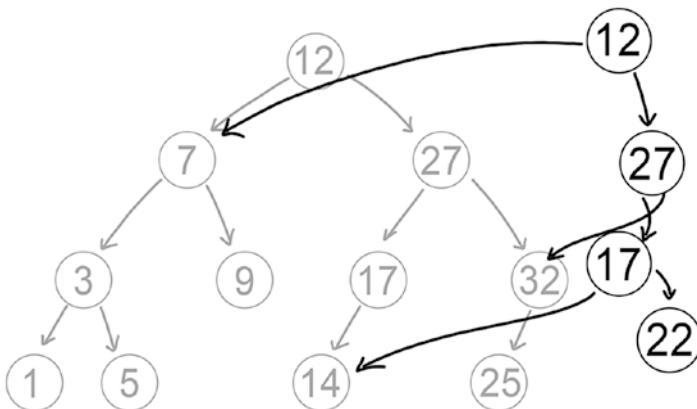
Figure 1-1. *A binary tree of numbers*

To insert a new number into the binary tree, you first need to locate the place that it belongs. Walking down to where it should be, you will discover either that it is less than a number that has no left path or greater than a number with no right path. Once there, your desire will be to "change" that node to add a new path. However, changing a node is not allowed: they are all part of an immutable data structure. So instead, you create a new node.

This new node should be to the left or right path of a parent, and so you will want to "change" that node as well. But again, changing the parent is not allowed. And so you create a new parent that points to the new child.

Continuing up the tree, you will eventually reach the root, as shown in Figure 1-2. No matter where you insert a new number, you will always end up creating a new root node. This new root node is effectively the new version of the tree. It represents the shape of the tree after the insertion. The previous root node still exists, and the nodes to which it points have not been modified. Any threads running in parallel searching that version of the tree can happily continue to do so. They will be unaffected by the new tree that shares most of its structure with the old one.

Figure 1-2. *After inserting 22, the new version of the binary tree shares most of its structure with the previous version*

This optimization would not be possible if threads could modify these data structures. By sharing structure, these two versions of the tree become sensitive to modifications. It's only because we have agreed *not* to modify the nodes that we can get away with this deep sharing of structure. Immutability enables structural sharing, and structural sharing optimizes immutability.

The Two Generals' Problem

Nowhere in computing is immutability more valuable than in sharing data among machines. But before we can truly understand why, we must first understand the scope of the problem. And there is no better way to do that than with the parable of the two generals.[2]
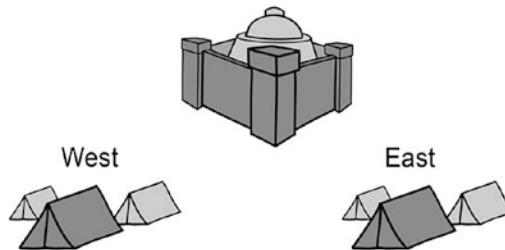
Imagine a besieged city. Within its walls, the defenses are insurmountable. A direct attack is almost certain to fail. Outside of the city are two armies, which have succeeded in cutting off its supply lines. The generals of these armies lie in wait, watching the city slowly weaken under the blockade.

At some point, the city's defenses will be weak enough to attack. The generals of these two armies—one in the East and one in the West—are constantly observing the situation through their network of scouts, spies, and messengers. They determine each

---

[2]Akkoyunlu, E. A.; Ekanadham, K.; Huber, R. V. "Some constraints and tradeoffs in the design of network communications." ACM SIGOPS Operating Systems Review. November 1975.

day whether the city is sufficiently weak. When the time comes, they will prepare an attack for the following day. This situation appears in Figure 1-3.
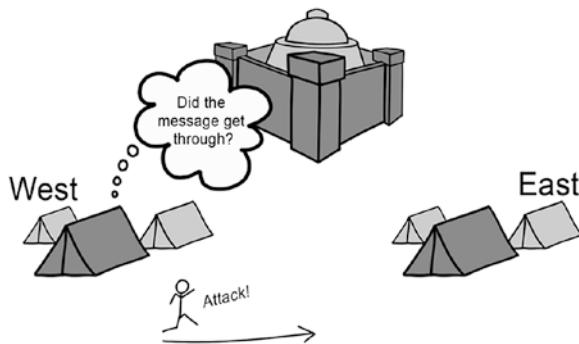
Figure 1-3. *Two armies encamped outside of a besieged city*

An attack from one army would not be sufficient. The attack would be repelled and the attacking army destroyed. The remaining army would not be able to maintain the blockade, and so it would be routed soon thereafter. Only a coordinated attack from both East and West will win the city.

Now imagine that you are the general of the West army. Your partner to the East is separated from you by enemy territory. You cannot communicate directly. You can only send messengers through hostile terrain with no guarantee of success. Any message could be lost, their carrier killed or captured. The two of you must devise a method of reliable communication built from unreliable components.

If you in the West determine that the city is weak enough, and that the time for attack has come, you will begin preparing your army. You will also send a messenger to the East to inform the other general that you will attack in the morning. If the messenger arrives safely, then the East general can begin preparations and join you in the attack. With your combined efforts, the attack is likely to succeed.

But if the messenger is killed or captured, the message will not arrive. If that happens, your army will set out in the morning to mount a lone attack against the city. Your army will be destroyed, and the siege will be lost. As Figure 1-4 shows, you are unsure of how to proceed. And so you must have assurance before the morning comes that the message has been received.
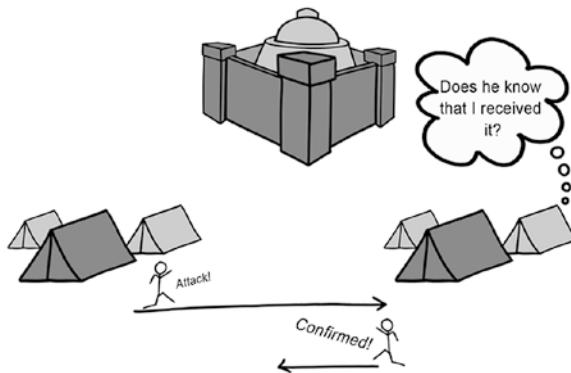
Figure 1-4. *The West general does not know whether they can attack*

A Prearranged Protocol

Let's try to devise a protocol that will give us some assurance that the message was received. Suppose you ask the East general to send a messenger in response confirming that your message was received. Now if you receive the confirmation before morning, you can confidently launch your attack. You know that the East general has received the message and will join you on the battlefield. If, on the other hand, you do not receive confirmation, then you will call off the attack, not knowing whether the original messenger made it through. As the general of the West army, you can be sure that you will not attack unless you know that the East general has received your message.

But while this protocol gives the West general those assurances, it fails to do so for the East general. Imagine now that you are on the East, and you have received a message informing you that the West will attack in the morning. You have plenty of time to begin preparations for your army. And, as per the protocol, you respond with confirmation. If the confirmation message reaches the West general, then the attack will proceed as planned.

But if that message is lost, then the West general will not attack. Remember, he is waiting for confirmation to know that you received his message. If you attack in the morning without knowing that the West general has received your confirmation, then your army could be defeated. And so you are left in uncertainty, as in Figure 1-5.

Figure 1-5. *The East general is uncertain*

Reducing the Uncertainty

This protocol is not sufficient. You try different strategies to improve upon it. The first strategy is to simply send more messengers. Instead of relying upon one messenger, you send two. The probability of two messages both being lost is certainly less than the probability of one being lost. But that probability is not zero. And so you try again.

You can send three messengers, four messengers. Choose any number you wish. As you increase the number, the probability of total message loss gets closer and closer to zero. But it never quite reaches it. You can never choose a number of messengers high enough to assure you that the message will be received.
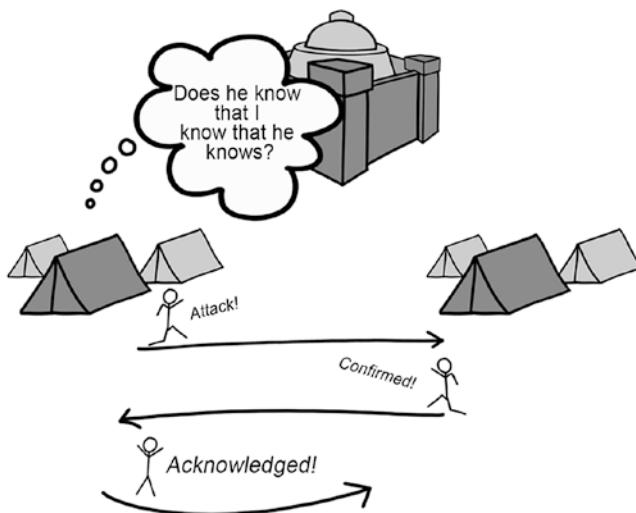
And so you change your approach. You send messengers out at a constant rate until the response is received. From the West, when you decide to attack, you send messengers with the *attack* message once every ten minutes. When you receive the first *confirmed* message from the East, you stop sending messages. As for the general on the East, he will reply with a *confirmed* message every time that an *attack* message is received. As long as he receives a steady stream of *attack* messages, he will respond at the same rate with confirmations. And once that stream stops, he can assume that the confirmation has been received.

Or can he? Can the lack of messages be taken as a signal? Is it possible that six messengers an hour continue to flow from the West, but all are captured? The general on the East has no way of ruling that out. And so he still runs the risk of attacking in the morning with no support from the West.

An Additional Message

As the East general, therefore, you make an additional demand of the protocol. In addition to an *attack* message from the West, and a *confirmed* message from the East, you require that the West respond with *acknowledged*. If you, on the East, receive *acknowledged* before the morning, then you know that *confirmed* was received in the West. You may therefore attack with confidence, knowing that the West general has received confirmation and will therefore join you. But if you receive no acknowledgment, then you must abstain.

While this new message provides new assurances to the East general, it again confounds the situation on the West. When the West general sends out an *acknowledged* message, he has no way of knowing whether it was received. If it was, then the East general will attack. If it wasn't, then the East general will abstain. And so, as Figure 1-6 illustrates, he has no assurance that his attack in the morning will be supported.

Figure 1-6. *The West general is again not sure if the East will attack*

The addition of one message has only moved the uncertainty to the other side of the conversation. It didn't actually solve the problem. We still have not yet discovered a protocol that will ensure that both armies either attack or abstain, when those two generals can only communicate via unreliable messages.

And indeed, we never will.

Proof of Impossibility

The Two Generals' Problem, as Jim Gray named it in 1978[3], has no solution. There is no finite protocol that can give both generals mutual assurance of an agreement. I'm not simply saying that no one has found a solution. I'm saying that no solution can exist.

E. A. Akkoyunlu, who published the original problem and the impossibility proof in 1975[4], named this mutual assurance *complete status*. He described interprocess communication protocols that negotiate transactions between participants. A protocol would ideally provide status to those participants regarding the outcome of every transaction. Akkoyunlu proved that a distributed system cannot achieve complete status in a finite number of messages.

His proof does not require that we exhaust all possible solutions. It leaves no room for clever tricks that we hadn't thought of. Instead, it is based on contradiction. Let anyone come up with a protocol and bring it to Akkoyunlu claiming that it provides complete status. Without even knowing how that protocol works, he shows that it does not uphold that claim.
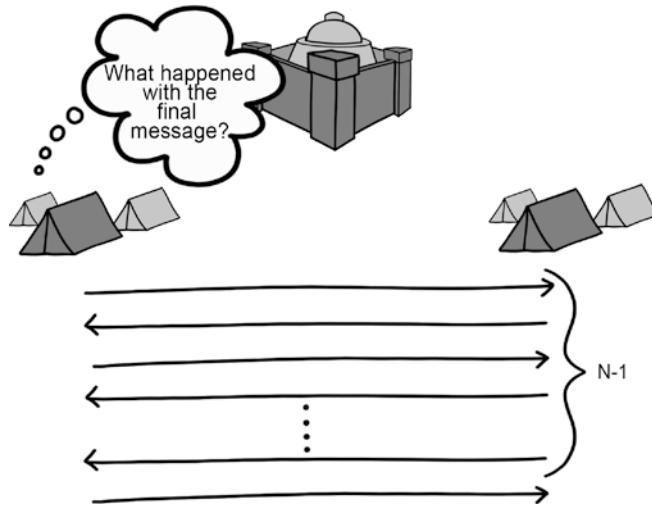
Suppose that you present a protocol that you claim provides complete status to two generals after a finite exchange of messages. At the end of this exchange, both generals will know that the other is going to attack. If the generals follow this protocol and it happens that no messages are lost, then there is a minimum number of messages that must have been exchanged to reach this point. We will call that number N. The number N is particular to the protocol.

Since N is the smallest number of messages that must be exchanged to reach complete status, we know that fewer would be insufficient. In particular, we have not reached complete status after $N-1$ messages. One of the generals must still be at the point where he is not sure whether the other is going to attack.

Since $N-1$ messages would be insufficient, the Nth message is important. Without it, the protocol would not work. And yet, the message is not guaranteed to arrive. The sender of the Nth message does not know whether it will be received. Therefore, the sender of the Nth message does not have complete status and will not receive complete status as there are no further messages in the protocol. This situation appears in Figure 1-7.

---

[3]Gray, Jim. (1978). "Notes on Data Base Operating Systems." Chapter 3. Operating Systems, an Advanced Course. Springer-Verlag, London, UK.

[4]Akkoyunlu, E. A.; Ekanadham, K.; Huber, R. V. (1975). "Some constraints and tradeoffs in the design of network communications." Published in SOSP 1975. DOI:10.1145/800213.806523.

Figure 1-7. *The sender of the final message does not have complete status*

This contradicts your claim that the protocol provides complete status within a finite number of messages. Therefore, we can conclude that no such protocol exists.

Relaxing Constraints

The Two Generals' Problem (TGP) is an analog for many of the problems we try to solve in distributed systems. Using only unreliable networks to pass messages between nodes, we must construct systems that nevertheless reach agreement with a high degree of certainty. The impossibility is the TGP would seem to tell us that this is a fool's errand. Fortunately, however, the problems that we solve in distributed systems are a little bit easier than this fictional analog.

Consider an ATM. A bank customer uses a terminal to withdraw cash from their account. This common everyday transaction appears to be a TGP-made real. On the West, you have an ATM terminal with the ability to dispense cash. On the East, you have a bank's central computer, which records the flow of money into and out of customer accounts. In between, the hostile territory of digital communications threatens to interrupt the delivery of messages.

Our desire is to ensure that the transaction either succeeds or fails. If it succeeds, the cash is dispensed and the customer's account is debited. If it fails, no cash is dispensed and no debit appears in the account. We wish to avoid an outcome which has success on one side and failure on the other. Customers would be very upset if their accounts were debited but no cash was forthcoming, and banks would lose money if their ATMs dispensed cash without a corresponding debit.

Redefining the Problem

The impossibility result of TGP tells us that this cannot be accomplished. And yet, millions of ATM transactions are processed every day.[5] Clearly something is out of alignment. What we have failed to recognize in the ATM example is that the constraints on the system are more relaxed than they appear at first. Let's take a closer look at the reason that the full TGP is impossible. From there, we can see how to relax the constraints and create a viable protocol.

The problem as originally stated has two strict constraints:

1. A general will not attack unless he has assurance that the other general will also attack.

2. The attack will come in the morning.

By the first constraint, the behavior of each general is based on what he knows about the behavior of the other general. As long as one general is in a state of uncertainty, both remain uncertain. There is no message that can simultaneously change both of their minds.

By the second constraint, there is a deadline. When that deadline arrives, they must achieve consensus. Any messages already en route at that time must have no effect on the final outcome. There will be no further messages to resolve any lingering uncertainty.

If we relax this pair of constraints, we can formulate a problem that has a valid solution. We can indeed find a protocol that exchanges complete status, as long as we allow one party to act in uncertainty and remove the deadline. Doing so destroys the narrative of the Two Generals' Problem, but it fits the ATM example. Indeed, we will find that this relaxed version fits *many* business problems that we solve with distributed systems.

---

[5]The 2013 Federal Reserve Payments Study reported 5.8 billion ATM withdrawals in 2012.

Decide and Act

We will first relax the constraint that a general will only attack if he is certain that his peer will as well. The West general decides that the time is right and prepares to attack regardless of what happens in the East. What is foolish behavior for a general could be a valid compromise for an ATM. When a customer withdraws money from their account through an ATM, one side or the other must act without full knowledge that the other will follow suit. Either the ATM must dispense the cash, or the central bank computer must record the debit. Consider the consequences and corrective steps of each decision, should it turn out to be one-sided.

Suppose that the bank records the debit, but the ATM terminal fails to dispense the cash. In that scenario, the customer leaves the terminal with no cash, but the central bank believes that they have their money. The consequence is that the customer is unsatisfied when they discover the problem, and their trust in the bank is eroded. The corrective action is to reverse the debit once the problem is discovered.

Now suppose that the ATM dispenses the cash, but the central bank fails to record the debit. In this scenario, the customer has left happy, and the ATM retries the communication until it is successful. In the meantime, it might be possible for the customer to withdraw money from another ATM, since the bank is unaware that their balance has been depleted. If so, the corrective action is to charge the customer an overdraft fee.

Clearly, one of these scenarios is better for both the bank and the customer. It protects trust, puts the power in the customer's hands, and gives the bank an additional revenue stream. And so in this situation, the designer of the distributed system determines that the ATM will dispense cash even while it is uncertain whether the central bank will record the debit.

Accept the Truth

The designer can only confidently make this decision if they relax the second constraint: that there is a deadline. Assume that the ATM has dispensed cash, but then experiences technical difficulties while communicating this fact to the central bank. It may take some time for a technician to repair the ATM terminal, thus reestablishing the communication channel. When the terminal shares with the bank that the cash was dispensed, the bank must honor this truth. It cannot reject the transaction based on the passage of time or the customer's current account balance.

The damage to the ATM may be so severe that the digital record of the transaction cannot be recovered. It may have experienced a full unrecoverable hard drive crash. In this case, additional forensics could be employed: count the cash remaining in the machine and determine whether the last transaction completed. If the ATM, including all of its cash, is totally destroyed, then even this method might not be available. But of course, in that case the bank has lost more than a single transaction. Accepting the truth means accepting some risk.

A Valid Protocol

Given these relaxed constraints, we can now devise a protocol that eventually achieves complete status. One side (the ATM in this case) reaches a point where it can confidently make a decision. It acts (dispenses cash) and then continues the protocol until it knows that the other side is aware of the decision. It continues to do so no matter how much time has passed, or what conflicting circumstances have intervened.

To reach the point of decision, the ATM communicates with the central bank. It verifies that the account holder has sufficient funds to dispense the requested cash. It also checks its local storage of bills to ensure that it will be able to complete its side of the transaction. In this process, the bank may place a temporary hold on the customer's funds. But this hold only reduces the likelihood of an overdraft; it cannot prevent it. The ATM for its part will put a temporary hold on its repository of bills: only one customer at a time may use the machine. If both of these checks pass, then the ATM dispenses the cash. It makes the final decision.

After it makes the decision, the ATM enters a second phase. In this phase, the decision has happened; the cash has been dispensed. The goal of this phase is simply to communicate this fact with the central bank. There is no time limit on the second phase, and the truth cannot be retracted.

This kind of protocol is what Jim Gray referred to in 1978 as a *Two Phase Commit* (2PC). In the first phase—commonly known as the *voting* phase—the coordinator receives from each participant confirmation that it can commit to the requested transaction. In the second phase—the *commit* phase—the coordinator informs each participant of its decision. In the preceding example, the ATM plays the role of coordinator and one participant. It is the sole decision maker, once it has gathered enough information to responsibly make that decision.

Examples of Immutable Architectures

The benefits of immutability have not gone unnoticed by distributed systems designers. Some of the most successful distributed systems in use today are built upon this concept. They derive capabilities from immutability that would be difficult to achieve otherwise. Three examples are Git, blockchain, and Docker.

Git is a distributed version control system popular among open source and corporate development teams alike. It offers the benefit of autonomy to each individual developer. A developer can make changes, switch among parallel lines of history, and resolve conflicts all within an isolated replica of the repository. When developers connect their replicas—whether directly to one another or to a shared central repository—they only trade information. No locking or consensus occurs during that exchange, keeping the interaction short.

Blockchain is an umbrella term for a collection of related architectures. The first blockchain was Bitcoin, a distributed currency based entirely on cryptographic algorithms. Most blockchains retain the economic aspects of a currency, but some layer additional features onto the core data structure. The prominent feature of a blockchain is a shared immutable ledger, providing assurance of the veracity of a singular, transparent history.

Docker is a technology for executing software within containers, as if the entire operating system and all dependencies were encased within a single isolated execution environment. It is an evolution beyond physical machines that truly ran isolated workloads and virtual machines that simulated that environment for the purposes of portability and scale. Docker achieves efficiencies that virtual machines lacked by a clever use of structural sharing and immutable disk images. This led the way to further advancement in orchestration such as Kubernetes clusters and mesh computing.

All of these examples use the benefits of immutability to enable their core defining capabilities. Interestingly enough, they all also happen to be open systems. Most likely that is simply a consequence of open software being readily available to analyze and uphold as architectural examples. There is no reason to believe that immutability would not be just as valuable to a closed system as it is to an open one. Let's analyze each one in a little more detail to see how immutability serves its goals.

Git

Git strives to give each developer autonomy by providing all of the necessary information in a replica of a repository. The repository is composed of individual changes to source code known as *commits*. Commits are immutable, and contain references to related commits. The whole of the repository is an ever-growing history of commits accreted over the life of the project.

A commit is a set of changes made to a project by a single developer at a single point in time. The identity of a commit is derived entirely from its contents: the names of the files affected, the changes made to those files, the name of the developer, and the reason and time of the change. Those contents are hashed, and the resulting hash code is henceforth used to identify the commit. This results in an immutable graph of commits like the one in Figure 1-8. Every developer who clones the repository will compute the same hash for each commit, thus making those identities deterministic and consistent.


```
*       249916e - (origin/master, origin/HEAD) Merge pull request #11 from micha
|\
| * 44644f4 - chore(package): update lockfile package-lock.json (5 weeks ago
| * 8aede8a - chore(package): update webpack to version 4.36.1 (5 weeks ago)
|/
| * 26223d5 - (origin/greenkeeper/jinaga-pin-2.3.8) fix: pin jinaga to 2.3.8
|/
| * a8613b0 - (origin/greenkeeper/jinaga-2.4.0) chore(package): update lockf
| * 1e480d3 - chore(package): update jinaga to version 2.4.0 (5 weeks ago) <
|/
*       4037b30 - Merge pull request #8 from michaellperry/dependabot/npm_and_ya
|\
| * a5dc439 - Bump lodash from 4.17.11 to 4.17.14 (5 weeks ago) <dependabot[
|/
*       d91d87d - Merge pull request #7 from michaellperry/dependabot/npm_and_ya
```

Figure 1-8. *A history of commits in a Git repository*

The current state of the source code can be constructed from the commits without reconnecting to the remote host, thus granting each node autonomy. The developer works disconnected from the server to construct a new set of commits on their own. While they work, they are not connecting to the remote host to lock files or check for the most recent changes. They are working in complete isolation; no round trip to the server is required.

When a conflict occurs, as it often does in source code, the developer finds within their local repository all of the information necessary to resolve it. They have the identity of the collaborators (possibly even themselves) involved in the conflict. They know exactly the context of the change—what the code looked like at the time it was modified.

And they even have from the commit comments some clue as to the intent of each programmer.

Based on all of this information, the developer can resolve the conflict themselves. They don't need to involve the server. In fact, because of the nature of Git branches, they can choose to let the conflict stand as long as they please. There is no immediate need to for the conflict to be resolved before work can continue. But when a resolution *is* made, it is recorded as another commit. That commit becomes part of the history so that all parties involved can see that the conflict has been resolved and understand the effect of this resolution.
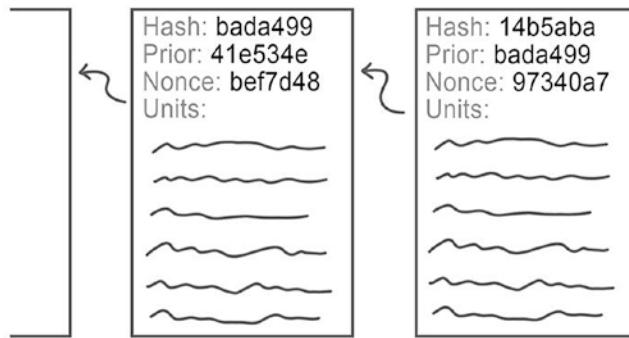
This mode of working is only possible because every commit is immutable. Every developer who has the same commit knows that their copy is just as good as any other. No other developer can modify the contents or the identity of a commit; all they can do is create new ones.

Blockchain

Blockchains store information as singular units (transactions, contracts, digital assets) aggregated into *blocks*. A block is simply a collection of these units surrounded by an envelope of metadata. Each blockchain defines its own block data structure, but they all share some common fields.

- A random number called a *nonce*

- A reference to the previous block

- A hash of the block's contents (including the nonce and the reference)

As a result, the current block is just the most recent collection of transactional units. It points back to the previous block, which points back again forming a chain. This chain, as depicted in Figure 1-9, represents the entire history of transactions since inception.

Figure 1-9. *Each block in the chain contains a hash and a reference to the previous block*

The immutability of a block is a consequence of the hash that is its identity. If the contents of a block were to change, the new hash would be different. Cryptographically strong hash functions are used so that it would be difficult to modify a block in such a way as to leave the hash undisturbed. And when I say "difficult" here, I use the term in the way that cryptographers use it. We are not allowed to say "impossible."

Because a block's hash (and hence its identity) includes the hash of its predecessor, any change to a block will ripple through all subsequent blocks and produce a new alternate history. Such tampering would be easily detected. Every node sees the same copy of every block. This is both the enabling characteristic and the most valuable feature of a blockchain. On the one hand, this enables immediate detection of tampering, and on the other hand it provides the benefit of a shared public auditable ledger.

Blockchains have a dark side, which many analysts (myself included) believe will be their undoing. In order to ensure that all nodes agree upon the same history of blocks, most blockchains feature a *proof of work*. This is an algorithm that is slow to run, yet fast to verify. For example, the blockchain could demand that the first several bits of a block's hash be 0. Because nodes have to waste CPU cycles computing proof of work, the rate of blocks added to the chain is held constant. The cost of falsifying history—both in terms of electricity and computing hardware—is greater than the value that could be derived. Unfortunately, this means that the cost of *legitimate* use of the blockchain is also very high relative to its value.

While proof of work may turn out to be the Achilles heel of blockchains, the benefit that it derives from *immutability* is solid. Only by ensuring that every participant has the same indelible copy of the ledger can this system provide the benefits of shared auditable history.

Docker

Docker goes beyond the capabilities of virtual machines because it organizes images in *layers*. A layer is an immutable portion of a file system with a reference to the layer below. An *image* is really nothing more than a reference to the topmost layer. For this reason, the other layers are also referred to as *intermediate images*.

For Docker to execute a workload, it creates a *container*. A container is a running instance of an execution environment, complete with its own simulated file system. When a container starts up, Docker allocates to it a special writable layer, which in turn points to the topmost layer of the image. This layer is initially empty.

At runtime, when a container reads from the disk, Docker will forward that read operation to the writable layer. As this layer is initially empty, the read request will fall through to the topmost layer of the image. If the requested data is in that layer, then it will be returned. Otherwise, it will move down to the layer below.

When a Docker container writes to the disk, it only modifies the writable layer. This layer is special, in that it is not shared among any of the other Docker containers and it is not persisted beyond the lifetime of its container. Any information written to that layer is lost when the container is deleted. Even if the container "overwrote" parts of the operating system, the lower layer containing that source data is unaffected.

The identity of a layer is a hash, similar to the identity of a Git commit or a blockchain block. The difference, however, is that it is a hash not of the contents but of the command that created it (including any source files in the case of an ADD or COPY command). The resulting structure is shown in Figure 1-10. To build an image, Docker starts with the base image: a name used in a registry to identify the hash of an existing layer. Starting from this base layer, Docker then scans the commands one by one and computes the hash of the resulting intermediate image. If that image is already in the repository, then it is retrieved rather than being reconstructed.

```
IMAGE                CREATED BY
3d53cdc8aa80         /bin/sh -c #(nop)  CMD ["pm2-runtime" "npm" …
8e48995a6b00         /bin/sh -c #(nop)   USER node
e022d570f832         /bin/sh -c npm install pm2 -g
277f48e981fd         /bin/sh -c npm install --production
77178a97af83         /bin/sh -c #(nop) COPY multi:56a9de3a81cb55e…
```

Figure 1-10. *Each Docker image is created by applying a command to the previous image*

Immutable layers allow a single Docker host to run several containers from the same or related images without duplicating the entire operating system for each one. Virtual machines cannot share images because those images are mutable. If one machine modifies an image, that change would become visible to the other VMs. But Docker can get away with sharing layers because those layers will not be modified. It's the structural sharing of layers that allows Docker to support orchestrators and meshes, coordinating several interconnected running containers all formed from a shared repository of images.

Each of these systems has harnessed the power of immutability to provide their own distinct advantages. Just as Pat Helland pointed out in "Immutability Changes Everything," this one idea is a recurring theme appearing at several layers of the technology stack and across many problem domains. As you learn to model business problems based on immutability, you will start to enjoy the advantages of a reliable audit history, just like blockchain. And as you learn to implement immutable data structures within your mobile apps and microservices, you will benefit from the same autonomy found in Git. Let the journey begin.

Forms of Immutable Architecture

There are consequences to designing a system using only immutable records. Some of them are the advantages that we've already explored: reliable communications, reduced blocking, increased autonomy, and improved auditability. Other consequences are less desirable. Many of them simply require a shift in thinking, while others demand entirely new solutions. As you adopt immutability into your application design, you will need to recognize how the architecture must change in response.

The trade-offs requiring shift to immutability have led to the emergence of different architectural styles. In this chapter, we will examine three of those styles: Event Sourcing (ES), Asynchronous Model View Update, and Historical Modeling. All three share the idea that state evolves from historical records. Where they diverge is in the ordering of those records. The first two styles assume that records can be viewed in *sequence*. They expect to be able to enumerate records in order. The third arises from the idea that historical records may be *partially* ordered. It does not allow enumeration. Instead, it trades that capability away to achieve some valuable results.

After this chapter, the remainder of the book will focus on the third style: Historical Modeling. But it will be important to put that choice into context. Each architectural decision is a trade-off among competing values. Let's explore all three architectures to get a better understanding of what those trade-offs will be. We'll begin with the concepts that they all have in common.

Deriving State from History

The art of immutable architecture is finding a balance. On the one hand, there is the recognition that immutable data structures offer significant advantages in parallel and

29

distributed computing. On the other hand, there is the recognition that the world that our systems model is full of mutation. Each of the architectures we will study finds this balance in their own way.

Here, it helps to give things distinct and meaningful names. We will call the things that change *objects*, and the things that do not change *records*. These choices are not arbitrary. They are based on concepts that humans have invented to organize information both with and without computers.

Historical Records

Let's go back in time to a world before computers. How was business transacted in this world? Rather than updating the current state of the world in a large database, information was recorded and shared in the form of documents.

Suppose a customer places an order for ten widgets. This decision is captured as a purchase order. The purchase order references the two parties: the buyer and the seller. It also references the product—widgets—by the catalog number assigned by the seller. The result would be a document similar to the one shown in Figure 2-1.

Purchase Order

Buyer Address City, State Zip	Seller Address City, State Zip			
Num	**Product**	**Qty**	**Price**	**Total**
101	Widget	10	19.95	199.50

Figure 2-1. A purchase order records the intent of a customer to purchase products from a vendor

Building Upon the Past

The purchase order is a historical record. It is an immutable document that records a decision as it was at a certain point in time. It makes reference to two parties: the buyer and the seller. Neither party can change the purchase order itself. They can only amend this document with another one.

The buyer and the seller are distinct legal entities. These entities were created with their own set of immutable records: documents that were filed with their respective regulating bodies as articles of incorporation. These documents were created well in advance of the purchase order.

The purchase order refers to a few other decisions that came before. For example, it refers to a catalog number. This is the result of a decision to publish the widget in a catalog of available products with a listed price. The catalog, once published, is not changed. It is only amended by publishing subsequent catalogs with ever-evolving lists products and prices.

Evolution of Understanding

Future documents will in turn refer back to this purchase order. Once the seller receives a copy of this document, they will create an invoice—a new document that requests payment. The buyer will write a document in response, a check that requests that the bank transfer funds to the seller. The seller will create a packing slip, which documents the items to be included within a shipment. The carrier will issue a bill of lading to the buyer, documenting delivery of the goods.

Historical records evolve our collective understanding over time. Each one is in itself immutable. But our view of the world changes as more documents are published.

Mutable Objects

The invention of the computer has greatly scaled up our capacity for handling transactions. But it has also subtly changed the way in which we think about what is true. Before computers, we each had to carry a bank book to calculate our account balance. Now, we can see it immediately on our cell phones. It used to be understood that the idea of "balance" was a derived one and differed based on one's knowledge of which checks had been cashed. Now, at least intuitively, "balance" has become an intrinsic property of an "account" object.

Like functional, modular, and structured programming before it, *object orientation* was one of the great advancements in software modeling. It originated from the observation that software systems exist to model the real world. It sought to first understand the behavior of the objects being modeled and then provide patterns and templates for implementation.

At its core, object-oriented programming assumes that at least *some* objects are mutable. It leaves room for immutable objects, but mainly describes the behavior and evolution of mutable objects. This assumption appears most notably in the concept of identity.

Identity

Of the fundamental elements of object orientation that James Rumbaugh defined in 1991, the three we frequently talk about are encapsulation, inheritance, and polymorphism. The one that we fail to mention is *identity*. Rumbaugh defines identity as a distinguishing property independent of identifying attribute. In object-oriented modeling, even if two objects have the same properties, they are different objects. Acting on one will not affect the other.

> [T]wo objects are distinct even if all their attribute values (such as name and size) are identical.[1]

When we translate that idea to object-oriented languages such as Java or C++, the concept of identity is mapped to a location in memory. An object becomes a block of memory allocated to store its current state. The identity of the object is the address of that memory. That is why sharing the identity of an object in C++ is achieved by "passing a pointer."

Memory addresses obey the rules of object-oriented identity. They represent uniqueness absent any identifying attribute. At any point in time, the state of two objects might be exactly the same. The chunks of memory at those two addresses might be byte-for-byte equal. But changing the memory at one location will have no effect on the memory at the other. A consumer of one object will perceive no change in behavior based on a modification of the other.

As we move further away from a single thread in a lone process on an isolated machine, this implementation decision starts to show its faults. Moving from one thread to many, we must introduce locking to protect the integrity of the data structure against simultaneous mutation. Moving from one process to many, we must map the shared object into independent memory spaces. And going beyond the boundaries of a single

---

[1]James Rumbaugh, et al. Object-Oriented Modeling and Design. 1991 Prentice-Hall, Inc. ISBN 0-13-629841-9.

machine, the concept of a pointer loses all meaning. Other forms of identity need to be introduced to compensate.

Evolution of State

Rumbaugh's definition of identity solves a problem for objects that can change state over time. But if we introduce the concept of immutability, it becomes less valuable. The reason for an object to have intrinsic identity is so that it can provide consistent, meaningful behavior as it changes over time. If I take a bite out of one apple, another remains whole. It would be a very strange world indeed if a bite appeared in *your* apple, or if I returned later to find mine completely restored.

Rumbaugh's identity is a recognition that objects in the real world change state in response to stimulus. They remember that state. Their observable behavior is based on their current state. For those changes in behavior to make sense in any model of the real world, objects in the model have to have intrinsic identity.

Projections

Our goal now is to use immutable records to model mutable objects. The records clearly are not the objects themselves. That would be insufficient, as records would not allow for the mutability that objects expect to have. Instead, the records must in some way represent *changes* to the objects. The immutable records *are* the mutations of the objects.

To achieve this goal, we will treat immutable records as *observed* state. They represent things that we actually saw and recorded. Objects, on the other hand, are *derived* state. They represent our interpretation of those observations and can change as new observations are made.

Two Kinds of State

Imagine a spreadsheet. In each cell, you can enter one of two things. Either you can enter a value, or you can enter a formula. A value represents some basic measurement, an observation of the system you are modeling. A formula, on the other hand, derives a new value from those observations. Formulas represent derived state.

As a mathematician, my favorite analog of this idea is a function. We will often write y as a function of x, thereby producing a plot. You can draw a vertical line anywhere on

that plot and you will hit only one point. The same cannot necessarily be said for any horizontal line. We say that *x* represents the *independent* variable and *y* the *dependent* variable. You get to choose *x*, but *y* is calculated by the function. Independent variables are observed, and dependent variables are derived.

In software, we have other names for this phenomenon. Derived state is sometimes referred to as a *projection* of the observed state. A pure function takes a value as an input and produces an output with no side effects. The output is deterministic, depending only upon the input. Where the input is observed state, the output is derived from that input. It is a projection of that observed state.

Derived state also appears in software as a *view model*. While a model is an object that generally supports the problem domain, a view model maps more specifically to a view. It projects the model for the purpose of display on the view. In data-binding frameworks, changes made to the view model appear directly on the view.

Projecting Objects

No matter what you call it—formulas, dependent variables, projections, or view models—derived state is a deterministic transformation of observed state. It adds no information to the system; it only presents the information that's already there in a different way. In mathematics, we say that it adds no new degrees of freedom to the system. In software, you might say that the view model is backed by the model. The important point is that the user gets to change observed state directly. They can only see the results indirectly projected onto the derived state.

In an immutable architecture, the historical records are observed state. The user gets to create new records directly through their actions. Those records capture decisions that the user has made.

The objects, on the other hand, are merely projections. They are ephemeral. The user does not get to set the state of an object. They can only see those objects change as a result of new historical records. Every one of these architectures has their own way of calculating that projection.

Event Sourcing

Historical records are the observed state of an immutable architecture. They represent past decisions. You could call these past decisions "events" and demand that they are the sole source of truth. That is the origin of the term *event sourcing* (ES).

While the term "event sourcing" could arguably be applied to any architecture that reconstructs state from a history of immutable records, the practice is a bit more specific. As Martin Fowler described it in 2009

> The fundamental idea of Event Sourcing is that of ensuring every change to the state of an application is captured in an event object, and that these event objects are themselves stored in the *sequence they were applied* for the same lifetime as the application state itself.[2]

The emphasis on "sequence they were applied" is mine. The idea of a sequence does not necessarily follow from the requirements of immutable historical records. But it is a reasonable assumption, and one shared by all implementations of ES that I have seen. I therefore consider *sequence* a defining characteristic of event sourcing.

Generating Events

In an event-sourced application, the user interacts (through a UI and possibly an API) with a domain model. The domain model does not respond immediately to the request. Instead, it validates the request and generates an event. The event is an immutable record of the user's intent. It is named and interpreted as a past-tense statement, as in "this thing happened": `OrderSubmitted`, `PlayerRegistered`, and `ResidentMoved`, for example. The naming convention reflects the truth that an event, once generated, cannot be ignored. Its effect might just be different from what the user intended.

By interacting with the domain model, the user experiences the application as if it followed a traditional object-oriented paradigm. They get the impression that objects have properties that change over time and that their actions directly cause that change. The application hides the fact that the object model is both a generator and a projection of a sequence of events.

---

[2]Martin Fowler. https://martinfowler.com/eaaDev/EventSourcing.html

The advantages that ES provides over a traditional object model begin with the same ones that we've already identified for all immutable architectures: increased scalability and auditability. In addition, they boast the ability to rebuild objects entirely from the stream of events. When a defect is fixed or a feature is added, the application can discard any cached versions of the domain model and reconstruct them using the new code. It also allows an event-sourced application to go back in time and replay only part of a sequence, seeing an object as it appeared in the past. This provides the user of the application with a powerful ability to perform temporal analysis.

Practitioners will often pair event sourcing with both Command Query Responsibility Segregation (CQRS) and Domain-Driven Design (DDD). This pairing is not a requirement for ES, nor are the implementations all in agreement how it is achieved. Some choose to pair just CQRS with ES, or just DDD with ES. This architectural decision affects how an application projects immutable records into mutable objects.

CQRS

Command Query Responsibility Segregation extends the object-oriented principle of Command Query Separation (CQS). Bertrand Meyer defines commands and queries as kinds of methods. He distinguishes them as follows:

> A command serves to modify objects, a query to return information about objects.[3]

Where CQS draws a line between methods, CQRS extends that line to segregate objects. In obedience of the single responsibility principle,[4] some objects are responsible for issuing commands and others for issuing queries.

In CQRS, *commands* are responsible for changing the state of the system. These are distinct from *queries*, which request information about current state. Commands and queries follow separate paths and often interact with different architectural components. Commands are often asynchronous, while queries are usually synchronous. In many implementations, they operate against different data stores.

---

[3]Bertrand Meyer. Object Oriented Software Construction, Second Edition. Prentice Hall. 1997. ISBN: 0-13-429155-4.

[4]Robert C. Martin. Agile Software Development, Principles, Patterns, and Practices. Prentice Hall. 2003. ISBN 978-0135974445.

When paired with ES, commands are further distinguished from *events*. Whereas a command is expressed as an imperative statement, an event is a past-tense statement. The command `SubmitOrder` results in the event `OrderSubmitted`. The command instructs the system of record to perform the user's intent. The event, on the other hand, is *produced* by the system of record and must be honored. The system of record is responsible for first validating and authorizing the command. It may choose to fail or ignore the message.

When the command is sent asynchronously, this further removes the user from the effect of their actions. Applications using the CQRS/ES architectural style with asynchronous commands will often expose an *eventually consistent* interaction to the user, making it clear to the user that their request will be processed at a later time.

DDD

When object-oriented programming was first introduced, it held the promise that objects in software could model objects in the world. As it was adopted into enterprise software development, objects started to model the world less and the computer more. Eric Evans refocused object-oriented analysis and design onto the problem domain with his 2004 book Domain-Driven Design.[5]

While the book offers advice on many phases of the software development process, we will focus only on the technical aspects. In particular, we will focus on the ontology and relationship of different kinds of objects.
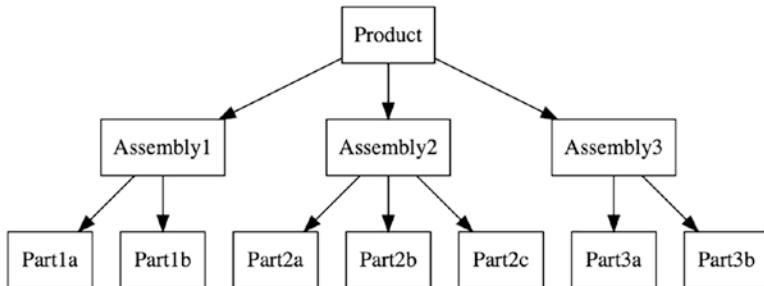
DDD recognizes two kinds of objects: entities and value types. An *entity* is an object that has identity. As we have already seen, object-oriented identity affords the object the ability to change over time. In contrast, a *value type* has no identity and is therefore immutable.

A `Customer`, for example, would be an entity, since it has identity and can change over time. A customer might have a mutable property `ShippingAddress`, the data type of which would be `Address`. An address in turn would have several different properties, like `Street`, `City`, `Country`, and others. But the shipping address is managed as a single unit, not as separate properties on `Customer`. The `Address` data type is a value type and has no identity of its own.

---

[5]Eric Evans. Domain Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional. 2004. ISBN: 0-321-12521-5.

Entities in DDD are organized within hierarchies called *aggregates*. An aggregate is a parent–child relationship. At the top of this hierarchy is the *aggregate root*. For example, an `Assembly` in one domain might be an aggregate containing many `Parts`. Several `Assemblys` are collected under one aggregate root called `Product`. The aggregate appears in Figure 2-2.

Figure 2-2. *An aggregate root owns its child entities*

As a rule, entities are not addressed outside of their aggregate root. Within the domain that we described previously, it would make little sense to talk about a `Part` without first identifying its `Product`. The `Product` is the aggregate root and therefore the entry point for all external references.

Event Sourcing replays past events in order to rebuild objects in the model. But that doesn't mean that the system reconstructs the entire object model on every query. That would not be an efficient way to handle a query of a single object. Most of the events in a model's history will have no effect on that object. So instead, event sourced applications break that history into independent streams. Each stream affects only a subset of the domain model. When ES is combined with DDD, that subset is the aggregate root.

A query in a DDD/ES application will identify an aggregate root. The application loads the stream of events for that given identity. The stream contains all events that affected the root or any other entity within the aggregate. Individual entities are not reconstructed from their own history; they arise from the history of their root entity.

Taking a Functional View

So far we have been concentrating on object-oriented analysis to describe the effect of events. But there is an equally valid interpretation using the ideas of functional programming. Consider a function that computes the state of a system after applying an event:

$$f(state_n, event) \rightarrow state_{n+1}$$

By "state of the system," we could be talking about a single object, an aggregate root, or even the entire object model. Practically speaking, you will want to choose a smaller boundary. But analytically, the outcome is the same.

The advantage of modeling system state using this function is that state—like events—becomes immutable. The function is a *pure* function: it does not cause side effects. More specifically, it does not change the state of its inputs. The function does not *modify* the incoming state; it returns a new state *derived* from the incoming state.

In functional programming, it is not uncommon to define *higher-order functions*. These are functions that take functions as parameters. One such function is *left-fold* (sometimes abbreviated as *foldl*). *Foldl* takes a binary operation—a function taking two parameters—and applies that operation over every element of a list.

For example, given the binary operation +, the starting point *0*, and a list of numbers, *foldl* will compute the sum.

$$foldl(+, 0, [3, 17, 2]) = 22$$

You can think about event sourcing in these functional terms. The binary operation is the function described previously, which takes the current state and returns the state after an event is applied. The starting point is the initial state of the system. And the list is the sequence of events. Greg Young uses functional constructs to describe event sourcing:

> When we talk about Event Sourcing, current state is a left-fold of previous behaviors.[6]

---

[6]Greg Young. http://codebetter.com/gregyoung/2013/02/13/projections-1-the-theory/

Commutative and Idempotent Events

We will find that the commutative and idempotent properties are useful in distributed systems. The commutative property allows us to apply an operation out of order and get the same result. The idempotent property allows us to repeat the operation without further effect. Since event sourcing is based on a sequence of operations, it is sensitive to both order and duplication. It is up to the application developer to ensure that order is preserved and duplicates are prevented.

The + operator is commutative: $a+b = b+a$. But in general, binary operations don't have to commute. Unless you are very careful in your selection of *f*, *foldl* will be sensitive to the order of items in the list. Event sourcing, therefore, is non-commutative by default. If a system is to respond consistently with respect to out-of-order events, then it is incumbent upon the developer to prove that the event-application function commutes when necessary. To be more precise, if the event-application function satisfies the following equation, then the events commute:

$$f(f(x, e_1), e_2) = f(f(x, e_2), e_1)$$

Just as not all binary operators commute, not all are *idempotent*. The + operator is one such example: $a+a \neq a$ (except in the special case that $a = 0$). And so a left-fold over a sequence containing duplicate numbers will inflate the result. If your distributed system allows duplicate events into the stream, then it is up to you to prove the following equation:

$$f(f(x, e), e) = f(x, e)$$

You can handle order and duplication in front of the event stream or behind it. Either prevent out-of-order and duplicate events from entering the stream, or carefully choose your event-application function.

Asynchronous Model View Update

The Elm programming language takes the functional view of a history of events quite literally. This language compiles to JavaScript and runs in the browser. It generates HTML from a source model. A pure function produces successive versions of the model as it handles messages. The pattern on which Elm is based is called *Model View Update*.

Inspired by Elm, Facebook created a suite of tools that extend this pattern to the server. The first of those tools was React, a front-end library for JavaScript that projects

a model into HTML. The next was Flux, a unidirectional data flow application design pattern. This defined the way that Facebook designed web and mobile apps. Redux was an implementation of Flux developed outside of Facebook by Dan Abramov. Dan was subsequently brought into Facebook to continue work on Redux, React, and the architecture in general. Finally, there is the back-end architecture, only parts of which are currently open sourced, upon which Facebook develops their APIs.

The Update Loop

When React and Redux are used together, they form a loop. This loop is the core engine of the Model View Update pattern. React transforms the model into the view, and Redux dispatches actions to update the model.

The *model* is simply a data structure. As an application developer, you define the data structure that you need. Since React targets single-page web applications, the model is typically the collection of state that a single user is viewing and manipulating within a page.
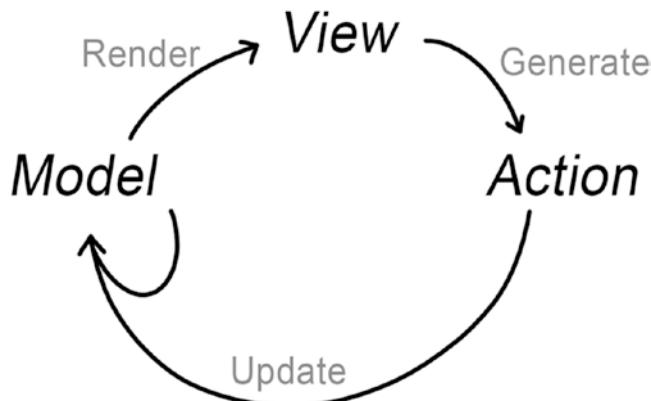
The *view* is what the user sees; in React it is HTML. A function called *render* transforms the model into the view. This function runs first after the model is loaded and then every time a new version of the model is produced. The results of subsequent executions are compared to determine what has actually changed and to update the Document Object Model (DOM). The developer does not have to be concerned with change tracking.

$$render(model) \rightarrow view$$

Finally, *update* is a function that computes the next version of the model. This is a pure function, so it does not modify the model. Instead, it produces a model as it would look after an *action* is applied.
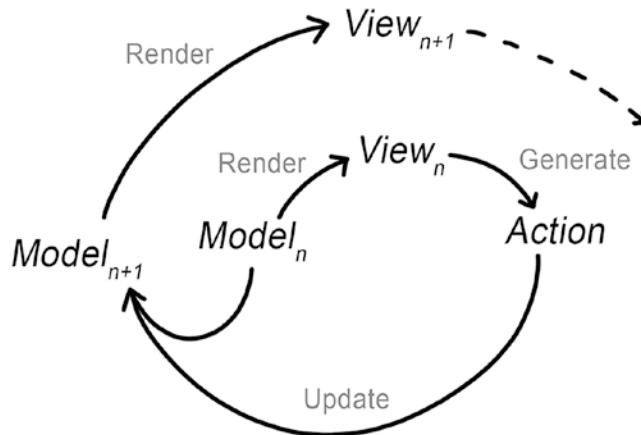
$$update(model_n, action) \rightarrow model_{n+1}$$

A Model View Update application begins with loading a model. It then enters a tight loop in which user interaction generates an action. The update function handles the action producing a new version of the model. The render function turns that new model into a new view. The framework compares the two versions of the view to determine what to change in the browser. The user is presented with the updated user interface, and the loop continues as shown in Figure 2-3.

Figure 2-3. *The model-view-update loop found in React and Redux*

Being a pure function, *update* does not modify the model. It produces a new version of the model, which is then rendered to produce a new version of the view. So it is more accurate to depict this loop not as a cycle, but as a spiral as in Figure 2-4.
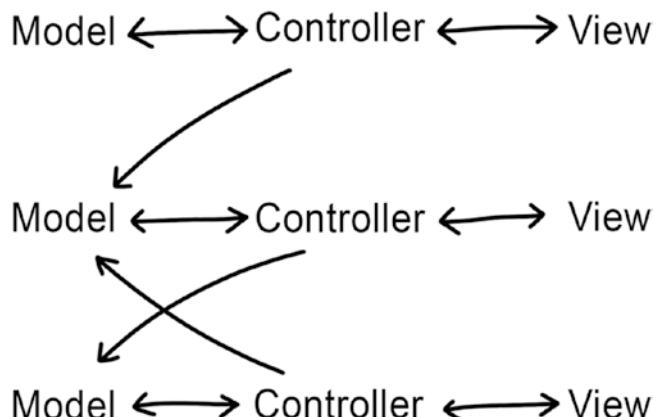
Figure 2-4. *Each iteration produces a new version of the model*

This image makes it more clear that there are no circular dependencies. Each iteration produces a new set of objects. The old objects are still available to support optimizations, such as minimizing DOM manipulation.

Unidirectional Data Flow

Flux and Redux were developed as a reaction to problems found in the *Model View Controller* (MVC) pattern. In MVC, a controller responds to changes in the model by updating the views. It also responds to user input in the view by updating the model. The controller coordinates data flow in two directions: both in from the user and outward from the application.

Bidirectional data flow is very simple to start with. With just a handful of controllers, following the thread of execution from view to model and back again is not difficult. But as more controllers are added, the number of paths increases super-linearly, as illustrated in Figure 2-5. It is difficult to know whether a new feature is going to produce a new edge case that causes cascading updates or circular dependencies.

Figure 2-5. *Several controllers coordinating with one another to update their models*

With unidirectional data flow, on the other hand, there is no chance of cascading updates. An action produces an update to state, which produces an update to the view. The view cannot respond to one action by producing another action. There is no possibility of cycles or runaway updates.
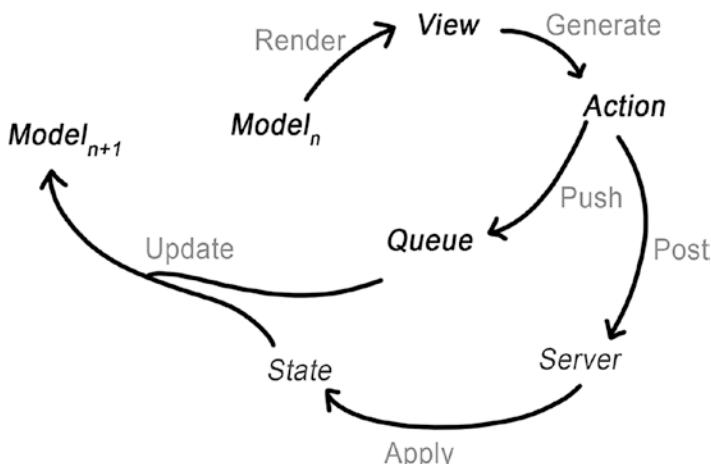
Unidirectional data flow also supports better unit testing. Start with a given state. When an action is handled, then the handler produces a new expected state. Initializing the state, applying the action, and verifying the resulting state are all easy to automate. The operations that are hard to automate—verifying that the view is correctly rendered and that user interaction is properly interpreted—are marginalized. Rendering a view is

simply a function on the model, and user input only produces an action. Unidirectional data flow minimizes the manual testing surface area.

Immutable App Architecture

Model View Update as practiced by React and Redux is only half of the picture. The other half happens on the server. While Redux has the luxury of operating on an in-memory store, when a mobile application communicates with its host server, the current state of the model is not always available. Lee Byron presented Facebook's solution at Render 2016.[7] He called it "Immutable App Architecture." But since that name is similar to the general term "immutable architecture" as I've been using it, I will call the pattern he presented *Asynchronous Model View Update*.

The pattern begins just as before with a render function projecting a model into a view. Also as before, the user's interactions with the view produce actions. At this point, however, the pattern diverges. On the client, actions are pushed to a *queue*. On the server, actions are applied to the *true state*. The update function combines the last known true state with all of the actions still in the queue. The resulting loop is shown in Figure 2-6.

Figure 2-6. *Asynchronous Model View Update*

---

[7]Lee Byron. Immutable User Interfaces. Render 2016. https://vimeo.com/166790294

The next iteration of the model is not based on the previous iteration. Instead, the client goes back to the last state fetched from the server. If a new state is received from the server asynchronously, then it becomes the new true state. Any actions that are represented within that state are removed from the queue. The state is known to be true; the queue contains actions that optimistically might become true.

The advantage of this architecture for Facebook was that they could more easily reason about new features. To add a feature, a developer needed only to add new actions and propagate them out toward the view. If those actions had an impact on other views, they would simply add the desired effect to the update function. The mobile application could work quickly, even while on a slow network. The effect of a user's interaction would be immediately visible, without waiting on a round trip to the server.

The Asynchronous Model View Update architecture optimistically interprets a series of actions. User actions are validated on-device with the expectation that most of them will succeed on the server. It is assumed that no other actions will intervene and that the result of executing the actions on the server will be the same as on the client. When this optimistic assumption is found to be false, the architecture simply discards the locally computed state and takes the server's version.

Historical Modeling

The immutable architectures that we just examined both make a distinction between immutable historical records and a mutable object model. They also assume that historical events occurred within a fully ordered sequence. But neither of these assumptions necessarily follow from the idea of using immutable records as the source of truth. If we model a system as a collection of related historical facts, we find that we can dispense with the mutable object model altogether and that facts don't necessarily have to occur in a sequence.

Let's begin with a slight change in terminology. Instead of referring to historical records as *events* or *actions*, let us call them *facts*. The reason for the name change is that facts obey a set of rules that do not necessarily apply to events in Event Sourcing, or actions in Asynchronous Model View Update. In particular, facts are partially ordered.

Partial Order

The term "partial order" comes from mathematics, and is distinguished from the term "full order." Start with a set of objects, be they numbers, words, science papers, data structures, what have you. Define a comparison operation that tells you whether one object comes before another. We will typically use the less than symbol (<) to represent this operation. If for any pair of objects in the set, we can use < to put one before the other, then the set is fully ordered. If we can only do that for *some* of the pairs, then the set is partially ordered.

As an example, consider the set of counting numbers and the familiar definition of <. *1<3* and *3<17*. In fact, for any pair of distinct counting numbers, you can use this operator to put one before the other. When *a* and *b* are different, either *a<b* or *b<a*. The set is fully ordered under the < operator.

But now let's change the definition of < and see what happens. Instead of the familiar "less than," let's say that *a<b* if *a* is a proper factor of *b*. That is to say, if *a* and *b* are distinct and dividing *b* by *a* leaves no remainder, then *a<b*. Now we can compare pairs of numbers and see what happens. *3<15* because 3 is a proper factor of 15. But we cannot say that *3<8* nor that *8<3*. Neither one is a factor of the other. And so the counting numbers are partially ordered under the operation of proper factor.

Whether we are talking about a total order or a partial order, the comparison operator that we choose must have a couple of useful properties. First, it must be transitive.

$a<b$ and $b<c \Rightarrow a<c$

It must also be non-reflexive. That is to say that an object does no "come before" itself.

$a \nless a$

Finally, the comparison operation must be unidirectional. That means that an object cannot come both before and after another one. More formally, this is written as follows:

$a<b \Rightarrow b \nless a$

All of these properties hold for a comparison operator that imposes either a partial order or a total order. The thing that distinguishes them is whether for any given pair one must come before the other. In a total order, if two distinct objects are not ordered one way, they must be ordered the other way:

$a \nless b \Rightarrow b<a$ (for *a* and *b* distinct)

This is not the case for a partial order. A partial order will allow both $a \nless b$ and $b \nless a$.

This does not only apply to numbers. English words are totally ordered alphabetically, as illustrated in a dictionary. They are partially ordered under the *contains* operator, as in "catalog" *contains* "cat". Total and partial orders can be found for many sets. This includes the set of historical records, or *facts*.

Predecessors

The way in which Historical Modeling puts facts into a partial order is to identify *predecessors*. For each fact, a historical model makes explicit which other facts must have come before. These aren't simply the list of all other facts that have occurred earlier in time: that would put facts into a sequence—a total order. Instead, predecessors are facts that must have happened before in order to make the current fact make sense.

If we return to our purchase order example, we can see a few predecessors in evidence. A purchase order is a document of the decision by a buyer to purchase items from a seller. The example purchase order from earlier in the chapter appears again in Figure 2-7.

Purchase Order
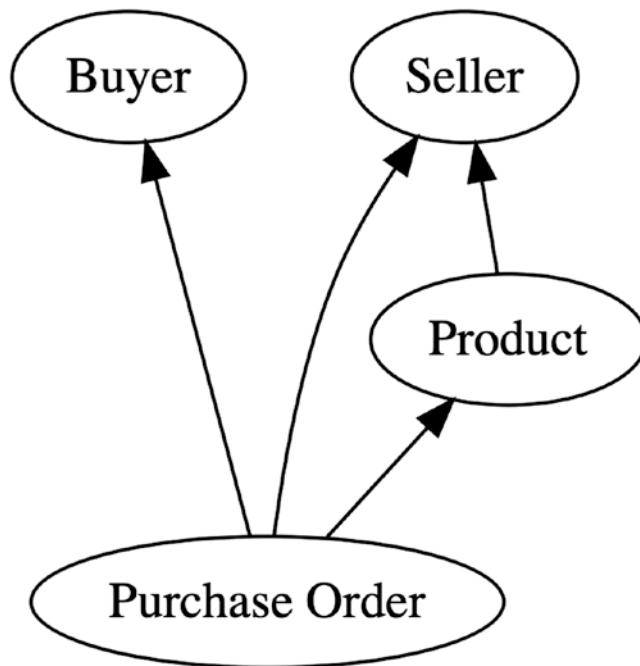
Buyer Address City, State Zip	Seller Address City, State Zip			
Num	**Product**	**Qty**	**Price**	**Total**
101	Widget	10	19.95	199.50

Figure 2-7. *A purchase order from a buyer to a seller*

The purchase order is a *fact*. It is a historical record that documents a decision. It is immutable: neither party can change the purchase order itself. They can only amend this document with another one.

The purchase order fact refers to a few other facts that came before. It refers to the buyer and the seller as distinct legal entities. These entities were created with their own set of historical facts—documents that were filed with their respective regulating bodies as articles of incorporation. These facts were created well in advance of the purchase order.

The purchase order fact also refers to product. This is a historical fact that the widget was published in a catalog of available products with a listed price. The catalog, once published, is not changed. It is only amended by publishing subsequent catalogs adding and removing products and changing their prices. The relationship of the purchase order to all of its predecessors is shown in Figure 2-8.
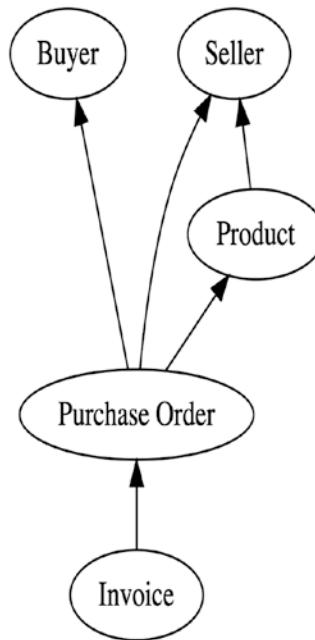
Figure 2-8. *Buyer, seller, and product are all predecessors of purchase order*

In this model, there is no predecessor relationship between purchase orders. It does not record that one purchase order was submitted earlier in time than another one. Predecessors are not simply facts that occurred earlier in time; they are prerequisites: things that must have been true for this fact to make sense.

Successors

It is useful to talk about the opposite direction of the predecessor relationship. A fact that refers to another one is its *successor*. Successors help us to evolve our understanding of a system over time. We cannot change a historical fact, but we can create successors.

Let's continue the story of the buyer and the seller. The seller receives a copy of the purchase order and then sends the buyer an invoice. The invoice is another historical fact. The predecessor of this fact is the purchase order. The successor of the purchase order is the invoice, as shown in Figure 2-9.

Figure 2-9. *Invoice is the successor of a purchase order*

The presence of a successor does not change the predecessor. Issuing an invoice does not alter the historical record that is the purchase order. However, the successor changes our *interpretation* of the predecessor. When we see the invoice, we now know that the state of the purchase order has changed. We know that it has been invoiced, and it would be incorrect to issue a second one.

It is important to recognize that there is no mechanism within a historical model to *prevent* the creation of additional successors. The model itself must allow for multiple invoices to the same purchase order. If we carefully control *who* can create those invoices, and on what machine, then we can avoid this situation in any practical scenario. But the model itself has no ability to lock the purchase order, or to prefer one invoice over another.
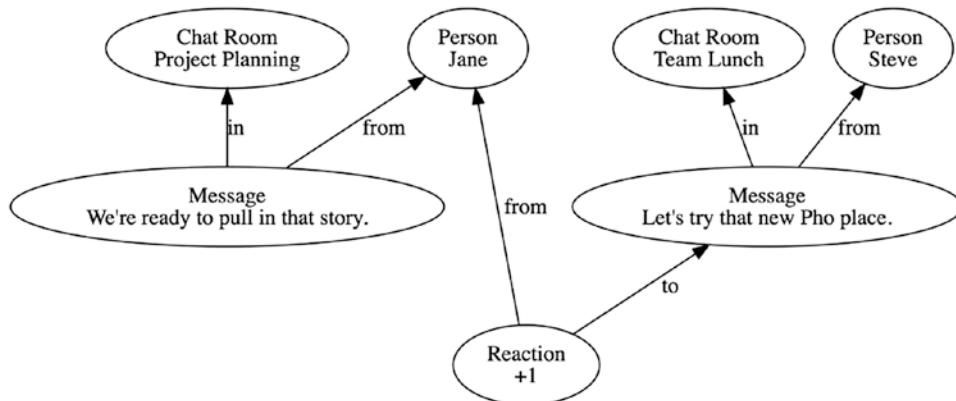
A fact does not know about its successors. New successors are added over time. To fully understand the state of a fact, we must query the historical model to discover if new

successors have been created. Current state is not a projection of historical facts into mutable objects; it is simply the collection of known successors.

Immutable Graphs

Like an event, a historical fact is immutable. But unlike an event, a fact refers to its predecessors. Taken together, these properties have interesting consequences.

The predecessors to which a fact refers to are themselves immutable facts. Those facts can in turn have predecessors. This produces a structure known as a *directed graph*. Each vertex in this structure is a fact, and each edge is a predecessor relationship. This relationship has a direction: it points from the successor to the predecessor. We've seen examples of these graphs presented earlier as they relate to purchase orders and invoices. Another example appears in Figure 2-10.
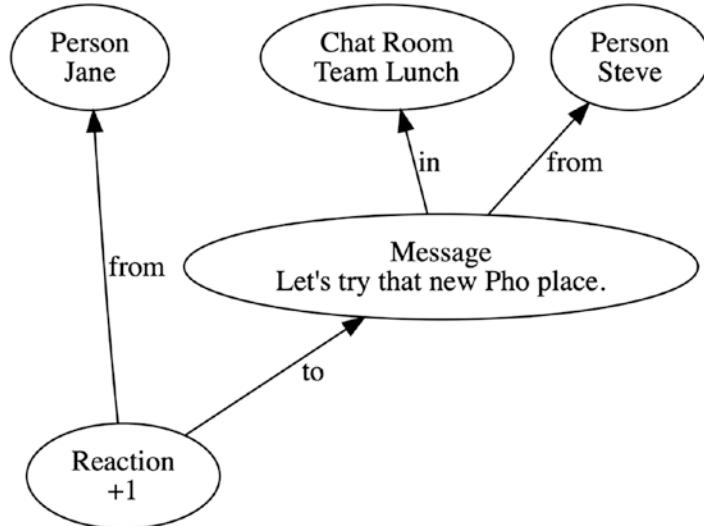
Figure 2-10. *A fact refers to its predecessors, which in turn refers to still more predecessors*

Since a fact refers to its predecessors, and the fact is immutable, it follows that a predecessor cannot be added to an existing fact. That predecessor relationship is *part* of the fact, and the fact cannot be modified. And so while it is possible to add successors to a fact, it is not possible to add predecessors. This is in keeping with our use of the predecessor relationship to define what comes before in the partial order. All predecessors must be known facts, recorded before the new one.

From any given fact, we can trace the graph along the predecessor paths. We will select a subgraph that includes the starting fact, all of its predecessors, and all of *their*

predecessors recursively. This process produces the *transitive closure* of the starting fact. If we compute the transitive closure of the reaction, we end up with the subgraph in Figure 2-11.

Figure 2-11. *The transitive closure of a fact contains every fact's predecessors*

To build the transitive closure, we started from one immutable fact and followed arrows only in a direction that cannot change. The subgraph is therefore immutable. For any given fact, the transitive closure will always be the same. Adding new successors to any of the facts in the graph will not change it. Those successors would never get added to the transitive closure.

Conversely, the transitive closure *identifies* the starting fact. There is no other fact for which the transitive closure would produce this same set. In a historical model, this is the *only* way to identify a fact. They do not have globally unique identifiers (GUIDs) or sequence numbers outside of this structure. The contents of the facts in the transitive closure are all you've got to tell one fact apart from another.

Collaboration

Machines within a distributed system can communicate by exchanging graphs of historical facts. As they do, they must be sure to send the transitive closure of each fact. They have to know that the recipient is aware of all of the predecessors at every step.

When a machine records a new piece of information—a decision that a user has made or the outcome of some business process—it does so by creating a new fact. It cannot create that fact based on predecessors of which it is not yet aware. It must either create those predecessors first, or have learned about them from its peers.

The predecessor relationship between facts captures the communication structure between machines. A successor from one machine can be seen as a response to its predecessor generated on another. When you observe the predecessor/successor relationship, you have evidence that the two machines communicated to make that happen. Conversely, when two facts are not related, then the two facts might have been created concurrently. This is the partial order of historical facts at play within distributed systems. The ambiguity of the ordering between unrelated facts leaves machines less constrained and, as we will see, better able to act autonomously.

Acyclic Graphs

The immutability of facts constrains them to know their predecessors at the time of creation. But there are two more constraints that we have to put on the system. First, we have to be able to construct the graph one fact at a time. And second, we cannot allow a fact to refer to itself as a predecessor. We must disallow both simultaneous creation and self-reference, lest we introduce cycles.

Every graph starts empty. It contains no facts. The first fact added to the graph therefore can have no predecessors. There is no existing knowledge upon which to build. The first fact is a root. A graph containing only one root has no cycles, because there are no edges.

Let time pass, and let more facts be added to the graph. Assume that the graph still contains no cycles. As I add a new fact to the graph, that fact may refer to any of the existing facts as predecessors. However, those existing facts may *not* refer to this new fact as a predecessor. I cannot change their predecessor relationships, and this new fact did not yet exist. I therefore cannot introduce a cycle by adding a single fact.

If we were to allow self-reference, then we could introduce a trivial cycle. And if we were to allow simultaneous insertion, then we might introduce two facts that have each other as predecessors. Since neither of these operations is allowed, the resulting graph of facts must not contain cycles. In mathematics, this kind of structure is known as a *directed acyclic graph*, and has many interesting properties. As we get deeper into the analytical and implementation details of historical modeling, we will take full advantage of the acyclic nature of the graph.

Timeliness

In a system based on the exchange of historical facts, not all parties will know about all facts at the same time. This is one of the greatest strengths of historical modeling, but also one of its important limitations. It is impossible to reject a fact based on the time at which you learn of it. The reason is that other parties will have learned about it earlier and would therefore have come to a different conclusion about the fact. For every party in the system to eventually reach the same conclusion, that conclusion cannot be based on timeliness.

This causes significant problems in systems that do not recognize this limitation. Several legal documents, such as tax forms, checks, and invoices, have explicit due dates or expiration dates. If the form is received after the required date, then it will not be honored. The sender must go to great lengths to prove that the document was written and transmitted on time, or suffer the consequences of a failed transaction. In such situations, the sender believes one thing—that they met the deadline—while the recipient believes something else. Only by arbitration of a central authority can these situations be resolved.

To design a system that does not rely upon a central authority, we must respect that documents will be received late. In a truly historical model, a fact is not rejected based on the time at which it was received. At best, we can record the fact that a fine was levied or an opportunity was lost due to the failure of information to arrive at a certain place by a certain time. But we cannot prove that the information did not exist somewhere else at that time. And when the fact arrives later, we must decide how we are to react to it. All parties must honor the existence of the facts, no matter when they learned about them, and draw the same conclusion. Perhaps that conclusion is that the sender still owes a fine. But timeliness alone did not determine that outcome.

Such are the rules of a historical model. They follow logically from the desire to capture the full history of a system with several parties, separated by time and space, exchanging historical facts. Those facts must be immutable. Two facts having the same transitive closure are indeed the same fact. We cannot guarantee—and therefore cannot rely upon—there being only one successor for any given fact. And we cannot change our interpretation of history based on the timeliness of our knowledge of it.

Limitations of Historical Modeling

For the remainder of the book, we will focus our attention on Historical Modeling. The other forms of immutable architecture are documented elsewhere, but Historical Modeling requires a bit more study. This choice, as we will see, offers many advantages: autonomy, scalability, and conflict resolution to name a few. But it is not without its limitations. We have already mentioned a few, but let us now explore them in better detail.

With the power of historical modeling comes some constraints. These constraints make it inappropriate to apply historical modeling to certain types of systems. In these situations, it is best to model all or part of the system statically—that is, using a method that captures current state—and integrate where appropriate. Fortunately, good integration strategies are available.

We will often find that we can pair a historical model with a *static* model. A static model, as the name implies, is based upon state. The model is mutable, centralized, and can enforce serialized access. Relational databases are good static models, as they have a long track record of supporting efficient locking.

No Central Authority

A historical model allows for decisions to be made with autonomy. Each decision is recorded in the local history and eventually shared with the rest of the system. As a result, the system cannot reject facts based on age or current state.

Decisions that were made in the past are approved locally, with only the information available at the time. No remote part of the system needs to be consulted. That decision cannot be rejected post facto.

This makes historical modeling inappropriate for parts of a system that require a central authority. For example, a conference room reservation system will need to know with certainty whether a room was available at a certain time. When a reservation is approved, the approver needs to know that no other reservation for the same room at the same time has been approved. That decision must be made by a central authority.

A historical model may be applied around the edges of a central authority, so long as that central authority itself is using a static model. The historical model can capture the fact that a request has been made. This occurs at the point of request, such as at a user's workstation or a device mounted by the door, and these facts find their way to a central authority. The historical model can also capture the fact that a request was approved.

This occurs at the central authority and moves out to the devices at the edge. But a historical model alone cannot say for certain whether a room is available at any given time. That would require that the model know that a reservation has not been approved, which is impossible given a subset of history.

To solve the problem, the system should include a central authority with a static model. The historical model records the reservation requests and approvals, but the static model determines availability. The central authority need not be a single machine; it could be a cluster of machines. As long as the members of this cluster have access to the same static model, they can act with singular authority. The shared static model needs a locking mechanism to help this cluster coordinate their actions. Relational databases support transactions, which answer this need well.

The central authority will then record the approval or rejection of the request as a successive fact. This fact will find its way back to the client from which the request came. The historical model provides all of the benefits previously mentioned: a complete history of the request, an eventually consistent view of current state, and a mergeable communication mechanism. The one component that is better modeled statically is the one that requires central authority: room availability.

No Real-Time Clock

A time-sensitive request must be fulfilled within a specified period of time. If it is not, the request is invalid. Such requests are common in real-time systems such as factory automation. A request for a door to open or a robotic arm to move must be fulfilled within a narrow span of time. If the message does not arrive in time, then the request must be rejected.

Facts in a historical model, however, are honored no matter what the time frame. The decision is made at the time that the fact is recorded and cannot be rejected thereafter. It may take an indeterminate period of time to transmit the fact. The recipient is simply informed of something that has already happened in history.

While it might be appropriate to model the input or output of a real-time factory automation system historically, the software that runs the factory itself should use a real-time model. These models are specifically designed to provide time-sensitive fail-safe behavior. If a message fails to arrive at the right time, the system defaults to safe operation. And once the message does arrive, late as it is, the system ignores it so as not to cause any damage.

No Uniqueness Constraints

In a historical model, any query for successors of a fact might return multiple results. It is not possible to constrain a query to return only one result. The consequence of this is that a domain that requires at most one result cannot effectively be modeled historically.

For example, a login that requires a unique user name should be supported by a static model. A historical model would be unable to enforce the uniqueness of a user name.

At best, a historical model might include a fact containing only the user name. Because a fact is uniquely identified by its value, there is logically only one fact with this exact user name. However, the fact could contain nothing that could differ from one user to the next. If it contained information in addition to the user name, then two or more facts could again exist with the same name. They would no longer be unique.

To correlate a distinct user name with a user, therefore, would require a successor fact. Identifying the user for a given user name would require a query for the successors of the user name fact. Such a query cannot be guaranteed to return at most one fact. The possibility always exists for it to return more.

To model a system that requires uniqueness constraints, you must use a static model. The model can be consulted to determine if the desired value is already in use. The indexing and transactional features of a relational database once more come into play.

That static model must also be centrally located. A replica of a static model cannot enforce uniqueness. An insertion into one copy would need to block in order to consult the others. Only if that unique value is not reserved in a quorum (usually a simple majority) of replicas can it be accepted. A consensus algorithm such as Paxos can be employed to reach a quorum.

If uniqueness is required, such as registering for a user name, a historical model could be used for registration requests, as well as for acceptance or rejection responses. The requests can be recorded as facts by clients at the edge of the system. These facts will make their way to a central authority that has access to a static model. The static model enforces uniqueness constraints. That central authority will decide whether to approve or reject the request based on the static model and then record that decision as a successor fact.

The response will find its way back to the client from which the request came; only then will the client know whether the requested user name is unique. They will query for successors to the request fact—the acceptance or rejection. Once they have one successor, they will know the answer to the uniqueness question. However, there is no

guarantee in the historical model itself that the request will have no further successors. That assurance comes only from the trust that a central authority is making the decision, with the help of a static model that can enforce uniqueness.

No Aggregation

After a certain amount of activity, a system might be expected to provide an aggregate or summary of that period's activity. For example, a financial ledger could be closed at the end of a day, a month, or a quarter. The system would then produce a summary that records the total of that period's transactions. From that point forward, no additional transactions would be allowed into that period.

A historical model cannot guarantee that all facts within a given period have been seen. The system responsible for generating the aggregate might not have all of the period's records at the required time. If it receives a fact after computing and recording the summary, then it is not permitted—by the rules of historical modeling—to reject it. The decision was made elsewhere, and the fact of that decision was merely shared.

Three strategies exist for dealing with aggregation of historical facts: central ledgers, map-reduce, and blockchains. A central ledger is by far the simplest of the three. A central ledger uses a static model to tally which facts have been included in which period. For example, it determines which financial transactions are part of which date of business or quarterly summary. It makes that decision within the tally as the facts arrive, regardless of when they occurred in history. The tally is a static model. The central authority uses this static model to guarantee that a transaction is not double counted, in other words, included in more than one period.

Map-reduce decentralizes the static model. No longer does a single static model have to contain all of the financial transactions that occur within a date of business. Instead, the transactions are distributed among several static models, called shards. To compute an aggregate for a date or a quarter, a coordinator sends a request to each of these shards. The shards each compute their own aggregate and then share that result with the coordinator. The coordinator combines all of the aggregates into one final answer. This works because no transaction is ever duplicated between shards. If it were, that duplication would lead to over-counting in the final result.

A blockchain is more complex, but avoids the need for a central authority. At many points within the system, individual facts are gathered into candidate blocks. The hash of each block is computed and tested for some arbitrary condition (e.g., a certain number

of leading zeros). This arbitrary condition is a proof of work that ensures that satisfactory blocks are found at a desired frequency. Each candidate block contains the hash of its predecessor, and no fact may appear in more than one block in a chain. Nodes within the system will honor the longest chain of satisfactory blocks.

When designing a system that requires aggregates over history, add a static model—whether singular or sharded—to the historical one. Model individual transactions historically. At a central authority, collect a list of ongoing historical facts into the static model. At regular intervals, close the tally of facts and compute a summary—either as an aggregate function or via map-reduce. This preserves the logical and technical benefits of historical modeling, while also allowing for aggregation.

How to Read a Historical Model

Throughout the rest of this book, we will be exploring many examples of historical models. To do so, we will need a language for describing them. This language will be part visual and part textual. The visual aspect of this language will aide in overall understanding, while the textual part will provide specificity.

The goal of the modeling languages, whether visual or textual, is to achieve a shared understanding about the decisions we are collectively making and the consequences of those decisions. Business analysts, product owners, and information architects will uncover the language and rules of the domain. Developers and system architects will describe the consequences of various decisions. And user experience designers will map the model to task-driven interfaces.

Throughout the process, the team is communicating using a common language. That language needs to be as free as it can be from jargon and implementation details. It should not talk about databases, APIs, services, or repositories. Instead it should focus on the entities and actions of the problem domain.

And yet, the language should also be precise. There should be no room for ambiguity in the system's requirements. When the developers implement a solution according to the specification, there should be no confusion as to how a correct program will behave. Indeed, the specification should even be sufficient to generate some of the code.

Starting from the baseline of immutable architecture, we can define just such a specification language. It carries with it enough constraints to ensure that a correct implementation will have desirable characteristics, like performance, scalability, security, and autonomy. And since one of those constraints is immutability, reasoning about the model will be possible.

The modeling language that we will describe—Historical Modeling—has two graphical components and one textual component. We will begin with the first of the graphical components: the *fact type graph*.

Fact Type Graphs
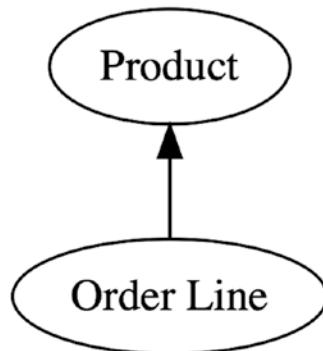
Within the visual language, we will create two kinds of graphs. One will represent the types of facts, and the other instances. Fact *type* graphs will be the more common of the two, so let's describe them first.

In a fact type graph, the type of a fact is represented as a labeled ellipse, as in Figure 3-1.

Figure 3-1. *A single fact type*
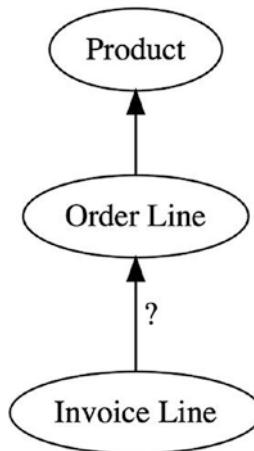
An arrow between two fact types indicates a predecessor/successor relationship. The type at the head of the arrow is a predecessor of the one at the tail. Figure 3-2 shows an example.

Figure 3-2. *Arrows point up toward predecessors*

In any predecessor/successor relationship, a predecessor can have zero or more successors. There is no way in a historical model to limit the number of successors to any given fact. Even though the graph shows only one order line, it implies that a product can have many order lines.
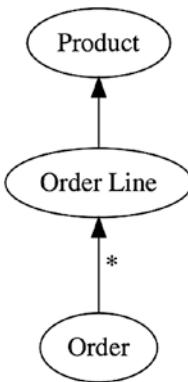
The predecessor side of the relationship, however, can be constrained. In the preceding graph, an order line is associated to exactly one product. With additional notation, we can indicate other cardinalities. In Figure 3-3, we indicate zero or one predecessors with a question mark.

Figure 3-3. *A question mark indicates an optional predecessor*

In this graph, an invoice line may or may not refer to an order line. The reference is optional. Some invoice lines are charges for specific products ordered and would therefore have an order line. Others are fees or discounts unrelated to a specific order line. These would have no predecessor.
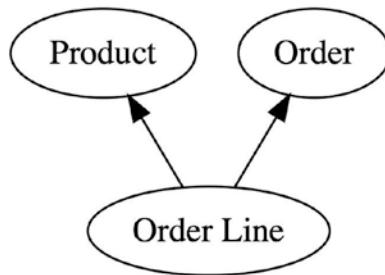
The third cardinality that can be depicted on a fact type graph is zero or more (a.k.a. many) predecessors. To indicate this relationship, we use an asterisk as in Figure 3-4.

Figure 3-4. *An asterisk indicates multiple predecessors*

This graph says that an order refers to zero or more order lines. Because the tail of the arrow always allows for multiples, this is a many-to-many relationship.

There is more than cardinality at play in this graph. It also represents the order in which facts can be created and the degree to which changes can be applied. The arrow in Figure 3-4 allows multiple order lines as predecessors. The arrow in Figure 3-5 is reversed and has no asterisk. Yet it still allows multiple order lines, only now as successors.
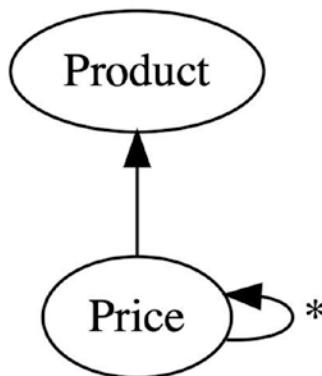
Figure 3-5. *Successor relationships always allow multiples, so this model still allows multiple order lines*

In Figure 3-5, the direction of the predecessor/successor relationship between order and order line is reversed. The successor end of an arrow always allows zero or more facts, and so both graphs indicate many order lines to an order. The distinction, however, is related to immutability. In Figure 3-4, the many order lines are predecessors of the order. They were known at the creation of the order. New order lines cannot be added to the order after the fact. Existing order lines cannot be removed after the order is created.

The collection of order lines is immutable. When we wish to call out this immutability, we will often say that the successor *captures* its predecessors, as in "An order captures many order lines."

In this domain, we want to lock down the order lines in an order, and so we will choose to make order line the predecessor (as in Figure 3-4). However, we will introduce the concept of a cart, to which a customer may indeed add order lines. It is important to choose the model based on the direction of the predecessor/successor relationships, not just based on cardinality.

For some patterns that we will study shortly, it makes sense for a fact to refer to other facts of its own type. We indicate this as a loop, as in Figure 3-6.
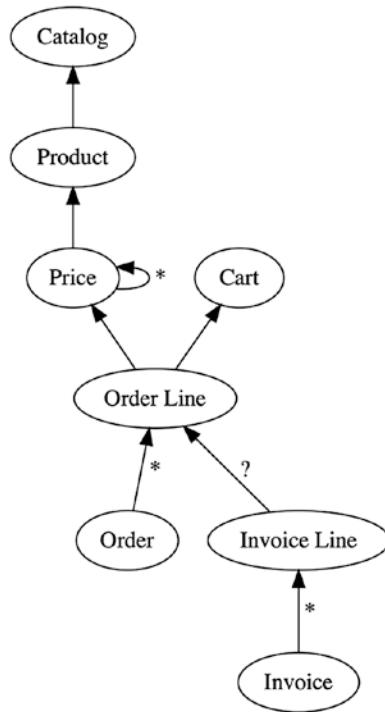
Figure 3-6. *A loop indicates that the previous prices are predecessors*

A price refers to not only a product as a predecessor, but also zero or more prices that came before. This indicates that a price replaces previous values. Even though this relationship introduces a cycle into the fact type graph, it does not allow cycles of fact instances. A given instance of a price cannot refer to itself as a predecessor. It can only refer to a previous price, the one that it is replacing.

There must have been a first price for the product. That fact instance would have no price predecessors. That is why we cannot represent this as a one-and-only-one relationship. Loops within a fact type graph will necessarily include an arrow that is either optional (zero or one) or—more commonly—many (zero or more). Because these references are the only way to model properties that change over time, we will often refer to them as "mutable," as in "A product has a mutable price."

Putting it all together, a complete model for an ordering domain appears in Figure 3-7.
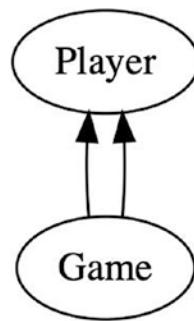
Figure 3-7. *A fact type graph showing various decisions for cardinality*

A product in a catalog has a mutable price. An order line in a cart captures a specific price for a product. An order captures a set of order lines. Order lines can be added to a cart, but the order is closed. An invoice line may or may not refer to an order line, and an invoice captures a set of invoice lines.

The fact type diagram expresses both the cardinality and the causality of the domain. Reading through it reveals a narrative of how a system came to be in a particular state and exposes the constraints on how that system can and cannot change.
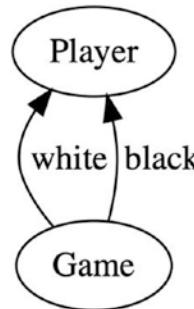
A Chess Game

When I first started thinking about Historical Modeling, I drew a few models to convince myself that it would be a useful way to analyze software. The first model that I drew was of a single chess game. Chess is a game between two players. The players are chosen before the game begins. Players are therefore predecessors of the game, as shown in Figure 3-8.

Figure 3-8. *A chess game has two players*

This immediately revealed that I needed a way to talk about different predecessors of the same type. At first, I thought about perhaps allowing an array of players and using the index to indicate the order of play. As I got into further models, I realized that predecessors had to be *sets*, not *arrays*, which invalidated that idea. Furthermore, the game supports precisely two players. An array would allow invalid games of 0, 1, or 3+ players to be represented. The natural conclusion was that predecessor relationships should support labels, as in Figure 3-9.

Figure 3-9. *The players in a chess game have labels*
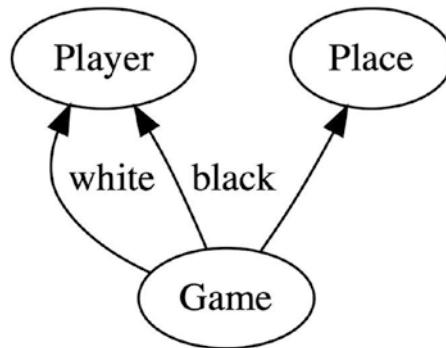
When no disambiguation is required, the labels can be omitted. The label always describes the predecessor, not the successor.

Important Attributes

In chess literature, the place at which a game is played is often used in the title. It represents an important attribute of the model. My first instinct was to record this attribute in the game fact. It would then be possible to display that information as part of the title of the game.

65

Upon further reflection, however, it becomes clear that that would not be quite right. Several games will be played at the same place. If the place is simply a field of the game, then the model does not faithfully represent that truth. The name of a place is not the place itself. And there is no way of finding all games played in a particular location.

The solution, as shown in Figure 3-10, is to represent place as a fact. The place is a predecessor of the game; the place existed before the game was played, and the game knows about that location. A game cannot be moved to a different place. The place is immutable and part of the game's identity.
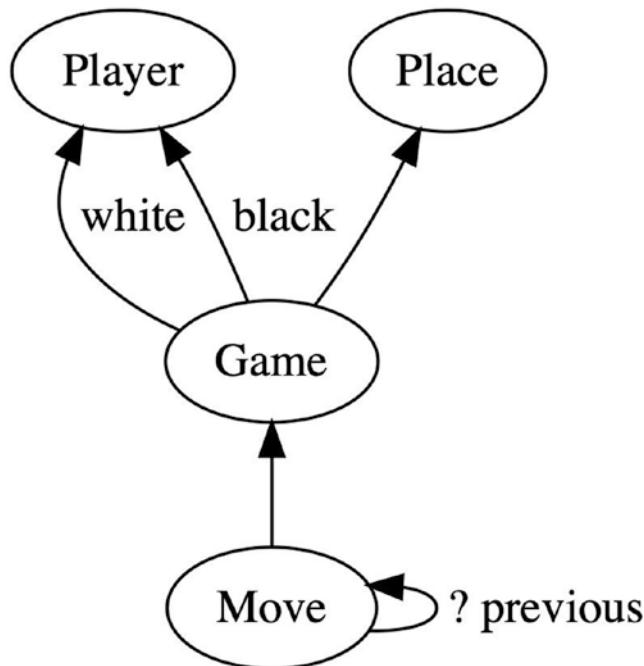
Figure 3-10. *Place is important enough to be extracted from game and become its own fact*

With this change, the model accurately represents the importance of place. It is a fact in its own right. It stands apart from any game, but it also provides a way to find a set of games. If an important attribute is hidden away as a field in another fact, then we have no way of talking about such a query. But if the important attribute is represented as a separate fact, then we can reason about it appropriately.

A Chain of Facts

The next part of the game that I needed to model was the moves. Considering that a predecessor is a thing that happened before, I decided to represent move order using predecessor relationships. The result was the diagram in Figure 3-11.
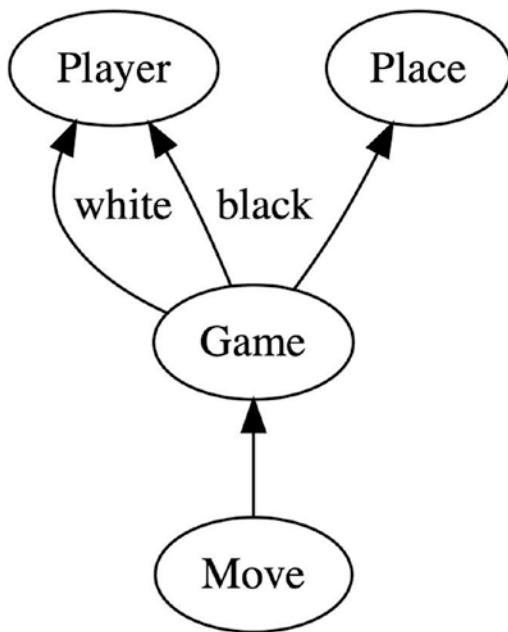
Figure 3-11. *A move refers to the previous move as its predecessor*

The question mark indicates that `previous` move is optional; the first move would have none. The predecessor captures only the immediately previous move, as `previous` allows *optional* predecessors, not *many* predecessors. The entire history of the game can be found in the chain of predecessors.

While the model requires that only one move *precede* another one, it does not require that only one move *follow* another. The structure of the model does not prevent a player from cheating by playing two possible moves.

The structure also does not prevent a move from following a move from a different game. There is no constraint that the prior move belongs to the same game. This will have to be expressed in a validation rule.

After analyzing this model, I was not particularly happy with the `prior` relationship. I realized that later moves in the game would be identified by a long list of previous moves. It seemed wasteful. I couldn't quite quantify the issue, because I hadn't yet determined how to implement this model. But I decided that a move could simply contain a move number as well as the squares affected. Starting at zero, white would play the even moves and black the odd moves. This flattened the model into the one in Figure 3-12.
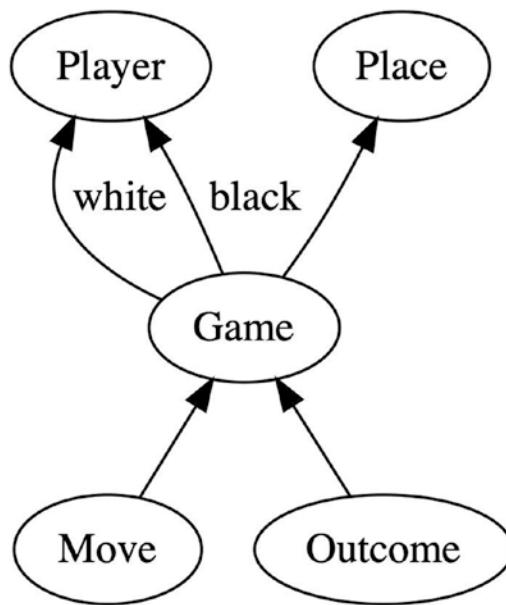
Figure 3-12. *A move refers to the previous move as its predecessor*

The model no longer represents the structure of the game. It does not record the idea of one move following another. But then again, given the inability to disallow forks, the structure wasn't providing much value. Furthermore, removing the `prior` reference eliminated the possibility of moves crossing games. Modeling is a trade-off between the rigor of the type system and the brute force of validation. Validation of sequential move order makes for a simpler model.

Endgame
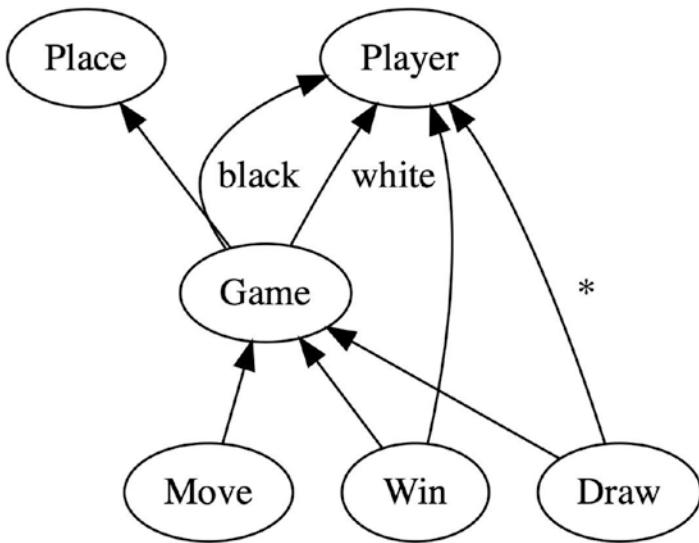
The last part of the chess game to model was the outcome. Was it a win for white (1-0), a win for black (0-1), or a draw (½-½)? The simplest solution is to create an outcome fact and record that value as a field, as shown in Figure 3-13.

Figure 3-13. *A new fact records the outcome of the game*

I found two problems with this simple solution. First, it offered no help when searching for games that a certain player has won. This might be an important query, so the model should support it. And second, it allowed for moves to continue past the outcome. The outcome should lock the moves into place.
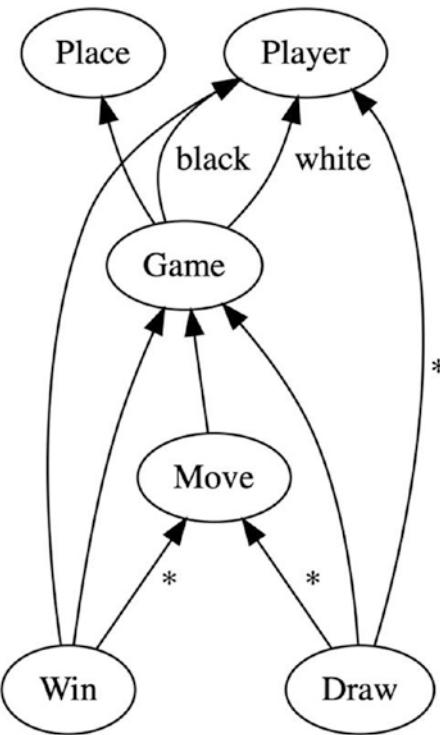
To solve the first problem, I decided to separate the Outcome fact into two different types: Win and Draw. A Win refers to one of the two players. The Draw—to help with the query—refers to both. This changes the model to the one in Figure 3-14.

Figure 3-14. *A game concludes with a win for one player, or a draw for both*

The model again requires validation to enforce that the winner must be one of the players from the game. But on the plus side, it is much easier to query for all wins and draws by a player. This could help with things like computing score.

To solve the second problem, I took advantage of the fact that predecessor relationships were immutable. If an outcome records all moves of the game as predecessors, then those moves are locked down when the game concludes, as Figure 3-15 illustrates. Sure, future Move facts could be recorded, but they would not have contributed to the outcome.
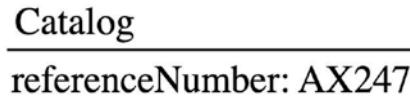
Figure 3-15. *A win or a draw locks in a set of moves*

As this example shows, the act of modeling a problem historically exposes several decisions that might otherwise be deferred as implementation details. The correct option is not always clear. But the choice has important downstream effects. One path might lead to a simpler model, but a difficult query. Another might constrain the system to render invalid inputs unrepresentable, only to put too much of the burden on the type system. Take the time to draw out several alternatives and analyze each on its own merits.

Fact Instance Graphs
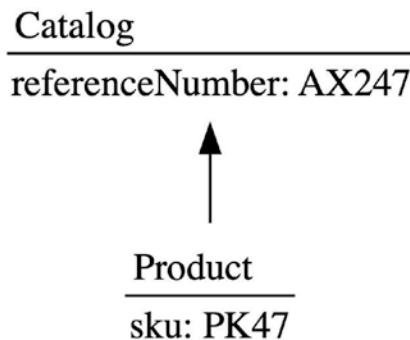
So far, the graphs we've drawn refer to fact *types*. The visual language of Historical Modeling also includes a kind of graph that represents fact *instances*. We will use this kind of graph less frequently. The goal of a fact instance graph is to illustrate a specific example of the state of a system at a certain point in time and at a certain node. It includes more details about the fact instances observed, but typically contains only a small number of facts.

Fact instances are distinguished from fact types by drawing them without a border. Instead, the body of the fact forms a rectangle of text, with a separator between the type and the contents, as in Figure 3-16.

Figure 3-16. *A catalog fact instance is shown with all of its values*
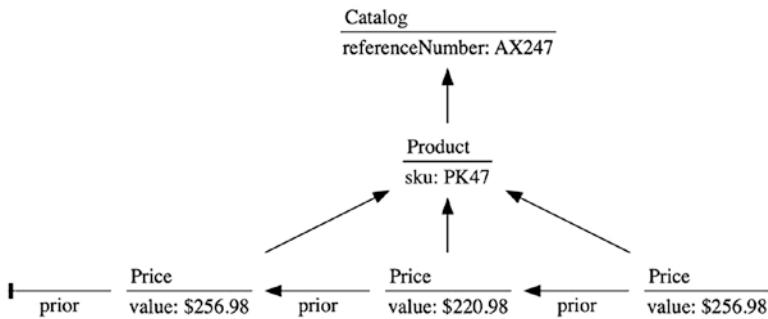
Predecessor/successor relationships are drawn with arrows as in the type diagram, but an instance diagram does not indicate cardinality. No relationships include the asterisk or question mark indicators. Figure 3-17 illustrates a relationship between a catalog and one of its products.

Figure 3-17. *A product points to its predecessor catalog*

When a predecessor reference is "optional" or "many," it allows zero predecessors. An instance with no predecessors in this role is drawn in an instance diagram as a terminating line. If the role is optional (?), then the terminator represents a null reference. If the role is many (*) as in this example, then the terminator represents an empty set.

Terminators are helpful to illustrate the changes in a mutable property. The root instance, or initial value, of a mutable property will have no predecessor. Subsequent instances will form a chain from the root, as in Figure 3-18.
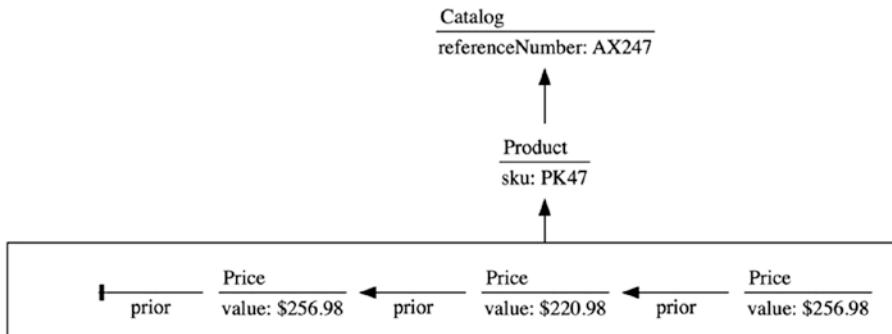
Figure 3-18. *The first price has no prior price*

Recall that the fact *type* diagram depicted only one price node. A small loop indicated that a price had zero or more prior prices. Now in the fact *instance* diagram, we see what that loop really means. It doesn't indicate that the model allows cycles. It means only that instances of a given type can refer to other instances of the same type. The loop has turned into a chain.

A fact instance diagram does not admit cycles. A fact never refers to itself as a predecessor. Nor can facts refer to predecessors that in turn refer to the original, directly or indirectly.

Since all instances of the price refer to a common predecessor Product, a shorthand can be used to group the common relationships. A box is drawn around all members of the group, as in Figure 3-19. The predecessor reference common to all members of the group is shown as a line from this box. If a successor referred to all members of the group, it would be represented as an inward arrow.
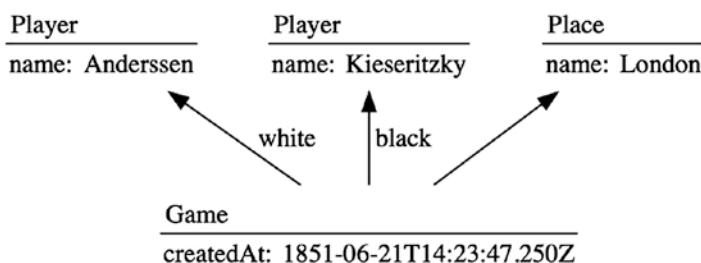
Figure 3-19. *A box indicates that all instances share a common predecessor*

Whereas fact *type* diagrams are general descriptions of a model, fact *instance* diagrams show specific examples. Instance diagrams help to illustrate the consequences of different modeling decisions. They offer a form of debugging prior to a model having been implemented. Let's walk through a specific example and show how it helps us reason through design decisions.

The Immortal Game

Previously, we imagined what any chess game might look like. Now, we will draw a specific instance of a chess game using that model. Before the game begins, we have two players: Anderssen and Kieseritzky. They meet in London. As Figure 3-20 shows, the game is a fact that joins those two players at that place at a certain point in time.

Figure 3-20. *A game between Anderssen and Kieseritzky in London in 1851*

It's important that we included the `createdAt` field within the `Game`. This represents the moment, from the perspective of the client, that the fact was created. In this example, we are using the exact moment that the game began in London. (The time of day in this example is fictional, as records to this level of accuracy do not exist.)
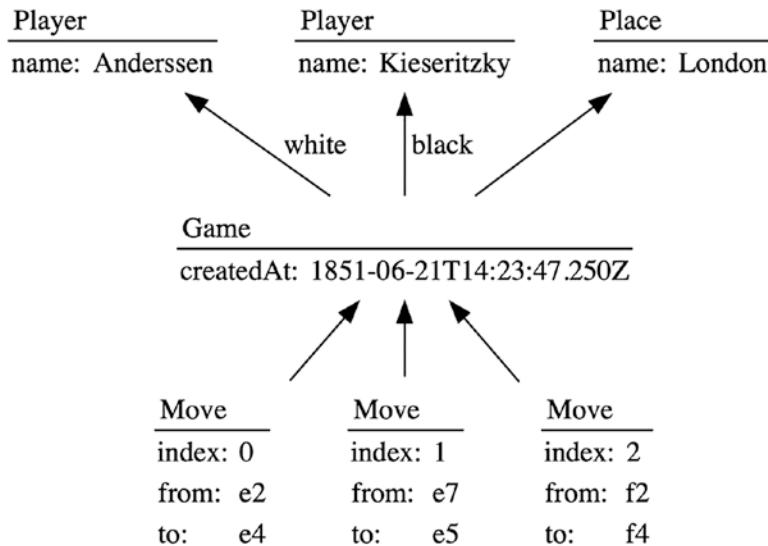
If we had modeled a `Game` without a `createdAt` field, then any game between Anderssen as white and Kieseritzky as black played in London would be the same game. A fact is uniquely identified by its type, the values of its fields, and the set of its predecessors. This model makes the reasonable assumption that two players will not simultaneously start two different games.

It is also important that `createdAt` represents the moment at which the fact became a fact, not the moment at which it became known to any particular computer. Of course, at the time there were no computers in London to capture the fact. Nevertheless, it existed. In modern systems, this creation time is often captured as a timestamp on the

client machine. After that, however, other machines that learn of this fact must honor that timestamp, no matter how much time has passed.
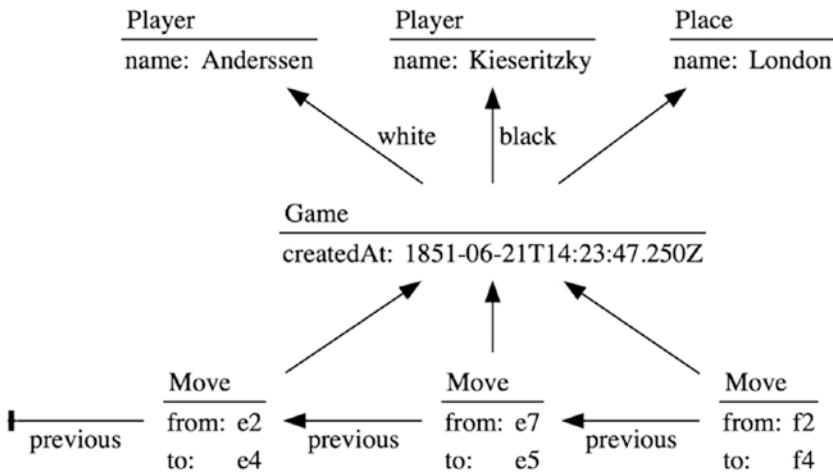
Collecting Moves

Now that the fact of the game is recorded, the players begin making moves. We can capture these moves as facts referring to the Game. Figure 3-21 shows the game after the first three moves.

Figure 3-21. *Moves are captured as successors of the game fact*

In a previous iteration of the model, we considered representing the relationship between a move and its previous move as a predecessor. This had the advantage of being accurate: a move *is* the predecessor of the one that follows. But it had the more significant disadvantage of creating long chains. Had we chosen this model, the game after three moves would look like Figure 3-22.
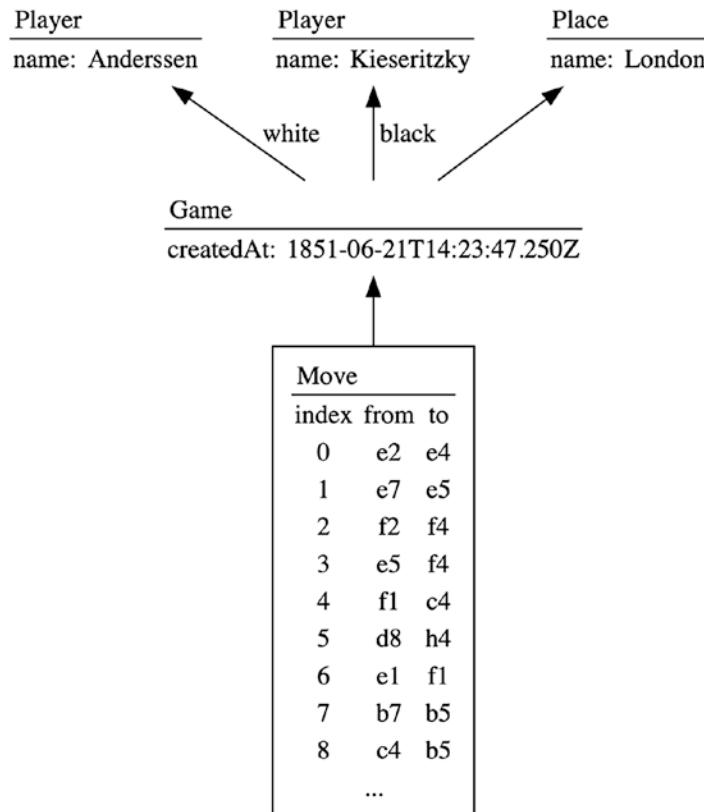
Figure 3-22. *In an alternate model, moves are represented as predecessors of one another*

As the game progresses, the chain would get longer. Understanding that the 20th move *is* the 20th move requires traversing this entire chain. Even *identifying* the 20th move means talking about the entire history, as predecessors are *part* of identity.

Because of the practical drawbacks of recording moves as a chain, we decided to model the system as having individual moves, each with an index. And so, we abandon Figure 3-22 and go back to Figure 3-21. That leaves us with the chore of validating those indexes to ensure that we have no gaps and no duplicates. We can simply add this to the validation that would already have to occur to defend against illegal moves such as moving into check. The model can only go so far to make invalid state unrepresentable.

As we collect more and more moves within the game, representing them as individual facts on the diagram becomes tedious. So instead, we group them together. The group has a common predecessor: the Game. Within the box, we can simply draw the set of moves as a table, as shown in Figure 3-23.
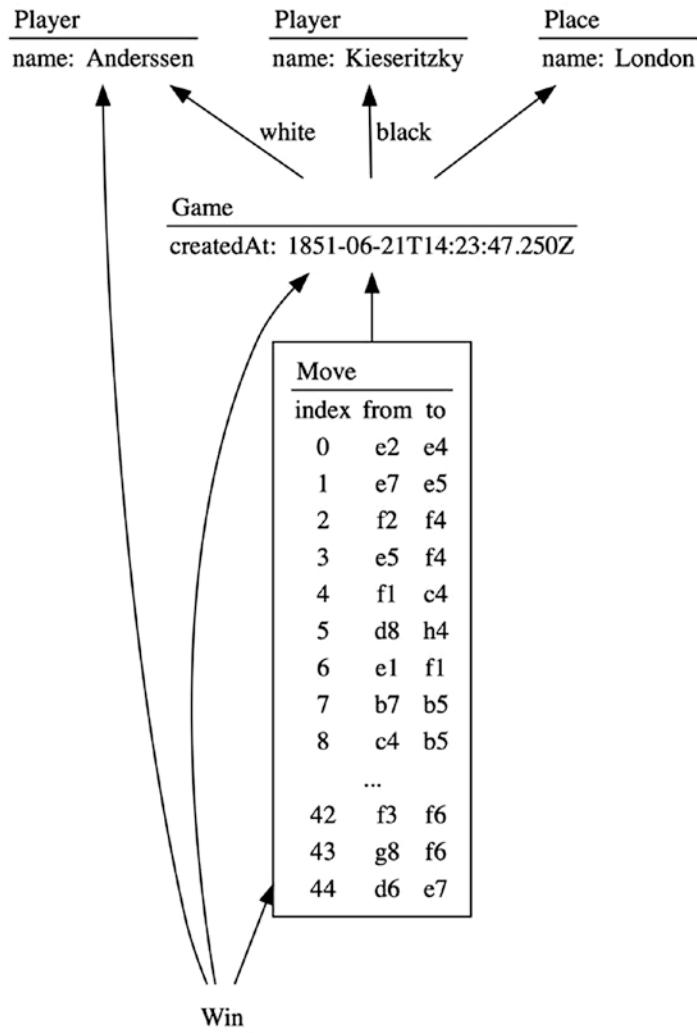
Figure 3-23. *Moves are grouped together under their common predecessor*

The grouping does not change the fact that each move is a separate record. The order of moves within the grouping does not imply any relationship between the facts. This is simply a convenience to constrain the size of the illustration.

A Brilliant Win

In the game that we are modeling, Anderssen sacrificed material brilliantly to secure a win for white. We will represent that win in Figure 3-24 with a fact that captures the game, the winner, and the set of moves as predecessors.

Figure 3-24. *A win captures several predecessors, including all moves of the game*

The Win fact has no fields. It needs no additional information. It says all that it needs to say with the predecessors it gathers together. We draw the predecessor set of moves as an arrow pointing toward the entire group. Every Move in the group is a predecessor of the Win.

As this example shows, fact instance graphs are quite different from fact type graphs. They represent individual instances of facts, complete with their type and fields. They show predecessor/successor relationships between facts rather than the roles between types. Whereas a fact type graph allows cycles between types, an instance graph does not permit cycles among instances. A cycle in the fact type graph unrolls into a chain in the fact instance graph.

Fact type graphs and fact instance graphs are used for different purposes. A fact type graph makes general statements about all possible models of given fact types. We use them to reason through the rules of a domain. A fact instance graph, on the other hand, illustrates a specific example data set. We use them to try various scenarios and debug a model before implementation.

Because fact type graphs better describe the general rules of a domain, we will rely more heavily upon them than on fact instance graphs. Fact instance graphs are often too specific to make general statements. In the remainder of this book, they will be used sparingly and only to illustrate specific patterns.

The Factual Modeling Language

While the visual language of Historical Modeling is useful for understanding the relationships within a model, it is not sufficient for rigorous reasoning. It cannot specify a model to such a degree that assertions can be proved, or code generated. To satisfy this need, we use the Factual Modeling Language, or Factual.

Using the Factual Modeling Language, we can write precise specifications. It describes all the data that is part of the model, the ways in which those data elements relate, and the rules by which we can query it. It even describes the rules by which data is secured and validated. This precise specification language allows us to reason about the requirements of a system and determine far in advance what we will be able to implement.

Declaring Fact Types

Types of facts are declared in Factual using the `type` keyword. The body of the type, enclosed in brackets, includes a list of fields and predecessors. Fields are declared in a style reminiscent of Pascal and related programming languages: a field name is followed by a colon and a type. Fields have native data types, such as `string`, `int`, and `bool`.

```
fact Catalog {
  referenceNumber: string
}
```

Predecessors are similarly declared. The primary difference is that a predecessor refers to another fact type. Predecessors can appear before, after, or interspersed with fields.

```
fact Product {
  catalog: Catalog
  sku: string
}
```

Cardinality indicators modify predecessor declarations. Singular predecessors have no modifier (as shown earlier). Optional predecessors are declared with the question mark modifier, and multiple predecessors are declared with an asterisk.

```
fact InvoiceLine {
  orderLine: OrderLine?
  total: decimal
  description: string
}

fact Invoice {
  lines: InvoiceLine*
  subtotal: decimal
  tax: decimal
}
```

A predecessor can refer to a type that has not yet been declared. It can even refer to the type in which it is declared. Such self-referential predecessors, as we discussed previously, are frequently used to refer to previous versions of mutable properties. Remember, while this introduces a cycle in the type graph, it does not permit cycles within the instance graph:

```
fact Price {
  product: Product
  value: decimal
  prior: Price*
}
```

The fact declaration syntax is designed to be familiar to developers. It is also simple enough to be a communication mechanism between developers and non-developers. It contains no behavior, no access modifiers, nothing that could be confused for code. And yet, it is precise and expressive enough to describe the fundamentals of a model.

Consider each fact as a decision that a person or another actor is making. The fact captures the details of that decision. It also shows which decisions came before in the form of predecessors. By stepping through the creation of facts, analysts tell the story of how a system evolves to solve business problems.
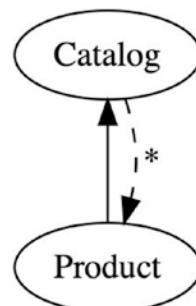
Querying the Model

As powerful as it is to declare the types of facts in a system, it is also useful to answer questions based on those facts. The Factual Modeling Language includes syntax for querying a model to find its current state. Queries are declared using the query keyword and given a descriptive name. All queries begin from the perspective of a given fact, which is given as the parameter to the query.

```
query productsInCatalog(c: Catalog) {
  match p: Product where p.catalog = c
}
```

The body of the query includes the match keyword. This introduces a clause that matches other facts related to the starting point. Give the alias and type of the fact to match and then a where clause. Following where, equate predecessors of each side.

The preceding query finds all products within a given catalog. When overlaid on the fact type diagram, the query can be thought of as traversing the predecessor relationship in the opposite direction. Figure 3-25 shows this as a dotted line.

Figure 3-25. *A query finds all successors of the predecessor*
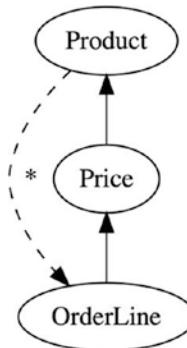
Since it follows the successor relationship, the aforementioned query can return multiple results. I've labeled the dotted line with an asterisk to illustrate this point. Most queries will include successor paths and therefore return multiple results.

Jumping Levels

The where clause is not limited to following a single predecessor reference. By chaining additional predecessor references, a query can reach further down the successor graph. For example, a line on an order captures a specific price of a product. In the model that we defined earlier, the product is a predecessor of the price. The order line therefore can only reach the product indirectly through the price fact. To find all order lines for a particular product, we traverse those two relationships in the opposite direction.

```
query orderLinesForProduct(p: Product) {
  match ol: OrderLine where ol.price.product = p
}
```

Since the relationship between OrderLine and Product is indirect, the query follows the intermediate predecessor. Starting at a Product, we match all OrderLines where the price's product is the starting point. Let's overlay the query onto the fact type diagram, as shown in Figure 3-26, to see how this jumps levels.
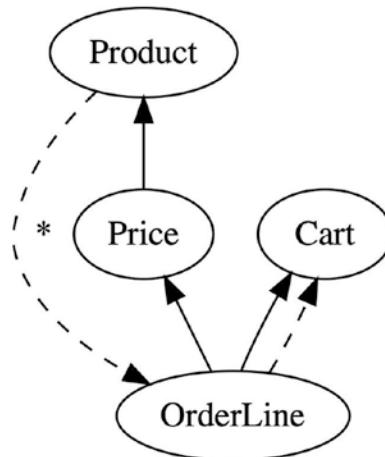
Figure 3-26. *A query matches successors two levels down*

Joining Matches

In addition to going deeper into the graph, we can also bounce back out. By appending additional matches, the query can turn and traverse up the graph along a different branch. Use the then keyword in place of match to indicate the join. We can use this, for example, to find all of the carts containing a given product.

```
query cartsContainingProduct(p: Product) {
  match ol: OrderLine where ol.price.product = p
  then c: Cart where c = ol.cart
}
```

The first match finds all of the order lines that indirectly reference the given product. The second match finds the cart containing that order line. It does so by simply following the predecessor reference. The full query, jumping down to order lines and then back up to carts, appears in Figure 3-27.

Figure 3-27. *A query matches predecessors of intermediate successors*

The match going down to successors can return many results. I've illustrated it with a cardinality indicator (*). But the match going up to its predecessor will return only one cart for an order line. That arrow has no cardinality indicator. Overall, the query will return multiple results, since it includes a successor match.

The query syntax we have just described is sufficient for traversing down any number of successors, up any number of predecessors, and bouncing off of facts to explore different directions. This covers the entire graph of connected facts. If one fact is in some

83

way related to another, then a query could be written to find a set that includes one fact, starting from the other.
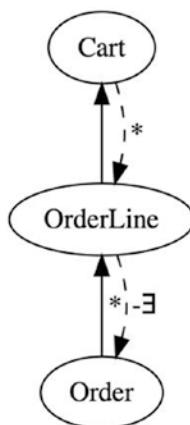
Existential Quantifiers

Often, we find it necessary to limit the number of facts matching a query. Rather than listing all related facts, we want to instead focus on a subset that is in a certain state. The state of a fact is not intrinsic to the thing itself; facts are immutable after all. Rather, the state of a fact is determined by the presence or absence of successors. We therefore constrain queries using existential quantifiers.

An existential quantifier takes the form of a such that not exists clause. The clause includes an alias, type, and where clause just like a match. However, rather than *returning* all matching facts, this clause *disqualifies* a match if any exist.

```
query linesRemainingInCart(c: Cart) {
  match ol: OrderLine where ol.cart = c
    such that not exists o: Order where o.orderLines = ol
}
```

The preceding query finds all of the lines in a cart that are not yet part of an order. It shows only the unordered lines, which might be useful for updating the user interface or generating a new order. When an Order is created referring to the order line, it is removed from the query results. The query, complete with the *not exists* clause (-∃), appears in Figure 3-28.
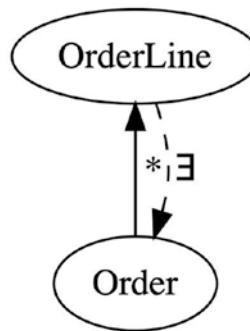
Figure 3-28. *A query matches successors for which a second-level successor does not exist*

An existential quantifier might be useful in other queries. If so, you can declare them independently using the predicate keyword. For example, it would be equivalent to name the preceding predicate as orderLineIsOrdered.

```
predicate orderLineIsOrdered(ol: OrderLine) {
  exists o: Order where o.orderLines = ol
}
```

The predicate refers only to the existential quantifier overlaid on the diagram in Figure 3-29.

Figure 3-29. *A predicate that tests for the presence of a successor*

The predicate is false for an order line so long as no successor order has been created. Once the order is created, the predicate becomes true. We can now refer to that predicate by name in a query.

```
query linesRemainingInCart(c: Cart) {
  match ol: OrderLine where ol.cart = c
    such that not orderLineIsOrdered(ol)
}
```
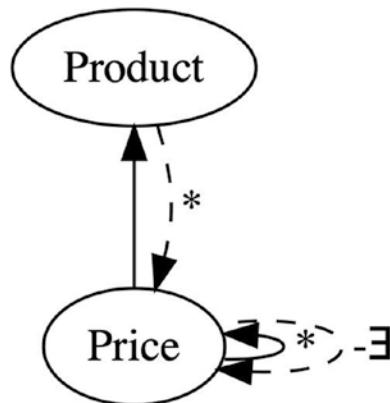
The query has the same effect as the original. The predicate has simply been extracted and named so that it can be reused.

Current Value

When used in this way, existential quantifiers help us determine the state of a fact within a workflow. Used slightly differently, they also help us determine the current value of a mutable property. A current `Price` of a `Product` is one that hasn't been superseded by a next `Price`.

```
query currentPriceOfProduct(pr: Product) {
  match p: Price where p.product = pr
    such that not exists next: Price where next.prior = p
}
```

I do not say *the* current price of a product. Indeed, I cannot guarantee that this query will return only one `Price`. While I would like to believe that a product has only one current price, the model does not allow me to constrain it to be singular. I must write this query as a multitude of current prices. This is generally true of all successor queries; the cardinality of results is always zero or more. As Figure 3-30 shows, the query includes a successor join (indicated with an asterisk).

Figure 3-30. *A current price of a product is one for which no successor exists*

A mutable property has zero or more current values. When only one value is returned, then there have been no concurrent edits. But when more than one value is returned, they are the candidate values. We will examine mutable properties in greater detail.

Authorization Rules

An important aspect of the model is who is authorized to perform which actions. Since a fact maps to a decision that a person has made, facts are the perfect proxy for the actions that a user can perform. We therefore model authorization rules as queries starting from the authorized fact.

Suppose, for example, that a user created a cart. We could capture the user's identity as a prerequisite.

```
fact Cart {
  createdBy: User
  createdAt: timestamp
}
```

Adding an order line to a cart can then be expressed as an authorization rule. Only the creator of the cart is allowed to add the order line. This is expressed using the `authorize` keyword.

```
authorize ol: OrderLine {
  match u: User where ol.cart.createdBy = u
}
```

The body of the authorization rule is exactly like the body of a query that returns a collection of users. If the user who initiated the action is in the query results, then that user is authorized to perform the action. We will discuss authorization in more detail in Chapter 7.

We will be using both the visual and the textual language of Historical Modeling throughout the remainder of this book. The visual language will be the primary tool for understanding a model. We will lean most heavily upon the type graph. The textual language, on the other hand, will be employed to specify a model to a useful degree of detail. This form will help us to reason through and prove assertions about a system's behavior.

As you apply Historical Modeling in your work, you will find yourself starting with the visual language. It is quite useful for initially planning a model and demonstrating the relationships among facts. However, you will quickly move to the textual language for implementation. The tools that you will use convert the textual language into both code and graphs. The code will form the basis of your system implementation, and the graphs

will document the model visually. The Factual Modeling Language will be your tool for building working models and sharing them with others.

A Chess Application

Now that we've analyzed the chess game model, and constructed an example game, let's see what it would look like as a Factual specification. The goal of this specification is to provide sufficient detail for writing application requirements and generating an implementation. We can begin by writing out the fact types that we've already defined visually.

```
fact Player {
  name: string
}

fact Place {
  name: string
}

fact Game {
  white: Player
  black: Player
  place: Place
  createdAt: timestamp
}

fact Move {
  game: Game
  index: int
  from: int
  to: int
}

fact Win {
  game: Game
  player: Player
  moves: Move*
}
```

```
fact Draw {
  game: Game
  players: Player*
  moves: Move*
}
```

Developers among you can probably already see how to generate data types, APIs, and database tables. Hold on; we will get to that. The most important thing, however, is that business analysts among you can also understand this model. All of the words, except the primitive types and the keyword `fact`, are in the language of the domain model. You might be left with a question like "How is a square represented as an integer?", but other implementation-specific questions are not raised by this domain-focused model.

Use Cases

Given just the descriptions of the facts, the team can already begin discussing use cases. What are the actions that a user of the system can take? Each fact is a use case.

The `Player` fact, for example, is generated when a new player registers in the system. The only field in `Player` is `name`, which means that a player is uniquely identified by name alone. It also means that a player cannot change their name. If these consequences are not acceptable, now is the time to change the model.

The `Game` fact tells of another use case: starting a game. In order to start a game, we need two players in a place. This sparks conversations about whether one of the players starts the game, or perhaps an organizer of the tournament at a particular place. What are the rules for determining who plays white? All of these decisions have to be made prior to recording the `Game` fact.

The model also reveals that we need a use case describing how a player makes a `Move`. It should include validation rules, like the white player makes the even indexed moves and the black player makes the odd indexed moves. Other validation rules cover what to do with duplicate indexes or gaps. Perhaps we also need to add some authorization rules, such as a move can only be made by a player of the game.

```
authorize m: Move {
  m.game.white
}
```

```
authorize m: Move {
  m.game.black
}
```

Walking through these use cases with the model in hand, we can evaluate the consequences of our choices. When necessary, we can make changes to the model to give us the desired capabilities. All of this can be done before we implement the first user story. We will discuss this in more detail in Chapter 5.

User Interface

Once we are satisfied that we have captured the right facts, we can begin mapping them to the user interface. This is where the Factual Modeling Language pairs nicely with information architecture, site maps, layouts, and wireframes. Factual expresses two dimensions of that design: what actions the user can take and what the user can see.

Actions

First, consider the actions that the user can take in terms of historical facts. You could determine that you need to provide create, update, and delete actions and therefore design a CRUD model. This is supported by the structural patterns that we will discuss in Chapter 8. More likely, however, you will find that you need a task-driven user interface. The user gathers together just enough information to perform a task. This task is then captured as a fact.

In the chess application, we may decide that the Place fact is best supported by a CRUD interface. A tournament administrator needs the ability to create, update, and delete places. To support this kind of interface, we will need to expand on the model. The name field, for example, would need to become a mutable property. And a new fact needs to be introduced to represent place deletion.

Other parts of the user interface will need to be a bit more task driven. Starting a Game would be an entire task involving both players and the Place. Making a Move is the primary task of the gameplay interface. With each action that a user can perform, record the fact that will be created in response.

Views

The second dimension of the user interface design that can be expressed in Factual is what the user can see. Think about a user asking questions of the model and how the user interface will answer those questions. Then, tie areas of the wireframes to queries in the Factual Modeling Language.

Consider what the player sees when they first log into the system. At that point, we have a Player fact. We can display a list of games that are currently in progress.

```
query gamesInProgress(p: Player) {
  match g: Game where g.white = p or g.black = p
    such that not exists w: Win where w.game = g
    and not exists d: Draw where d.game = g
}
```

In the corner, we can display the player's score. A win is worth one point, and a draw is worth half a point. This is computed from the following queries:

```
query wins(p: Player) {
  match w: Win where w.player = p
}

query draws(p: Player) {
  match d: Draw where d.player = p
}
```

Indicate on the wireframe that the score to display is 1*wins(p).count + 0.5*draws(p).count. This provides enough precision to completely specify the desired behavior. It also implies that when a Win or a Draw is recorded, this score needs to be updated.

The Factual Modeling Language provides enough detail to express these requirements. By reasoning from this language, business analysts, developers, and designers alike can determine the consequences of their decisions. When they change a decision to get better results, they can see how that change will affect other parts of the system.

Each of the forms of expression described here has a place. We usually start at a whiteboard making rapid changes to a model using fact type graphs. This captures the important objects and user actions and shows how they are related. Then we will

write out some examples using fact instance graphs. This helps validate the model as we explore specific scenarios. Then finally, we take the time to express in the Factual Modeling Language the specific structure, queries, and authorization rules of the system. This ensures that we have completely analyzed and documented and can implement the system precisely.

All of these artifacts support team communication. Whether graphical or textual, the forms in which a model is expressed are designed to be produced and consumed by all members of a team. They are intended to focus exclusively on the problem domain, not on any particular implementation details. And yet, because these artifacts obey the laws of immutable architecture, they will lead to a sound implementation.

Let us now focus on those implementation details. The kinds of systems that we design using immutable architecture will tend to be distributed systems. Let's explore the nature of distributed systems and see why immutable architectures such as what we've described here work so well in that context.

PART II

Application

Location Independence

In the not too distant past, most programs ran on a single computer. After the proliferation of JavaScript in the web browser, apps on mobile phones, and microservices in the cloud, most programs that we write today run across many computers. Whereas distributed systems used to be a specialty, today they are the default. We need to update other defaults to meet that demand.

One of the defaults that we need to update is the assumption that data has a location. Some systems try to treat remote objects as if they are local. DCOM uses object identifiers to make a proxy look like a local instance of a remote object. Remote procedure calls (RPCs) try to hide the reality of network communication behind an interface that looks like a normal function. The problems with these systems have been well covered elsewhere, so I will not rehash them here.

The assumptions of locality that I want to examine are a bit more subtle. Even when we replace RPCs with messages, and object identifiers with URLs, it is easy to assume that data has a location. We make that assumption whenever we identify a "source of truth" or a "system of record." We rely upon location whenever a single node generates unique identifiers. Our default mode of programming what happens *at a machine* leaks into the behaviors that we program into the system as a whole.

So many of the behaviors that we've come to expect from our systems depend upon location. We expect items to be sequentially ordered. We expect the system to reject duplicate names. We expect that when the user updates a property of an object, it will have the value that they just assigned. Indeed, the expectation that properties to even *have* single values is a location-dependent assumption.

A system that depends upon location will misbehave when that location becomes unavailable. If we strive instead for location independence, we will construct systems that are more responsive, resilient, and reliable. They can act autonomously without communicating with remote nodes. They can tolerate network failures without introducing defects. And the decisions that a user makes in isolation will be honored when other nodes and users learn of that decision.

Modeling with Immutability

At its core, the assumption of location is all about mutability. A variable is a place that stores a value. Programs use the variable to address that location and read its value. After the variable is updated, we expect the program to read the new value the next time it looks. Scale this up to the level of the distributed system, and you have location dependence. A system depends upon data being in a location specifically because that data is allowed to change.

If we search instead for a model of computing that is based on immutability, then dependence upon location fades away. If an object cannot change, then every copy of it is just as good. There is no need to know where the object is stored, where it was created, or which subsystem is the source of truth.

Of course, we need to model domains that change over time. So, the concepts of time and change need to be re-examined in the light of immutability.

Synchronization

It's not uncommon to talk about managing data in distributed systems as a synchronization problem. But even this term comes from a place of putting data in a location. Synchronization is the task of changing data in two or more places at more or less the same time. When the data in two locations differs, those locations are out of sync. A location-dependent system will seek to synchronize them.

When data no longer has location, concurrent changes are allowed to happen. A temporary disagreement between two nodes is not a synchronization problem to be solved, but an opportunity for them to converge over time. A location-independent system uses a different definition of time so that it can describe concurrency. It relaxes its assumptions so that changes are no longer linear. This helps the system to ensure that concurrent changes don't cause conflicts.

A location-independent system is not concerned with synchronization, but with causality. It seeks to understand which events caused which other events. Where synchronization describes the agreement of data structures stored in different locations, causality describes the history of the data itself, no matter where it is stored. Causality is a weaker constraint than synchronization, but one that is much easier to achieve.

Exploring Contracts

In this chapter, we will take a tour of some important contracts that a distributed system can uphold. These will include some simple and obvious ones like "I expect to read what I just wrote." We will also see some very elusive and powerful ones like "All nodes eventually converge to the same state." We will see what it takes in general to uphold one of these contracts. Along the way, we will make trade-offs to give up some guarantees in order to gain others.

What we will find is that the contract that we think we want—the one equivalent to the assumption that data has a location—is not achievable in many situations. Instead the contract that we trade this off for will be one that permits access to data independent of location. If we recognize when we've made the assumption that data is stored in a location, we can choose instead to expect a different contract.

Identity

The first task in the quest for location independence is to separate the identity of an object from where it is stored. When location is part of identity, objects have a certain affinity for machines at that location. To achieve the best results, users should be able to identify objects just as easily from any location, without the need to communicate.

Auto-incremented IDs

Whenever a relational database is involved, you are likely to find auto-incremented IDs. Most database management systems include a mechanism for generating them on INSERT. The most common way to identify an object is to use the number that was generated when it was inserted into a table.

The auto-incremented ID is a great way to produce unique primary keys. They are monotonically increasing, which makes them ideal for clustered indexes. You will never split a page when inserting a new record with an increasing primary key. They are perfect foreign keys, much more compact than any other column that the referenced table might have. And they do not change. Most database management systems take precautions to discourage or prevent updates of auto-incremented columns. That helps preserve their uniqueness and utility as references.

So, it would seem that an auto-incremented ID would be the ideal identifier for an object in a system backed by a relational database. But while these IDs are perfect for representing identity *within* a database, they are poor choices for extending identity *beyond* the database. The convenience of doing so has made them the default choice for identity but has caused many problems downstream.

The core of the issue is that an auto-incremented ID is generated at a certain location. It only has meaning within a single database. While it's true that that single database may be clustered and spread across several machines, it is still logically a single location. It is accessible by a single connection string and may easily become unavailable to remote clients.

Environment Dependence

If you have ever promoted software from a development environment, to testing, to staging, and then to production, you are well aware that the IDs generated in each environment do not translate to the others. The object that gets ID 1337 in test will not be the same as object 1337 in production. This can be mildly annoying when you back up the testing database to restore to development in order to replicate a bug. After the restore, all IDs refer to the same objects in each environment. But as you start working with one system or the other, the IDs start to diverge. That means you cannot easily import incrementally more data from testing without dropping the development database.

It becomes more than mildly annoying when moving data between staging and production. A common practice is to back up production data and restore it in a staging environment. Then you can update the staging environment to the latest version of software, applying any necessary database updates simultaneously. After a quick smoke test, you are assured that the deployment was successful. It would be desirable at that point to just swap the staging environment into production, but that won't work unless production was taken down during this process. If it was still up, then most likely new data have been inserted into the production database, receiving new auto-incremented IDs. These IDs are meaningless in the staging database.

Auto-incrementing IDs cross the threshold from annoyance to impediment when we try to implement a warm standby disaster recovery solution. The goal is to have a replica of production data in a geographically isolated datacenter to mitigate against a

localized outage. Before the outage occurs, records stored in the production database are shipped to the remote database with as low a latency as reasonable. Latency needs to be low in order to ensure a minimum of data loss in the event of a failure. When a failure occurs, the application should "fail over" to the remote replica. Before the failover, the production database is responsible for generating IDs. After the failover, the remote database becomes responsible.

Just as the latency of the data transfer should be low, the time required for the failover should also be low. Unfortunately, latency cannot be zero, and the cut over can never be instantaneous. It is difficult to get the timing just right of importing all of the production data before turning on ID generation. Reducing latency, especially between geographically dispersed locations, becomes more expensive the closer we get to zero. Losing data during a failover can be even more costly. And the longer we wait for the data to arrive, the longer we have to postpone generating new IDs.

I have been on many long, costly projects to set up disaster recovery. Some of them have even been successful. After a few false starts, we managed to get the system to fail over reliably. But "failing back" is a much bigger challenge. After resolving the original production issue, we had to run the entire process in the opposite direction. I've never seen this done without taking the system offline for an extended period of time. It would be much easier to do if we didn't put the extra burden of generating location-specific IDs onto the database.

Parent–Child Insertion

The awkwardness of using an auto-incremented ID as identity becomes apparent when dealing with parent–child relationships. The parent record has a primary key. The child records each have a foreign key. The database enforces referential integrity of foreign keys, so the parent record must be inserted before the children. Child insertion cannot begin until the parent insertion has completed and produced the auto-incremented ID.

We don't often think about the database and the application as being two separate locations, but that is in fact what they are. The application produces INSERT instructions and transmits them to the database for execution. Under normal circumstances, the application could produce multiple INSERT statements and ask the database to execute them in a batch. But with a parent–child relationship, the application must wait until the parent insertion completes before it can learn its primary key. Only then can it generate the batch of child insertions.

Object relational mappers (ORM) perform, among other things, the task of inserting parent–child relationships. From the outside, it looks as if we can build a graph of objects and then execute a single command to save the changes. But within the ORM, that single operation is spread over several batches of INSERT commands, sent to the database in just the right order.

ORMs hide this behavior from applications as well as they can, but it does leak through the abstraction. When an object exposes the primary key as a property— so that it can use that as an external identity—that primary key is initially zero or a negative number. After the command to save the objects to the database, that primary key becomes positive. The primary key of an object is not supposed to change, but the necessity of going to a different location to generate an auto-incremented ID forces the ORM to violate that invariant.

When an application is close to its database, we can attempt to hide the truth of auto-incremented IDs within ORMs. But as a node gets further away from its central database, the dependence upon location becomes harder to conceal.

Remote Creation

Consider a mobile application. It has its own local database to store a copy of the user's data for quick access, even when the device is on a slow network. Let's further assume for simplicity that this local data has a similar schema as the central database.

When the device fetches data from the central application, it stores the objects with the provided IDs. From then on, it can present that data quickly by performing local queries against its own copy. The user can even make changes. Those updates are applied first to the local copy and then stored in a queue to be sent to the central application.

Everything is working well for queries and updates. But the problem arises when we try to insert new objects. The local database cannot use an auto-incremented ID to create new records. If it did, it would often generate an ID that the central database has already used for a different object. So, if the auto-incremented ID was used as the identity of the object, the application would have to make a round trip to the central database in order to get a correct ID.

For this reason, the simple solution is often not the one used in mobile applications. They will instead choose a local database that does not rely so heavily upon foreign keys. This at least allows the mobile client to create entire structures of objects before knowing their identity. That postpones the problem of location-specific identity far enough for

most applications, but it is not a complete solution. A complete solution would remove the location-specific component—such as the auto-incremented ID—from the identity entirely.

URLs

Web applications that follow the REST architectural style tend to use Hypertext as the Engine of Application State (HATEOAS). Every operation that the application performs is a request against a resource. With each request, the application transitions to a different state. When hypertext is used as the engine of that state, the identities of available resources are returned as references within each response.

Identity in the REST architectural style is defined by a Uniform Resource Identifier (URI). This is a hierarchical identity so that the generator can ensure that new URIs are unique. A common practice is to use the domain name of the generator as the first level of that hierarchy. A domain name identifies a small collection of nodes that are often closely located.

For an application to select and issue the next command, and so transform into the next state, it needs some way to send the command to the correct host. For this reason, the URIs used in HATEOAS are often not just identifiers, they are Uniform Resource Locators (URLs). A URL has the same hierarchical structure as a URI, but now it has an additional constraint. A URL must be addressable. It must carry enough information for an application to send a command to the host that will execute it.

URLs carry the domain name, not just as an identity namespace but also so that a client can resolve the domain name to an IP address. That IP address must be capable of routing the subsequent command to a host that will execute it. So, the domain name is closely tied to the location of the resource.

When URLs are used as the identity of resources, it can be very difficult to move a resource from one location to another. Either that new location must be addressable using the same domain name, or the identity of the resource must change. Ideally, identity would never change. It should be immutable. But on the Web, the identity of a resource changes every time the server responds with a 301 or 308 permanent redirect. The client is expected to update its reference to that resource and use the newly provided identity from then on. Unfortunately, the old identity must remain addressable to serve those 301 or 308 responses, as there is no way to know when all clients have updated their references. Clients must contact the remote server to learn the canonical form of the URL.

Location-Independent Identity

We've examined just a couple of ways that the identity of objects in an application are often coupled to their location. When identity is based on an auto-incremented ID, that ID only has meaning in a specific location and can only be generated there. When identity is based on URLs, the location of the node that responds to subsequent commands is given right in the identifier. When identity is dependent upon location, objects show a certain affinity for their location of origin. Applications start to have trouble using those objects when their locations become unavailable.

The ultimate solution to each of these problems is to identify objects without respect to location. A location-independent identity has three useful properties:

- It can be generated from any node.

- It is immutable.

- It can be compared.

Generating a unique identity from any node solves the problem of latency during remote inserts. Whether it is a geographically remote disaster recovery datacenter, or a mobile device on a slow network, a node that is capable of generating its own identities can work much faster. Immutable identities solve the problem of keeping old domains addressable indefinitely. And comparison between identities allows clients to know when they are talking about the same object. If they had to contact the origin location to learn the canonical form of the identity before comparison, they could not complete their transaction in isolation.

With a little extra thought, we can come up with identities that meet these three conditions. Such identities are not location specific and support continued operation of isolated nodes. The following are just a few examples.

Natural Keys

Probably the best example of a location-independent identity—and the one that should be the default in any application design—is the natural key. Examine the domain that you are modeling in your application. Does it already have an attribute that uniquely identifies concepts in that domain? Is that attribute immutable? If so, consider using that as a natural key within the model.

If you are building a scheduling application and need to identify rooms, look to see if the rooms are already numbered. Those numbers are good candidates for natural keys

within your system. Room numbers may change over time, but a scheduling app already takes time into account. A new room number means a new room, but past events already took place in the old room. The application doesn't care that the old room was in the same physical space.

Applications that manage articles, stories, or questions will often assign them tags. A good natural key is the canonical name of the tag in a primary language (e.g., English). The name can be canonicalized by converting all letters to lower case, dropping punctuation marks, and replacing spaces with hyphens to make them more URL friendly. A mapping will be necessary to convert the tag fermats-last-theorem to the full phrase "Fermat's Last Theorem," or to provide translations into other languages. But the natural key is easier to generate on any machine than a synthetic ID would be.

Some natural keys are primary keys generated by an external system. If you are integrating with the US tax system, you will probably identify people and companies by their tax ID. If you receive an invoice from a vendor, a good natural key for that object would be the vendor-provided invoice number. There is usually no good reason to generate a new identity when the system on the other end of an integration has already provided one.

GUIDs

When a natural key is not available, we have mechanisms for generating IDs that do not collide across machines. These are universally unique identifiers (UUIDs). Or if, like me, you came to them via Microsoft COM, globally unique identifiers (GUIDs). Whether you call it a UUID or a GUID, it is a 128-bit number represented in hexadecimal in a hyphenated format that is recognizable to most developers.

Originally, GUIDs were generated using the MAC address of the originating machine and a timestamp. Then, as GUID generation became more frequent, the timestamp was replaced with a counter. Finally, it was recognized that random GUIDs were probably just as good.

GUIDs are intended to be globally unique, but collisions have been known to occur. While some systems use a GUID to represent every row in a database, my practice has been a bit more reserved. I generate a GUID only for the most rarely created objects at the highest level and then only if natural keys are not practical.

Timestamps

One of the easiest ways to identify an object that a user has created is to use the time at which the user created it. This works well at human scales, especially when there is only one human involved. The granularity of timestamps should be less than a second to ensure that even the fastest of human-generated actions gets a unique value. Millisecond granularity is reasonable and often achievable.

While it is tempting to compare timestamps to determine which event happened before another, this should be avoided. Timestamps are only increasing within a single machine. And even then, the clock of the machine may be adjusted forward or backward. Adjustments such as daylight saving and crossing time zones are not the concern; timestamps should always be captured in UTC. But small corrections to fix clock drift should be allowed.

Timestamp alone is not sufficient to identify objects in a system with a large number of users. They should only be used in combination with other forms of identity.

Tuples

Using just one identity, like a timestamp, is often not enough to avoid collisions. But bring different forms of identity together, and the combination is stronger than any of its parts. A tuple is an ordered list of values, where each member has its own type and meaning.

Tuples are often written as a parenthesized list: (that-conference, day-2, 10:00, 136). But it is just as valid to write a tuple as a path: /that-conference/day-2/1000/136. This gives them a hierarchical feel that makes them suitable for use in URLs. (Yes, URLs can be used in an application, just not as identities of objects.) The hierarchy implies that the object has just one owner, which is identified by the tuple having one fewer element. In the preceding example, the session held in room 136 is owned by the 10:00 time slot on day 2.

The transparent nature of tuples makes them susceptible to human interpretation. This is both a benefit and a drawback. While it is often useful to be able to see the implied relationships between objects just by their identities, this can sometimes cause confusion. In some cases, a strict hierarchy does not exist, yet the tuple implies one by its choice of values and order. And in other cases, the values in the tuple represent mutable concepts. We can choose either to change these values, and thus change the identity of an object, or to keep the old values and risk confusion.

Hashes

To avoid the confusion caused by a transparent data structure like a tuple, we can instead choose an opaque structure like a hash. A hash function takes a tuple as an input and produces a value. The function is deterministic: the same tuple will always produce the same hash. But ideally, the function should also be unpredictable: it should be hard to find a tuple that produces a given hash.

Hashes have additional benefits over their source tuples. Where a tuple contains elements of variable length, like strings, hashes are always the same size. Furthermore, while tuples tend to chunk data together, hashes tend to spread it apart. And while tuples can be easily reverse engineered, hashes are one way. This makes them better suited to problems that require a degree of security.

Many systems that use hashes for identity choose to do so for one of these reasons. Blockchains use hashes to identify transactions so that the contents cannot be easily altered. Changing one element of a transaction—such as the sender, recipient, or amount—will alter the hash. And finding a different transaction that produces the same hash is an intractable problem.

Git uses hashes to identify commits. It does so not for their security. Instead, since Git is based on the file system, having an identifier of constant size helps them fit into file names and data structures. The tuple that it starts with includes the name and email of the author (natural keys), the differences between the two versions, and a timestamp (to the second). That source tuple is of variable length and can be quite large for significant differences. The resulting hash, however, is 256 bits, or 64 hexadecimal digits.

Public Keys

In keeping with the security theme, public keys are excellent ways to identify principals such as individuals or corporations. Public keys are often used to digitally sign messages, proving their authenticity. Only someone with access to the private key could produce the signature.

A certificate is a fully vetted identifier for a principal, often including their name, physical location or legal jurisdiction, and identity of the vetting party. Certificates form their own kind of hierarchy, as the identity of the party who signed the certificate is provided as a public key.

Blockchain systems use a public key as the only means of identifying a party. Each transaction records the sender and recipient by their public keys. To pay someone in

Bitcoin, you need only know their public key. That is sufficient to identify them uniquely to any node within the distributed network.

Random Numbers

When other forms of identity are not available, an application can always fall back on random numbers. Public keys are really nothing more than two random numbers that have been verified to be prime and then multiplied. And modern GUIDs are often generated completely at random, rather than using MAC address or timestamp. So it is a valid choice to simply use random numbers directly, as long as they are big enough and random enough.

Like timestamps, random numbers should never be used as the only form of identity. They should be combined with other identifiers to create a tuple. Since the random number is not fit for human interpretation, producing a hash of that tuple is often the next step. In cryptography, a random number added to a tuple prior to hashing is called a "nonce," a number used once. In this case, we are using the nonce to distinguish an object from others that share the same tuple values.

When choosing a random number generator, it is best to stick with a cryptographically strong algorithm. Algorithms used to generate public keys, shared secrets, and nonces are specifically selected to produce unpredictable results. While you will most likely not be relying upon these random number generators for securing data, you will be using their output as part of an object's identity. Having two nodes use the same predictable random number generator means that the chance of a collision is high.

Choose the most appropriate mechanism for generating unique identities for objects. Whatever method you choose, avoid anything that would tie the identity of an object to the location that generated it. Instead, choose a generator that meets the following criteria:

- Any node should be equally capable of generating identities without consulting a central database.

- Identity must be immutable.

- Peers should be able to compare identities to know when they are talking about the same object.

Identity is the first step to location independence. The next step is to ensure consistent behavior without respect to location.

Causality

As we begin to reason about the behavior of a distributed system, we are going to try to construct a chain of events. Our goal is to predict what will happen at some distant node sometime in the future. The way to get to that prediction is to analyze the effect that local actions may cause.

Causality itself is a hard concept to measure. You can say that tipping one domino caused the next one to fall. But would the second one have fallen on its own? We would like to say for certain that it would not. However, as anyone who has built a large domino chain knows, that is a hard claim to assert.

The causes of many events in a distributed system can be just as complex and inscrutable as a chain of dominoes. And yet we still desire some predictability from the system. And so, we have to find a reasonable stand-in for causality that is easier to measure and useful for making predictions.

While we cannot always say with certainty that one event caused another, we can say for certain that the cause happened before the effect. As this book is being written, time travel is still impossible. Perhaps "happened before" is enough. Maybe it is sufficient to use the order of events as a stand-in for causality. Let's apply this notion of causality to steps in a program and compare this with our intuition.

Putting Steps in Order

We often think about a program as a sequence of steps. The steps happen in order as the program executes. It is easy to look at two steps executing in the same program, such as the one in Figure 4-1, and say that one happened before another.


```
for i = 1 to 100
    if i mod 15 == 0
        print "Fizbuzz"
    else if i mod 3 == 0
        print "Fizz"
    else if i mod 5 == 0
        print "Buzz"
    else
        print i
```

$- - -$ 6 mod 15 == 0 $\longrightarrow$ 6 mod 3 == 0 $\longrightarrow$ Print "Fizz" $- -$

Figure 4-1. *Steps in a process*

Using the order of steps as a stand-in for causality leads us to say that one step in a program causes the next. In some sense, this is true. A program executes sequentially, so it's reasonable to say that executing one step will cause the computer to then execute the next one. Even if the two steps operate on different objects and do not depend upon one another, they are at least temporally coupled. The program would not get to the second step without having executed the first.

You may be thinking that a `goto` statement that jumps to the second step violates this notion of causality. The program executes the second step without having executed the first. However, in this situation we would observe that the `goto` happened before the second step. It is not the order in which the steps appear in code that is interesting to us. It is the order in which they occur at runtime. And so it was the `goto` that caused the step to occur. This agrees with our intuition about a statement that causes execution to jump to another. "Happened before" is looking like a good measure of causality.

When we try to generalize steps in a single program to multi-threaded or multi-process systems, things get a little trickier. We cannot say quite so clearly which of two steps executing in different processes happened before the other. The processes can be running on parallel threads or even on different machines. There is no single clock that can help us to put those steps in order.

We can, however, observe that two processes running independently do not cause any behavioral changes in one another. They are not causally connected. As long as they don't communicate, nothing that happens in one can influence the other.

When they *do* communicate, causality is clearly asserted. If one process sends a message, and another process receives it, then we know that the *send* step happened before the *receive* step. And in a very real sense, the sending of a message caused its receipt. With this fact in hand, we can start to causally order steps that have occurred in different processes. This is precisely how Leslie Lamport defined the order of events in his 1978 paper on distributed systems.[1]

The Transitive Property

The relationship that one step happened before another has another useful property: it is *transitive*. That is, if one step happened before a second, and the second happened before a third, then we know that the first happened before the third. This is easy to

---
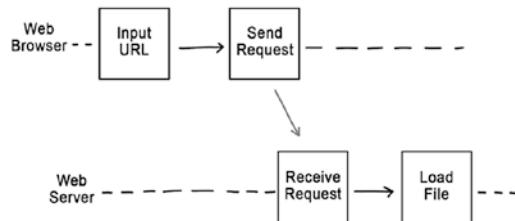
[1]Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7):558–565, July 1978.

see when all of the steps are in the same process. Those steps are in sequence. But the transitive property holds just as well when we cross process boundaries.

Take, for example, a web browser. The user commands the browser to navigate to a given URL. The browser has just executed two steps: input a URL from the user and send a request to the web server. Since these steps happened in the same process, we know that one happened before the other: "input URL" happened before "send request."

Now let's look at what happens in the web server. When it receives the request, it will load the requested file from the hard drive. The receipt of the request and the loading of the file are two steps within the same process. We can therefore say that the receipt happened before the load.

Since these two processes are talking to each other by passing messages, we can also put some of the steps across processes in order. We can say for certain that the send of the request happened before its receipt, even though they happened in different processes. As Figure 4-2 illustrates, the transitive property allows us to then chain these events together. We can say for certain that the user input happened before the file load.

Figure 4-2. *Order of steps in two processes*
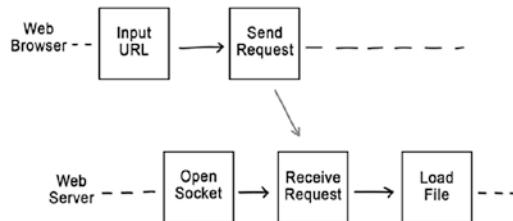
Because we are using "happened before" as a stand-in for causality, what we are really asserting here is that the user input *caused* the file to be loaded. This fits well with our intuition. We can imagine that the user intended for the web server to load the file, and so this causal chain of events served to realize the user's intent. We might also assume that the web server probably would not have loaded that particular file, had the user not entered the URL. But intent and might-have-been are difficult to reason about. "Happened before," however, is very clear.

We can clearly state when we know one step happened before another. We can also clearly state when we have no idea.

Concurrency

While it was possible in our previous example to say for certain that user input happened before loading a file, it will not always be the case. Some steps happening in different processes will not be so easily put in order, even using the transitive property.

Take, for example, the step where the web server opens a socket. As Figure 4-3 demonstrates, "open socket" happens before "receive request" and is executed in the same process. And as we saw in the previous analysis, "input URL" also happened before "receive request." But the transitive property does not allow us to say which of "open socket" and "input URL" happened before the other. They both happened before "receive request," but that doesn't imply anything about their relative order. Lamport called two steps that cannot be put in order *concurrent*.

Figure 4-3. *No causal connection between "open socket" and "input URL"*

This definition of concurrent is a bit different than others that you might have heard. Concurrent operations in a multi-threaded system might be running in parallel. You have the sense that if two events are concurrent, then they happen at the same time.

By Lamport's definition of concurrent, we don't know whether two steps indeed happened at the same time. They could have been separated by a large span of actual time on a physical clock. It could be, for example, that the web server opened the socket hours before the user input the URL. In fact, that is quite likely. But what could be and what is likely do not hold sway in this conversation. It is precisely the fact that *we cannot know* that makes these two events concurrent.

In a very real sense, concurrency is what makes distributed systems so difficult to think about. If there were no concurrent steps, we could put all of the steps in order. If every step can be ordered relative to every other step, then we would end up with a totally ordered sequence. It would be much easier to think about that kind of system, because it always behaves as if the whole network is running on a single machine.

While a totally ordered system would be easier to think about, it would not have the properties that we desire in a distributed system. It would not scale as we added more hardware, since totally ordered steps cannot be run in parallel. It cannot autonomously serve clients in different locations, because the steps the program takes to serve one client would need to be put in order with others in real time. And it would not allow for disconnected operation, since the steps running on the disconnected computer would be out of sequence with the rest of the network. And so, concurrency is both the hero and the villain of this story.

Partial Order

If you were to compare any two steps running in the same process, you could tell which of the two came first. Those steps are *totally ordered*. They happen in sequence.

If, however, you compare two steps running in different processes, you *might* be able to tell which came first. If one preceded the sending of a message, the receipt of which preceded the second, then the transitive property tells us that the first happened before the second. But if that is not the case—if the two steps are concurrent—then you cannot tell which came first. Because sometimes you can tell and sometimes you can't, the execution of steps in a multi-process system is said to be *partially ordered*.

Since we are using "happened before" as a stand-in for causality, we can say that causality itself is partially ordered. Some things are causally related: we can clearly say which is the cause and which is the effect. The user input of a URL into a browser caused the web server to load a file. But some things are not causally related. The web server opening a socket did not cause the user to input a URL, nor did the input of the URL cause the web server to open a socket.

Partial order imposes fewer constraints on a system than does total order. It frees up some steps to happen in parallel. It permits devices to act autonomously while disconnected. It gives nodes the ability to act independently without constant synchronization. Recognizing that causality is partially ordered gives us a powerful tool for analyzing distributed systems. We can better understand their capabilities as well as their limitations. And we can make better choices about trade-offs between the two.

The CAP Theorem

Probably the most famous mathematical idea in all of distributed systems is the CAP Theorem. It was postulated by Eric Brewer at the 2000 Symposium on the Principles of Distributed Computing.[2] Formally proven by Seth Gilbert and Nancy Lynch in 2002,[3] the CAP Theorem relates the ideas of consistency, availability, and partition tolerance. It is often quoted as saying you can only have two of the three.

Consistency means different things in different contexts. Unfortunately, as it usually appears, it doesn't have a very useful definition. For example, if you're familiar with relational databases, then you probably first heard of consistency as it relates to the ACID properties of a transaction: atomic, consistent, isolated, and durable. Atomic is easily defined as all or nothing. Isolated simply means that concurrent transactions don't affect one another. And durable means that the change persisted.

But consistent in this context is not so easy to define. The working definition is that a consistent transaction is one that does not violate any invariants. It "commits only legal results."[4] The trouble is that the invariants that define a legal result come from two sources: the database management system and the application. Database management system invariants include guarantees like "primary keys are unique" and "foreign keys reference rows that exist." Application-defined invariants, when they exist at all, are defined in terms of the problem domain, such as "all balances are zero or positive." If we were talking only about the well-defined guarantees generally adopted by database management systems, we might have some chance of proving some generally applicable theorems. But with all of the choices that an application can make in determining its own domain-specific invariants, we find it very difficult to write a meaningful proof. Therefore, we will use a more precise definition.

Defining CAP

The definition of **consistency** that the CAP Theorem uses is specifically related to nodes in a distributed system. It says that if I ask two different nodes for a value, they will give

---
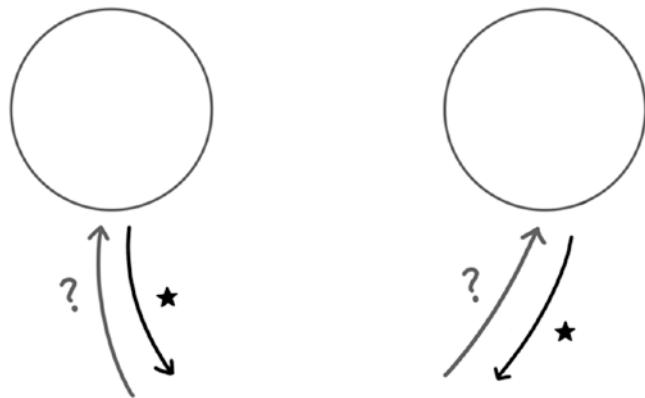
[2]E. Brewer, "Towards Robust Distributed Systems," Proc. 19th Ann. ACM Symposium on the Principles of Distributed Computing (PODC 00), ACM, 2000, pp. 7-10.

[3]Seth Gilbert and Nancy Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51–59.

[4]Haerder, T; Reuter, A. December 1983. "Principles of Transaction-Oriented Database Recovery." Computing Surveys. 15 (4): 287–317.
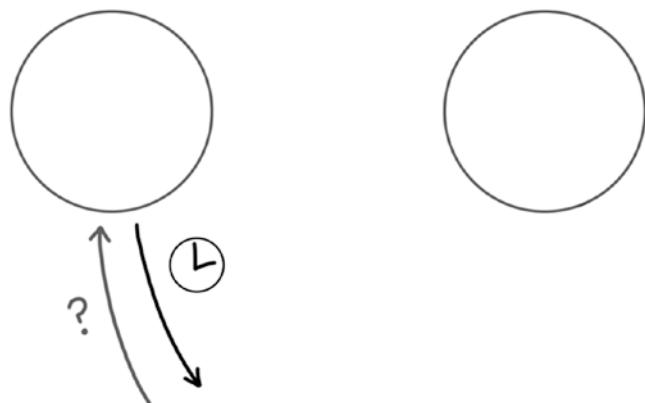
me the same answer, as illustrated in Figure 4-4. If the nodes are consistent, then their answers will agree. If the answers disagree, then the nodes are not consistent.

Figure 4-4. *Consistency*
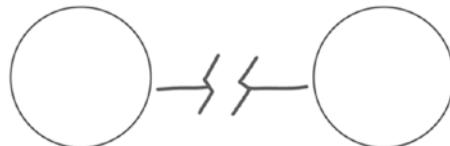
That's very different from the definition used in ACID. In fact, you could argue that it's closer to atomic than to consistent. Either both of the nodes have the latest version of a value, or neither does. But where atomic—and indeed each of the ACID guarantees—is about changes to a single database, consistent in CAP is about nodes in a distributed system.

Continuing on, the A in CAP is for **availability**. A node is available if it responds in a reasonable amount of time to any request, as shown in Figure 4-5. This leaves one to ask: "What is a reasonable amount of time?" The answer to that is "About the time it takes to heal a network partition."

Figure 4-5. *Availability*

So, what is a network partition? That's the P in CAP: **partition tolerance**. A network partition is a condition that prevents messages from flowing in a network, indicated in Figure 4-6. Partitions are only temporary, however. After some period of time, the connection will be restored. But in the meantime, partition tolerance promises that the system will continue to function.

Figure 4-6. *Partition tolerance*

Armed with these definitions, we can finally state the assertion of Brewer's conjecture. No distributed system, no matter what algorithm it uses, can simultaneously guarantee consistency, availability, and partition tolerance at any given interval. If during that interval the network is partitioned, then the system will either be inconsistent or unavailable.
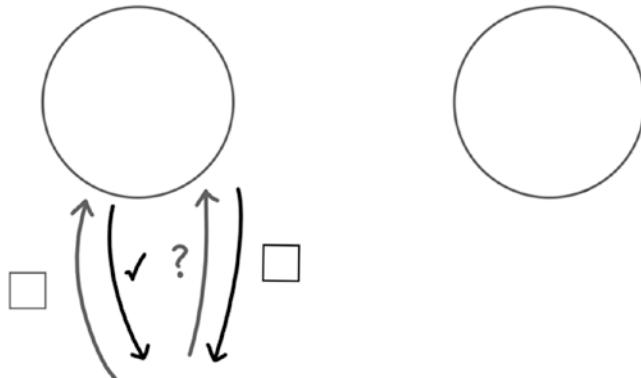
This is one of those delightful theorems that challenge you to find an algorithm that works, like Gödel asking you to write a formula that determines whether another expression is true, or Turing imploring you to write a program that determines whether another program terminates, or two generals commanding you to find a reliable way for them to communicate. The proof doesn't have to guess what you might come up with. It can simply demonstrate, by pain of logic, that whatever you've dreamed up will not be equal to the task.

Proving the CAP Theorem

Imagine that you have a system made up of different computers, which we'll call nodes. Each node has its own internal state. That state, however, is invisible to us. The only thing we can do as an outside observer is to send messages to the nodes and see how they respond.

The message that we will send to the nodes will be *read* and *update*. If we send a node a *read*, it will send us back a value. If we send it an *update*, it will presumably write down the value and then respond with confirmation. I say *presumably*, because we can't really see its internal state.

The only way we can observe a node is by sending it messages. And the messages have the following contract: if I *update* a node in isolation, and then after it confirms, send it a *read*, it will return the value that I just updated, as in Figure 4-7. The node is acting as if it is saving the state for us to later retrieve.

Figure 4-7. *Update and read*

Be careful here. We can't really tell what these nodes are doing. Their internal operation is left unconstrained. That is important for the proof to be general. If we dictated that they truly *were* storing internal state, that would limit the kinds of algorithms we could make assertions about.

Similarly, the messages *read* and *update* do not constrain our choice in the algorithm either. We do not have to devise a way to send reads and updates to achieve consistency. In fact, these two messages might not even be used by the algorithm. They only exist as a way of setting up a test.

Test an Algorithm
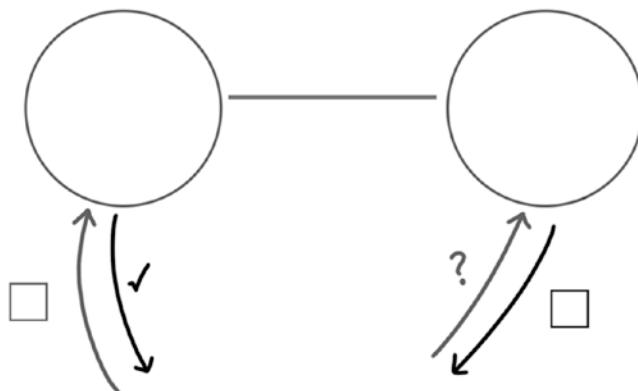
And so, this is the challenge. I ask you to provide an algorithm. You can devise any algorithm you like. You choose the steps. You choose the data structures. I will load this algorithm into two nodes. They communicate with one another by passing messages between them. You choose what messages they will use.

Then, I'll run a test. I will begin by observing that the nodes are initially consistent. I can tell that they are consistent by sending *read* messages to each and observing that they return the same answer.

Next, I'll send one of them an *update* and wait for it to respond with confirmation. Since it confirmed, I can test the contract by sending that same node a *read*. If it is

behaving properly, it will return that value I just sent it. In about half of the tests, I'll perform this check. I'll reject your algorithm if it ever fails to uphold the contract.

In the other half of the tests, I'm going to turn to its neighbor to perform the *read*, as in Figure 4-8. If I get the same value that I just *update*d, then the system is demonstrating *consistency*. If both the *update* and the *read* return within a certain interval of time, then the system is demonstrating *availability*. To be completely fair, I will even let you tell me what the interval should be.

Figure 4-8. *Test the algorithm*
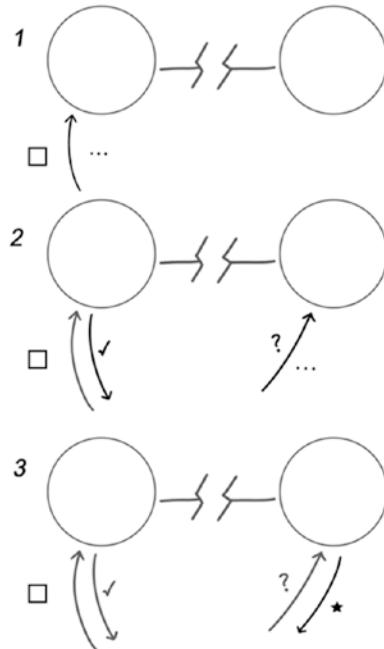
But that's where I play my trick. During the test, I will create a network partition. The two nodes will not be able to communicate with one another during this interval. The partition will last just a little bit longer than the duration you defined. While communication will eventually be restored, it will not be fast enough for the algorithm to exhibit availability and still be consistent.

And so, the algorithm is going to have to choose one of three behaviors, illustrated in Figure 4-9.

1. The first node might block during *update* until it can communicate the value and then confirm the result. If so, then *update* takes longer than the specified interval, and so the system is not available.

2. The second node might block during *read* until it can retrieve the value from the first. If so, then *read* takes longer than the specified interval, and so again the system is not available.

116

3. The system might decide to return before the interval has expired. If so, there was no way that the value I *update*d will be able to propagate to the second node to be *read*. The second node cannot return the same value as the first, and so the system is not consistent.

Figure 4-9. *Three possible behaviors when the network is partitioned*

During this interval of network Partition, the system cannot be both consistent and available. And so, it seems that we are doomed to choose.

Eventual Consistency

If we cannot expect different nodes within a distributed system to have the same state, then what can we hope to achieve? How can we get any work done if we get a different answer from every node that we ask?

Consistency at any given instant may be out of our reach, but all hope is not lost. We can achieve consistency if we wait long enough. Eventually, nodes will come into agreement with one another. This is a concept referred to as eventual consistency.[5]

While it might be desirable to demand consistency at any given instant, it might not be practical. If we loosen our constraints, we find that we can achieve a much more palatable trade-off. Instead of insisting upon consistency at every given instant, perhaps we can tolerate a lesser degree of agreement. The conversation needs to get a bit more nuanced.

Kinds of Consistency

Marc Shapiro, a researcher at the French National Institute for Computer Science and Control Science (Inria), and Nuno Preguiça, associate professor at Faculdade de Ciências e Tecnologia da Universidade (FCT), sought to understand consistency trade-offs at a formal level. They had each designed special-purpose solutions to achieve eventual consistency, including Treedoc, a replicated data structure for collaborative text editing.[6] Each one of these projects required its own formal proof. They wanted a more general result.

Based on their prior results, Shapiro and Preguiça, together with their colleagues, identified three different kinds of consistency.[7] The distinctions among them lead to the general result that they sought. They redefined the kind of consistency used in the CAP Theorem as **strong consistency**. That is the guarantee that all nodes will report being in the same state at any given time. They used the term **eventual consistency**, on the other hand, to mean that nodes will eventually reach the same state, as long as they can continue to talk to one another. This may require some additional consensus algorithm, such as conflict resolution.

The reliance upon consensus algorithms introduces more than a small degree of overhead. The nodes might need to elect a master to make the final decision,

[5]Terry, D. B.; Theimer, M. M.; Petersen, K.; Demers, A. J.; Spreitzer, M. J.; Hauser, C. H. (1995). "Managing update conflicts in Bayou, a weakly connected replicated storage system." Proceedings of the fifteenth ACM symposium on Operating systems principles - SOSP '95. p. 172.

[6]Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, Mihai Leţia. A commutative replicated data type for cooperative editing. 29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), Jun 2009, Montreal, Québec, Canada. pp.395-403, ff10.1109/ICDCS.2009.20ff. ffinria00445975.

[7]Shapiro, Marc; Preguiça, Nuno; Baquero, Carlos; Zawirski, Marek. Conflict-free Replicated Data Types. Institut National de Recherche en Informatique et en Automatique, No 7687, 2011.

introducing a bottleneck. Or they might run a complicated and chatty algorithm like Paxos to determine by majority decision what the final state shall be. For these reasons, Shapiro and Preguiça decided to distinguish a third kind of consistency. **Strong eventual consistency** promises that all nodes reach the same state the moment they all receive the same updates. The nodes do not need to talk among themselves to reach a consensus and resolve conflicts.

The CAP Theorem showed us that strong consistency is incompatible with availability. Allowing for consensus algorithm means that the eventual consistency may incur some undesirable overhead. And so, we, like Shapiro and Preguiça, will focus our attention on strong eventual consistency (SEC).

Strong Eventual Consistency in a Relay-Based System

With SEC as our stated goal, let's construct a useful example. Let's build a distributed system based on relaying messages and see what properties it must have to satisfy SEC.

This distributed system is made up of nodes connected in some kind of network. The network is connected, which is to say that, unless the network is partitioned (which will occur from time to time), there is a path from any node to any other node. These paths don't have to be direct; they may go through any number of intermediate nodes.

Some nodes receive new information from outside of the network. When they do, they formulate a message that they themselves process and then send along the network to neighboring nodes. When a neighbor receives the message, it processes it and relays to *its* neighbors. Each node is running some kind of algorithm to determine when to forward a message and to whom. That algorithm guarantees that eventually, every node will receive every message.

It's important to observe that we are explicitly *not* requiring that every node receive each message exactly once. Nor are we requiring that every node receive the messages in the same order. Whatever forwarding algorithm we come up with only has to ensure eventual delivery.

Now let's consider the internal state of a node within the distributed system. As the node processes a message, it transitions from one state to another. The message can be viewed as a function, taking the starting state as an input and producing the resulting state as the output. The system is strongly eventually consistent (SEC) if, after seeing all of the messages, all nodes arrive at the same state. We can determine what properties those functions must have in order to achieve SEC.

First, every node's response to each message must be **idempotent**. If a node sees the same message twice in a row, then it must end up in the same state as if it had seen it only once. And second, every node's response to each *pair* of messages must be **commutative**. If the node sees two messages in one order, it must end up in the same state as if it had seen them in the opposite order.

Taken together, idempotence and commutativity are sufficient to prove SEC. So long, that is, as every node eventually sees every message at least once. This result is only valid for the kind of relay-based distributed system that we defined. It assumes that messages are forwarded exactly as they are, not filtered, altered, or summarized. We will find a more general result in the next section, but for now, let's examine this relay-based system.

Idempotence and Commutativity

Mathias Verraes joked on Twitter (Figure 4-10):
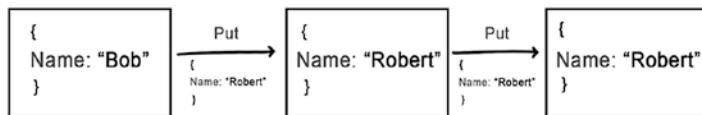
Figure 4-10. *https://twitter.com/mathiasverraes/status/632260618 599403520*

Like all good jokes, this one is absolutely true. It is hard to guarantee that a message is delivered exactly once—not lost and not duplicated. It is even harder to guarantee that messages will arrive in the order in which they were sent.

Network protocols have been invented to specifically try to address these two hard problems. AMQP, for example, is a messaging protocol that can be configured to deliver a range of guarantees. It can be used as a best-effort transport, in which the message is guaranteed to be received no more than once. It can also be tuned up to reliable delivery, which guarantees that a message will be received at least once, but possibly more than once and possibly out of order. With a bit more overhead, it can perform anti-duplication, which attempts to guarantee exactly once delivery. And with a herculean amount of effort, it can serialize messages in a channel, guaranteeing that they will be delivered in the same order they are sent, though you wouldn't be happy with the performance.

Authors of infrastructure components that rely upon AMQP, such as RabbitMQ, often advise that a consumer be written to tolerate duplicate messages.[8] The cost of running a message queue with anti-duplication or serialized channels can be prohibitive. Instead, they recommend that you make your downstream nodes tolerate messages that arrive multiple times, or out of order. That's precisely what idempotence and commutativity mean.

A downstream node that tolerates duplicate messages is *idempotent*. It will remain in its current state upon seeing the duplicate message. A classic example of an idempotent node is an HTTP server receiving a PUT message. The message carries the desired state of the resource given by the URI. If it receives the PUT message a second time, the HTTP server simply sets the desired state again, as demonstrated in Figure 4-11. The end result is the same as if the HTTP server had received only one PUT message.

Figure 4-11. Idempotent put

A downstream node that tolerates out-of-order messages is said to be *commutative*. This comes from the mathematical commutative property, which says that an operator has the same result no matter which way its operands are given. The commutative property of addition says that $a+b = b+a$. Multiplication is also commutative, but

---
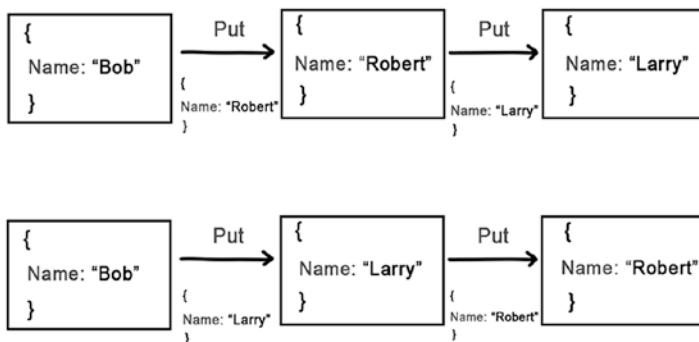
[8]"[C]onsumer applications will need to perform deduplication or handle incoming messages in an idempotent manner." RabbitMQ Reliability Guide. https://www.rabbitmq.com/reliability.html

subtraction and division are not. In a similar sense, a node is commutative with respect to two messages if it ends up in the same state no matter which message it sees first.

Deriving Strong Eventual Consistency

A node might be idempotent with respect to a set of messages, but not commutative. For example, an HTTP server receiving two different PUT requests for the same resource will behave differently based on the order. The resource will end up in the state described by the last message it sees. Change the order of the messages, and you change the final state of the resource, as in Figure 4-12.

Figure 4-12. *Non-commutative PUT*
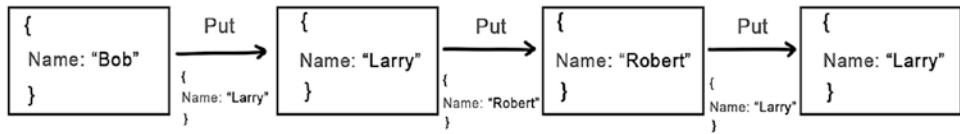
Strong eventual consistency requires both idempotence and commutativity. Let's go back to our working definition of strong eventual consistency to see why this is the case.

A relay-based distributed system is SEC if all nodes, upon seeing the same set of messages at least once in any order, reach the same state. Of course, they must all start in the same state. If the set of messages was empty, the problem would not be interesting: all nodes would still be in the start state. And if the set contained only one message, eventual consistency would only rely upon idempotence. Nodes that receive duplicate copies of that one message will remain in the same state.

And so, we only need to carefully consider the case in which the set contains more than one message. Let's consider how this might play out. If every node received each message exactly once, then we could argue based on commutativity alone that they would all reach the same state. Or, if every node received each message in order, but some were doubled or tripled, then we could argue based only on idempotence. It's

the fact that things can get jumbled up that causes us to have to stop and consider the possibilities.

Take, for example, a pair of PUT requests to an HTTP server. As we noted previously, PUT is idempotent, but it is not commutative. So, if an HTTP server sees the same PUT message duplicated immediately, it will not change state. However, if there were some intervening messages in between the duplicates, then we have a problem. If a second PUT was received *between* the first one and its duplicate, as in Figure 4-13, then the HTTP server would overwrite its change when the duplicate arrives. In order to behave in an eventually consistent manner, the node would have to ignore the duplicate altogether.

Figure 4-13. *Intervening PUT*

And so for a given set of messages *{ m1, m2, m3, ... }*, the scenario that we have to carefully consider is if *m1* is duplicated after having received some number of intervening messages. We would like to say that

m1+m2+m3+m1 =

m1+m2+m3

As it turns out, we can indeed prove this for an idempotent and commutative set of operations. First, we observe that the duplicated message (*m1*) is commutative with the message that was received just before it (*m3* in this example). We can therefore swap their places without changing the state of the node. And so

m1+m2+m3+m1 =

m1+m2+m1+m3

We just swapped the *m1* and *m3* at the end. This moves the duplication of the message one step earlier in the sequence.

But now, we observe that we can use the commutative property again, this time with *m2*. That is to say

m1+m2+m1 =

m1+m1+m2

And so, the duplication moves one more step earlier. We can keep doing this until we have moved the duplicated message right up next to the original. At this point, we simply employ the idempotent property of the duplicated message to assert that receiving it twice is just as good as receiving it once. In other words

m1+m1 =

m1

And so, we have shown that, because the node is both idempotent and commutative with respect to the set of messages, it will reach the same state after seeing one of the messages duplicated, no matter how many other messages have intervened:

m1+m2+m3+m1 =

m1+m2+m3

And this generalizes to any number of messages. This reduces the problem back down to receiving some set of messages in any order, but with no duplicates. We only need to rely upon commutativity to ensure that any such sequence will yield the same result.

And that is why our PUT example does not exhibit strong eventual consistency. While it is idempotent, it is not commutative. Both properties are required to achieve SEC in a relay-based distributed system.
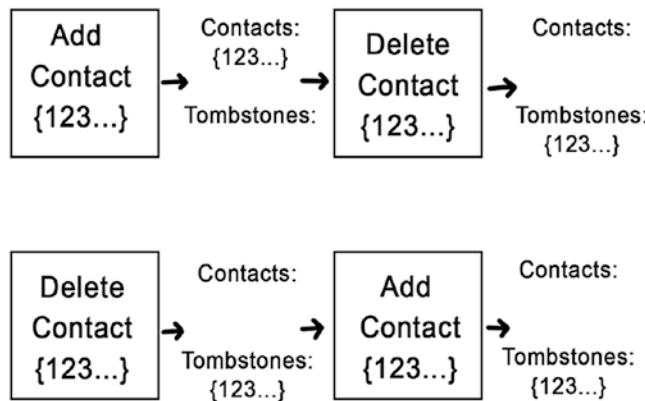
The Contact Management System

A friend and I created a contact management system, back in the days when personal digital assistants (PDAs) connected to your workstation via RS-232 serial port. At the time, the state of the art was Microsoft's ActiveSync. We thought we could build a better product.

The solution we came up with was a message store-and-forward system where the nodes (workstations and PDAs) processed messages in an idempotent and commutative fashion. The messages included things like "add contact," "update contact," and "delete

contact." Contacts were uniquely identified by GUID, which made add operations trivially idempotent and mutually commutative.

Delete operations took a little more work in order to commute with adds. If the delete is processed first and the add second, the result should be the same as if they were handled in the usual order. That is to say, delete followed by add should result in the contact being absent. We accomplished this by keeping a list of all contact GUIDs that had been deleted, even if the contact itself was not present at the time. And then, when the add was processed, if its GUID was in the list (what is commonly referred to as a tombstone, shown in Figure 4-14), then the contact was not added.
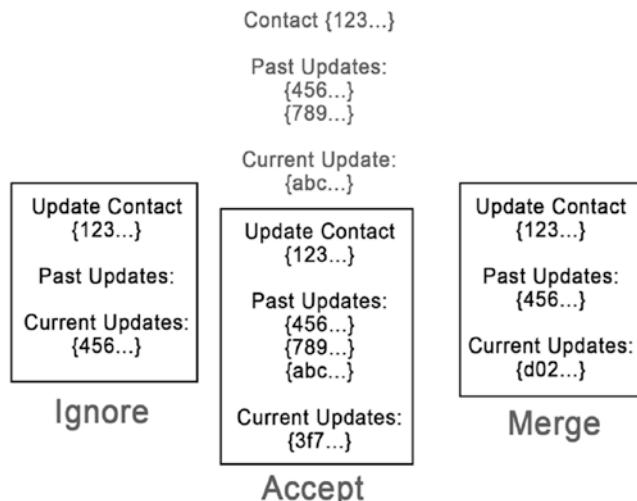
Figure 4-14. *Tombstones*

Update operations were the hardest to get right. As with HTTP PUT, the trivial implementation of update is idempotent but not commutative. To solve this problem, we assigned each update message a GUID as well. Each node kept track of the GUID of the most recent update that set a contact's properties. It would also keep a list of update GUIDs that it saw in the past. When the user changed a contact, it would add the current GUID to the list of past GUIDs and then generate a new current GUID. It included both the current GUID and the list of past GUIDs with the update message.

When a node received an update, it would first check whether that update's current GUID was already in its own list of past updates. If so, it would ignore the update. If not, it would perform the opposite check: was its current GUID in the message's list of past GUIDs? If this second check passed, it would accept the update, taking the entire list of past updates.

As long as one of these two checks pass, then updates commute with one another. When received out of order, the future update will deliver the GUID of the past update. The past update would subsequently be ignored.

If both checks fail, however, there is more work to do. In this scenario, a concurrent update is detected. The user had modified the same contact on two different nodes while they were disconnected. Our response to this was to merge the two sets of properties. Where a field, such as phone number, was the same, we kept that value. Where they were different, we just concatenated them. That meant each of the fields allowed for multiple values. Fortunately, it's already understood that a contact can have multiple phone numbers. Figure 4-15 shows examples of these three scenarios.

Figure 4-15. *Compare past and current GUIDs*

In addition to merging the properties, the node would also merge the GUIDs. The list of past GUIDs was the union of the current and incoming lists. And the current GUID? That's where our data structure was a little more complicated than what I first described. The current GUID was also a list. Usually it contained only one element. But after a merge, it contained two (or even more if additional concurrent updates were detected).

This merge is commutative (ignoring concatenation order, which we were happy to do). Each side of the concurrent update would perform the merge upon seeing the other's message. They would both get the same list of past GUIDs, and they would both get the same list of current GUIDs. When the user subsequently edited this merged
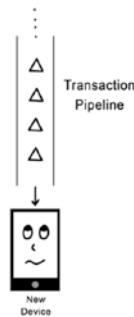
contact, both of the current GUIDs would be added to the past list. And so both sides would happily replace its automated merge with the user's manual one.

Replaying History

This solution worked pretty well. It was strongly eventually consistent (though we didn't know that term at the time). We proudly showed prospective buyers that we could disconnect a PDA, make changes, and then sync it back up. After all of the messages flowed back and forth, all clients had the same list of contacts in the same state.

During the synchronization process, however, things looked a little sketchy. If a large number of changes had happened on one side, all of those edits would replay before our eyes on the other. Given the speed of networks at the time, you could easily read the list of names as they were added, modified, and subsequently deleted while history replayed on the device.

Adding a new device to this system revealed the full extent of the issue. Since it was based entirely on processing messages exactly as they had been originally sent, the entire history of messages was persisted in a central repository. We referred to this as the transaction pipeline. When a new device was introduced to the transaction pipeline for the first time, as in Figure 4-16, it would pull down and process every one of those messages. That means that it would see all of the past edits. It would even see contacts that had long since been deleted. As history grew, the time required to add a new device grew proportionally.

Figure 4-16. *A new device is introduced to the transaction pipeline*

My friend and I never sold an installation of this contact management system. In the end, it proved to be just as clunky as the Microsoft product that we were competing against. Perhaps we could have found a way to prune history, or to download snapshots.

But if we had known about conflict-free replicated data types, they would have offered a better solution.

Conflict-Free Replicated Data Types (CRDTs)

We achieved a useful result for a distributed system based on processing and forwarding messages. If every node sees every message, and nodes forward the messages unaltered, then two properties are sufficient to achieve strong eventual consistency (SEC):

- Idempotence (ignore duplicates)
- Commutativity (don't depend upon order)

Prove these properties about the way nodes process messages, and you already have a very reliable system. I have built many systems using exactly this technique. It pairs well with infrastructure components such as Amazon SQS, Rabbit MQ, and MSMQ that ensure broadcast and delivery of messages. It requires only a minimum set of guarantees from those components, helping them to work at scale without becoming over-constrained.

But this isn't the most general result. We can optimize our distributed system further if we allow nodes to modify messages. Instead of requiring that a node forwards exactly the same messages it received, we can allow the node to summarize its knowledge and send fewer messages. This is the strategy employed by conflict-free replicated data types (CRDTs).

State-Based CRDTs

Shapiro, Preguiça, and colleagues described two general solutions to the strong eventual consistency problem: state-based CRDTs and operation-based CRDTs. Operation-based CRDTs require a delivery protocol that ensures once-and-only-once delivery and preserves causal order. We would prefer to find a solution that does not place so high a constraint on infrastructure components. Fortunately, state-based CRDTs have no such restriction. State-based and operation-based CRDTs can each emulate one another and are therefore equivalent. For these reasons, we can put aside operation-based CRDTs for this discussion and focus entirely on the state-based variety.

A conflict-free replicated data type is a data structure that exists not at one location, like a typical abstract data type; it exists in multiple locations. Each node in a distributed

system has its own replica of the CRDT. Operating on a replica of a CRDT closely meets our requirements for location independence. Each replica can serve queries in isolation, without communicating with other replicas. And each replica can process commands that immediately alter its state. The effects of these commands will be shared with other replicas in an eventually consistent manner. All replicas will converge to the same state when all updates have been delivered.

The key is to understand what it means for an update to be delivered.

Partially Ordered State

Each replica of a state-based CRDT has internal state. As an application designer, you get to choose the form of that internal state. It is based on the problem you are trying to solve. But that state has to satisfy a few conditions.

- It must support a "happened before" (causality) relationship that defines a *partial order*.

- All updates must *increase* the state in that partial order (the previous version "happened before").

- It must support a *merge* operation that takes two states and produces a new one that is greater than both of them (both previous versions "happened before" the merged version).

To be useful, the "happened before" relationship should help us detect concurrent updates. We want to avoid creating a total order and instead capture the partial order inherent in causality.
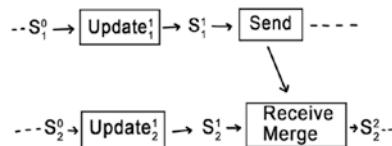
Unlike our relay-based distributed system, updates do not have to be idempotent or commutative. That's because updates will be executed only on a single replica. Within a single process, we can easily control how many times and in what order updates are applied. A CRDT does not rely upon message relay like the system we just analyzed.

Shapiro, Preguiça, and colleagues proved that these three conditions are sufficient to guarantee SEC. All replicas will converge to the same state after all updates are delivered. So, what does it mean for an update to be delivered to a replica?

Causal History

When we examine the state of a replica within a single process, we will find only two operations that cause it to change: updates and merges. An update occurs when that node executes some command from outside the network. Perhaps the node is running a client application and responding to user input. A merge occurs when another node shares the state of its replica. This happens on the *receive* operation of a network communication.

Recalling Leslie Lamport's definition of causality, we can say that the state of a replica after an update is *caused by* the update; the update is in its *causal history*. Lamport also showed us that the *send* operation of a network communication causes the *receive*. And so, the updates that occurred on the origin node before the send, illustrated in Figure 4-17, are in the causal history of the merged state.

Figure 4-17. *Both updates are in the causal history of s_2^2*

Following this logic, the causal history of a replica includes

1. All updates that have occurred previously in that node

2. All updates in the causal histories of the states that were merged from other nodes.

Remember, the updates themselves are not shared between nodes. Only the resulting states. The necessary conditions imposed by partially ordering the states ensures that they have just enough information about the updates that were performed on them.

And so now we can finally answer the question. What does it mean for an update to be delivered to a replica? It means that that update is in the causal history of its current state.

Vector Clocks

That's a lot to process. Let's make it a bit more concrete. How could we have implemented the contact management system as a CRDT? For simplicity, we will look only at the update contact case, which changed the properties of a contact. We'll represent each contact as a CRDT, where each workstation, PDA, and server node have its own replica.

Let's start by defining the state. Each contact is going to have a set of properties, like name, phone number, and email address. We will store those in the state.

By our first condition, the state needs to support a "happened before" operator to give us causality. Clearly just looking at the properties of a contact, we cannot tell which of two versions came first. For that, we will need to keep some sort of version number. A simple monotonically increasing version number would satisfy our first and second conditions. We will be able to see which version came first, and we will increase the version number with each update.

Unfortunately, a simple version number does not help us to identify concurrent updates. It does not capture the true partial order of causality. So instead, we will keep a separate version number for each node. This is a data structure known as a *vector clock.*[9] Figure 4-18 shows an example.


```
{
  Node 1:3
  Node 2:1
  Node 3:7
}
Name:"Robert"
Phone:"555-1212"
Email:"rob@email.com"
```

Figure 4-18. *A vector clock as part of the contact CRDT*

To compare two vector clocks, we compare each node's version number. If all of the version numbers in the first vector clock are less than or equal to the second, then the first one "happened before" the second. This is a partial order, because it's possible for

---

[9]Friedmann Mattern. Virtual time and global states of distributed systems. In Int. W. on Parallel and Distributed Algorithms, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

one node's version to be lower in one, while another node's version is lower in the other. When this happens, the two clocks are not causally related.
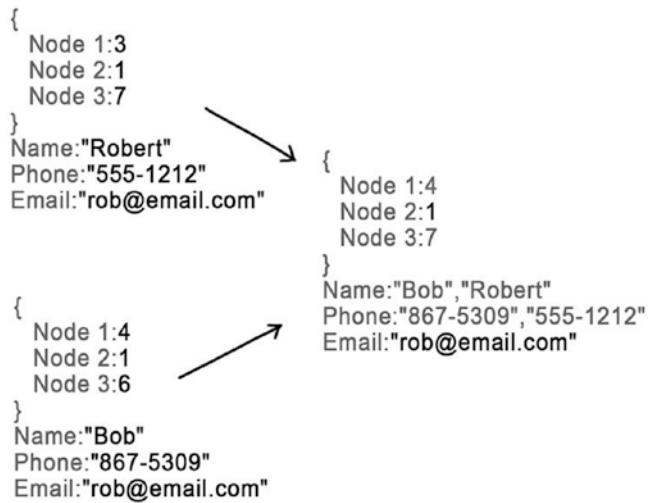
When a node updates a contact, it increments its own version number in the vector clock, as demonstrated in Figure 4-19. This ensures that the new version has a greater version, as required by the second condition. State moves forward in causal time with every update.

```
{                           {
  Node 1:3                    Node 1:3
  Node 2:1                    Node 2:2
  Node 3:7                    Node 3:7
}                           }
Name:"Robert"               Name:"Robert"
Phone:"555-1212"            Phone:"555-1212"
Email:"rob@email.com"       Email:"robert@email.com"
```

Figure 4-19. *Node 2 updates the email of a contact and bumps its own version number*

When a node merges state from a remote node, as in Figure 4-20, it takes the maximum of each of the version numbers. This ensures that the new vector clock is greater than each of the two original vector clocks. Strictly speaking, if one vector already "happened before" the other, then the merge will just give us the greater of the two. In this case, it is not greater than, but equal to the later version. What this operation is really doing is computing the least upper bound of the two vectors. That is more precisely what is required by the SEC conditions.

```
{
  Node 1:3
  Node 2:1
  Node 3:7
}
Name:"Robert"
Phone:"555-1212"
Email:"rob@email.com"
```

```
{
  Node 1:4
  Node 2:1
  Node 3:7
}
Name:"Bob","Robert"
Phone:"867-5309","555-1212"
Email:"rob@email.com"
```

```
{
  Node 1:4
  Node 2:1
  Node 3:6
}
Name:"Bob"
Phone:"867-5309"
Email:"rob@email.com"
```

Figure 4-20. *Merging two contacts takes the maxima of all version numbers*

We can see that vector clocks satisfy the necessary conditions for SEC. Since we put a vector clock into the state of our CRDT, we have a way to compare two versions to see which one happened before. Incrementing one version number when we update produces a vector clock that is greater than the previous one, so updates increase the state. Merging two vector clocks produces a set of numbers that are all greater than or equal to the previous two vector clocks, and so merge produces a state greater than the source states.

Most importantly, vector clocks capture the causal history of the replica. During a merge, we can detect concurrent updates. If all version numbers in the vector clock of one of the two states are greater than or equal to the other one, then that state represents the more recent version. The values of the contact's properties will simply be copied from the greater of the two. But if neither vector clock "happened before" the other, then a concurrent update has occurred. That tells us that we need to merge the contact's properties.

A vector clock is a tool for building state-based CRDTs. It gives us a way to define a partial order between states that supports update and merge operations. When used properly, replicas that include a vector clock will converge to the same value, once all updates appear in their causal history.

If my friend and I had built the contact management systems using vector clocks, then introducing a new device to the system would be a simple download. It would get only the current state of each contact and the vector clock representing the causal

history of that state. When a user makes a change on this new device, it would simply add itself to the vector clock at version 1 and share its new value.

A History of Facts

Vector clocks are not the only conflict-free replicated data type. Any data structure that is partially ordered, increases on update, and computes the least upper bound on merge can be used as a CRDT. Researchers have already defined many such data structures for use in different situations.[10] With a little work, you can design the CRDT that is exactly right for you.

There is one data structure that is applicable in a surprisingly large number of circumstances. It has a well-defined partial order. It increases on updates. And it comes equipped with a valid merge operation.

I'm talking, of course, about the humble set.

Sets

A set is a collection of items that has a couple of interesting properties:

1. It contains no duplicates.

2. It is unordered.

The first property means that an element is either in the set or it is not. The set does not remember how many times we tried to add an element. The second property tells us that no element comes before or after any other element. The set doesn't remember the order in which we added the elements.

It's interesting to observe that set insertion satisfies the two conditions necessary for SEC in a relay-based system. Because of the first property, set insertion is idempotent. If we try to insert an element that is already in the set, it remains unchanged. And because of the second property, set insertion is commutative. Inserting elements in the opposite order yields the same set. These two properties mean that set insertion behaves well in the face of duplicated or out-of-order messages.

---

[10]Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. [Research Report] RR-7506, Inria – Centre Paris-Rocquencourt; INRIA. 2011, pp.50. ffinria-00555588f.

Idempotence and commutativity are not required for using a set as an operation-based CRDT. The properties that are required for CRDTs are partial order, increasing updates, and a merge that computes the least upper bound. As long as we do not allow elements to be removed from a set, we can easily achieve all three.

Partial Order

Sets are partially ordered under the subset relationship. One set is a subset of another if it only contains elements that can be found in the other one. If we use subset as "happened before," then we have defined a partial order.

Take, for example, the set {🚗, 🚴, 🚂}. It is a subset of {🚗, 🚴, 🚆, 🚂}. Every element in the first is also in the second.

Now consider a third set {🚴, 🚆, 🚂}. It is also a subset of the second. But neither the first nor the third is a subset of the other. The first set contains an element not found in the third (🚗), and the third contains an element not found in the first (🚆). Therefore, they are not related under the subset relationship.

The fact that some sets can be compared while others cannot is what makes this a partial order. We can use that partial order to represent causality. A subset "happened before" a superset.

Update

The only update operation that we will allow on a set is an insert. If you try to insert an element that the set already contains, then the set is unchanged. But if the element was *not* already in the set, then the new set has everything that was in the old set plus one additional item. So set insertion, when it modifies the set, increases its value within the partial order.

If you think about the set as containing all of the knowledge of a replica, you can see how set insertion increases that knowledge. The replica still knows everything that it knew before. But after the update, it now knows a little bit more.

This also illustrates how a set can represent the causal history of a replica. Recall that the causal history of a replica includes all of the updates that have occurred in its past. By enumerating the elements of a set, you can clearly see all of the insertions that have occurred over time.

Merge

A valid merge operation in a CRDT will compute the least upper bound of the two values. The least upper bound of two sets under the subset partial order will be the set that contains every element from both sides. It will be a superset of each one. To compute the least upper bound, we simply have to take the set union.

Consider again the two sets {🚗, 🚴, 🚂} and {🚴, 🚜, 🚂}. Neither is a subset of the other. But we can compute the smallest set that is a superset of both of them. That will be the set union: {🚗, 🚴, 🚜, 🚂}.

This follows our intuition about a merge, as well as the conditions required for SEC. If one node merges all of a remote node's knowledge into its own, then it ends up with the union of the two. Knowledge grows as a result of that merge.

Thinking of this as the combination of two causal histories also makes intuitive sense. The causal history after a merge includes all of the updates that occurred both locally and remotely.

Historical Records

Let's formalize this intuition about a set representing the causal history of a replica. Instead of looking at sets of transportation emoji, let the elements in the set be actual historical records.

When a user performs an action at a node, we capture that action as a historical record. We make note of what they were trying to do, what they were trying to do it to, and what options or parameters they chose while doing it. The record is simply a structure that collects all of this information. It captures all of the pertinent data that was part of the user's decision.

When we put these historical records into a set to form a causal history, we will notice four things:

1. We need to distinguish between similar records.

2. We cannot remove a record once it is inserted into the set.

3. We cannot change a record that is already part of the set.

4. Some of the records are related to one another.

These four observations give rise to the rules of historical models.

Distinguishing Between Records

Revisiting the contact management system, we can identify the first action that a user might perform at a node. They will want to create a new contact. When a new contact is created, it doesn't have any properties. Those can be changed later.

The causal history that a user creates will look something like this:

> { ContactCreation }

When they try to create a second contact, however, they run into a problem. The record *ContactCreation* already exists in the set. It cannot be inserted again.

To insert more records of contact creation into the causal history, we need to distinguish among them. This is where a location-dependent system would use an auto-incrementing identifier. It would produce a causal history that looks like this:

> { ContactCreation(1), ContactCreation(2) }

The problem with this strategy becomes apparent when we merge one node's causal history with that of another. Merge is accomplished with a set union. If they were both generating identifiers using an auto-incrementing counter, then they would produce the same identifier for different contacts. The set union would merge different contacts into one.

So instead, the node will choose a location-independent identifier. A natural key would be best, but in this case we don't have an immutable natural identifier in the problem domain. We aren't asking contacts for their date of birth, government ID number, and DNA sample. We will just have to be satisfied with a GUID.

> { ContactCreation(74aac247-a86a-4af8-9db4-cf1387f8a1fb),
>
> ContactCreation(5853e3fe-059a-4180-af0a-f969260be882) }

And so we have discovered that a historical record is uniquely identified only by its content. It has no other identifier. This is to ensure that causal histories merge in a location-independent manner.

Removing a Record

The next action that a user of the contact management system wants to perform is deleting a contact. The most natural way to represent a deletion is to remove the creation of the contact from causal history. Removing the second contact (5853…) from the set would bring us down to this:

$$\{ \text{ContactCreation(74aac247-a86a-4af8-9db4-cf1387f8a1fb)} \}$$

But this strategy won't work. It violates the second condition of a state-based CRDT. Update operations may only increase the state of a replica within the partial order. Removing an element from a set creates a subset, not a superset. This takes the state backward in causal time.

It becomes apparent that we've made a mistake when we share state with other nodes. Suppose the user creates a contact, and then their device shares its state with another node. That contact creation is now part of the other node's replica.

Now suppose that the user removes that contact, and the node incorrectly represents that by removing it from the set. When the remote node at some point in the future shares its state with the user's node, the replica sets will be merged. The contact that they had deleted will suddenly reappear. Just search for "deleted contact reappears" in your favorite search engine to see just how common this bug is.

Instead of removing a historical record from the causal history, we have to instead create a new historical record. That new record represents the deletion of a contact.

$$\{ \text{ContactCreation(74aac247-a86a-4af8-9db4-cf1387f8a1fb)},$$

$$\text{ContactCreation(5853e3fe-059a-4180-af0a-f969260be882)},$$

$$\text{ContactDeletion(5853e3fe-059a-4180-af0a-f969260be882)} \}$$

We have restored the condition that updates increase state within the partial order. This new set is a superset of the prior one. And merging state with other nodes will never cause the contact to reappear.

We have just discovered that a historical record cannot be deleted. A record can *represent* the deletion of an entity. But it cannot be removed from causal history.

Changing a Record

The next action that a user might want to perform at a node will be setting the properties of a contact. When they do, we'll record a historical record of their actions. It includes which contact they are modifying and the values they set for those properties. The resulting causal history looks something like this:

> { ContactCreation(74aa…),
>
> ContactModification(74aa…, "Bob", "555-1212") }

On the second edit, the naive solution will be to modify the record within the set:

> { ContactCreation(74aa…),
>
> ContactModification(74aa…, "Robert", "555-1212") }

We can already see why that doesn't work. The new set is not a superset of the original one. This violates the second condition: updates must increase state in the partial order. We have, in fact, created a new set that is not causally related to the old one.

To fix this problem, we can partially order the modification records. One way to do this is to add a vector clock:
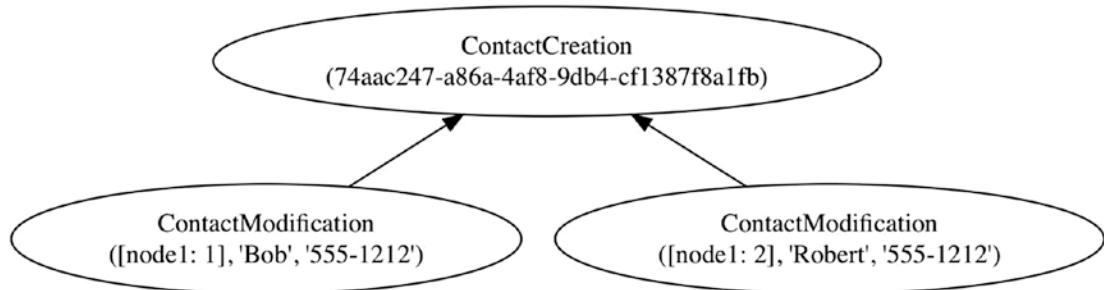
> { ContactCreation(74aa…),
>
> ContactModification(74aa…, [node1: 1], "Bob", "555-1212"),
>
> ContactModification(74aa…, [node1: 2], "Robert", "555-1212") }

We have discovered that once a record is part of history, it cannot be modified. We can add a new record that *represents* a modification to an entity, but the old records must remain.

Since we already have a complete set of the historical records, the vector clock is a bit redundant. Remember, vector clocks help us to turn a simple data structure into a CRDT. It captures the partial order of causality. But now that we are working with a *set* of simple data structures, we can rely upon the set to capture causality. We have the opportunity to optimize a bit.

Records Are Causally Related
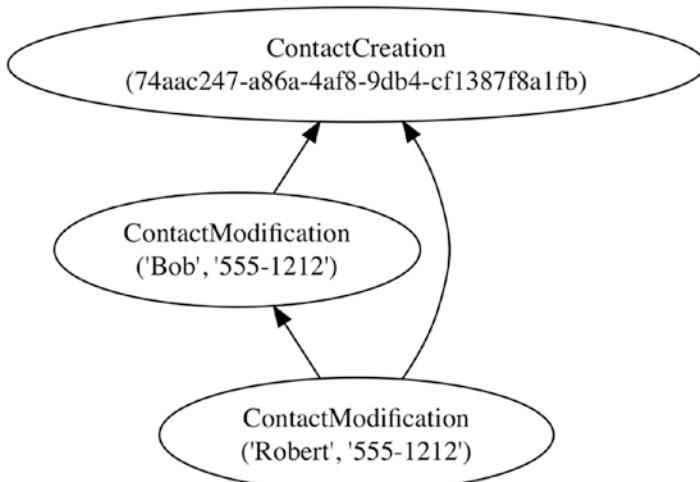
Looking carefully at the set of historical records reveals several relationships. The most obvious one is that the *ContactModification* records contain the GUID of the *ContactCreation* record. As Figure 4-21 shows, we can represent this relationship directly by drawing an arrow from the modification to the creation.

Figure 4-21. *Contact creation precedes contact modification*

This graph is still a set. Every vertex of the graph is a historical record in the set. All we have done is replaced the implied relationships of common GUIDs with explicit arrows.

The second observation we can make is that the vector clocks are actually references to one another. The clock [node1: 2] represents an update that occurred on node 1 bringing its version from 1 to 2. It refers to the previous clock [node1: 1] by inference. As Figure 4-22 shows, we can replace the vector clocks with arrows.

Figure 4-22. *Vector clocks are replaced with explicit arrows*

Changing the vector clocks to arrows preserves the partial order between modifications. It is still easy for us to compare two *ContactModification* records to see which one came before the other. If we can trace a path from one to the other along the arrows *in the correct direction*, then the record at the head of the last arrow "happened before" the record at the tail of the first one.

The graph captures the partial order of causality.

This is true in general, not just for the modifications. *ContactCreation* happened before *ContactModification*. The user must have created a contact before setting its properties. If the contact wasn't created on the local node, then it must have been created remotely and merged into the set. By Lamport's causality, even that remote creation happened before the local modification.
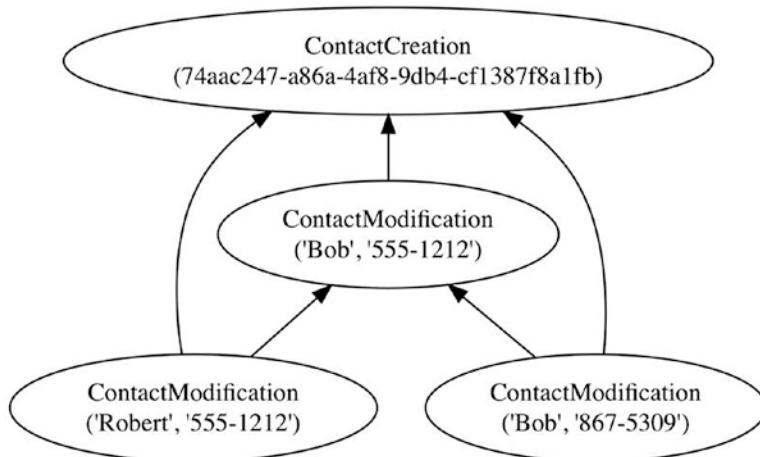
Benefits of Explicit Causality

We have captured the causal relationships between historical records as arrows in a directed graph. Doing so is more than just an optimization. It also enforces preconditions of the user's actions. A user cannot modify the properties of a contact that hasn't been created. Nor can they delete one that doesn't exist. When the causal relationships between records were only implied by a shared GUID, the data structure did nothing to help us ensure that these preconditions were met. But now that it explicitly captures the arrows, these preconditions are enforced by the existence of the record at the head.

Another benefit is that we have just traded away less important information for more important information. The vector clock included the names of the nodes at which modifications were made. It needed this information only so that a node knew which version number to increment on update. After that, the names of the nodes are unimportant. The arrows discard the names of the nodes in favor of explicit references to prior versions. It doesn't matter whether that prior version was produced on the same node, or arrived as the result of a merge.

In exchange for discarding unimportant information, the explicit arrows provide us with much more important information. They tell us how an entity has changed over time. This is useful in computing a better merge.

Suppose that after the contact is created and initialized, it is shared with a remote node. Against that replica, another user makes a different modification. After the local user receives the merge, they see the graph in Figure 4-23.

Figure 4-23. *A graph after merging concurrent modifications*

The partial order among modifications shows us that the two leaves of the graph are not causally related. Neither one happened before the other. We therefore need to merge the two sets of properties to display to the user.

The concatenation-based merge that we did before would produce a contact with two names and two phone numbers. If all we had were two data structures and two vector clocks, this is the best we could hope to achieve. But having the graph gives us a third data point. We can see the *nearest common ancestor* of the two leaves.

Comparing the left branch with the nearest common ancestor, we can see that the local user changed only the name. And comparing the right branch reveals that the remote user changed only the phone number. This allows us to perform the much more reasonable merge of displaying the most recent name and the most recent phone number.

> name: "Robert"
>
> phone: "867-5309"

This three-way merge happens only on display. The set (or graph) of historical records is not modified in any way. Furthermore, all nodes perform this three-way merge in exactly the same way. They all have the same graph, so they will all compute the same result. So even when the history captures causally unrelated records, it does not result in a conflict. Every node converges to the same value.

With this final observation, we have discovered that historical records are related to one another explicitly. This relationship captures the partial order of causality. The record at the head of the arrow happened before the one at the tail.

Historical Facts

Because these records are no longer simple flat data structures, I hesitate to call them records anymore. Nor do I like referring to them as historical *events*, as that evokes Event Sourcing. Event Sourcing captures a totally ordered sequence of historical events, but does not capture explicitly the relationships *between* historical events. And so I refer to these elements of the causal set as *historical facts*.

The arrows point toward the *predecessor* of a historical fact. The earlier fact preceded the latter one. The latter, I call the *successor*. Arrows are only inserted into the graph with the successor, never afterward. This enforces preconditions, preserves the partial order of the causal relationship, and has the extra benefit of preventing cycles. The contents of a fact combined with its set of predecessor arrows are all that distinguish it from other facts. Because facts have no extrinsic identity, nodes can refer to facts in a location-independent manner. The entire causal set is what I call the *historical model*.

A **historical model** is a state-based CRDT that captures the causality among historical facts as a directed acyclic graph. The arrows of the graph impose a partial order that shows which facts happened before which other facts. Facts in the graph can be referenced, queried, and used with no dependence upon the location of the node.

The graph supports two operations: insert and merge. Inserting a new historical fact moves the graph forward along the causal timeline, because the resulting graph is a superset of the original. Merging two historical models computes their least upper bound and therefore helps remote replicas achieve strong eventual consistency.

Conclusion

We have identified the properties that a distributed system must have in order to be location independent. They must exhibit location independence both in identity and in behavior.

A location-independent identity does not imply any affinity of an object upon its location of origin. Any node should be able to generate and compare a unique

immutable identity for a new object without communicating with other nodes. This reduces latency and increases autonomy of isolated nodes within a distributed system.

Location-independent behavior permits a node to query and transact with replicas of objects in isolation. These replicas achieve strong eventual consistency when all replicas converge to the same state once all updates are delivered. This can either be achieved by means of an idempotent and commutative relay system or by a more sophisticated conflict-free replicated data type.

From these constraints, we derived a set of rules that help us to define systems that operate in a location-independent fashion. I call this set of rules *Historical Modeling*. The reasoning laid out in this chapter demonstrates that a historical model satisfies the requirements for strong eventual consistency.

But before we code up a historical model, we should understand exactly what they tell us about the systems we intend to build. Let's explore these rules first as a means of analyzing a problem. Among other things, the rules quickly reveal when we make the assumption that data has a location. This analysis will uncover potential problems long before they have a chance to cause issues in production.

Analysis

Immutable architecture ensures that we build a system within constraints that computers can easily meet. When we start from the assumption of immutability, we have confidence that the resulting system will have desirable characteristics. What remains now is to understand how to analyze a problem domain within those constraints. We must discover the elements of a problem and express them as immutable units.

The primary goal of analysis is to communicate a shared understanding between the development team and the product owner. The analyst extracts the essence of the problem domain and documents that understanding in a way that the product owner can verify. For this to be effective, the analysis needs to be understandable without training or jargon.

The secondary goal is to capture the expected behavior so that it can be correctly implemented and tested. For this, the analysis needs to be documented and presented to developers and testers. Artifacts need to be precise enough that an incorrect implementation will fail these tests. They should guide and constrain development and answer questions that arise during implementation.

The means of communication therefore needs to satisfy several audiences simultaneously. It needs to be easily understood by experts in any business domain, not just technical domains. It also needs to be precise enough to generate meaningful tests. And it needs to represent ideas that can actually be implemented. Our challenge now is to define a language that solves all of these problems at once.

Requirements and analysis documents are typically written in prose. Unfortunately, natural human languages are poor choices when precision and unambiguous communication are the goals. Prose contains assumptions that the author fails to articulate and that readers fail to recognize. It contains jargon that conveys meaning for some, but confusion for others. And human languages do not compile; they are not constrained to describe only solutions that can be implemented, nor can they be executed to test a given solution.

The artifacts of our design will therefore not be written in English or any other natural language. They will be written in the precise language of mathematics. Yet, to rid the artifacts of jargon that would require a mathematical education, they will be constrained to very simple concepts. The language of analysis maps precisely onto the language of Historical Modeling. It produces a set of statements that can be unambiguously verified. It also can only describe systems that can be implemented within the constraints of distributed systems.

Use Cases

In 1992, Ivar Jacobson published the first guide for "use case analysis." *Object-Oriented Software Engineering: A Use Case Driven Approach*[1] (OOSE) described a system for documenting software behavior. It breaks that behavior down into goal-directed units called "use cases." In each use case, an actor is using the system to accomplish a goal. While Jacobson originally developed use cases as part of a process called Objectory (Object Factory), he later joined forces with Grady Booch and James Rumbaugh to bring use case modeling and analysis to the Unified Modeling Language (UML).

While the majority of OOSE focuses on the detailed textual language for describing use cases, the book also introduces a high-level graphical language called use case diagrams. The purpose of a use case diagram is to illustrate the relationships among several use cases and the actors who initiate them. Use cases can be related to one another in several ways. A larger use case can *include* several smaller ones if the smaller use cases are performed in the process of achieving the larger result. A parent use case can *generalize* more specific use cases to collect their common attributes and steps. And a more detailed use case can *extend* a less detailed one by providing additional steps that occur under certain conditions.

In practice, use case diagrams have found much less utility than their more detailed textual counterparts. In many of the requirements diagrams that I have read, a use case diagram is included mostly as a table of contents, if it is drawn at all. Martin Fowler has even expressed that "these diagrams are of little value,"[2] focusing instead on the far more detailed textual form of the use case.

---

[1]Ivar Jacobson. Object Oriented Software Engineering: A Use Case Driven Approach Revised Printing edition. 1992. Addison-Wesley.

[2]https://martinfowler.com/bliki/UseCase.html

But given the problems with capturing behavior in prose, perhaps more focus *should* be given to the graphical language. If we solve some of the issues with the graphical language, then it could become a precise, unambiguous, and easily understood artifact. Inspired by Jacobson's original work, we can define a diagram that addresses the same need, but in a way that provides a more valuable foundation for system analysis. We can accomplish this by unifying the size of an activity and the relationships between activities.

From Use Case to Decision

The first problem that we can solve is that use cases can be expressed in several different sizes. To resolve this problem, we can change the nature of the diagram to reduce the size of each activity to an atomic unit. Traditionally, a use case is centered around a single goal. Achieving that goal requires multiple steps. Diagramming multistep use cases increases redundancy, inconsistency, and complexity.

Redundancy increases because use cases can share steps. A use case containing several steps might be interrupted, or branch into multiple scenarios. Examples of interruption include authorization failure, low inventory, or a need for approval. If the steps are interrupted, the use case lists the recovery steps as alternatives to the primary successful flow. A use case scenario is less severe than a full interruption. It is triggered by some condition that requires additional steps. If the steps are simple, they can be listed within the single use case. If they are complex, then perhaps they would be best broken out into an extension use case. Either way, a use case can easily balloon into an overlapping collection of flows.

Inconsistency arises within the size and scope of use cases. Multistep use cases are not of uniform size. Two people analyzing the same problem might come up with wildly varying granularities of use cases. Even a single individual breaking down a problem might choose small use cases for activities they know to great detail and large use cases for things they do not yet fully understand. The very structure of a use case diagram reveals that the size is not always uniform: some use cases include others. The smaller use case's steps appear as part of the larger one.

Complexity is a consequence of having many interrelated steps within a single use case. Development teams often find use cases too large and complex to estimate and complete in a single iteration. Use cases are often broken down into user stories for this reason. A user story is much closer to a single atomic decision that a user makes and the

system responds to. There is often a prerequisite state that the system must already be in for the user story to make sense. That state would be achieved by executing earlier user stories, which had already been completed during earlier iterations of development. This level of granularity allows a team to plan, track, and execute their work. A product backlog will contain user stories, but not use cases.

By insisting that an activity be an atomic unit, we approach a more uniform unit of size. Each bubble represents not an entire use case, but a single atomic decision. Figure 5-1 shows a single use case on the left. On the right, it lists the individual decisions that the user made in order to accomplish the goal stated in the use case.

Figure 5-1. *A use case is broken down into multiple atomic decisions*

Breaking down use cases into atomic decisions adds value to the resulting diagram. The edge cases and recovery scenarios can be more explicitly depicted and understood. Each resulting activity is similarly sized; they are all one atomic decision. The team can see more clearly how to plan their work to accomplish the goals depicted within the diagram.
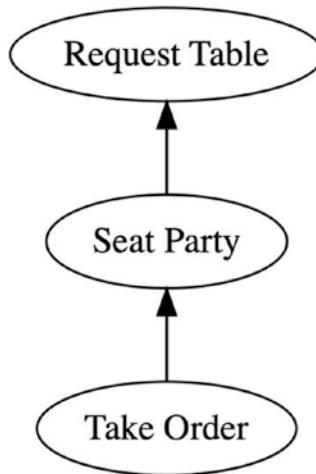
From Extension to Succession

The second problem with the use case diagram is that several different relationships can be drawn between use cases. Use cases can extend, include, or generalize one another. Understanding the distinction among these three relationships requires additional training. It introduces jargon *about* the use case diagramming process that is not an important part of the domain that it is modeling.

When use cases are broken down to the level of atomic decisions, however, the picture becomes a bit simpler. Decisions have only one relationship: they only precede one another. A decision that must be completed before another can begin is a *prerequisite*. The succession of decisions from prerequisite to subsequent tells a story of how an actor accomplishes a goal.
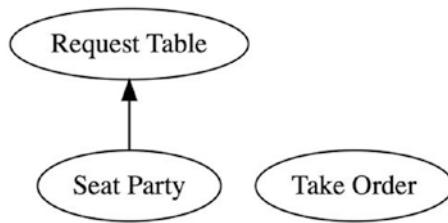
The prerequisite relationship is a strict one. When an analyst draws a line from one decision to its prerequisite, they claim that the prerequisite must be completed before the subsequent activity can begin. If that is not strictly the case, then the relationship should not exist.

Consider, for example, the typical set of activities that happen when a customer enters a restaurant. First, the customer requests a table, giving the size of their party. The host finds them a table, possibly asking them to wait, and then seats them. Once seated, a server takes their order. Given that typical sequence of events, you might draw each activity as the prerequisite of the next, as in Figure 5-2.

**Figure 5-2.** *A sequence of activities in a restaurant system*
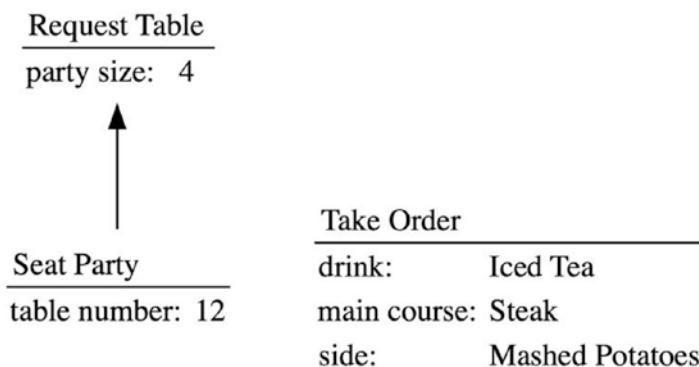
But is this really the correct depiction? Are there situations in which a server can take a party's order without them being seated? What about a to-go order, a catering order, or a customer at the bar? Seating a party is not truly a prerequisite to taking their order. It is only an activity that *usually* comes earlier and only in some scenarios. A more accurate analysis of this problem would be the one shown in Figure 5-3.

Figure 5-3. *Taking a party's order is not predicated upon seating them*

In addition to being strict, the prerequisite relationship carries information forward. A prerequisite provides information to subsequent activities. There is no need to duplicate the information from a prior activity into a later one. If a subsequent decision does not rely upon the information in the prerequisite decision, then perhaps that relationship should not exist.

In the earlier example, the request for a table includes the size of the party. That provides information that constrains the task of seating the party. The very relationship between Seat Party and Request Table indicates that we know the number of people to seat. An example instance diagram appears in Figure 5-4.

Figure 5-4. *Seat party has available to it the party size from request table. Take order does not need that information*

The Take Order activity did not require knowing where the customer was seated. As long as a server can communicate with a customer—even if it's over the phone or at the bar—they can complete the Take Order activity. This is another indicator that tells us that seating the party is not a prerequisite.

When use cases are large, it makes sense to have multiple complex types of relationships between them. A large use case can include smaller ones, and a common

use case can generalize specific ones. But when activities are expressed in their most atomic level, only one relationship is important. A diagram that depicts the succession of activities in the form of prerequisites illustrates the possible paths to achieving a goal. Each activity maps to a *fact* in the historical model, and the prerequisite relationship maps to *predecessors*. This kind of diagram can be much more than an introduction to a requirements document. It can be the starting point for some truly powerful analysis.

Data
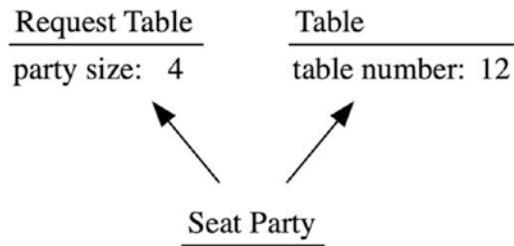
We've broken down a system's use cases into atomic decisions. Each decision cannot be broken down any further. It is made by one person, has no internal conditions, and cannot be interrupted. The relationships between those decisions are only *prerequisite* relationships. Now we can start looking more closely at the data carried with each decision.

We've seen how the data from prerequisite decisions is available to subsequent decisions. That data is available because of the prerequisite relationship. This allows us to analyze how information accumulates as we chain more decisions together into a deeper path of succession. We can further analyze that information content to discover identifiers, cardinalities, and mutation.
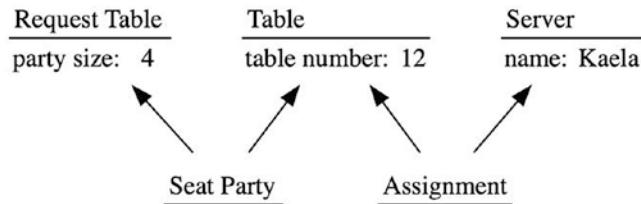
Identifiers

Hidden within many data structures are values that identify people, objects, or other entities. When you find an identifier hidden within some data, consider extracting it to a separate fact. Replace the field with a predecessor relationship pointing toward that fact.

Continuing our analysis of the restaurant system, we can see an identifier in the form of the table number. We decide to elevate this concept to its own fact, moving the table number into it to serve as its identity. When we replace the identifier in the Seat Party activity with a predecessor relationship, we end up with the diagram in Figure 5-5.

Figure 5-5. *Table number is elevated to its own fact*

Elevating an identifier to a fact creates a new point of interest in the application. This accomplishes two things. It first allows us to observe all of the successors of this new fact. We can identify, for example, which parties were seated at a table over time. But perhaps more importantly, it gives us an anchor to say more things about the identified entity. In this case, we now have a place to say which server is assigned to a table. This helps us to understand who is responsible for attending to the party, as well as to balance future seating so that all servers have a roughly equal workload. The result appears in Figure 5-6.

Figure 5-6. *A server is assigned to a table*

As you elevate identifiers to facts, you will find that the model contains a mixture of activities and entities. The activities came from use cases, broken down into their atomic decisions. The entities are identified people, objects, places, or concepts that those decisions are about. Nevertheless, the diagram contains only one kind of relationship: predecessor. This is either a prerequisite decision—one that must have come before—or an entity involved in that decision.

Cardinality

Continuing our analysis of a set of decisions, we can identify parts that have zero, one, or many replicas. These indicate points of cardinality that need to be addressed in the model. A part that allows only zero or one replica becomes an optional (?) predecessor.

152

A part that allows any number could either become a multiple (*) predecessor, or a successor. The difference between the two has to do with whether new parts can be added after the fact.

The `Take Order` fact in the previous example calls out for multiplicity. We originally modeled it as a single unit having three fields, as shown in Figure 5-7.
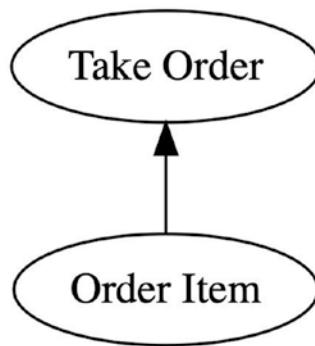

Figure 5-7. *An example take order fact having three fields*

In this example, it made sense to label each of those fields as `drink`, `main course`, and `side`. This particular order was made up of those three parts. But in general, we don't need to enforce that every order has exactly those three things. Some orders will have appetizers. Some will have desserts. Some might have two sides. While a typical order might follow a pattern or template, there is no value to restricting the items in the order to specific categories.
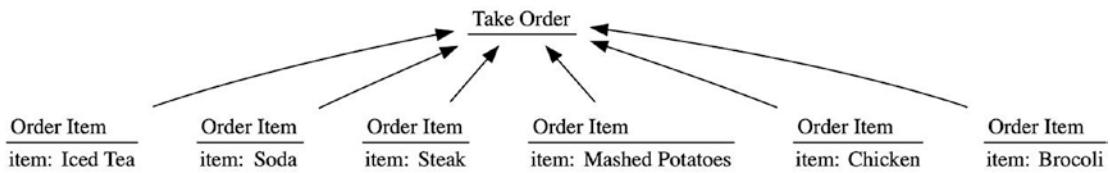
Furthermore, the order is presumably for the entire party. We are likely to have multiple drinks, multiple main courses, and so on. Different people might even order appetizers as main courses. At this point in the analysis, it is important to ask whether these distinctions are relevant for the solution you are modeling. Is it important to know which person ordered the steak, or will the server simply ask when they reach the table? Is it important to model when to deliver the salad, or will the server keep track of each course on their own?

The decisions you make while analyzing the cardinality of the model determine the relationships you will emphasize. These will reflect the values of your product owner. One valid restaurant model would allow the flexibility of adding items to an order at any time. A different, but equally valid model would lock an order down so that it can be controlled at each part of the preparation, delivery, and payment. If the product owner values flexibility over control, then you would come up with a model such as the one in Figure 5-8 that allows successors to be added at any time.
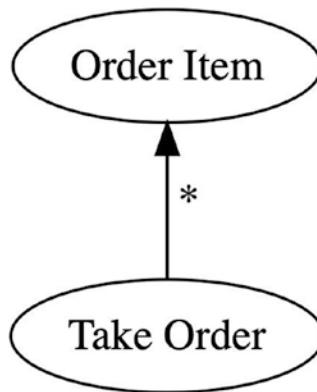
Figure 5-8. *Take order allows multiple order items to be added as successors*

A fact can always have multiple successors. We do not indicate that cardinality on the diagram, as it is implied. An example of facts in that structure appears in Figure 5-9.
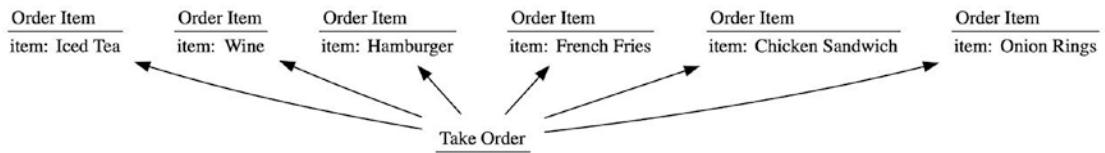

Figure 5-9. *Multiple items represented as successors to the order*

The decision to model items as successors emphasizes the fact that new items can be added to an order at any time. If the product owner makes a different set of decisions, then you might choose a different model. For example, in a fast food restaurant, the order is taken in its entirety, prepared, and then delivered. Modifications cannot be made along the way. That may cause you to instead make the `Order Items` predecessors of `Take Order` as in Figure 5-10.

Figure 5-10. *Multiple order items are predecessors of take order*

Since `Order Item` is a predecessor, we *do* indicate the cardinality. Order items cannot be added after the fact. The asterisk (*) indicates that multiple order items are present when the order is captured. An example set of facts matching this model appears in Figure 5-11.

Figure 5-11. *A strict organization of items as predecessors, which cannot be changed*

Simply saying that an order has many items is not quite enough to capture the nuances of the process. Knowing whether items can be added afterward is an important part of the model. These reflect the values of the product owner and translate into capabilities of the product. Capturing them in the model is an important first step toward analyzing the consequences of those decisions.

Mutation

The objects that we have analyzed and captured in the model have so far come from three different sources. Some were decisions made by an actor on the way to achieving a goal. Others were identifiable entities about which those decisions were made. And still
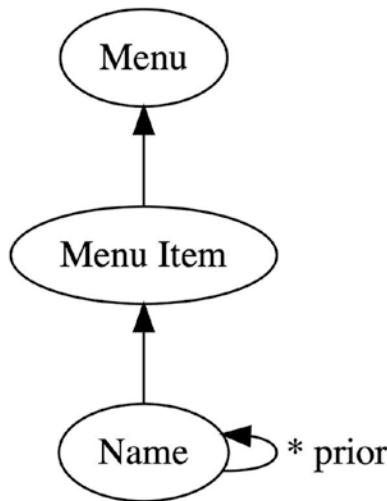
others were smaller parts of those decisions and entities that appeared from analysis of cardinality. In every case, these objects are immutable. The past decision, identity, or part will not change.

This does not always match our intuition. When we look at the state of a system, we imagine that it is changing over time. We see that state as mutable. But what we have modeled so far is a succession of decisions that have *caused* the apparent evolution of state change. The model represents those past decisions; it does *not* represent the state itself.

At some point, we will just need to capture a mutable property. We might, for example, just want to record a person's phone number. We are not interested in the various decisions that lead to them having a service contract with a phone company. Those decisions would be important in a different domain, but they do not contribute to the domain that we are trying to model. We just want to know how to call someone on the phone.
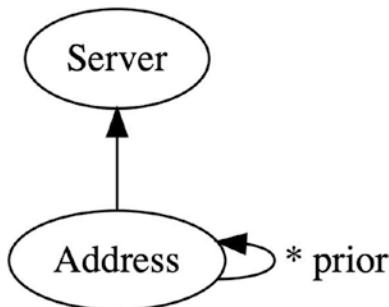
Modeling mutable state should be considered a last resort. If you can imagine a property being the consequence of a set of business decisions, then model those decisions. The total of a restaurant check, for example, is a consequence of the items ordered, the taxes and gratuities applied, and any discounts that the restaurant might be offering. Properties such as check total should not be represented as mutable state. Only use this pattern for values that are not an outcome of the business process being modeled.

For example, the name of an item in the menu might be modeled as a mutable property. We are not concerned with the business process that leads to that particular name. We simply want to document the fact that the name can be changed. We do so by writing the property as a fact separate from the entity that it describes. This property has a predecessor reference back to itself called `prior` and has multiplicity (*). The pattern is illustrated in Figure 5-12.

Figure 5-12. *The name of a menu item is a mutable property*

Some mutable properties tend to change as an entire unit. A mailing address, for example, has several components, like street number, city, and postal code. Favor capturing these composite values as a single mutable property, rather than as individual properties, as in Figure 5-13. This prevents the model from becoming unreasonably detailed, while documenting an important grouping concept within the data.

Figure 5-13. *An address is a single mutable property, even though it has many components*

The traditional analysis practice of defining a data dictionary makes no distinction between process properties and mutable properties. A data dictionary is more concerned with listing the various properties that can be attached to entities, no matter how they arise. But to truly analyze a system, we need to understand the processes that

brought about those values. When possible, those processes should be modeled as successive decisions within the problem domain. But sometimes, a mutable property is just a mutable property.

Views

After modeling the decisions, entities, and properties of a domain, we have a good understanding of its basic structure. The model so far captures how a process evolves over time. Each decision builds upon those of the past in a latticework of increasing knowledge.
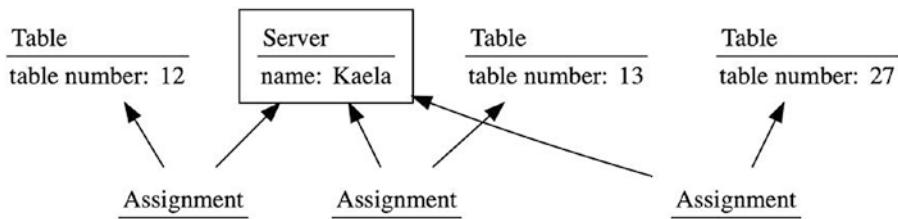
In order to make a decision, a user of the system needs to have access to that knowledge. The job of the system is twofold: provide that information and capture the resulting decision. That new decision becomes a new fact and contributes to the information that users will receive in the future. The views that a system presents to its users can therefore be expressed as a function of their past decisions. One of our jobs as an analyst is to describe that function.

Finding a Place to Start

To pull data out of a historical model, we need to identify a starting point. We cannot simply query the entire model as a whole. Fortunately, we usually have a few good candidates.
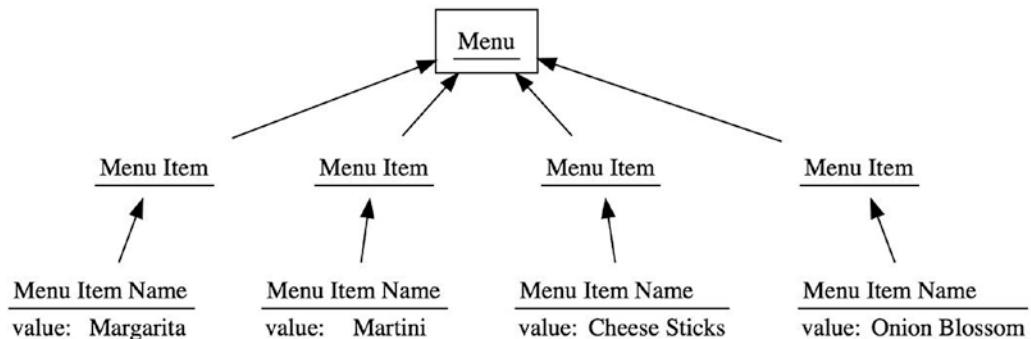
Most applications require a user to log in. As soon as they do, we have a starting point: the user themselves. Once in the application, they will navigate from page to page. As they do, they will change their starting place. The application will provide them information based on that point in the model. From there, they can continue to navigate or make a decision that will be captured in the model.

Let's continue building our restaurant model to see how the user can be the initial starting place. A server will log into the system at the start of their shift. From there, they will see all of the tables to which they are assigned. By logging in, the server has identified the starting point of the model, which is outlined in Figure 5-14.

Figure 5-14. *A server logs into the system to see their table assignments*

Even when an application does not require the user to log in, it usually has a clearly defined starting place. This is typically a top-level entity that defines the scope of the application. For example, if the restaurant guest has a device at their table for ordering drinks or appetizers, they will typically not need to log in first. The starting point of the information that they can search is the menu. From here, they can navigate through menu items, as shown in Figure 5-15.
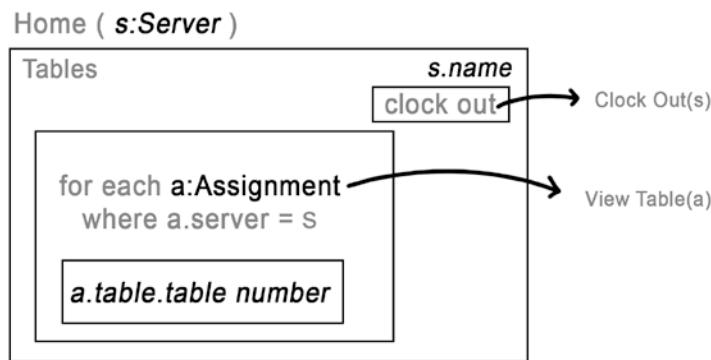
***Figure 5-15.*** *A guest starts with the menu to search through menu items*

Once we have identified a starting place, we can describe the information that will be presented to the user. We will use wireframes and queries to do so. These will in turn lead to refinements of the model in an iterative inward spiral of analysis.

Annotated Wireframes

Wireframes are a powerful tool for communicating the information that will be presented to the user. They are applicable to any system that displays information on a screen. They are effective for web, mobile, and desktop applications alike. And they become even more powerful when annotated to show precisely what information to display.

Every wireframe will have a starting point. The starting point is clearly indicated and puts the remainder of the annotations into context. The elements within the wireframe are then annotated to document the information that they present. These annotations take the form of a query from the starting point. For example, a wireframe of the server's home page is shown in Figure 5-16.

Figure 5-16. *A wireframe of the server's home page includes annotations on various elements*
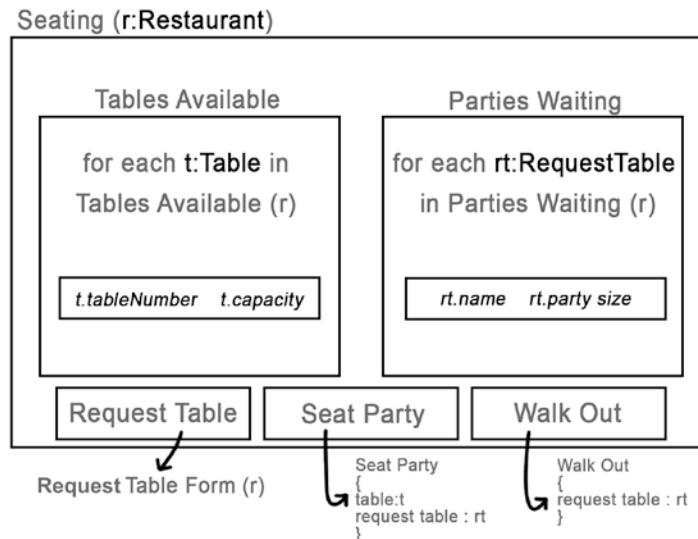
As the user navigates from one page to the next, they select a new starting point. Navigation appears as a line leaving the wireframe and carries with it the new starting point for that destination. Sometimes it is the same (as in the navigation to Clock Out), and sometimes it drills into a narrower context (as in View Table).

Annotations need not be represented directly on the wireframe the way that they are depicted in Figure 5-16. They could be shown as footnotes to keep the wireframe simpler, or to allow space for sample text. The important idea, though, is that the annotations be precise. The Factual query language conveys just the right precision to ensure that behavior is unambiguous and assumptions are made explicit.

Removal from Lists

Historical facts cannot be deleted. Views are projections of historical facts. Yet any view that simply grows unbounded will soon become useless. The specification for a view needs a way for items to be removed from lists. This operation is accomplished with a not exists clause.

Consider the host making the decision to seat a party. What information do they need to make that decision? They need to know the tables available, the parties waiting, and the capacities and sizes of each. With this information at hand, they can perform a few tasks. They can enter a request for a table, seat a party, or indicate that a waiting party has walked out. That information might be displayed in a view such as the one wireframed in Figure 5-17.

Figure 5-17. *A host sees the tables available and parties waiting and can take appropriate actions*

The "Request Table" button takes the host to a form where they can capture the details of an arriving party. It initiates a navigation. The "Seat Party" and "Walk Out" buttons, on the other hand, immediately create new facts. They record the decisions related to the selected table and table request. These decisions will influence the information displayed on the view.

But exactly *how* will these facts influence the view? What are the events that lead to a table becoming available? What makes it no longer available? These answers become clear when we define the queries referenced in the wireframe: tablesAvailable and partiesWaiting. Let's start with the first.
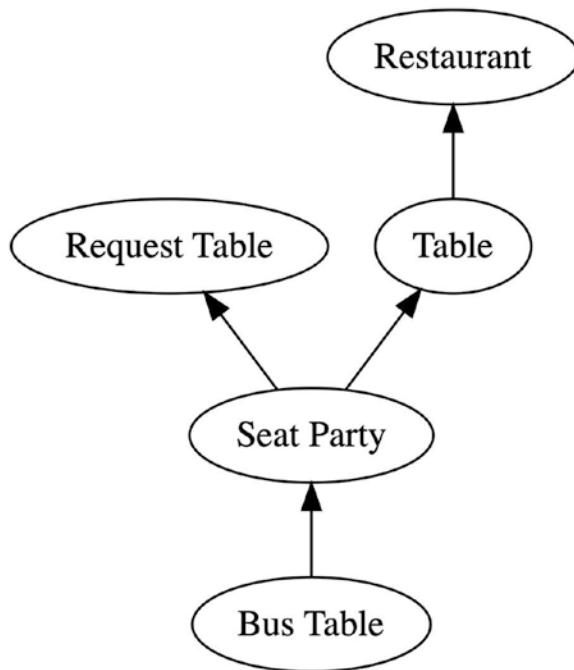
Consider the list of tables that are available for seating. We aren't simply adding to and removing from a list; there's a business process taking place. When the restaurant

opens, all tables are available. One is removed from the list when a party is seated. It is added again when the table is bussed. We can model this process with a single query:

```
query tablesAvailable(r: Restaurant) {
  match t: Table where t.restaurant = r
    such that not exists s: SeatParty where s.table = t
      such that not exists b: BusTable where b.seatParty = s
}
```

This query says a lot in a very small statement. It says that seating a party removes the table from the available list. It also says that bussing the table puts it back. It makes clear that we need to have a relationship between a table and a restaurant, so that we have a starting point for the query. And it tells us that we need a relationship between Bus Table and Seat Party. The revised model is shown in Figure 5-18.
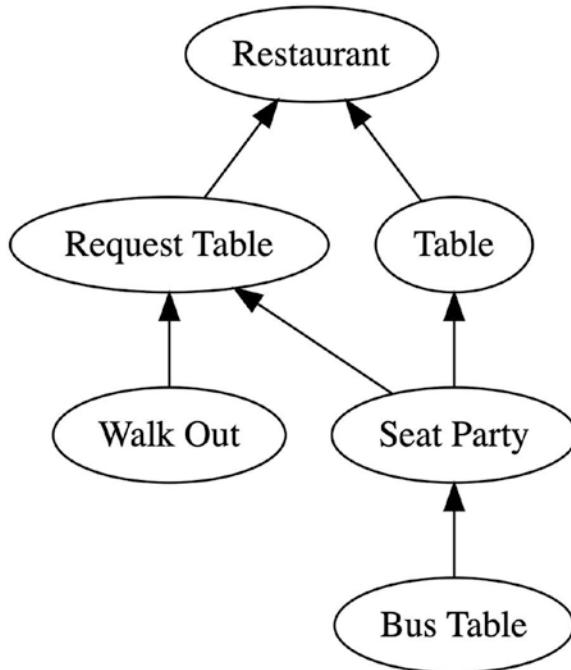

Figure 5-18. *New relationships suggested by the need to show tables available*

Following similar logic, we can analyze what adds and removes parties waiting for a table. A party is added to the list when they request a table. They are removed when they are seated. The party is also removed when they walk out. This leads us to define the view of parties waiting with the following query:

```
query partiesWaiting(r: Restaurant) {
  match rt: RequestTable where rt.restaurant = r
    such that not exists s: SeatParty where s.requestTable = rt
    and not exists w: WalkOut where w.requestTable = rt
}
```

These additional facts and relationships further refine our model. We can now see that Request Table must be within the scope of a Restaurant. We can also see how Walk Out is related to Request Table. The more complete picture appears in Figure 5-19.

Figure 5-19. *A request for a table happens within the scope of a restaurant and can be aborted with a walkout*

Taking these two queries together, we can find implied requirements. We can see, for example, that two parties should not be seated at the same table at the same time. This can be inferred from the fact that a table will be selected from the list `tablesAvailable`, which does not contain any tables with a current `Seat Party`. More subtilely, the queries reveal that seating a party removes both the party and the table. This is apparent because `Seat Party` appears in the `not exists` clause of each query.

Other systems of analysis would require us to consider these activities in terms of cause and effect. Requesting a table *causes* the party to be added to the list. Seating a party *removes* the party and the table. We end up defining a state machine describing how events change the state of aggregates. It is easy to forget to update the view state in response to an event, thus leading to incomplete specifications and bugs.

The precise nature of the Factual query language reveals assumptions about how state evolves with the introduction of new facts. Had the requirements for these views been expressed in prose, it would be easy to overlook these assumptions. A product owner familiar with the process of running a restaurant might not even see the assumptions that they are making. Of *course* two parties can't be seated at the same table. Why *wouldn't* you remove both the party and the table upon assignment?

A less precise form of specification would require the analyst to discover these unstated assumptions and raise the questions. If the analyst misses them, then the developers might run into edge cases. And if developers miss them, then testers might file a defect. If we demand precision, then our analysis will uncover assumptions, make behaviors explicit, and avoid waste.

Collaboration

As the model continues to evolve, we will want to take note of which actors are responsible for which decisions. Labeling the model with the actors gives us a clear picture of how the system will ultimately be used as a collaborative tool. People will work together *through* the system. It will be their means of communication.
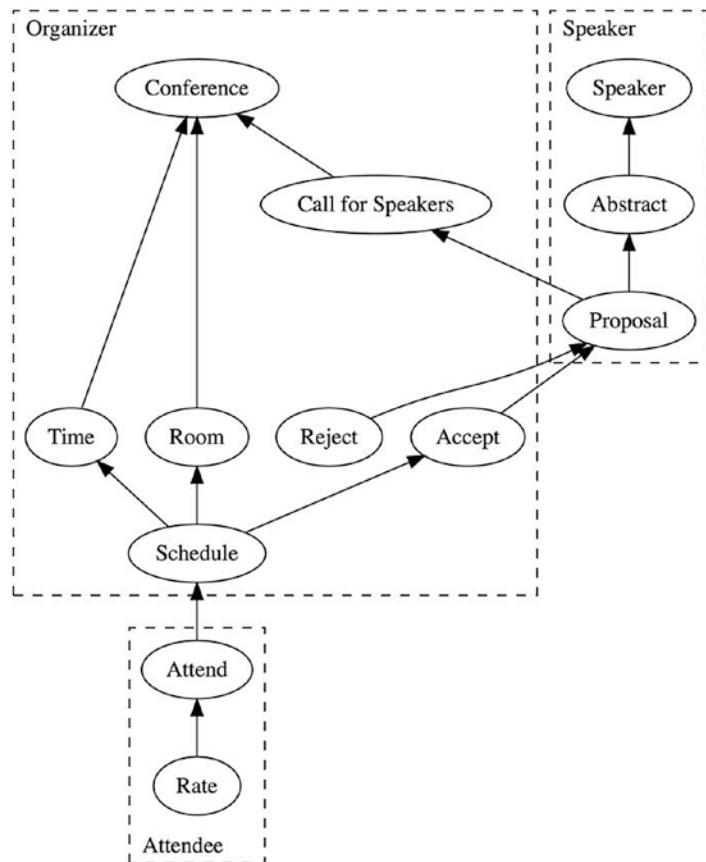
In a use case diagram, actors are drawn external to the system as stick figures. Arrows indicate which use cases particular actors are responsible for undertaking. While this is an accurate depiction—actors *are* outside of the system—it obfuscates points of collaboration within the model. It is much clearer to draw lines of responsibility within the model itself.

Regions

In my own work as an analyst, I have used several tools to indicate which actor is responsible for which decision. On a whiteboard or using sticky notes, I might assign each actor a color. In a notebook, I might jot a small annotation at the top of each fact. But the most versatile method, and the one that we will use here, is to divide the model into regions.

A region is an area of real estate within the model having clearly demarcated boundaries. All of the facts in a region are decisions for which a single actor is responsible. It is common practice in process modeling systems to use swimlanes to organize regions of responsibility. While regions in a historical model do not have to be swimlanes, the practice carries over quite naturally for those that are accustomed to it. When used, swimlanes are typically oriented vertically, because time progresses down the page.

Let's switch to a different model to illustrate this concept more clearly. Imagine a system that helps conference organizers to select speakers and then schedule their sessions for attendees. In this system, there are several actors, each making different decisions. The organizer is primarily responsible for putting on the conference and selecting the presentations. The speaker proposes presentations for the conference. And the attendee chooses which sessions to attend, rating them afterward. Figure 5-20 illustrates each of these three actors as a separate regions.

Figure 5-20. *Conference organizers, speakers, and attendees all have different responsibilities within a single model*

The organizer and speaker regions are roughly vertical swimlanes containing the decisions for which they are responsible. The process begins with the organizer creating the Conference and the speaker creating the Abstract. The organizer puts out a Call for Speakers, and speakers submit Proposals. If the proposal is accepted, then the organizer schedules the talk. The attendee only comes in at the tail end of the process to Attend and Rate sessions. Rather than show the attendee's actions in a vertically distinct swimlane, Figure 5-20 depicts them in a small region. This choice still conveys the important information: who is responsible for each decision. It also makes it clear when a line crosses a responsibility boundary.

Crossing Boundaries

Once the regions of responsibility are clearly arranged, an important feature of the model emerges. It takes the form of an edge crossing the boundary from one region to another. A decision that depends upon one outside of the decision maker's control is an important point of collaboration. It indicates that one actor made a decision and then published it so that another actor can respond. These points in the model are referred to as *pivots*.

When a pivot appears in the model, we know that the information provided by one actor needs to be made visible to others. We have two examples in Figure 5-20: sending out a call for speakers and publishing a schedule. The conference organizer is responsible for both of these decisions. In the first, the organizer advertises the speaker call on mailing lists, social media, websites, and through their professional network. In the second, the organizer deploys the schedule to their website and mobile app and hands out printed pages. The audience for each publication is indicated by the tail of the arrow: the call for speakers is targeted toward speakers, while the schedule is intended for the attendees.

In a computer system, pivots often appear as interfaces between interdependent subsystems. This is especially true if the actors on each side of the pivot are members of different organizations. In this scenario, it is not uncommon for a pivot to be expressed as an API. The receiving actor—the one at the tail of the arrow—makes an API available to business partners so that they can publish facts such as the one at the head of the arrow. This would be the case if a conference organizer used an API for a CFP (call-for-papers) website or social media platform to publish the call for speakers.

Even when actors are members of the same organization, pivots will often manifest as connections between subsystems. If actors are in different departments, for example, it is not uncommon for their systems to collaborate through message passing. In these scenarios, the fact at the head of the arrow will take the form of a message published to a topic or queue. This might also be accomplished with an internal API, a shared database connection, or a batch file. As an analyst, it is important to understand the connections that already exist and the opportunities available to interface with legacy systems.

Consumer-facing pivot points are also important features of a model. However, they often take very different forms within a deployed application. A public pivot point like the attendee's arrow into schedule can simply appear as information on a web page. The information may be searchable, or it might be listed in its entirety. Usually, a page refresh is required to update the view, but for some pivot points, the website or mobile app may choose to provide notifications. Annotate the model with these requirements so that the development team can choose the correct implementation of the pivot.

Conversations

In some models, several pivots appear between two actors. Usually, the pivots will swap directions and form a chain of facts traveling back and forth between the two. These represent conversations happening between two actors *through* the system. One actor publishes some information, another actor responds to it, and the original actor builds upon that response. These conversations usually provide the greatest value of the system and are therefore the most important features to analyze.

The model in Figure 5-20 contains one example of a conversation. After the conference organizer publishes a `Call for Speakers`, the speaker responds with a `Proposal`. The organizer then responds with `Accept` or `Reject`. This represents a back-and-forth collaboration between the organizer and each speaker. The organizer is presumably having these conversations with many speakers simultaneously. Each conversation carries with it the entire context: which call for speakers, which proposal, and whether it was accepted or rejected.
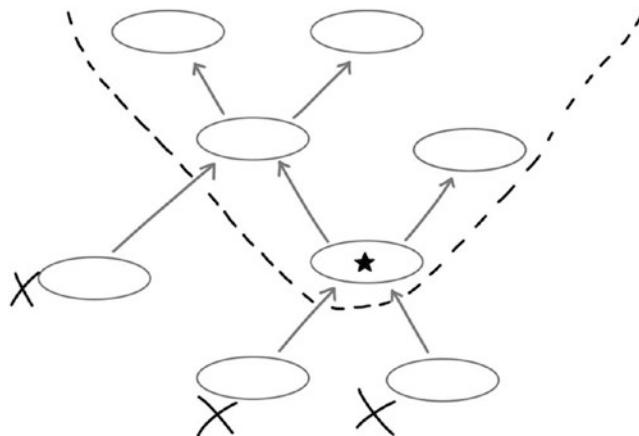
Publishing Facts

Conversations indicate that a multistep process must occur between two actors. Each actor is made aware of the decisions made on the opposite side of that conversation. This is even true of the last decision—the one for which there is no further response. In the preceding example, either `Accept` or `Reject` is the final decision in this conversation. Yet even though the speaker has no response in this model, they are made aware of the decision. Such will usually be the case with any conversation. Nevertheless, the model does not make that explicit. Use an annotated view or some other form of Factual query to make this assumption explicit.

Whereas the first pivot in the conversation is published to a wide audience, subsequent messages are directed to specific actors. The `Call for Speakers` is published on social media, but the subsequent acceptance or rejection takes place via direct message or email. This narrowing of scope can be inferred from the structure of the diagram. The `Speaker` is an indirect predecessor (an ancestor) of the `Proposal`. This relationship indicates that the speaker is particularly interested in responses to that proposal. That interest implies that the system needs some form of direct notification targeting a speaker. We will formalize this concept and explore the mechanisms to implement it in *Chapter 12*.

Integrating Subsystems
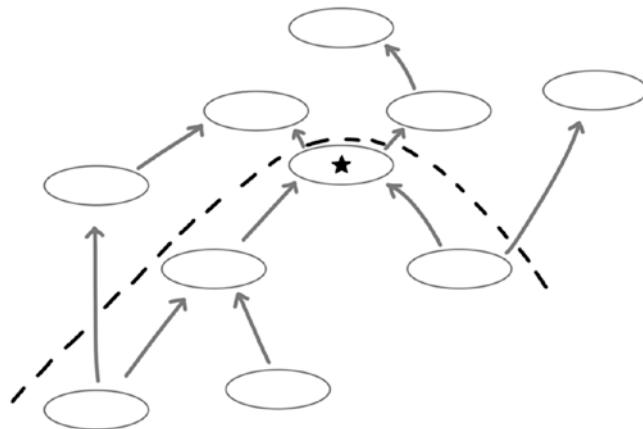
When the actors on opposite sides of a conversation are using different subsystems, each subsystem will usually need to carry a copy of the entire conversation. Whether they are in different organizations, or simply in different departments, their respective applications or microservices will store their own version of all of the data exchanged. The model shows what information this might include: follow the arrows up from the facts involved in the conversation. All of the facts in this upward cone—called the *transitive closure*—are likely to be duplicated to some degree between the systems. Figure 5-21 illustrates how this set can be found. If you discover that any information *outside* of the transitive closure is shared between the systems, then you have cause for concern. This information may change and will require other conversations if it is to be kept in sync.

Figure 5-21. *The transitive closure includes all predecessors and their predecessors. It does not include successors*

On each side of the conversation, consideration must be given to data ownership and security. There may be requirements governing the movement of data outside of a country's jurisdiction. Regulations and standards such as Europe's General Data Protection Regulation (GDPR), the United States' Health Insurance Portability and Accountability Act (HIPAA), the Payment Card Industry Data Security Standard (PCI-DSS), or California Consumer Privacy Act (CCPA) will determine how data is controlled, stored, secured, and accessed. These guidelines may even include policies under which data must be destroyed. While destroying data is typically *not* allowed within a historical model, exceptions must be made to accommodate these requirements. Fortunately, the model helps identify the set of all records that must be expunged: it is the cone extending

down from the entity to be erased or forgotten, as depicted in Figure 5-22. When this cone crosses from one region into another, then the system requires policies to ensure that partners are in compliance.

Figure 5-22. *The records to be deleted from a historical model include all successors and their successors*
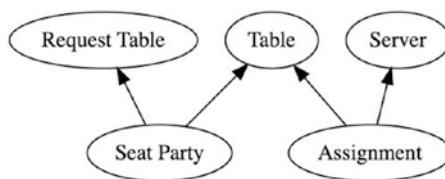
Pivot points are important features of a model to analyze. They are opportunities for the application to interact with external systems. While the job of the analyst is not to design that integration point, it is important to understand where they are and how they will manifest. Different forms of interfaces will constrain the system in different ways. Some will provide real-time notification, while others will require scheduled polling or the occasional refresh. As an analyst, call out the locations of pivots, and gather as much information about their constraints and requirements as you can assemble.

Valid Orderings

In building the model, we have assembled a graph of related decisions. Each arrow indicates that two decisions must be made in a certain order: the predecessor always occurs before the successor. But equally telling are the pairs of decisions between which there are no arrows. These are the places where order is fluid. Two facts may have a common ancestor, but as long as there is no path from one to the other, those two facts can occur in either order.

Look back at the model for seating and table assignment at a restaurant (Figure 5-23). The two facts `Seat Party` and `Assignment` have a common ancestor: `Table`. And yet, there is no way to walk in the direction of the arrows from one to the other. This indicates that

a party can be seated at a table before or after that table is assigned to a server. The order of these two decisions is not constrained.

Figure 5-23. *No arrow or chain of arrows leads from seat party to assignment, or vice versa*

Each of these two valid orderings—the party is seated before the table is assigned, or the table is assigned before the party is seated—implies a different requirement. First, because parties might already be seated when a server receives their table assignments, they should be told which tables are already occupied. And second, because parties might be seated afterward, a server should be notified when a party is seated at one of their tables. Either order is valid, but they evoke different behaviors from the system.

Eliminating Race Conditions

In this simple example, we have only two valid orderings. But in a more realistic system, the number of orderings can grow significantly. Finding and accounting for all possible permutations of events can be a daunting task. An analyst might find that the job gets cumulatively more difficult as each new requirement is added. What starts as a simple list of a few possibilities eventually explodes into a labyrinth of edge cases, each one being revisited and expanded with every new feature.
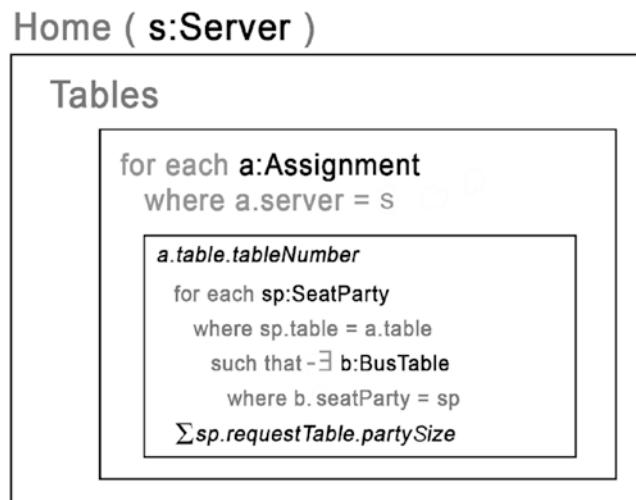
It is important for an analyst to understand how a system behaves under different valid orderings of events. Describing each of the permutations will help testers identify scenarios that must be examined. Explaining the possible behaviors will help developers code for edge cases. And most importantly, comparing the outcomes of different valid orderings will help uncover race conditions. A *race condition* arises when the final state of a system depends upon the order in which events occurred. To avoid race conditions, the analyst must demonstrate that all valid orderings converge to the same state.

Fortunately, the Factual queries with which we annotate our views provide the proof that we need. When we express the behavior of the system using a Factual query, rather than a series of cause and effect, we are guaranteed that the state converges to the same set of results no matter which valid ordering occurred.

Furthermore, those queries also provide a mechanism for enumerating all valid orderings. They reveal the events that could possibly affect the information presented to a user. Moreover, they show us how to respond to each of those events. They indicate how to use facts that have come before to update the view or notify the user. From a single specification, we can derive the correct behavior for any possible ordering of events.

Responding to Different Valid Orderings

Let's revisit the view that shows the server their table assignments. Earlier, we only identified each assigned table. But now, let's augment the view to also display the size of any party seated at that table. The modified annotation appears in Figure 5-24.

Figure 5-24. *The annotated server home view now shows the total party size at each table*

Concatenating the two annotations gives us a query for all of the parties assigned to a server. Let's write that as a Factual query so we can better understand how this view behaves.

```
query partiesAssignedToServer(s: Server) {
  match a: Assignment where a.server = s
  then sp: SeatParty where sp.table = a.table
    such that not exists b: BusTable where b.seatParty = sp
}
```

When read directly, the query gives instructions on how to populate the view. First, find all assignments for the server. Then for each assignment, find all parties seated at the table. This interpretation of the query gives us the initial behavior of the system. It executes the query when the view is loaded to find all seated parties.

Now let's examine the query to see how we should respond to changes. Because the results of the query are displayed to the server, we want to notify the server when the results change. Let's look at the structure of the query to determine which events can cause that change.

The query is composed of three clauses: one based on `Assignment`, a second based on `SeatParty`, and a third based on `BusTable`. This is a clue that any of these three events can lead to a notification. We can split the query along these seams to determine which servers to notify of these events.

The first clause tells us that the view will change when there is a new `Assignment`. When the manager assigns a table to a server, the query will be affected *for that server*. The server will need to be notified. The assignment tells us exactly *which* server to notify. Any parties already seated at the assigned table will appear in this view as a result. More formally, we can write the following Factual queries to give us the server to notify and the parties that were added:

```
query serversToNotify(a: Assignment) {
  match a.server
}

query partiesAdded(a: Assignment) {
  match sp: SeatParty where sp.table = a.table
    such that not exists b: BusTable where b.seatParty = sp
}
```

This pair of queries describes the behavior of the system when an `Assignment` occurs *after* a `SeatParty`. Using the second query, we find all existing parties still seated at the newly assigned table. If the result is not empty—meaning that the table is occupied—then the first query tells us which server to notify.

But there is another valid ordering that the `partiesAssignedToServer` query tells us about. The second clause of the query implies that the view will change when there is a new `SeatParty`. Upon seating a party, we have to notify the server assigned to that table. We can find the server by *inverting* the query. For completeness sake, the query giving the party added is listed as well.

```
query serversToNotify(sp: SeatParty) {
  match a: Assignment where a.table = sp.table
  then a.server
}

query partiesAdded(sp: SeatParty) {
  match sp
}
```

Notice that the second query simply returns the new SeatParty fact. It has no additional clauses. In particular, it does not mention the BusTable fact, which would ordinarily remove it from the results. The reason for this is that this query describes how the system responds when a new SeatParty fact is created. At the moment of creation, there can be no BusTable fact: there has not been time for any successors to be created. The formal justification for this decision is given in *Chapter 9*.

Using the inverted query, we can determine the behavior of the system when Assignment happens *before* SeatParty. Given the fact that a party is being seated, we find all assignments for that table. Each assignment gives us a server. We notify each server that they have a new party: the one that was just seated.

The third clause—the creation of the BusTable fact—does not lead to the addition of parties. It leads to removal of parties from the server's attention. If our goal was updating the view, we would need to include that scenario in our analysis. But as we are currently concerned with notification, we can make the decision only to notify servers of parties *added*, not *removed*. We will therefore skip the third scenario.

At first, you may need to convince yourself that this pair of behaviors gives all of the servers that need to be notified for each event, as well as all parties that the server needs to learn about. They cover all valid orderings of the events and let nothing fall through the cracks. You might want to reason through several different scenarios to determine why this is true.

Later, however, you will learn that inverting a query is a mechanical process. It can be done for *any* query, and it always produces a complete and correct result. You will learn how to perform this process in *Chapter 9*. The important takeaway for now is that analyzing a system from the perspective of Factual queries will reveal *all* of the valid orders of events. It describes exactly how the system should behave in each of those permutations. And it always converges to the same outcome.

Consequences

You have iterated over your model several times, and each pass has refined the facts a little bit more than the last. You have a clear picture of how the use cases break down into individual decisions. You know how those decisions relate to one another. You have identified the actors responsible for each decision. From that, you have identified pivots and conversations within the model that support points of collaboration. You've expressed the information that those actors see in terms of queries and from those identified which events lead to notifications and updates.

Now you can answer some very real questions about the capabilities of your model. From the design decisions that have led to this point, you can derive the constraints under which the resulting application will perform. You can determine the consequences of your modeling decisions. This will help you decide if these consequences are satisfactory and, if not, show you what compromise you need to make before you build the system.

The consequences of historical modeling are not arbitrary constraints. They limit our capabilities to only those things that can be easily done in a distributed system. If there is something that the model does not allow, then that is because implementing that feature in a distributed system would be prohibitive. It would cause blocking, loss of autonomy, or reduced scalability. Consider carefully whether you need that feature. If you do, you will need to implement it with a static model. You should be aware of the compromise that you are making when you do so. Let's examine three of these constraints in detail: indexes, number of results, and order of results.
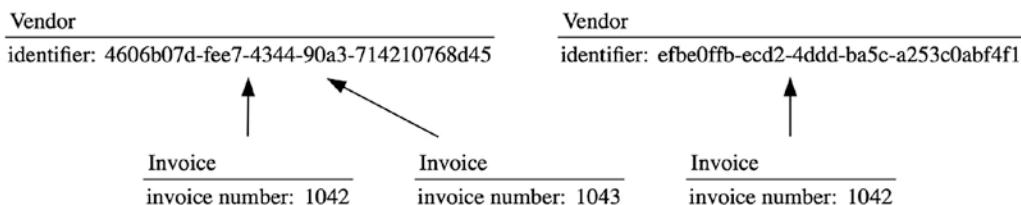
Indexes

The first constraint that you will need to consider will be how the historical model can be indexed. This will affect uniqueness, navigation, and searching. For uniqueness constraints, consider that you cannot enforce that only one fact in a distributed system has a given value for one of its fields, unless that is the only field that it has. For navigation, consider that you cannot query for facts based on only one of its fields. The best you can do is to reconstruct a fact given all of its fields and then query for successors. Searching, on the other hand, is an activity best done outside of a historical model; determine what should be searchable and annotate how the information will be sourced.

Uniqueness Constraints

Uniqueness constraints are often quite desirable and yet difficult to implement in a distributed system. In an accounts receivable system, for example, you may wish to impose a uniqueness constraint on invoice number. If you have identified the invoice fact to have extra fields in addition to the invoice number, then you will not be able to enforce this constraint in a distributed system. You will need to collect all invoices into the same place and only then verify that no two have the same number. Please see the *Outbox* pattern in *Chapter 8* for ways to implement this concept.

In a historical model, the entire collection of fields—including predecessors—uniquely identifies a fact. Let's use this premise to model an invoice with a uniqueness constraint. Invoice numbers are not universally unique; they are only unique within the scope of a single vendor. And so our `Invoice` fact would also have a `Vendor` predecessor. The combination of `Vendor` and invoice number is sufficient to identify an invoice. It must also therefore be sufficient to *construct* an `Invoice` fact, as illustrated in Figure 5-25.

Figure 5-25. *An invoice that has a unique invoice number per vendor cannot have any other fields*

And so to model an invoice that has a unique invoice number, you would need to ensure that the invoice number was the only field in the fact, aside from the `Vendor` predecessor. The fact could have no additional fields, such as `shippingAddress` or a `Customer` predecessor. Adding these fields would weaken the constraint on the invoice number and allow invoices with different addresses or for different customers to have the same number.

Invoice numbers are an example of a *generated* unique identifier. Not only must the uniqueness of the identifier be enforced, but a new invoice must also be given a new invoice number. Even if the unique identifier is isolated, as in Figure 5-25, generation must still occur in a single place. Make a note in your model that a certain field is generated, and define the rules by which it will be unique.

Navigation

Next, let's examine the effect of indexes on navigation. As we saw previously, facts are the starting point for queries in a view. As the user navigates from one view to the next, they select a new fact as the starting point of the next view. We cannot query from a field; only from a fact. If we *want* to start from a field, then we must be able to construct the corresponding fact. In other words, such starting points must be the *only* fields in their corresponding fact.

We observed this first when we extracted `table number` from the `Seat Party` fact to create the `Table` fact. Because of this choice, the table number can now be used as an indexed field. It isn't the field that the system indexes, but rather the entire `Table` fact. Servers can navigate the data model as they select one table or another to drill into more detailed views.

Breaking out an indexed field into its own fact is the only way to initiate a historical query from a field. A SQL `WHERE` clause, on the other hand, can specify a field name, even if the target table contains more fields. But SQL executes on a single database, or database cluster. In other words, it executes at a given location. Even a distributed query using a map-reduce mechanism like Hadoop scales only to a given cluster. Both SQL and NoSQL queries alike are outside of the more constrained historical query. Make a note in the model if this kind of query is required, so that the development team can integrate accordingly.

Searching

And finally, let us examine how indexing affects searching. Searching differs from other indexed query activities in that it allows much more flexibility. A search can match against part of a field, as in a prefix, substring, or SOUNDEX search. Searches can also include composition, such as Boolean operators or conditionals. Search engines typically have complex and expressive query languages.

Searching is an inherently location-dependent activity. All of the records that you search need to be in the same location. That single "location" may indeed be a cluster of distributed nodes, but we consider it a single location for a number of reasons. First, the nodes tend to be homogeneous: each one was created specifically to be a member of the cluster and therefore tends to run the same operating system, search engine, and data store. Second, the nodes tend to be co-located. They are rarely geographically dispersed and almost never operated by different organizations. And third, the number of nodes

tends to be well-known or constrained. The distributed search algorithm needs to have a good idea when all of the nodes have reported in so that it can aggregate and present the results.

Static data stores are best suited for search. Good examples include Elasticsearch or other Lucene variants. These data stores are optimized for indexing complex records or documents using a multitude of different kinds of indexes. Some indexes are designed for numerical data, allowing range filtering and sorting. Other indexes are optimized for filtering chunks of text for substring operations. To use these systems effectively, a team must understand how documents are to be fed into the search engine and the various ways in which they will be indexed. This is where analysis can provide a great deal of value.

Write a query that determines which facts are eligible to search. No user will execute that query and view all of the results; there will simply be too many. Instead, a service will use that query to subscribe to facts that need to be indexed. It will transform the facts into searchable records and add them to the data store. This service can then run continuously, looking for additions, modifications, and deletions of documents that must be applied. The mechanism for identifying those events is detailed in *Chapter 9*. Document the source query, each of its inverses, and the mapping from those facts into a searchable document.

Expected Number of Results

The second constraint that a historical model imposes upon our system is the number of results that we can expect from any given query. In a SQL query, it is feasible to write a WHERE clause that you know will match at most one record. This occurs when the filter is based on a unique index or primary key. The database management system enforces these constraints and implicitly applies the assumption of a single result to the query. Developers will often carry this assumption into their code and ignore any subsequent rows that the query might return. Occasionally, they will take the extra precaution of logging or raising an exception when more than one result is returned. But they will almost never adjust their code to allow for more results.

A historical query, however, cannot restrict the number of expected results. The reason is the same as the analysis we just performed previously: the lack of uniqueness constraints. Since uniqueness cannot be scalably verified in a distributed system, historical models offer no such guarantee. This forces you to consider what should

happen if a query returns multiple results when you only expect one. Should the data be aggregated? Should the multiple results be listed? Should the view highlight more than one result as an issue to be resolved?

Sometimes a combination of factors makes it extremely unlikely that a query will return more than one result. We have seen a good example of this in the restaurant system. Recall that the host view removed a table from the list when a party was seated. This was accomplished by sourcing that list from the following query:

```
query tablesAvailable(r: Restaurant) {
  match t: Table where t.restaurant = r
    such that not exists s: SeatParty where s.table = t
      such that not exists b: BusTable where b.seatParty = s
}
```

Because this query is one of the sources for the command that creates SeatParty, this view will not construct two SeatParty facts with the same Table—at least not until the first seating has been bussed. This would therefore lead us to conclude that the query for parties seated at a given table would return at most one result.

```
query partiesAtTable(t: Table) {
  match s: SeatParty where s.table = t
    such that not exists b: BusTable where b.seatParty = sp
}
```

The model, however, makes no such guarantee. Our confidence in this constraint is based only upon the behavior of a single view. Consider other possibilities that would circumvent that view. Is it possible that another subsystem could assign parties? Could two different hosts have the view open at the same time? If we allow such flexibility, then it is possible that two seatings for the same table might occur. In the server view, we opted to allow for multiple results by summing the party sizes (note the Σ). Other valid choices might have been to list the parties, or alert the server of a potential problem. No matter what your choice, you must acknowledge that every query can return multiple results.

There is one exception, however. A query that is based entirely on predecessors is constrained by the cardinality of those predecessors. In other words, a query that matches on a singular predecessor will always return one result. If that predecessor is optional, then it could return no results. But unless that predecessor has a cardinality

of `many` (*), then it will return no more than one. We've seen an example of such a query when determining which server to notify of a new table assignment.

```
query serverToNotify(a: Assignment) {
  match a.server
}
```

This query will always return exactly one server. Queries like this often arise as a result of query inversion. They rarely occur directly in views. The reason is simple: these queries cannot change. The facts upon which they are based are immutable. No new fact will cause this existing fact to have a different predecessor.

No Implicit Order

The final constraint on distributed systems—as exposed by the historical model—is that query results are not ordered. Even when two nodes have reached consensus and return the same set of facts from a query, they could return those facts in very different orders. We would like to believe that the order in which facts appear is the order in which they were created. But in a distributed system, there is always the possibility that two decisions were made concurrently, each without knowledge of the other. When this happens, the nodes on which those decisions were first captured will certainly disagree to the order in which they were made.

Implicitly, the results of queries are sets, not lists. Sets have no order, only membership. There is nothing that can be inferred from the order of the results, only from the presence or absence of facts. Software, however, is full of lists. Iterating over a collection in order is a common feature of most languages. We cannot even present an unordered set to the user; it will always appear as a list. If we iterate over the results of a query, then we will find that they are in *some* order. We just have to be careful not to depend upon *which* order.

One goal of a distributed system is to have different nodes reach a consistent state. When we build a system based on the principles of Historical Modeling, we can use CRDTs (conflict-free replicated data types) to prove that we will reach consistency, as shown in *Chapter 4*. The data type that we chose was the *set*, not the *list*. Had we chosen to use lists, the math would not have worked out. As it is, we have proven that our sets will converge. The final step is to prove that the *projections* of those sets—the information that we display to the user—will converge as well.

Aggregates

The projections that we present will be one of two kinds: aggregates or iterations. Let's examine aggregates first. An aggregate is simply a value computed from an ordered collection. The sum of a list of numbers is an aggregate. Maximum, product, and standard deviation are also aggregate functions that can be computed over a list of numbers. One important feature of all of these aggregate functions is that they are *commutative*. The sum is the same no matter the order of the numbers you add. The same can be said for maximum, product, and standard deviation. Commutative aggregates are useful because they ignore the order of the list.

Other kinds of aggregates are not commutative. One example that appears frequently is string concatenation. Given a list of strings, it is common to append them to one another, separated by commas. While this aggregate is useful, it is not commutative; the result of string concatenation depends upon the order of those strings. As a result, two nodes may compute different results. Before using this kind of aggregate, you should first ensure that the facts are in a deterministic order.

Iterations

This brings us to the second kind of projection that we need to consider: the iteration. Iterations appear on the user interface as lists. They also appear as non-commutative aggregates such as string concatenation. An iteration makes it apparent to the user when there is more than one result and implies a certain order to those results. Since the results don't have an implicit order, an iteration needs to impose one.

Find some feature of each fact that can determine an order. For example, you might order Table entities by their table number. Before presenting the results to the user, order by that field. As long as every node orders results in the same way, the iteration (or non-commutative aggregate) will appear to be consistent.

You might find that you want to put facts in order according to a mutable property. Being mutable, these fields are not members of the facts themselves. They are members of successor facts, which might not be present at every node at the same time. When this is the case, be sure to document a further query that returns these property facts. Express the order of the main results in terms of a projection of the secondary results. As you do so, remember that the secondary query can *also* return multiple results, and so it too needs a commutative aggregate or deterministic ordering.

Creation Order

Finally, there is one way to deterministically order events in a historical model by creation time: record the creation time as a field. This is certainly the simplest and most direct way to impose a deterministic order on results. However, this has two consequences. First, it requires that the clocks on the source nodes be synchronized. To the extent that the clocks are out of sync, new facts might be inserted before older ones. Second, and more importantly, this forces the creation time into the identity of the fact. Two facts that are created at different times, but would otherwise have the same field values, will now be considered as discrete facts. Sometimes this is desired behavior; other times it is not. Only include the creation time in a fact if you need an extra identifier, as discussed in the *Identity* section in *Chapter 4*.

The choices you make while analyzing a problem will have consequences upon the final implementation. Some things that are easy to say are not easy to implement. Even thinking about the problem in terms of a small set of nodes or collaborators is not sufficient to expose all of these assumptions. Distributed systems impose their own unique set of constraints.
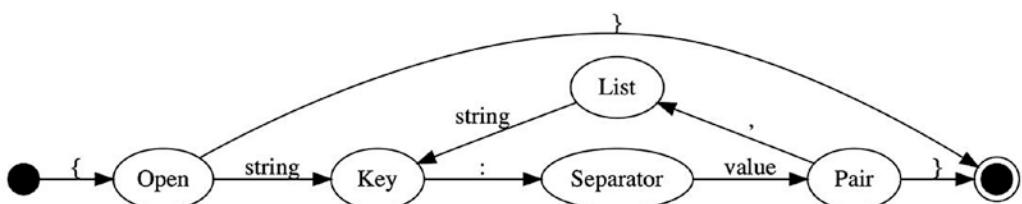
The rules of Historical Modeling are deliberately restrictive so that the consequences of these choices become clear. Rather than allowing uniqueness or search on any field, historical models can only be indexed on the fact itself. Rather than being able to restrict the number of possible results, historical queries always allow multiples. And rather than implying an order, historical results require that projections either be commutative or explicit. Augment your analysis with the extra information necessary to clarify your assumptions, so that the team understands any compromises they might have to make before they write a single line of code.

CHAPTER 6

State Transitions

Among the most powerful tools available to a software developer is the finite state machine. This mechanism—sometimes illustrated as a state transition diagram—describes a multistep process. The machine moves from one state to the next as it encounters input. Each unit of input determines which arc of the graph the state machine follows. That arc decides how the input is processed and what state the machine finds itself in to receive the next unit.

This tool is a natural choice for solving problems such as parsing computer languages and input files. The specification for the JSON data interchange format,[1] for example, is described in terms of state machines. The language is broken down into discrete structures, each defined by a state transition diagram. When the parser expects to see a particular structure, it enters into the states depicted in the graph. For example, the diagram for parsing an object can be drawn as Figure 6-1.

Figure 6-1. *State machine for parsing a JSON object, as described in ECMA-404*

The diagram as it appears in the specification is drawn a bit differently, but here we name each of the states and label the edges with the input that causes that transition. This is a common way of drawing state transition diagrams. A machine's state is mutable: it changes as the machine consumes input.

---

[1]The JSON Data Interchange Syntax. Standard ECMA-404. ECMA International. Second Edition, December 2017.

Given the expressive power of state machines, it is no surprise that many people have applied them to distributed systems. Chris Patterson, creator of the distributed systems framework MassTransit, recommends using state machines for managing sagas.[2] Johnathan Oliver, industry-recognized expert on distributed systems, observes that "Process is best implemented using a state machine."[3] These and other leaders in this space have demonstrated how to use state machines to manage business processes, break up long-running transactions, and protect against message duplication and ordering.

My experience, however, has shown that state machines are a poor choice for both understanding and implementing business processes in a distributed system. State machines are good for implementing parsers, but not distributed business processes. All of the input to a parser comes from a single source: a block of memory or a character stream. Input to a business process, on the other hand, arrives from many sources. It often represents business decisions made by different people with different views of the system. Parsing happens all in one place, so knowing the state of the parser is easy. But business processes happen in many locations simultaneously, making it difficult to know the single state of the system.

Let's explore some of the challenges that you can expect to face while applying state machines to distributed systems. We will first try to solve those problems using the tools readily at hand. But eventually we will find that the problem directs us toward a different solution. We'll discover how representing state transitions as immutable facts solves both analytical and technical problems in distributed systems. And then we will see how to rebuild both our understanding and our implementation on top of those new techniques.

Many Properties

When working on a supply chain management system, I ran into the first of the problems with state transition at scale. We were using an enterprise resource planning (ERP) solution to build the application. Like many ERP systems, the one we were using was extremely customizable. It allowed application developers to define their own entities, properties, and operations. It also allowed them to define a state machine.

---

[2]Chis Patterson, State Machine for Managing Sagas. Los Techies. 2009. https://lostechies.com/chrispatterson/2009/01/17/state-machine-for-managing-sagas/

[3]Sagas with Event Sourcing. https://blog.jonathanoliver.com/cqrs-sagas-with-event-sourcing-part-i-of-ii/
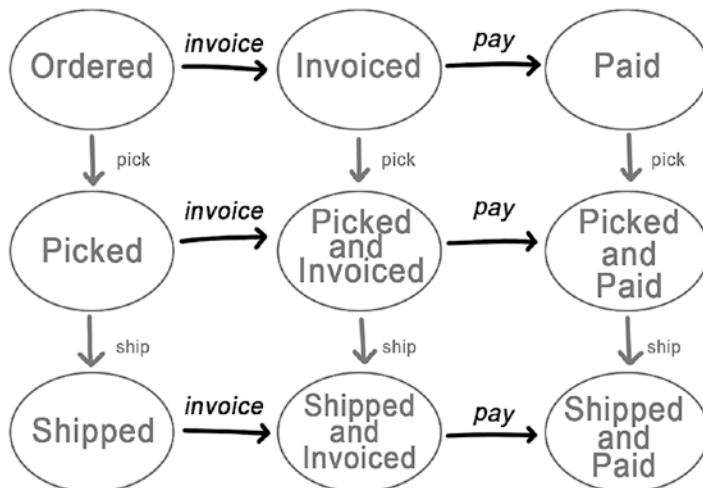
Within the ERP system, a developer would identify the states that an entity would transition through as it progressed along a business process. They would define which state transitions were permitted and which were forbidden. Each transition was an arrow between two states and represented a step in the process. Developers would attach actions to those steps in order to customize the process.

For simple state machines, this model was manageable. But as the system became more complex, we found ourselves multiplying new features by the number of existing states. It became apparent to us that the state of an entity represented more than one property. Sometimes those properties interacted, and other times they did not.

Shipping and Billing

To demonstrate the problem, let's examine a somewhat simpler example. Suppose we have built a system that accepts orders for products in a warehouse. Once the order is placed, the shipping department picks the product and ships it to the customer. Meanwhile, the billing department invoices the customer and receives payment. We want to allow these two operations to happen independently. Since we are using an ERP system that defines a state machine per entity, we design a graph of states that combines the two ideas, like the one in Figure 6-2.
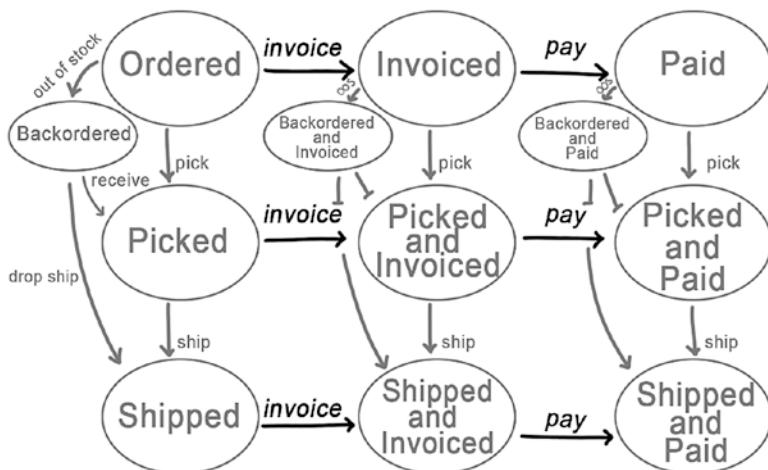
Figure 6-2. *States of an order: billing moves to the left, and shipping moves down*

This state machine allows shipping and billing to operate orthogonally. As the shipping department picks and ships orders, the state moves down the page. And as the billing department invoices and receives payment, the state moves across. Eventually, like a cab driver making their way through Manhattan, the state reaches the lower right.

Introducing Back-Orders

After the system has been in operation for a while, the company realizes that they are turning away business when they run out of stock. To remedy this situation, they expand their operations to include back-orders. When a product is out of stock, rather than pick it, the shipping department orders it from their supplier. They can either receive it themselves and then ship it to the customer, or they can ask the supplier to drop-ship it directly to its destination. The modified state transition diagram appears in Figure 6-3.
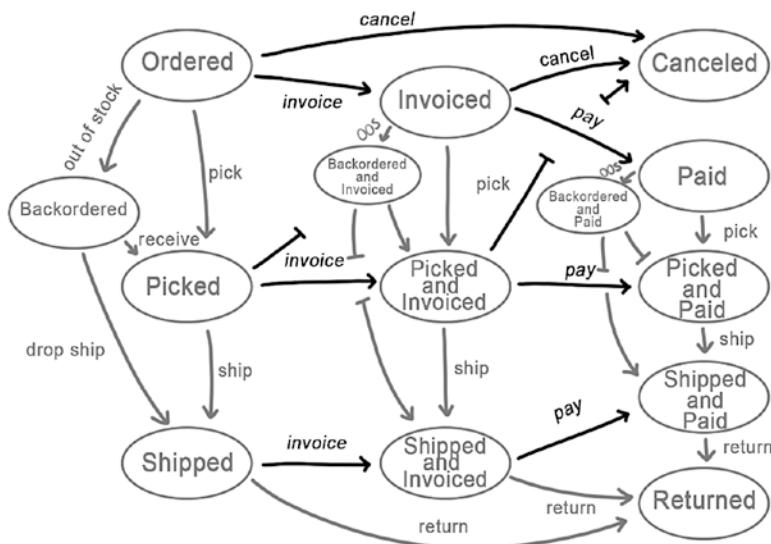
Figure 6-3. *Out-of-stock products are back-ordered. Back-ordering does not interact with billing*

Adding the back-order feature required us to add three states to the diagram. While the product is back-ordered, the order can still be invoiced and paid. We therefore need to combine the back-ordered state with both the invoiced and paid states. Because this new feature has no interaction with the existing billing feature, the new transitions have exactly the same actions associated with them. There is no difference between drop-shipping an order before or after it has been paid.

186

As we add features to this model, we will find that we multiply the number of new states by the number of existing states in independent features. There were three states in the billing process, so one new state became three. The more independent processes are at work within a single state machine, the larger this combinatorial explosion becomes.

Cancellations and Returns

After a bit more time, the company decides that it is not properly accounting for cancellations and returns. A cancellation occurs before the order is shipped and may involve a refund. A return, on the other hand, occurs after the order has shipped and requires restocking. The new state transition diagram is in Figure 6-4.

Figure 6-4. *Cancellations happen before shipping; returns happen afterward*
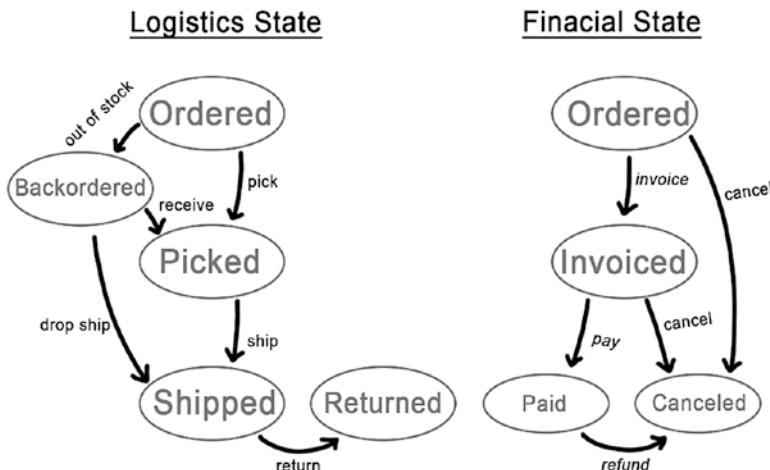
A cancellation is handled almost entirely by the billing department. Because the product has not yet been shipped, there is little work for the shipping department to do. They simply have to recognize that an order has transitioned and that the `ship` operation is no longer allowed. There is no arrow from the `Canceled` state labeled `ship`.

A return, on the other hand, is handled by the shipping department. The item has already been shipped, and so they need to restock it upon receipt. Only the transition from `Shipped and Paid` to `Returned` involves the billing department. In that situation, they must issue a refund.

When adding features that interact with existing states, we can often avoid a combinatorial explosion. However, we pay the price in terms of complexity. We must now examine every existing state to determine the correct course of action should that new operation occur at that time. Some of those transitions will require compensating actions, while others will not.

Parallel State Machines

An application developer faced with this issue might reach for the nearest solution at hand. This solution is not the best, but it is the most readily available. With a small refactoring, a single co-mingled state can be broken into two or more parallel states. In this particular example, the simple fix for the problem is shown in Figure 6-5.

Figure 6-5. *The logistics and financial states of the order entity are separated*

Instead of a single state, the order entity has two. The logistics state keeps track of the process as seen by the shipping department. The financial state captures the process of the billing department. Insofar as those processes can progress independently, these two state machines run in parallel.

When those processes interact, however, an operation needs to be sensitive to both states at the same time. If an order is returned, we expect the logistics state to start from `Shipped`. The operation will transition it to `Returned`. However, it must also consider whether the financial state is `Paid`. If so, it must issue a refund. In any case, it must move the financial state to `Canceled` so that the customer is not charged.

Parallel state machines relieve the combinatorial explosion that occurs when one state models independent processes. They also make the complexity a little easier to express, because edge cases don't need to be considered for every distinct combination. They are near to hand for an application developer using mutation to build a system. But they are not the ideal solution; they still leave several problems unsolved. One of those problems has to do with aggregates—entities with many children.

Many Children
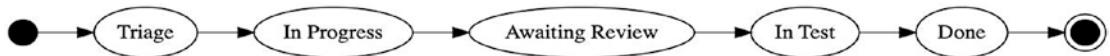
After solving the problem of an entity having many properties, we are still left with the problem of an entity having many children. On the one hand, each child might individually have its own state. But, on the other, the state of the parent might depend upon the state of the children. This leads to an interaction among state machines even as they are created and destroyed dynamically.

The parent state machine tracks an overall process. At a certain point, the process branches. Each child entity must individually progress through a child process in parallel. Only after all children have finished do we allow the parent to proceed. After the main process has advanced, adding a child might arrest progress and move the parent state backward. Deleting that child should again push the parent state forward.

Describing all of these interactions as state transitions becomes incrementally more tricky, as each new scenario spins a growing web of edge cases. Our first inclination is to create a mechanical solution: "When in this state and this happens, do that and move to that state." Such logic quickly becomes difficult to reason through. The web of edge cases becomes a hiding place for defects. We cut this Gordian Knot by representing state not as a mechanical set of transitions, but as a declarative function.

Software Issue Tracking

One of my clients uses a popular issue tracking solution for managing bug fixes, features, user stories, and other changes to their software. Like an ERP system, this program lets users customize their workflow by defining states and transitions. By default, a bug might start in `Triage`, transition through `In Progress` and `In Test`, and finally end up in `Done`. My client, however, has modified this workflow. They are required to ensure that all changes have been reviewed for regulatory compliance. They have therefore inserted `Awaiting Review` before the change is `In Test`. The modified state transition diagram of a defect appears in Figure 6-6.

189

Figure 6-6. *The state transition diagram for a software defect, modified to ensure changes are reviewed*

Features go through a similar workflow. Every software change is motivated by a feature or a bug fix. Because of this, my client can be confident that every change that makes it to production has gone through the review step. This solves the problem that was originally before them: ensuring compliance. However, it creates others.

Every software change is a commit in the version control system. It may take several commits to fix a bug. It is common for developers to separate refactoring changes from fixing changes to make the intent more clear. And yet, the entire bug is reviewed as a single unit. Because this state transition diagram is implemented in a commercial issue tracking tool, my client is constrained in how they can break down the entities. The tool does not allow them to represent the state of a commit as it relates to an issue. And so, they capture the review at the `Bug` entity and not the `Commit` entity.

Child State

To solve this problem without modifying the issue tracking system, my client employs other third-party tools. One of them is a change review system that allows developers to comment on individual lines of each commit. Within this tool, developers and reviewers have a conversation over the code to discuss motivation, recommendations, and possible corrections. At the end of this process, the reviewer changes the state of the commit. The state transition diagram applied at the commit level appears in Figure 6-7.
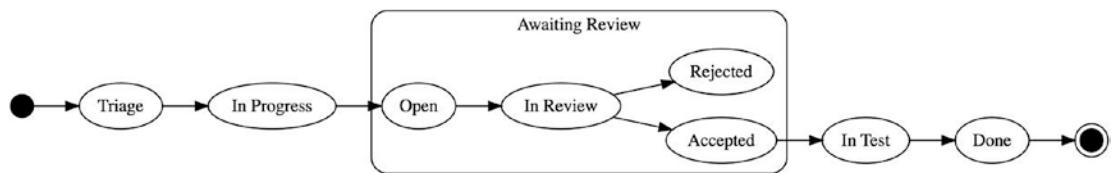
Figure 6-7. *The state transition diagram of a commit, as implemented in a separate tool*

Combining these two tools requires discipline. A developer moves the issue tracking tool to `Awaiting Review` and then invites reviewers to join the conversation in the

change review system. Reviewers move the commits through their individual workflows, requesting additional commits to resolve any issues that come up. Only after all commits in the branch are in `Accepted` is the code merged and the defect moved to `In Test`.

Composite State Transition Diagrams

Assuming that we could combine these two tools to suit our needs, a developer might choose to merge the two state transition diagrams. A bug would move through the parent state machine, and each commit would move through the child state machine. The composite state transition diagram appears in Figure 6-8.

Figure 6-8. *A composite state transition diagram allows the parent to progress only after all children have reached a terminal state*

To implement this diagram mechanically, we need to consider how the state of the parent interacts with the state of the children. If the parent is `Awaiting Review` and a child becomes `Accepted`, check whether all other children are also `Accepted`; if so, transition the parent to `In Test`. If a `Rejected` child is removed from the branch, see if it was the last one; if all others are `Accepted`, move the parent forward. And if a parent is `In Test` or `Done` and a new child is added, move the parent back to `Awaiting Review`.

Does that mechanism account for all possible scenarios? It is difficult to tell. The burden of proof is on the developer implementing the solution. Verifying all of those edge cases is up to the tester. And as we add more states, the number of edge cases grows.

A Declarative Function of States

Rather than a mechanical solution, we can define a declarative one. Declarative solutions lend themselves to proof much more easily than mechanical solutions. It is easy to look at a declarative statement and see whether all possible conditions are listed. The function computes the overall workflow based on the states of the individual components.

We can describe the workflow of a bug as a function of the parent and child states. For this we will invoke a *universal quantifier*. That's just a fancy phrase meaning "for all." The parent states `Triage` and `In Progress` map directly to the workflow. But the difference between `Awaiting Review` and `In Test` depends upon the child states. A bug is in test if, *for all commits in the branch,* the commit is in `Accepted`. Otherwise, it is awaiting review.

This logic can be written declaratively as in the following pseudo code:

```
workflow (bug) =
  if bug.state = Triage
    then Triage
  else if bug.state = InProgress
    then InProgress
  else if not for all commit in bug.branch, commit.state = Accepted
    then AwaitingReview
  else if bug.state = InTest
    then InTest
  else Done
```

The check for all commits being `Accepted` stops the workflow from progressing beyond `Awaiting Review`. It doesn't matter if the parent state has moved on; any child that is not completed will hold the workflow back. This declarative description means that we don't have to write a machine that handles every edge case. If a new commit is added while the bug is in test, the universal quantifier causes this Boolean expression to return to awaiting review. And if a commit is removed from the branch, the universal quantifier reevaluates and allows the workflow to progress. Without a declarative expression, a developer would have to code for those edge cases and prove that all possible scenarios have been explored.

The declarative function finds the edge cases caused by interactions between parent and child states. Writing the function in terms of states is a short step from managing state transitions through mutation. However, it still is not the ideal solution. Now that we have a couple of examples at our disposal, and have explored a few candidate solutions, we can now dive into the more difficult issue of conditional validation. This will finally lead us to abandon solutions based on mutability.

Conditional Validation

As entities move through a process, they don't just change state. They also accumulate data. As we transition an entity from one state to the next, we will want to record that new data. We ideally perform both operations within a single transaction so that the presence of data is consistent with the current state.
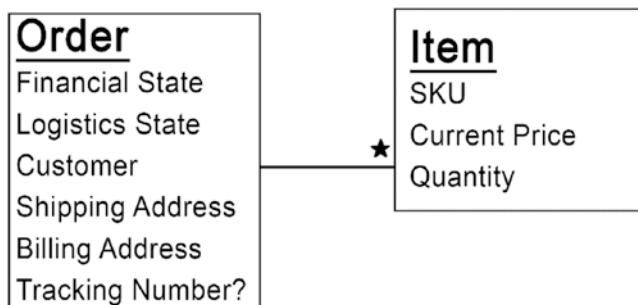
Depending upon the state of the entity, data fields might be null, or they might require a value. The validation of the data fields is conditional upon the state of the entity. The type systems of our database and programming language tools typically do not capture such conditional validation. So we find ourselves weakening the declared types in order to compensate.

We've examined two examples: order fulfillment and software change tracking. In each of these examples, we found solutions to state transition issues as they arose. But we haven't analyzed the other fields of the entities. Let's take a look at those fields and see if we have any conditional validation.

Nullability Based on State
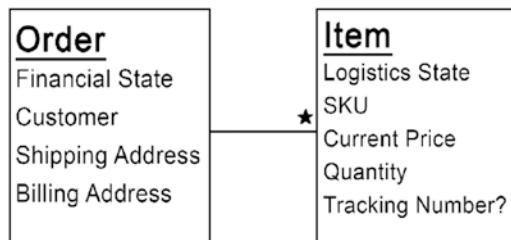
In the order fulfillment example, we want to record information about the order and each item that it contains. For the order, we record the customer, shipping address, and billing address. For each item, we capture the SKU, current price, and quantity. When we ship the item, we will also want to include the tracking number. Where does this belong?

At first, it might seem that the tracking number should be part of the order, as shown in Figure 6-9. When the order is placed, we don't yet have a tracking number, so we allow this field to be `null`. It is only filled in when the order is shipped. And so a shipped order will have a non-null tracking number. Setting the state to `Shipped` and filling in the tracking number would happen in a single transaction.

Figure 6-9. *Tracking number is a nullable field in order, which also holds logistics state*

But perhaps the logistics state should be part of the item, as in Figure 6-10. Each item can be back-ordered, picked, shipped, and returned individually. If so, it makes sense to put the tracking number on the item instead of the order. Again, when an item is added to an order, it doesn't have a tracking number. And so this field is `null` until the item is shipped.

Figure 6-10. *Tracking number is a nullable field in item, where we have moved logistics state*
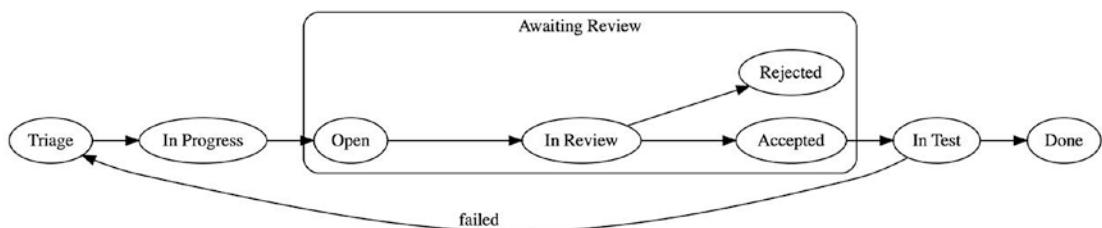
Whether we put the tracking number on the order or on the item, we face conditional validation. The tracking number must be `null` if the logistics state is `Ordered`, `Backordered`, or `Picked`. It must not be `null` if the logistics state is `Shipped` or `Returned`. Once we have passed a certain point in the workflow, the tracking number is expected to be present. Validation of this field is conditional upon the state of the entity.

Cycles in State Transition

Let's look at another customization that my client made to the software change tracking system. The system allows users to add custom fields to bugs and features. My customer added a field to keep track of the tester who verified the change. When the bug or feature is first created, this field is null. Once it has been tested, the tester fills in their own name and moves it to Done. Filling in the field and changing the state are both done in the same transaction.

This new field completes the picture. From the commit, we had a permanent record of who made the change. From the review system, we know who reviewed each commit. And now this new field tells us who tested the fix. But the field also creates a problem.

There's an arrow in the state transition diagram that we haven't drawn yet. A tester can evaluate a change and find that it is defective. If so, they fill in their name and move it back to Triage. This backward arrow creates a cycle, as shown in Figure 6-11.

Figure 6-11. *Failing a test moves the bug back to the beginning of the workflow*
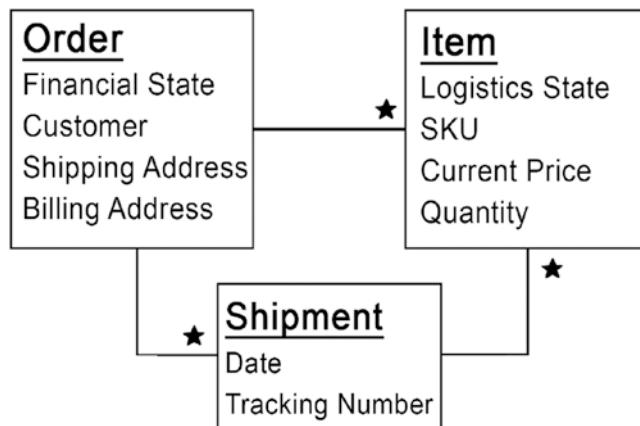
This cycle means that we cannot write the same kind of conditional validation that we saw earlier. We cannot say with certainty that if a bug is in Triage or In Progress, the Tester field is null. It will be null on the first time through, but not if the bug was previously failed. Worse yet, what happens if a different tester verifies the bug the second time? Which name ends up in the field?

Collect Data During Transitions

Conditional validation forced us to declare fields as nullable when they actually record required data. Once the state has progressed beyond a certain point, we want to ensure that that data is captured. We never want to allow tracking number to be null once an order has shipped. Nor do we want to allow tester to be null once the bug has been

tested. To change these fields so that they don't allow nulls, we need to remove them from their entities and place them in a new object. This object will be created only when the entity transitions to the target state.
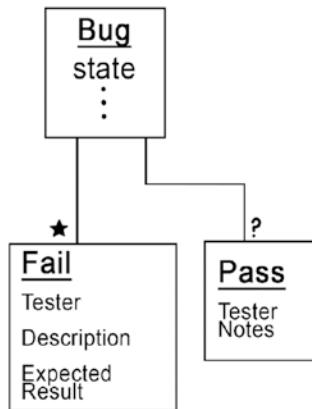
For example, when an order is shipped, we can create a `Shipment` object, shown in Figure 6-12. The tracking number is not a field of the order, nor is it a field of the item. Instead, it is a field of the `Shipment`. This object records which items were shipped and collects other data generated at that time, such as the tracking number.

Figure 6-12. *An order has multiple shipments, each with a non-nullable tracking number*

With this model, we can enforce that the tracking number is not `null`. Before the order is shipped, no `Shipment` object exists. But afterward, the object is created, and the tracking number has to be filled in.

Consider the bug tracker example. What object we can move the `Tester` field into? When a tester fails a bug fix, they can create a `Fail` object. This object could capture not only the tester but also the description of the test failure, expected result, and perhaps screenshots, logs, and other supporting materials. Conversely, when a tester passes a bug fix, they can create a `Pass` object. This object captures the tester and any additional notes. These two objects are shown in Figure 6-13.

Figure 6-13. *A bug has many fail objects and zero or one pass object. These objects record the tester*

This model creates two new objects. Each object has a `Tester` field which cannot be `null`. We no longer have conditional validation; before the bug is tested, these objects don't even exist. In addition, we have solved the problem with state cycles. Each time a bug is tested, we create a new object. Each one can have a different `Tester`, and previous values are never overwritten.

Immutable State Transitions

By introducing new objects, we have solved the problem of conditional nullability. Before an entity moves beyond a certain state, the additional object does not exist. Afterward, the object contains non-nullable fields. We no longer need to store the field within the entity. This means that we don't need to compromise the field declaration to allow the weaker type.

We have also solved the problem of cycles. Each time an entity transitions through an iteration, it accumulates more data. Each pass is recorded in a separate object. Prior passes are not overwritten.

In either case, the new object is immutable. Each of these objects represents a historical fact. There is no reason to go back in time and change these facts. We've replaced mutation with object creation. Perhaps now we can use these immutable objects to eliminate even the mutable field recording the state itself.

The Question Behind State

The problems of many properties, many entities, and conditional validation have led us to a place where we record information about a state transition in an immutable object. Our models currently have one or more mutable state fields on each entity, in addition to the immutable historical records. The application both changes state and adds historical detail in a single transaction, ensuring that the two are consistent. This leads one to wonder: is one of these redundant?
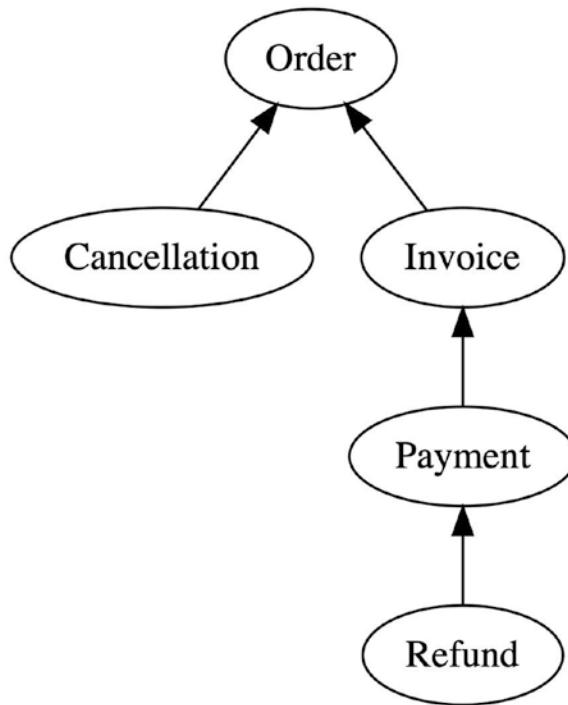
Translating a State Machine to a Historical Model

Let's go through each of the examples we've explored so far and construct a historical model of the state transitions. If we can compute the state of each entity from these historical facts, then we can be assured that the mutable state field is redundant. With that assurance, we can eliminate the redundant fields from data storage. To store something that could be computed from something else is to invite defects. When redundant fields exist, it is possible to store an inconsistent set of values. Eliminating the redundant field eliminates this class of defects.

Once we have a way of computing the current state, we can ask a larger question: what is the reason for determining the state of an entity in the first place? By examining what state is actually used for, we can map those questions down to the historical facts. This analysis will reveal that in many domains, we do not need to know the state of an entity at all. We can answer the questions behind state directly from the historical facts.

Order Fulfillment

In the order fulfillment system, we tracked the financial state of an order and the logistics state of an item through a pair of parallel state machines. Let's model each of these state machines as a history of facts, starting with the financial state. The documents that affect financial state appear in Figure 6-14. I've associated each document with its predecessors to represent causality.

Figure 6-14. *Historical facts representing the documents that the billing department manages*

The financial state of an order can then be written as a function of these documents. We will use *existential quantifiers* or statements of the form "there exists." If there exists an invoice for the order, then the order has been invoiced. If there exists a payment for that invoice, then the order has been paid.
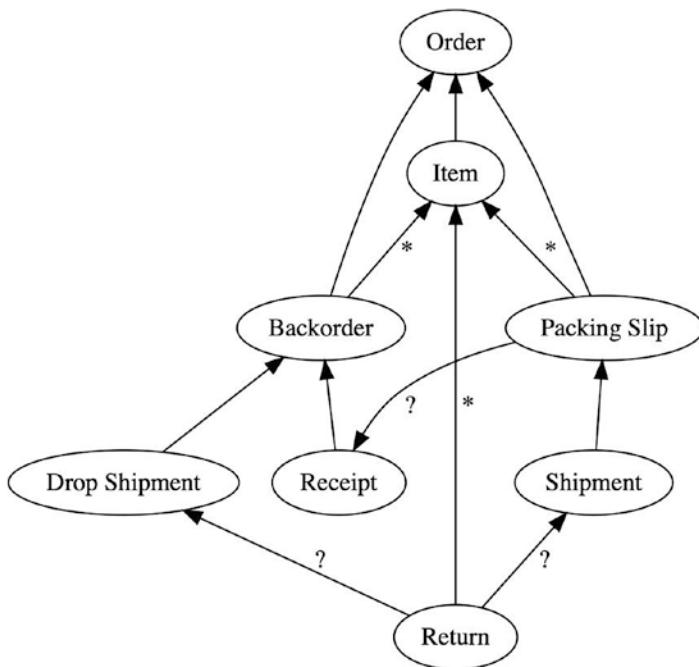
```
financial state (order) =
  if there exists Cancellation
    then Canceled
  else if there exists Invoice then
    if there exists Payment then
      if there exists Refund then
        Canceled
      else Paid
    else Invoiced
  else Ordered
```

Adding a new document to history affects the financial state of the order. We don't need any handlers to receive the documents and update the state. There is no machine for processing messages. The state is simply a declarative function on the existence of documents.

The logistics state is a little more complicated. We captured the tracking number in an immutable `Shipment` object to indicate that multiple items were shipped for a given order. The fact type diagram in Figure 6-15 retains that relationship between a shipment, its order, and the shipped items, while also accounting for other activities that affect logistics state.

Figure 6-15. *Historical facts representing actions of the shipping department*

When the shipping department picks items from the warehouse, they produce a packing slip. The existence of a packing slip therefore indicates that those items have been picked. That package will later be shipped, at which time it receives a tracking number.

Meanwhile, out-of-stock items will be back-ordered. A set of back-ordered items can be drop-shipped or received. If received, the warehouse will create a new packing slip, resulting in a shipment.
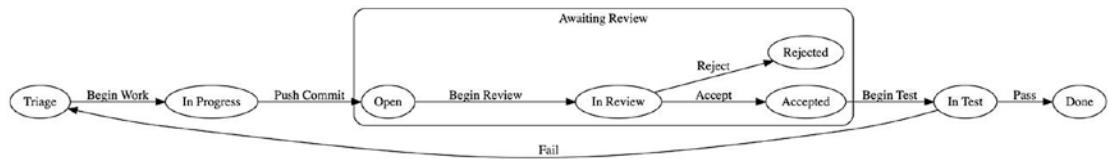
Items can be returned from either a drop shipment or a warehouse shipment. The entire shipment need not be returned, so the return has to specify which items were included. The history of facts tells the story. The following function determines the current state of an item based on the existence of those facts:

```
logistics state (item) =
  if there exists PackingSlip
    if there exists Shipment
      if there exists Return
        then Returned
      else Shipped
    else Picked
  else if there exists Backorder
    if there exists Receipt
      then Received
    else if there exists DropShipment
      if there exists Return
        then Returned
      else DropShipped
    else BackOrdered
  else Ordered
```

These two histories live side by side in the order fulfillment system. The billing department is concerned primarily with the financial documents, while the shipping department deals with the logistics events. They intersect only in reconciling payments with shipments and refunds with returns. With one additional query, we can identify orders that are out of balance and need to be corrected. We will write that query a little later. Meanwhile, let's design a historical model for the software change tracking system.
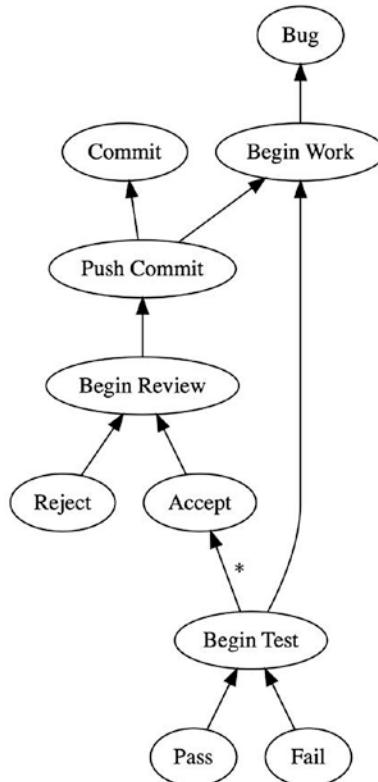
Software Change Tracking

Taking another look at the software change tracking system, we can identify the state transitions that take place with respect to a bug or a commit. We've identified two such transitions already: Pass and Fail. The other transitions appear as arrows in the state transition diagram. In Figure 6-16, we label those arrows so that we can see the full set.

Figure 6-16. *Transitions in the software change tracking system are labeled with the actions that trigger them*

Pivoting from a state transition diagram into a fact type graph, each of these transitions becomes a fact. We associate the facts with the entities that they affect. We also associate each fact with the state transition that preceded it. This ensures that we cannot take a state transition too early. The resulting fact type graph appears in Figure 6-17.

Figure 6-17. *A graph of facts representing the state transitions of a software change tracking system*

The transition `Begin Test` is performed on the bug, and so its predecessor is the last action performed on that bug: `Begin Work`. However, it relies upon all of the commits pushed as a result of that work being accepted. It therefore also has a predecessor collection of `Accept` facts. From this graph of historical facts, we can determine the state of an individual commit, or of the bug as a whole. Let's start with a function that gives the state of a commit. We express this function in terms of the `Push Commit` fact, as we don't need to consider commits that haven't been pushed.

```
state (pushCommit) =
  if exists BeginReview
    if exists Reject
      then Rejected
    else if exists Accept
      then Accepted
    else InReview
  else Open
```

This function expresses a subtle but important design decision: rejection vetoes acceptance. It tests for a `Reject` fact before it tests for an `Accept` fact. If somehow both exist, the commit is rejected.

Now that we have this function, we can use it to express the state of the bug. The bug's state is based not only on the existence of bug-related actions but also the state of all commits. For that reason, this function combines both existential and universal quantifiers.

```
state (bug) =
  if there exists BeginWork
    if there exists PushCommit
      if for all pushCommit in branch, state(pushCommit) = Accepted
      and there exists BeginTest
        if there exists Fail
          then Triage
        else if there exists Pass
          then Done
        else InTest
      else AwaitingReview
    else InProgress
  else Triage
```

Once again, failure vetoes passage. Furthermore, the bug requires both that all commits are accepted and that we explicitly begin testing. If either of these checks fails, it remains awaiting review. We don't have to code for the combination of these events to advance the state of the bug. The declarative nature of a state function takes care of that for us.

Reasons for Computing State

This exercise demonstrates that we can determine the state of an entity based on the existence of historical facts. Given this, we should drop the mutable state field from the entity records. Doing so eliminates the possibility that we fail to update it correctly while inserting a new historical fact.

Having dropped the mutable state field, we can now consider why we had it in the first place. Under what circumstances are we going to call these new declarative state functions? What is the question that we were using state to answer?

Handling the Next Action

State machine–based patterns show us one of the reasons that we want to know the state of an entity: to understand how to respond to the next action. A message handler usually follows a predictable series of steps:

- Look up the entity (by correlation ID or some other property of the message).

- Determine the state of the entity.

- Validate the message.

- Operate on the entity.

- Update the state.

The state determines how to perform the subsequent operations. We could run the declarative function to determine the current state and then from that state determine the strategy for handling a message. Or we could skip a step and simply determine the strategy directly from history. In many domains, deciding how to respond to an action is much simpler than determining state.

For example, in the order fulfillment system, the strategy for responding to a cancellation request depends upon the financial state of the order. If the order is Paid,

then we issue a refund. If not, then we simply cancel the order. Determining the state of the order is somewhat complicated, as we saw earlier. And yet determining whether an order is paid is much simpler:

```
paid (order) =
  there exists Invoice
    such that there exists Payment
      such that there does not exist Refund
```

Furthermore, the work of determining whether an order is paid is exactly the work required to find the payments that need to be refunded. And so we can boil it down to a single Factual query:

```
query nonRefundedPayments(o: Order) {
  match p: Payment where p.invoice.order = o
    such that not exists r: Refund where r.payment = p
}
```

When we receive a request to cancel an order, we run this query. If it finds any payments that have not yet been refunded, we issue refunds for them. Doing so has the side effect of transitioning the order to the `Canceled` state if we were so inclined as to run it. But if we can determine how to handle the next action without running the state function, why run it?

Finding Work Items

In addition to determining how to handle the next action for an entity, states are often used to find all entities requiring a next action. Rather than looking up the state for a given entity, systems will query for all entities that are in a given state. One common reason for running such a query is to present the results to the user as a list of work items.

For example, a software change tracking system will commonly display issues in swimlanes. Each column represents a state. When a user is looking for work that needs to be done, they will scan the issues in a given state and select one to pull forward. A developer would look in the `Triage` swimlane for bugs that are ready to be worked on. Going through all of the bugs and running the state function would be a slow way to provide this user interface. Fortunately, a Factual query gives us those results directly from history.

```
query bugsInTriage(s: Sprint) {
  match b: Bug where b.sprint = s
    such that not exists bw: BeginWork where bw.bug = b
}
```

For the sake of this query, we put all of the bugs in a sprint. This gives us a starting point for the query. We first look for all bugs in the sprint and then limit them to only those for which work has not yet begun. In other words, we only want the work items that will accept our next action. The developer can choose any one of them and create a new Begin Work fact, thus removing it from the swimlane.

That was a pretty simple example. What about something more complicated? A tester, for example, might be looking for all of the bugs that are In Test so that they can select one to verify. A bug is only in test if no commits have been reviewed and rejected.

```
query bugsInTest(s: Sprint) {
  match bw: BeginWork where bw.bug.sprint = s
    such that not exists r: Reject
      where r.beginReview.pushCommit.beginWork = bw
  then bt: BeginTest where bt.beginWork = bw
    such that not exists p: Pass where p.beginTest = bt
    and not exists f: Fail where f.beginTest = bt
}
```

The last clause filters the list based on the next action. We are only concerned with bugs that have not passed or failed. Queries for work items always include a not exists clause based on the next action. Once we perform the next action, the query will no longer return the result. This updates the user interface and removes the work item from the user's list.

Executing Compensating Transactions

One final reason for knowing the state of an entity is to determine if there are any compensating transactions that need to be applied. This is one of the features for which the Saga Pattern[4] was originally invented. Most database management systems provide a mechanism for executing several operations in one atomic transaction. These

---

[4]Hector Garcia-Molina, Kenneth Salem. Sagas. Department of Computer Science, Princeton University. 1987.

transactions are intended to be short-lived. Holding a transaction open for an extended period of time can block other operations, seriously impacting the scalability of your solution. The Saga Pattern associates compensating transactions with intermediate steps, so that they can be rolled back should a problem arise. Compensating transactions allow you to commit the intermediate database transactions and yet still handle long-running activities.

Another way to think about compensating transactions is to consider whether a state is desirable or not. Ordinarily, database transactions would prevent a system from entering undesirable states, as the transaction can be rolled back. In a Saga, however, database transactions are committed more frequently to improve scalability. It is possible for a sequence of smaller database transactions to leave the system in an undesirable state. A compensating transaction is a corrective action that can be taken if the system is left in such a state.

One example from the order fulfillment system has to do with reconciling returns with refunds. It is undesirable for the financial state of an order to be Paid while the logistics state of its items is Returned. We can arrive at this state in a couple of ways: we can receive payment for items that have been returned, or we can receive a return for items that are already paid for. A state machine–based implementation of the Saga Pattern would look for each of these situations in a separate handler. The payment handler would look for items in the Returned state, and the returns handler would look for orders in the Paid state. But doing so duplicates logic that could simply be expressed in one place. The following Factual query identifies items that are both paid and returned:

```
query itemsRequiringRefunds(s: Seller) {
  match i: Item where i.order.seller = s
    such that there exists p: Payment where p.invoice.order = i.order
      such that not exists rf: Refund
        where rf.payment = p and rf.items = i
    and there exists r: Return where r.item = i
}
```

A service runs this query to determine which items need to be refunded. If it returns any items, then it processes those refunds. After doing so, adding a Refund fact removes the item from the preceding query. The service did not need to determine the state of items and their orders to find the entities that required compensating transactions. It ran that query directly against the history and then acted upon the results.

In most domains, I have found that querying history directly produced exactly the information I needed without determining state. To determine whether a button should be enabled, just query that its action does not yet exist. To distribute a queue to worker nodes, query for items not yet worked. Historical queries are more direct and less error prone than managing state machines. And it is not just for analytical reasons; they have a significant technical advantage as well.

Single Source of Truth

We've identified an alternative to state machine–based solutions for business problems. First, we modeled transitions not as changes to the mutable state of an entity, but instead as immutable records. And then, we wrote declarative functions that query the existence and absence of such records. These functions answer two questions: what is the next action for a given entity and what entities can accept a given action. Answering these questions directly from history turned out to be simpler than keeping track of states. As an additional benefit, we will find that it also solves some of technical problems that arise in distributed systems.

In order to process requests, a state machine needs to do two things. First, it needs to know with certainty the state of an entity. And second, it needs to determine whether the request is valid when the entity is in that state. As a consequence, clients have less autonomy. They must rely upon a privileged set of nodes to process requests on their behalf. Clients must consult the single source of truth to know what has happened as a result of their actions.

Orchestrators

Many state machine–based distributed systems employ message-driven architectures. In such a system, each step in the process is associated with a *command* message. A node called an *orchestrator* receives each message and executes the appropriate transaction. The orchestrator loads the current state from the entity identified by the *correlation ID* of the command. It determines which transaction to apply based on the state of the entity. The transaction modifies the entity, and the state machine determines the next state.

Consistent State

When we studied the CAP Theorem in Chapter 4, we defined consistency as the property that two nodes in a distributed system will report an entity as being in the same state. For an orchestrator to know the state of an entity, it must achieve consistency with other orchestrators in the system. This cannot be relaxed. Eventual consistency is not sufficient, because the state machine must process the command and provide a definitive result. To achieve consistency, orchestrators will often share a common database.

The orchestrator must be able to obtain a lock against an entity in order to determine how to process a command. For this reason, it is not uncommon to send all commands to a central set of orchestrators. This body of orchestrators and the database in which they maintain entity state becomes the single source of truth. It is the sole authority on the state of those entities.

Central Validation

A state machine determines which operations are valid based on the state of an entity. Payment can only be applied if the order is in the `Invoiced` state. A bug can only be failed if it is in the `In Test` state. In a distributed system, the users who initiate these operations are not co-located with the state machines. They are using clients.

A client issues commands for an orchestrator to process. Because the orchestrators must be consistent with one another, the CAP Theorem tells us that they must become unavailable in the face of a network partition. To overcome this, commands are often queued. An orchestrator will process the command at some later time, once the network partition has been healed.

Since the outcome of the command depends upon the state of the entity, the client cannot predict exactly what will happen. Clients lie outside of the single source of truth. They must wait for the command to reach the orchestrator and then for the result to make its way back to the client. And by the time the results make it back, the user has moved on.

By relying upon a consistent state to validate operations, the state machine–based solution has sacrificed autonomy. A client cannot predict the outcome of a request on its own. It cannot determine whether the request will succeed or fail, as other requests of which it is unaware may have moved the entity into a different state. Instead, clients must rely upon a single source of truth, push commands to a queue, and await the results.

Convergent Histories

Instead of relying upon a centralized static model as the single source of truth, we can allow each client to represent its own truth. The truth is whatever the user has done, it is not the responsibility of an orchestrator to validate the user's actions. It is the responsibility of the system to understand what has happened and to combine those histories into a cohesive story.

Define Immutable Records

To begin, we model each user action as an immutable record. This record captures the information that was produced at that time. Each record refers to previous actions as predecessors. These aren't simply a list of all events that occurred in the past; these previous actions are specifically the ones that lead to the new action. The predecessors represent the information that the user had while making this decision.

For example, a `Shipment` fact captures the tracking number. This information is produced when the user takes action. Furthermore, a `Payment` refers to a predecessor `Invoice`, but is not related to any `Shipment`s that might have already happened. The invoice is causally related to the payment, but the shipment is not.

Query for the Next Action

Once the immutable records are modeled, write a query for each kind of action. Query for all of the records that would be predecessors of that action. Add a clause including only those for which the action does not exist. This query tells you if this is the next action for an entity.

For example, for a `Commit`, query for `Begin Review` without a subsequent `Reject` or `Accept`. If that query returns one or more records, then the commit is still in review. Accept or reject is the next action. Furthermore, the list of records returned by the query are predecessors of the `Accept` or `Reject` records that will be created.

Write another query that begins higher up the chain. Instead of starting at a `Commit`, start with the `Sprint`. This query will list all entities for which a particular next action is required. This provides a set of work items to present to the user.

Capture Actions Locally

When the user takes an action, add the record to the local history immediately. Creating the record will influence the results of the queries. The user will immediately see the entity removed from one set of work items and added to the next. They will observe that the new next action of the entity has advanced. This gives them immediate feedback that their actions have been honored.

The user is making their decision based on the information that they have available. The system should not pause to check the current state, or take a remote lock to ensure that the user's action is consistent. It should not push the record to a queue and wait for an orchestrator to process it later. Commit the action to history at the client, and let parallel histories converge.

Define Compensating Actions

Finally, identify undesirable states that might arise due to convergent histories. Such states can arise because we've given clients the autonomy to act based on the information that they currently have available. Write a query to find entities in undesirable states, and then design processes to perform compensating transactions.

It's possible that an order will have both a `Refund` and an unreturned `Shipment`. Create a query looking for orders in this state. Present the results to a representative, prompting them to call the customer asking them to either return the product or pay another invoice. It is up to the analyst to identify appropriate compensations for such situations and not the responsibility of a state machine to try to prevent them.

Users are the source of truth. They are distributed across the system. They do not participate in a consistent, orchestrated state machine. They do not care about the current state of an entity. They just need to know what has happened and how to interpret those parallel histories. They need the assurance that their actions will be honored and will become part of that convergent history.

CHAPTER 7

Security

A common approach to application security is role-based access control (RBAC). Under this system, an administrator assigns individuals to roles and then authorizes those roles to perform certain actions within the system. As we adopt immutable architectures, RBAC becomes more challenging. Requiring an administrator to assign roles and permissions reduces the autonomy of individual users. Consulting a single source of truth for those roles and permissions reduces the autonomy of client nodes. The access control model begins to work against the advantages that we fought so hard to achieve.

RBAC is typically applied at the organization level. A team of administrators define the roles and operations within an organization. They manage a set of resources, upon which the operations are performed. That organization is the single beneficiary of the system. In a multi-tenant environment, however, the division of roles and responsibilities becomes much more complex. The myriad of tenants may not agree on a single body of administrators to manage access to their resources. Instead, they will seek to maintain autonomy over their own assets. This will lead them away from a centralized form of access control and toward more distributed trust models.

Our desire for autonomy drives us toward a decentralized model of access control. The expansion of distributed systems across multiple tenants removes the organizational structures that we might have otherwise depended upon. And so we look elsewhere for a solution. Instead of a role-based access model, we find inspiration in public key infrastructure (PKI) and delegation of authority. With these tools, we can build a system of security on top of a model of immutable facts.

Proof of Authorship

A historical record represents a decision that a person made within a distributed system. Before we can determine whether to trust that particular decision, we have to have some assurance of the identity of the person who made it. We seek proof of the authorship of a fact. Modern digital systems rely upon *public key infrastructure* (PKI) to provide that proof. PKI is based on the existence of a trapdoor function: a mathematical function that is easy to compute in one direction, but difficult to invert.

Key Pairs

Suppose that I have a pair of functions. Each function is the inverse of the other; if the outcome of one function is fed into the other, the original input will emerge. For example, the functions *x+3* and *x-3* are such a pair. Adding three and then subtracting three gives you back the original value. You could think of many more examples and probably come up with several different ways of generating new pairs.

If I gave you one of these functions—say *x+3*—you would probably be able to tell me what its inverse is. Computing the inverse of such a simple function is not difficult. But what if I gave you a function like x^{37} *mod 1829*? It might take you a bit to work out what the inverse is. A computer given the right algorithm could find it quickly enough: the inverse is x^{823} *mod 1829*, as demonstrated in Figure 7-1. But if I make these numbers significantly larger, then even the most powerful digital computer will have a hard time finding the inverse.

$$36 \rightarrow x^{37} \bmod 1829 \rightarrow 842 \rightarrow x^{823} \bmod 1829 \rightarrow 36$$

$$42 \rightarrow x^{823} \bmod 1829 \rightarrow 858 \rightarrow x^{37} \bmod 1829 \rightarrow 42$$

Figure 7-1. *Some functions that are inverses of one another can be used in asymmetric cryptography*

Functions of this form are examples of trapdoor functions. These were generated with a protocol known as RSA, named after its inventors Rivest, Shamir, and Adleman.[1] It is really easy for you to compute the modular exponent. It is really difficult for you to compute its inverse (known as the discrete logarithm). It's only because I generated this pair at the same time that I'm able to find the inverse myself.

If you look closely at these functions, you can see that they have an upper bound. The function $x^{37} \bmod 1829$ only has an inverse if $x < 1829$. This is a consequence of the *pigeonhole principle*. The modulus 1829 limits the number of possible results the function can return; it determines how many pigeon holes you have. If you try to put more pigeons into the function, there will be at least two sharing the same pigeon hole. This function is deliberately constructed to ensure that every input has a distinct output, which is necessary to make sure it has an inverse within the same domain.

RSA gives us a protocol to generate a function and its inverse at the same time. If I give one of the functions to you, you will have a hard time finding the inverse; it's a trapdoor function. We'll call the one that I share the *public key*. The function that I keep is the *private key*. I can use this pair of inverse functions to prove authorship.

Digest

If you run a hash function over a stream of data, you'll produce a digest. The digest will have a fixed size, determined by the hash function you choose. It's important to choose a hash much smaller than the limit of public/private key pair. Pad this digest with some random data and feed it into the private key function. The result is the signature.

If someone knew your public key, they could verify your signature. All they would need to do is run your signature through your public key function to get back your padded digest, as shown in Figure 7-2. They could then compute the digest of the message themselves and see if it matches with yours. If so, then they have confidence that the message came from you. This works because it would be very difficult to invert your public key to find your private key. Without your private key, they could not produce a signature that would contain the correct digest.

[1]R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM. February 1978.

Figure 7-2. Running the digest of a document through the private key produces a digital signature

If you wanted consumers of an immutable record to know that it came from you, you could produce a digest. The procedure must be repeatable to ensure that everyone produces the same digest. We must therefore choose a *canonical form* for immutable records. One procedure that I like to use is as follows:

- Serialize the record in JSON.

- Include the record's type as a field.

- Replace predecessors with objects having only a ref field, the value of which is the digest of the predecessor.

- Sort collections of predecessors by their digest and eliminate duplicates.

- Sort fields alphabetically.

- Remove extraneous whitespace.

- Encode the text using UTF-8.

- Compute the SHA-256 hash of the stream.

As an example, a forum post might be serialized as follows before whitespace is removed:

```
{
  "author": {
    "ref": "MSHFIBOX5JkupOYu7ZZuIKJHVtow3vtAK/7f4GYmKVqdcKMcVg9AURmgU9RQA
    tJwQjaYguJSJZzlwFctOTqrCw=="
  },
```

```
"forum": {
  "ref": "PQu2HVVqExAOr1kO9lK+rWHzui5YsdO7+g5VkgnNRsJqPnpsy5rzjSfIpnd79
  aea8jjoPe+YIiouOz3xcJvQUQ=="
},
"text": "Posted my first forum message.",
"type": "ForumPost"
}
```

After removing whitespace, applying UTF-8 encoding, and taking SHA-256 hash, you can produce a signature using your private key. Any consumer of this record would compute the same digest. If they had your public key, they could verify your signature. Insofar as only someone with your private key could produce a valid signature, they can be assured that the message was from you.

The immutable record itself does not contain a signature. It could not, because the signature is produced from the digest of the contents of the record. Instead, the signature is carried in an envelope. As one node communicates a collection of immutable records to another, it provides with each a list of public keys and signatures.

Authorization

Having proved that you have authored a fact, you might next seek to establish that you are permitted to do so. How might you assert your claim of authorization in a distributed system? If we could appeal to some central authority, then perhaps it could validate your claim. But to retain autonomy, we would like to avoid such an authority. We must therefore establish a means of authorization that the recipient can verify on their own.

To begin, we will define a means of exchanging public keys. When a node interacts with an agent, it should be able to verify claims of authorship. Then, the recipient will consult a set of rules to determine which agents are authorized to produce which records. These rules are based entirely upon related records.

Principal Facts

The easiest way to exchange public keys is to make them part of the historical model itself. Every agent that is capable of producing facts is a *principal*. Users are principals; autonomous services are principals; authorized entities are principals. Every principal creates a historical fact that represents themselves.

The principal fact contains as a field the user's public key. It typically contains no additional fields. Because the identity of the immutable record is derived from its contents, the identity of the principal *is* its public key. For example, a user of forum software would be represented by the fact in Figure 7-3.

User
_____
publicKey: MIIEogIBAAKCAQEAzPhOXAP...

Figure 7-3. *A principal fact has a public key*

A principal fact needs no signature. A signature would not prove anything useful. The principal fact contains no additional claims, so the signer would only be reasserting that this is their public key.

Authorization Query

When a node receives a fact, the new fact is considered *contested*. The permission of the author to create that fact remains in doubt. The node must verify that the author has authority to create the contested fact. To do so, it will run an authorization query. This query determines which principal facts are authorized to create the contested fact. If the envelope of that fact contains a signature from an authorized principal, then the fact is *permitted*. Only permitted facts are stored, used in subsequent queries, and forwarded to other nodes.
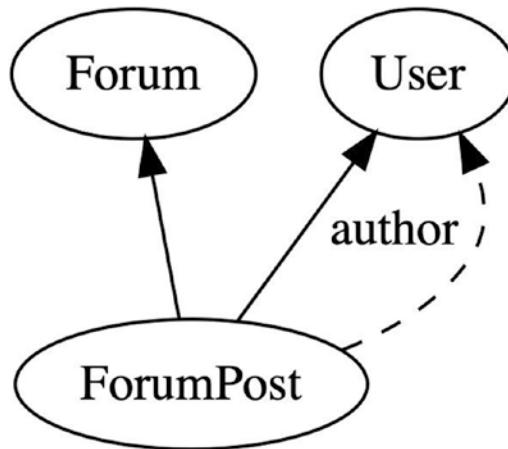
For example, the forum post that we recently observed is an example of the following kind of fact:

```
fact ForumPost {
  forum: Forum
  author: User
  text: string
}
```

To verify that the post is signed by the author, each recipient will run the following authorization rule:

```
authorize p: ForumPost {
  match u: User where p.author = u
}
```

Overlaid on top of the fact graph, the authorization query appears in Figure 7-4.

Figure 7-4. *A forum post with the authorization query that determines which user can create it*

This rule instructs the recipient to find the principal fact that is the author of the post. It uses the publicKey from that principal fact. If the envelope contains a valid signature from that public key, then the post is authorized. If not, then the recipient immediately rejects the fact.

When used as described previously, authorization rules can prevent others from forging messages. If your public key becomes well-known, then a would-be forger might attempt to create a new forum post with your principal as the author predecessor. However, without being in possession of your private key, they would be unable to create a matching signature. The forger would have to settle for generating a new public/private key pair. Such messages would be authorized, but their author would be a different principal. No recipient would be fooled into believing that the forum post was from you.

Initial Authorization

While the preceding authorization rule prevents forgery, it still permits any principal to post to the forum. In other domains, it is desirable to constrain behaviors to a restricted set of principals. Without relying upon a centralized authority to administer these restrictions, we will turn to the model itself to seed the initial authorization.

Suppose that instead of an open forum, we want to model a personal blog. Only the creator of the blog is permitted to post. The blog can be described with the following fact:

```
fact Blog {
  creator: User
  identifier: guid
}
```

An authorization rule verifies that the blog is indeed authored by its creator:

```
authorize b: Blog {
  match u: User where b.creator = u
}
```

Now we can define a blog post with the following fact:
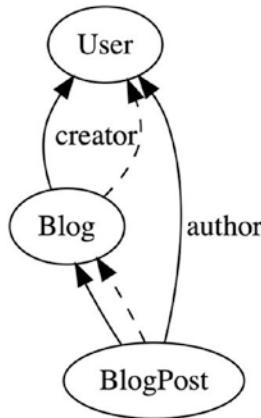
```
fact BlogPost {
  blog: Blog
  author: User
  postedAt: datetime
}
```

For this domain, we are leaving mutable properties such as the title, text, and tags as separate facts. Only the date and time distinguish one blog post from others by the same author in the same blog.

But the most important thing is that now we can write an authorization rule that allows only the creator of the blog to post to it.

```
authorize p: BlogPost {
  match u: User
    where p.author = u
    and p.blog.creator = u
}
```

The authorization rule for a blog post traverses the fact graph as shown in Figure 7-5.

Figure 7-5. *A blog post can be authored only by the blog's creator*

This query will only return the author principal if they are also the blog's creator. If someone tries to create a fact claiming to be from an author other than the creator of the blog, then this query will return no results. Seeing no principals authorized to sign the fact, the recipient will reject it outright. Creating a blog provides the initial authorization list, without the need for any party to consult with a centralized administrator.

Grant of Authority

The authorization queries that we've written thus far return only the creators of the root entity. This is an ideal starting point for a decentralized system; if you made it, you alone control it. It cleanly dispenses with the need for a body of administrators to define roles and permissions, or to grant access to certain resources.

But this is only a starting point. For most domains, authorization cannot remain solely with the creator of an entity. The creator must be able to transfer authority to another party. Such transfers can either be limited or indefinite. If a transfer is limited, it is constrained in scope to one single occurrence. But if it is indefinite, then authorization persists.

221

Limited Authority

A creator can grant another principal authorization for a single instance of an entity. To do so, the creator identifies the authorized principal as a predecessor to the new entity. An authorization rule grants this principal permission to create successors.

This is best seen with an example. Suppose that the creator of a blog would like to invite a guest to post on the site. They create the guest post for the selected user.

```
fact GuestPost {
  blog: Blog
  guest: User
  createdAt: datetime
}
```

Only the blog creator can add a guest post. Clients use the following authorization rule to enforce this constraint:

```
authorize gp: GuestPost {
  match u: User where gp.blog.creator = u
}
```
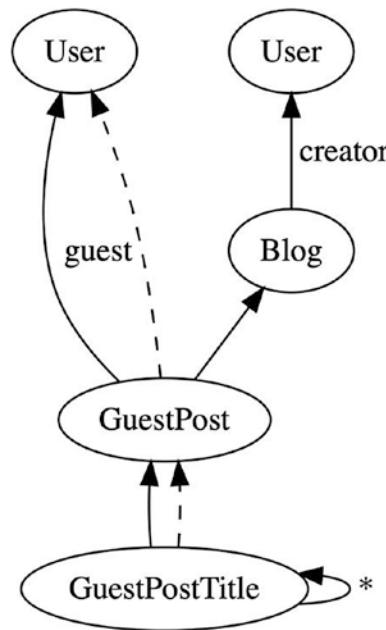
Guests are then authorized to set the title, write the text, and other subsequent operations. For example, to set the title, the guest issues the following kind of fact:

```
fact GuestPostTitle {
  post: GuestPost
  title: string
  prior: GuestPostTitle[]
}
```

The following rule authorizes the guest to issue these successors:

```
authorize t: GuestPostTitle {
  match u: User where t.post.guest = u
}
```

The diagram of the authorization rule in this graph appears in Figure 7-6.

Figure 7-6. *Only the guest can set the title of the guest post*

The guest does not have authorization to create additional posts. Their permission is limited to the single child that has been created on their behalf.

Indefinite Authorization

When the creator of an entity wishes to share authority indefinitely with others, they can create a fact documenting that decision. The grant must be signed by the initial creator. It identifies the entity for which authorization is granted. And it names the principal with which authority will be shared. Then, recipients must be instructed to honor additional authorization rules.

Suppose, for example, that the creator of a blog wishes to invite others to post as well. They don't just want to give them a fixed number of guest posts. They want to share authorization indefinitely. To do so, the blog creator issues a grant.

```
fact BlogGrant {
  blog: Blog
  subject: User
  createdAt: datetime
}
```
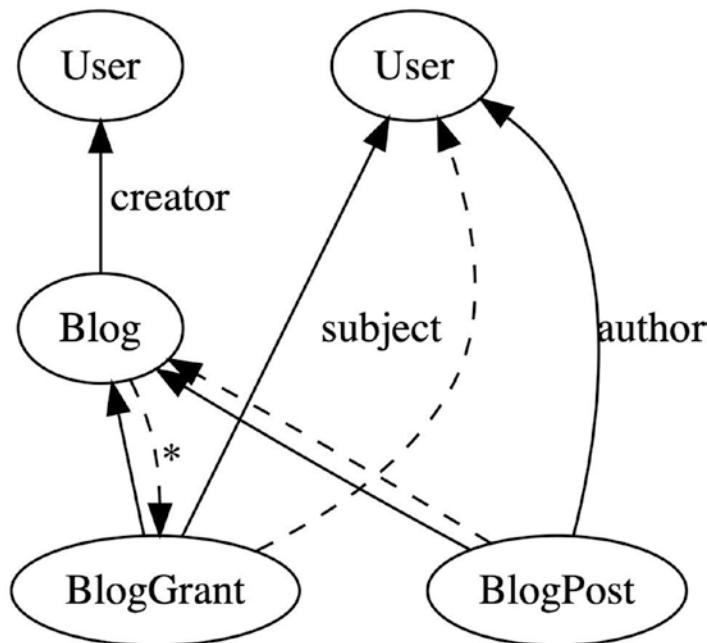
Only the blog creator can issue such a grant. To enforce this, we write an authorization rule requiring that the grant come from the blog's creator:

```
authorize bg: BlogGrant {
  match u: User where bg.blog.creator = u
}
```

Once the grant is issued, the subject should be authorized to post. We write a rule stating such.

```
authorize p: BlogPost {
  match bg: BlogGrant
    where bg.blog = p.blog
  then u: User
    where p.author = u
    and bg.subject = u
}
```

This authorization rule zigzags through the fact graph as shown in Figure 7-7.

Figure 7-7. Every subject of a grant is authorized to create a new blog post

It's important to recognize that this authorization rule is *in addition to* the previous authorization rule for BlogPost. A client honoring both of these rules will allow the creator to post and allow all subjects granted authority to post. A client does not have just one authorization rule per type. Instead, the set union of all authorization rules defines the set of principals that will be validated. If the envelope contains a signature from any member of that set, then the fact is authorized.

Transitive Authorization

The authorization rule mentioned earlier permits the subject of a BlogGrant to post on another's blog. It does not, however, put them on equal footing with the blog's creator. It does not permit them to then extend that authorization to others by issuing further grants. If doing so would be a desirable feature of the domain, then clients will need to be given one more rule.

```
authorize next: BlogGrant {
  match bg: BlogGrant
    where bg.blog = next.blog
```

```
  then u: User
    where bg.subject = u
}
```

Just as with all authorization rules, this one is combined with others for the same contested fact. A different rule authorizes the blog's creator to issue grants. This one adds to it—via set union—authority for subjects to extend those grants to others. The recursive nature of this definition implies that grants can be extended to any number of generations.

Revocation

All of the approvals and grants documented earlier include a creation date. This is not mere audit detail. This is a design decision that allows for revocation. A one-time approval or an indefinite grant can be revoked by a subsequent fact. Authorization rules simply need to include a such that not exists clause to make it so.

If a blog creator wishes to revoke a prior grant, then they can issue a fact of the following form:

```
fact BlogGrantRevoke {
  grant: BlogGrant
}
```

The creator is authorized to issue these revocations for their own blog. The authorization rule enforcing that is as follows:

```
authorize r: BlogGrantRevoke {
  match u: User where r.grant.blog.creator = u
}
```

If other grantees are similarly empowered, then the following rule is added:
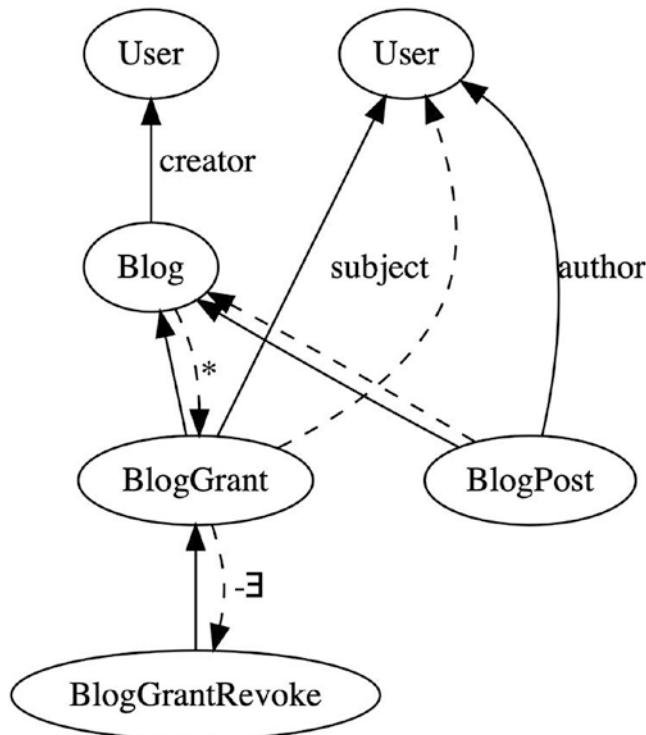
```
authorize r: BlogGrantRevoke {
  match bg: BlogGrant
    where bg.blog = r.grant.blog
  then u: User
    where bg.subject = u
}
```

We can now add a such that not exists clause to the rule so that this fact revokes a previous authorization.

```
authorize p: BlogPost {
  match bg: BlogGrant where bg.blog = p.blog
    such that not exists r: BlogGrantRevoke where r.grant = bg
  then u: User
    where p.author = u
    and bg.subject = u
}
```

Adding the such that not exists clause to the authorization rule results in the diagram shown in Figure 7-8.

Figure 7-8. *The authorization rule checks for the existence of a blog grant revoke*

And this is where it becomes important to add a distinguishing field such as `createdAt` to the grant. Without it, authorization could only be granted to a subject once. Once revoked, there would be no way to create a new grant. It would be indistinguishable from the revoked grant.

Authorization Upon Receipt

Authorization rules are evaluated immediately upon receipt of a fact. They are not tested later to determine retroactively whether a fact should have been authorized. There are two important reasons for this: preservation and performance.

If authorization rules were evaluated retroactively, then revoking access would invalidate all prior actions that the party had performed. In most domains, this is not the desired outcome. An employee who is terminated should have their access revoked. Yet all of the work that they did for the company up to that point should be preserved. If the authorization rules were run for those work items after their termination, then those facts would be invalid.

The other consideration is performance. Retroactive evaluation is a recursive, time-consuming operation. If we wanted to use the authorization rules to validate facts every time they are used, we would have to apply them to all facts touched in a query. But to validate a fact, we must run its authorization rules. Those rules are themselves queries. Tracing these queries back to the facts that they touch, their authorization rules, and the queries for those rules leads to an explosion of validation up the historical graph. This process is not guaranteed to terminate, and even when it does, it is incredibly time consuming.

While evaluation upon receipt addresses these two important issues, it also causes a problem with respect to eventual consistency. One node may determine that a fact is valid, while another decides that it is invalid. This happens when one has received a revocation fact and the other has not. There is no causal relationship between the revocation and the contested fact. Neither is the predecessor of the other, so either one may occur first. Once this happens, no further messaging will bring these two nodes back into agreement.

Given the danger of inconsistency, revocation should be used with extreme caution. Provide the capability in the models that you design, but caution users to exercise that capability only sparingly. Build in some other mechanism to revoke privileges across the system. For example, control access to the private key for employees. When they are

terminated, simply destroy the private key, rather than revoking grants of authorization. Or build a clock into the model. Periodically renew grants during the next period. If a principal leaves the group, simply neglect to renew their grant, rather than revoking past grants.

Confidentiality

The security considerations we've talked about so far have dealt with the ability to write data into the system. We now turn our attention to the ability to read data from the system. As with writes, we want to control reads without yielding to a centralized group of administrators. We will be conducting business over an evolving network topology that may include partner nodes outside of our direct control. And so we once again turn to PKI for inspiration on implementing trust without a central trusted authority.

The desire for autonomy is not one-sided. On the one hand, we want personal control for ourselves and our devices. We want to be assured that we can act without the need to connect to a central system of record. When working with partners in a distributed system, we should expect that they will want the same autonomy for themselves. We allowed them that autonomy by applying a *trust but verify* policy: we validated records for ourselves against an agreed-upon set of authorization rules. But granting them autonomy also gives them some degree of access to our sensitive information. We must now consider how to keep messages private in such an environment.

Untrusted Nodes

Confidentiality, as I'm using it here, means having reasonable assurance that the information that you wish to convey to a specific party will not be intercepted by others. The challenge in a distributed system is that the intended recipient might not be in direct communication. Messages might need to be stored on a third-party node that has greater uptime and accessibility than either of the parties' devices. Mail is stored on mail servers, not transmitted directly from sender to recipient. Direct messages are posted to shared channels. Records intended for one party can be found on untrusted intermediaries.

If we assume that we can trust our third-party providers implicitly, then we might be satisfied with simply encrypting data in transit. We might use a secure protocol like TLS to upload a private message to a social media provider, knowing full well that it will be decrypted on the server. It might even be stored in plaintext, to be reencrypted when the recipient initiates their own TLS connection to the same server. But if we do *not* want to assume that we can trust our intermediaries implicitly, then transport-layer encryption alone is not sufficient.

Asymmetric Encryption

As we saw in **Proof of Authorship**, public and private keys are nothing more than inverse functions. Put a number into one, and it produces an answer; put that answer into the other, and you get back the original number. We proved authorship by applying the private key first. We can send private messages by inverting the process.

If we wanted to send a number to a recipient, we can run it through their public key. The result can be stored on an untrusted node with little fear. Any party not in possession of the private key will have a very hard time inverting the function to find the original value. This is the basis for achieving confidentiality in a distributed system with third-party nodes. But before we can apply this protocol, we have to contend with the size limit.
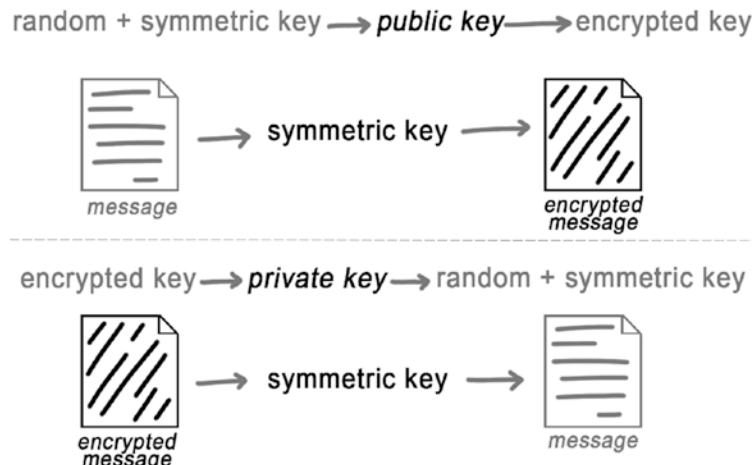
Asymmetric Size Limit

If you recall, the functions that we produced using the RSA protocol only had inverses within a certain range. The function $x^{37} \bmod 1829$—which we used as our public key—can only produce outputs in the domain *0-1828*. The pigeonhole principle prevented us from accepting any inputs that would cause us to double up on an output. And so this public key can only be used with 1829 distinct inputs; it only supports 10-bit messages. Real RSA key pairs are much larger: 2048 or 4096 bits in common usage. Nevertheless, there is a size limit. This was why we signed the *hash* and not the original message. And for the same reason, we cannot encrypt the original message using the recipient's public key.

Instead we will apply a *symmetric cypher* to the message. Symmetric keys don't have the same size limitation as asymmetric key pairs. They can be used to encrypt a message of arbitrary length. However, the same key is used both to encrypt and to decrypt the message. We must therefore keep the symmetric key private. That is what we pass through the recipient's public key.

Encrypt the Symmetric Key

And so the strategy that we use to ensure confidentiality in a distributed system is to encrypt the contents of immutable records before transmission. The sender first generates a random number to be used as the symmetric key. Effective symmetric keys can be significantly smaller than the size limit of a similarly effective public key. We can therefore pad and encrypt this symmetric key using the recipient's public key and store the result in the beginning of the immutable record. We can then encrypt the contents of the message using the symmetric key and store that encrypted blob at the tail of the record. The recipient can reverse the process, using their private key to reveal the symmetric key and extract it from the random padding. It can then decrypt the tail of the record to uncover the body of the message, as shown in Figure 7-9.

Figure 7-9. *The sender encrypts the symmetric key using the recipient's public key*

Third parties in the middle of this exchange will not be able to easily decrypt the record without the recipient's private key. We've succeeded in ensuring confidentiality when untrusted nodes lie between the participants. More than encrypting during transport—which exposes message contents to intermediaries—we've encrypted *prior to* transport. Even if the third-party node takes no additional precautions, we have protected the message at rest.

Encrypting Historical Facts

The protocol described previously is a popular mechanism for exchanging private messages over untrusted media. It is the core procedure in the OpenPGP protocol[2] used for private email exchange. It is commonly applied to messages stored on distributed file systems like IPFS, the InterPlanetary File System. In those and other scenarios, the protocol has a limit: it protects only the contents of the message, not the metadata surrounding the message. While an interceptor won't be able to read the contents of a OpenPGP email, they would be able to identify both the sender and the recipient. This is simply the consequence of the third party needing to know how to route messages. When this protocol is used within a historical model, the same limitation applies.

In a historical model, only the body of a confidential fact is encrypted. The type and—more importantly—the predecessors of that fact are not. The reason is that third-party nodes must be able to execute queries and return private facts. If the predecessors were encrypted, then the third-party node would not be able to determine which facts were the successors sought in the query.

For a similar reason, the identity of a confidential fact is based on the hash of its encrypted representation, not the original body. If other facts are subsequently recorded using the private fact as a predecessor, the intermediate parties need to understand that relationship. They need to be able to produce the hash of the record without decrypting it.

Limit the Distribution of Confidential Facts

Because the types and predecessor relationships are freely visible to intermediate parties, we still have to be careful with whom we share encrypted confidential messages. We can avoid shared distributed ledgers like blockchains that rely upon public scrutiny of metadata in order to function. We can limit the peers with which we directly share encrypted facts to only those that we can trust just enough not to infer meaning from the predecessor/successor relationships. We can even spread our trust among several intermediates so that no one of them has a complete picture of our interactions with other parties.

To prevent confidential facts from spreading further than they need to, these intermediaries must be given a set of rules by which they are permitted to release this information. Unlike the authorization rules previously discussed, we cannot execute

---

[2]J. Callas, et al, IETF Network Working Group Request for Comments 4880. November 2007.

these rules ourselves. We have to trust that the intermediaries that we choose to work with are enforcing them with each request. The rules tell them who is permitted to receive certain facts.

Distribution Rules

Whereas an authorization rule told us who was permitted to issue a fact, a distribution rule tells who is permitted to execute a query. We begin by defining the queries permitted. These can be expressed using the Factual query language, as we've already shown throughout the book. Give every server node a list of allowed Factual queries. Any client issuing a query not on this list is immediately rejected.

The next step is to identify the permitted starting points for those queries. Each Factual query has one starting point. It typically matches successors of that starting fact and then perhaps zigzags through the graph from there. The distribution rule expresses the set of principals who are permitted to start a query from that point. And just like with an authorization rule, this is done using a query.

Suppose that we have written the following query for private messages sent to an individual:

```
query privateMessagesToRecipient(r: User) {
  match m: PrivateMessage where m.recipient = r
}
```

A sufficiently restrictive server node will not accept any queries until they have been provisioned to do so. It will not accept any query for `PrivateMessage` facts until it is given the preceding query and the following distribution rule:

```
distribute privateMessagesToRecipient for r: User {
  match r
}
```

This distribution rule is as simple as possible, but it demonstrates the important points. It identifies a query, in this case the one defined just earlier. It starts with the same fact as the query that it controls. From that starting point, it matches the principals who are permitted to execute the query. This particular distribution rule allows only the recipient themselves to query for private messages.

Each query has a distinct canonical form. The server can use this as well as the type of the starting fact to look up the associated distribution rules. If the server node

receives a query that does not match one that it has been provisioned to distribute, it will reject the query. If the client cannot prove that they are acting on behalf of at least one principle matching the distribution rule, then the server will reject the query. If these two checks pass, then the server executes the query and returns the results. The server may not be able to interpret the contents of the resulting facts, but it has done what it can to limit distribution to only those that should be able to do so.

Evidence

Proving that a client is acting on behalf of a principal is not a trivial matter. How it is accomplished depends upon whether the query is coming from a client or another server. If the user is logged into the client, they will have an authorization token to share with the server. The token must have been issued by a *security token service* (STS) that both the client and the server trust. The client provided their credentials to the STS, and the STS signed the token. In this scenario, the server that the client is using directly will often be acting as a keystore as well. It will store the user's private key so that the user can log in from any device. This server must be completely under the user's control.

While client–server communication can be authorized by a token, server-to-server communication cannot. A server initiates a connection to another server without user intervention. There is no user to supply credentials to an STS and generate a token. Furthermore, the two servers may not have an STS in common that they both trust. A security token would not provide satisfactory evidence that the query is on behalf of a given user.

To provide satisfactory evidence, the querying server must use the private key of the principal initiating the query. Given that this server is also likely to be the keystore, this process can be done without user intervention. The protocol begins with the querying server invoking a query from a certain starting point and identifying the principal's public key. If the target server determines that the identified principal is permitted to execute the query, it responds with a randomly generated challenge. The querying server answers the challenge in a way that proves that it is in possession of the private key. The trick is that it must not simply execute the private key and give back the answer. To do so would be to allow the server to use a message digest as a challenge, and thus generate a valid signature, or to execute a man-in-the-middle attack and produce a valid challenge response. Instead, a *zero-knowledge proof* protocol must be employed. Such a protocol proves that the querying server has the private key, but does not allow the target server

to gain any knowledge about it. David Chaum and his colleagues[3] give several protocols for proving that the querying server knows the discrete logarithm (inverse of an RSA key) of a given value without revealing that value.

Attacks and Countermeasures

None of these precautions, however, protect us from third parties that maliciously, negligently, or through legal compulsion share our information with others. For that we may need to take additional precautions, based on the sensitivity of the metadata. Additional measures include the following:

- Periodically changing key pairs to mask a party's identity.

- Mix one-time tokens with random users in key exchanges.

- Generate a new key pair—and thus a new principal fact—for each interaction.

- Store identity mapping tables offline, or on a privately managed server.

- Generate a hum of meaningless facts between unrelated predecessors to hide the signal in the noise.

Such countermeasures are not necessary for every domain. But when the relationships among facts are valuable, and intermediate parties cannot be fully trusted, then additional care is warranted.

One more attack vector must be considered when storing encrypted facts on untrusted intermediate nodes. Data at rest is susceptible to offline attacks. Online services will throttle or block failed attempts to access data to prevent brute-force attacks. But when the attacker has the encrypted data in their possession, they can run as many attempts as their computing power will allow. Such an attack against modern cryptography algorithms will be expensive. But a determined attacker with a valuable enough payload might be willing and—eventually—able to discover the symmetric key.

There are several things that we could do to make their job easier. For example, the more that a symmetric key is reused, the more samples an attacker has to work with.

---

[3]Chaum, David; Evertse, Jan-Hendrik; van de Graaf, Jeroen (1987). An Improved Protocol for Demonstrating Possession of Discrete Logarithms and Some Generalizations. Advances in Cryptology – EuroCrypt '87: Proceedings. Lecture Notes in Computer Science.

And the longer the encrypted message, the more likely they are to discover an exploitable pattern. And so, to protect information from offline attacks, do exactly the opposite:

- Use a strong random number generator to produce symmetric keys.

- Use each symmetric key only once.

- Keep messages as short as possible.

- Pad messages with a large amount of random data.

- Generate additional meaningless facts to hide the valuable ones.

Even with all of these precautions in place, you must assume that the crypto systems that we use today will eventually become obsolete. This has been found true of the algorithms of the past, and we have no reason to believe that the future won't reveal weaknesses in today's technology. In fact, it's reasonable to believe that quantum computing will someday render all of today's asymmetric algorithms ineffective. Perhaps all that a determined attacker needs to do is wait.

My only advice to mitigate this problem is to ensure that the value of your information degrades over time. If you are exchanging payment information, make sure that those instruments expire before the encryption is broken. If you are collaborating on a secret plan, make sure that you execute before it's too late. If you have something that you want to keep secret forever, don't encrypt it and give it to a third party.

Secrecy

Employing PKI, distribution rules, and countermeasures, we have gained a level of confidentiality. We can now send a message to a specific recipient through untrusted nodes and have some assurance that the contents will remain private for some time. This form of confidentiality works well when communicating with one person. Many of our collaborations, however, involve groups. We will now generalize these techniques so that the group can communicate secretly.

We'll begin by defining a shared *workspace* in which the collaborators interact. This might take the form of a project containing work items, plans, and resources. Or it might be a department such as accounting, where payroll, accounts payable, and assets are all stored. Our goal is to control access within this workspace. We've already demonstrated the use of authorization rules to grant authority to create new facts in

a workspace. Now we extend our strategy of confidentiality to keep those facts secret, visible only to those invited to the workspace.

Shared Symmetric Key
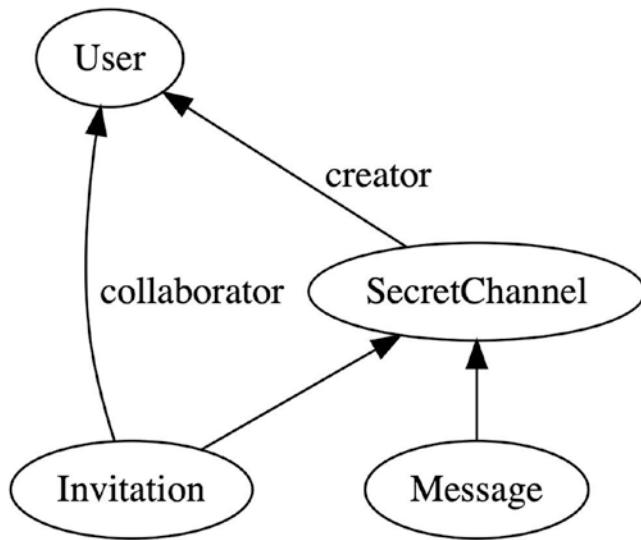
To keep the data within the workspace secret, we will create a shared symmetric key. Anyone in possession of this key will be able to decrypt the contents of the facts within the workspace. We must therefore have a protocol for exchanging that symmetric key with other members of the workspace within the historical model. We must do this without revealing it to untrusted third parties that store and forward the facts.

A workspace in a historical model takes the form of a fact. It has only one predecessor, the creator, and only one field, its identity. When a user initially creates the workspace, they generate a random symmetric key. This symmetric key is not stored in the workspace fact. Instead, it is shared with each participant individually.

To share the symmetric key with other participants, the creator sends invitations. An invitation is a successor to the workspace fact naming the recipient as a predecessor. Its only field is the shared symmetric key. The workspace creator sends the first invitation to themselves. The symmetric key is packed in random padding and encrypted using their own public key. Now the shared key is stored in the model in such a way that only the creator can decrypt it. They can send invitations to other collaborators immediately, or as they join the team.

A Secret Discussion Channel

To demonstrate this protocol, let's consider a secret channel to which collaborators can post messages. The channel itself is nothing more than a fact having an identifier. Invitations are successor facts sent to collaborators. The model appears in Figure 7-10.

Figure 7-10. *A secret channel to which collaborators are invited*

The model is expressed in Factual as follows:

```
fact User {
  publicKey: string
}

fact SecretChannel {
  creator: User
  identifier: guid
}

fact Invitation {
  secretChannel: SecretChannel
  collaborator: User
  sharedKey: string
}

fact Message {
  secretChannel: SecretChannel
  body: string
}
```
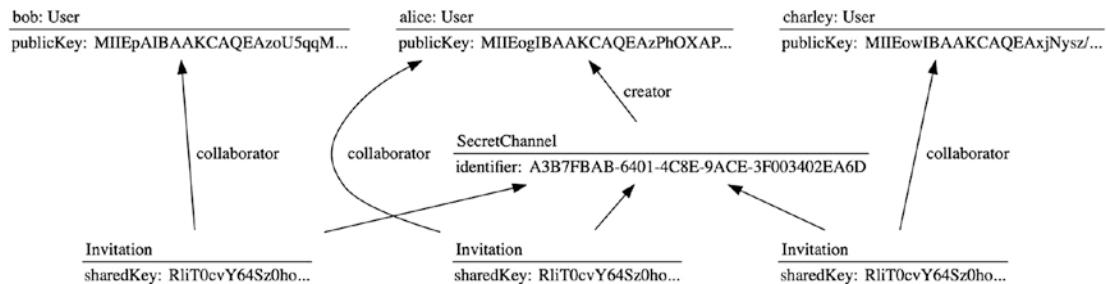
Creating a Secret Channel

Alice wants to set up a secret channel so that she can talk with Bob and Charley. She begins by generating the globally unique identifier for the channel as well as a random symmetric key. With these, she creates the SecretChannel fact and three Invitation facts. One of the invitations is to herself. The resulting facts are illustrated in Figure 7-11.

Figure 7-11. *Alice has created a private channel and invited both Bob and Charley*

Even though the preceding diagram shows the plaintext contents of the Invitation facts, each one is encrypted using the public key of the associated collaborator. Each collaborator can individually decrypt their own invitation. Doing so reveals a shared symmetric key.

Team Distribution Rules

Even though the contents of the messages are encrypted using the shared symmetric key, it is still wise to limit the distribution of those facts. Assuming that the intermediate third-party nodes that we choose to use are well behaved, they will follow the rules that we define for distribution. Here is the query for secret messages and the corresponding distribution rule:

```
query messagesInSecretChannel(sc: SecretChannel) {
  match m: Message where m.secretChannel = sc
}

distribute messagesInSecretChannel for sc: SecretChannel {
  match i: Invitation where i.secretChannel = sc
  then u: User where i.collaborator = u
}
```

This distribution rule permits the server to respond to queries from collaborators who have an invitation to that secret channel. Queries from other users will be denied, so that they will not be able to perform offline cryptanalysis on the messages to discover the shared symmetric key.

Compare the distribution rule to the corresponding authorization rule:

```
authorize m: Message {
  match i: Invitation where i.secretChannel = m.secretChannel
  then u: User where i.collaborator = u
}
```

This rule authorizes every collaborator to post messages within the secret channel. It uses the same `Invitation` as the distribution rule. However, this need not be the case. We could define a different type of invitation called a `ReadOnlyInvitation`. By defining a distribution rule based on these types of invitations—but not defining a similar authorization rule—we can give some users read-only access to the messages.

Limit the Scope of a Shared Key

Just as we want to limit the distribution of encrypted facts, we want to limit the number of facts that are encrypted using the same symmetric key. Reusing a symmetric key provides a would-be attacker with more examples of ciphertext to analyze. When sending a fact to a single recipient, we were able to generate a distinct symmetric key for each message (commonly called a *session key*). Unfortunately, that same mechanism will not work for workspaces with large numbers of collaborators. The session key would need to be encrypted with the public keys of each collaborator in turn. Some of those collaborators may not have even joined the team at the time that the encrypted fact was created.
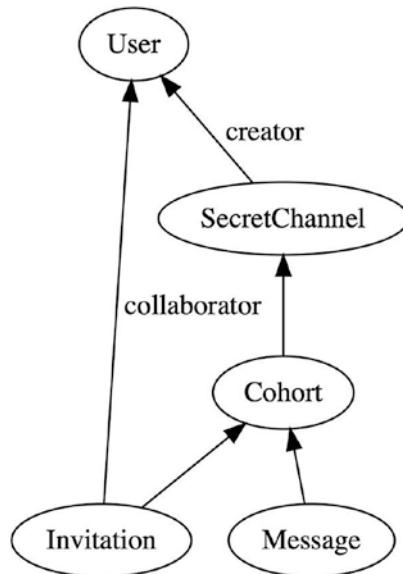
And so we must resort to reusing the shared symmetric key for all secret facts within the workspace. And in so doing, we run up against the problem of revocation once again. We want to be sure that we protect the ongoing work of the remaining collaborators after a former collaborator leaves the team. We can easily enough define an `InvitationRevoke` fact as a successor to an `Invitation`. With a `such that not exists` clause in the distribution rule, intermediate nodes will no longer distribute protected facts to the former collaborator. But that individual still has access to the shared symmetric key. If they were able to coerce the intermediate node to give up its cache of encrypted facts, they would have the tools to decrypt them.

Cohorts

One way to resolve the revocation problem is to move all of the remaining collaborators to a new shared key and issue them new invitations. The former collaborator would not have access to this new key. So while they could continue to decrypt the information they saw while a member of the project, they would not have access to any new information.

Moving collaborators to a new shared key is an inconvenience, but one that can be managed. All of the facts that they created before the move would be encrypted with the old shared key. The collaborators will want to create new facts using the new key, but still have the ability to reference old facts as predecessors. Work must continue uninterrupted.

To model this solution, we insert a level of indirection. Between the workspace and the subsequent work, we inject a *cohort*. This fact represents the group of collaborators that all shared a symmetric key at the same time. All subsequent facts in the workspace are associated with a cohort, which tells us which symmetric key it was encrypted with. Modifying the secret channel model in Figure 7-10, the cohort fits in as illustrated in Figure 7-12.

Figure 7-12. *Collaborators are invited to join a cohort*

Periods

Another solution to the revocation problem is to identify a clock within the domain. Use the clock as a cohort, as demonstrated previously. Each period of time gets its own shared symmetric key. All remaining collaborators receive new invitations at the beginning of each new period. Former collaborators fall away. While this leaves the window of attack open until the beginning of the next period, it has the advantage of being closely tied to the domain. A restaurant might define a period to be a single date of business. A school might define it to be a single semester. And such domains often align the boundaries of cohort membership with the clock. The *Period pattern* is covered in more detail in **Chapter 8**.

We've assembled a suite of tools that can help us control access to a historical model without relying upon an administrative body. Starting with the foundation of PKI, we've used public/private key pairs to identify principals, such as users or autonomous systems. Using the private key to sign a digest, we can prove authorship of each fact. Each node can then run authorization rules for itself to determine whether it believes that the author was authorized to perform the requisite action. And using the public key to encrypt a symmetric key, we can keep facts confidential whether intended for one individual or for a team. All of the information needed to enforce these rules lives within the model itself. This gives each node autonomy to operate securely in a collaborative environment, without needing to rely upon the availability of a trusted third party.

Patterns

You now have a set of tools with which you can build systems that are naturally resilient and reliable. You might even have a good idea how to apply these tools to solve the business problems that your customers face. But there might be some gaps.

In this chapter, we will take a systematic look at how to apply the rules of historical modeling to solving real-world problems. Starting from common issues, we will derive historical solutions. Then using the analytical tools we've developed in the previous chapters, we will examine the consequences of those decisions. The result will be a catalog of patterns that we can reference as we construct new models. This catalog will not be comprehensive, but it will provide a good foundation for exploring new solutions to problems you will face in the future.

Structural Patterns

A large portion of the software that we write for business customers falls under the category of *forms over data*, sometimes known as *CRUD*. This is the kind of software that presents the user with the ability to create, read, update, and delete entities. It is not glamorous work, but it needs to get done.

Relational models and hypermedia models seem to be conceived with CRUD applications in mind. Databases map these four operations to the four primary commands: INSERT, SELECT, UPDATE, and DELETE. Hypermedia applications using POST, GET, PUT, and DELETE seem to reflect the basic operations of CRUD.

But the implementation of CRUD operations in a historical model is not so clear and direct. The most obvious point of dissonance is that a historical model does not allow for updates or deletes. The user wants to perform these operations, but the underlying model does not permit them. And so, we have to find a way to simulate these operations.

Where relational and hypermedia modeling provide direct analog to CRUD operations, historical models require a bit more consideration. To reconcile the

differences between the needs of CRUD and the capabilities of Historical Modeling, let's walk through the CRUD concepts one by one. We will construct patterns that allow us to simulate each of them within the strict rules of immutability.

Entity

Motivation   Represent the creation of an entity.

In a forms-over-data application, a user needs the capability of creating new things. Usually, they will click a button and be presented with a form. Once they fill it out and click another button, the system creates an entity and gives it identity.

The identity of a row in a relational database is sometimes generated by an auto-incrementing ID. This strategy is not appropriate for a historical model, as doing so would rely upon a location-dependent identifier. Different nodes might generate the same ID for different entities. A location-independent identifier is required.

Another point of difference is the initialization of a new entity. Relational databases have INSERT statements, which set all of the columns of a new row to their default or provided values. But in a historical model, it makes less sense for the construction of an entity to initialize its properties. Some future operation will want to modify those properties. The historical fact itself is immutable, so using it to store the initial version of a set of mutable properties is awkward. Doing so would make the initial version something different from the future updates. It would also make those initial values part of the identity of the entity, even after they have been subsequently changed.

The *Entity* pattern focuses on constructing a location-independent identity and avoids initializing mutable properties.

Structure

An entity is a historical fact that contains only identifying information. It contains a natural key, GUID, timestamp, or some combination of those and other location-independent identifiers.

```
fact Entity {
  identifier1: type
  identifier2: type
}
```

Issuing this kind of fact is equivalent to creating the entity. It represents both the identity and the creation of the entity itself.

Example

A product can be represented by a fact that simply captures the SKU (stock keeping unit):

```
fact Product {
  sku: string
}
```

The description, price, quantity on hand, back-order status, and other properties of the product are not stored within the fact. These properties are mutable. The fact is immutable. It represents the identity of the product and the fact that it was created.

Consequences

An entity must use location-independent identity. It cannot use auto-incremented IDs, URLs, or any other location-dependent identifier.

An entity does not contain mutable properties. Any mutable properties that should be associated with the entity are applied with a subsequent fact.

If two nodes create entities with the same identifiers, then they are the same entity. The nodes may not be aware of each other at the time of creation, but any nodes who learn of the two entities will assume that they are the same.

If auditing information—such as the creator, location, or time of creation—is added to the entity, then that becomes part of its identity. Choosing to make that information part of the identity is one way of circumventing the previous consequence—that two entities with the same identifiers are the same entity. Do this only if it is important to the model. Otherwise, keep auditing information outside of the facts themselves.

Related Patterns

An entity that includes the identity of its parent follows the *Ownership* pattern.

While the entity's fact cannot be deleted, the *Delete* pattern simulates the removal of an entity.

Mutable properties are not included within the entity's fact. Instead, they are attached using the *Mutable Property* pattern.

Ownership

Motivation   Represent a strict hierarchy among entities.

It is not uncommon for a model to have a strict hierarchy. In Domain-Driven Design,[1] this structure is referred to as an *aggregate*. In a relational model, this is a special kind of *one-to-many* relationship where each child has only one parent, sometimes with a cascade delete constraint. This kind of strict ownership is often called a *parent–child* relationship.
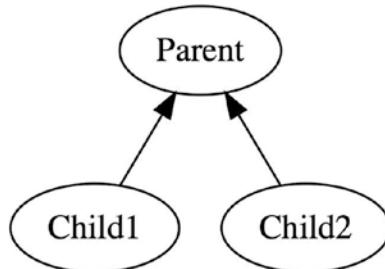
Identifiers often reflect the strict ownership of an entity. In REST, resource identifiers have a path structure that reveal which ones are contained within others. In file systems, each folder exists strictly within one parent folder. The path to the folder includes the identity of the parent.

In a historical model, it is not strictly necessary to identify one predecessor as the owner of a successor. Yet, this is often a relationship that occurs in the problem domain. We will therefore typically represent that special relationship via convention.

The *Ownership* pattern documents a strict parent–child relationship between a successor and one of its predecessors.

Structure

The parent of an entity is represented as a predecessor of its identifying fact, as in Figure 8-1.

Figure 8-1. *Each child belongs to only one parent*

---

[1]Eric Evans. Domain-Driven Design: Tacking Complexity In the Heart of Software. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA ©2003. ISBN:0321125215.

The parent predecessor is listed first within the child fact's fields. It precedes even the child entity's identifiers. While this convention does not make the parent behave differently than any other predecessor, it is a low-cost way of documenting the desired owner relationship.

```
fact Child {
  parent: Parent
  identifier: type
}
```

The child fact is sometimes given a name that includes the name of the parent fact. This is *not* a strict convention and may be violated when the relationship is obvious or names get too long.

```
fact Owner {
  identifier: type
}

fact OwnerItem {
  owner: Owner
  itemIdentifier: type
}
```

Queries for child entities often start from their parent. Given a parent, the query returns all children.

```
query childrenOfParent(p: Parent) {
  match c: Child where c.parent = p
}
```

However, there are occasionally circumstances wherein a query references a child by some other relationship than the parent. When this happens, the query should include the condition that the parent entity has not been deleted.

```
query childrenRelatedTo(r: Relation) {
  match c: Child where c = r.relatedChild
    such that not exists d: ParentDeleted where d.parent = c.parent
}
```

Authorization rules are often expressed along ownership boundaries. To give someone the ability to create child entities, we assign elevated permissions on the parent. These authorization rules must be explicitly defined, as ownership itself is not an implicit part of a historical model.

```
authorize c: Child {
  match a: Authorization where a.object = c.parent.creator
    such that not exists r: Revocation where r.authorization = a
  then u: User where u = a.subject
}
```

Example

An order belongs strictly to the company to which it is placed. Each order also has a distinguishing attribute—a GUID—to separate it from other orders for the same company.

```
fact Order {
  company: Company
  orderGuid: guid
}
```

This fact does not use the conventional name CompanyOrder. The owner prefix in this case simply lengthens the name with no real value. It can be assumed that many of the entities in this system are owned by the company.

An order will contain line items. By convention, we give this child fact a composite name, which includes the name of the parent.

```
fact OrderLine {
  order: Order
  createdAt: datetime
}
```
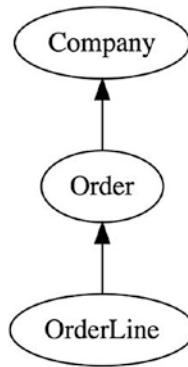
This fact *does* follow the naming convention, as otherwise it could not be assumed that a Line belongs to an Order. Perhaps the system also models Invoices with their associated InvoiceLines.

Both Order and OrderLine follow the convention of listing the owner first among the fields, even before identifiers of the child entities.

The `createdAt` time distinguishes among order lines within the same order. Timestamps are not sufficient identifiers on their own, but combined with other identifiers—such as the parent entity in this case—they can be effective. It is expected that order lines will be added by a single user from a single node and that the number of order lines will be very small.

Also note that `createdAt` is a timestamp captured from the actual creation of the order line. It is the clock time of the workstation that the user was using. This is not the time at which a web server or some other downstream node learned of the order line. It is the time that the user physically clicked the button in the browser or client app.

The resulting model appears in Figure 8-2.

Figure 8-2. *An order line belongs to one order, which in turn belongs to one company*

Consequences

The identity of the parent is part of the identity of the child. Because prerequisites are immutable, children cannot be moved to another parent. Ownership is non-transferable.

The parent must exist before the child can be created. Ownership does not apply to a collection of individual entities that are later grouped after they are constructed.

The *Ownership* pattern encourages multi-tenancy. The identity of a root owner tends to become part of the identities of most other entities. To do otherwise opens the possibility of contamination from neighboring nodes under the control of other organizations, especially if they tend to generate overlapping identifiers.

Related Patterns

If ownership needs to be transferred, consider the *Membership* pattern instead.

The *Ownership* pattern is a special case of the *Entity* pattern, where the entity's identifiers include the identity of an owner.

Delete

Motivation   Represent the deletion of an entity.

Historical facts are immutable; they can be neither modified nor destroyed. But deletion is a regular part of business applications. Deletion, therefore, is simulated by the addition of a new fact.

It is a common practice in a relational database to include a `deleted` column on a table. This is a Boolean flag that is set when the row is intended to be removed. All queries include a `WHERE` clause that excludes deleted rows. This is a pattern known as *soft deletion*.

The *Delete* pattern of historical facts, however, is a little different. Setting a flag is a modification. A historical model does not permit modification. Therefore deletion cannot be simulated by setting a flag. It must be represented as the creation of a new fact.

Structure

A deletion fact takes the entity that it deletes as a predecessor. By convention, the name of the deletion fact is `Deletion` appended to the name of the entity.
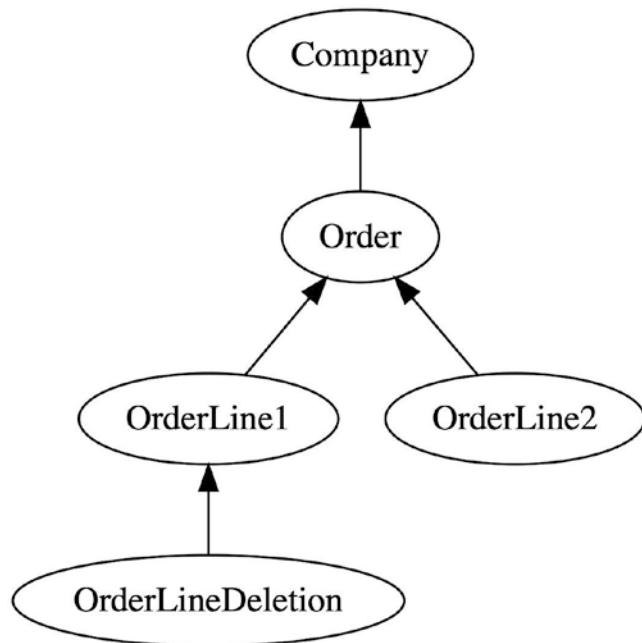
```
fact Entity {
  identifier: type
}

fact EntityDeletion {
  entity: Entity
}
```

Any query for that predecessor must include a `not exists` clause that excludes entities that have been deleted. For example, if the preceding entity had an owner, then the query for children would be expressed as follows:

```
query entitiesInOwner(o: Owner) {
  match e: Entity where e.owner = o
    such that not exists ed: EntityDeletion where ed.entity = e
}
```

Example

In a previous example, order lines were added to an order. If the application allowed a user to remove lines from an order, it would represent those as OrderLineDeletions, as in Figure 8-3.

Figure 8-3. *An order line has been deleted from an order*

The query for lines in an order should exclude all deleted lines:

```
query linesInOrder(o: Order) {
  match ol: OrderLine where ol.order = o
    such that not exists old: OrderLineDeletion where old.orderLine = ol
}
```

Consequences

If the deletion fact has no identifiers to distinguish it from other deletions of the same entity, then the entity can only be deleted once. To allow future restoring of the entity, add a distinguishing identifier. A timestamp will be sufficient in most cases.

Related Patterns

If deletion should be reversible, consider using the *Restore* pattern.

Restore

Motivation	Reverse a prior deletion.

Almost every application that permits deletion employs one of two methods to mitigate accidental deletion. The more common is confirmation. But some will offer a way to restore a deleted entity.

Restoration may begin in a recycle bin that lists all of the deleted entities. Or it may only be available immediately after deletion in the form of undo.

Structure

A restoration fact references a prior deletion. By convention, it appends the word Restoration to the name of the entity. The deletion has an extra identifier, usually a timestamp. The restoration has no extra identifiers.

```
fact Entity {
  identifier: type
}

fact EntityDeletion {
  entity: Entity
  deletedAt: timestamp
}
```

```
fact EntityRestoration {
  deletion: EntityDeletion
}
```

Any query for the entity includes a not exists clause on Deletions, which in turn has a not exists clause on Restorations. If the preceding Entity had an owner, the query for child entities would appear as follows:

```
query entitiesInOwner(o: Owner) {
  match e: Entity where e.owner = o
    such that not exists ed: EntityDeletion where ed.entity = e
      such that not exists er: EntityRestore where er.deletion = ed
}
```
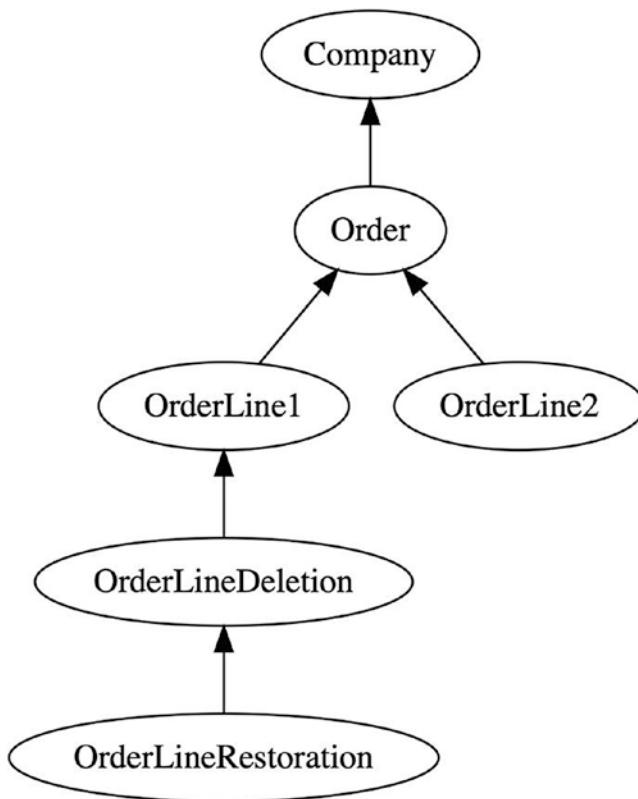
If the user is offered a recycle bin from which to restore entities, it displays the results of a query where a Deletion exists. Notice that this is exactly the same as the previous query except that the not has been dropped from exists ed: EntityDeletion.

```
query entitiesInRecycleBin(o: Owner) {
  match e: Entity where e.owner = o
    such that exists ed: EntityDeletion where ed.entity = e
      such that not exists er: EntityRestore where er.deletion = ed
}
```

The symmetry of these queries makes the deletion and restoration activities atomic. Creating a Deletion both adds the entity to the recycle bin and removes it from the application. Later creating a Restoration both removes the entity from the recycle bin and reintegrates it into the application.

Example

In a previous example, we saw a model that supported deletion of lines from an order. To support restoration of deleted lines back to the order, we would add an OrderLineRestoration fact to the model, as in Figure 8-4.

Figure 8-4. *An order line previously deleted from an order has been restored*

In the *Delete* example, the `OrderLineDeletion` did not require any additional identifier. However, to support *Restoration*, `OrderLineDeletion` now has a timestamp field.

Consequences

If the `Deletion` fact does not have an additional identifier—like a timestamp—then the entity can only be deleted and restored once. Thereafter, it would not be possible to delete the entity again. The second deletion would not be distinct from the first, which had been restored. This is almost certainly not the desired behavior. Therefore, the timestamp is effectively a requirement.

Related Patterns

Restore is an extension of the *Delete* pattern.

If the entity can be reconstructed under a new identity with no loss of fidelity, then consider using the simpler *Delete* pattern. This is often preferable when the entity has no mutable properties and does not participate in workflow. But if properties, workflow, or any other successors are possible, then the *Restore* pattern is more appropriate.

Membership

Motivation   Add entities to groups that can be reorganized over time.

Whereas strict ownership prevents entities from moving from one parent to another, some business applications *do* require this kind of flexibility. An employee can be added to one department and then transferred to a different one later in their career. A project may be part of one portfolio upon initiation, but then reorganized into a different one later. These grouping relationships are not strict ownership, but a more flexible membership.

Structure

The relationship between the member and the group is represented as a fact having both the member and group as predecessors. The membership fact has an additional identifier—usually a timestamp—that allows a member to be removed and re-added to a group over time.
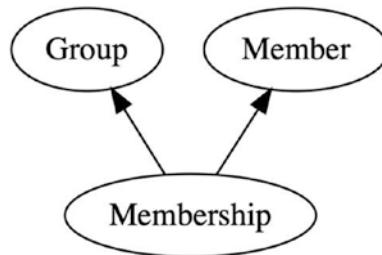
By convention, the member is listed first among the membership fields, before the group. Both appear before the differentiating identifier.

```
fact Group {
  identifier: type
}

fact Member {
  identifier: type
}
```

```
fact Membership {
  member: Member
  group: Group
  createdAt: timestamp
}
```

Whereas with *Ownership,* the parent is a predecessor of the child, in *Membership,* the group and member are not causally related. As Figure 8-5 illustrates, they have a common successor in the membership.
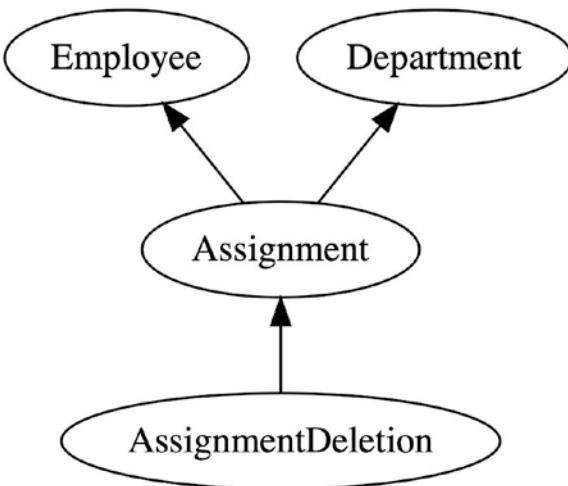
Figure 8-5. *Membership is a successor of both the group and the member*

To find all of the members of a group, query through the membership:

```
query membersOfGroup(g: Group) {
  match ms: Membership where ms.group = g
    such that not exists msd: MembershipDeletion where msd.membership = ms
  then m: Member where m = ms.member
}
```

Example

Employees can be reassigned to different departments over time. Representing the assignment as a distinct fact—rather than a direct predecessor relationship between department and employee—allows the employee to be reassigned without changing their identity. Figure 8-6 depicts this relationship.

Figure 8-6. *An employee is assigned to a department and subsequently removed*

When querying for employees in a department, be sure to include only assignments that have not been deleted:

```
query employeesOfDepartment(d: Department) {
  match a: Assignment where a.department = d
    such that not exists ad: AssignmentDeletion where ad.assignment = a
  then e: Employee where e = a.employee
}
```

Consequences

The model cannot enforce the business rule that an entity belong to only one group. There is no mechanism that prevents two membership facts from having the same entity predecessor. In a relational model, a uniqueness constraint *could* enforce that requirement. But a uniqueness constraint on one node does not prevent an insertion on another. A uniqueness constraint cannot be enforced in an eventually consistent manner across multiple nodes.

Creation and addition to a group are not an atomic process. In the *Ownership* pattern, the parent is created before the entity. In *Membership*, however, membership is created *after* the entity. If the only queries that reach the entity are through membership, then this has little consequence. However, if there is another query that reaches the entity, it may be observable as an *orphan* for an indeterminate period of time. If the application developer wishes to hide orphans, they should add an exists clause to the query.

For example, if *Employee* defined previously included a *Company* owner predecessor, then the following query would include employees not assigned to a department:

```
query allEmployeesOfCompany(c: Company) {
  match e: employee where e.company = c
}
```

To exclude unassigned employees from the results, the developer adds an exists clause requiring that an `Assignment` has been made and not subsequently deleted:

```
query allEmployeesOfCompany(c: Company) {
  match e: employee where e.company = c
    such that exists a: Assignment where a.employee = e
      such that not exists ad: AssignmentDeletion where ad.assignment = a
}
```

Related Patterns

If the model requires that the entity be a member of only one group, and that group cannot change, then consider using the *Ownership* pattern instead.

If the model requires that membership in one group be replaced with membership in another group, then consider applying the *Entity Reference* pattern. Model membership the group as a reference to the group fact, superseding prior references for the same entity. While this will not prevent concurrent changes, it will at least make removal from one group and addition to another an atomic operation.

Mutable Property

Motivation   Represent values that change.

Historical facts are immutable. They do not change. Yet users expect to be able to change properties. The *Mutable Property* pattern represents changes to properties over time using only immutable facts.

It is desirable in a distributed system for nodes to be able to act in isolation. A user should have the autonomy to change a property without requiring a connection to any

other node. The user might be on a mobile phone that is temporarily disconnected from the server. Or it might simply have a slow network connection, and the latency of performing a connected update would be undesirable.

With capability of autonomous change comes the possibility of conflicts. The disconnected user might change the same property as someone who is connected. Or two users on a slow connection might change the same property at more or less the same time. When each of their changes propagates to the other, the conflict will be detected. The system needs to include the capacity for resolving those conflicts.

Structure
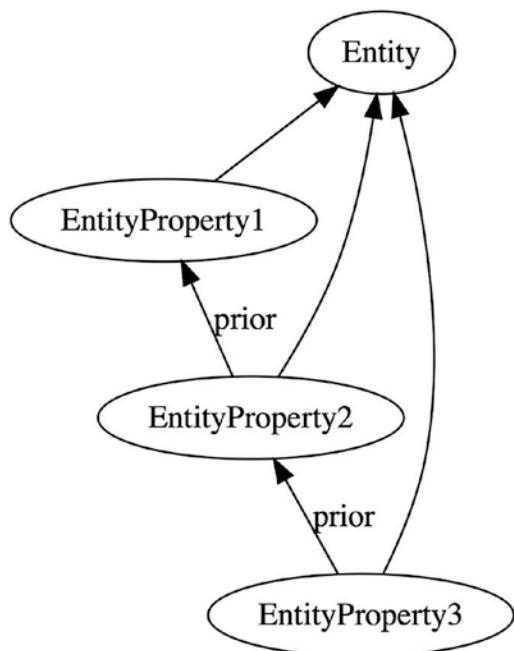
A mutable property is represented as a fact having the entity as a predecessor and the value as a field. To keep track of changes over time, it records prior versions in a predecessor set.

By convention, the name of the fact appends the property name to the entity name. The set of prior versions is conventionally called `prior`. This set is empty for the initial value.
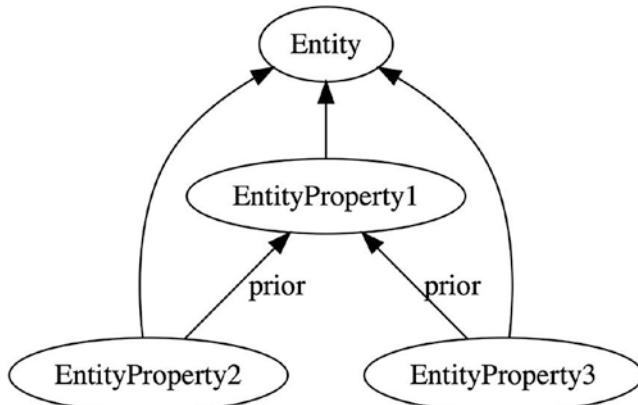
```
fact Entity {
  identifier: type
}

fact EntityProperty {
  entity: Entity
  value: type
  prior: EntityProperty*
}
```

As a user changes the property, the `prior` set captures only the most recent version. Under ordinary circumstances, this forms a linear chain of property facts, as Figure 8-7 demonstrates.

Figure 8-7. *In a chain of versions, each points back to its immediate predecessor*

If two users (or one user on two devices) change a property concurrently, the graph will fork. The result will be a tree like the one in Figure 8-8 with more than one leaf.
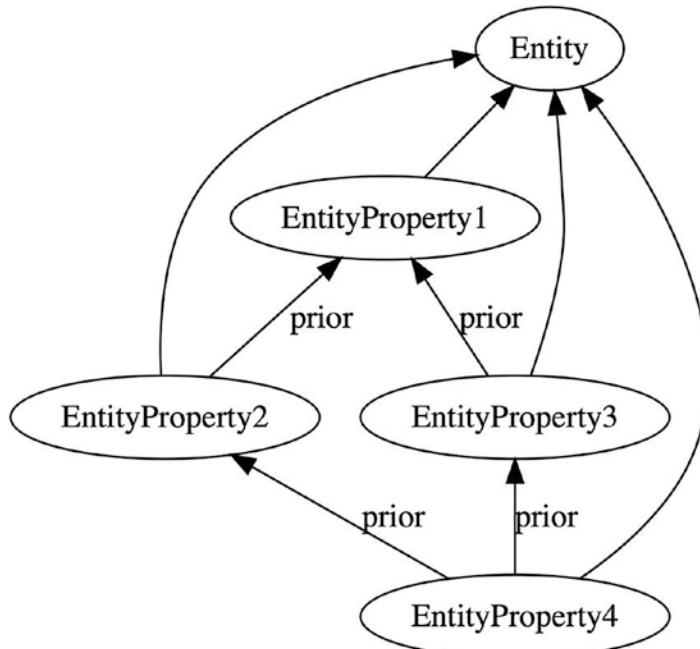
Figure 8-8. *Concurrent changes result in multiple leaves*

When a node computes a tree with multiple leaves, it recognizes a concurrent change. In this situation, the application will typically present all leaves as candidate values. Each leaf represents a value that was concurrently set for the property and has not been superseded. The user can select among the candidate values and resolve the dispute.

Alternatively, an application can compute a resolution on its own. This is usually accomplished via a simple function over the leaves, such as a maximum. In rare situations, however, the application developer may choose to base the resolution on the nearest common ancestor of all leaves. One example is a source control system like Git that computes a three-way merge. Such a complex function is not appropriate for most applications.

In any case, the node determines what to present, but it *does not generate any new facts*. Facts are only generated as a result of a user's decision. When the user changes a property from a concurrent state, the system includes all of the leaves of the tree in the new fact's `prior` set. This results in a graph like Figure 8-9 that again has a single leaf.

Figure 8-9. *Concurrent changes result in multiple leaves*

To compute the set of leaves, a node simply runs a query with a `not exists` clause on the `prior` set:

```
query valuesOfProperty(e: Entity) {
  match p: EntityProperty where p.entity = e
    such that not exists n: EntityProperty where n.prior = p
}
```

If the query returns one fact, then that fact represents the most recent version. If it returns many facts, then they represent the leaves and can be used as candidate values.

A property does not have to have a single `value` field. It is not uncommon for multiple values to change as a unit. In Domain-Driven Design, this situation arises when a property uses a *value type*. In these situations, all of the components of the value type appear as fields in a single property fact.

Example

An order in our example company has a billing address. This is a set of fields that change as a unit. It makes no sense to change, for example, the state without also changing the city and street. The fields of a billing address are therefore treated as a single atomic fact.

```
fact OrderBillingAddress {
  order: Order
  street: string
  city: string
  state: string
  zipCode: string
  prior: OrderBillingAddress*
}
```

The current billing address of an order is given by the following query:

```
query billingAddressOfOrder(o: Order) {
  match ba: OrderBillingAddress where ba.order = o
    such that not exists n: OrderBillingAddress where n.prior = ba
}
```

If the query returns one fact, then that represents the most recent billing address. If, however, it returns multiple billing addresses, then concurrent changes have occurred and the facts represent the candidate billing addresses. The application presents all candidates to the user so that they can research and resolve the issue.

An order will also include a shipping address. This is represented as a separate fact from the billing address, even though it has mostly the same fields.

```
fact OrderShippingAddress {
  order: Order
  street: string
  city: string
  state: string
  zipCode: string
  prior: OrderShippingAddress*
}
```

A similar query gets the latest shipping address. While it is unusual to change only one part of an address at any given time, it is not uncommon to change only the shipping address or only the billing address. That is why the application developer chose to make them separate facts.

While concurrent changes to billing address will result in multiple leaves, concurrent changes between billing address on one side and shipping address on the other will not. The system will simply present the most recent billing address beside the most recent shipping address. This reflects the intent of the application developer, as expressed by the decision that shipping and billing address have no causal relationship between them.

Consequences

Nodes observing the *Mutable Property* pattern can act autonomously. They can record a new value for a property without first connecting with any other node to prevent concurrent changes.

Said another way, concurrent changes *cannot be prevented*. There is no mechanism within a historical model that would ensure that only one change can be made at any given time. Properties can neither be locked nor serialized.

Nodes will recognize that concurrent changes have occurred *post facto*. All nodes will eventually receive the same graph, compute the same leaves, and therefore come to the same conclusion. Concurrent changes do not result in conflict.

When a user attempts to modify a property, the application should first verify whether the value has actually been changed. The application might, for example, display a dialog box with "OK" and "Cancel" buttons. The user might click "OK" even if they made no change. If the application created a new fact without checking whether the value had changed, it would create an unnecessarily complicated history.

The mutable property fact should not contain any auditing information. This allows two different users to change a property to the same value without introducing a concurrent update. If the fact contained, for example, the user or timestamp, then two concurrent changes to the same value would appear as distinct facts. The result would be an unnecessary merge between similar changes.

The response to multiple leaves must be based only on the information in the facts themselves. It must not be based, for example, on the order in which the facts arrived at the node. The result is a function that is commutative and deterministic; it computes the same result at every node regardless of message order. That is why the results of queries are unordered *sets* and not ordered *lists*.

If a node computes a resolution to a concurrent change, it must do so only on read. It must not attempt to create a new fact to resolve the concurrent changes. To do so would be to introduce the possibility of a never-ending storm of concurrent resolutions. Consensus algorithms such as Paxos are carefully constructed to avoid these resolution storms, but without such careful consideration, storms can easily arise. At any rate, strong eventual consistency demands convergence without consensus. This is achievable if all nodes run the same deterministic function on read.

The *Mutable Property* pattern cannot guarantee that a property has a single value. The query will always result in a set. Applications must be written to expect that that set might have multiple values. While it is sometimes tempting to introduce a location-specific rule to prevent concurrent updates—only one user is allowed to change a property, or only one node can be used to make that change—such rules are ultimately difficult to enforce and impose undesirable constraints on the system.

A query for the current value of a property could return an empty set. This represents the situation in which the property has not been initialized. On remote nodes, this could also indicate that the entity has been transmitted, but its initial properties have not yet arrived. Creation of an entity is not atomic with initialization of its properties. If an application developer intends to present only entities that have been initialized, they could add an `exists` clause based on the property fact.

Related Patterns

If the mutable property represents a relationship with another entity, the pattern becomes an *Entity Reference*.

Entity Reference

Motivation	Represent a mutable relationship between entities.

Where *Ownership* and *Membership* are strict grouping constructs, some relationships between entities are simple references. These references don't imply any kind of belonging or grouping, but rather just an association.

An entity reference is a property that points to another entity. In Domain-Driven Design, the referenced entity is typically an *aggregate root*, possibly in a different *bounded context*. In an object-oriented language, the entity reference is a pointer to another object. And in a relational database, it's a foreign key. The reference is typically mutable and often will be initialized to NULL.

A relational database will use foreign keys to represent *Ownership*, *Membership*, and *Entity Reference*. To distinguish among them, first, look to the cardinality. A many-to-many relationship typically denotes *Membership*. A one-to-many relationship that has a cascade delete constraint represents *Ownership*. A less constrained one-to-many relationship—especially one that allows NULL—is probably an *Entity Reference*.

Structure

The structure of an entity reference looks very similar to a *Mutable Property*. It is a fact having the primary entity and referenced entity as predecessors. The referenced entity is often nullable. Just as a mutable property does, the fact keeps the set of prior versions of the entity reference.

```
fact EntityReference {
  entity: Entity
  referencedEntity: ReferencedEntity?
  prior: EntityReference*
}
```

265

The distinction between the primary and referenced entity is an important one. The primary entity is the one with the reference property. Creating a new `EntityReference` fact changes the value of that property for the primary entity. The `prior` set will include other `EntityReference` facts that refer to the same primary entity.

Querying for the current value of an entity reference begins with the primary entity. Like a property query in the *Mutable Property* pattern, it matches references that have not been superseded. The query includes one additional clause that follows the entity reference.

```
query referencedEntity(e: Entity) {
  match er: EntityReference where er.entity = e
    such that not exists n: EntityReference where n.prior = er
  then re: ReferencedEntity where re = er.referencedEntity
}
```

Unlike *Mutable Properties,* entity references permit queries in the opposite direction. To query from a referenced entity back to all entities that reference it, include the `not exists` clause on `prior`. This prevents the query from returning entities with references that have been superseded.
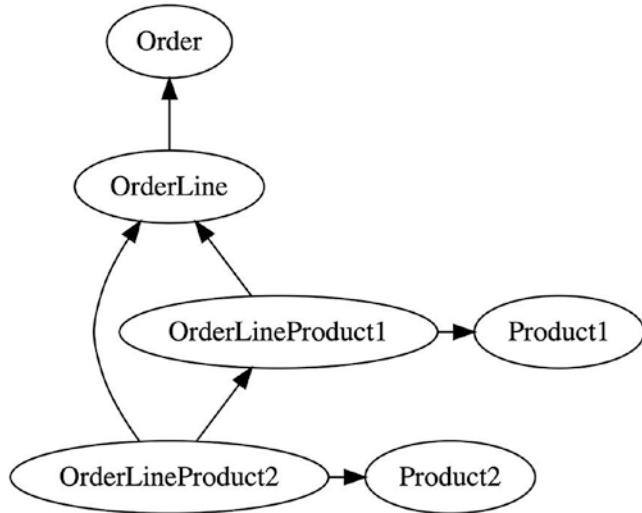
```
query entitiesReferencing(re: ReferencedEntity) {
  match er: EntityReference where er.referencedEntity = re
    such that not exists n: EntityReference where n.prior = er
  then e: Entity where e = er.entity
}
```

Example

An order line references the product that was purchased. This relationship is optional: some order lines represent fees, discounts, or services not listed in the catalog. The `OrderLine` therefore has a reference to the `Product` entity.

```
fact OrderLineProduct {
  orderLine: OrderLine
  product: Product?
  prior: OrderLineProduct*
}
```

This creates the relationship demonstrated in Figure 8-10.

Figure 8-10.   *Two versions of an order line, each referencing a different product*

A query for the product referenced by an order line begins like any other mutable property query. But then it contains an extra clause to get the referenced Product.

```
query productForOrderLine(ol: OrderLine) {
  match olp: OrderLineProduct where olp.orderLine = ol
    such that not exists n: OrderLineProduct where n.prior = olp
  then p: Product where p = olp.product
}
```

Traversing the graph from the opposite direction, we can query for orders that purchase a given product. This query includes the same not exists clause.

```
query ordersContainingProduct(p: Product) {
  match olp: OrderLineProduct where olp.product = p
    such that not exists n: OrderLineProduct where n.prior = olp
  then o: Order where o = olp.orderLine.order
}
```

This similarity between the two queries makes them behave atomically. When an order line is changed to reference a different product, both of the queries are affected. The first query will no longer return the previously referenced product, and the second query will no longer return the order.

Consequences

Just as with the *Mutable Property* pattern, an *Entity Reference* cannot guarantee that only one entity is referenced. The query for the current reference returns a set. An application must respond appropriately to a set larger than 1. This represents a concurrent update of entity references.

The results of the query could also be the empty set. This could occur—just as in *Mutable Property*—when the reference has not yet been initialized. But it could also occur when the reference has been set to NULL.

Related Patterns

This is a variant of the *Mutable Property* pattern in which the value of the property is a reference to another entity.

This is sometimes used as an alternative to the *Membership* pattern to indicate that an entity should be a member of only one group. While it cannot enforce that rule, it at least makes the transfer from one group to another an atomic operation.

Workflow Patterns

While CRUD operations make up an important part of business application development, they do not tell the entire story. The next set of business operations to consider are concerned with taking entities through a workflow. Workflow is typically the realm of business process modeling, state transition diagrams, and flow charts. It is the study of collaborative steps that move work from inception to completion.

Tracking the flow of work through a system that allows mutation is an exercise in frustration. If each step of the process has the potential to change the work item, then reasoning about the behavior of the system requires careful analysis of all possible permutations. Allowing for parallel execution, mutability often leads to race conditions.

But within an immutable architecture, workflow is a much simpler process. It begins with capturing the work to be performed in an immutable object. Any further changes to the source object are ignored. Then, we set up a query to identify which work items are ready for any given process. Finally, we capture the outcome of that process in an immutable way that atomically moves the work item along to the next step.

Adding workflow to an application turns it from an anemic forms-over-data model into a system that can assist with communication among collaborators. Not every subdomain of an application needs workflow, but the most central bounded contexts that provide the most business value often do.

Transaction

Motivation   Capture a known state of an entity to perform an atomic unit of work.

The structural patterns that we just explored allow an entity to change over time. The changes are captured as immutable facts, but the accrual of new facts as the user interacts with the system simulates changes to an entity. At some point, the user will decide to take some action. Any further changes to the entity after that point should not affect that action.

Users might be adding items to a shopping cart. They can remove items, replace them, and restore them back to the cart. They can change the quantity, product, shipping options, delivery address, and any other property. The structural patterns in the previous section allow those operations.

But then when the user submits the order, the items and all of their properties should be locked down. No additional items can be added, and no properties can be modified. Processing may begin at any time, and a change to an order in flight would be disruptive to business.

The *Transaction* pattern takes advantage of immutability for business processing. It records the information about a request for work in such a way that it cannot be modified after work begins.

Structure

A *Transaction* identifies as a predecessor an entity that it is acting upon. Whereas that entity was originally a starting point for children, mutable properties, and other successors, the transaction now seeks to lock it down. It does so by inverting the predecessor/successor relationship.

Where *Ownership* placed the parent as a predecessor of its children, *Transaction* makes children predecessors of parents. Successors can be added over time, but predecessors are immutable. Recording children as predecessors prevents further creation or deletion.

269

The transaction also identifies the specific versions of *Mutable Properties*. These become direct or indirect predecessors of the transaction. Again, the relationship is inverted so that any further modifications to those properties do not affect the transaction.

A `TransactionItem` as described as follows is a child of a `Transaction`. The `Transaction` has a set of `TransactionItem` predecessors. Furthermore, a `TransactionItem` captures one specific version of a mutable property.

```
fact TransactionItem {
  itemContext: ChildEntity
  property: ChildProperty
}

fact Transaction {
  transactionContext: ParentEntity
  items: TransactionItem*
}
```
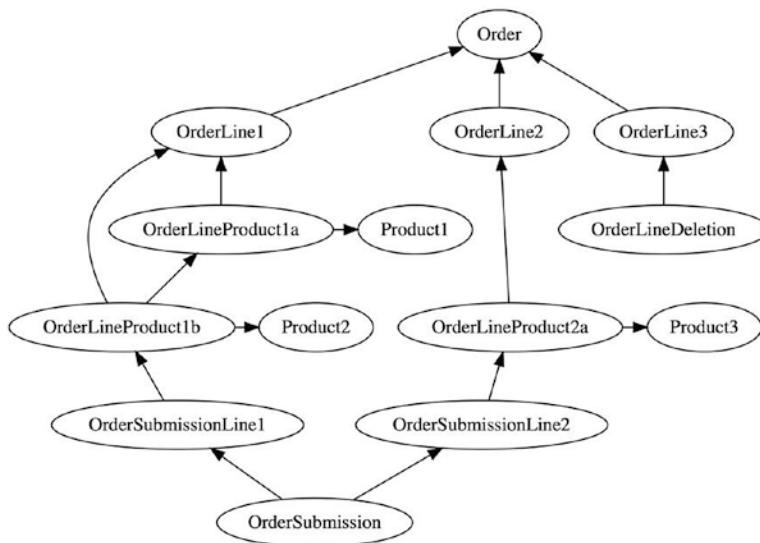
The transaction and all of its items are captured at a single machine. This is typically the workstation that the decision maker is using. When a user decides to issue a transaction, the system captures the state of the objects as they are known to that user at that time. Creating a transaction does not require the machine to communicate with any other node, as all of the information required is local.

Example

When a customer submits an order, they lock down its current state. They cannot make further changes to the order. They can only request a subsequent return and new orders.

We start from the order structure currently in place, using patterns like *Entity*, *Delete*, and *Mutable Property*. Then we create a parallel model—demonstrated in Figure 8-11—that inverts the predecessor/successor relationships. Items in the order become predecessors so that new items cannot be added.

Figure 8-11. *An order submission inverts the model to lock in predecessors*

Any lines that have been deleted are not included in the OrderSubmission. Other lines might be subsequently deleted, or deleted lines later restored. Neither change will affect the OrderSubmissionLines that have been captured.

All of the arrows point out of the OrderSubmission. All of the information required to process the order can be found by traversing the graph in one direction. Given an OrderSubmission, any node will compute exactly the same order. This locks in the items, products, and quantities.

Consequences

Once a transaction is recorded, subsequent changes to the entities or properties will have no effect. All of the information in the transaction is recorded in predecessor relationships. Predecessors are immutable, so the transaction is locked down.

All nodes receiving the transaction see it in exactly the same state. The identity of a fact includes the identities of its predecessors. Any difference in predecessors such as transaction items or property versions would necessarily result in different facts.

A transaction is processed atomically. Items may arrive at a node ahead of their transaction. But processing begins with a query for a transaction, not an item. Items will remain dormant until the subsequent transaction arrives, at which time all items will take effect simultaneously.

All necessary information must be in the transitive closure of the transaction. Starting at the transaction fact, follow all predecessors. From those facts, recursively follow their predecessors. The transitive closure is the set of all facts thus visited.

Related Patterns

The *Transaction* pattern inverts the predecessor/successor relationship found in the *Ownerhship* and *Mutable Property* patterns.

A transaction is often placed in a *Queue* or an *Outbox*, and is usually associated with a *Period*.

Queue

Motivation	Manage work to be processed manually.

Work that a person needs to handle is often presented in a list. The user interface shows the user a set of work items that requires their attention. The user selects a work item and navigates to part of the application where they can handle it.

The user might get interrupted. So the work remains on the queue until they actually complete it. If another user observes the queue, they will be able to see the same work item.

The *Queue* pattern presents a set of work items that are ready for manual processing. It ensures that a work item is removed from the queue when it is completed.

Structure

A queue is nothing more than a query that returns facts representing work to be done. The query starts from some root-level entity—for example, an *Owner*—and matches children for which an action has not been performed.
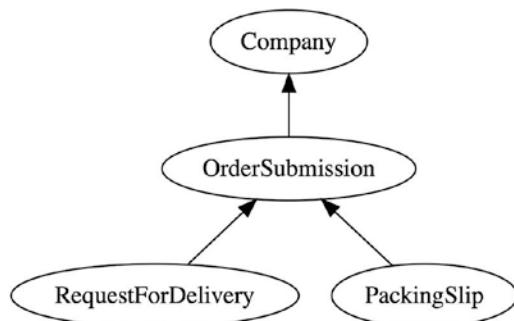
```
query workToDo(o: Owner) {
  match wi: WorkItem where wi.owner = o
    such that not exists a: Action where a.workItem = wi
}
```

In the process of performing the requested work, the user will create an `Action` fact. The action records the outcome of the work.

Because the action appears in the `not exists` clause, the work item is removed from the query once the action has been performed. Recording the action and removing work from the queue occur in a single atomic operation.

Example

Once an order is submitted to our hypothetical company, the shipping department picks product to fulfill a submitted order and prints a packing slip. The logistics department, meanwhile, calls for a delivery truck. Each of these is a manual process. The relationship among order submissions, delivery, and packing slips is captured in the model depicted in Figure 8-12.

Figure 8-12. *An order submission triggers both the request for delivery and the packing slip*

The shipping manager knows what orders to pick based on a query. The query looks for order submissions that do not yet have a packing slip.

```
query ordersToPick(c: Company) {
  match o: OrderSubmission where o.company = c
    such that not exists ps: PackingSlip where ps.orderSubmission = o
}
```

Once the order is picked, the shipping manager prints a packing slip. The act of doing so removes the order from this queue.

Meanwhile, the logistics team runs another query to find orders that do not yet have a delivery request:

```
query ordersToShip(c: Company) {
  match o: OrderSubmission where o.company = c
    such that not exists rd: RequestForDelivery where rd.orderSubmission = o
}
```

They call for a truck and then enter the RequestForDelivery into the system. Once they do so, the order no longer appears in the query. It has been removed from the queue.

We have deliberately chosen not to have a predecessor/successor relationship between RequestForDelivery and PackingSlip. The delivery request can be made before the product is picked. Or, based on volume, the warehouse might find themselves backlogged and choose to delay the request for delivery.

When the shipping manager predicts that the warehouse is about to be backlogged, they notify logistics to switch to a different query. Now they wait for orders to have been picked before requesting delivery.

```
query pickedOrdersToShip(c: Company) {
  match o: OrderSubmission where o.company = c
    such that exists ps: PackingSlip where ps.orderSubmission = o
    and not exists rd: RequestForDelivery where rd.orderSubmission = o
}
```

The creation of a PackingSlip atomically moves the work from the shipping manager's queue into the logistics queue. The subsequent creation of the RequestForDelivery removes it from the logistics queue.

The packing slip is not a hard prerequisite. It is not a predecessor of the request for delivery. But by switching from one queue to another, the company can adjust its business process to better respond to circumstances.

Consequences

The action performed on a work item is used in the not exist clause of the queue. As a result, recording the action and removing the work item from the queue is a single atomic operation.

Unlike a FIFO (first in first out) queue, the queue query does not impose an order on the work items. The results of a query are a *set*, not a *list*. If order is important, place a timestamp on the work item fact. Use that timestamp to order the set for presentation to the user.

Related Patterns

If work is to be performed automatically instead of manually, then the *Outbox* pattern is more appropriate.

Work items in a queue are often *Transaction*s.

Work items are often grouped by unit of time. This recognizes a natural period that the business already recognizes. Application of the *Period* pattern has the extra benefit of preventing the queue query from slowing down as history accrues.

Period

Motivation   Bound the accrual of facts with discrete time slices.

The time required to query a historical model is governed by the number of successors that the starting point or intermediate fact has. If we start each query from the root of the graph, those queries would get slower over time. Starting further down the graph at a fact that has a bounded number of successors will keep performance constant as we accrue more facts.

Any feature of the system that limits the number of successors is a good candidate for subdividing the graph. The one that is most readily available is time. The *Period* pattern subdivides the historical graph by discrete units of time. While the total number of facts is expected to grow, the number *per unit period* will remain somewhat more bounded.

In addition to the performance benefits, associating facts with a period often captures an important business concept. Accountants tend to close their books on daily, monthly, and quarterly periods. They do this not just to limit the size of a ledger but also to give themselves reporting boundaries. The *Period* pattern seeks to do the same with application data.

Structure

A period is a fact that has one *Owner* predecessor to give it context and one discrete time value. The time is measured in coarse units; it is not a continuous timestamp.

```
fact Period {
  owner: Owner
  time: discreteTime
}
```

Typical choices for the discrete time unit are calendar or business day, month, quarter, or year. For high-throughput systems, units may go down to the hour, but rarely smaller.

Work items include a period as a predecessor. Queries for work start at the period.

```
query workToDo(p: Period) {
  match wi: WorkItem where wi.period = p
    such that not exists a: Action where a.workItem = wi
}
```

Results from two or more queries are unioned together to produce an overlapping query. The overlap is chosen to allow plenty of time for remote nodes to connect and share their work items and for those work items to be processed before the period rolls off.

The period has no additional fields, so that the owner and discrete unit of time produce a unique fact. All nodes creating work items produce the same fact. And each query for work items creates the starting point in the same way.

Periods are sometimes captured hierarchically. The largest period—say a year—falls directly under the owner. The next period down—for example, a quarter—has the larger period as a predecessor. Periods organized in this way must share a boundary; month and week cannot be organized in a hierarchy. This is usually done for reporting rather than performance reasons.
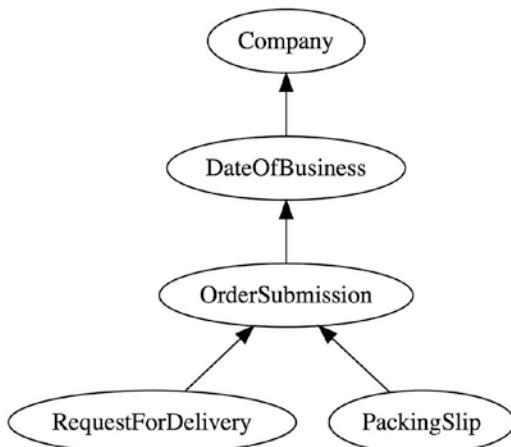
Example

In the previous example, we added order submissions directly to the company. As time passes, the system searches more orders within the company to find the ones that have not yet been picked or shipped. We can make things easier on the system and record an important dimension of the model at the same time.

The `DateOfBusiness` fact has one predecessor and one field:

```
fact DateOfBusiness {
  company: Company
  businessDate: date
}
```

We insert a `DateOfBusiness` between `Company` and `OrderSubmission`, as shown in Figure 8-13. This fact captures the date on which the order was submitted.

Figure 8-13. *An intermediate fact groups orders submitted to a company by date of business*

Date of business is not strictly determined by the computer clock. An order may be counted toward the next date of business if it is placed after hours, or if it occurs on a weekend or holiday. In fact, the company may even choose a policy wherein orders placed after 3:00 are associated with the next date of business. The period is an operational construct, not a physical one.

Not all nodes need to advance to the next period at exactly the same time. There is no need to rigorously synchronize the clocks across the workstations on which users are submitting orders. If one workstation starts submitting orders into the next date of business while another workstation remains on the current one, then those orders are simply counted in different periods. This will not cause any problems as long as there is no causal relationship between the order submissions. And the fact that the developer chose not to make one `OrderSubmission` the predecessor of the other indicates that there *should* be no causal relationship.

To determine the orders to pick for any given date of business, we find those that do not yet have a packing slip:

```
query ordersToPick(dob: DateOfBusiness) {
  match o: OrderSubmission where o.dateOfBusiness = dob
    such that not exists ps: PackingSlip where ps.order = o
}
```

We run this query for two dates of business—the previous one and the current one—and union the sets. Any given order submission occurs in only one date of business, as indicated by the singularity of its predecessor, so this practice does not risk duplication. But the overlap *does* prevent us from missing orders. As long as the period is significantly longer than the SLA, we will have received and processed a day's orders before we roll the query forward too far.

Consequences

A work item should have only one associated period. If a unit of work is broken into smaller units, and those each have their own period, then it would be possible to split the work between two periods. Think of a train moving across a switch at the same time that the switch is thrown. If the cars are not connected, then there is no problem. But if they are attached to one another, this could cause some unintended consequences.

Two or more periods should be queried for work items. The overlap provides a buffer of time for work items to arrive and be processed. If upstream nodes can be offline, the number and duration of overlapping periods must be chosen to allow them to reconnect. As long as the expected time to receive and process work items is significantly shorter than the duration of overlapping periods, then work will not typically be lost.

There is no mechanism in the model to guarantee, however, that work won't be delayed beyond the oldest period queried. The processing system should be flexible enough to be manually reset to pick up lagging work items. A query from the *Owner* one predecessor higher than the period can encompass all periods. While this query would be slower, it would look back in time for any missed work items. A business decision can then be made to determine the best corrective action.

Related Patterns

Periods are often used as the starting point for queries in the *Queue* or *Outbox* pattern. An unbounded queue gradually becomes a performance problem. But a queue bounded by the expected number of work items per period is much easier to manage.

The work items within a period are often *Transaction*s.

Outbox

Motivation   Send work to an external system that does not follow immutable architecture principles.

Distributed systems are heterogeneous. Components designed with differing architectural constraints will need to interact with one another. We will find ourselves sending requests from an immutable system into a location-dependent system.

At the boundary between immutable and location dependent, we have an API. The immutable system runs a service that calls the API whenever a fact appears in a *Queue*. It then records the results of that API call in a new fact that removes the work from the queue.

A single instance of a service would be easy to implement, but it would ensure neither high availability nor high throughput. For those properties, we need redundancy. And that's where implementing a service gets difficult.

When sending work to a location-dependent API, it is often beneficial to limit the number of duplicate requests. If the system is not idempotent, it might incorrectly duplicate the work. If so, we would like to ensure—as nearly as we can—that requests are sent exactly once. But even if the downstream system *is* idempotent, multiplying every request by the number of parallel services is unnecessarily wasteful.

The *Outbox* pattern provides a mechanism by which parallel services avoid sending duplicate work requests to third-party systems. It cannot prevent duplication altogether, but it can take steps to reduce them.

Note that there is no corresponding *Inbox* pattern. When information is received from an external system, it is simply turned into a fact. No special conversion pattern is necessary in this direction.

Structure

The *Outbox* pattern integrates with location-dependent services by becoming location-dependent itself. Unlike the other patterns presented here, this one is not implemented entirely within the rules of immutable architecture. Instead, it uses a location-specific journal to keep track of successful API calls.

Journaling

The journal records the result of API requests made to the remote system. The index into the journal is the hash of the work item fact that triggered the API call. The journal contains all pertinent data received from the API. It only records successful API calls.

When everything works correctly, the service performs the following actions in order:
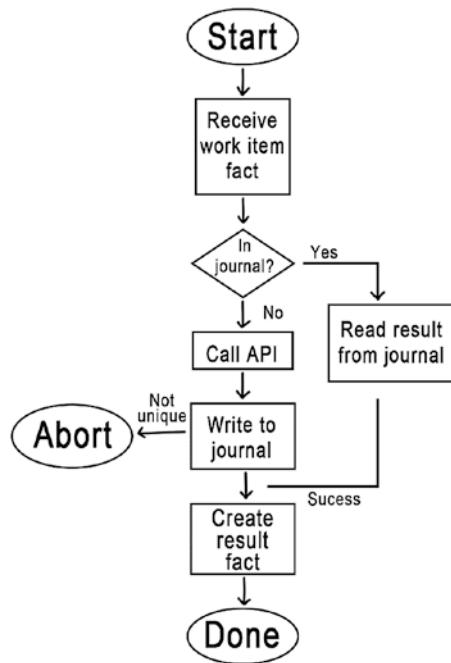
1. Receive a work item fact from a queue query.

2. Call the API.

3. Store the results of the API call in the journal.

4. Create a fact with the results of the API call.

The fact created in step 4 also has the effect of removing the work item from the queue. The next time the service runs the query, the work item fact will not be present. This is the "happy path."

When things don't work correctly, the service may fail partway and find itself repeating these steps. The journal is intended to reduce the probability that the API will be called more than once. It does so by providing a way to skip the API call in step 2 in some failure scenarios.

After a service receives a fact (step 1), it checks the journal for a matching row. The journal is indexed by the hash of the work item fact. If a matching fact is found, then a previous or parallel invocation of the service had completed step 3. The service reads all of the information about the result of the API call and proceeds to step 4 to create the fact.

After the service makes the API call and receives a successful result, it attempts to insert that information into the journal. The journal, however, has a uniqueness constraint on the work item fact hash. The insert will therefore fail if a parallel service inserted its results first. When this happens, the service has just detected a duplicate call to the API. It aborts and lets the parallel invocation finish the job. The full flow of the journaling algorithm appears in Figure 8-14.

Figure 8-14. *Journaling reduces the likelihood of duplicate API calls*

Random Processing Delays

Journaling reduces the chances of duplicate successful API calls, but it does not prevent them. One of the ways in which duplication can still occur is for two nodes to run the service on the same work item in parallel. We can take additional steps to make parallel execution less likely.

The simplest way to reduce the likelihood of parallel execution is to introduce a random processing delay. Consider a service that uses polling to query the queue for work items. It wakes up at regular intervals and runs its query. If it finds some work items, it processes *one of them* and runs the query again. It does not process all of them, because doing so adds time during which a different service could wake up and run the same query. It simply selects one work item at random and leaves the rest in the queue.

We can configure all of the nodes to wake up the service on the same schedule. Perhaps we simply create a cron job that runs once a minute. But, to reduce the likelihood of parallel execution, we wait a random number of seconds before running the query.

This is a very simple technique. Combined with the journal and a relatively fast downstream API, it can be quite effective. But it is only appropriate for low-throughput interfaces. It introduces unnecessary latency and limits the frequency with which work items can be processed.

Rendezvous Hashing

When low latency and high throughput are required, a more sophisticated mechanism can be employed. Rendezvous Hashing[2] is a technique for uniquely allocating objects to nodes. It is often used in distributed caches. We will adapt it to instead allocate work item facts to services. A similar algorithm—Consistent Hashing[3]—can be adapted just as well.

To begin with, each service instance generates a random number when it starts up. It registers its number with the other services. The service registry could be implemented with a gossip protocol, a distributed hash, or even the same database that is used to keep the journal. The only requirement is that the other services become aware of this new member shortly after the service comes online.
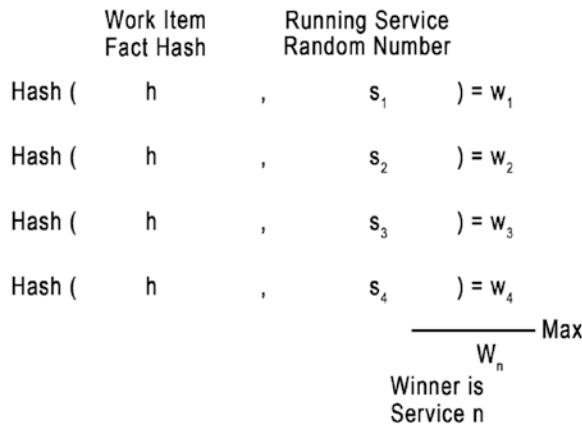
Once it has registered, a service subscribes to new work items entering the queue. Unlike the random processing delay solution, services do not poll. They are notified via webhooks, broadcast, or a publish–subscribe message queue as soon as work is available. Which mechanism they use depends upon your chosen communication infrastructure.

When a service receives a work item, it computes the hash. But before it checks the journal, it pairs the fact hash with each of the random numbers of all of the registered services, including itself. It computes the hash of each pair, producing a *weight*. The service with the highest weight is the one that should process the work item. If that winner is the service itself, then the service checks the journal and processes the work item. The algorithm is depicted in Figure 8-15.

---

[2]Thaler, David; Chinya Ravishankar. "A Name-Based Mapping Scheme for Rendezvous." University of Michigan Technical Report CSE-TR-316-96.

[3]Karger, D.; Lehman, E.; Leighton, T.; Panigrahy, R.; Levine, M.; Lewin, D. (1997). Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing. ACM Press New York, NY, USA. pp. 654–663.

$$\text{Hash}(\quad h \quad , \quad s_1 \quad) = w_1$$
$$\text{Hash}(\quad h \quad , \quad s_2 \quad) = w_2$$
$$\text{Hash}(\quad h \quad , \quad s_3 \quad) = w_3$$
$$\text{Hash}(\quad h \quad , \quad s_4 \quad) = w_4$$

Work Item Fact Hash — Running Service Random Number

Max W_n — Winner is Service n

Figure 8-15. *A node computes weights for one work item to determine the winning service*

All services will compute the same weights for a work item. They will therefore all select the same winner. Only that winner will process the work item, resulting in less chance of parallel execution.

Service Failure

Unfortunately, nodes fail. When a service stops responding, its work items will remain in the queue longer than expected. Fortunately, the other services can detect this.

Since all services compute the weights for all work items, each service can see where it falls in the rank. If a service determines that it is the second-place winner, then it keeps track of the work item. If it sees it again after a timeout, then it assumes that the first-place winner has failed. It processes the work item and removes the failed node from the registry.

If a service that has *not* failed finds itself removed from the registry, it just creates a new random number and comes back in. The timeout should be high enough to make this scenario unlikely, but low enough that failures don't go undetected for too long.

Failure detection can be generalized beyond the second-place winner. Third-, fourth-, and higher-place winners can set longer timeouts on the work items. This will mitigate against a simultaneous failure of multiple services, as such would be caused by an infrastructure or network outage.

Example

We are integrating our example company with a third-party accounts receivable system. When an order is submitted, we send it off to be invoiced. Our first step is to define a *Queue* of orders to invoice.
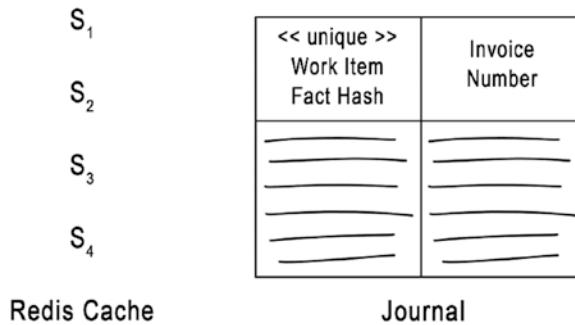
```
query ordersToInvoice(dob: DateOfBusiness) {
  match o: OrderSubmission where o.dateOfBusiness = dob
    such that not exists i: Invoice where i.orderSubmission = o
}
```

With that in place, we create a service that subscribes to this queue. As a service starts up, it generates a random number and inserts a record into a shared Redis cache. When a new OrderSubmission is created, the node that created it broadcasts a notification. The service subscribes to that notification to learn about new work items.

Upon notification, the service runs the query to find work items. It pairs the hash of each work item with the random number of each service in the Redis cache. It hashes this pair to compute the weight of that work item for that service. All of the work items for which the service itself has the highest weight continue to the next step.

The service checks a shared SQL database for a journal entry by that work item's hash. Finding none, it makes the API call and inserts the resulting invoice number into the journal. After that insert succeeds, it creates an Invoice record containing the returned invoice number.

If the service had found an existing journal entry, it would have instead loaded the invoice number and created the Invoice fact without calling the API. And if the insertion failed because of a uniqueness constraint violation, it would abort the processing of that work item after alerting the operations team of a likely duplication. Figure 8-16 shows the pair of artifacts that make up the outbox.

Figure 8-16. *Services share a distributed cache and persistent table to support the Outbox pattern*

Consequences

There is no guaranteed mechanism to prevent duplicate calls to a third-party API. Even with this pattern in place, duplication will occasionally happen. Downstream services should be coded to be idempotent.

Upon service startup, a delay in notifying other services leaves open a window in which both could believe themselves to be the first-place winner. To mitigate against parallel processing in this scenario, delay processing by the new service until enough time has passed for all other services to finish processing any work items in flight.

The fact generated from the results of the API call must not contain any information not captured in the journal. If it contains, for example, auditing information such as the timestamp or IP address when and where it was recorded, then the fact would be different from other facts representing the same API call. The model would contain duplicate data even when the journal prevented duplicate API calls.

Related Patterns

For manual processes, present a *Queue* to the user using a simple query.

The work items in the outbox are typically *Transaction*s.

The query for the outbox usually begins at a *Period*. Overlap periods by significantly greater than the downstream SLA to prevent loss of work items.

Designing from Constraints

The patterns presented here are a starting point for building applications using only immutable historical facts. They emulate—as closely as they can—the behaviors that people have come to expect from business applications. And they do so using only the capabilities of immutable distributed data.

Where these patterns diverge from expected behavior, they reveal constraints about the medium in which they are rendered. A *Mutable Property* cannot have a single value. And we cannot enforce that an entity have *Membership* in only one group. Those truths reveal that the application is distributed across several nodes, each of which has autonomy to capture concurrent changes. Also, we cannot say for sure when a *Period* is closed. We can only assume that enough time has passed to allow distant nodes to connect and share their work items.

We cannot give the users of our applications exactly what they have come to expect from centralized systems. The rules of immutable architecture prohibit it. The reason is simple; those promises *cannot* be kept in a strong eventually consistent manner. Architectures that nonetheless provide these behaviors must compromise some aspect of their distributed nature in order to do so.

An application built according to these patterns acknowledges the constraints imposed by distributed nodes. It starts from those constraints and builds toward expected behavior, never promising more than what can be reasonably delivered.

These patterns are more than guidance on how to build a distributed application; they are a means of communication. They make it possible for application designers to talk to stakeholders about constraints without first teaching them about strong eventual consistency and the CAP Theorem. They permit us to speak in generalities without reasoning through specific scenarios in which distributed nodes might cause us problems. They frame a conversation about application design that helps all participants set expectations and keep them.

PART III

Implementation

Query Inverses

Mathias Verraes' joke about two hard problems in distributed systems is based on a saying by Phil Karlton formerly of Netscape. He quipped "There are only two hard things in Computer Science: cache invalidation and naming things."

I am notoriously bad at naming things (as you have no doubt discovered in earlier chapters). However, I have a rock solid solution to cache invalidation. *Query inverses* determine not only which caches should be invalidated but also precisely how they should be updated.

When we think of caching, we often think of improving performance or scalability. But the most important cache is the one that the user is looking at. User interfaces are caches of query results stored temporarily in view models and browser DOMs. When that cache is invalidated, the user expects their view to automatically update. Failure to update the UI leads to a frustrating user experience. The browser's F5 button and "pull to refresh" are interface metaphors that admit defeat.

On the face of it, you would expect caching of immutable data to be easy. If data doesn't change, the cache is always up-to-date. The point of the cache, however, is not to store a copy of the immutable facts. It's to store a copy of query results. It is query results—not raw facts—that appear on a user interface. And so we must find a strategy for determining when a query's results have been affected.

The results of a query change as new facts are introduced. For each new fact, we must answer two questions:

1. Which caches or UI components are invalidated?

2. What results should be added to or removed from those caches and components?

Query inversion answers one—and sometimes both—of those questions. The views that are affected can always be found by traversing the query from the introduced fact backward. The results to add or remove can usually be given by the tail of the query from

the introduced fact forward. And even when the inverse does not answer the second question, then we simply need to rerun the query to bring the view up-to-date.

Mechanizing the Problem

Determining what to update when certain events take place is the bread and butter of application development. We do it intuitively all the time. Whether responding to user-input events, API commands, or messages in a queue, a developer decides what state is out-of-date. It might be a view model that needs to raise property changed events or an array of results that needs to be flushed and reconstructed. Developers make those decisions.

Intuition, however, fails in two important ways. First, it is easy to miss dependent state that needs to be updated. And second, update logic needs to be revisited as requirements are added. For the first problem, we simply test until we think we've found all of the reasons for a view to change. And for the second problem, we analyze each new feature for how they interact with existing behaviors. We add these as acceptance criteria, modify all affected areas of code, and test for regressions. Because of this, new changes take longer to make and have a greater chance of introducing bugs.

A better solution is to remove cache invalidation and UI update decisions from developer's hands. If the system could decide which caches to rebuild and which user interface components to update all on its own, then that logic would not clutter the code. The decisions can be made mechanically without the risk of human error. And they can be updated automatically as the system grows, keeping each new feature as quick and safe to add as the first one.

The good news is that the system already has enough information to make cache invalidation decisions. Every cache and user interface component is initially populated with a query. That query describes a path through the system's data that yield the results to store or render. By processing the query when it is first executed, the system can determine what new facts will affect the results in the future. All we need to do is invert the query and begin watching for those new facts.

Let's begin by finding a formal vocabulary for describing queries. This will help us not only to execute them but to mechanically invert them.
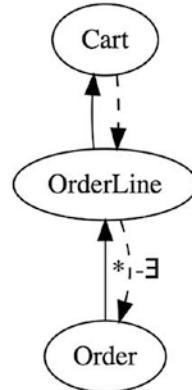
The Anatomy of a Query

A query written in the Factual Modeling Language describes the types and relationships among facts. It gives the conditions under which certain facts should be included. A Factual query is a declarative statement of the results that we seek. This description needs to be translated into something actionable.

A query that we studied in Chapter 5 identified the lines in a cart, excluding those that had been ordered:

```
query linesRemainingInCart(c: Cart) {
  match ol: OrderLine where ol.cart = c
    such that not exists o: Order where o.orderLines = ol
}
```

We overlaid that query on the fact type graph as shown in Figure 9-1.
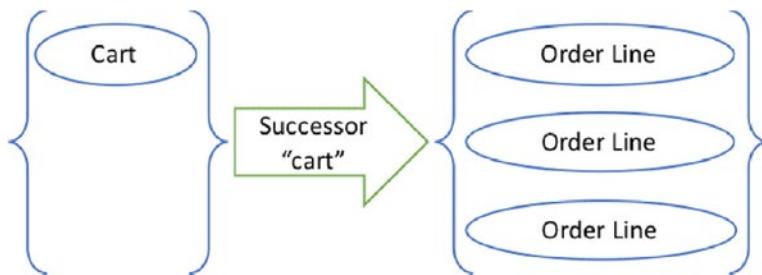
Figure 9-1. *The query visualized as a path over the fact graph*

The dotted arrows indicate the path along which the instance graph will be traversed. It describes how we will execute the query. We can formalize that process as a pipeline. This will be used not only for executing queries but also for computing their inverses.

A Sequence of Steps

A query is a sequence of *steps*. Each step moves from one fact type to another along an edge. The edge is the predecessor/successor relationship between the two facts. Each edge has a *role*: the name given to the predecessor within the successor's type definition. So we can define a step between facts of type A and B as going up to the predecessor or down to the successor along an edge with a given role.

The first step in our example query finds order lines `where ol.cart = c`. This implies a successor step. Given a cart, this step finds all successor order lines in the role "cart". We can draw this step as shown in Figure 9-2.

Figure 9-2. *The first step in the pipeline takes a set of carts to a set of order lines*
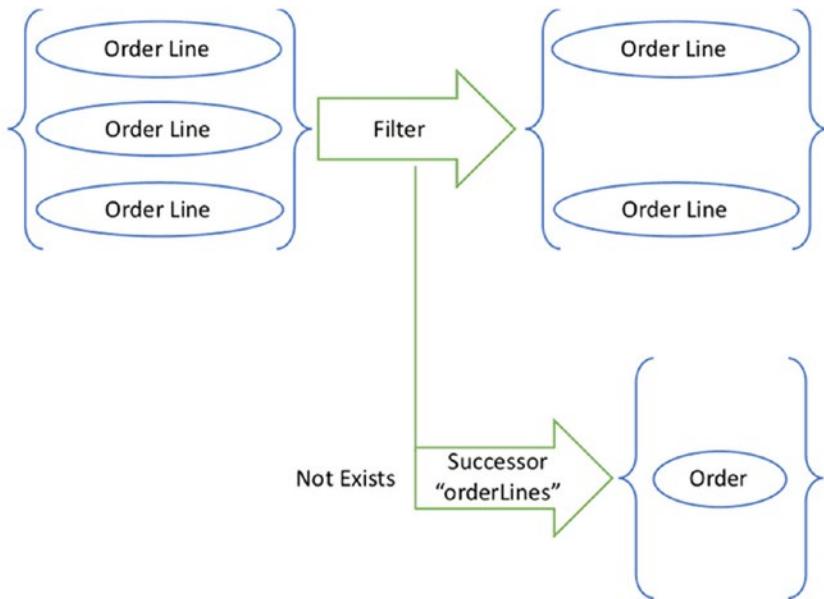
A query is a path from one fact type A to another fact type Z. We will build a pipeline of components, each one taking a set of input facts and producing a set of output facts. The initial set contains only one fact: the starting point of the query.

Filter by Existential Condition

Some components within the pipeline will not be steps at all. They will not move up to a predecessor or down to a successor along an edge. Instead, they will filter the incoming set. Filters are given as a Boolean expression of existential conditions. Each condition's quantifier is either *exists* or *not exists*. The body of the condition is another query, which starts at the fact type where the filter is applied.
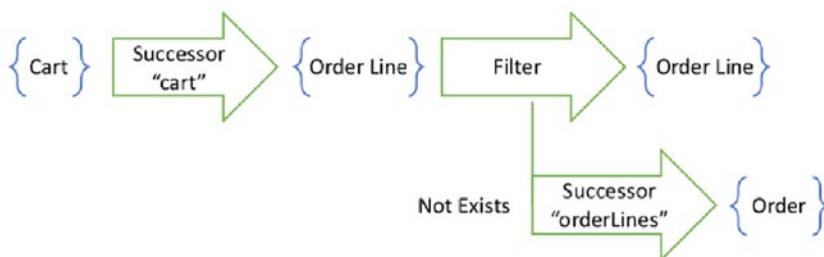
In the example query, we filter out all order lines that are not a part of an order. We accomplish this with an existential condition based on the subquery `where o.orderLines = ol`. It keeps only those order lines where there does not exist a successor order in the role "orderLines". And so we add a filter component that feeds

each fact into a subordinate pipeline. It keeps only those facts for which the subordinate pipeline yields no results. This component is shown in Figure 9-3.

Figure 9-3. *The second step filters order lines based on a subquery*

Joining these two halves into a complete pipeline, as shown in Figure 9-4, gives us a mechanism for computing the query. But it does more. It formalizes the steps that we've so far illustrated with dotted lines overlaid atop fact type diagrams.

Figure 9-4. *A pipeline composed of three operations*
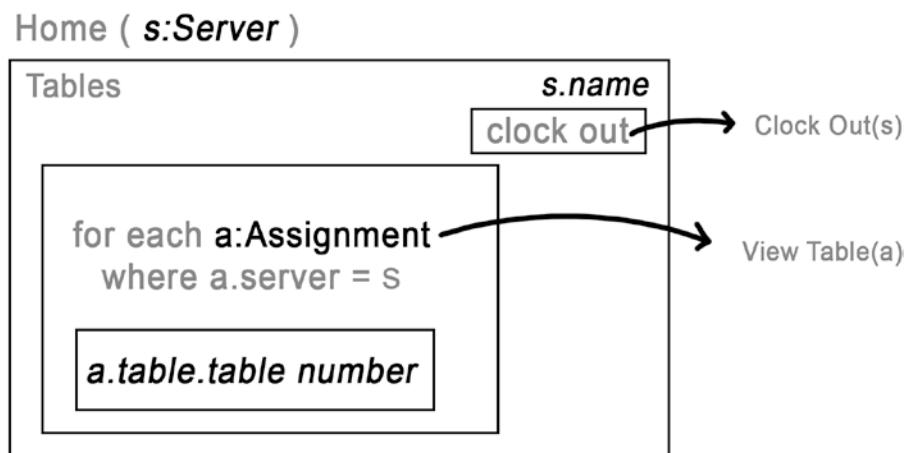
These are the components that we will manipulate to compute the inverse of the query. And it starts by computing the affected set. To understand how to do that, let's return to our familiar zigzag paths.

The Affected Set

The first question that a query inverse answers is which caches and views are affected by a new fact. It identifies those caches and views by the starting point from which the query was first executed.

Every query has a starting point. Any particular cache or user interface component holds the results of a query for a specific starting fact. Different users will be running the same queries, but each will start at a different fact. Hence their caches will be different. As they navigate through the user interface, they will render components by running queries. While many of these components will run the same query, each one will start from a different fact. The starting point of a query determines which user interface component to update, or which cache to invalidate.

In Chapter 5, we showed an example of a view displaying the table assignments for a server at a restaurant. The wireframe appears again for reference in Figure 9-5. Each server will log into the system and bring up the same page. They will see different results because the user interface component uses a query starting from a different fact. When a new `Assignment` fact is introduced, the system computes the affected `Server` to determine which view to update.
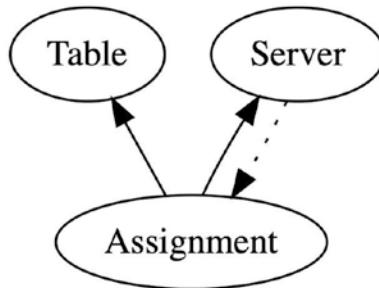
Figure 9-5. *The user interface component showing assigned tables starts at the server fact*

Computing the Affected Set
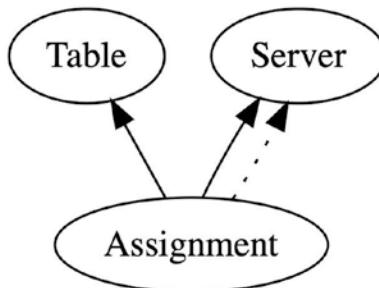
When the table list view first loads, it runs the query shown in the wireframe. That query determines which tables to show in the list. But that query also provides all of the information needed to compute the affected set of any subsequent `Assignment`. That is the role of the inverse.

The original query, when overlaid on top of the restaurant graph, is shown in Figure 9-6. It is a path from the starting `Server` to the successor `Assignment`.

Figure 9-6. *The query for a server's table assignments is a single step to its successor*

The inverse is simply the opposite step, back up from the `Assignment` to the `Server`. When a new `Assignment` is introduced, this query tells us which server's view to update. This matches your intuition and is exactly the same logic you would use if responding to a table assignment event. But as this simple example demonstrates, the process can be automated. The inverse query appears in Figure 9-7.
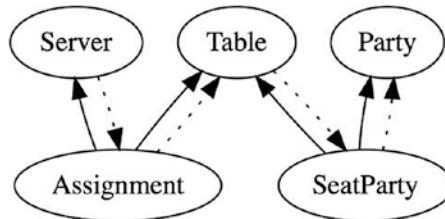
Figure 9-7. *The inverse query simply follows the same step in the opposite direction*

295

Inverting Longer Queries

The preceding example is extremely simple. It demonstrates the process of query inversion, but does not reveal any of the nuance. When we start to examine longer queries, we find that we have a few more considerations.
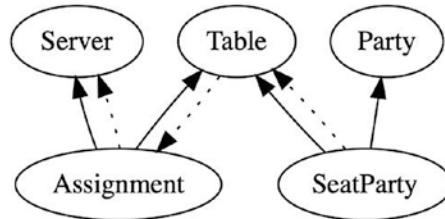
A query with only a single step can only be affected by one kind of introduced fact. It has only one inverse. But a query with multiple steps can be affected if any of them is introduced. Longer queries have more inverses. So the process of query inversion is not simply reversing the chain of steps. It is producing all reversed chains that point back to the starting point.

Consider this more complicated scenario. We render a view that displays all parties that a server needs to take care of. Specifically, this view selects all assignments for the server, then for each table, selects the party that was seated. This multistep query appears in Figure 9-8.

Figure 9-8. *A query of four steps yields all parties seated at a table assigned to a server*

There are several facts that could affect this view. We need to produce an inverse for each step along the way. For example, when a new `SeatParty` fact is introduced, the query in Figure 9-9 determines which server's views are affected.

Figure 9-9. *When a SeatParty fact is introduced, all servers assigned to the table are affected*

Each of the inverses is a query from a different fact type back to a `Server`. All told, this query has four inverses. They start at each of the types `Assignment`, `Table`, `SeatParty`, and `Party`. They get progressively longer as they all terminate at the server.

Unsatisfiable Inverses
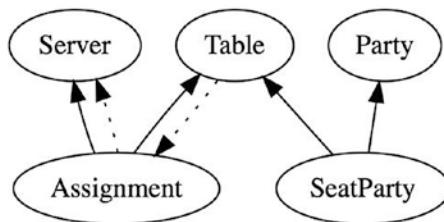
A query inverse is executed when a new fact is introduced. This could be the result of a user action. Or it could be that the new fact arrived from some other device. In either case, the fact that was introduced was not present before and is therefore not represented in the query results. That is, after all, the very reason that we are updating the view.

Knowing that the new fact has just arrived at this node tells us something quite useful. It assures us that the fact currently has no successors.

For a fact to have successors, it must have been introduced before the successors were. A user can only take actions that reference existing facts. And for a peer device to share a fact with this node, it must have first shared all of its predecessors. Said more formally, facts are introduced in topological order.

Because the fact that was just introduced does not yet have any successors in the system, we can be assured that any queries that start with a successor step will yield no results. In particular, any of the inverses that begin by stepping down to a successor will necessarily yield an empty affected set. Such inverses are *unsatisfiable*. They will never find a view that needs to be updated. They can therefore be ignored.

An example of an unsatisfiable inverse appears in Figure 9-10. When a new `Table` fact is introduced, we can be assured that it has not yet been assigned to a server. We can therefore assume that no server's view will be affected by that new table. We will ignore this inverse as it will have no effect.

Figure 9-10. *When a new table is introduced, the query for affected servers begins with a successor step*

Considering this optimization visually, observe the original query as a zigzagging path through the graph. It has hills and valleys as it goes up to successors and down to predecessors. This optimization tells us that we only need to consider inverses that start in valleys. If an inverse starts on a hill, then the first step of the inverse query is down to a successor. Of the four inverses that we computed from the original query, only two are satisfiable: the ones starting from the `Assignment` and from the `SeatParty`.
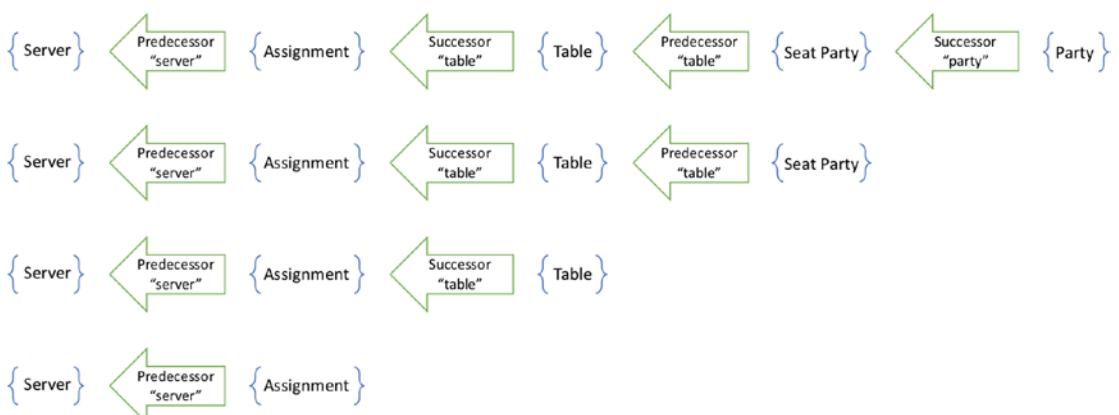
Walking Backward

Let us now express the original query as a pipeline. The pipeline starts with a set of `Servers` (containing only the starting fact). It ends with a set of `Partys`. Along the way, it takes four steps, alternately toward successors and predecessors. The full pipeline appears in Figure 9-11.
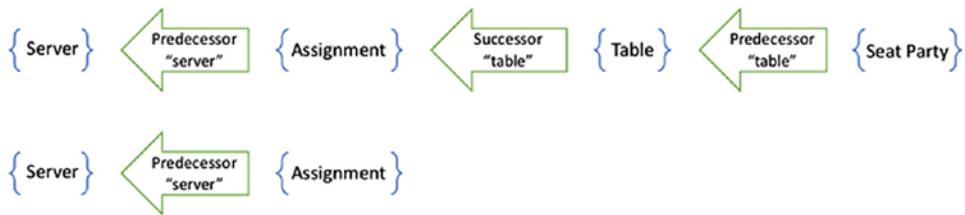
Figure 9-11. *A pipeline from a set of servers to a set of parties that they take care of*

To compute the inverses of this query, we take each subset of the pipeline starting from the `Server`. We will reverse each of the steps in that pipeline. Reversing a step simply exchanges `Successor` for `Predecessor` and vice versa. This operation produces the four pipelines shown in Figure 9-12.

Figure 9-12. *The four inverses of the original query have the opposite steps*

Only two of these inverses are satisfiable. Those are the ones that begin with a Predecessor step. We can ignore the other two. The complete set of satisfiable inverses appears in Figure 9-13.

Figure 9-13. *Only two of the inverses are satisfiable*

To render the party list, we execute the original pipeline starting from the Server. Then, whenever a new SeatParty fact is introduced, we execute the first inverse pipeline in Figure 9-13 to find the affected views. And, whenever a new Assignment fact is introduced, we execute the second inverse pipeline. There are no other events that could affect the view. This mechanical process finds all ways in which the view could be affected.

Proof of Completeness

The claim that there are no other events that could affect the view needs to be defended. It is by no means obvious. Let's take a moment to do so before we proceed.

To begin, we need to define what we mean by "event." An event could be something that the user themselves has done on this node, or it could be information that arrived from some other node. If it happened on this node, it could either be a transient action such as navigation or a permanent action such as saving a new fact. Transient actions will lead to new starting points for queries and will therefore replace old views with new ones. Permanent actions are always analyzed as introductions of new facts. Our analytical tool of Historical Modeling allows no other permanent actions, such as modifications or deletions.

So every event that could affect an existing view would therefore be the introduction of a fact. This might be one that the user introduced themselves or one that arrived from a peer node. In either case, if we were to rerun the query of an affected view, that new fact would have some potential effect on the results.

Our assertion of completeness is equivalent to saying that the inverses find all starting points that could be affected by the introduced fact. To convince ourselves that this is true, we examine the sets of facts that appear along the pipeline. If a fact appears in one of those sets, then it potentially has an effect on the pipeline's results. If it does not, then it could not have an effect.
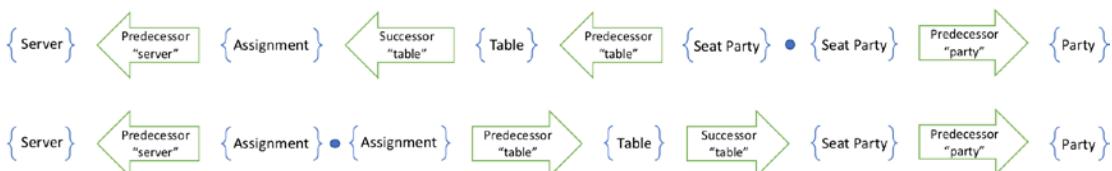
Observe that the reverse of a predecessor or successor step yields a superset of the original starting set. If we start from one set of facts and find all successors in a given role, we are picking out all edges by role and predecessor. Going in the opposite direction, we are picking from among the same edges, but now by role and successor. The edges that produced a result in the original step will be among those included in the reverse step. And so the reverse of a predecessor or successor step will yield all starting facts that could possibly contain a given destination fact.

So far this proof only considers predecessor and successor steps. We will complete the proof by considering filters after we examine existential conditions.

New Results

Knowing which view to update is a great start. Once we know that a view is affected, we can just run its query again from its starting point to find the new state. But we can often do better. Sometimes, the query inverse tells us exactly what new results should be added to the view.

We analyzed the query for parties that a server waits upon. The two inverses that we discovered reverse only part of the original query. Starting from an introduced fact, those parts lead us back to the affected set. The remainder of the query, however, leads us forward to the new results. There is no need for us to run a potentially expensive query. Since we already know the introduced fact, we only need to run part of the query to find the new results. The remainder of each inverted query appears in Figure 9-14.

Figure 9-14. *The remainder of the query (shown after the dot) gives the new results to add to the affected view*

To execute these inverses, start with an introduced fact. Execute the reverse pipeline to find the affected set. Then execute the forward pipeline to find the new results.
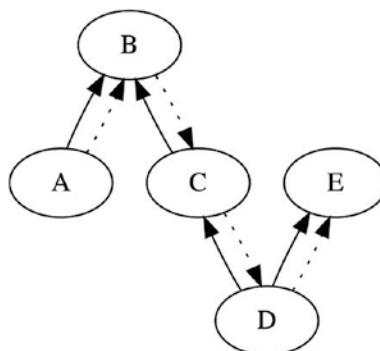
When a party is seated, for example, a new `SeatParty` fact is introduced. Query for all servers assigned to the table at which the party was seated. To each of their views, add the party. And when a new table assignment is introduced, follow the pipeline backward to the affected server. Update that server's view with all parties currently seated at that table.

Your intuition may have led you to write exactly this code. You might have realized that you don't need to rerun the whole query to update the view. But following a mechanical process to automatically produce the optimal algorithm, you now have confidence that no edge cases were missed and no new features will break this code.

Forward Optimization

As before, we know that the introduced fact has no successors. Just as we used this observation to remove unsatisfiable inverses, we can also identify unsatisfiable results. If the forward query begins with a successor join, then the entire inverse can be eliminated. We might be able to find some affected views, but we know that this new fact will introduce no new results.

Take the abstract example in Figure 9-15. We want to render a view starting at fact A that displays information about facts E. The query zigzags through the graph, jumping from predecessor to successor and successor to predecessor.

Figure 9-15. *Some queries might take several successor steps in a row*

Computing the inverse at every point along the path, we find that the reverse pipeline from C starts with a predecessor step. It might at first appear satisfiable. However, observe that the forward pipeline from C to D begins with a successor step. This inverse is therefore *not* satisfiable. Only the inverse starting at D yields new results for an affected view. Inverses always start from the valleys of the fact type graph.
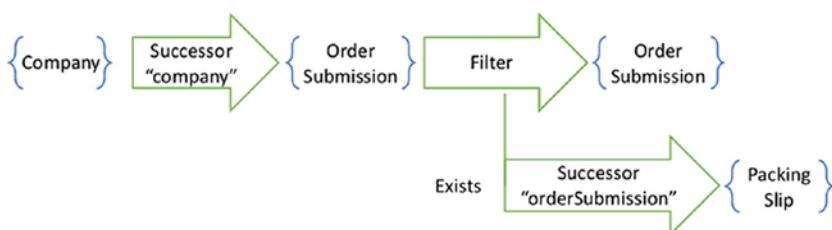
Existential Conditions

So far we've examined simple queries. They are a sequence of steps, each toward a predecessor or successor. However, most queries used to populate a user interface or seed a cache are more complex. They are composed not just of predecessor and successor steps but also of exists and not exists conditions. Let's add these operators to our basic inverses and see how they evolve.

The easier of the two kinds of existential condition is *exists*. We'll leave *not exists* aside for now. This kind of condition filters facts to only those for which a subsequent query yields results. Let us recall an example from Chapter 8.

```
query pickedOrders(c: Company) {
  match o: OrderSubmission where o.company = c
    such that exists ps: PackingSlip where ps.orderSubmission = o
}
```

To begin the process, we convert this Factual query into a pipeline. The clause where o.company = c implies a successor step. Starting at Company, it seeks successor OrderSubmissions in the "company" role. The such that clause begins a subquery. The subquery tests for the existence of a PackingSlip using another successor step: where ps.orderSubmission = o. Chaining these operations together gives us the pipeline in Figure 9-16.
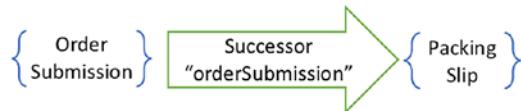
Figure 9-16. *A pipeline having an exists condition*

Once we have this pipeline, we can execute the query. The set of `OrderSubmissions` leaving the filter component will be a subset of those entering. Each of the order submissions is fed into the successor step, and only those that produce a result are kept. Consider, then, what happens to the results when a new `PackingSlip` is introduced. If it brings the number of subquery results from 0 to 1, then it causes its parent `OrderSubmission` to appear in the results.

Recursive Inversion

To break this pipeline down so that it can be inverted mechanically, it is best to apply recursion. Once we reach the *exists* condition, we can push the upper portion of the pipeline onto a stack. This focuses our attention on just the lower portion of the pipeline—the subquery.

Figure 9-17. *The entry point of a subquery is the type on which the filter operates*

We know how to invert this query: a `PackingSlip` affects its predecessor `OrderSubmission` as shown in Figure 9-18. The new result is the packing slip itself. Since the affected pipeline does not start with a successor step (it starts with a predecessor of role "orderSubmission"), we know the affected set is satisfiable. And since the remaining pipeline does not start with a successor step (it is simply the null pipeline), we know that the new results are also satisfiable. And so the `PackingSlip` affects the `OrderSubmission` by introducing a new result.
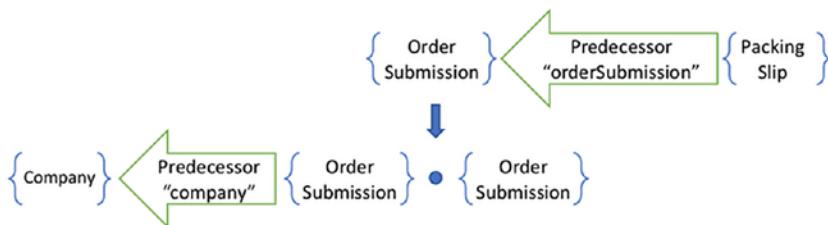
Figure 9-18. *The inverse of the subquery adds the introduced packing slip to the affected order submission*

Popping the stack and returning to the upper portion, we reason that introducing a `PackingSlip` causes the introduction of an `OrderSubmission`. We arrive at this conclusion based on the *exists* quantifier. An order submission without a subsequent packing slip would be filtered out. So introducing the packing slip adds it into the pipeline.

We can pop the stack and continue analyzing the pipeline as if it was the `OrderSubmission` that was introduced instead of the `PackingSlip`. We could, that is, except for one small caveat. Since the `OrderSubmission` was already known to this node, it might possibly have other successors. We therefore cannot apply the valley optimization that we used previously. We must retain the inverse even if the existential condition does not appear in the valley of the graph. The valley optimization only applies to the innermost subquery.

The inverse of this query, therefore, appears in Figure 9-19. We start by walking back from the introduced packing slip to its affected order submission. The members of that set will now satisfy the condition in the parent query. From here, the affected set of the whole query is the reversed pipeline from that condition. And the new results are the remaining forward pipeline.

Figure 9-19. *The inverse starts with the inverse of the subquery and continues as if the affected fact was introduced*
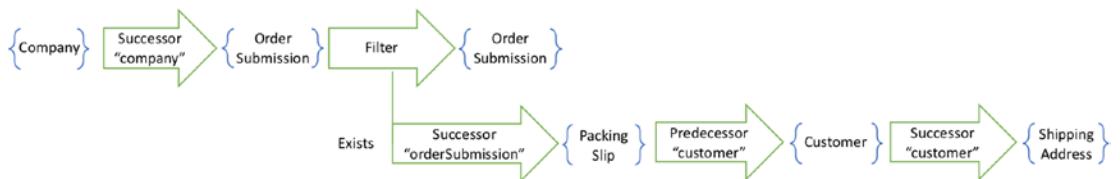
Tail Conditions

In the preceding example, we don't care about the packing slip that was added to the subquery results. All we cared about was the fact that results were added. The inverse of the subquery shown in Figure 9-18 shows the added pipeline after the dot. But we dropped that detail as unimportant in Figure 9-19.

In general, however, that detail matters. The subquery tells us which filter condition might be affected by the introduced fact. It does not, however, guarantee us that the condition is indeed `true`. To have this guarantee, we need to know that the introduced

fact resulted in some actual additions. This is why we will in general want to execute the tail of the subquery. The affected filter condition is only guaranteed to be `true` if the tail yields some results.

Let's add a few more steps to the original query to illustrate what happens. Suppose that we want to know not just that we have a packing slip but also that we have the shipping address of the recipient. The revised query pipeline now looks like the one in Figure 9-20.
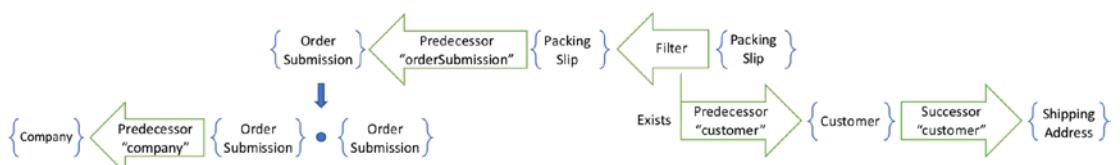
Figure 9-20. *The revised pipeline has two extra steps in the subquery*

As before, we process this pipeline recursively. The subquery has two inverses: one starting at the `PackingSlip` and one starting at the `ShippingAddress`. We will concern ourselves only with the first. That subquery inverse has a nontrivial tail pipeline, as shown in Figure 9-21.

Figure 9-21. *The inverse of the subquery retains the two new steps in its tail*

When we encounter this inverse, we know that it only adds results to the subquery when the tail pipeline produces output. Only then will it make the *exists* condition `true`. To filter the order submissions for which this condition has become `true`, we will turn the tail pipeline into a filter, as demonstrated in Figure 9-22.

Figure 9-22. *The tail of the subquery becomes a condition of a filter*

305

In general, we must always include the tail of an *exists* subquery inverse as a filter condition. In some cases, that filter condition turns out to be a tautology. It will always be satisfied, as in the example featuring a trivial tail pipeline. When the condition is a tautology, the filter can be optimized away.
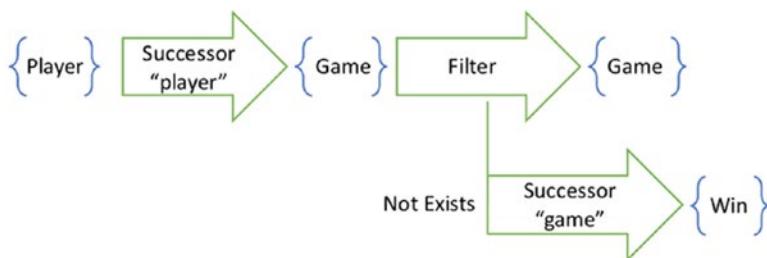
Removing Results

The *exists* clause was the simpler of the two existential quantifiers. It only has the effect of introducing new results. The other clause—*not exists*—allows a new possibility. This clause can remove results. The consequences are far more significant than simply negating a condition.

Not exists clauses are more common in practice than exists clauses. The reason is simple: only a not exists clause can remove results from views. Without them, views would grow unbounded and quickly become unusable. Let's go back to Chapter 3 to one of the first queries that we examined.
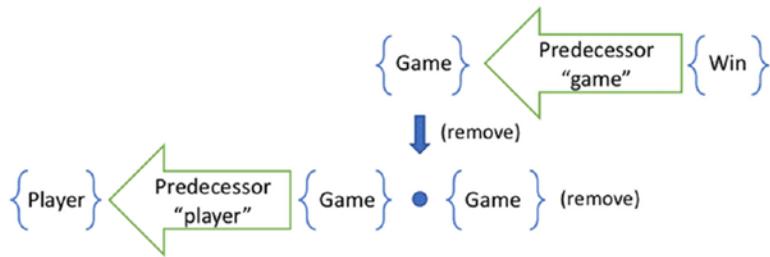
```
query gamesInProgress(p: Player) {
  match g: Game where g.player = p
    such that not exists w: Win where w.game = g
}
```

The pipeline for this query has a now-familiar structure, shown in Figure 9-23. The last time we saw this structure, the filter used the *exists* quantifier. Now it uses the *not exists* quantifier. Executing this pipeline is just as before, except that the condition is negated. The filter preserves only Game facts where the subquery returns *no* results.

Figure 9-23. *A not exists clause filters based on a subquery*

The biggest difference is in the inverse. Introducing a `Win` causes the `Game` to be *removed* from the filtered set. This bubbles up to the outer query and implies that the game is removed from the overall results. As Figure 9-24 illustrates, we must keep track of the effect of each subquery. We need to indicate whether it adds or removes the affected fact.

Figure 9-24. *The inverse of a not exists clause causes the removal of results in the outer pipeline*

When Removal Isn't Removal

An inverse that removes results is significantly different from an inverse that adds them. It's not just a matter of reversing a sign. The difference is in what we can guarantee. We cannot always know that the result that the inverse wants us to remove should truly be removed. It is much more difficult to prove a negative.
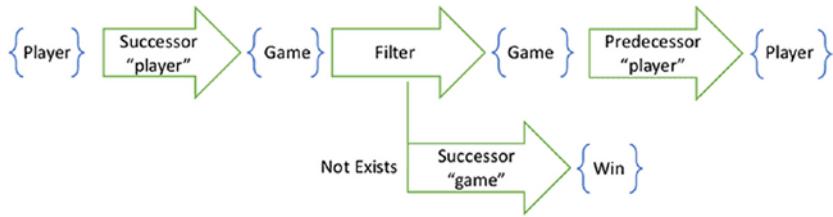
When a result is added to an affected query, we can guarantee that the new result would actually be returned were we to run the query again. We would like to make a similar claim about removed results. We would like to assert that removed results would *not* be returned from the query. Unfortunately, this is not always possible.

Suppose that the query did not look for games in progress, but rather for their opponents. The Factual query in that scenario would be this:

```
query opponentsInGamesInProgress(p: Player) {
  match g: Game where g.player = p
    such that not exists w: Win where w.game = g
  then o: Player where g.player = o
}
```
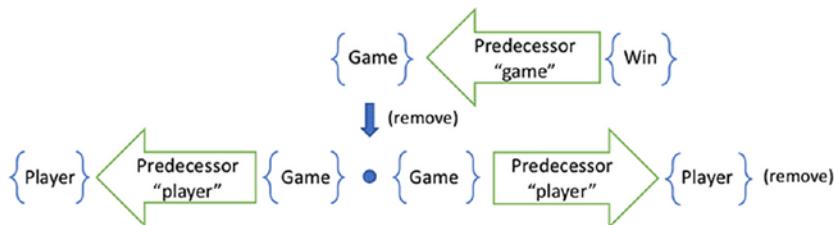
The pipeline for this query looks like the previous one, with one additional step. After filtering the games, it takes a predecessor step toward the player, as shown in Figure 9-25.

In this particular case, that will include the starting player, too, but that's not the issue that concerns us. We can easily filter out the starting player to render a meaningful view.

Figure 9-25. *A pipeline for finding current opponents takes an additional predecessor step to the player*

The issue that concerns us is that the inverse has an nontrivial "remove" pipeline. It takes a step that the inverse of the previous query did not. We can see that additional predecessor step in Figure 9-26.

Figure 9-26. *The inverse removes the player of the removed game*

This additional step would have us remove the opponent of a game that just ended. When the Win is introduced, the Game cascades down and removes the opponent from the affected view. However, it is possible that another game with this same opponent is in progress. There is more than one way to traverse the graph to get to this opponent. Removing one of these paths does not guarantee that no other paths exist.

The previous query was safe. We could guarantee that any game will be removed from the cache when a successive win was introduced. No matter how we traversed the graph, the filter was applied to the game itself. But if the pipeline extends *beyond* the filter, then we can no longer make that guarantee. If the remove set is a nontrivial pipeline, then we have to check all possible paths.

In situations like this, a removal cannot guarantee that the removed result is not still a valid result by some other path. As a consequence, we must run the original query again. The affected set tells us which caches to invalidate. But the remove set does not tell us which results to remove. At best, it gives us a set of candidates that we can check.
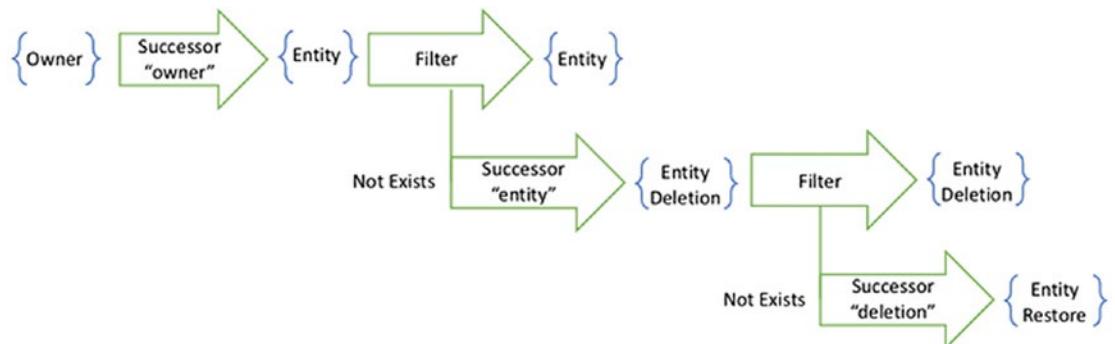
Nested Subqueries

As we process nested subqueries, we will need to keep track of whether they add or remove results. It is not uncommon for a *not exists* subquery to have a *not exist* subquery of its own. When this occurs, introducing a fact to the innermost subquery will *add* a result to the view. Each recursive descent into a *not exists* subquery flips the direction of the effect.

A typical example of a nested subquery using *not exists* clause is the *Restore* pattern found in Chapter 8. Under this pattern, introduction of an EntityDeletion fact removes an entity from query results, and a subsequent introduction of an EntityRestore fact reinstates it.

```
query entitiesInOwner(o: Owner) {
  match e: Entity where e.owner = o
    such that not exists ed: EntityDeletion where ed.entity = e
      such that not exists er: EntityRestore where er.deletion = ed
}
```

The query becomes a pipeline with nested filter steps, as shown in Figure 9-27. Both of these steps apply a NotExists quantifier.
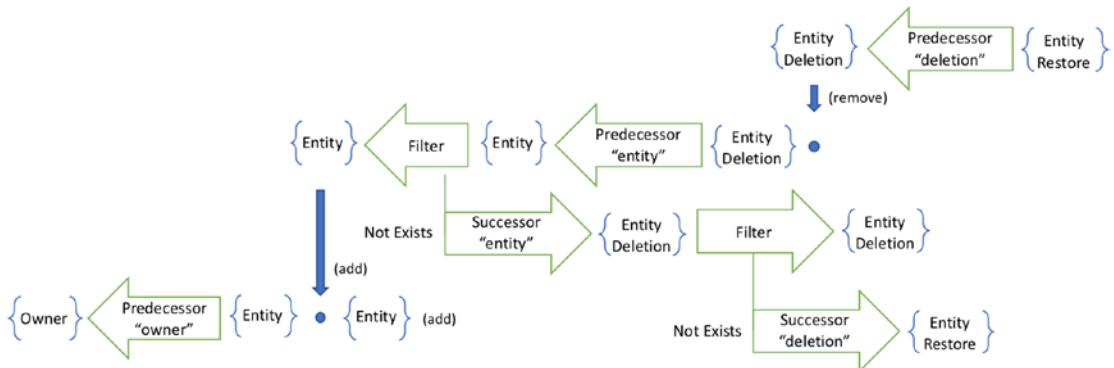
Figure 9-27. *The restore pattern yields a pipeline with nested not exists filters*

To find the inverses of this query, we traverse the pipeline recursively. After descending two levels, we invert the innermost query. This tells us that introducing an EntityRestore fact affects the predecessor EntityDeletion. It has the effect of removing that EntityDeletion. This is an effect that we can guarantee, since we have inverted a *not exists* clause and can prove that the successor exists.

Popping the stack, we then determine the effect of removing the `EntityDeletion`. The inverse tells us the affected set of `Entity` facts. Removing the entity deletion would ideally have the effect of *adding* the entity. However, this is not an effect we can guarantee.

There might have been another entity deletion. Sure, we can prove that we removed one of them. But we have not yet proven that we've removed all of them. And so, before we identify the added entities, we rerun the original filter condition. The full inverse is shown in Figure 9-28.

Figure 9-28. *In inverse of two nested not exists clauses is a pipeline that causes the addition of facts*

In general, we can guarantee the effect of an inverse based on adding a fact. The innermost subquery is always of this form, as it captures the introduction of a new fact. But any inverse based on removing a fact cannot be guaranteed. There may be other paths through the graph that keep the condition as it is. Therefore, we must always append a filter testing the original condition to the inverse.
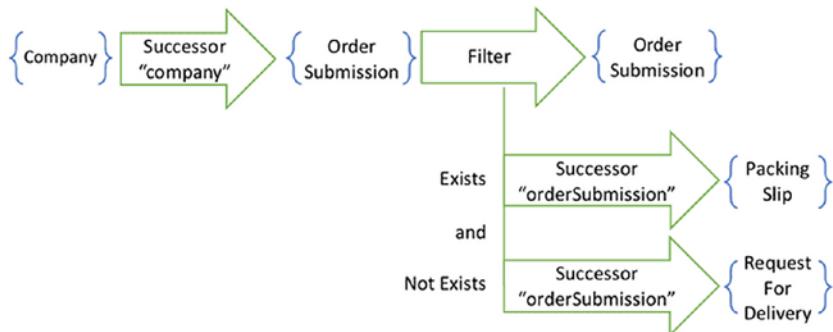
Tautological Conditions

In the previous examples of inverses, I left off an important detail. The inverse must preserve the existential condition of the original query. I left it off both for clarity and because I chose examples in which it didn't matter. In each of the previous examples, the existential condition is guaranteed to be true under the circumstances in which it was applied. A logical predicate that is always true is called a *tautology*. When we recognize tautologies within inverses, we can optimize them away.

The first example of an existential condition that we examined was the query for picked orders. This example was incomplete. The full query from Chapter 8 included an additional clause looking for a RequestForDelivery. This clause eliminated drop-shipped orders.
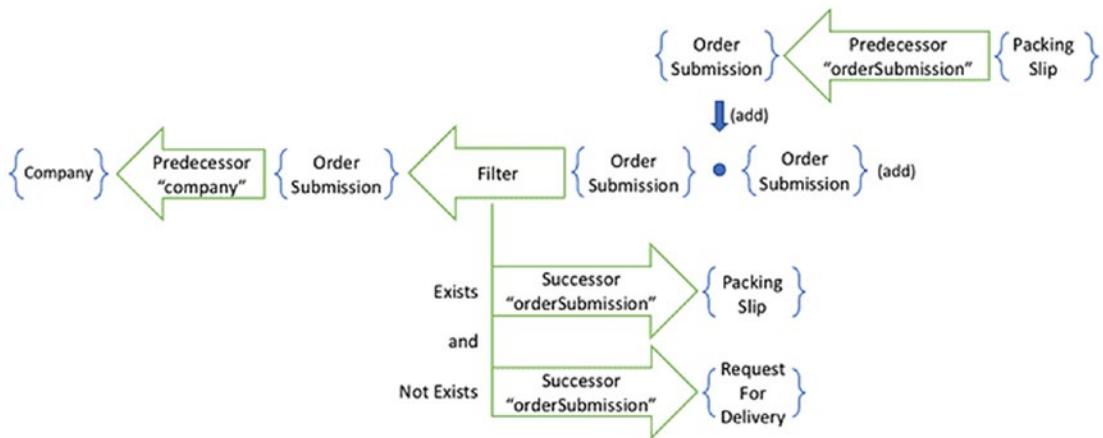
```
query pickedOrdersToShip(c: Company) {
  match o: OrderSubmission where o.company = c
    such that exists ps: PackingSlip where ps.orderSubmission = o
    and not exists rd: RequestForDelivery where rd.orderSubmission = o
}
```

While a human analyst understands the motivation of the query, a mechanical query inverter does not. Rather than relying upon our intuition about drop-shipped orders, let's invert this query mechanically. This time, we will preserve the conditions within the inverse and look for tautologies. We begin by constructing the pipeline from the original query, as shown in Figure 9-29.
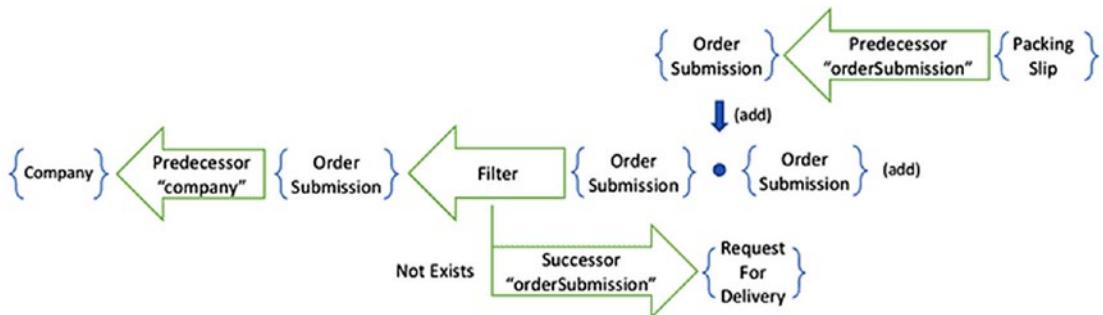
Figure 9-29. *A query with a compound condition defines a pipeline with two subqueries*

This pipeline has several inverses, which a mechanical inverter would produce by recursively walking each of the steps in each subquery. As before, we will focus on only one inverse: the one that describes the response to introducing a PackingSlip. This time, however, we will preserve the filter within the inverse. The full inverse pipeline appears in Figure 9-30.

Figure 9-30. *The inverse pipeline retains the existential conditions of the original query*

At this point, we can optimize the pipeline. We know that the pipeline begins with the introduction of a packing slip. The order submissions that cascade down into the second line have a successor packing slip. We therefore know that the first existential query must be satisfied. There certainly exists a successor packing slip in the "orderSubmission" role. This was precisely the packing slip that initiated the process. We can therefore eliminate this condition from the filter, as shown in Figure 9-31.

Figure 9-31. *We can eliminate tautological conditions from a filter*

The pipeline gives us no knowledge about the remaining condition. We must still, therefore, verify it before we add the resulting order submission. The first time that we examined this inverse, the RequestForDelivery clause was not in the pipeline. We therefore ended up with a filter having only a tautological condition. We optimized away the filter, as it would have no effect.

The resulting inverse implies a procedure. When a packing slip is introduced, add the order submission to the cached results, but only if it is not drop-shipped. As a human analyst, one might ask "why would we create a packing slip for a drop-shipped order?" Doing so would be absurd. This observation might lead a human designer to eliminate this check. But a mechanical inverter has no such bias. It will produce code that is logically correct even if it makes no business sense. And as a result, it will work correctly in nonsensical edge cases where a solution based upon intuition would fail.

In general, we will find two scenarios by which a condition is a tautology. First, we will have *exists* clauses seeking facts that were just introduced. And second, we will have *not exists* clauses seeking successors of newly introduced facts. The one we just examined is an example of the first. The second is an extension of the valley optimization. We eliminated inverses that began outside of a valley. Those inverses began with successor steps, either in their affected set or their new results. By the same token, we can consider any *not exists* clauses at the very head of the inverse beginning with a successor step to be a tautology.

When we reverse the *not* on the two general cases mentioned earlier, we find a different optimization. The opposite of a tautological predicate is an *unsatisfiable* predicate. It is possible that an inverse will contain a *not exists* clause seeking introduced facts. Given that the inverse fires only when those facts are introduced, we know these conditions to be false. It is also possible that an inverse will contain an *exists* clause for successors of introduced facts. Again, since these facts were newly introduced, these conditions must be false. Unsatisfiable conditions within an *or* statement can be eliminated; those within an *and* statement make the whole statement unsatisfiable. A mechanical query inverter will optimize away these statements as well and even optimize away the entire inverse if the filter condition is deemed unsatisfiable.

Proof of Completeness Continued

Now that we have a better understanding of the effect of filters on the query inverse, we can finish the proof of completeness. We've already shown that an inverse made up only of successor and predecessor steps does not exclude any possible affected starting points. We will now show the same for inverses containing filters.

A predecessor or successor step has the property that the inverse produces a superset of the original starting set. We relied on this to show that all starting points that would include the introduced fact would be in this superset. Filter components do not

have this property. A filter in either direction yields a subset of the starting set. For filters, we must consider whether we are adding or removing a fact.

If we are adding a fact—such as for the innermost subquery responding to the introduced fact—then we can assert that the added fact would only have an effect on the query if it passed through the filter in the original pipeline. If it was filtered out, then it would have no downstream effect. Therefore, applying the filter in the opposite direction, even though it produces a subset, produces precisely the subset that would truly be affected.

If we are removing a fact, on the other hand, this logic does not hold. We are seeking an affected set of starting points whose pipelines *used to* contain the removed fact, but *no longer do*. Therefore, a filter would exclude precisely the starting points that would be affected by the removed fact. Reversing the filter does not solve the problem. The affected set could be filtered for more than one reason, and any of the remaining filters might incorrectly exclude it.

The solution, then, is to remove any filters from the affected set of remove inverses. The only filter that these inverses should have is the one at the end that verifies that the filter condition is indeed `false` for these affected starting points. Any other filter would reduce the affected set and potentially miss effects.

Potential vs. Actual Change

When reasoning through the effect of existential conditions, we made an assumption. We assumed that the effect of a subquery inverse would always be to change the filter condition from `false` to `true` and thus add the result or from `true` to `false` and thus remove it. Now that all of the complexity of subquery inverses, tautologies, and unsatisfiable predicates are on the table, we can acknowledge that that assumption is not in fact true. Once we do so, we can then see what simple extra step we need to take to defend against those cases where it is false.

We first introduced the assumption while examining the *exists* clause. There, we said that if the introduced fact brings the number of subquery results from 0 to 1, then the condition becomes `true`, and new results are added. We then proceeded to find inverses of the subquery, which we can safely assert will increase the number of subquery results. However, we had no evidence that they brought the number of results from 0 to 1. If the subquery already had results, then this new one would have no appreciable effect. The filter condition is already satisfied.

By continuing as if the new fact brought the number of subquery results from 0 to 1, we made the assumption that the new results it adds were not already results of the original query. Because this assumption is false, we might find ourselves adding results to a cache or view that are already present. And so, the consequence of making this assumption for an *exists* clause is duplication.

Removing Absent Results

A similar assumption arose unstated while we examined the *not exists* clause. Here, we proceeded to evaluate inverses of the condition under the assumption that any new fact would bring the number of results from 0 to 1. When viewed through the *not exists* clause, this would change the filter condition from `true` to `false` and therefore remove results. But if the subquery already had results, one more would not make the filter condition any more false. The effect of running an inverse in these circumstances would be to attempt to remove a result that was not in the cache.

There is an easy solution to this problem, and a hard one. Let me first describe the hard one, as it is interesting, but ultimately pointless. Because the graph of facts is immutable, introducing a new fact does not destroy information. We still have the capability of seeing the graph as it was before the fact was introduced. We could therefore compute the filter condition as it was just before the introduction of the fact that triggered the inverse.

If we run the filter condition—whether an *exists* or *not exists* clause—while ignoring the new fact, we can see if it used to be `false`. If so, we know that the introduction of the new fact was precisely what made it `true`. We would know that the subquery set changed from 0 to 1, and therefore the effect is real.

This solution is interesting because it is a specific example of a powerful analysis technique. It demonstrates that we can answer hypothetical "what if" questions by running queries while ignoring subsets of the directed acyclic graph. These questions can provide insights as to how the system appears on different nodes. They give us a powerful "time travel debugging" capability popularized by frameworks like Elm and Redux. But alas, for solving the problems of cache duplication and removing absent results, there is a much simpler solution.

Caches Are Sets

The results of a Factual query are always sets. They do not impose an order, and they do not permit duplicates. Therefore, any cache or view seeded by a query will have at its core a set. It might further apply an order on top of that set or project the set through some aggregate function, but what it started with was an unordered set of distinct facts.

As long as a cache or view retains the set of facts, it can easily protect against the assumption that every potential change is an actual change. If an inverse reports that an introduced fact adds a set of results, the cache simply performs the set union. It ignores any results that are already in the set. It only adds the ones that are truly new. Similarly, to process an inverse that removes results, the cache performs a set difference. It only removes results that were actually in the set to begin with.

It is tempting when creating a view to simply keep track of the projected DOM, XAML, or other visual elements. And if users of a cache are only interested in a mutable projection of some facts, not the raw immutable results themselves, then it stands to reason that the mutable projection is all that the cache needs to store. However, the simplest solution to the cache invalidation problem is to keep the set that leads to the projection. In response to a query inverse, perform a set union or difference. And then project the part that has changed.

Query Inversion in Practice

As you've just witnessed, query inversion is a complicated business. There are several edge cases, scenarios, and optimizations to consider. Fortunately, the process can be automated. Developers need not compute the inverse of a query by hand.

The mechanical process of query inversion is guaranteed to produce an affected set that identifies all possible invalid caches. Furthermore, it often produces an add or remove set that describes exactly what set union or difference to compute. But even when the inverse is too complex to produce such exact instructions, running the query again will produce the correct results.

I have produced a couple of implementations of the query inversion algorithm in the past. The first was for an open source project called Correspondence. The second was for a more recent open source project called Jinaga. Both of these projects derived inverses from the developer's original queries to determine which views to update as new facts are introduced. Their behavior was surprising to me, even knowing the math behind

them. And I found that my confidence and productivity were greatly improved by having a query inverter working on my behalf.

Following publication of this book, I will maintain an open source reference implementation of the query inversion algorithm. With the help of readers, I may find edge cases and optimizations that I missed in this chapter. Those improvements will become part of the reference implementation. Any projects wishing to use query inversion can learn from that implementation and compare results as a form of automated testing. You may find a link to that reference implementation at the book's website on Apress, at at `immutablearchitecture.com`.

SQL Databases

Immutable models are all about constraints. An application must not modify data. It is disallowed from deleting or overwriting information. These constraints are the axioms upon which the mathematical structure of immutable architecture is built. Ensuring that these constraints are met is the responsibility of every layer of the application stack, right down to the data storage system.

An immutable model can be backed by any kind of data store. One of the most powerful, flexible, and popular forms of storage engine is the relational database. Translating the constraints of the immutable model into a relational database takes discipline. We must avoid some of the capabilities that relational databases permit. To begin with, we must prohibit the use of UPDATE and DELETE. These two operations are part of the *Data Manipulation Language* (DML). Only the nondestructive DML command—INSERT—may be used in an immutable data store.

Limiting DML is a good start, but there is more we must do to store an immutable model in a relational database. We've defined queries against an immutable model according to certain constraints. They must start at a known fact. They can only traverse the graph along predecessor/successor relationships. And they allow only existential conditions; data cannot be filtered by value. These constraints restrict our use of *Data Query Language* (DQL). DQL includes the ever powerful and composable SELECT, WHERE, and JOIN keywords. When storing an immutable model in a relational database, however, we must restrict ourselves to a subset of patterns.

To work best with this subset of DQL, we will also constrain our use of the *Data Definition Language* (DDL). DDL statements CREATE TABLE, INDEX, and other objects. We will discover the subset of DDL that works best to define a schema for immutable records.

With all of these constraints, it would seem that we are losing the power and flexibility of relational databases. But in truth, we are gaining much more in exchange. The subset to which we limit ourselves will produce an extremely efficient data access layer. And where efficiency can be improved, we will find repeatable patterns of

optimization. The steps we follow to produce DDL, DML, and DQL will be extremely mechanical. It will be so predictable that we will even be able to automate some or all of our SQL code generation. Taken to the extreme, we will even find a way to move a data model fluidly from one store to another, deploy new code without evolving the database schema, and even tackle nonlinear data versioning.

To begin, we will learn how to write DDL to define a relational schema that stores an immutable model. The subset that we will use is carefully chosen to meet the constraints of immutable architecture. Designing to those constraints may at first appear to take capabilities away. But as we will see, it truly gives us capabilities that we could never have had before.

Identity

When we first introduced historical records as application data, we learned that a record is uniquely identified by the values of its fields and the identity of its predecessors. While this is a useful definition of identity for proving theorems about idempotence and convergence of histories, it is not very useful for database design. If every row were identified by the sum of its columns, then a foreign key would be a full copy of the parent table. This would not be at all practical.

A relational database requires that we identify historical records using some form of surrogate key. The most natural choice for most relational database systems is an auto-incremented ID. They provide the most optimal storage efficiency and query performance. They do, however, have the disadvantage of being location specific, as discussed at length in Chapter 4. We will therefore outline a mechanism for using auto-incremented IDs for their internal advantages, but map them to content-based IDs to overcome the disadvantages.

Content-Addressed Storage

Conceptually, the content of a historical record *is* its identity. Practically, however, we need a stand-in for the content. The solution is to derive an identifier from the content using a hash function. This practice is known as *content-addressed storage*. This approach only works for content that does not change. And so it is an appropriate mechanism for identifying immutable records.

At first glance, this seems like an oxymoron. To load a record, you must know its address. Yet to find its address, you must first know its contents. It seems like you could never get started. In practice, however, it is not difficult at all to find a starting point. If someone is writing a record, then they know its contents. But if someone is reading a record, then they are given its hash. The starting point simply changes based on whether you are reading or writing.

Consider a blockchain on which a transaction needs to refer to a digital document. Space on the blockchain itself is far too expensive, which prohibits us from storing the document within the transaction. Instead, we store the hash. The actual document will be stored in a much less expensive distributed database, such as the InterPlanetary File System (IPFS).[1]

The author of the document has the original contents and can therefore compute the hash. Since the document is immutable, this hash will never change and can therefore be a reasonable surrogate for the document's identity. A reader of the transaction, on the other hand, does not have the original document. They cannot compute the hash to determine the identity. That's OK, however, because they can read the identity from the transaction.

Looking up a document in the distributed file system is a matter of fetching an object by its hash. Once retrieved, the reader can verify the identity of the document by computing its hash. Insofar as they can trust that it would be difficult to construct a fraudulent document with the same hash, they can trust that this immutable document has not been tampered with.

Advantages

In addition to the tamper-resistant nature of using a document's hash as its identity, we have several other benefits. Some of these were already covered in Chapter 4. Others will be new. Let's take some time, now that we have an understanding of how to model a system using immutable records, to evaluate these advantages.

As noted with the example of a document in a distributed file system, the writer computes the hash prior to storing the record. This implies that even before communicating with another node, the author of the record knows its identity. There is no need for the node to communicate with some external service before learning the

---

[1] https://ipfs.io

identity. This gives it the autonomy to continue working, even creating other related records and referring to that identity, before it needs to connect to a peer.

There is also the advantage that every node will compute the same identity for the same record. This provides a natural de-duplication benefit. Imagine if your photos were identified by the hash of their contents, rather than the folder that they occupy and the filename that the camera gave them. If that were the case, then you could never accidentally create duplicate photos by re-importing from digital media or sharing back and forth with friends. The same advantage exists in business applications. Preventing duplicates provides idempotence and avoids doubling the effect of a transaction.

Finally, consider the problem of merging data from disparate nodes. If the identifier of those records was based on anything other than their contents, then two things could go wrong. First, it would be possible for two records with the same contents to have different identifiers, each allocated by a different node. And second, two distinct records could coincidentally share a common identifier. Both of these problems make it difficult to determine where the data structures intersect and where they diverge.

When using content-addressed storage, every node computes identity in exactly the same way. If two records share an ID, they are the same record. And if they are the same record, they have the same ID. When any other mechanism is used, then IDs have to be re-mapped as data moves from one identifier space to another.

Hash Collisions

When a hash is used as a stand-in for the contents of a record, one important concern must be addressed: hash collisions. The identity of a record is unique. Its identifier must be similarly distinct from the identifiers of other records. If the contents of two different records produced the same identifier, then references to one would be confused for references to the other. Given the pigeonhole principle, we know for certain that collisions are possible for any record larger than the hash size. The question, then, is whether any of those other records producing the same hash will ever arise in practice.

One possibility is that a nefarious actor will *deliberately* construct such a record, which would allow them to forge signatures and replace documents. To combat this, we rely only upon cryptographically strong hashes, which are designed and continuously tested to prevent such activity. Cryptographers have constructed ever sophisticated hashing algorithms with ever-increasing hash sizes in an arms race against these attacks.

Another possibility is that a collision will happen by mere *chance*. In a data set with n records, there are $n(n-1)/2$ ways to pair them. We must allow a hash to have enough

possible values to limit the probability that *any one of these pairs* has the same hash. Simplifying a bit, the number of pairs is almost $n^2/2$, so the number of possible hashes h must be sufficiently larger than that. The probability of a collision is very closely approximated by the following equation:

$$p = 1 - e^{-n^2/2h}$$

Using this approximation, we can determine the risk of a collision. Let the number of bits in *n* be *s* and the number of bits in *h* be *b*. This lets us compute the number of bits in the odds of a collision. The probability of a collision $p = 1 / 2^q$, where *q* is given by

q = b - 2s + 1

See the sidebar for the derivation of this formula. For example, if we have a data set with a billion records, and we use a SHA-1 hash, we can compute the probability of a collision *anywhere within the data set* as follows:

1 billion records $\approx 2^{30}$

hash size = 160 bits

Probability of collision $\approx$ 1 in $2^{160-2\cdot30+1}$ = 1 in 2^{101}

So with a SHA-1 hash, the probability of a collision anywhere in a 1 billion record data set is about 1 in 10^{30}. If we improve the hash to SHA-2 with 256 bits, the probability plummets to 1 in 10^{59}. For any realistic data set, this risk is vanishingly small.

PROBABILITY OF A HASH COLLISION

Starting with the approximation of the probability of a hash collision, we can simplify

$$p = 1 - e^{-\frac{n^2}{2h}}$$

$$1 - p = e^{-n^2/2h}$$

$$ln(1-p) = -n^2 / 2h$$

Given that we are only considering scenarios in which the probability is close to zero, we can apply the small value approximation that $ln(1 + x) = x$:

$$p = n^2/2h$$

Replacing the quantities with their power-of-two definitions, we get

$$2^{-q} = (2^s)^2/2 \cdot 2^b$$

$$2^{-q} = 2^{2s}/2^{b+1}$$

$$2^{-q} = 2^{2s-b-1}$$

$$-q = 2s - b - 1$$

$$q = b - 2s + 1$$

This formula tells us that for every doubling of the data set, we need to increase the hash size by 2 bits in order to maintain the same low collision probability.

Avoid Hashes As Primary Keys

Hashes are fantastic identifiers. However, they are poor primary keys. The most obvious reason is size. Most modern computer systems have a natural word size of 64 bits. Relational database management systems take advantage of that word size for their native types and data structures. Hashes, on the other hand, are typically 160, 256, or 512 bits, depending upon the algorithm chosen. Larger identifiers can be noticeably slower and more cumbersome to manage.

Large primary keys can lead to bloated logs, as log entries must use the entire key to refer to rows. This impacts not only storage but also performance. Committing a transaction involves writing the log stream. Replication requires shipping it to other machines. Both of these operations take time.

Large identifiers will also bloat indexes. Scanning an index with larger identifiers means loading more pages to find the data that you want. Rebuilding indexes takes longer. And query operations cannot take place at the machine's native word boundary. The result is slower overall query performance.

Finally, most relational database systems will use the primary key as a clustered index. This means that the physical storage of the records will be sequenced according to this key. The most optimal way to allocate space for the next record is to append it to the block. The least optimal way is to insert it randomly somewhere in the middle. Hashes appear random. Using a hash as a clustered index will cause fragmentation and frequent page splitting.

Table Structure

We can apply this knowledge to constrain the Data Definition Language to produce well-behaved immutable models. To get the advantages of content-addressed storage while retaining the efficiency of auto-incremented IDs, we will store both. The auto-incremented ID will be the primary key of the table. It's value, however, will never leave the database. To the outside world, it will appear as if each record is identified only by its hash. We will impose a uniqueness constraint on the hash so that we have the necessary duplication protection.

Since a hash is larger than the word size of the computer, most database engines do not map them to native data types. Instead, they are typically stored in binary or text columns. If stored as text, they are base-64 encoded. This makes them easier to view as a human operator, but increases the storage requirements by ⅓rd. In the following examples, we will use the text option, as that is more portable. But please consider whether binary would be better for your application.

Let's recall one of the very first historical facts that we documented. This was a Catalog fact having nothing other than a natural key to distinguish it from other catalogs. We will convert this into a SQL table.

```
fact Catalog {
  referenceNumber: string
}
```

The table for a fact will contain the primary key, the hash, and a column for each immutable field. This example has only one field, so the resulting table has just three columns. This is how the table would be defined in PostgreSQL.

```
CREATE TABLE catalog (
  catalog_id SERIAL PRIMARY KEY,
  catalog_hash VARCHAR(88) NOT NULL UNIQUE,
  reference_number VARCHAR(50) NOT NULL
);
```

The reason for choosing 88 characters is that this application uses a 512-bit SHA-2 hash. When encoded in base-64, a 512-bit number takes 88 ASCII characters. To find this value for any hash size, divide the number of bits by 24, round up, and then multiply by 4.

It is tempting to define a uniqueness constraint covering the fields of the table. We know that the values of those fields identify the fact, and so no other fact can have those distinct values. In the preceding example, we might want to define a uniqueness constraint on the reference number. While this would work for this specific example, it breaks down in practice. We will see this more clearly when examining predecessors with cardinality *many*. In all cases, the uniqueness constraint on the hash is serving the same purpose. We will therefore rely upon that constraint alone.

To compute the hash of a record, find a canonical form. I prefer to use JSON with the fields sorted alphabetically and all whitespace removed. Compute the hash of the UTF-8 encoded string representation.

```
$ echo -n '{"reference_number":"AX247"}' | \
> openssl dgst -sha512 -binary | \
> base64
8rCOhVD...ifdw==
```

This table structure permits insertion without the need to wait for the primary key to be generated. The caller already knows the hash of the record, and that is all that is required to uniquely identify the row. When the catalog_id is needed later, it can be retrieved by catalog_hash.

If the application attempts to insert a duplicate record, the uniqueness constraint will prevent it. There is no reason to treat this as an error. The application has simply confirmed that the record had been stored. Most relational database engines will allow you to ignore a uniqueness constraint violation. In PostgreSQL, for example, use the clause ON CONFLICT DO NOTHING.

```
INSERT INTO catalog
  (catalog_hash, reference_number)
  VALUES ('8rCOhVD...ifdw==', 'AX247')
  ON CONFLICT DO NOTHING;
```

Relationships

When we translate predecessor relationships into relational tables, they become foreign keys. To gain the space and performance advantages of integers, the foreign key will reference the auto-incremented ID of the predecessor, not the hash. This keeps our indexes small and fast. But these foreign keys do not leave the database.

Predecessor relationships come in three cardinalities: *one*, *optional* (0 or 1), and *many* (0 or more). If the relationship allows exactly one predecessor, then we represent it directly as a foreign key in the fact table. For example, we can translate a product residing in a catalog into a table. The Factual definition of the product is as follows:

```
fact Product {
  catalog: Catalog
  sku: string
}
```

The resulting table has an auto-incremented ID, a hash, the fields, and foreign keys for predecessors. Be sure to create an index on the foreign key, as most relational database systems will not do this automatically.

```
CREATE TABLE product (
  product_id SERIAL PRIMARY KEY,
  product_hash VARCHAR(88) NOT NULL UNIQUE,
  catalog_id INT NOT NULL REFERENCES catalog,
  sku VARCHAR(50) NOT NULL
);

CREATE INDEX product_catalog
  ON product (catalog_id);
```

Inserting Successors

Now with the schema defined, we can determine how to best structure our Data Modification Language. When inserting successor rows, the caller does not know the ID of the predecessor. That's because the ID never left the database. This was an important design decision to protect location independence. As a consequence, we need the database to look up the predecessor ID whenever it performs an insert.

We will look up the ID of the predecessor using the hash of the predecessor. Performing a lookup within an INSERT statement requires the INSERT ... SELECT syntax. The calling application computes the hashes of the predecessor and successor fact. The query looks up the predecessor ID by hash and inserts the successor hash directly.

```
INSERT INTO product
  (product_hash, catalog_id, sku)
  SELECT 'fKO2Oge...5GFw==', catalog_id, 'PK47'
    FROM catalog
    WHERE catalog_hash = '8rCOhVD...ifdw=='
  ON CONFLICT DO NOTHING;
```

To compute the hash of a successor from a canonical form, the hash of the predecessor must be used. In the canonical JSON that I prefer, the predecessor is represented by an object having a `ref` field. The value of that field is the hash of the predecessor. Predecessors are sorted alphabetically among the rest of the fields.

```
$ echo -n '{"catalog":{"ref":"8rCOhVD...ifdw=="},"sku":"PK47"}' | \
> openssl dgst -sha512 -binary | \
> base64
fKO2Oge...5GFw==
```

Optional Predecessors

An optional relationship allows zero or one predecessor. In the canonical JSON form, the *zero* case is represented by a `null` value for the predecessor field. And in a relational table, the foreign key column allows NULLs. Foreign keys that can be null are cause for concern in relational database design. The sorts of queries that we will perform, however, tend to avoid these concerns.

In some relational database systems, an index on a nullable column will skip rows in which the column is null. As a result, queries for NULL values will perform a full table scan. Fortunately, the forms of queries that we will build will join on non-null values. The indexes on predecessor foreign keys—even the nullable ones—will always be used in these kinds of queries.

Many Predecessors

Storing the foreign key directly in the fact table is appropriate for cardinalities *one* or *optional*. It does not work for cardinality *many*. Relationships with many predecessors require an associative table. By convention, I call these associative tables *predecessor tables*. The reason is that they conceptually hold the predecessor references of a given fact. They borrow their name from the fact that declares them.

To see an example, let's look at the price of a product. This property could change over time. Each individual price change is recorded as a `Price` fact. Following the *Mutable Property* pattern, this fact has many `prior` predecessors.

```
fact Price {
  product: Product
  value: decimal
  prior: Price*
}
```

The `price` table is constructed from this fact using the patterns already discussed. The table has an ID and a hash. It also has a column for each of the fields and *one* or *optional* predecessors. It does not, however, have any columns for *many* predecessors.

```
CREATE TABLE price (
  price_id SERIAL PRIMARY KEY,
  price_hash VARCHAR(88) NOT NULL UNIQUE,
  product_id INT NOT NULL REFERENCES product,
  value DECIMAL(10, 2) NOT NULL
);

CREATE INDEX price_product
  ON price (product_id);
```

The *many* relationship is represented with an associative table. The name of the table is derived from the successor—the fact that defines the relationship. The table contains only the foreign keys of the predecessor and successor rows, neither of which can be null. The pair of foreign keys is unique, encoding the truism that predecessor collections are *sets*, not *lists*; they cannot contain the same predecessor for the any given successor. Both of the foreign keys are independently indexed.

```
CREATE TABLE price_predecessor (
  price_id INT NOT NULL REFERENCES price,
  prior_price_id INT NOT NULL REFERENCES price,
  UNIQUE (price_id, prior_price_id)
);
```

```
CREATE INDEX price_predecessor_price
  ON price_predecessor (price_id);

CREATE INDEX price_predecessor_prior_price
  ON price_predecessor (prior_price_id);
```

And now it should be a bit more clear why we do not add uniqueness constraints on the fields of a fact. If we defined such constraints on a table that had an associated *predecessor table*, then the predecessors would not be included in the unique index. In this case, had we defined a uniqueness constraint on the fields, it would be impossible to insert two prices with the same product ID and value, even if they had different prior sets. This is precisely what happens when a price returns to its former value.

The predecessors of a fact are part of its identity. But the predecessor references are stored in a different table. Uniqueness constraints cannot cross table boundaries. Therefore, we leave off uniqueness of the fields altogether. The uniqueness constraint on the hash solves the problem perfectly well.

Canonical Hash of a Set

The hash of a fact having many predecessors is based on all of the hashes of those predecessors. The canonical form that I recommend represents many predecessors as a JSON array. If there are no predecessors, the array is empty rather than null. If there happens to be one predecessor, the array contains one reference object. Reference objects are sorted alphanumerically by base-64 encoded hash so that there is only one canonical representation of the set. For example, the first price of a product would be identified by the hash of the following JSON document with whitespace removed:

```
{
  "prior": [],
  "product": {
    "ref": "fKO2Oge...5GFw=="
  },
  "value": 256.98
}
```

The subsequent price would get the hash of the JSON document with one prior reference:

```
{
  "prior": [
    {
      "ref": "ZlUYAZV...ZQZA=="
    }
  ],
  "product": {
    "ref": "fKO2Oge...5GFw=="
  },
  "value": 220.98
}
```

Inserting Many Predecessors

A fact with an empty predecessor set can be inserted exactly as we've done before. This row represents a complete fact with no related rows in the predecessor table.

```
INSERT INTO price
  (price_hash, product_id, value)
  SELECT 'ZlUYAZV...ZQZA==', product_id, 256.98
    FROM product
    WHERE product_hash='fKO2Oge...5GFw=='
  ON CONFLICT DO NOTHING;
```

When inserting a fact with one or more predecessors in the set, however, the DML needs to be extended. Rows are inserted into predecessor tables at the same time as the successor. Inserts are performed within a database transaction to avoid the possibility of using a partially written fact in a query.

```
BEGIN TRANSACTION;

INSERT INTO price
  (price_hash, product_id, value)
  SELECT 'DRFoFgF...BNpw==', product_id, 220.98
    FROM product
```

```
    WHERE product_hash='fKO2Oge...5GFw=='
  ON CONFLICT DO NOTHING;

INSERT INTO price_predecessor
  (price_id, prior_price_id)
  SELECT successor.price_id, predecessor.price_id
    FROM price as successor, price as predecessor
    WHERE successor.price_hash = 'DRFoFgF...BNpw=='
      AND predecessor.price_hash IN ('ZlUYAZV...ZQZA==')
  ON CONFLICT DO NOTHING;

COMMIT TRANSACTION;
```

By using the IN clause in the predecessor insert statement, we can list the hashes of all predecessors that we want to insert. The database will produce all of the primary keys in a single statement.

In this particular example, the predecessor and successor tables are the same type. For that reason, the insert statement needed to create aliases to disambiguate the predecessor and successor rows. This will not always be necessary, but you may wish to always use these aliases for the sake of consistency.

Queries

We've converted fact specifications into constrained statements in the Data Definition Language and Data Manipulation Language. We will now convert pipelines into Data Query Language. When generating a query, you will find it much easier to start from a pipeline than from the Factual query definition. Half of the work has already been done. We just turn each step of the pipeline into a join and each filter into a subquery.

Every query has a starting point. We express the starting point by its hash. The ID never left the database, so we cannot start a query from an ID. The initial shape of the query selects FROM the starting type and lists the hash in the WHERE clause.

```
SELECT reference_number
FROM catalog
WHERE catalog_hash='8rCOhVD...ifdw==';
```

The initial query is almost never used in its basic form. Any application that can provide the hash probably already has the fields. Instead, we start from that initial query and add steps. Each step becomes a SQL join.

Joins

Every step in a pipeline follows a role up to a predecessor or down to a successor. The role has cardinality *one*, *optional*, or *many*. A step following a role with *one* or *optional* cardinality becomes a simple foreign key join. If it is a step up to the predecessor, then the foreign key is in the current table. If it is down to the successor—as in the following query—then the foreign key is in the remote table:
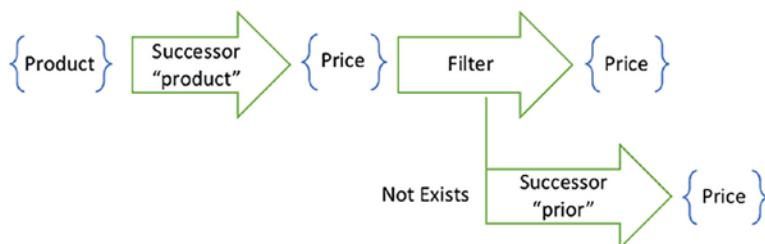
```
SELECT product_hash, sku
FROM catalog
JOIN product ON catalog.catalog_id = product.catalog_id
WHERE catalog_hash='8rCOhVD...ifdw==';
```

For steps following a *many* predecessor role, we need to join through the *predecessor table*. We will see an example of that when we dive down to the subquery.

Correlated Subqueries

When we process a filter in the pipeline, we translate that into a subquery. In particular, it becomes a *correlated subquery*, as it depends upon the values of the current row. A filter will either use the quantifier *exists* or *not exists*. These translate directly into SQL EXISTS and NOT EXISTS clauses.

Suppose we want to select the current price for a product. The pipeline for this query appears in Figure 10-1.

Figure 10-1. *The pipeline for the current price of a product filters prices based on a subquery*

We start with the product. Define a WHERE clause based on its hash. We will then walk down to the successor prices for that product.

```
SELECT price_hash, value
FROM product
JOIN price ON price.product_id = product.product_id
WHERE product_hash = 'fKO2Oge...5GFw==';
```
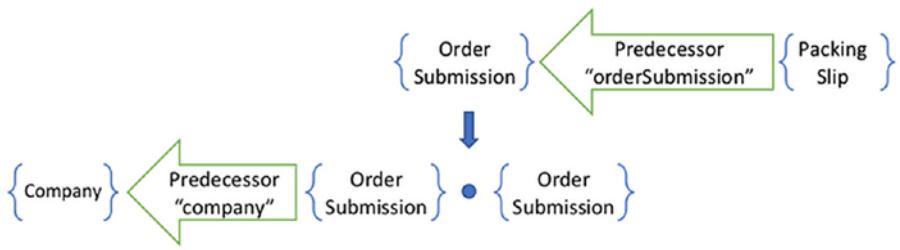
At this point, the pipeline sends us into a filter. The subquery starts at a price. It steps to a successor price using the prior role. This becomes a join using the price_predecessor table.

```
SELECT price_hash, value
FROM product
JOIN price ON price.product_id = product.product_id
WHERE product_hash = 'fKO2Oge...5GFw=='
  AND NOT EXISTS (
    SELECT 1
    FROM price_predecessor
    WHERE price_predecessor.prior_price_id = price.price_id
  );
```

This query will skip the prices that have been replaced. It will only return the current prices. The filter in the pipeline becomes a correlated subquery containing all of the steps of the subordinate pipeline.

Derived Tables

When computing an inverse, we visit subqueries recursively. The innermost subquery becomes the starting point of the inverted pipeline. We cascade the results of that pipeline into another one. One example appears in Figure 10-2.

Figure 10-2. *The set produced by the topmost pipeline feeds into the next pipeline down*

To represent a cascading set in a SQL query, embed the origin pipeline as a subquery. Unlike filters, however, these are not correlated subqueries, they are *derived tables*. Whereas correlated subqueries appear in the WHERE clause, derived tables appear in the FROM clause. A SQL statement to implement the pipeline in Figure 10-2 would select the order submission as a derived table as follows:

```
SELECT company_hash, order_submission_hash
FROM (
  SELECT company_id, order_submission_hash
  FROM packing_slip p
  JOIN order_submission os
    ON p.order_submission_id = os.order_submission_id
  WHERE packing_slip_hash = 'xxxxx...yyyy=='
) AS sub
JOIN company
  ON sub.company_id = company.company_id
```

In this example, the subquery refers to the initial pipeline: packing slip to order submission. It contains the hash of the starting point. The subquery returns the foreign keys and hashes that will be used in the outer query. To the left of the dot, the pipeline computes the affected set. In this case, it takes a step to the predecessor company. This translates into the company join. To the right of the dot, the pipeline computes the new results. In this example, the order submission itself is added to the view. And so the subquery needs to return enough information to materialize those objects.

Selecting Results

All of the queries that we've studied so far select a few columns from the result table. In general, you might also need a little more information. To materialize the results into objects that your application can use, you may find that you need to bring information in from predecessors. For that, you will need additional joins.

All joins derived from steps will be *inner joins*. That is, the query will only include results for which both sides of the relationship exist. Outer joins—or specifically *left joins*—will only be used to load the predecessors of the resulting fact. If those predecessors have cardinality of *optional* or *many*, then a left join is necessary to ensure that results are complete.

In the example that we just explored, the hashes alone are not enough to materialize the new results. In addition, we need to include information about the products ordered. This information is available in predecessor tables. To reach that information, we will perform a left outer join.

```
SELECT company_hash, order_submission_hash, order_line_product_hash
FROM (
  SELECT company_id, order_submission_hash, order_submission_id
  FROM packing_slip p
  JOIN order_submission os
    ON p.order_submission_id = os.order_submission_id
  WHERE packing_slip_hash = 'xxxxx...yyyy=='
) AS sub
JOIN company
  ON sub.company_id = company.company_id
LEFT JOIN order_submission_predecessor
  ON sub.order_submission_id = order_submission_predecessor.order_
  submission_id
LEFT JOIN order_submission_line
  ON order_submission_predecessor.order_submission_line_id =
    order_submission_line.order_submission_line_id
```

In this example, the predecessors have a *many* cardinality. For this reason, we have represented that relationship with a predecessor table. It is possible that a result might have no predecessors. If this were the case and we used an inner join, the result would

be excluded. But by using the left outer join, we ensure that the result appears even if it has no predecessors.

Optimization

From the examples earlier, you should have an appreciation for the richness and complexity of DQL that a pipeline could produce. But you might have some concerns with the less-than-ideal constructs that they sometimes generate. In the previous examples, we translated directly from pipelines to SQL. Now we will optimize that SQL to implement the same pipeline, but in a more efficient way.

When we translated an inverted query into a derived table, we ended up creating SQL that was unnecessarily nested. It included a derived table that was subsequently joined to other tables. Instead of using a derived table, we could simply combine the subquery joins with the main query joins. Not all inverses will produce derived tables that could be optimized away, but the simplest ones usually will. Rewriting the SQL produces code that is easier to understand and maintain.

```
SELECT c.hash
FROM (
  SELECT b_id
  FROM a
  JOIN b ON b.a_id = a.a_id
  WHERE a.hash = 'a_hash'
) AS sub
JOIN c ON c.b_id = sub.b_id

SELECT c.hash
FROM a
JOIN b ON b.a_id = a.a_id
JOIN c ON c.b_id = b.b_id
WHERE a.hash = 'a_hash'
```

While this form of optimization makes for better code, a query engine will produce exactly the same plan for either SQL statement. If you are writing this code by hand, then it makes sense to optimize it. But you might find it more productive to simply let a code generator produce DQL from a pipeline. In that case, let the code generator produce the

code that is easiest to generate, not the code that is easiest to understand. The runtime performance will be exactly the same.

Spurious Joins

One problem that is a little more difficult for a query engine to optimize away is a join that is not needed. These spurious joins will often arise when the pipeline steps to a predecessor and then immediately down to a different successor. For example, to find the name of a student enrolled in a class, we might use the pipeline shown in Figure 10-3.

Figure 10-3. *Finding the name of a registered student involves stepping up to the student and then down to the name*

A direct translation of this pipeline to SQL would produce a query with two joins:

```
SELECT name
FROM registration
JOIN student
  ON registration.student_id = student.student_id
JOIN student_name
  ON student.student_id = student_name.student_id
WHERE registration_hash = 'xxxxx...yyyy=='
```

Careful inspection, however, reveals that the join to the student table yields no additional information. The student ID is already known. We can therefore optimize away the intermediate join.

```
SELECT name
FROM registration
JOIN student_name
  ON registration.student_id = student_name.student_id
WHERE registration_hash = 'xxxxx...yyyy=='
```

Joining directly from one sibling to the next without going through the parent is possible because we don't need any additional information. If some field of the parent *is* required, however, then this optimization is not available to us. A SQL generator could look for this pattern and produce the optimized query. Such an optimization would produce a more efficient query plan.

Covering Indexes

As you analyze the query plans produced by these kinds of queries, they will be dominated by *index seeks* and *index scans*. An index seek is the most efficient way that a database engine can implement a join. For each key in the input set, the engine seeks to a specific place in an index to find the corresponding key in the output set. An index scan is similar, but used for joins that return many related rows—successor joins or *many* predecessor joins. We can take advantage of these fast query operations because we have created indexes for all foreign keys.

At the end of the query plan, however, you will usually find a *clustered index lookup*. The query has produced a set of primary keys through the various index seek and index scan operations. It is now using those primary keys to look up rows in the table to extract the values in the SELECT clause. This last operation might take a significant portion of the query time.

To speed up queries that select additional columns, it is sometimes wise to include those selected columns in a *covering index*. When creating an index based on a foreign key, you can include data columns. The values in those columns will be copied into the index so that they are readily available.

```
CREATE INDEX price_product
  ON price (product_id)
  INCLUDE (value);
```

While this could improve the performance of heavily used queries, covering indexes incur a cost. Since the index includes additional data, it will consume more space. Inserts will take slightly longer, and index operation will take a little extra time. It will be less time than the index seek followed by the clustered index lookup, but the difference in performance should not be ignored. Covering indexes should be used sparingly and only after profiling your queries.

Where Not Exists

The most significant performance concern arises from existential conditions. Many of the queries that we produce in real-world applications will include a `WHERE NOT EXISTS` clause. The clause arises most often from the Delete, Mutable Property, and Queue patterns. It results in an *anti join*. This is an index scan that only returns input rows for which *no* output row is produced. In and of itself, this is not a slow operation, especially because it can use a foreign key index. The issue arises when it is performed over a large input set, even when it produces a small output set.

The index scan will be performed for every input row. Performance is not related to the number of output rows. This means that performance problems can hide in queries that seem small. Based on the way the pattern is used, we can estimate how many keys will be scanned vs. how many results will be produced. You could define the percentage of *waste* to be ratio of excluded inputs to the total number of inputs. For performance to be roughly correlated with value, we would like waste to be bounded, and reasonably low.

Mutable Properties

If the `WHERE NOT EXISTS` clause occurs in a query for the current value of a mutable property, then the input set will include all historical values. The output set will contain only those versions that have not been superseded. The waste, therefore, would be the ratio of superseded versions to total versions. For a mutable property that changes frequently, waste would be high. But if the property changes relatively infrequently, waste is low.

For this reason, the Mutable Property pattern should only be applied to slowly changing properties. Things that change often over the lifetime of an entity produce significant waste. Slowly changing properties are things like a person's name or a company's address. Frequently changing properties are things like status or progress. Workflow patterns are often better alternatives for frequently changing properties.

Deletion

Another place in which `WHERE NOT EXISTS` often appears is a query that excludes deleted entities. The input set will include all entities that have ever existed. The output set will contain only those that still exist. The waste is the ratio of deleted entities over all entities. When most entities are deleted, waste is high.

Think back to the last CRUD application that you built. There was probably one large table that included the primary entity that the app worked with. Perhaps it was a customer table, which you queried frequently to produce a large list of current customers. Since customers come and go, more and more customers were deleted over time, compared to a relatively stable number of current customers.

If you used an actual DELETE statement, then you kept waste bounded. But if you used a soft delete, then waste grew over time. With a soft delete, you set a field to simulate a deletion and then selected only those rows where the flag is not set. The soft delete preserved data, but it came with a performance cost.

A deletion in a historical model is similar to a soft delete in a CRUD model. It preserves data, but incurs the same performance costs. Fortunately, we have a couple of options for regaining that performance.

The first option to consider is to add a clock to the model. Please see the Period pattern in Chapter 8. A clock is a starting point for a query that slowly advances over time. Some models have natural periods, like dates of business in a retail system or semesters in an academic domain. In others, the clock is a bit more artificial. In either case, a query based on a small number of periods limits the amount of waste due to deletions. Only survivors move from one period to the next.

If no reasonable clock can be found, then a more drastic solution might be called for. Measure your performance and calculate the actual percentage of waste. If you find that most entities are deleted, and that that is affecting query performance over time, then consider a managed index. This solution is most applicable to queues, so we will examine it from that context.

Queues

The Queue pattern, as documented in Chapter 8, includes a successor fact that records the outcome of taking action upon a work item. The queue is fed by a query that selects work items for which no outcome exists. These translate into pipeline filters, which eventually become WHERE NOT EXISTS clauses in SQL. The input set to the anti join is the set of all work items. The output set contains only the work items not yet processed. Since every work item will eventually be processed, this query approaches 100% waste.

When relying upon an anti join, we are taking away the SQL engine's most valuable tool: direct addressability. The items in the queue are not directly addressable. Instead, the index stores specifically those items that are *not* in the queue. The engine has to scan

that index for every work item to see which ones are not excluded. It would be much faster if we could convince the SQL engine to directly index those that were included.

To so convince the engine, we need to provide it something that it can directly index and address. Something that represents the *absence* of a successor. One option is to re-examine the soft delete flag.

A soft delete as implemented in a CRUD system is an UPDATE that takes the place of a DELETE. Instead of completely destroying the record, the application updates it to set a flag. Such a flag can be indexed to provide direct access to those records that have not been marked deleted. When relatively few of the rows are marked, the index is not selective. Most query engines will avoid a nonselective index because a table scan would turn out to be faster. But if most of the rows are marked—as in work items processed from a queue—then the index becomes quite selective. The query engine can use the index to directly address the relatively small number of unprocessed work items.

Even when it is selective, an index on a flag has a space disadvantage. All rows are indexed, even the ones for which the flag is set. The index will never be used to find the flagged work items, because the query specifically looks for the non-flagged ones. We could save some space if we excluded completed work items from the index altogether.

Some database management systems include a *partial index* feature. If this feature is available, you can create an index that has a condition. Specify that condition to include only unprocessed work items in the index, and that index will be used to directly address those items.

```
CREATE INDEX work_item_unprocessed
  ON work_item (queue_id)
  WHERE processed = FALSE;
```

But even if your database management system does not support this feature, you can simulate it. Create a table that you will use as an index yourself. Insert rows into this table to indicate that the flag is *not* set, and delete them to indicate that it *is* set.

```
CREATE TABLE work_item_unprocessed (
  queue_id INT NOT NULL,
  work_item_id INT NOT NULL UNIQUE
)

CREATE INDEX work_item_unprocessed_queue
  ON work_item_processed (queue_id);
```

Whether you add a mutable flag and define a partial index, or you add a table to store unprocessed work items, you will have to manage the index yourself. For this reason, I call this technique a *managed index*. I recommend using triggers for index management. While the index could be managed by the application, it exists to address a database performance concern. The solution should therefore be entirely specified within the database.

To use the index table approach, create two triggers. The first inserts into the index table when a new work item is created. The second deletes from the index table when the outcome is created. For example, the following pair of triggers manages an index table for a queue:

```
CREATE FUNCTION insert_work_item_unprocessed() RETURNS TRIGGER AS $$
  BEGIN
    INSERT INTO work_item_unprocessed
      (queue_id, work_item_id)
      VALUES (NEW.queue_id, NEW.work_item_id);
    RETURN NULL;
  END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER work_item_created AFTER INSERT ON work_item
  FOR EACH ROW
  EXECUTE FUNCTION insert_work_item_unprocessed();

INSERT INTO work_item
  (queue_id, description)
  VALUES (47, 'Do some work');

CREATE FUNCTION delete_work_item_unprocessed() RETURNS TRIGGER AS $$
  BEGIN
    DELETE FROM work_item_unprocessed
      WHERE work_item_id = NEW.work_item_id;
    RETURN NULL;
  END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER work_item_outcome_created AFTER INSERT ON work_item_outcome
  FOR EACH ROW
  EXECUTE FUNCTION delete_work_item_unprocessed();
```

Join through the index table instead of using the `WHERE NOT EXISTS` clause. The query optimizer will use the table to directly address items remaining in the queue. It will not need to perform a scan over all historical work items in order to perform an anti join. The results will be much faster, especially for work items where 100% of them are eventually excluded from the query.

Integration

Not all of an application's features will be served out of an immutable database. Some queries do not lend themselves to traversing predecessor and successor relationships from a single starting point. Some table structures are better denormalized. When a mutable, state-based structure is more appropriate, use that structure. Integrate between the immutable database and the mutable one.

In separating mutable from immutable data, be sure to keep the data structures isolated from one another. Do not mix the two table structures. Do not include foreign keys from one model within the other. And avoid any form of cross-joining between the two models. Even though it may make sense to use the same relational database engine for the two models, do not combine them into the same database. They have distinct purposes and patterns of use.

Legacy Application Integration

When introducing immutable architecture to an organization, you will not have an opportunity to rewrite all of your systems in this style. Nor should you seek such an opportunity. It would introduce a great deal of risk with rewards that would be realized far too late. Instead, let legacy applications continue to exist having static, mutable data models. Find the best integration points to work with those legacy applications with minimal risk and rework.

Some legacy applications might have been written with event-driven architectures in mind. If so, leverage those integration points as places where historical facts are produced and advertised. More likely, however, a legacy application will simply be a collection of stateless behaviors on top of a mutable data model. It will not throw off events that could be turned into facts. There will not be an easy way to modify the application source code to create those integration points.

In these cases, I suggest using the database as a point of integration. This is not to say that two applications should share a common database: that practice leads to tight coupling and rapid stagnation. Instead, use the tools inherent in the database to extract facts from state changes. These tools include scanners, triggers, and change data capture.

Scanners

The most direct way to extract facts from data is to scan for them. A scanner is a handwritten query that projects the current state of an application into the set of facts that gave rise to that state.

To design a useful scanner, you must understand which parts of the model might change and which parts are immutable. Relational database engines don't have many tools to enforce or document immutability, but they have a few. One such tool is the uniqueness constraint.

When a table is given a uniqueness constraint, the designer is documenting the fact that the included columns form a natural key for the row. Those columns represent an identity independent of the auto-incremented surrogate key. Even though the database engine does not strictly prohibit an application from changing these columns, doing so is strongly discouraged. If a database designer has gone through the trouble of declaring a uniqueness constraint, you can be fairly confident that the columns are immutable.

Begin by producing a series of queries that select out the immutable data from the source system. You should be able to project these results directly into facts. If those facts already exist in the target immutable model, then no harm done; they will not be duplicated. Just project these results and insert the corresponding facts.

Next, you will be left with the mutable data. Projecting these will require a bit of history. To help keep track of its work, the scanner should keep a scratch pad of the mutable values that it last saw per entity. When it finds different values, it can use this scratch pad to find the prior versions and make those facts predecessors of the new facts. The specific details of the scratch pad depend heavily upon your application and chosen technology stack. Suffice to say that it should contain enough information to fully reconstruct the historical tree of prior versions.

It is common practice for a static database to be designed with auditing columns. These record the date on which the row was inserted and the date of the most recent update. A scanner can take advantage of these auditing columns to restrict the amount of work that it must do. Keep a bookmark in the scanner's scratch pad, separate from either

the source database or the target immutable model. Include a `WHERE` clause that limits the scanner to rows that have been inserted or updated after the bookmark. Update this bookmark only after you are confident that the new facts have been persisted.

When migrating from any legacy database into an immutable data store, the scanner is always the first tool to build. You can construct the scanner layer by layer, first scanning for high-level facts having no predecessors and then building scanners for successors lower down in the graph. As you build the successive layers, continuously deploy the scanner to production. Scanning is not done just once. It will be done many times over the life of the project. Do not make the mistake of treating this like a one-and-done data migration. Invest the time in making the process repeatable.

Triggers

Triggers are perhaps the most contentious tool in the data modeler's toolkit. Perhaps they were overused in the past, leading several application developers to shy away from them and even warn others away. But for extracting information from a database, database tools are the most appropriate, especially if our aim is to minimize changes to legacy code.

We already used triggers to optimize queries for work items that had not yet been processed. We can also use them to infer facts from data modification operations. The original intent may be locked away in application code that we dare not modify, but the effect of that intent can be readily observed. Some well-crafted triggers can translate an `INSERT` into a new entity, an `UPDATE` into a mutable property, and a `DELETE` into a tombstone. They can insert these new facts into a set of staging tables to be processed through the same channels as user actions.

Triggers of this kind cannot take the place of scanners. They will have been put into place after the static data model has been in operation for some time. A trigger cannot go back in time and fire on all of the inserts, updates, and deletes of the past. A scanner is still required to bring in all of the legacy data.

Triggers, if implemented, will need to work in conjunction with scanners. At some point, you may choose to turn off the scanner while simultaneously enabling the triggers. Do this only if you have sufficient confidence that the triggers will catch all possible changes. If you use bulk inserts, for example, triggers may not fire. Alternatively, you may choose to implement triggers *using* the scanner. Write the scanner as a series of stored procedures. Then the trigger simply fires the scanner whenever it detects information being modified. This gives us the best of both worlds: a scanner running on a timer to collect bulk data changes and triggers running in real time to provide lower latency.

Change Data Capture

The patterns described previously can be implemented in any database management system. Some systems, however, provide more support. SQL server, for example, provides specific features that can be leveraged to extract changes from a database. The most powerful of these is change data capture.

When configuring change data capture, an administrator defines source tables. When DML operations against any of these source tables enter the transaction log, the capture process is notified asynchronously. The process inserts a record of each change into a corresponding change table. An application process can periodically pull records from the change tables and convert them into historical facts.

Change data capture has the advantage of using the transaction log to asynchronously identify changes. Triggers, on the other hand, fire synchronously with the DML command. This means that they have the potential to block or even break the legacy application. Change data capture is built into the database management system, whereas triggers need to be coded by hand. On the other hand, triggers could be customized to handle more complex scenarios. The choice between the two is by no means clear.

We have explored several alternatives for extracting facts from a static database. Legacy application integration must of course work in both directions. For getting information *into* a legacy application, please see the Outbox pattern in Chapter 8 or use the techniques described for reporting databases in the following section.

Reporting Databases

Executing reports directly from an immutable database can be challenging. The queries that compute the current state of an entity tend to work well for singular starting points. They are not well suited to aggregation, as is often required in reports. They are also not well equipped to group by mutable properties, as is common in reporting scenarios. For effective reporting, an immutable data model should be projected into a denormalized reporting model.

To build an effective reporting database from an immutable model, decide which facts will be reflected in the static database. Not all immutable records need to be reported against. Perhaps you can even subdivide the reports into categories and produce separate projections of the data for each. Top-level predecessors are fed into all of the reporting projections, but successors might only influence a few.

Once identified, write a job that will call a procedure for each fact in a historical table. This might be a scheduled database job, an application cron job, or even driven by a trigger. The choice depends upon your tolerance for latency and the degree to which you feel the application should be involved. This job runs completely within the scope of the immutable data store. The procedures that it calls, however, run within the reporting database.

Those procedures should in fact be stored procedures. They take as parameters all of the fields of the fact that they are processing. The procedure transforms an entity fact into an `INSERT` statement, a tombstone fact into a `DELETE`, and any mutable property or workflow fact into an `UPDATE`. The stored procedure then returns the ID of the row that it just affected. This helps the job to keep track of predecessors.

As the job calls procedures to handle facts, it keeps a map. It maps the hash of each fact to the ID that the handling procedure returned. Then, when it calls a procedure for a successor fact, it passes the mapped ID in place of the predecessor. This gives the stored procedure all of the information it needs to set up foreign keys.

The work of transforming facts into database changes is done completely within stored procedures. This keeps the information about the reporting database schema localized to the database. The individuals in charge of producing and optimizing reports will find this choice of tool to be near at hand. This solution provides the proximity, convenience, and autonomy required to produce the best reporting solution.

Application-Agnostic Stores

Throughout this chapter, we've done a lot of hand coding to turn a Factual model into a relational database. We've converted fact types into tables, predecessors into associations, and pipelines into queries. Understanding this process was helpful, but in the end, it did not require any creativity. It turned out to be quite mechanical. If we fully embrace automation, we can generate code, simplify database deployments, and avoid many of the headaches of versioning.

An immutable data store imposes a significant set of constraints. One of them is of course that rows cannot be updated or deleted. But in deriving the structure of an immutable database from pipelines and inverses, we've picked up a few other constraints. The `WHERE` clause chooses the first row by hash. All `JOIN`s are by predecessor or successor foreign keys. Outer joins are only used for the final predecessors. Far from being a burden, these constraints help us to write efficient and correct queries.

In particular, these constraints do not permit us to query the database by any of the data fields. We cannot find all products under a certain price, or all shipments over a certain weight. These queries cannot be efficiently written against an immutable database and should instead be executed against a static projection, like a reporting database. Under such constraints, one might begin to wonder why we should store price weight in columns at all.

Databases that support ad hoc queries—such as reporting databases—benefit from the ability to define an application-specific schema. If a report needs to filter or group by a field, then that field needs a column. But pipelines prohibit us from running ad hoc queries. The only reason for having data columns is to be able to store and retrieve fields. If that purpose can be served in another way, then all tables start to approach the same shape.

A Generic Fact Table

Most relational database engines have an efficient way to store collections of named values. In PostgreSQL, for example, a JSON object can be stored in its binary form to save the overhead of quoting and escaping strings. This is primarily done to allow the database engine to index fields of a JSON object, but we need not use it in that way. We can simply store and retrieve entire objects. And even if your chosen database engine does not have a binary JSON type, you can easily store JSON objects as strings.

Once you begin storing fields in JSON columns rather than individual columns, the differences between application-specific tables start to disappear. They all have an auto-incrementing surrogate key. They all have the hash of their corresponding fact. The last remaining reason for having different tables is to distinguish among facts of different types.

Even the type can be generalized. We can uniquely identify a fact type by its name and store that in a `type` table to give it a surrogate key. Then, we can store the fact ID, type ID, fact hash, and JSON fields in a `fact` table. With this application-agnostic structure, we've retained all of the important information.

```
CREATE TABLE type (
  type_id SERIAL PRIMARY KEY,
  name VARCHAR(50) NOT NULL UNIQUE
);
```

```
CREATE TABLE fact (
  fact_id SERIAL PRIMARY KEY,
  type_id INT NOT NULL REFERENCES type,
  fact_hash VARCHAR(88) NOT NULL,
  fields JSONB NOT NULL,
  UNIQUE (fact_hash, type_id)
);
```

The unique index combines the hash with the type. If it did not, two facts with the same fields but a different type will collide.

Predecessor Relationships

With all facts stored in the same table, all predecessor relationships can be grouped together as well. In an application-specific immutable database, singular predecessors are represented as foreign keys in the successor table. Only the multiple predecessors were extracted to associative tables. But in an application-agnostic data store, we will extract all predecessor relationships.

Just as every fact has a type, every predecessor relationship has a role. A role is the named relationship defined within a fact type. For example, in the Product fact type, the catalog is a role.

```
fact Product {
  catalog: Catalog
  sku: string
}
```

Every role is declared by one type and targets another. The catalog role is declared by Product and targets Catalog. It takes both the declaring type and the name to uniquely identify a role. We capture that in a table to give a role a surrogate key.

```
CREATE TABLE role (
  role_id SERIAL PRIMARY KEY,
  declaring_type_id INT NOT NULL REFERENCES type,
  target_type_id INT NOT NULL REFERENCES type,
  name VARCHAR(50) NOT NULL,
  UNIQUE (declaring_type_id, name)
);
```

The role ID allows us to define a table for each predecessor relationship. Here, I'll name this table edge to indicate that this is an edge in a directed acyclic graph. The edge joins one successor to one predecessor within a role. All three columns together form a unique key.

```
CREATE TABLE edge (
  predecessor_id INT NOT NULL,
  successor_id INT NOT NULL,
  role_id INT NOT NULL,
  UNIQUE (successor_id, role_id, predecessor_id)
);

CREATE INDEX edge_predecessor
  ON edge (predecessor_id, role_id);
```

The unique index defined earlier doubles as a way to get from successor to predecessor. Using the prefix of the index, a query could join from a known successor ID and role to find a predecessor. That index, however, would be useless in the other direction. And so we've defined a second index starting at the predecessor ID when joining down the graph.

With these four tables—type, fact, role, and edge—we can store any immutable data model. Using the same techniques we studied in previous sections, we can generate DQL directly from pipelines against this application-agnostic data store. Moreover, query generation could be automated. You wouldn't want to write very many of these queries by hand, but a machine can do it with ease. The open source library Jinaga is just one example of a system that parses specifications into pipelines, computes inverses, and generates SQL queries on behalf of an application.

The benefits of application-agnostic data stores begin with code generation, but they go much further. It is much easier to deploy a single database schema that does not change as you add features to an application. Many applications can share space within the same data store, reducing the overall cost of operations. It is easier to move data from one store to another if you can be assured that they have the same schema. But most significantly, application-agnostic data stores solve the versioning problem.

Versioning

There is a hidden immutability constraint built into any application-specific database. As an application evolves, it is inevitable that the schema will evolve with it. Immutability demands that a fact does not change. As we've stated many times already, this means no UPDATEs and no DELETEs. But this also means no ALTER TABLEs. It isn't just the values that can't change; it's also the shape.

When extending an immutable data structure, it is best to define entirely new fact types. You should avoid adding fields or predecessors to an existing fact type. The reason is simple: existing facts do not have those fields or predecessors. To add them would be to modify an immutable fact. Any modification changes the fact's identity. All of the mathematical and analytical structures that we've carefully assembled throughout this book crumble on the shifting sands of changing identities.

Fortunately, most extensions that you might want to make over the lifetime of an application take the form of new successors. You might want to add a new property to an entity. That's fine: mutable properties are modeled as successor facts. You might need to add another step to a workflow. Piece of cake! Each step is a successor fact. Adding a new successor to an existing model is not a breaking change. It's precisely the kind of extension that immutable models absorb well.

But sometimes you find that you really do need to change the shape of a fact type. You might find that a top-level fact needs to be nested under a previously invisible owner. Or you might have thought that one field was sufficient to uniquely identify an entity, only to later discover that it takes two. In these cases, you truly do intend to extend the identity of a fact by adding predecessors or fields. How can you do that without modifying existing facts?

The answer is versioning. Existing facts retain the shape with which they were first defined. When we rehydrate those facts from the data store, we don't want to set the new predecessors to null or define defaults for the new fields. We still want to load those old facts using the old schema. That is the only way to preserve their identity.

Avoid Sequential Version Numbers

It is common practice to assign sequential version numbers to types. You see this in APIs, which often retain old versions of endpoints so that they don't break old versions of clients. One version of an endpoint takes and returns old versions of data structures,

while a newer version of the same endpoint takes and returns new data structures. It is even common practice to use the sequential version number within the URL of the endpoint.

In an immutable data structure, however, sequence numbers are to be avoided. Just as with auto-incremented identifiers, a sequence number is location specific. It implies that all versions originated from the same place. With a primary key in a database, that place was the database server. With a sequential version number, that place is a developer.

At first glance, it seems reasonable to assume that every version of a fact type will originate from the same developer. It seems sensible that they will emerge one at a time in a predictable order. But the truth is that developers work in teams. They do not necessarily finish work in the same order that they started it. They might not even deploy work in the same order that they finish it. The order in which a version appears might be different from the order in which the number is assigned.

Fortunately, the versioning problem can be solved in exactly the same way as the identity problem. A version of a fact type is defined by its shape. The distinct set of fields and predecessors defined by that type *is* the version of the type. And the version can be represented by a hash.

Structural Versioning

Following an algorithm similar to computing the hash of a fact, we first define a canonical form of a fact type. Begin with the type name. Sort the fields and predecessors alphabetically by name and include a description of their type. For fields, it would be their native type, while for predecessors, it would be their cardinality and the name of the target type. Serialize the canonical form, and compute the hash.

```
$ echo -n 'Price{prior:Price*;product:Product;value:decimal}' | \
> openssl dgst -sha512 -binary | \
> base64
8/HZgkV...lZMw==
```

The version is stored with each fact. To optimize storage, you can insert a new version table in between fact and type.

```
CREATE TABLE version (
  version_id SERIAL PRIMARY KEY,
  type_id INT NOT NULL REFERENCES type,
  version_hash VARCHAR(88) NOT NULL UNIQUE
);
CREATE TABLE fact (
  fact_id SERIAL PRIMARY KEY,
  version_id INT NOT NULL REFERENCES version,
  fact_hash VARCHAR(88) NOT NULL UNIQUE,
  fields JSONB NOT NULL
);
```

A fact was created with a specific version of the type. It needs to be rehydrated with that same version. Build each of the versions into the application, so that it can load the fact into the appropriate data structure. If the application encounters a version hash that it does not recognize, then it simply ignores the fact *and its successors*. It must have been created with a later version of the application, and so that entire subset of the graph represents data that it could not correctly interpret.

Roles should *not* use specific versions of facts. The predecessor/successor relationships between facts that we use within a query will persist from version to version. New versions should be able to participate in those relationships right alongside old versions. If we included the version ID in the role, then new versions of successors could only be related to new versions of predecessors. Once we begin pulling on that thread, the tapestry of application continuity quickly unravels.

The name of the type is semantic; its version is structural. Structural versioning preserves the identity of existing facts as an application evolves. It allows newer software to load data written by older versions, but protects older software from the corruption of misinterpreting new data. Versioning is a much more difficult problem when the database knows the schema of each fact type. Only with an application-agnostic data store can the versioning problem be completely solved.

Communication

Many of the architectural choices that we make constrain the way in which messages are delivered and the way in which they are processed. Two common examples of message delivery are REST APIs and service busses. Which one you choose often dictates how messages are processed. In a REST API, a message is sent as a synchronous HTTP request. The server processes the request and sends back the response. In a service bus, a message is sent asynchronously by pushing it to a queue or publishing it to a topic. The recipient processes the message as it pulls it from the queue and publishes the result, if any, for downstream consumption. Patterns for communication and processing are tightly bound.

Immutable architectures give us a way to separate those two concepts. We can make choices about communication based on how close the sender is to the recipient, whether they are controlled by the same organization or whether they are more or less permanently connected. On the other hand, we can make choices about processing based on how a conversation is intended to progress and whether the initiator expects a response. We can choose how to exchange facts between nodes irrespective of our choice of how those nodes process the messages.

The models that we have analyzed and constructed define *what* the nodes are talking about. They do not define *how* those nodes will talk. To maximize autonomy, each node will have just the subset of the model that it requires to serve its users and make its decisions. They all participate in an information exchange to share subsets with one another. To make the most appropriate communication choices, we need to understand those subsets, the needs of each node, and the constraints of different communication protocols.

Delivery Guarantees

As the parable of the Two Generals taught us, a node cannot know whether the message it just sent will be received. It has no guarantee based solely on the sending of the message. Instead, it only learns about successful delivery when it receives a subsequent message from the remote node. Knowledge is delayed. Guarantees can only be fulfilled by retrying until that knowledge arrives.

Fortunately, we can build more reliable delivery guarantees on top of less reliable protocols. This is evident in the OSI model of networking, shown in Figure 11-1, which subdivides the stack into seven layers. The model describes many quality-of-service factors, not just delivery. To focus just on delivery guarantees, we only need to examine three layers: network, transport, and application.

Figure 11-1. *The OSI model of network communication divides protocols into seven layers of abstraction*

At the network layer, no communication protocol offers a delivery guarantee. The network is concerned with addressing, routing, and error correction, but not with delivery. It is not responsible for establishing long-lived connections between nodes, retrying failed packets, or even reporting on success or failure.

At the next level up, the transport layer takes on the responsibility of reporting successful delivery back to the sender. It provides confirmation, but not necessarily disconfirmation. It is often not possible to prove that a message was *not* received. But ultimately, the transport layer must give up at some point. It cannot make a guarantee before sending a message that it will keep trying until the message is received.

It is only at the application layer that protocols begin to offer such guarantees. If a message is given to a durable protocol, then it will do everything that it can to ensure that the message makes it to the intended recipient. It will keep trying to send the message until it knows that it was received. It will resume after a power failure. Some protocols even make additional promises about the order, uniqueness, and latency of delivery. The more a durable protocol promises, the more expensive it will be. We will therefore accept the weakest promise that we can tolerate.

Best Effort

The term "best effort" is an unfortunate moniker. While it would seem to imply that there is no greater effort that could be applied to solving the problem of delivery, it in fact means the opposite. A best-effort service will not try to resend a message upon failure. In fact, it will not even report on the success or failure of delivery. It is the quality-of-service (QOS) equivalent of a shrug.

All protocols at some point are built on best-effort layers. In most modern applications, this usually means the Internet Protocol (IP). Some protocols extend that limited quality of service up to the application layer. These include User Datagram Protocol (UDP) and IP multicast. When latency is more important than delivery, these are appropriate choices. They can be used alongside more durable protocols to provide services such as presence, streaming, and health monitoring.

To build on top of a best-effort protocol, the recipient must provide feedback upon receipt. This gives the sender confirmation that the message has been delivered.

Confirmation

At the transport layer of the OSI model, some protocols rely upon confirmation that a packet has been received. This is often done to throttle communications, holding some packets back until earlier packets have been confirmed. But in many cases, this is also used to establish a duplex connection between the two nodes. The most prominent example is the Transmission Control Protocol (TCP), which is built on top of IP.

When a duplex—or two-way—connection has been established, each node knows that it can successfully route packets to the other. That connection offers a tunnel through which messages can be sent and received. Peers can rely upon the fact that if bytes are received, they arrive in order and with a very low probability of error. As long as the connection remains open and has not timed out, the TCP protocol will retry packets until they have been confirmed. No application intervention is required.

Many application protocols rely upon duplex connections to provide delivery confirmation to their consumers. The examples are too numerous to examine, but certainly the best known and most widely used is the Hypertext Transfer Protocol (HTTP). Despite "hypertext" in the name, this protocol has become the de facto standard of all sorts of information exchange on the Web, not only HTML but also SOAP, JSON, and gRPC. HTTP upholds delivery guarantees by constraining how nodes may change state upon receipt of various messages.

Safe Methods

The HTTP specification speaks of two properties of methods: safety and idempotence. The first category of methods that we will examine are those that have the property of safety. A *safe method* does not change the state of the server upon receipt. Verbs like GET and OPTIONS are safe.

Upon receiving a safe request, a server may retrieve information, but it may not *alter* its state in any observable way. Caching a response, while technically a state change, is not directly observable to a client. Caches are therefore allowed for servers processing safe methods.

As a client, you can feel confident in sending a safe request that you will not trigger any unwanted state changes. You can retry a GET on a different connection if you did not receive a response. The server should theoretically respond in the same way, assuming that no state changes occurred in the interim. Of course, there is no way for the client or the protocol to *enforce* this convention. It is entirely up to the server to refrain from changing state in response to a safe method.

Idempotent Methods
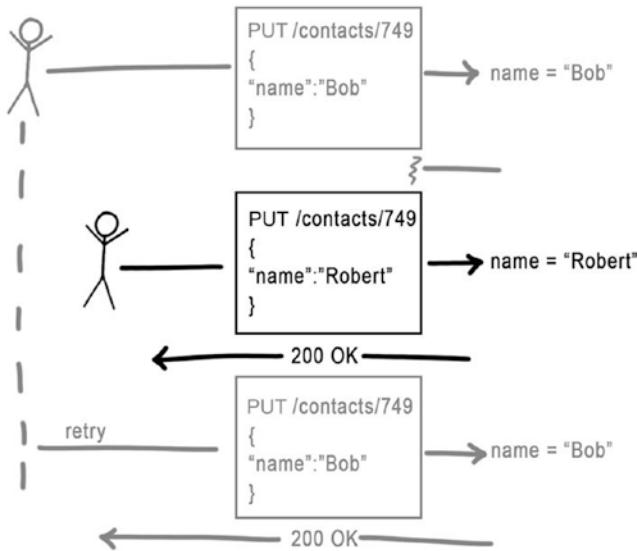
The second property of methods that HTTP defines is idempotence. This promises that the state of the server will change only upon the first receipt of a distinct message, not a subsequent receipt. All safe methods are by default idempotent; the server will not change state on even the first receipt, let alone the second one. And so the second category that we examine are idempotent, but not safe.

As we've learned in *Chapter* 4, idempotence is an important property of a message handler. It allows peers to retry messages without fear of changing state. If the first message was indeed received, then the second receipt will not change state further. In HTTP, PUT, PATCH, and DELETE are examples of idempotent verbs.

While idempotence is required for reliable message delivery, the reason for these verbs being labeled idempotent is *not* to permit retries and recover from duplication. It is simply based on the semantics of updating, patching, or deleting a resource. An update sets the state of a resource to a known quantity. A patch sets only some of the properties, but still to known quantities. Logically, one would anticipate that upon duplication, the resource is already in the desired state. Updating or patching the resource would have no effect. A similar argument applies for deleting an already-deleted resource.
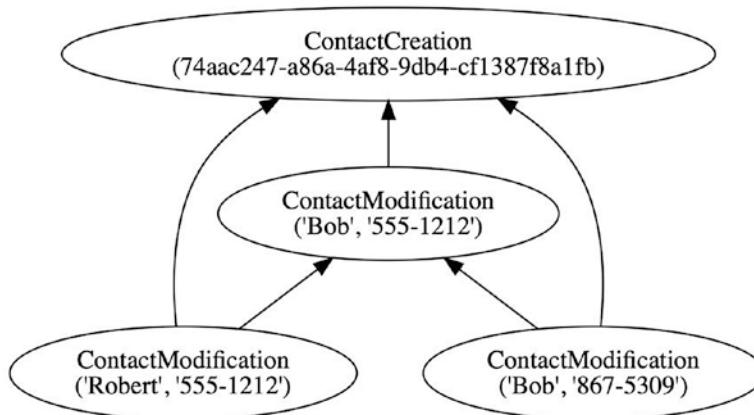
Unfortunately, this line of reasoning only considers duplicates without intervening changes. If the resource changes between the original update and the duplicate, then the duplicate will reset the state of the resource *back* to a previous state. An eventually consistent handling of the message would ignore the duplicate, rather than applying it.

Consider the example in Figure 11-2. The first client issues a PUT request to update the value of a resource to "Bob". That command takes effect, but the connection is interrupted before the response gets back. Meanwhile, a second client issues a PUT request to update the value of the same resource to "Robert". That client sees their response. The first client makes a new connection and retries the message. If the server changes the value back to "Bob", as the HTTP protocol would suggest, then it is not ignoring the duplicate message.

Figure 11-2. *A server responding to a duplicate request after a different request has intervened*

The HTTP server is behaving in an idempotent manner. And yet, the system is not eventually consistent. The problem is that idempotence is not sufficient. As we showed in Chapter 4, a protocol must also be commutative to ensure eventual consistency. If the server in Figure 11-2 had responded in the same way regardless of message order, then the outcome would have been better. Suppose that it treated "Bob" followed by "Robert" the same way as "Robert" followed by "Bob", for example, by allowing the resource to be in a superposition of the two candidate values. Then, the subsequent receipt of the duplicate message would simply be ignored. Recall the diagram from Chapter 4, repeated in Figure 11-3, illustrating this solution.

Figure 11-3. *A data structure that allows a resource to have multiple simultaneous versions is both idempotent and commutative*

The HTTP guarantee of idempotence is only semantic. The kinds of actions that idempotent verbs represent tend to bring a resource to a known state, even after the second application. But they offer no protection against message duplication. HTTP offers no guarantees of commutativity.

Non-idempotent Methods

The third category to consider are the methods that are neither safe nor idempotent. These methods offer no guarantees. They may change state upon every receipt. The POST verb is an example of this kind.

The semantics of POST make it likely that a change will in fact occur. In response to this request, a server creates a new resource and returns its identity in a 201 Created response. Presumably, the identity of the resource was not already known to the client before the request. If it was, the response would not need to include the URL. We can reasonably assume that in most implementations of POST, the identity was generated on the server.

When the identity is generated on the server, then there is little a client can do to prevent duplication. If the connection is lost before it receives the 201 Created response, the client has no recourse but to try again or report the error to the user (who is likely to try again). A second try will likely result in the creation of a second resource. HTTP makes no guarantee that the server will do any different.

Retry Within a Connection

Whatever the state-change guarantee of the HTTP method, the transport layer can only provide confirmation within the scope of a connection. Connections are relatively short-lived and reside entirely in memory. They represent a single-threaded communication channel between two peers.

Applications built on connection-oriented protocols need not retry messages on the same connection. TCP guarantees byte order, which implies that the retry would not be received until after the original message. But if the connection fails, then all bets are off. The sender has no knowledge of whether unconfirmed messages were received or not.

A connection can use delivery confirmation to guarantee that a message *has been* received. It cannot guarantee that a message *will be* received at some point in the future. Confirmation is a necessary, but not sufficient condition for durability. While HTTP only forwards the connection semantics from the transport layer, other application-layer protocols add durability.

Durable Protocols

When the user of an application initiates a command, they would like to have some confidence that the effect of their command will last. The protocols described in the previous section will simply force them to wait until a remote peer confirms receipt of their message. But doing so robs the user of some autonomy. They can no longer make a decision and issue a command without involving the remote peer. To have the greatest autonomy, they should be able to work in isolation. And so they demand more from their protocol.

A *durable* protocol is one that guarantees that a message will eventually be delivered. Delivery confirmation is necessary, but not sufficient. Durable protocols need to continue to retry until such confirmation is received, even over long periods of time or power outages. Durable protocols therefore require durable storage at the sending side, which can only be provided at the application layer of the OSI model. Two of the most common forms of durable message storage are queues and topics. Popular examples of these forms are Advanced Message Queueing Protocol (AMQP) and Apache Kafka.

Queues

AMQP is a standard application-layer protocol for exchanging messages in queues. It is implemented by such queueing systems as RabbitMQ, Apache ActiveMQ, and Azure Service Bus. AMQP is a configurable protocol, offering several levels of service. Some of those service levels provide *at least once delivery* or the guarantee that the sender will keep trying until a message is received. This promise survives beyond a single connection or session. It even survives power outages.

To provide this guarantee, AMQP implementations store messages on the client side. When a user publishes a message to an AMQP provider, it stores the message immediately, before contacting any remote peers. At that point, the application can be assured that the message will be delivered. The engine begins the background process of creating a connection, transmitting messages, and receiving confirmations. Once a message is confirmed, it can be removed from client storage. Until then, it must be preserved.

Topics

Where AMQP defines queues, Kafka defines topics. A topic is a persistent stream of records. Unlike a queue, records in a topic are not removed when they are consumed. This allows a Kafka topic to support multiple subscribers, each of which receives all messages.

In addition to multiple subscribers, message retention allows topics to provide a stronger delivery guarantee. Since all past messages are still in storage, a topic can determine whether a message is a duplicate. It can ignore the duplicate, preventing it from being sent to the subscribers. This level of guarantee is referred to as *exactly once delivery*.

Duplicate detection only lasts as long as the messages are persisted. Not all topic implementations store messages indefinitely. Kafka topics, for example, have a configurable retention period that defaults to 7 days. If a duplicate message arrives after the retention period expires, then it will be sent to subscribers.

For immutable architectures, at least once delivery is sufficient. Such applications are based on data structures that persist historical facts indefinitely. If a duplicate is received, it will be detected, as the fact is already in storage. And since records are identified by their content-based address, collisions are prevented at the storage level. Even though a durable protocol might offer exactly once delivery guarantees, enabling that configuration might prove to be as expensive as it is unnecessary.

Message Processing

In addition to delivery guarantees, we must also consider the timing of message processing when evaluating communication protocols. *Synchronous* protocols require that the message be processed immediately upon receipt, as the peer is actively waiting on the result. *Asynchronous* protocols allow the recipient to process the message later. These protocols tend to offer greater autonomy, as remote nodes are not waiting on one another.

An application based on immutable architecture tends to value autonomy over most other factors. Each node has precisely the subset of information that it needs to support the decisions that it and its users will need to make. Therefore, asynchronous protocols tend to be preferred.

Most Protocols Are Asynchronous

This choice between synchronous and asynchronous message processing is not completely isolated from the choice of delivery guarantee. A protocol offering only best-effort delivery is not going to inform the client node about the success or failure of the message. It is certainly not going to wait for the server to process the request. These protocols therefore only support asynchronous message processing.

On the other extreme, protocols that offer durability guarantees will make that promise immediately upon storing the message on the client side. The actual communication with the server might take place shortly thereafter or might be deferred for a long period of time. The protocol has no way of signaling back to the client application that the message has been delivered. Such protocols therefore typically do not require that the remote peer process the request immediately and tend to be asynchronous in nature. Immutable architectures favor these kinds of protocols.

Only in the middle, where the client application receives a delivery confirmation, does it make sense to require synchronous message processing. The client application is actively waiting on a response. That response could well include the results of processing the message.

HTTP Is Usually Synchronous

HTTP by default is a synchronous protocol. When a client sends a request, it waits for the server to make a response. That response is both a delivery confirmation and the results of the message processing. HTTP response codes include such information as whether

a resource was created (`201 Created`), whether the client was authorized to access that resource (`403 Forbidden`), or whether the processing resulted in a conflict (`409 Conflict`). These responses imply at least some degree of synchronous processing.

HTTP does not, however, require that the server process the response immediately. Some HTTP response codes (most notably `202 Accepted`) are intended to reflect that the processing will happen asynchronously. In this case, information about the outcome of the process is not included within the response. It only serves as delivery confirmation.

In the current application landscape, most traffic over the public Internet is based on HTTP. Asynchronous protocols are not quite as popular outside of the firewall. AMQP can be tunneled over TLS and is sometimes exposed on the Internet. But more frequently, it is kept secured within an organizations datacenter, or exposed on the boundary between organizations. Mobile applications favor HTTP over other protocols, and browser-based clients use HTTP almost exclusively. Perhaps in the future, using asynchronous protocols on the Internet will become more commonplace. But for now, attaching a public client to a server usually involves HTTP. But this does not mean that we have to use it synchronously. Even request/response protocols can be used asynchronously.

Data Synchronization

The word "synchronization" is another unfortunate term when applied to data. Synchronization literally means to make two systems progress at the same time, or at least the same rate. Two people can synchronize their watches so that they both read the same time. But synchronizing data is specifically *not* about time. The goal is autonomy, not synchrony. What we seek when we synchronize data is *consistency*. If you ask two nodes the same question, they will give the same answer. They can do this because they have the same information, not because they are operating at the same time.

Nodes in an immutable architecture have a subset of data at their disposal. This allows the users and processes on that node to make decisions without consulting other nodes. The procedure that we refer to as *data synchronization* is just the process of exchanging immutable facts with peer nodes so that their data structures converge. Each node will have only the subset that it requires, but where those subsets overlap, the rules of conflict-free replicated data types (CRDT) guarantee that consistency has been reached when the procedure is complete.

Building on top of immutable data structures, we can now decide independently which protocols to use in this procedure. What kind of delivery guarantees do we require? Should messages be processed synchronously or asynchronously? Are we restricted to common open protocols, or can we choose bespoke options with more desirable characteristics? Will peers be addressable, or will we have to wait for them to call us? Will nodes be permanently connected, or will they connect only occasionally?

To answer these questions, we will examine three main use cases. Each of these represents a different communication structure that is commonly found in immutable architectures. Each one requires a slightly different set of protocol choices.

For communication between servers within an organization, we will favor less ubiquitous but more asynchronous protocols such as AMQP and Kafka. This will help us to build an immutable microservices architecture. For communications between organizations, we will instead favor the more common REST APIs and webhooks, leading to lower infrastructure coupling. And for occasionally connected clients like mobile apps and progressive web apps (PWAs), we will use HTTP as an asynchronous protocol.

Within an Organization

Data synchronization within an organization is a bit of a luxury. One group controls all of the servers, all of the data stores, and the entire network. We have the luxury of selecting our preferred tools, meaning that we can use AMQP or Kafka if we choose. We also have the luxury of a fast, always-available connection between microservices. We will not abuse this luxury by calling from one to the other on every request, but we can keep their data stores synchronized.
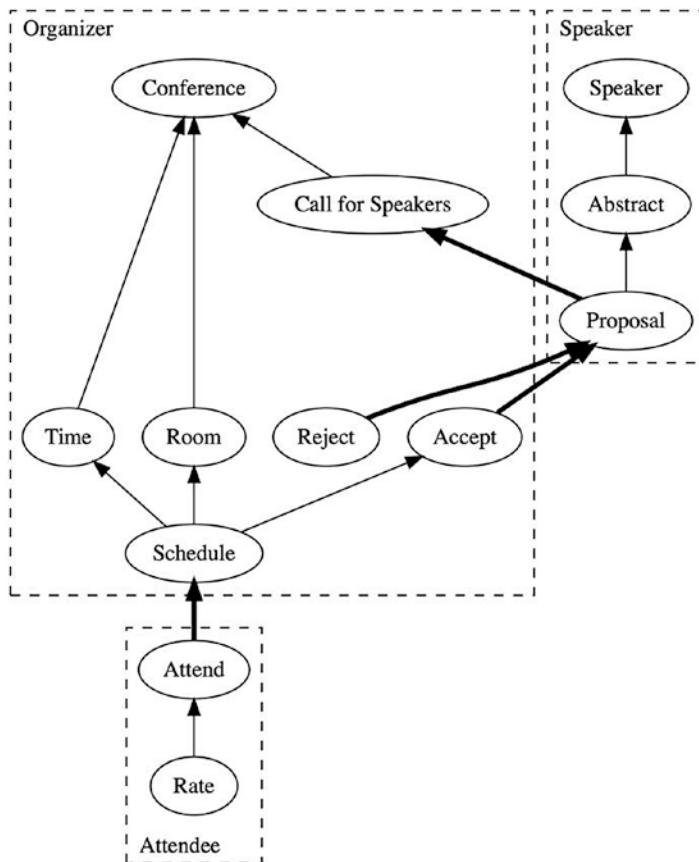
With this kind of luxury, it is easy to get complacent. Intraorganizational architectures will sometimes share databases between services. They will often relax security controls within the firewall. And they will ignore versioning concerns, since they could deploy both sides of a connection at the same time. Each of these compromises to architectural integrity comes at a cost to future flexibility. They increase the coupling between services for the sake of convenience. When deploying microservices within a single organization, you can take advantage of the luxuries that you have while still avoiding unnecessary coupling.

To understand exactly *how* we are going to synchronize data between microservices, we must first determine *what* they are. Then we can analyze the boundaries between them to decide the best means of integration. The outcome of the analysis from Chapter 5 is your guide to where the boundaries should be drawn.

Pivots

When producing a historical model, we identified regions. These were areas of the model in which all of the facts originated from a particular actor. When a predecessor/successor relationship crosses a boundary between two regions, two actors are collaborating with one another. I call such a predecessor/successor relationship a *pivot*.

The diagram that we used to introduce regions is reproduced in Figure 11-4. I have highlighted the pivots.

Figure 11-4. *A model highlighting the pivots, where arrows cross region boundaries*

During the analysis phase, a region represented the responsibilities of a single actor or set of actors. As we transition into implementation, we will construct a microservice for each audience. And so each region now represents a microservice. In this example, conference organizers have a microservice for collecting proposals and defining

schedules. Speakers have a microservice for viewing calls, submitting proposals, and learning about acceptance. Finally, attendees have a microservice for viewing a conference schedule, selecting sessions to attend, and submitting ratings. The pivots are points of integration among these microservices.

The microservice at the head of a pivot needs to publish the predecessor fact so that the microservice at the tail can subscribe to it. Let's begin with the topmost predecessor in the causal chain, the *call for speakers*. This fact is in the *organizer* microservice.

Multiple Subscribers

Pivots at the top of the causal chain tend to be places where facts are published for multiple subscribers. The publisher might not have one specific use case in mind, and future subscribers could be added at any time. But even when there is a known use case, as in this example, sending a message to a specific subscriber introduces unnecessary coupling. And so top-level pivots are good candidates for topics, such as those provided by Kafka.

The microservice at the head of the arrow publishes a message to a topic when the predecessor fact is created. This message includes all of the information contained in the fact and all of *its* predecessors. To compute the set of all facts included in the message, perform a *transitive closure* over the predecessors. Recursively visit the predecessor relationships until the entire set is gathered. The message should contain all of this information, and *only* this information.
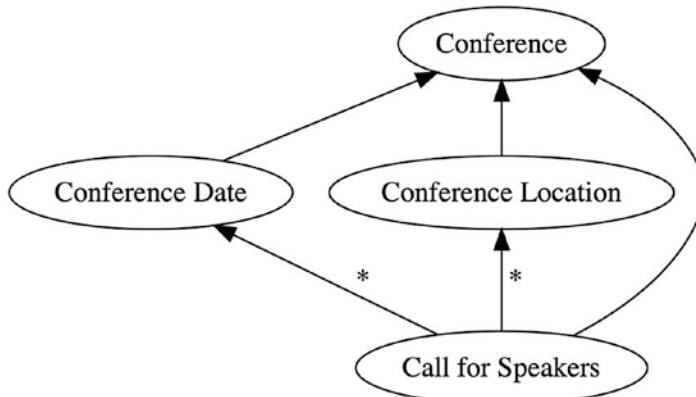
Two problems arise when publishing a message that contains more than the transitive closure of the predecessors. The first is that message is not deterministic. If the message contains internal database IDs, the time of creation, or any other detail not already part of the facts, then running the process again produces a different message. The process could be repeated for any number of reasons: there was an infrastructure glitch, the fact was produced by two redundant instances, the user clicked the submit button twice, and so on. If any of these situations arise, we want the process of generating a message from a fact to be deterministic, so that the downstream consumer can practice idempotence and ignore the duplicate.

The second problem is that the message might contain information in successor facts. Successors can be created either before or after the message is published. If the message contains successor information, then subscribers will not learn of new successors created after the fact. In the example in Figure 11-4, the transitive closure of the *call for speakers* fact includes the *conference*. It does not include *times* or *rooms*.

If speakers needed to know (for some reason) the number of rooms at the conference, this information might or might not be available at the time of publication. If a room is added later, they will not learn about it.

There are some successors that you will *want* subscribers to learn about. For example, the *conference date* (not shown in Figure 11-4) will be an important part of knowing whether to submit a proposal. Given that that is likely to be a mutable property of a conference, it will be modeled as a successor. Analysts have two options for resolving this problem: they can turn the successor into a *predecessor* of the published fact, or they can publish the successor.

To turn the successor into a predecessor, apply the *Transaction* pattern described in Chapter 8. An example is shown in Figure 11-5. The published fact is a transaction that brings together all of the successors that are current at the time of publication. This brings those successors into the transitive closure. An organizer can change the date or location of the conference after publishing a call for speakers, but they now have a clear indication of the information that a speaker had when they proposed their sessions. They can use this information to publish a new call for speakers and contact the speakers who replied to the earlier one.
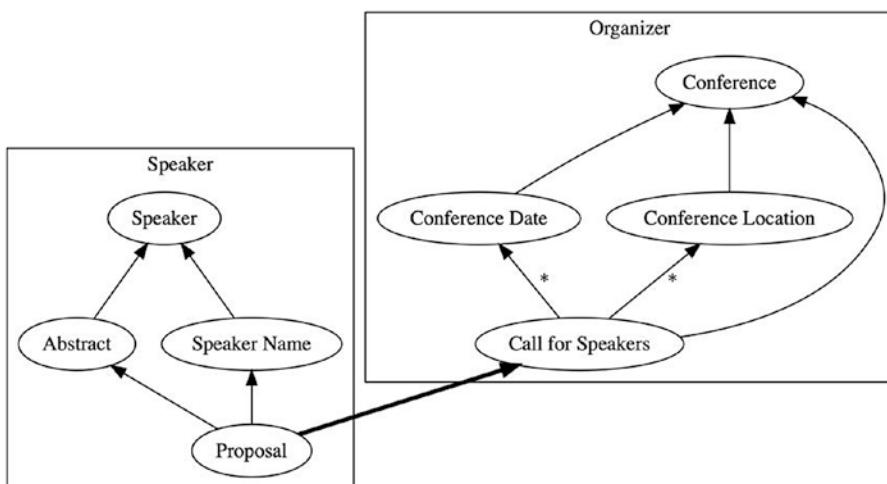
Figure 11-5. *A call for speakers is a transaction that captures the current date and location*

To publish the successor, produce an additional topic. Subscribers to the pivot can also subscribe to this topic. They will correlate the messages between the two topics using their common predecessor, in this case the *conference*. If the conference date is changed after the call for speakers has been published, then the subscriber will see that change and update their data store.

Responses

Facts on the tail of a pivot represent responses to messages. Responses are directed back toward earlier microservices. It therefore makes sense to use queues instead of topics for these kinds of messages. The producer of the original message includes the name of a response queue. Subscribers post response messages to the given queue. This manages coupling between publisher and subscriber, because the queue name is provided dynamically. In Figure 11-4, the *proposal* fact is a response to the call for speakers. It is directed toward the *organizer* microservice. That microservice will therefore create and manage a queue specifically for accepting proposals.

The response message, like the original message, is composed from the transitive closure of the fact. In this case, that means that the *proposal* contains information about the *abstract* and the *speaker*. If successors are required (such as speaker name), then the response should follow the Transaction pattern, as shown in Figure 11-6. Notice also that the message contains the *call for speakers*. The original message is a predecessor and will therefore be part of the transitive closure. This gives the original microservice sufficient information to correlate the responses.

Figure 11-6. *A proposal gathers together all of the facts that will be necessary for the organizer to make a decision*

Since the speaker microservice knows about the *call for speakers* topic, it is tempting to also have it know about the *proposal* queue. Doing so would seem not to increase coupling between the two services. But that would be a mistake. Only the organizer

microservice knows *how* it will respond to proposals. It might change the topology in a future release. It alone should be responsible for deciding where the response queue is located.

Pivots further down in the causal chain also represent responses. In Figure 11-4, the *reject* and *accept* facts are responses to the *proposal*. These kinds of responses should follow the same pattern. When the speaker microservice generates the *proposal* message, it includes the name of *its own* response queue.

Notifications

Not all responses are direct successors of pivots. And not all two-way conversations between microservices appear as arrows crossing region boundaries in both directions. Sometimes the relationships are hidden a little deeper in the model.

Every conversation between microservices ends with a message that has no response. This message serves only as a *notification*. It informs the recipient of the outcome of a process. These appear as leaves in the model below pivots. The *rate* fact is an example in Figure 11-4.

When a conference attendee rates a session that they have attended, they are simply giving feedback to the organizer. They do not expect any response to that rating. There is therefore no further pivot below the *rate* fact indicating that the conversation continues. The rating is pushed to a queue that the organizer provides, just as any other response. The name of that queue will be included in the *schedule* message, as that is the predecessor of the nearest pivot.

Between Organizations

When servers are not under our direct control, we lose a bit of the luxury that we might have had within a single organization. We can no longer choose from all available protocols to select the best possible fit. And we don't have any governance over the way in which peer systems will be modeled. Partners might not even be using immutable architecture. We adapt by implementing additional constraints and transforming our services to be more familiar.

One of the constraints of crossing organization boundaries is that the communication protocols need to be supported by both sides. This usually means that asynchronous protocols like AMQP are replaced by synchronous protocols like HTTP. The timing of message processing is not the issue; it is simply adoption. The more

371

widely adopted protocols today tend to be those that support synchronous processing by default. A compromise can often be reached by using HTTP in an asynchronous manner.

Async over HTTP

External organizations will often need to publish messages to your services. Semantically, these are *commands*, instructing your service to perform some kind of business function. In a historical model, these are simply predecessors of a pivot, created in a remote region of the model. If we were working entirely within the scope of a single organization, we might choose a topic or queue to publish these messages. But since we are providing an endpoint for a partner, we will instead use HTTP. We can design the endpoint with additional constraints, not part of the HTTP specification, to make it work well with immutable architectures.

According to the HTTP specification, POST is neither required to be safe nor idempotent. However, an endpoint provided to partner organizations will clearly benefit from idempotence. This does not rule out HTTP POST. It only means that we implement the server to uphold stronger guarantees than the specification requires.

First, we ensure that the body of the message contains enough information to generate a unique identity. When we receive this request, we will generate a fact. The contents of this new fact need to be completely determined by the contents of the message. We will not use the time of receipt, a server-generated ID, or any other nondeterministic data to produce this fact. This guarantees that if the partner repeats this request, they will generate the same fact. That is the first step to making POST idempotent.

Second, we generate the URL of the resource using only information from the new fact and computing the transitive closure of the new fact to find the graph of all predecessors. Pull fields from these predecessors, and assemble them into a path. Append that path to the host name of the exposed endpoint to compose the URL. Assuming that we have used all of the fields, this generates a one-to-one mapping between facts and URLs. When the partner makes a subsequent call to the endpoint, we will be able to pull the components out of the path and reconstruct the fact.

And third, respond to the POST immediately after the new fact is stored. Do not wait until the request is processed. Before storing the fact, you will have the opportunity to run the authorization rules to make sure the partner is authorized to make this request. But there is no need to wait until the request is processed. You can complete processing asynchronously.

An endpoint implemented according to these constraints will be idempotent. Any subsequent POST of the same request will yield the existing fact. Because the service is using content-addressed storage, it will recognize that the fact already exists. It simply responds with the same URL as it had originally produced.

Such an endpoint is also durable. It does not respond until the fact is stored. A side effect of storing the fact might also be adding it to a topic or queue for further processing. The delivery confirmation of 201 Created indicates that this storage has occurred and has been committed. The sender may stop sending at that point; the message has been saved.

Finally, this endpoint is location independent. The URL does not contain any server-generated IDs. If the request had been handled by a different server, it would have produced the same URL and the same fact. We are free to reorganize our infrastructure, fail over to a backup datacenter, or mirror requests to different geographical regions. None of these implementation details will be visible to our partners.

Webhooks

If our infrastructure were completely within our control, we could just post responses to a queue. When working with partners, though, we sometimes don't have the luxury of using queueing protocols. Yet we still want to pass the names of queues across organizational boundaries to reduce coupling between peer services.

The equivalent of a response queue in HTTP is a webhook. A webhook is an HTTP endpoint intended for use as a callback, a place to which to send responses. One service registers a webhook with another by providing an endpoint URL. The other service POSTs to this endpoint whenever there is new information to report about the topic.

A response in a historical model appears as the immediate or eventual successor of a pivot. We should generate webhooks based on the pivot's predecessor. As described previously, compute the transitive closure of the predecessor and extract all fields of those facts. Construct a path and append it to a host name. That URL can now be used as a webhook. The service listening at that host can reconstruct the predecessor from the path.

Since the path contains all of the information necessary to reconstruct the predecessor, the body of the message does not need to include it. The body is all of the information necessary to create the successor fact *except* for the predecessor identified in the path. The service handling the webhook will follow all of the constraints of the command endpoint described earlier to ensure that responses are idempotent, durable, and location independent.

Emulating REST

In many integrations, an organization that has adopted immutable architecture will be integrating with one that has not. We might not have the luxury of defining the API so that it works well with immutability. We might have to adhere to an API that the partner has defined or provide one that is more familiar to them. In those situations, we can both consume and implement REST APIs from immutable services.

To consume a REST API from an immutable model, apply the Outbox pattern as described in Chapter 8. The Outbox pattern creates a bridge between a historical model and a third-party API. The caller maps facts that the partner needs to know about into API calls. They record a journal of the responses from those API calls indexed by the hash of the facts. While this pattern cannot turn a REST API into an idempotent, durable data exchange, it provides at least a little protection against infrastructure failures. The rest is up to the partner.

To produce a REST API with an immutable model, we apply the Structural patterns in Chapter 8 to map all of the incoming requests into semantically equivalent facts. A POST maps to an Entity fact and likely one or more Mutable Property facts. A PUT or PATCH maps to one or more Mutable Properties. A DELETE generates a Delete fact. Based on the semantics of the domain, other patterns could be brought into play.

Where possible, generate URLs as described previously using only information found in the transitive closure. Ideally, all of the information needed to generate the fact will be present in the request. That would produce a truly idempotent API. However, this will not always be possible. In particular, Mutable Property facts cannot be generated based only on the desired value of those properties. They need to know their predecessors, which is not something traditionally given in a REST API.

To find the predecessors of a Mutable Property, the service will need to run a query. Find all facts that have not been superseded.

```
query valuesOfProperty(e: Entity) {
  match p: EntityProperty where p.entity = e
    such that not exists n: EntityProperty where n.prior = p
}
```

If the query results in one fact, and the value of that fact matches the desired value in the PUT or PATCH, then ignore the request. The property already unambiguously has the desired value. But if the number of results is *not* 1, or the current value is different, then create a new property fact having these results as predecessors. This algorithm

374

allows the client to resolve conflicts by putting the desired value. Unfortunately, it does not capture what the client *actually believed* the original value to be and therefore record the real causal graph. Only a client participating in the immutable model and using an appropriately designed API could do that.

To GET a resource from a historical model, you will need to run several property queries. Generate the starting entity fact from the URL as described previously. Then run queries for all properties that you intend to return. If any of those queries returns more than one result, apply a conflict resolution function to determine the desired result. REST consumers are not used to properties having more than one value. Do *not* save the results of your conflict resolution. GETs are supposed to be safe.

A REST API produced from a historical model will compromise some of the benefits of immutability. It will only be as idempotent as a traditional REST implementation. It will not have the commutativity guarantees of an end-to-end immutable architecture. But it will be more familiar to partners who have not yet adopted these strategies.

Occasionally Connected Clients

The third common scenario for integrating with an immutable architecture is to support offline mobile or web clients. Whereas most mobile apps and websites in use today must have a continuous connection to a back-end API, an offline client can interact with the user even when that connection is interrupted. They have their own storage, their own outbound message queue, and can participate in conflict detection and resolution. Native mobile applications have storage capabilities from the operating system; web clients can use advanced browser features, operating as progressive web apps (PWAs).

Mobile and web applications designed to be used in this way are typically *offline first*. All of the data presented to the user is loaded from local storage, not an API call. Every user action is stored locally and pushed to a queue, not sent to the server. Synchronizing local storage with server history takes place in the background. The user can see the progress of that activity, but they are not blocked by it.

Occasionally connected clients will greatly outnumber servers. They will come online with nothing more than a download or a bookmark from the user. They might be used for a long period of time, or they might be visited once and quickly abandoned. When a client leaves the ecosystem, the server will not receive any notification. It would therefore be wasteful for servers to keep track of meta-information on behalf of the clients. The protocol for synchronizing an occasionally connected client puts the storage burden entirely on the client.

375

Client-Side Queue

As the user interacts with the client application, it will generate facts. These facts will be stored in its local subset of the historical graph. They will also be added to an outgoing queue. The user is permitted to continue interacting with the application as soon as the fact is stored and the message is queued. They do not have to wait for it to be sent to the server.

Mobile applications can use a local SQLite, Core Data, or Realm database for both fact and queue storage. To design the fact storage, see all of the advice given in Chapter 10. The outgoing queue is simply a record of which facts have not yet been sent to the server. It could be as simple as a table of foreign keys into the fact storage.

Progressive web applications can use IndexedDB to store facts and queues. This browser feature is not as rich as a SQL database. Instead, it is simply a set of name/value pair collections. Consider using one collection per type of fact. The keys of these collections are the hashes of the facts. In addition, the PWA has a collection for the outbound queue, indexed using a monotonically increasing key.

To send the outgoing messages to the server, the mobile application or PWA calls an HTTP endpoint. This is *not* a RESTful endpoint providing the usual semantics of POST, PUT, PATCH, and DELETE. That kind of endpoint compromises the value of an immutable architecture and is intended for use by clients that do not participate in the historical model. Instead, this is a more constrained endpoint to which messages can be POSTed in an idempotent and commutative way, as described for intraorganizational command transfer.

To reduce latency and make the most efficient use of the network, clients will batch several outbound facts into a single request. The contents of a POST will be a collection of facts of various types. My favorite way to encode a batch is as a JSON object in which the keys are base-64 encoded hashes. This makes it easy for the server to find incoming facts by their hash and helps to ensure that a fact is not unnecessarily duplicated within the same batch. The body of each fact contains the type, the fields, and the hashes of its predecessors.

Assuming that the predecessors were already known to the server before the upload began, it would have no trouble finding them by their hash and establishing the link in its own database. However, this assumption cannot be guaranteed in practice. A client might not be talking to the same server from one session to the next. Servers may be spread across different datacenters to gain redundancy or geographic proximity.

It is therefore wise to include the transitive closure over the predecessors of all outgoing facts. This is why it is important to eliminate unnecessary duplication within the batch.

When the server receives the batch, it must store each of the facts in turn. Storing a fact requires executing authorization queries and setting up foreign keys. For those reasons, the server must have already stored the predecessors. It therefore processes the incoming batch in *topological order*. It recursively visits all predecessors before handling each message. When it visits a fact, it first looks in a temporary data structure to see if it has already visited that fact. If not, it verifies the hash, then looks in its own database for that record. If it is present, it moves on. If not, it runs the authorization rules and saves the new fact.

When it is done, the server responds with a 200 OK message. After that, the client can delete all facts sent in the batch from its outbound queue. The client continues until the queue is empty.

Client-Side Bookmark

Because clients outnumber servers, all of the meta-information is kept on the client. This includes the outbound queue that we just discussed. And it also includes information about *inbound* facts. Rather than keeping a per-client queue on the server, each client keeps its own *bookmark*.

A bookmark is a placeholder within a sequence of facts. It identifies the last fact that the client has received and stored. The client can ask for a batch of facts *greater than* a given bookmark, and the server will respond with both a collection of facts and a new bookmark. That new bookmark corresponds to the last fact in the batch.

Because we need to know which facts came after a given bookmark, these identifiers must be *totally ordered*. A total order is one that allows us to compare any two elements in the sequence. We can tell for sure whether one element is before or after another. In every other sense, however, facts are *partially ordered*. You know that a predecessor came before a predecessor, but you cannot compare two facts that are not causally related. Furthermore, facts are usually identified by their hash, which does not obey any kind of order. We therefore need a new method of identifying facts for use with a bookmark.

The identity of a fact within a sequence must be monotonically increasing. Later facts cannot be given identifiers less than or equal to earlier ones. If that were ever violated, then a client using a bookmark from an earlier fact would miss later facts on subsequent requests. Timestamps alone are not sufficient for this purpose, as two facts

could be stored at exactly the same time. An auto-incremented ID is the best choice. Even then, extra precautions must be taken to avoid reading a later ID before earlier IDs have been committed. One such precaution is to remove facts from the end of the batch until one is found that is old enough for concurrent writes to have settled. This implies that clients might not receive the absolute latest information until a subsequent read, but it mitigates against writes that happen out of order of ID allocation.

Imposing a total order on a partially ordered collection has a serious drawback. It means that bookmarks are location specific. If the mobile device or PWA were to connect to a different server on a subsequent fetch, the bookmark that it received from the last fetch would be meaningless. Different back-end nodes might have put the partially ordered facts into different total orders. For this reason, the client needs to keep a separate bookmark for each data store it contacts.

A datacenter having a load-balanced cluster of servers all sharing a common database is not a concern. No matter which server the client uses, the shared database generates the monotonically increasing IDs. The issue only arises when servers use different data stores. So bookmarks are really per database, not per server. Each database should generate its own unique identifier and use that to distinguish its bookmarks from those of other databases.
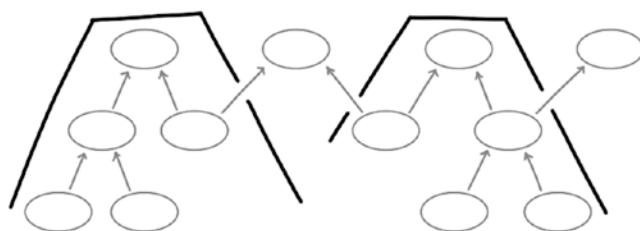
The client sends all of its bookmarks with the request. The server determines which bookmark is associated with the database it is using. If the client has no bookmark for that database, it starts at the beginning. The server then responds with a batch of facts, the database ID, and the new bookmark. The client stores all of those facts and updates its bookmark for that specific database. It repeats until the request yields no new facts.

In most network topologies, including a database ID is an overabundance of caution. The entire population of mobile clients can be served from a single database. As long as that remains true, then the clients will each have one bookmark that marches steadily forward. However, if the day ever comes that the database needs to fail over to a standby, then there is no guarantee that the order of insertions will be consistent between the two. Clients will find themselves redownloading the data set, but they will be able to detect and ignore duplicates. They will also be guaranteed not to miss any information as a result of data stored in different orders across different databases.

Choosing a Subset

A mobile or PWA client rarely needs to fetch the entire contents of the data model. These clients will have a single user, and that user will have access to only a subset of the data. Occasionally connected clients should fetch only the facts that their user needs. Based on an understanding of the model and how it will be used, we can divide facts into subsets. A particular user will have access to a small number of these subsets. The client will therefore need to keep track of separate bookmarks. It has one bookmark per database per subset.

A subset of a model can be defined by a single root fact. The subset includes all of the direct and indirect successors of that root. Imagine a cone extending down from the root and gathering together everything it touches, as illustrated in Figure 11-7. That is the subset of the model that the user needs to interact with that root.
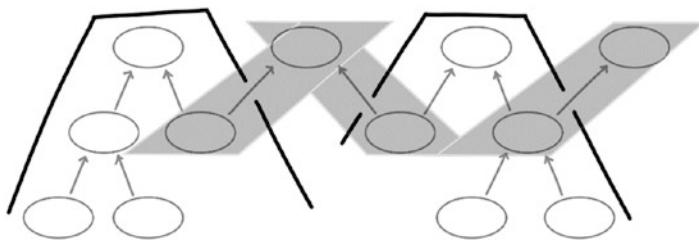
Figure 11-7. *A subset of a model is the cone of direct and indirect successors of a given root*

There are two kinds of facts that make for good subset roots: groups and periods. A group is a top-level fact participating in the Membership pattern defined in Chapter 8. A membership fact has two predecessors: a group and a user. It grants the user membership into the group and therefore access to its resources. Membership facts often determine authorization and distribution rules. A user need not see facts outside of the groups of which he is a member.
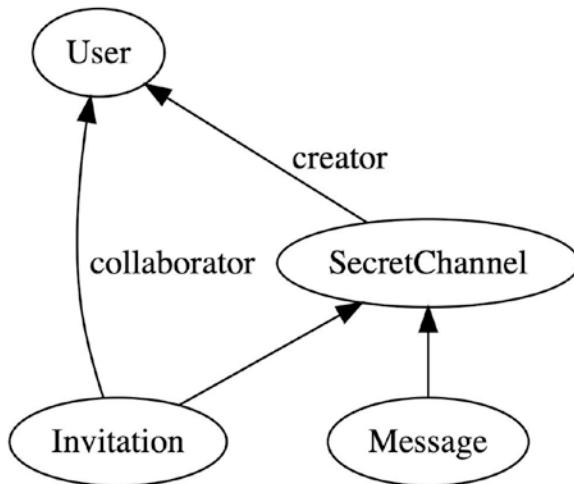
A period is a near top-level fact participating in the Period pattern. This fact breaks successors down across time. A natural clock within the problem domain moves forward and points to the current period. This might be a date of business at a restaurant or a semester at a school. New facts are added to the current period. So client apps can focus only on recent periods and ignore older ones.

When a user interacts with an occasionally connected client, they only need the successors of the groups to which they belong and the most recent periods. But to understand a successor, they also need the transitive closure of its predecessors. For this reason, the subset that is actually downloaded to the device includes predecessors of those successors. The cone bounces back up the graph, forming a lattice structure as illustrated in Figure 11-8.

Figure 11-8. *The facts downloaded to a device include the cone of successors of a root and all of their predecessors*
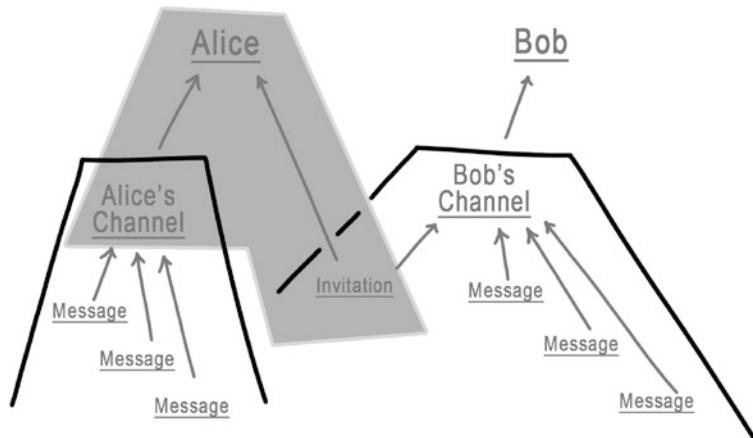
Recall the example of the secret channel that we studied in Chapter 7. In this example, the creator of a secret channel sent an invitation to their collaborators. Members of the channel could then exchange messages with one another. The diagram is recreated in Figure 11-9.

Figure 11-9. *A secret channel is a group to which collaborators are invited*

If we were to construct an occasionally connected mobile app or PWA for this model, `SecretChannel` would be an excellent choice of a fact that identifies a subset. If the user of the app is the creator or a collaborator in the channel, then they would expect all of the messages to be downloaded to their device. The group defines a subset of the graph containing the transitive closure of its successors.

This example illustrates one more root fact and one more rule of subsets. The user themselves should be the first root. This gives them a subset of all of the groups that they have created or been invited to join. That subset, however, should stop at the roots of other subsets. In Figure 11-10, for example, Alice both created a channel and was invited to Bob's channel. The subset with Alice as the root includes her channel and the invitation, but does not include the messages in her channel. She specifically needs to pull those messages from that subset.

Figure 11-10. *The subset of facts under Alice includes her channel, but not the messages in her channel*

The algorithm for identifying facts within a subset boils down to a recursive traversal of the graph. Start at the identified root. Recursively visit all successors of that fact. Add that successor and its transitive closure of predecessors to the subset. If the visited fact is not itself a root, continue with its successors. This will give the set of facts that an occasionally connected client needs to serve its user.

Each client keeps a different set of bookmarks for each root. When it fetches facts from the server, it identifies the root by hash. The server responds with a batch of facts that are in that subset. In the case of the model in Figure 11-10, the first root is the logged in user. That fetches the channels that they have created and the invitations that have

been sent to them. With that information, the client makes additional requests for each of those channels. This fetches the subset of messages that the user can see. And each channel has its own bookmark.

Avoiding Redundant Downloads

With queues for uploading facts and bookmarks for downloading them, we are starting to construct an algorithm for background data synchronization in an occasionally connected app. But as we put the two together, a problem emerges. All of the facts that a client uploads will be appended to the total order on the server. They will be greater than the client's bookmark. That means that they will be downloaded again to the client the next time it fetches. This is a waste of bandwidth.

We would like the client to fetch only the facts that it itself did not upload. We can get close to this behavior by simply performing the download first, then the upload. The client downloads facts greater than its current bookmark. It stops when the fetch returns no new facts. Then it uploads batches from its outgoing queue. It stops when the queue is empty.

At that point—ideally—the only facts greater than its bookmark would be the ones it just uploaded. So if we could update the bookmark without redownloading those facts, we would avoid the redundancy. The problem is that other facts may have been added in the meantime. Other clients might have uploaded their facts, or other processes might have created information that should be sent to the client. And so we cannot assume that we can update the bookmark without missing something that happened concurrently.

A good optimization is to send with the fetch request a list of hashes. These are the facts that the client just finished uploaded. The server will filter out these facts from the response. In the ideal scenario, no new facts have been added, and so the entire download batch is filtered. In this case, the server returns an empty collection and a new bookmark. The client updates their bookmark, and we have avoided redundant downloads.

If, on the other hand, new facts were added concurrently, then the server would return only those new facts. It would also return the latest bookmark, including the new and the filtered facts. Upon seeing that response, the client would store the concurrent facts and update its bookmark. We achieve the correct behavior and avoid the redundant downloads.

This solution keeps all of the meta-information on the client. The client keeps track of the uploaded fact hashes in memory during the background sync operation. The server receives this information in the request and only uses it to filter that response. It does not store any per-client information in order to optimize network usage. And if something fails on the client, then it simply falls back to the correct, if suboptimal, redownloading of facts.

Generated Behaviors

From the beginning of the software industry, people have been responsible for writing programs. A large number of decisions go into every program written. People have to be vigilant not to introduce defective behavior as they make those complex interconnected decisions.

Over time, the level of detail implicit in those decisions has risen to higher and higher abstractions. Early software developers worked at the machine level. Modern software is increasingly written in higher-level languages. In the past few decades, managed runtimes have improved in performance and productivity to the point that it is becoming harder to justify writing code to the metal. Developers are making fewer and fewer detailed decisions. As we continue up the ladder of abstraction, we can use our new vantage point to look across to other islands.

Software is written in islands of behavior. The database is designed to store and query data for a specific application. The business logic operates on objects in memory. The user interface presents data and responds to user inputs. Network APIs handle security and communications. Modern application development is an exercise in bridging those gaps for every application. Between the database and the business logic, developers construct a data access layer either by hand or with an object relational mapper (ORM). Between the business logic and user interface, developers write view models. And between the business logic and network, they design custom controllers, actions, and proxies for bespoke APIs.

Immutable architecture gives us an opportunity to change that. Rather than custom islands of application behavior, we can construct a runtime of application-agnostic components. An immutable runtime will bridge the gaps with generated behaviors. We have already shown how a few of these generic components could work. A generic data store persists facts declared in the Factual Modeling Language and executes generated query pipelines. Those same pipelines are automatically inverted to inform the user interface when a new fact arrives from a network peer. Authorization rules determine which of those incoming facts can be trusted and stored.

With each technical decision that we automate, we connect islands of the system that were once developed separately. Only a couple of gaps remain to be bridged. For one, we need to project the results of a query onto the user interface. And for another, a server must determine which facts to share with a peer. These two gaps can be closed with two final concepts: projections and interest. Once we close these gaps, we will transition from a purely human-driven development paradigm to one that supports higher-level reasoning. We will gain not only productivity and confidence but also a new degree of communication and collaboration.

Projections

Queries in the Factual Modeling Language produce sets of facts. While these result sets are useful for a developer, they are not approachable in their raw form to a user. An immutable runtime needs to transform raw facts into objects that the user can more easily consume. To bridge the gap between query results and user interface, we define a *projection*.

A projection is a function that maps a group of fact sets into an object structure. The front-end framework can take it from there to produce the user interface. A functional framework such as React would express UI components as a function of the object structure. A data-binding framework such as XAML would use the object structure as a view model. In either case, the framework starts with a projection of the facts, not the facts themselves.

Projections raise the level of granularity. They aggregate several small disparate facts into a cohesive representation. A single query produces a set of facts, all of the same type. These might be all entities that should appear on a list or all possible values of a single property. The user interface, however, will need to combine several result sets. To produce a list, it will need not only the set of entities but also the candidate values of several of their properties.

Defining Projections

Consider the example of a restaurant host viewing the set of available tables. Within that list, they will want to see the table number, the capacity, and the name of the server assigned to the table. To get the list of tables, the application will run the following query:

```
query tablesAvailable(r: Restaurant) {
  match t: Table where t.restaurant = r
    such that not exists s: SeatParty where s.table = t
      such that not exists b: BusTable where b.seatParty = s
}
```

Once the application has the table entities, it will need to get the properties. Some of these properties are immutable and therefore directly available within the Table fact. For example, the table number and seating capacity are stored within the fact and do not change. But some properties will change over time and therefore will only be visible within successors. These will require additional queries. For example, to display the name of the server currently assigned to the table, they will need to run the following query for each entity:

```
query serverAssignedToTable(t: Table) {
  match a: Assignment where a.table = t
    such that not exists d: AddignmentDelete where d.assignment = a
  then s: Server where s = a.server
}
```

The application could perform the second query for each result returned from the first. Doing so would take longer than necessary. This kind of performance problem is commonly known as *SELECT N+1*. The application executes 1 query to find the entities and then makes *N* queries to find some property of each of those *N* entities.

It would be far more performant to combine those queries into a single pipeline. That is precisely what an immutable runtime would do with a projection. We can project the two previous queries into a single structure. Such a projection can be represented with the following pseudocode:
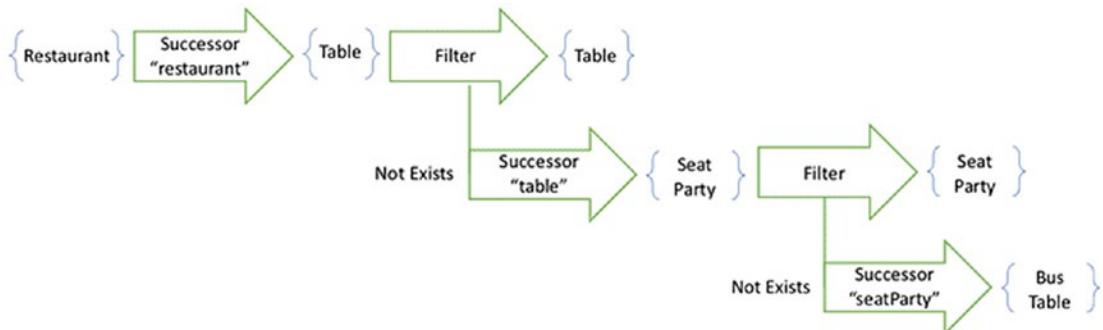
```
projection seatingView(r: Restaurant) {
  tables: for each t: Table in tablesAvailable(r) {
    tableNumber: t.tableNumber,
    capacity: t.capacity,
    server: for each s: Server in serverAssignedToTable(t) {
      name: s.name
    }
  }
}
```

The exact expression is going to differ for each programming language. For example, in Jinaga—an immutable runtime for JavaScript—a projection looks like a JavaScript object literal where the fields are defined using functions. The important part about this pseudocode is that it declares an object structure based on a handful of queries. If this expression were evaluated imperatively, we would have a SELECT N+1 problem. But since this projection is expressed declaratively, we have the opportunity to produce a single combined pipeline.

Projection Pipelines

Like a query, every projection has a starting point. The starting point of the preceding projection is a `Restaurant`. The projection then defines a set of fields (in this example, just the `tables` field). Some of these fields are simply functions of the starting point. And some of them, like the one shown earlier, are based on queries.

To execute a projection, a runtime builds a *projection pipeline*. This is an amalgam of all of the query pipelines that appear within the fields. It starts with the pipelines for each of the fields. The pipeline for the `tablesAvailable` query appears in Figure 12-1.
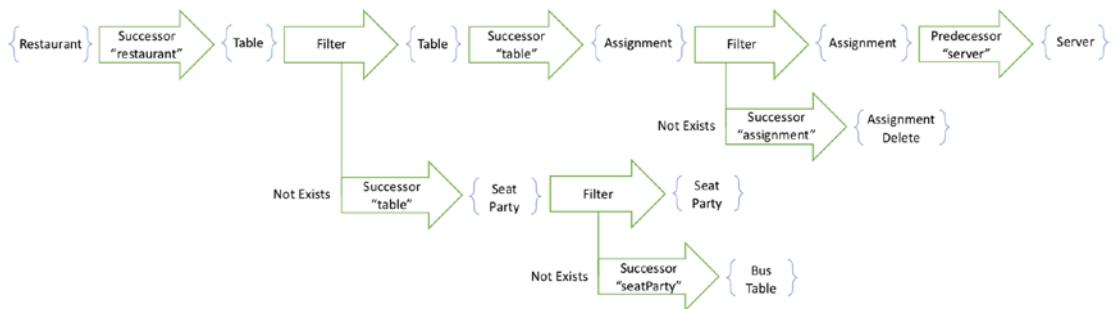
Figure 12-1. *The projection begins with the pipeline to find tables currently available in a restaurant*

If we had more than one field, then we would graft each of the pipelines together at the same starting point. If two fields produced pipelines that followed the same first step, then their pipelines would be grafted *after* that common step. The pipelines share steps until the point that they diverge. An example of a combined pipeline appears in Figure 12-3. In this case, however, we just have the one pipeline, so we continue to the next phase.

In the next phase, the runtime walks recursively down to fields defined on projected entities. In this example, the `table` entity has three fields: `tableNumber`, `capacity`, and `server`. For two of these fields, the expression simply pulls a value from the fact. For the third, however, it specifies a child query.

For each child query, the runtime joins the pipeline to the end of the parent pipeline. The pipeline in Figure 12-2, for example, finds the values of the `server` fields of all available tables.

Figure 12-2. *Join the server pipeline to the table pipeline to find all servers at once*

Projection pipelines can get large and complex. If it were the responsibility of a human developer to maintain these pipelines and write the corresponding database queries, they could easily make a mistake. But this is a process that can be automated. Starting from the declarative projection, a framework grafts several pipelines into a single composite structure. It computes the inverse of this structure to understand exactly what component state or view model to update when a new fact arrives. It transforms this combined pipeline into a database query to be executed more efficiently against the data store. This avoids the SELECT N+1 problem, while producing an object model that readily supports the user interface.

Interest

The final gap to close is the one between the business logic and the network. In Chapter 11, we manually decided how to share facts with peers. We found pivots within the model—predecessor/successor relationships that crossed regions. These became points of collaboration not only between users but also between nodes. Then we decided how to express that collaboration: queues, topics, REST APIs, webhooks, and others. Now we will examine a way to automate that decision and take it out of the hands

of the human software developer. We replace it with an algorithm for automatically determining what facts a node needs to know about. Peers using immutable runtimes will exchange information with peers to express *interest* in the facts that they want.

A node is interested in a fact if the existence of that fact changes the behavior of the node. The behavior of a node depends upon its projection pipelines and their starting points. If a fact influences the results of a projection from an expected starting point, then the node that presents that projection from that starting point is interested in that fact.

To express interest, a node shares its projection pipelines and starting points with its peers. They determine which facts to share based on that information. So which facts could influence the behavior of a pipeline? Certainly the starting point has a great influence. But of course, the target node already knows about that fact. There is no need to send it a fact that it already knows. Instead, the peer examines the projection pipeline to find other facts that could influence it.

Every step that the pipeline takes visits a new set of facts. If the step finds successors in a certain role of a certain type, then every one of those successors could influence the behavior of the pipeline. The node that uses that projection is interested in all of those successors.
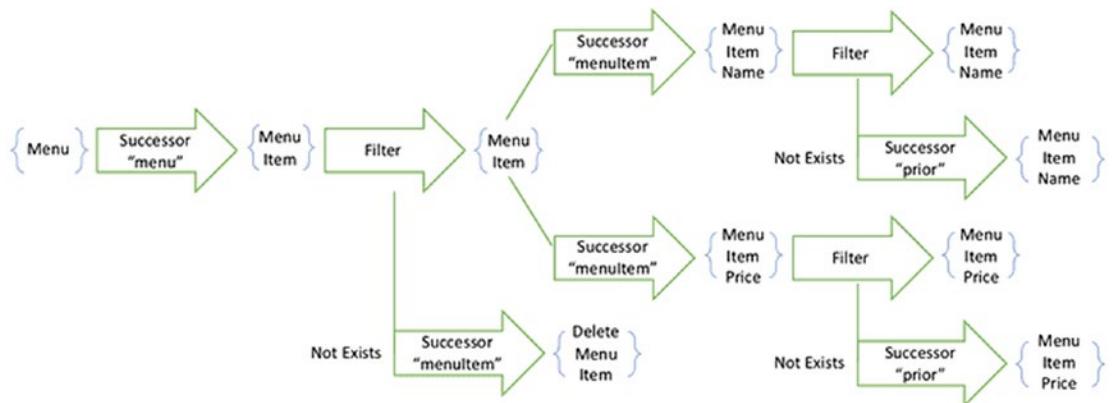
To understand a fact, a node must also know about all of its predecessors. When a node expresses interest in a fact, it implicitly expresses interest in those predecessors. Those facts, of course, have predecessors of their own. The runtime computes the transitive closure of the interest set to find all of the required facts. A predecessor step within the projection pipeline adds no facts to the interest set that would not already be present.

When a pipeline includes a filter, that filter introduces a child pipeline. Any fact that influences the child pipeline influences the behavior of the filter. And so the runtime evaluates the child pipeline recursively. Beyond the filter, the pipeline continues. But only facts that pass the filter can influence the remainder of the pipeline. And so only facts that make it past the filter can further contribute to the interest set. This leads to some interesting and counterintuitive consequences. These are most obvious with respect to two common patterns, both described in Chapter 8: Delete and Period.

Interest in Deleted Entities

The Delete pattern uses a fact to indicate that an entity has been deleted. When a node learns about that fact, it loses interest in all information about the entity—all information, that is, except for the fact that it has been deleted. That has to be preserved.

Consider the projection pipeline in Figure 12-3. It shows the pipeline necessary to display all items on a menu, including their name and price. The first filter in the pipeline excludes deleted menu items. When it eliminates a deleted item, the node loses interest in the name and price properties.

Figure 12-3. *A projection pipeline that gets the names and prices of all items on a menu*

In order to execute that filter, the node needs to know about the menu items and their deletions. It remains permanently interested in those facts. After the filter are the steps that explore the properties of the menu item. *Those* steps will never see a deleted menu item. And so, even though a node is interested in the entity and its deletion, it is not interested in the properties of deleted entities.

This is a subtle and important optimization of the interest set. You might think that a node will not be interested in the deleted entities at all. Once an entity is excluded by the deletion filter, then it no longer appears on the user interface. If it does not appear, then why would a node be interested in it?

The reason for the continued interest becomes clearer as you consider how interest propagates through a network of nodes. Suppose a new node comes online and does *not* learn about deleted menu items. Then suppose that it communicates with a peer that has not yet learned that a particular menu item has been deleted. The peer will helpfully

tell the new node about the menu item, its names, and its prices. The new node will believe that this menu item exists and display it to its user. And so a node that does not retain interest in deletion facts will incorrectly display deleted entities.

You may have observed this behavior in some popular synchronization engines. For example, if you use Microsoft Active Directory, it is possible to observe lingering objects if a domain controller is disconnected beyond the tombstone lifetime.[1] A *tombstone* in Active Directory is a record of an entity's deletion. It is analogous to a `Delete` fact. Active Directory does not preserve tombstones indefinitely. Instead, it defines a lifetime of between 60 and 180 days, depending upon the operating system. If a node is disconnected for longer than that lifetime, or if the clocks have drifted significantly, then its peers may discard the tombstone. That causes the deleted entity to magically reappear.

The only truly fool-proof way of defending a node against deleted entities is for it to continue to be interested in the entity facts and their deletion facts. A node needs to learn about the tombstones no matter how old they are. This seems like a large set of data to retain, but in fact it prevents an even larger set of data from lingering. Sharing this complete list of tombstones prevents a node from becoming interested in all of the property changes and other actions associated with these entities.

Retaining interest in deleted entities has traditionally been one of the more controversial results of immutable architectures. I have had countless conversations with peers who attempt to solve the problem with rules about when a system can forget about entities. The results are always similar to the lingering-object issue that Active Directory faces. If you remain unconvinced, then perhaps the interaction between interest and the Period pattern will provide a more palatable solution. Periods are a way of imposing lifetimes on interest without falling into lingering-object defects.

Interest in Past Periods

The Period pattern describes regularly occurring spans of time during which activities take place. The duration of a period is a time frame that makes sense for the application: a date of business, a calendar month, an academic semester, or a fiscal year. When an activity occurs at a particular node, that node believes the system to be in one single period. It records the activity as a successor of the period fact.

---

[1]Information about Lingering Objects in a Windows Server Active Directory Forest. Microsoft Knowledge Base 910205.

Each node determines its own interest set. Different nodes will express interest in different periods. Nodes closer to the edge of the network tend to operate only within recent periods. A point of sale terminal might only display data within the current date of business. It serves the needs of its users in the short term and leaves long-term history to reporting servers living closer to the center of the network.

To respect the bandwidth and storage limitations of smaller edge devices, we wish to limit their interest set to only the more recent periods. Nodes get to choose the starting points of their projection pipelines. If those pipelines start with the current period, then the node is no longer interested in the past.

The period clock advances for each individual edge node. Close to a time boundary, different nodes might believe the system to be in different periods. But this will not cause any problems sharing facts with more central nodes. Edge node clocks don't need to agree so long as central nodes accept facts outside of what they believe to be the current period.

To apply this solution, a node starts its projections not from a top-level owner, but from a period defined *within* that owner. For example, the front-of-house terminal within a restaurant that the host uses would begin not from the `Restaurant` fact, but from a successor `RestaurantDateOfBusiness`. As its clock progresses, the front-of-house terminal loses interest in the previous date of business for the restaurant. Instead, it starts its projection pipelines from the new date of business. It communicates out to its peers the starting point of the pipeline as the date changes.

This strategy allows edge nodes to limit their storage requirements, while central nodes take on the storage burden. Different nodes get to decide their own interest set. Edge nodes express interest in recent periods by providing a pipeline that starts with a period. Central nodes express interest in deep history by starting from the periods' predecessor.

Sharing Interest

Each node is responsible for defining its own interest set. It does so by defining its own projection pipelines and starting points. For the node to receive the facts that it expresses interest in, it must share those projection pipelines and starting points with its peers.

An application-agnostic immutable architecture will include a network protocol for exchanging pipelines and starting points. Rather than requiring human developers to design a bespoke API for the application at hand, it will adapt its behavior as the application is developed. The behavior of network interaction is generated based on pipelines, starting points, and authorization and distribution rules.

A node expresses interest over an application-agnostic network protocol by providing its projection pipelines and starting points. The peer then executes the pipeline to find facts that the node is interested in. Based on the permanence of the relationship with the node, it may decide to cache those facts and invert the pipeline in order to invalidate the cache. Or it may decide that the node is ephemeral, like a mobile device, and rely upon it to maintain bookmarks as described in Chapter 11.

The starting points for projection pipelines depend upon the purpose that the node serves. Personal devices such as mobile devices tend to start with the User fact of the individual who owns the device. All other projections are reachable via navigation from the results of the first. The node grafts all of those pipelines together in order to produce one single structure originating from the User fact. All groups to which that user gains membership, all entities that that user creates, and all decisions made by or for that user are all reachable from this one all-encompassing projection pipeline.

Nodes more central to the network tend to be shared by many users. These belong to organizations. The organization will be represented by a fact within a multi-tenant model. The projections that these kinds of nodes tend to run will produce organization-wide reports, websites, and publicly facing APIs. Their projection pipelines will start from the organization fact.

Starting points will change slowly as shared nodes are reconfigured for different tenants, new users log in to personal devices, or the period clock advances. Projection pipelines will also change slowly, as new versions of software are installed on each device. An application-agnostic protocol allows each node to exchange information about these slow changes to update their peers about the sets of facts in which they are interested.

Losing Interest

As time passes, nodes will lose interest in facts. When an entity is deleted, nodes will no longer be interested in their properties and relationships with other entities. As the clock advances, nodes lose interest in the day-to-day decisions that occurred within a period. Even though facts are immutable and cannot *conceptually* be deleted, it sometimes makes sense for a node to *practically* delete data.

Nodes at the edge of a network tend to be personal devices. These are relatively small machines with limited storage, bandwidth, and connections. To respect their limits, it is practical to avoid storing and transmitting facts that they have lost interest in.

Through query inversion, a node can recognize when a new fact has removed other facts from its interest set. It can tell, for example, that a `Delete` fact has caused the filter of a pipeline to become false and therefore remove all of the properties of that entity from its interest set. An application-agnostic runtime could remove such facts from storage. Before this strategy is applied, however, a couple of caveats must be observed.

First, before the facts are removed, the edge node must have shared those facts with more permanent peers. If the user of a personal device has modified the property of an entity just as another user has deleted it, then the device might learn about the deletion before it has a chance to upload the property change. If the property change—which the device is no longer interested in—is deleted from storage, then no other node will ever learn of it. When the deleted entity is restored, or a historical report is run, the user's decision will be lost.

The second caveat is that no other node can depend upon this device as a source of facts. Edge devices tend to connect to more central devices to receive new facts. The only facts that they upload are the ones that the user of the device created themselves. But if the personal device is also a server of an even more removed edge device, then its data store cannot be purged. A laptop to which a companion device is attached must retain the history of facts that its satellite might need, even if it itself has lost interest in them.

If query inversion proves to be too complex or cumbersome for an application-agnostic runtime, a simpler strategy exists. The runtime could periodically swap one data store into the background and start filling up a new store. Projections are served out of both data stores, the background one providing stability of past data and the foreground one contributing freshness of new events. Over time, all of the facts that the node is interested in are copied to the foreground store, and all of the outgoing facts are drained from the background queue. A that point of quiescence, the background store can be purged with no loss of information or change of behavior.

Whatever the strategy, it is important to recognize the difference between the practicality of purging facts and the inductive rigor of retaining interest. A node must retain interest in deleted entities and their tombstones in order to correctly defend against lingering objects. As long as a node is interested in a fact, that fact cannot be purged. Only under the strictest of conditions can facts be removed from storage. This can only happen on edge nodes, only after durably sharing that fact with a more central node, and only when interest is truly lost.

Immutable Runtimes

With the introduction of projections and interest, the last of the gaps have been bridged. We can now generate the connections among business logic, user interface, and network communications. Generating behavior is not just about making things easier. With end-to-end behavior generation, we can ensure correctness of applications to a degree that would not otherwise be possible. This gives developers the freedom to explore different solution spaces without being locked into the first one that works. And it gives organizations the autonomy to innovate their business processes even while integrating with other organizations.

There are several ways to generate behavior. Generating behavior might mean generating code. Some code generators are used as templates or scaffolds. Once the code is generated, the developers own it. They are responsible for modifying and maintaining it. Other code generators are run as part of the build process. They produce files that are not intended for the developers to modify. They transpile one language to another so that a domain-specific language can be easily consumed by a general-purpose language.

Generating behavior might also mean altering behavior at runtime to match a specification. Managed runtimes alter their behavior in response to the programs that they execute. Garbage collection, reflection, and serialization are generated behaviors that emerge during runtime. No code generator or transpiler produces a program that performs these operations. The runtime provides these services as they are needed.

Let's imagine an application development solution that uses all of these techniques to generate the behavior of a distributed system. It will make low-level implementation decisions on behalf of the developer to bridge the gaps among storage, business logic, user interface, security, and network communications. It will also bridge the gaps between applications running on different nodes, even those deployed at different times and authored by different organizations. For some aspects of behavior generation, it will rely upon code generation, allowing developers to write directly in the Factual Modeling Language and producing code in their preferred general-purpose language. For other aspects, it will use managed runtimes, permitting behavior to change without compiling or deploying new code. The goal is not rapid application development. It is correctness and autonomy.

Model Generation

Our aspirational application development process begins with a description of the problem domain. We will describe the entities, properties, activities, relationships, and decisions inherent within that domain. All of those concepts are modeled as historical facts. The tool we will use in that description is of course the Factual Modeling Language.

The first thing that the application runtime needs to do is translate fact declarations into data types in a general-purpose language. This is best accomplished with a code generator running continuously during the build process. The output of this code generator is native code so that it can be understood within the integrated development environment (IDE) and type checker. But it is not intended for developers to modify after generation.

The defining characteristic of facts is that they are immutable. A few general-purpose programming languages support immutable data structures, but most default to mutability. The code generator should use appropriate language idioms to discourage modification of the objects representing facts. In languages that have strong protection semantics like Java and C#, this is almost entirely possible. But languages like JavaScript with weak or no protection, it is simply up to the developer not to mutate these generated data structures.

The generated code needs to preserve the structure of the facts. It should be able to compute the canonical hash of a fact from the properties of an object. It should help in determining when a fact already exists and when it is being created for the first time. It needs to understand the predecessor/successor relationships so that developers can express queries. Some of these behaviors can be written into the code, some can be extracted via reflection, and others require that the generated code preserve structural details at runtime.

Query Execution

After the developer has specified just a few layers of the data model, they can begin writing queries. In the application development cycle, queries provide the first chance for a developer to test their hypotheses. They have expressed the structure of the entities and decisions in terms of facts with the belief that these facts will support the requirements of the finished software. If they can write a query that satisfies a requirement, then they have more confidence that the model is correct.

It is vital that a developer have a way to iterate on a model and set of queries quickly, especially during the early phases of application development. A complete immutable runtime should include a playground in which developers can define facts, create instances, and then run queries. Think of this as the equivalent of a SQL design tool like MySQL Workbench, pgAdmin, or Microsoft SQL Server Management Studio (SSMS). It is an exploratory and diagnostic tool for the developer, not for the application.

For an application to execute a query requires an application runtime. The runtime framework translates a query expressed in Factual or in the native language into a pipeline. From there, the pipeline can be executed against any number of data stores, grafted into projection pipelines, or inverted. The pipeline is just a data structure internal to the immutable runtime that can be manipulated for several purposes.

In some incarnations of our aspirational application development system, a code generator turns Factual queries into a pipeline description stored within the code. It generates the function that is the entry point for executing this pipeline, providing the necessary type casts to ensure that the function takes the appropriate starting point and produces the appropriate fact set. In other incarnations, the runtime is able to convert a native language query—such as .NET Linq—into a pipeline. It is careful to validate that the query does not stray outside of the set of capabilities of the deliberately limited Factual query.

Testing

As the application developer iterates over the model and queries, they will want to encode their expectations in repeatable tests. Automated testing has long been a cornerstone of agile application development, even becoming the driving force of application design in test-driven development (TDD). In the application development cycle envisioned here, testing plays a diminished role. Tests do not drive the design or even prove correctness. They simply check our work.

When humans are responsible for designing the ins and outs of software systems, they will inevitably make mistakes. These mistakes lead to defective behavior. We call them "bugs" and pretend that they are inevitable. But as we've seen several times throughout this book, it is possible to prove theorems about the behavior of software. If humans write the data access layer, business logic services, and view models for the user interface, then it is possible for them to introduce bugs. But if an immutable runtime is computing query inverses according to mathematically proven rules, then we can have

confidence that the UI will be updated correctly when state changes. There is little value in testing the generated behavior of the runtime. Tests should focus on validating the intent of the program, not just the implementation.

The developer writes tests in their preferred general-purpose language using the generated code and immutable runtime. The tests execute against an application-agnostic data store that operates completely in memory. This will almost certainly be different from the persistent data store used in production. But the developer is not testing that different data stores correctly execute queries. They are testing that they have written the queries that match their intent.

User Interaction

After the developer has iterated a few times over the model, queries, and tests, they reach a point where they have confidence that the system described meets the intended requirements. At this point, they can begin mapping that description to the user interface. The UI has evolved in parallel, as a series of wireframes. Now, it becomes augmented with Factual queries and mathematical formulas. When we have both a description of the intended UI and a declaration of a model and queries, we can finally start bringing the two together.

The developer declares a projection in their general-purpose language. Projections incorporate queries, which the code generator has converted into functions. The projection mixes these functions with the aggregation and calculation tools that the language itself provides. In Linq, for example, the generated function might return an `IQueryable`. This represents the query itself, not its results. The language features already built into Linq can compose `IQueryable` streams, map them through lambdas and other native features, and construct a complex object perfectly suited to support the user interface.

Based on the target UI platform, the immutable runtime provides additional support. In an Extensible Application Markup Language (XAML) application, the runtime implements the `INotifyPropertyChanged` event in order to translate from query inversion to data binding. In a React application, the runtime stores the projected data in component state and provides hooks for efficient access. No matter what front-end technology is chosen, the developer works at the level of declarative projection based on queries over immutable facts.

Finally, the developer responds to input events. When a user clicks a button or modifies a field, the developer translates this action into a new fact. The runtime takes all of the necessary actions to bridge the gaps between the user interface and other parts of the system. It stores the fact using the application-agnostic data store. It queues the fact to be shared with more central nodes. It determines interest to push it out to peers. And it runs query inverses to determine which other parts of the user interface have been affected. The human developer is no longer on the hook for any of these technical decisions.

User interaction might not take the form of a UI that appears on a web page or mobile device. It might be an API consumed by non-immutable systems. These techniques still apply, just with some technical modifications. URLs map to starting points. Projections map query results onto data structures. Those data structures are then translated into JSON, XML, or some other transport format. And operations such as POST and PUT map to creation of new facts that represent creation and updates of entities. The biggest difference is that APIs tend to rely less upon real-time updates than user interfaces do. And so inverses play a lesser role, that of invalidating caches rather than pushing updates.

Developers may choose to implement another layer of tests at this stage. They have tested the queries and the model that went into the projections. Now they have an opportunity to test the projections themselves. They can express the intent that when a user presses a certain button, data should appear in a certain list. That intent can be tested against the projection—the view model or component state—rather than against the user interface itself. This gives developers a tool for validating interaction scenarios without taking a dependency upon more malleable user interface components. And these tests can still be run using an in-memory data store with no loss of confidence.

Collaboration

The final phase of the aspirational application development cycle is to define the collaboration between users, between systems, and between nodes. The behaviors taken by one actor will have an impact on the behaviors observed by another actor. We express these points of collaboration in an immutable application using interest.

The application developer registers all of their projection pipelines with the runtime. This represents the intent that the application expresses interest in every fact necessary to update its interface. Developers might take this opportunity to optimize projections to

produce better interest calculations or provide additional projections that support other features of the application. But the default position of registering the set of projections used in the UI should produce reasonable results.

Next, application developers trigger interest from certain starting points. For example, when the user logs in, they will inform the runtime that the set of projections starting from that user have become active. This will trigger the runtime to start pulling related facts so that the projections will be updated as they arrive. If an application includes navigation from one view to the next, the developer has some choices to make. They can graft subsequent screens' projections onto the end of the primary screen's projection, thus ensuring that the facts are already available before the user navigates. Or, they can wait until that navigation occurs, minimizing the bandwidth consumed until the user decides what to view. Either of these choices is easily expressed with a few function calls to the immutable runtime.

The runtime for its part shares these projection pipelines with its peers. As the application requests that pipelines be run for a given starting point, the runtime sends out that serialized starting point and the composed interest pipelines. Peers execute the pipelines to determine the facts in the interest set. They also run the transitive closure to find all predecessors of those facts. Finally, they use bookmarks to negotiate which facts the peer already knows about and which need to be shared. As they share facts, the peers update their bookmarks. All of this is accomplished with no additional guidance from the developer.

This is the most natural time for developers to introduce authorization and distribution rules. Immutable runtimes execute authorization rules upon receipt of a new fact to determine for themselves whether to trust and store it. The runtime executes distribution rules as it sends facts to filter out data that a peer does not have permission to see, even if it has expressed interest. These rules could have been expressed earlier in the cycle, but collaboration gives developers the perfect opportunity to validate them.

The few decisions that the developer *did* make—how to combine projections, when to navigate to new starting points, and authorization and distribution rules—deserve a new layer of tests. The developer writes a set of tests that show that when one user takes an action on one device, another user sees a certain effect on another. They demonstrate that protected data is not visible to a peer and that unauthorized activities will be rejected. These tests are written against projections, as those are the expressions that map most closely to the user's actual experience. But they incorporate the behavior of the code that registers interest pipelines, starting points, and security rules.

During the test, the runtime uses not only an in-memory data store but also an in-process communication simulator. It does not need to set up interconnected nodes or use unreliable infrastructure during the test. This simulator will not be used in the production application, but it enables testing of the important decisions that the developer makes during this phase. The correct behavior of the real communication system is governed by the mathematics of conflict-free replicated data types (CRDTs) and covered by the runtime vendor with its own suite of tests.

Developers iterate over these phases, building onto the model, writing new queries, constructing new projections, and enabling more complex collaborations. Through the entire process, the decisions that they make for the current phase are consistent with the ones that they made in the past. The tools enforce that consistency through code generation and type checking. Developers build up a working application over several iterations. At no point can the developer express one intent to the data access layer and a different intent to the networking subsystem. They cannot introduce bugs based on disagreement between islands of behavior. All behavior is generated—whether by transpiled code or interpretive runtime—from the expression of intent. This gives developers the confidence to explore new business solutions, and the autonomy to try new ideas.

Immutable Organizations

As immutable architectures gain acceptance, they will begin as experiments within individual organizations. Application developers will try out the concepts, patterns, and techniques described in this book. They will adopt frameworks that provide immutable runtimes, such as the Correspondence and Jinaga open source projects that the author maintains. As they find success, immutability will find its way into other business units within the organization.

The spread will be gradual. As business units need to integrate with one another, they will be faced with the choice. Do we adopt an immutable collaboration between these services, or do we use a more traditional Simple Object Access Protocol (SOAP), Representational State Transfer (REST), or Enterprise Service Bus (ESB) style interaction? Business units who are slow to adopt immutability will find that they need to implement additional adapter layers and will not get the benefits of eventual consistency and conflict resolution. But those that buy into a new style of application development will soon discover its advantages in rapid and reliable exploration and experimentation.

Decision Substrate

Business units—not departments—will be the driving force of immutability adoption. A business unit includes all functions necessary for the generation of value within one step of the value stream. In order to operate effectively, the unit needs to innovate quickly to respond to changing demands. If they can capitalize on shifts in environmental forces, they can demonstrate value to the whole organization. If they cannot, then their functions may be outsourced. With these pressures at play, business units have found themselves in the past moving to cloud platforms while the organization's IT department stayed behind with on-premises networks. These same pressures will drive them to adopt more exploratory and autonomous architectures.

Eventually, the backbone of the organization will be a shared distributed immutable data exchange. Business units will independently develop solutions on top of this substrate. Some will be valuable, and others will die away. But those that stick will become the source of reliable business data for other experiments. The immutable data structure that emerges will be the decision infrastructure upon which the organization builds its value stream.

Certainly in the short term, and very likely in the long term as well, immutable applications will run side by side with static applications. These will require integration points. Using the Outbox pattern, webhooks, and emulated mutability, developers of immutable applications will find ways to bridge those gaps. Over time, they will publish those integrations as reusable components for other immutable application developers. The integration points will become small immutable applications living on their own within the architecture, proxies for external services.

Organizations that evolve an immutable decision substrate will find themselves at an advantage over their static counterparts. They will have a reliable log of all business decisions that went into generating value within the company. They will exchange information securely and freely across business unit boundaries. And they will have the freedom to innovate on new solutions without the overhead of large enterprise hegemonies.

Globally Distributed Systems

As powerful as immutable architecture can be within a single application or a single organization, the benefits multiply as immutable organizations integrate with one another. As they discover each other, they will pierce the veil of emulated mutation

and expose the immutable facts beneath. They will provide access to their data synchronization protocols so that the two organizations can intermingle their immutable models.

As immutability spreads organization to organization, security boundaries will remain firmly in place. The authorization and distribution rules will continue to control access to sensitive facts. Asymmetric keys will continue to identify individual actors, and their digital signatures and shared keys will continue to protect data at rest and through untrusted nodes. Security concerns will be enforced at the logical boundaries between regions, not just at physical network boundaries and with proprietary APIs.

Organizations will limit coupling between their models by keeping integration points high up in the causal chain. They will agree upon predecessors, and each create their own successor facts. They will evolve these models using structural versioning so that different partners can take their integrations in their own directions without central coordination.

As pairs of immutable organizations join into clusters, they will form networks of interconnected models shared among all participants. They will build upon common application-agnostic network and security protocols, and each maintain their own data stores. From these clusters will emerge a shared global distributed system, exchanging immutable records in a securely integrated history.

In years past, this kind of emergent global network would have seemed ridiculous. But we have already seen two successful examples. The Internet is not a centralized managed set of servers. It is a system of open standards, implemented by a myriad of vendors and operated by a loose consortium of cooperating organizations. And blockchains are not central databases that rely upon physical and network security to control access. They are shared public ledgers of immutable records executing on diverse nodes running competing implementations of an open standard.

Based on these examples, I have confidence that the global immutable network will emerge. It will not have the mutable mindset of REST APIs that bind them to location specificity. Nor will it have the inefficiencies of proof of work locking it into a linear history. Instead, it will apply the findings of decades of mathematical research to produce truly globally distributed systems that are location independent, convergent, secure, and autonomous.

Index

A

Access control model, 213
Acknowledged message, 16
Action fact, 273
Acyclic graphs, 52
Advanced Message Queueing Protocol
(AMQP), 362, 363
Affected set
computing, 295
inverting longer queries, 296, 297
query inverse, 294
unsatisfiable inverses, 297, 298
user interface component, 294
walking backward, 298–300
Aggregate function, 316
Aggregation, 57, 181
Algorithm—Consistent Hashing, 282
Apache ActiveMQ, 363
Apache Kafka, 362, 363
Application-agnostic immutable
architecture, 393
Application-agnostic network
protocol, 394
Application-agnostic stores
code generation, 351
constraints, 348
databases, 349
generic fact table, 349, 350
predecessor relationships, 350, 351
versioning, 352–354

Application-defined invariants, 112
Application protocols, 358
Assignment fact, 294
Asymmetric encryption
size limit, 230
symmetric key, 231
Asymmetric keys, 404
Asynchronous protocols, 364
Asynchronous message processing, 364
Asynchronous Model View Update, 29
immutable app architecture, 44, 45
unidirectional data flow, 43, 44
update loop, 41, 42
Async over HTTP, 372, 373
Atomic, consistent, isolated, and
durable (ACID), 112
Attend and Rate sessions, 166
Authorization
initial, 220, 221
principal fact, 218
query, 218, 219
rule, blog post, 220, 248
authorize keyword, 87
Auto-incremented IDs
environment dependence, 98, 99
parent–child insertion, 99, 100
remote creation, 100, 101
Availability, 116
Awaiting Review, 190–192
Azure Service Bus, 363

D

T