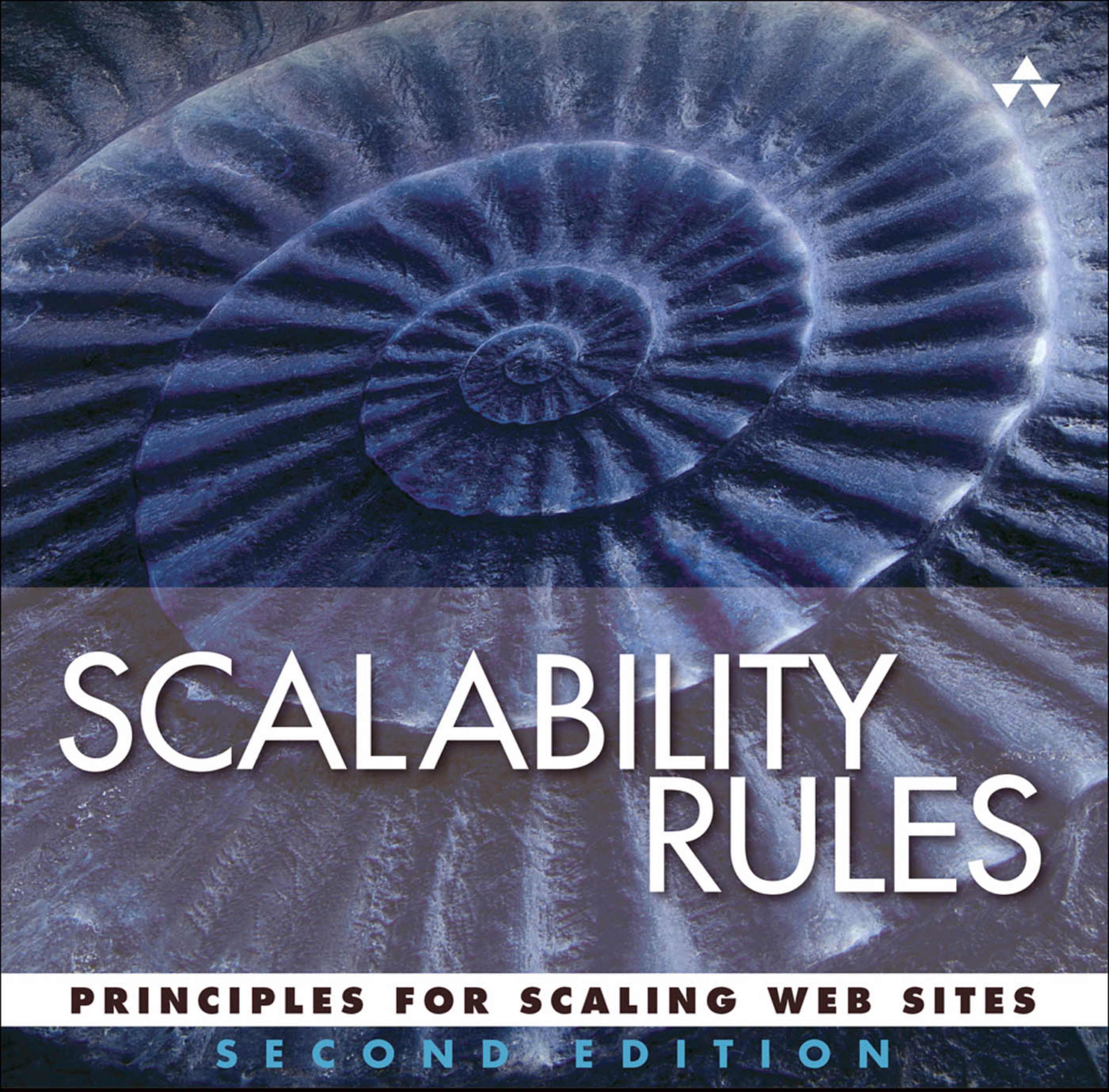


SCALABILITY RULES

PRINCIPLES FOR SCALING WEB SITES

SECOND EDITION

MARTIN L. ABBOTT MICHAEL T. FISHER

"Whether you're taking on a role as a technology leader in a new company or you simply want to make great technology decisions, Scalability Rules *will be the go-to resource on your bookshelf."*
—**Chad Dickerson,** CTO, Etsy

Praise for the First Edition of *Scalability Rules*

"Once again, Abbott and Fisher provide a book that I'll be giving to our engineers. It's an essential read for anyone dealing with scaling an online business."

—**Chris Lalonde**, GM of Data Stores, Rackspace

"Abbott and Fisher again tackle the difficult problem of scalability in their unique and practical manner. Distilling the challenges of operating a fast-growing presence on the Internet into 50 easy-to-understand rules, the authors provide a modern cookbook of scalability recipes that guide the reader through the difficulties of fast growth."

—**Geoffrey Weber**, VP, Internet Operations, Shutterfly

"Abbott and Fisher have distilled years of wisdom into a set of cogent principles to avoid many nonobvious mistakes."

—**Jonathan Heiliger**, VP, Technical Operations, Facebook

"In *The Art of Scalability*, the AKF team taught us that scale is not just a technology challenge. Scale is obtained only through a combination of people, process, *and* technology. With *Scalability Rules*, Martin Abbott and Michael Fisher fill our scalability toolbox with easily implemented and time-tested rules that once applied will enable massive scale."

—**Jerome Labat**, VP, Product Development IT, Intuit

"When I joined Etsy, I partnered with Mike and Marty to hit the ground running in my new role, and it was one of the best investments of time I have made in my career. The indispensable advice from my experience working with Mike and Marty is fully captured here in this book. Whether you're taking on a role as a technology leader in a new company or you simply want to make great technology decisions, *Scalability Rules* will be the go-to resource on your bookshelf."

—**Chad Dickerson**, CTO, Etsy

"*Scalability Rules* provides an essential set of practical tools and concepts anyone can use when designing, upgrading, or inheriting a technology platform. It's very easy to focus on an immediate problem and overlook issues that will appear in the future. This book ensures strategic design principles are applied to everyday challenges."

—**Robert Guild**, Director and Senior Architect, Financial Services

"An insightful, practical guide to designing and building scalable systems. A must-read for both product building and operations teams, this book offers concise and crisp insights gained from years of practical experience of AKF principals. With the complexity of modern systems, scalability considerations should be an integral part of the architecture and implementation process. Scaling systems for hypergrowth requires an agile, iterative approach that is closely aligned with product features; this book shows you how."

—**Nanda Kishore**, CTO, ShareThis

"For organizations looking to scale technology, people, and processes rapidly or effectively, the twin pairing of *Scalability Rules* and *The Art of Scalability* is unbeatable. The rules-driven approach in *Scalability Rules* not only makes this an easy reference companion, but also allows organizations to tailor the Abbott and Fisher approach to their specific needs both immediately and in the future!"

—**Jeremy Wright**, CEO, BNOTIONS.ca, and Founder, b5media

Scalability Rules

Second Edition

This page intentionally left blank

Scalability Rules

Principles for Scaling Web Sites

Second Edition

Martin L. Abbott
Michael T. Fisher

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2016944687

ISBN-13: 978-0-13-443160-4
ISBN-10: 0-13-443160-X
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
1 16

Editor-in-Chief
Mark L. Taub

Executive Editor
Laura Lewin

Development Editor
Songlin Qiu

Managing Editor
Sandra Schroeder

Full-Service Production Manager
Julie B. Nahil

Project Editor
Dana Wilson

Copy Editor
Barbara Wood

Indexer
Jack Lewis

Proofreader
Barbara Lasoff

Technical Reviewers
Camille Fournier
Chris Lalonde
Mark Uhrmacher

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
The CIP Group

This book is dedicated to our friend and partner "Big" Tom Keeven. "Big" refers to the impact he's had in helping countless companies scale in his nearly 30 years in the business.

This page intentionally left blank

Contents

Preface

Thanks for your interest in the second edition of *Scalability Rules*! This book is meant to serve as a primer, a refresher, and a lightweight reference manual to help engineers, architects, and managers develop and maintain scalable Internet products. It is laid out in a series of rules, each of them bundled thematically by different topics. Most of the rules are technically focused, and a smaller number of them address some critical mindset or process concern, each of which is absolutely critical to building scalable products. The rules vary in their depth and focus. Some rules are high level, such as defining a model that can be applied to nearly any scalability problem; others are specific and may explain a technique, such as how to modify headers to maximize the "cacheability" of content. In this edition we've added stories from CTOs and entrepreneurs of successful Internet product companies from startups to Fortune 500 companies. These stories help to illustrate how the rules were developed and why they are so important within high-transaction environments. No story serves to better illustrate the challenges and demands of hyper-scale on the Internet than Amazon. Rick Dalzell, Amazon's first CTO, illustrates several of the rules within this book in his story, which follows.

Taming the Wild West of the Internet

From the perspective of innovation and industry disruption, few companies have had the success of Amazon. Since its founding in 1994, Amazon has contributed to redefining at least three industries: consumer commerce, print publishing, and server hosting. And Amazon's contributions go well beyond just disrupting industries; they've consistently been a thought leader in service-oriented architectures, development team construction, and a myriad of other engineering approaches. Amazon's size and scale along all dimensions of its business are simply mind-boggling; the company has consistently grown at a rate unimaginable for traditional brick-and-mortar businesses. Since 1998, Amazon grew from $600 million (no small business at all) in annual revenue to an astounding $107 billion (that's "billion" with a *B*) in 2015.[1] Walmart, the world's largest retailer, had annual sales of $485.7 billion in 2015.[2] But Walmart has been around since 1962, and it took 35 years to top $100 billion in sales compared to Amazon's 21 years. No book professing to codify the rules of scalability from the mouths of CTOs who have created them would be complete without one or more stories from Amazon.

Jeff Bezos incorporated Amazon (originally Cadabra) in July of 1994 and launched Amazon.com as an online bookseller in 1995. In 1997, Bezos hired Rick Dalzell, who was then the VP of information technology at Walmart. Rick spent the next ten years at Amazon leading Amazon's development efforts. Let's join Rick as he relays the story of his Amazon career:

"When I was at Walmart, we had one of the world's largest relational databases running the company's operations. But it became clear to the Amazon team pretty quickly that the big, monolithic database approach was simply not going to work for Amazon. Even back then, we handled more transactions in a week on the Amazon system than the Walmart system had to handle in a month. And when you add to that our incredible growth, well, it was pretty clear that monoliths simply were not going to work. Jeff [Bezos] took me to lunch one day, and I told him we needed to split the monolith into services. He said, 'That's great—but we need to build a moat around this business and get to 14 million customers.' I explained that without starting to work on these splits, we wouldn't be able to make it through Christmas."

Rick continued, "Now keep in mind that this is the mid- to late nineties. There weren't a lot of companies working on distributed transaction systems. There weren't a lot of places to go to find help in figuring out how to scale transaction processing systems growing in excess of 300% year on year. There weren't any rulebooks, and there weren't any experts who had 'been there and done that.' It was a new frontier—a new Wild, Wild West. But it was clear to us that we had to distribute this thing to be successful. Contrary to what made me successful at Walmart, if we were going to scale our solution and our organization, we were going to need to split the solution and the underlying database up into a number of services." (The reader should note that an entire chapter of this book, Chapter 2, "Distribute Your Work," is dedicated to such splits.)

"We started by splitting the commerce and store engine from the back-end fulfillment systems that Amazon uses. This was really the start of our journey into the services-oriented architecture that folks have heard about at Amazon. All sorts of things came out of this, including Amazon's work on team independence and the API contracts. Ultimately, the work created a new industry [infrastructure as a service] and a new business for Amazon in Amazon Web Services—but that's another story for another time. The work wasn't easy; some components of the once-monolithic database such as customer data—what we called 'the Amazon customer database or ACB'—took several years to figure out how to segment. We started with services that were high in transaction volumes and could be quickly split in both software and data, like the front- and back-end systems that I described. Each split we made would further distribute the system and allow additional scale. Finally, we got back to solving the hairy problem of ACB and split it out around 2004.

"The team was incredibly smart, but we also had a bit of luck from time to time. It's not that we never failed, but when we would make a mistake we would quickly correct it and figure out how to fix the associated problems. The lucky piece is that none of our failures were as large and well publicized as those of some of the other companies struggling through the same learning curve. A number of key learnings in building these distributed services came out of these splits, learnings such as the need to limit session and state, stay away from distributed two-phase commit transactions, communicating asynchronously whenever possible, and so on. In fact, without a strong bias toward asynchronous communication through a publish-and-subscribe message bus, I don't know if we could have ever split and scaled the way we did. We also learned to allow things to be eventually consistent where possible, in just about everything except payments. Real-time consistency is costly, and wherever people wouldn't really know the difference, we'd just let things get 'fuzzy' for a while and let them sync up later. And of course there were a number of 'human' or team learnings as well such as the need to keep teams small[3] and to have specific contracts between teams that use the services of other teams."

Rick's story of how he led Amazon's development efforts in scaling for a decade is incredibly useful. From his insights we can garner a number of lessons that can be applied to many companies' scaling challenges. We've used Rick's story along with those of several other notable CTOs and entrepreneurs of successful Internet product companies ranging from startups to Fortune 500 companies to illustrate how important the rules in this book are to scaling high-transaction environments.

Quick Start Guide

Experienced engineers, architects, and managers can read through the header sections of all the rules that contain the what, when, how, and why. You can browse through each chapter to read these, or you can jump to Chapter 13, "Rule Review and Prioritization," which has a consolidated view of the headers. Once you've read these, go back to the chapters that are new to you or that you find more interesting.

For less experienced readers we understand that 50 rules can seem overwhelming. We do believe that you should eventually become familiar with all the rules, but we also understand that you need to prioritize your time. With that in mind, we have picked out five chapters for managers, five chapters for software developers, and five chapters for technical operations that we recommend you read before the others to get a jump start on your scalability knowledge.

Managers:

- Chapter 1, "Reduce the Equation"
- Chapter 2, "Distribute Your Work"
- Chapter 4, "Use the Right Tools"
- Chapter 7, "Learn from Your Mistakes"
- Chapter 12, "Miscellaneous Rules"

Software developers:

- Chapter 1, "Reduce the Equation"
- Chapter 2, "Distribute Your Work"
- Chapter 5, "Get Out of Your Own Way"
- Chapter 10, "Avoid or Distribute State"
- Chapter 11, "Asynchronous Communication and Message Buses"

Technical operations:

- Chapter 2, "Distribute Your Work"
- Chapter 3, "Design to Scale Out Horizontally"
- Chapter 6, "Use Caching Aggressively"
- Chapter 8, "Database Rules"
- Chapter 9, "Design for Fault Tolerance and Graceful Failure"

As you have time later, we recommend reading all the rules to familiarize yourself with the rules and concepts that we present no matter what your role. The book is short and can probably be read in a coast-to-coast flight in the United States.

After the first read, the book can be used as a reference. If you are looking to fix or re-architect an existing product, Chapter 13 offers an approach to applying the rules to your existing platform based on cost and the expected benefit (presented as a reduction of risk). If you already have your own prioritization mechanism, we do not recommend changing it for ours unless you like our approach better. If you don't have an existing method of prioritization, our method should help you think through which rules you should apply first.

If you are just starting to develop a new product, the rules can help inform and guide you as to best practices for scaling. In this case, the approach of prioritization represented in Chapter 13 can best be used as a guide to what's most important to consider in your design. You should look at the rules that are most likely to allow you to scale for your immediate and long-term needs and implement those.

For all organizations, the rules can serve to help you create a set of architectural principles to drive future development. Select the 5, 10, or 15 rules that will help your product scale best and use them as an augmentation of your existing design reviews. Engineers and architects can ask questions relevant to each of the scalability rules that you select and ensure that any new significant design meets your scalability standards. While these rules are as specific and fixed as possible, there is room for modification based on your system's particular criteria. If you or your team has extensive scalability experience, go ahead and tweak these rules as necessary to fit your particular scenario. If you and your team lack large-scale experience, use the rules exactly as is and see how far they allow you to scale.

Finally, this book is meant to serve as a reference and handbook. Chapter 13 is set up as a quick reference and summary of the rules. Whether you are experiencing problems or simply looking to develop a more scalable solution, Chapter 13 can be a quick reference guide to help pinpoint the rules that will help you out of your predicament fastest or help you define the best path forward in the event of new development. Besides using this as a desktop reference, also consider integrating this into your organization by one of many tactics such as taking one or two rules each week and discussing them at your technology all-hands meeting.

Why a Second Edition?

The first edition of *Scalability Rules* was the first book to address the topic of scalability in a rules-oriented fashion. Customers loved its brevity, ease of use, and convenience. But time and time again readers and clients of our firm, AKF Partners, asked us to tell the stories behind the rules. Because we pride ourselves in putting the needs of our clients first, we edited this book to include stories upon which the rules are based.

In addition to telling the stories of multiple CTOs and successful entrepreneurs, editing the book for a second edition allowed us to update the content to remain consistent with the best practices in our industry. The second edition also gave us the opportunity to subject our material to another round of technical peer reviews and production editing. All of this results in a second edition that's easier to read, easier to understand, and easier to apply.

How Does *Scalability Rules* Differ from *The Art of Scalability*?

The Art of Scalability, Second Edition (ISBN 0134032802, published by Addison-Wesley), our first book on the topic of scalability, focused on people, process, and technology, whereas *Scalability Rules* is predominantly a technically focused book. Don't get us wrong; we still believe that people and process are the most important components of building scalable solutions. After all, it's the organization, including both the individual contributors and the management, that succeeds or fails in producing scalable solutions. The technology isn't at fault for failing to scale—it's the people who are at fault for building it, selecting it, or integrating it. But we believe that *The Art of Scalability* adequately addresses the people and process concerns around scalability, and we wanted to go into greater depth on the technical aspects of scalability.

Scalability Rules expands on the third (technical) section of *The Art of Scalability*. The material in *Scalability Rules* is either new or discussed in a more technical fashion than in *The Art of Scalability*. As some reviewers on Amazon point out, *Scalability Rules* works well as both a standalone book and as a companion to *The Art of Scalability*.

Notes

1. "Net Sales Revenue of Amazon from 2004 to 2015," www.statista.com/statistics/266282/annual-net-revenue-of-amazoncom/.
2. Walmart, Corporate and Financial Facts, http://corporate.walmart.com/_news_/news-archive/investors/2015/02/19/walmart-announces-q4-underlying-eps-of-161-and-additional-strategic-investments-in-people-e-commerce-walmart-us-comp-sales-increased-15-percent.
3. Authors' note: Famously known as Amazon's Two-Pizza Rule—no team can be larger than that which two pizzas can feed.

Register your copy of *Scalability Rules, Second Edition*, at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780134431604) and click Submit. Once the process is complete, you will find any available bonus content under "Registered Products."

Acknowledgments

The rules contained within this book weren't developed by our partnership alone. They are the result of nearly 70 years of work with clients, colleagues, and partners within nearly 400 companies, divisions, and organizations. Each of them contributed, in varying degrees, to some or all of the rules within this book. As such, we would like to acknowledge the contributions of our friends, partners, clients, coworkers, and bosses for whom or with which we've worked over the past several (combined) decades. The CTO stories from Rick Dalzell, Chris Lalonde, James Barrese, Lon Binder, Brad Peterson, Grant Klopper, Jeremy King, Tom Keeven, Tayloe Stansbury, Chris Schremser, and Chuck Geiger included herein are invaluable in helping to illustrate the need for these 50 rules. We thank each of you for your time, thoughtfulness, and consideration in telling us your stories.

We would also like to acknowledge and thank the editors who have provided guidance, feedback, and project management. The technical editors from both the first and second editions—Geoffrey Weber, Chris Lalonde, Camille Fournier, Jeremy Wright, Mark Urmacher, and Robert Guild—shared with us their combined decades of technology experience and provided invaluable insight. Our editors from Addison-Wesley, Songlin Qiu, Laura Lewin, Olivia Basegio, and Trina MacDonald, provided supportive stylistic and rhetorical guidance throughout every step of this project. Thank you all for helping with this project.

Last but certainly not least, we'd like to thank our families and friends who put up with our absence from social events as we sat in front of a computer screen and wrote. No undertaking of this magnitude is done single-handedly, and without our families' and friends' understanding and support this would have been a much more arduous journey.

This page intentionally left blank

About the Authors

Martin L. Abbott is a founding partner of AKF Partners, a growth consulting firm focusing on meeting the needs of today's fast-paced and hyper-growth companies. Marty was formerly the COO of Quigo, an advertising technology startup acquired by AOL in 2007. Prior to Quigo, Marty spent nearly six years at eBay, most recently as SVP of technology and CTO and member of the CEO's executive staff. Prior to eBay, Marty held domestic and international engineering, management, and executive positions at Gateway and Motorola. Marty has served on a number of boards of directors for public and private companies. He spent a number of years as both an active-duty and reserve officer in the US Army. Marty has a BS in computer science from the United States Military Academy, an MS in computer engineering from the University of Florida, is a graduate of the Harvard Business School Executive Education Program, and has a Doctorate of Management from Case Western Reserve University.

Michael T. Fisher is a founding partner of AKF Partners, a growth consulting firm focusing on meeting the needs of today's fast-paced and hyper-growth companies. Prior to cofounding AKF Partners, Michael held many industry roles including CTO of Quigo, acquired by AOL in 2007, and VP of engineering and architecture for PayPal. He served as a pilot in the US Army. Michael received a PhD and MBA from Case Western Reserve University's Weatherhead School of Management, an MS in information systems from Hawaii Pacific University, and a BS in computer science from the United States Military Academy (West Point). Michael is an adjunct professor in the Design and Innovation Department at Case Western Reserve University's Weatherhead School of Management.

This page intentionally left blank

1

Reduce the Equation

By nearly any measure, Jeremy King has had a successful and exciting career. In the mid-1990s, before the heyday of the Internet, Jeremy was involved in Bay Networks' award-winning SAP implementation. From there, Jeremy joined the dot-com boom as the VP of engineering at Petopia.com. Jeremy often jokes that he received his "real-world MBA" from the "University of Hard Knocks" during the dot-com bubble at Petopia. From Petopia, Jeremy joined eBay as the director of architecture for V3, eBay's then-next-generation commerce platform. If Petopia offered a lesson in economics and financing, eBay (where Jeremy later became a VP) offered an unprecedented education in scaling systems. Jeremy spent three years as the EVP of technology at LiveOps and is now the CTO at WalmartLabs.

eBay taught Jeremy many lessons, including the need for simplicity in architecture. For context, when Jeremy joined in 2001, eBay stood with rarefied companies like Amazon and Google at the extreme edge of online transaction processing (OLTP) and scale. For the full year of 2001, eBay recorded $2.735 billion[1] in gross merchandise sales as compared to Walmart's worldwide sales of $191.3 billion (online sales were not reported)[2] and Amazon's total sales of $3.12 billion.[3] Underneath this heady success, however, lay a dark past for eBay.

In June of 1999 eBay experienced a near death sentence because of an outage lasting almost 24 hours.[4] The eBay site continued to undergo outages of varying durations for months after the outage of June 1999, and although each was caused by different trigger events, all of them could be traced back to the inability of the site to scale to nearly unprecedented user growth. These outages changed the culture of the company forever. More specifically, they focused the company on attempting to set the standard for high availability and reliability of its service offering.

In 1999, eBay sold most of its merchandise using an auction format. Auctions are unique entities because, as compared to typical online commerce transactions, auction items tend to be short-lived and have an inordinately high volume of bids (write transactions) and views (read transactions) near the end of their expected duration. Most items for sale in traditional platforms have a relatively flat number of transactions through their life with the typical seasonal peaks, whereas eBay had millions of items for sale, and all of the user activity would be directed at a fraction of those items at any given time. This represented a unique problem as database load, for instance, would be primarily on a small number of items, and the database (then a monolith) that supported eBay would struggle with physical and logical contention on these items. This in turn could manifest itself as site slowness and even complete outages.

Jeremy's first eBay job was to lead a team to redefine eBay's software architecture with the goal of keeping incidents like the June 1999 outage from happening again. This was a task made even more complex by the combination of eBay's meteoric growth and the difficulty of the auction format. The project, internally named V3, was a reimplementation of the eBay commerce engine in Java (the prior implementation was C++) and a re-architecture of the site to allow for multiple "sharded" databases along the X, Y, and Z axes described in Chapter 2, "Distribute Your Work."

The team approached every component with a maniacal focus, attempting to ensure that everything could be nearly infinitely scaled and that any failure could be resolved quickly with minimal impact. "My primary lesson learned," Jeremy indicated, "was that we treated everything as if it had the same complexity and business criticality as the actual bidding process on an auction. Absolutely everything from the display of images to eBay's reputation [often called Feedback] system on the site was treated similarly with similar fault tolerance.

"It's important to remember," continued Jeremy, "that this was 2001 and that there were very few companies experiencing our growth at our size online. As such, we really couldn't go anywhere—whether to a vendor or an open-source solution—to solve our problems. We had to invent everything ourselves, something I'd really prefer not to do."

At first glance, it is difficult to tease out the lesson in Jeremy's comments. So what if everything was built in the same bulletproof fashion as an auction? "Well," said Jeremy with a laugh, "not everything is as complex as an auction. Let's take the reputation engine, for instance. It doesn't have the same competition over short periods of time as auctions. As such, it really doesn't need to have the same principles applied to it. The system simply scales more cost-effectively and potentially is as highly available, if you take an approach that recognizes you don't have the same level of transactional competition on subsets of data over short periods of time. More importantly, Feedback has one write transaction per item sold, whereas an auction may have hundreds of attempted write transactions in a second on a single item. This isn't the same as decrementing inventory; it's a complex comparison of bid price to all other bids and then the calculation of a 'second price.' But we treated that and everything else as if we were solving the same problem as auctions—specifically, to be able to scale under incredible demand, for small subsets of data, much of it happening in the last few seconds of the life of an auction."

That being clear, we wondered what the impact of making some pieces of V3 more complex than others might be. "That's easy," said Jeremy. "While V3 overall was a success, I think in hindsight we could have potentially completed it faster or cheaper or both. Moreover, because some aspects of it were overly complex or alternatively not as simple as the problem demanded, the maintenance costs of those aspects are higher. Contrast this approach with an architecture principle I've since learned and applied at both Walmart and LiveOps: match the effort and approach to the complexity of the problem. Not every solution has the same complexity—take the simplest approach to achieve the desired outcome. While having a standard platform or language can seem desirable from a scaling, maintainability, or reuse aspect, taking the simple approach on a new open-source project, language, or platform can vastly change the cost, time to market, or innovation you can deliver for your customers."

Jeremy's story is about not making things harder than they need to be, or putting it another way, keeping things as simple as possible. Our view is that a complex problem is really just a collection of smaller, simpler problems waiting to be solved. This chapter is about making big problems small, and in so doing achieving big results with less work.

As is the case with many of the chapters in *Scalability Rules*, the rules here vary in size and complexity. Some are overarching rules easily applied to several aspects of our design. Some rules are very granular and prescriptive in their implementation to specific systems.

Rule 1—Don't Overengineer the Solution

Rule 1: What, When, How, and Why

What: Guard against complex solutions during design.

When to use: Can be used for any project and should be used for all large or complex systems or projects.

How to use: Resist the urge to overengineer solutions by testing ease of understanding with fellow engineers.

Why: Complex solutions are excessively costly to implement and are expensive to maintain long term.

Key takeaways: Systems that are overly complex limit your ability to scale. Simple systems are more easily and cost-effectively maintained and scaled.

As is explained in its Wikipedia entry, overengineering falls into two broad categories.[5] The first category covers products designed and implemented to exceed their useful requirements. We discuss this problem briefly for completeness, but in our estimation its impact on scale is small compared to that of the second problem. The second category of overengineering covers products that are made to be overly complex. As we earlier implied, we are most concerned about the impact of this second category on scalability. But first, let's address the notion of exceeding requirements.

To explain the first category of overengineering, exceeding useful requirements, we must first make sense of the term *useful*, which here means simply "capable of being used." For example, designing an HVAC unit for a family house that is capable of heating that house to 300 degrees Fahrenheit in outside temperatures of 0 Kelvin simply has no use for us anywhere. The effort necessary to design and manufacture such a solution is wasted as compared to a solution that might heat the house to a comfortable living temperature in environments where outside temperatures might get close to –20 degrees Fahrenheit. This type of overengineering might have cost overrun elements, including a higher cost to develop (engineer) the solution and a higher cost to implement the solution in hardware and software. It may further impact the company by delaying the product launch if the overengineered system took longer to develop than the useful system. Each of these costs has stakeholder impact as higher costs result in lower margins, and longer development times result in delayed revenue or benefits. *Scope creep*, or the addition of scope between initial product definition and initial product launch, is one manifestation of overengineering.

An example closer to our domain of experience might be developing an employee time card system capable of handling a number of employees for a single company that equals or exceeds 100 times the population of Planet Earth. The probability that the Earth's population will increase 100-fold within the useful life of the software is tiny. The possibility that all of those people would work for a single company is even smaller. We certainly want to build our systems to scale to customer demands, but we don't want to waste time implementing and deploying those capabilities too far ahead of our need (see Rule 2).

The second category of overengineering deals with making something overly complex and making something in a complex way. Put more simply, the second category consists of either making something work harder to get a job done than is necessary, making a user work harder to get a job done than is necessary, or making an engineer work harder to understand something than is necessary. Let's dive into each of these three areas of overly complex systems.

What does it mean to make something work harder than is necessary? Jeremy King's example of building all the features constituting eBay's site to the demanding requirements of the auction bidding process is a perfect example of making something (e.g., the Feedback system) work harder than is necessary. Some other examples come from the real world. Imagine that you ask your significant other to go to the grocery store. When he agrees, you tell him to pick up one of everything at the store, and then to pause and call you when he gets to the checkout line. Once he calls, you will tell him the handful of items that you would like from the many baskets of items he has collected, and he can throw everything else on the floor. "Don't be ridiculous!" you might say. But have you ever performed a `select (*) from schema_name.table_name` SQL statement within your code only to cherry-pick your results from the returned set (see Rule 35 in Chapter 8, "Database Rules")? Our grocery store example is essentially the same activity as the `select (*)` case. How many lines of conditionals have you added to your code to handle edge cases and in what order are they evaluated? Do you handle the most likely case first? How often do you ask your database to return a result set you just returned, and how often do you re-create an HTML page you just displayed? This particular problem (doing work repetitively when you can just go back and get your last correct answer) is so rampant and easily overlooked that we've dedicated an entire chapter (Chapter 6, "Use Caching Aggressively") to this topic! You get the point.

What do we mean by making a user work harder than is necessary? In many cases, less is more. Many times in the pursuit of trying to make a system flexible, we strive to cram as many odd features as possible into it. Variety is not always the spice of life.

Many times users just want to get from point A to point B as quickly as possible without distractions. If 99% of your market doesn't care about being able to save their blog as a .pdf file, don't build in a prompt asking them if they'd like to save it as a .pdf. If your users are interested in converting .wav files to MP3 files, they are already sold on a loss of fidelity, so don't distract them with the ability to convert to lossless compression FLAC files.

Finally, we come to the far-too-common problem of making software too complex for other engineers to easily and quickly understand. Back in the day it was all the rage, and in fact there were competitions, to create complex code that would be difficult for others to understand. Medals were handed out to the person who could develop code that would bring senior developers to tears of acquiescence within code reviews. Complexity became the intellectual cage within which geeky code slingers would battle for organizational dominance. For those interested in continuing in the geek fest, but in a "safe room" away from the potential stakeholder value destruction of doing it "for real," we suggest you partake in the International Obfuscated C Code Contest at www0.us.ioccc.org/index.html. For everyone else, recognize that your job is to develop simple, easy-to-understand solutions that are easy to maintain and create shareholder value.

We should all strive to write code that everyone can understand. The real measure of a great engineer is how quickly that engineer can simplify a complex problem (see Rule 3) and develop an easily understood and maintainable solution. Easy-to-follow solutions allow less-senior engineers to more quickly come up to speed to support systems. Easy-to-understand solutions mean that problems can be found earlier during troubleshooting, and systems can be restored to their proper working order faster. Easy-to-follow solutions increase the scalability of your organization and your solution.

A great test to determine whether something is too complex is to have the engineer in charge of solving a given complex problem present his or her solution to several engineering cohorts within the company. The cohorts should represent different engineering experience levels as well as varying tenures within the company (we make a distinction here because you might have experienced engineers with very little company experience). To pass this test, each of the engineering cohorts should easily understand the solution, and each cohort should be able to describe the solution, unassisted, to others not otherwise knowledgeable about it. If any cohort does not understand the solution, the team should debate whether the system is overly complex.

Overengineering is one of the many enemies of scale. Developing a solution beyond that which is useful simply wastes money and time. It may further waste processing resources, increase the cost of scale, and limit the overall scalability of the system (how far that system can be scaled). Building solutions that are overly complex has a similar effect. Systems that work too hard increase your cost and limit your ultimate size. Systems that make users work too hard limit how quickly you are likely to increase the number of users and therefore how quickly you will grow your business. Systems that are too complex to understand kill organizational productivity and the ease with which you can add engineers or add functionality to your system.

Rule 2—Design Scale into the Solution (D-I-D Process)

> ### Rule 2: What, When, How, and Why
>
> **What:** An approach to provide JIT (just-in-time) scalability.
>
> **When to use:** On all projects; this approach is the most cost-effective (resources and time) to ensure scalability.
>
> **How to use:**
>
> - Design for 20x capacity.
> - Implement for 3x capacity.
> - Deploy for roughly 1.5x capacity.
>
> **Why:** D-I-D provides a cost-effective, JIT method of scaling your product.
>
> **Key takeaways:** Teams can save a lot of money and time by thinking of how to scale solutions early, implementing (coding) them a month or so before they are needed, and deploying them days before the customer rush or demand.

Our firm is focused on helping clients address their scalability needs. As you might imagine, customers often ask us, "When should we invest in scalability?" The somewhat flippant answer is that you should invest (and deploy) the day before the solution is needed. If you could deploy scalability improvements the day before you needed them, your investments would be "just in time" and this approach would help maximize firm profits and shareholder wealth. This is similar to what Dell brought to the world with configure-to-order systems combined with just-in-time manufacturing.

But let's face it—timing such an investment and deployment "just in time" is simply impossible, and even if possible it would incur a great deal of risk if you did not nail the date exactly. The next best thing to investing and deploying "the day before" is AKF Partners' *Design-Implement-Deploy* or *D-I-D* approach to thinking about scalability. These phases match the cognitive phases with which we are all familiar: starting to think about and designing a solution to a problem, building or coding a solution to that problem, and actually installing or deploying the solution to the problem. This approach does not argue for nor does it need a waterfall model. We argue that agile methodologies abide by such a process by the very definition of the need for human involvement. You cannot develop a solution to a problem of which you are not aware, and a solution cannot be manufactured or released if it is not developed. Regardless of the development methodology (agile, waterfall, hybrid, or whatever), everything we develop should be based on a set of architectural principles and standards that define and guide what we do.

Design

We start with the notion that discussing and designing something are both significantly less expensive than actually implementing that design in code. Given this relatively

Table 1.1 **D-I-D Process for Scale**

	Design	Implement	Deploy
Scale objective	20x to infinite	3x to 20x	1.5x to 3x
Intellectual cost	High	Medium	Low to medium
Engineering cost	Low	High	Medium
Asset cost	Low	Low to medium	High to very high
Total cost	Low to medium	Medium	Medium

low cost, we can discuss and sketch out a design for how to scale our platform well in advance of our need. For example, we clearly would not want to deploy 10x, 20x, or 100x more capacity than we would need in our production environment. However, the cost of discussing and deciding how to scale something to those dimensions is comparatively small. The focus then in the (D)esign phase of the D-I-D scale model is on scaling to between 20x and infinity. Our intellectual costs are high as we employ our "big thinkers" to think through the "big problems." Engineering and asset costs, however, are low as we aren't writing code or deploying costly systems. Scalability summits, a process in which groups of leaders and engineers gather to discuss scale-limiting aspects of a product, are a good way to identify the areas necessary to scale within the design phase of the D-I-D process. Table 1.1 lists the phases of the D-I-D process.

Implement

As time moves on, and as our perceived need for future scale draws near, we move to (I)mplement our designs within our software. We reduce our scope in terms of scale needs to something that's more realistic, such as 3x to 20x our current size. We use "size" here to identify that element of the system that is perceived to be the greatest bottleneck of scale and therefore in the greatest need of modification to achieve our business results. There may be cases where the cost of scaling 100x (or greater) our current size is not different from the cost of scaling 20x. If this is the case, we might as well make those changes once rather than going in and making changes multiple times. This might be the case if we are going to perform a modulus of our user base to distribute (or share) users across multiple (N) systems and databases. We might code a variable Cust_MOD that we can configure over time between 1 (today) and 1,000 (five years from now). The engineering (or implementation) cost of such a change really doesn't vary with the size of N, so we might as well make Cust_MOD capable of being as large as possible. The cost of these types of changes is high in terms of engineering time, medium in terms of intellectual time (we already discussed the designs earlier in our lifecycle), and low in terms of assets as we don't need to deploy 100x our systems today if we intend to deploy a modulus of 1 or 2 in our first phase.

Deploy

The final phase of the D-I-D process is (D)eploy. Using our modulus example, we want to deploy our systems in a just-in-time fashion; there's no reason to have idle assets diluting shareholder value. Maybe we put 1.5x our peak capacity in production if we are a moderately high-growth company and 5x our peak capacity in production if we are a hyper-growth company. We often guide our clients to leverage the cloud for burst capacity so that we don't have 33% of our assets waiting around for a sudden increase in user activity. Asset costs are high in the deployment phase, and other costs range from low to medium. Total costs tend to be highest for this category because to deploy 100x necessary capacity relative to demand would kill many companies. Remember that scale is an elastic concept; it can both expand and contract, and our solutions should recognize both aspects. Therefore, flexibility is key, because you may need to move capacity around as different systems within your solution expand and contract in response to customer demand.

Designing and thinking about scale come relatively cheaply and thus should happen frequently. Ideally these activities result in some sort of written documentation so that others can build upon it quickly should the need arise. Engineering (or developing) the architected or designed solutions can happen later and cost a bit more overall, but there is no need to actually implement them in production. We can roll the code and make small modifications as in our modulus example without needing to purchase 100x the number of systems we have today. Finally, the process lends itself nicely to purchasing equipment just ahead of our need, which might be a six-week lead time from a major equipment provider or having one of our systems administrators run down to the local server store in extreme emergencies. Obviously, in the case of infrastructure as a service (IaaS, aka cloud) environments, we do not need to purchase capacity in advance of need and can easily "spin up" compute assets for the deploy phase on a near-as-needed and near-real-time basis.

Rule 3—Simplify the Solution Three Times Over

Rule 3: What, When, How, and Why

What: Used when designing complex systems, this rule simplifies the scope, design, and implementation.

When to use: When designing complex systems or products where resources (engineering or computational) are limited.

How to use:

- Simplify scope using the Pareto Principle.
- Simplify design by thinking about cost effectiveness and scalability.
- Simplify implementation by leveraging the experience of others.

Why: Focusing just on "not being complex" doesn't address the issues created in requirements or story and epoch development or the actual implementation.

Key takeaways: Simplification needs to happen during every aspect of product development.

Whereas Rule 1 dealt with avoiding surpassing the "usable" requirements and eliminating complexity, this rule addresses taking another pass at simplifying everything from your perception of your needs through your actual design and implementation. Rule 1 is about fighting against the urge to make something overly complex, and Rule 3 is about attempting to further simplify the solution by the methods described herein. Sometimes we tell our clients to think of this rule as "asking the three hows": How do I simplify my scope, my design, and my implementation?

How Do I Simplify My Scope?

The answer to this question of simplification is to apply the Pareto Principle (also known as the 80-20 rule) frequently. What 80% of your benefit is achieved from 20% of the work? In our case, a direct application is to ask, "What 80% of your revenue will be achieved by 20% of your features?" Doing significantly less (20% of the work) while achieving significant benefits (80% of the value) frees up your team to perform other tasks. If you cut unnecessary features from your product, you can do five times as much work, and your product will be significantly less complex! With four-fifths fewer features, your system will no doubt have fewer dependencies between functions and as a result will be able to scale both more efficiently and more cost-effectively. Moreover, the 80% of your time that is freed up can be used to launch new product offerings as well as invest in thinking ahead to the future scalability needs of your product.

We're not alone in our thinking on how to reduce unnecessary features while keeping a majority of the benefit. The folks at 37signals, now rebranded as Basecamp, are huge proponents of this approach, discussing the need and opportunity to prune work in both their book *Rework*[6] and in their blog post titled "You Can Always Do Less."[7] Indeed, the concept of the "minimum viable product" popularized by Eric Reis and evangelized by Marty Cagan is predicated on the notion of maximizing the "amount of validated learning about customers with the least effort."[8] This "agile" focused approach allows us to release simple, easily scalable products quickly. In so doing we get greater product throughput in our organizations (organizational scalability) and can spend additional time focusing on building the minimal product in a more scalable fashion. By simplifying our scope, we have more computational power because we are doing less. If you don't believe us, go back and read Jeremy King's story and his lessons learned. Had the eBay team reduced the scope of features like Feedback, the V3 project would have been delivered sooner, at lower cost, and for relatively the same value to the end consumer.

How Do I Simplify My Design?

With this new, smaller scope, the job of simplifying our implementation just became easier. Simplifying design is closely related to the complexity aspect of overengineering. Complexity elimination is about cutting off unnecessary trips in a job, and simplification is about finding a shorter path. In Rule 1, we gave the example of asking a database only for that which you need; `select(*) from schema_name.table_name` became `select (column) from schema_name.table_name`. The approach of design simplification suggests that we first look to see if we already have the information being requested within a local shared resource like local memory. Complexity elimination is about doing less work, and design simplification is about doing that work faster and easier.

Imagine a case where we are looking to read some source data, perform a computation on intermediate tokens from this source data, and then bundle up the tokens and computation into an object. In many cases, each of these verbs might be broken into a series of services. In fact, this approach looks similar to that employed by the popular MapReduce algorithm. This approach isn't overly complex, so it doesn't violate Rule 1. But if we know that files to be read are small and we don't need to combine tokens across files, it might make sense to take the simple path of making this a simple monolithic application rather than decomposing it into services. Going back to our time card example, if the goal is simply to compute hours for a single individual, it makes sense to have multiple cloned monolithic applications reading a queue of time cards and performing the computations. Put simply, the step of design simplification asks us how to get the job done in an easy-to-understand, cost-effective, and scalable way.

How Do I Simplify My Implementation?

Finally, we get to the question of implementation. Consistent with Rule 2, the D-I-D process for scale, we define an implementation as the actual coding of a solution. This is where we get into questions such as whether it makes more sense to solve a problem with recursion or iteration. Should we define an array of a certain size, or be prepared to allocate memory dynamically as we need it? Do we make the solution, acquire open source for the solution, or buy it? The answers to all these questions have a consistent theme: "How can we leverage the experiences of others and existing solutions to simplify our implementation?"

Given that we can't be the best at building everything, we should first look to find widely adopted open-source or third-party solutions to meet our needs. If those don't exist, we should look to see if someone within our own organization has developed a scalable solution to solve the problem. In the absence of a proprietary solution, we should again look externally to see if someone has described a scalable approach to solve the problem that we can legally copy or mimic. Only in the absence of finding one of these three things should we embark on attempting to create the solution ourselves. The simplest implementation is almost always one that has already been implemented and proven scalable.

Rule 4—Reduce DNS Lookups

> **Rule 4: What, When, How, and Why**
>
> **What:** Reduce the number of DNS lookups from a user perspective.
>
> **When to use:** On all Web pages where performance matters.
>
> **How to use:** Minimize the number of DNS lookups required to download pages, but balance this with the browser's limitation for simultaneous connections.

> **Why:** DNS lookups take a great deal of time, and large numbers of them can amount to a large portion of your user experience.
>
> **Key takeaways:** Reduction of objects, tasks, computation, and so on is a great way of speeding up page load time, but division of labor must be considered as well.

Many of the rules in this book are focused on back-end architecture for a software as a service (SaaS) solution, but for this rule let's consider your customer's browser. If you use any of the browser-level debugging tools such as Mozilla Firefox's plug-in Firebug,[9] or Chrome's standard developer tools, you'll see some interesting results when you load a page from your service. One of the things you will most likely notice is that similarly sized objects on your page take different amounts of time to download. As you look closer, you'll see that some of these objects have an additional step at the beginning of their download. This additional step is the DNS lookup.

The Domain Name System (DNS) is one of the most important parts of the infrastructure of the Internet or any other network that utilizes the Internet Protocol Suite (TCP/IP). It allows the translation from domain name (www.akfpartners.com) to an IP address (184.72.236.173) and is often analogized to a phone book. DNS is maintained by a distributed database system, the nodes of which are called name servers. The top of the hierarchy consists of the root name servers. Each domain has at least one authoritative DNS server that publishes information about that domain.

This process of translating domains into IP addresses is made quicker by caching on many levels, including the browser, computer operating system, Internet service provider, and so on. While caching significantly improves the performance of name resolution, today's pages can have hundreds or even thousands of objects served from multiple domains. Each domain requires a name resolution, and these resolution requests can add up to time that's noticeable to the consumer.

Before we go any deeper into our discussion of reducing the DNS lookups, we need to understand at a high level how most browsers download pages. This isn't meant to be an in-depth study of browsers, but understanding the basics will help you optimize your application's performance and scalability. Browsers take advantage of the fact that almost all Web pages are composed of many different objects (images, JavaScript files, CSS files, and so on) by having the ability to download multiple objects through simultaneous connections. Browsers limit the maximum number of simultaneous persistent connections per server or proxy. According to the HTTP/1.1 RFC,[10] this maximum should be set to two. However, many browsers now ignore this RFC and have maximums of six or more. We'll talk about how to optimize your page download time based on this functionality in the next rule. For now let's focus on our Web page broken up into many objects and able to be downloaded through multiple connections.

Every distinct domain that serves one or more objects for a Web page requires a DNS lookup. This lookup may be resolved in an intermediate cache or require a full round trip to a DNS name server. For example, let's assume we have a simple Web page that has four objects: (1) the HTML page itself that contains text and directives for other objects, (2) a CSS file for the layout, (3) a JavaScript file for a menu item, and (4) a JPG image. The HTML comes from our domain (akfpartners.com), but the

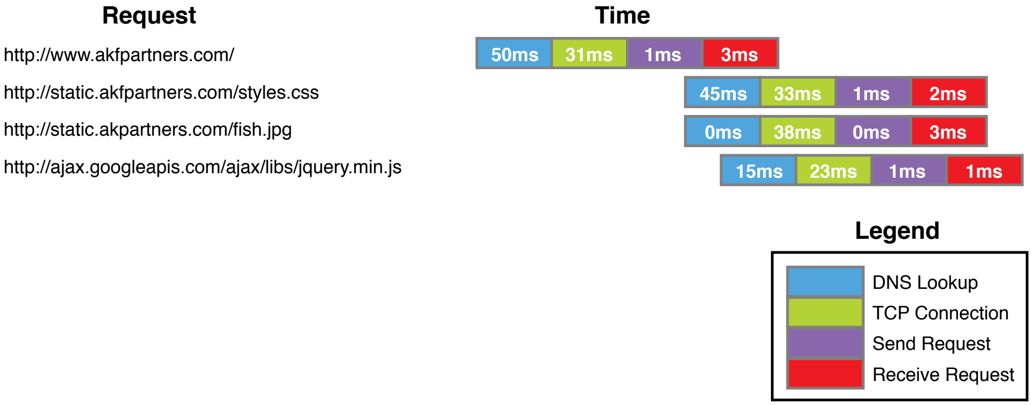


Figure 1.1 Object download time

CSS and JPG are served from a subdomain (static.akfpartners.com), and the JavaScript we've linked to from Google (ajax.googleapis.com). In this scenario our browser first receives the request to go to page www.akfpartners.com, which requires a DNS lookup of the akfpartners.com domain. Once the HTML is downloaded, the browser parses it and finds that it needs to download both the CSS and JPG from static.akfpartners.com, which requires another DNS lookup. Finally, the parsing reveals the need for an external JavaScript file from yet another domain. Depending on the freshness of the DNS cache in our browser, operating system, and so on, this lookup can take essentially no time up to hundreds of milliseconds. Figure 1.1 shows a graphical representation of this.

As a general rule, the fewer DNS lookups on your pages, the better your page download performance will be. There is a downside to combining all your objects into a single domain, and we've hinted at the reason in the previous discussion about maximum simultaneous connects. We explore this topic in more detail in the next rule.

Rule 5—Reduce Objects Where Possible

Rule 5: What, When, How, and Why

What: Reduce the number of objects on a page where possible.

When to use: On all Web pages where performance matters.

How to use:

- Reduce or combine objects but balance this with maximizing simultaneous connections.
- Look for opportunities to reduce weight of objects as well.
- Test changes to ensure performance improvements.

Why: The number of objects impacts page download times.

Key takeaways: The balance between objects and methods that serve them is a science that requires constant measurement and adjustment; it's a balance among customer usability, usefulness, and performance.

As we indicated in the discussion of Rule 4, Web pages consist of many different objects (HTML, CSS, images, JavaScript, and so on). Browsers download these objects somewhat independently and often in parallel. One of the easiest ways to improve Web page performance and thus increase your scalability (fewer objects to serve per page means your servers can serve more pages) is to reduce the number of objects on a page. The biggest offenders on most pages are graphical objects such as images. As an example, let's take a look at Google's search page (www.google.com), which by its own admission is minimalist in nature.[11] At the time of writing, Google had just a handful of objects on its page, including a small number of .pngs, some scripts, and a style sheet. In our very unscientific experiment the search page loaded in about 300 milliseconds. Compare this to a client that we were working with in the online magazine industry, whose home page had more than 300 objects, 200 of which were images and took on average more than 12 seconds to load. What this client didn't realize was that slow page performance was causing them to lose valuable readers. Google published a white paper in 2009 claiming that tests showed that an increase in search latency of 400 milliseconds reduced their daily searches by almost 0.6%.[12] Since then, many of our clients have indicated varying positive user performance increases related to faster page response times.

Reducing the number of objects on the page is a great way to improve performance and scalability, but before you rush off to remove all your images there are a few other things to consider. First is obviously the important information that you are trying to convey to your customers. With no images your page will look like the 1992 W3 Project page, which claimed to be the first Web page.[13] Since you need images and JavaScript and CSS files, your second consideration might be to combine all similar objects into a single file. This is not a bad idea, and in fact there are techniques such as CSS image sprites for this exact purpose. An image sprite is a combination of small images into one larger image that can be manipulated with CSS to display any single individual image. The benefit of this is that the number of images requested is significantly reduced. Back to our discussion of the Google search page, one of the two images on the search page is a sprite that consists of about two dozen smaller images that can be individually displayed or not.[14]

So far we've covered the concept that reducing the number of objects on a page will improve performance and scalability, but this must be balanced with the need for modern-looking pages requiring images, CSS, and JavaScript. Next we covered how these can be combined into a single object to reduce the number of distinct requests that the browser must make to render the page. Yet another factor is that combining everything into a single object doesn't make use of the maximum number of simultaneous persistent connections per server that we discussed previously in Rule 4. As a recap, this is the browser's capability to download multiple objects simultaneously from a single domain. If everything is in one object, having the capability to download two or more objects simultaneously doesn't help. Now we need to think about breaking these objects up into a number of smaller ones that can be downloaded simultaneously. One final variable to add to the equation is that part we mentioned about simultaneous persistent connections per server, which will bring us full circle to our DNS discussion noted in Rule 4.

The simultaneous connection feature of a browser is a limit ascribed to each domain that is serving the objects. If all objects on your page come from a single domain (www.akfpartners. com), then whatever the browser has set as the maximum number of connections is the most objects that can be downloaded simultaneously. As mentioned previously, this maximum is suggested to be set at two, but many browsers by default have increased this to six or more. Therefore, you want your content (images, CSS, JavaScript, and so on) divided into enough objects to take advantage of this feature in most browsers. One technique to really take advantage of this browser feature is to serve different objects from different subdomains (for example, static1.akfpartners.com, static2.akfpartners.com, and so on). The browser considers each of these different domains and allows for each to have the maximum connects concurrently. The client that we talked about earlier who was in the online magazine industry and had a 12-second page load time used this technique across seven subdomains and was able to reduce the average load time to less than 5 seconds.

No discussion regarding page speed, as addressed in this rule and Rule 4, would be complete without a nod toward overall page weight and the weight (or size in terms of bytes) of the objects that constitute that page. Leaner (smaller) is almost always faster. That said, society has been conditioned, with the availability of ever-increasing bandwidth, to expect rich and "heavy" pages. It is always wise to make a page as light as possible to achieve the desired result. And where pages must be heavy, implement Gzip compression to reduce the page weight transferred and ideally the total response time of the page.

Unfortunately there is not an absolute answer about the ideal size (number or weight) of objects or how many subdomains you should consider. The key to improving performance and scalability is testing your pages. There is a balance among necessary content and functionality, object size, rendering time, total download time, domains, and so on. If you have 100 images on a page, each 50KB, combining them into a single sprite is probably not a great idea because the page will not be able to display any images until the entire 4.9MB object downloads. The same concept goes for JavaScript. If you combine all your .js files into one, your page cannot use any of the JavaScript functions until the entire file is downloaded. The only way to ensure that you have the best possible page speed is to test different approaches and find the one that is best for you.

In summary, the fewer objects there are on a page, the better for performance, but this must be balanced with many other factors. Included in these factors are the amount of content that must be displayed, how many objects can be combined, maximizing the use of simultaneous connections by adding domains, the total page weight and whether penalization can help, and so on. While this rule touches on many Web site performance improvement techniques, the real focus is how to improve performance and thus increase the scalability of your site through the reduction of objects on the page. Many other techniques for optimizing performance should be considered, including loading CSS at the top of the page and JavaScript files at the bottom, minifying files, and making use of caches, lazy loading, and so on.

Rule 6—Use Homogeneous Networks

Rule 6: What, When, How, and Why

What: Ensure that switches and routers come from a single provider.

When to use: When designing or expanding your network.

How to use:

- Do not mix networking gear from different OEMs for switches and routers.
- Buy or open-source for other networking gear (firewalls, load balancers, and so on).

Why: Intermittent interoperability and availability issues simply aren't worth the potential cost savings.

Key takeaways: Heterogeneous networking gear tends to cause availability and scalability problems. Choose a single provider.

As a firm, we are technology agnostic, meaning that we believe almost any technology can be made to scale when architected and deployed correctly. This agnosticism ranges from programming language preference to database vendors to hardware. The one caveat to this is with network gear such as routers and switches. Almost all the vendors claim that they implement standard protocols (for example, Internet Control Message Protocol RFC 792,[15] Routing Information Protocol RFC 1058,[16] Border Gateway Protocol RFC 4271[17]) that allow for devices from different vendors to communicate, but many also implement proprietary protocols such as Cisco's Enhanced Interior Gateway Routing Protocol (EIGRP). What we've found in our own practice, as well as with many of our customers, is that each vendor's interpretation of how to implement a standard is often different. As an analogy, if you've ever developed the user interface for a Web page and tested it in different browsers such as Internet Explorer, Firefox, and Chrome, you've seen firsthand how different implementations of standards can be. Now, imagine that going on inside your network. Mixing vendor A's network devices with vendor B's network devices is asking for trouble.

This is not to say that we prefer one vendor over another—we don't. As long as the vendors are a "reference-able" standard utilized by customers larger than you, in terms of network traffic volume, we don't have a preference. This rule does not apply to networking gear such as hubs, load balancers, and firewalls. The network devices that we care about in terms of homogeneity are the ones that must communicate to each other to route traffic. For all the other network devices that may or may not be included in your network such as intrusion detection systems (IDSs), firewalls, load balancers, and distributed denial of service (DDOS) protection appliances, we recommend best-of-breed choices. For these devices choose the vendor that best serves your needs in terms of features, reliability, cost, and service.

Summary

This chapter was about making things simpler. Guarding against complexity (aka overengineering—Rule 1) and simplifying every step of your product from your initial requirements or stories through the final implementation (Rule 3) give us products that are easy to understand from an engineering perspective and therefore easy to scale. By thinking about scale early (Rule 2), even if we don't implement it, we can have solutions ready on demand for our business. Rules 4 and 5 teach us to reduce the work we force browsers to do by reducing the number of objects and DNS lookups we must make to download those objects. Rule 6 teaches us to keep our networks simple and homogeneous to decrease the chances of scale and availability problems associated with mixed networking gear.

Notes

1. "eBay Announces Fourth Quarter and Year End 2001 Financial Results," http://investor.ebay.com/common/mobile/iphone/releasedetail.cfm?releaseid=69550&CompanyID=ebay&mobileid=.
2. Walmart Annual Report 2001, http://c46b2bcc0db5865f5a76-91c2ff8eba65983a1c33d367b8503d02.r78.cf2.rackcdn.com/de/18/2cd2cde44b8c8ec84304db7f38ea/2001-annual-report-for-walmart-stores-inc_130202938087042153.pdf.
3. "Amazon.com Announces 4th Quarter Profit 2002," http://media.corporate-ir.net/media_files/irol/97/97664/reports/q401.pdf.
4. "Important Letter from Meg and Pierre," June 11, 1999, http://pages.ebay.com/outage-letter.html.
5. Wikipedia, "Overengineering," http://en.wikipedia.org/wiki/Overengineering.
6. Jason Fried and David Heinemeier Hansson, *Rework* (New York: Crown Business, 2010).
7. 37signals, "You Can Always Do Less," Signal vs. Noise blog, January 14, 2010, http://37signals.com/svn/posts/2106-you-can-always-do-less.
8. Wikipedia, "Minimum Viable Product," http://en.wikipedia.org/wiki/Minimum_viable_product.
9. To get or install Firebug, go to http://getfirebug.com/.
10. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, Network Working Group Request for Comments 2616, "Hypertext Transfer Protocol—HTTP/1.1," June 1999, www.ietf.org/rfc/rfc2616.txt.
11. The Official Google Blog, "A Spring Metamorphosis—Google's New Look," May 5, 2010, http://googleblog.blogspot.com/2010/05/spring-metamorphosis-googles-new-look.html.

12. Jake Brutlag, "Speed Matters for Google Web Search," Google, Inc., June 2009, http://services.google.com/fh/files/blogs/google_delayexp.pdf.
13. World Wide Web, www.w3.org/History/19921103-hypertext/hypertext/WWW/TheProject.html.
14. Google.com, www.google.com/images/srpr/nav_logo14.png.
15. J. Postel, Network Working Group Request for Comments 792, "Internet Control Message Protocol," September 1981, http://tools.ietf.org/html/rfc792.
16. C. Hedrick, Network Working Group Request for Comments 1058, "Routing Information Protocol," June 1988, http://tools.ietf.org/html/rfc1058.
17. Y. Rekhter, T. Li, and S. Hares, eds., Network Working Group Request for Comments 4271, "A Border Gateway Protocol 4 (BGP-4)," January 2006, http://tools.ietf.org/html/rfc4271.

This page intentionally left blank

2

Distribute Your Work

In 2004 the founding team of ServiceNow (originally called Glidesoft), built a generic workflow platform they called "Glide." In looking for an industry in which they could apply the Glide platform, the team felt that the Information Technology Service Management (ITSM) space, founded on the Information Technology Infrastructure Library (ITIL), was primed for a platform as a service (PaaS) player. While there existed competition or potentially substitutes in this space in the form of on-premise software solutions such as Remedy, the team felt that the success of companies like Salesforce for customer relationship management (CRM) solutions was a good indication of potential adoption for online ITSM solutions.

In 2006 the company changed its name to ServiceNow in order to better represent its approach to the needs of buyers in the ITSM solution space. By 2007 the company was profitable. Unlike many startups, ServiceNow appreciated the value of designing, implementing, and deploying for scale early in its life. The initial solutions design included the notions of both fault isolation (covered in Chapter 9, "Design for Fault Tolerance and Graceful Failure") and Z axis customer splits (covered in this chapter). This fault isolation and customer segmentation allowed the company to both scale to profitability early on and to avoid the noisy-neighbor effect common to so many early SaaS and PaaS offerings. Furthermore, the company valued the cost effectiveness afforded by multitenancy, so while they created fault isolation along customer boundaries, they still designed their solution to leverage multitenancy within a database management system (DBMS) for smaller customers not requiring complete isolation. Finally, the company also valued the insight offered by outside perspectives and the value inherent to experienced employees.

ServiceNow contracted with AKF Partners over a number of engagements to help them think through their future architectural needs and ultimately hired one of the founding partners of AKF, Tom Keeven, to augment their already-talented engineering staff. "We were born with incredible scalability from the date of launch," indicated Tom. "Segmentation along customer boundaries using the AKF Z axis of scale went a long way to ensuring that we could scale into our early demand. But as our customer base grew and the average size of our customer increased beyond small early adopters to much larger Fortune 500 companies, the characterization of our workload changed and the average number of seats per customer dramatically increased. All of these led to each customer performing more transactions and storing more data. Furthermore, we were extending our scope of functionality, adding significantly greater value to our customer base with each release. This functionality extension meant even greater demand was being placed on the systems for customers both large and small. Finally, we had a small problem with running multiple schemas or databases under a single DBMS within MySQL. Specifically, the catalog functionality within MySQL [sometimes technically referred to as the information_schema] was starting to show contention when we had 30 high-volume tenants on each DBMS instance."

Tom Keeven's unique experience building Web-based products from the high-flying days of Gateway Computer, to the Wild West startup days of the Internet at companies like eBay and PayPal, along with his experience across a number of clients at AKF, made him uniquely suited to helping to solve ServiceNow's challenges. Tom explained, "The database catalog problem was simple to solve. For very large customers we simply had to dedicate a DBMS per customer, thereby reducing the burst radius of the fault isolation zone. Medium-size customers may have tenants below 30, and small customers could continue to have a high degree of multitenancy [for more on this see Chapter 9]. The AKF Scale Cube was helpful in offsetting both the increasing size of our customers and the increased demands of rapid functionality extensions and value creation. For large customers with heavy transaction processing demands we incorporated the X axis by replicating data to read-only databases. With this configuration, reports, which are typically computationally and I/O intensive but read-only, could be run without impact to the scale of the lighter-weight transaction (OLTP) requests. While the report functionality also represented a Y axis (service/function or resource-based) split, we added further Y axis splits by service to enable additional fault isolation by service, significantly greater caching of data, and faster developer throughput. All of these splits, the X, Y, and Z axes, allowed us to have consistency within the infrastructure and purchase similar commodity systems for any type of customer. Need more horsepower? The X axis allows us to increase transaction volumes easily and quickly. If data is starting to become unwieldy on databases, our architecture allows us to reduce the degree of multitenancy (Z axis) or split discrete services off (Y axis) onto similarly sized hardware."

This chapter discusses scaling databases and services through cloning and replication, separating functionality or services, and splitting similar data sets across storage and application systems. Using these three approaches, you will be able to scale nearly any system or database to a level that approaches infinite scalability. We use the word *approaches* here as a bit of a hedge, but in our experience across hundreds of companies and thousands of systems these techniques have yet to fail. To help visualize these three approaches to scale we employ the AKF Scale Cube, a diagram we developed to represent these methods of scaling systems. Figure 2.1 shows the AKF Scale Cube, which is named after our partnership, AKF Partners.

At the heart of the AKF Scale Cube are three simple axes, each with an associated rule for scalability. The cube is a great way to represent the path from minimal scale (lower left front of the cube) to near-infinite scalability (upper right back corner of the cube). Sometimes, it's easier to see these three axes without the confined space of the cube. Figure 2.2 shows the axes along with their associated rules. We cover each of the three rules in this chapter.

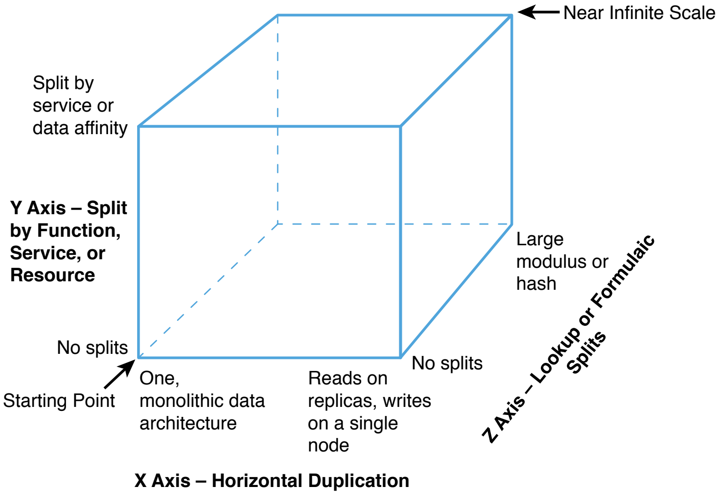

Near Infinite Scale
Split by service or data affinity
Y Axis – Split by Function, Service, or Resource
No splits
Starting Point
One, monolithic data architecture
Reads on replicas, writes on a single node
No splits
Large modulus or hash
Z Axis – Lookup or Formulaic Splits
X Axis – Horizontal Duplication

Figure 2.1 AKF Scale Cube

Not every company will need all of the capabilities (all three axes) inherent to the AKF Scale Cube. For many of our clients, one of the types of splits (X, Y, or Z) meets their needs for a decade or more. But when you have the type of viral success achieved by the likes of ServiceNow, it is likely that you will need two or more of the splits identified within this chapter.

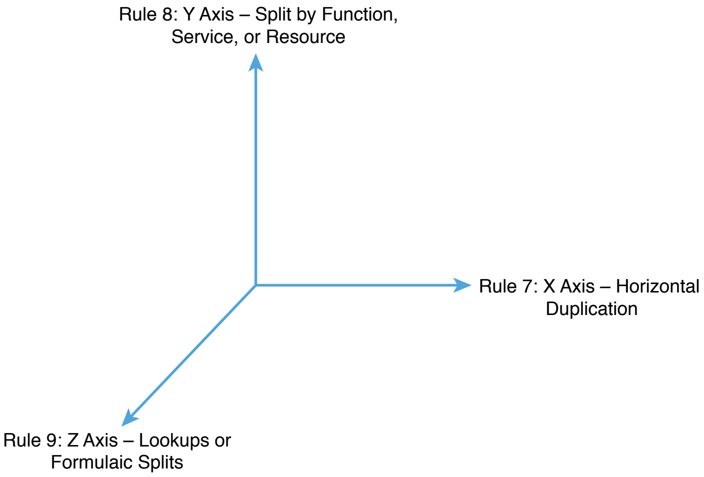

Rule 8: Y Axis – Split by Function, Service, or Resource
Rule 7: X Axis – Horizontal Duplication
Rule 9: Z Axis – Lookups or Formulaic Splits

Figure 2.2 Three axes of scale

Rule 7—Design to Clone or Replicate Things (X Axis)

Rule 7: What, When, How, and Why

What: Typically called horizontal scale, this is the duplication of services or databases to spread transaction load.

When to use:

- Databases with a very high read-to-write ratio (5:1 or greater—the higher the better).
- Any system where transaction growth exceeds data growth.

How to use:

- Simply clone services and implement a load balancer.
- For databases, ensure that the accessing code understands the difference between a read and a write.

Why: Allows for fast scale of transactions at the cost of duplicated data and functionality.

Key takeaways: X axis splits are fast to implement, are low cost from a developer effort perspective, and can scale transaction volumes nicely. However, they tend to be high cost from the perspective of operational cost of data.

Often, the hardest part of a solution to scale is the database or persistent storage tier. The beginning of this problem can be traced back to Edgar F. Codd's 1970 paper "A Relational Model of Data for Large Shared Data Banks,"[1] which is credited with introducing the concept of the relational database management system (RDBMS). Today's most popular RDBMSs, such as Oracle, MySQL, and SQL Server, just as the name implies, allow for relations between data elements. These relationships can exist within or between tables. The tables of most OLTP systems are normalized to third normal form,[2] where all records of a table have the same fields, nonkey fields cannot be described by only one of the keys in a composite key, and all nonkey fields must be described by the key. Within the table each piece of data is related to other pieces of data in that table. Between tables there are often relationships, known as foreign keys. Most applications depend on the database to support and enforce these relationships because of its ACID properties (see Table 2.1). Requiring the database to maintain and enforce these relationships makes it difficult to split the database without significant engineering effort.

Table 2.1 **ACID Properties of Databases**

Property	Description
Atomicity	All of the operations in the transaction will complete, or none will.
Consistency	The database will be in a consistent state when the transaction begins and ends.
Isolation	The transaction will behave as if it is the only operation being performed upon the database.
Durability	Upon completion of the transaction, the operation will not be reversed.

One technique for scaling databases is to take advantage of the fact that most applications and databases perform significantly more reads than writes. A client of ours that handles booking reservations for customers has on average 400 searches for a single booking. Each booking is a write and each search a read, resulting in a 400:1 read-to-write ratio. This type of system can be easily scaled by creating read-only copies (or replicas) of the data.

There are a couple of ways that you can distribute the read copy of your data depending on the time sensitivity of the data. Time (or temporal) sensitivity is how fresh or completely correct the read copy has to be relative to the write copy. Before you scream out that the data has to be instant, real time, in sync, and completely correct across the entire system, take a breath and appreciate the costs of such a system. While perfectly in-sync data is ideal, it costs . . . a lot. Furthermore, it doesn't always give you the return that you might expect or desire for that cost. Rule 19, "Relax Temporal Constraints" (see Chapter 5, "Get Out of Your Own Way"), will delve more into these costs and the resulting impact on the scalability of products.

Let's go back to our client with the reservation system that has 400 reads for every write. They're handling reservations for customers, so you would think the data they display to customers would have to be completely in sync. For starters you'd be keeping 400 sets of data in sync for the one piece of data that the customer wants to reserve. Second, just because the data is out of sync with the primary transactional database by 3 or 30 or 90 seconds doesn't mean that it isn't correct, just that there is a chance that it isn't correct. This client probably has 100,000 pieces of data in their system at any one time and books 10% of those each day. If those bookings are evenly distributed across the course of a day, they are booking one reservation just about every second (0.86 second). All things being equal, the chance of a customer wanting a particular booking that is already taken by another customer (assuming a 90-second sync of data) is 0.104%. Of course even at 0.1% some customers will select a booking that is already taken, which might not be ideal but can be handled in the application by doing a final check before allowing the booking to be placed in the customer's cart. Certainly every application's data needs are going to be different, but from this discussion we hope you will get a sense of how you can push back on the idea that all data has to be kept in sync in real time.

Now that we've covered the time sensitivity, let's start discussing the ways to distribute the data. One way is to use a caching tier in front of the database. An object cache can be used to read from instead of going back to the application for each query. Only when the data has been marked expired would the application have to query the primary transactional database to retrieve the data and refresh the cache. We highly recommend this as a first step given the availability of numerous excellent, open-source key-value stores that can be used as object caches.

The next step beyond an object cache between the application tier and the database tier is replicating the database. Most major relational database systems allow for some type of replication "out of the box." Many databases implement replication through some sort of *master-slave* concept—the master database being the primary transactional database that gets written to, and the slave databases being read-only copies of the master database. The master database keeps track of updates, inserts, deletes, and so on in a binary log. Each slave requests the binary log from the master and replays these commands on its database. While this is asynchronous, the latency between data being updated in the master and then in the slave can be very low, depending on the amount of data being inserted or updated in the master database. In our client's example, 10% of the data changed each day, resulting in one update per second. This is likely a low enough volume of change to maintain the slave databases with low latency. Often this implementation consists of several slave databases or read replicas that are configured behind a load balancer. The application makes a read request to the load balancer, which passes the request in either a round-robin or least-connections manner to a read replica. Some databases further allow replication using a master-master concept in which either database can be used to read or write. Synchronization processes help ensure the consistency and coherency of the data between the masters. While this technology has been available for quite some time, we prefer solutions that rely on a single write database to help eliminate confusion and logical contention between the databases.

We call the type of split (replication) an X axis split, and it is represented on the AKF Scale Cube in Figure 2.1 as the X axis—Horizontal Duplication. An example that many developers familiar with hosting Web applications will recognize is on the Web or application tier of a system, running multiple servers behind a load balancer all with the same code. A request comes in to the load balancer which distributes it to any one of the many Web or application servers to fulfill. The great thing about this distributed model on the application tier is that you can put dozens, hundreds, or even thousands of servers behind load balancers all running the same code and handling similar requests.

The X axis can be applied to more than just the database. Web servers and application servers typically can be easily cloned. This cloning allows the distribution of transactions across systems evenly for horizontal scale. Cloning of application or Web services tends to be relatively easy to perform and allows us to scale the number of transactions processed. Unfortunately, it doesn't really help us when trying to scale the data we must manipulate to perform these transactions. In memory, caching of data unique to several customers or unique to disparate functions might create a bottleneck that keeps us from scaling these services without significant impact on customer response time. To solve these memory constraints we'll look to the Y and Z axes of our scale cube.

Rule 8—Design to Split Different Things (Y Axis)

Rule 8: What, When, How, and Why

What: Sometimes referred to as scale through services or resources, this rule focuses on scaling by splitting data sets, transactions, and engineering teams along verb (services) or noun (resources) boundaries.

When to use:

- Very large data sets where relations between data are not necessary.
- Large, complex systems where scaling engineering resources requires specialization.

> **How to use:**
> - Split up actions by using verbs, or resources by using nouns, or use a mix.
> - Split both the services and the data along the lines defined by the verb/noun approach.
>
> **Why:** Allows for efficient scaling of not only transactions but also very large data sets associated with those transactions. Also allows for the efficient scaling of teams.
>
> **Key takeaways:** Y axis or data/service-oriented splits allow for efficient scaling of transactions, large data sets, and can help with fault isolation. Y axis splits help reduce the communication overhead of teams.

When you put aside the religious debate around the concepts of services- (SOA) and resources- (ROA) oriented architectures and look deep into their underlying premises, they have at least one thing in common. Both concepts force architects and engineers to think in terms of separation of responsibilities within their architectures. At a high and simple level, they do this through the concepts of verbs (services) and nouns (resources). Rule 8, and our second axis of scale, takes the same approach. Put simply, Rule 8 is about scaling through the separation of distinct and different functions and data within a site. The simple approach to Rule 8 tells us to split up our product by either nouns or verbs or a combination of both nouns and verbs.

Let's split up our site using the verb approach first. If our site is a relatively simple e-commerce site, we might break it into the necessary verbs of signup, login, search, browse, view, add to cart, and purchase/buy. The data necessary to perform any one of these transactions can vary significantly from the data necessary for the other transactions. For instance, while it might be argued that signup and login need the same data, they also require some data that is unique and distinct. Signup, for instance, probably needs to be capable of checking whether a user's preferred ID has been chosen by someone else in the past, whereas login might not need to have a complete understanding of every other user's ID. Signup likely needs to write a fair amount of data to some permanent data store, but login is likely a read-intensive application to validate a user's credentials. Signup may require that the user store a fair amount of personally identifiable information (PII) including credit card numbers, whereas login does not likely need access to all of this information at the time that a user would like to establish a login.

The differences and resulting opportunities for this method of scale become even more apparent when we analyze obviously distinct functions like search and login. In the case of login we are mostly concerned with validating the user's credentials and potentially establishing some notion of session (we've chosen the word *session* rather than *state* for a reason we explore in Rule 40 in Chapter 10, "Avoid or Distribute State"). Login is concerned with the user and as a result needs to cache and interact with data about that user. Search, on the other hand, is concerned with the hunt for an item and is most concerned with user intent (vis-à-vis a search string, query, or search terms typically typed into a search box) and the items that we have in stock within our catalog. Separating these sets of data allows us to cache more of them within the confines of memory available on our system and process transactions faster as a result of higher cache hit ratios. Separating this data within our back-end persistence systems (such as a database) allows us to dedicate more "in memory" space within those systems and respond faster to the clients (application servers) making requests. Both systems respond faster as a result of better utilization of system resources. Clearly we can now scale these systems more easily and with fewer memory constraints. Moreover, the Y axis adds transaction scalability by splitting up transactions in the same fashion as Rule 7, the X axis of scale.

Hold on! What if we want to merge information about the user and our products such as in the case of recommending products? Note that we have just added another verb—*recommend*. This gives us another opportunity to perform a split of our data and our transactions. We might add a recommendation service that asynchronously evaluates past user purchase behavior against users who have similar purchase behaviors. This in turn may populate data in either the login function or the search function for display to the user when he or she interacts with the system. Or it can be a separate synchronous call made from the user's browser to be displayed in an area dedicated to the result of the recommend call.

Now how about using nouns to split items? Again, using our e-commerce example, we might identify certain resources upon which we will ultimately take actions (rather than the verbs that represent the actions we take). We may decide that our e-commerce site is made up of a product catalog, product inventory, user account information, marketing information, and so on. Using our noun approach, we may decide to split up our data into these categories and then define a set of high-level primitives such as create, read, update, and delete actions on these primitives.

While Y axis splits are most useful in scaling data sets, they are also useful in scaling code bases. Because services or resources are now split, the actions we perform and the code necessary to perform them are split up as well. This means that very large engineering teams developing complex systems can become experts in subsets of those systems and don't need to worry about or become experts on every other part of the system. Teams that own each service can build the interface (such as an API) into their service and own it. Assuming that each team "owns" its own code base, we can cut down on the communication overhead associated with Brooks' Law. One tenet of Brooks' Law is that developer productivity is reduced as a result of increasing team sizes.[3] The communication effort within any team to coordinate team efforts is a square of the number of participants in the team. Therefore, with increasing team size comes decreasing developer productivity as more developer time is spent on coordination. By segmenting teams and enabling ownership, such overhead is decreased. And of course because we have split up our services, we can also scale transactions fairly easily.

Rule 9—Design to Split Similar Things (Z Axis)

> ### Rule 9: What, When, How, and Why
>
> **What:** This is very often a split by some unique aspect of the customer such as customer ID, name, geography, and so on.
>
> **When to use:** Very large, similar data sets such as large and rapidly growing customer bases or when response time for a geographically distributed customer base is important.

> **How to use:** Identify something you know about the customer, such as customer ID, last name, geography, or device, and split or partition both data and services based on that attribute.
>
> **Why:** Rapid customer growth exceeds other forms of data growth, or you have the need to perform fault isolation between certain customer groups as you scale.
>
> **Key takeaways:** Z axis splits are effective at helping you to scale customer bases but can also be applied to other very large data sets that can't be pulled apart using the Y axis methodology.

Often referred to as *sharding* and *podding*, Rule 9 is about taking one data set or service and partitioning it into several pieces. These pieces are often equal in size but may be of different sizes if there is value in having several unequally sized chunks or shards. One reason to have unequally sized shards is to enable application rollouts that limit your risk by affecting first a small customer segment, and then increasingly large segments of customers as you feel you have identified and resolved major problems. It also serves as a great method for allowing discovery—as you roll out first to smaller segments, if a feature is not getting the traction you expect (or if you want to expose an "early" release to learn about usage of a feature), you can modify the feature before it is exposed to everybody.

Often sharding is accomplished by separating something we know about the requestor or customer. Let's say that we are a time card and attendance-tracking SaaS provider. We are responsible for tracking the time and attendance for employees of each of our clients, who are in turn enterprise-class customers with more than 1,000 employees each. We might determine that we can easily partition or shard our solution by company, meaning that each company could have its own dedicated Web, application, and database servers. Given that we also want to leverage the cost efficiencies enabled by multitenancy, we also want to have multiple small companies exist within a single shard. Really big companies with many employees might get dedicated hardware, whereas smaller companies with fewer employees could cohabit within a larger number of shards. We have leveraged the fact that there is a relationship between employees and companies to create scalable partitions of systems that allow us to employ smaller, cost-effective hardware and scale horizontally (we discuss horizontal scale further in Rule 10 in the next chapter).

Maybe we are a provider of advertising services for mobile phones. In this case, we very likely know something about the end user's device and carrier. Both of these create compelling characteristics by which we can partition our data. If we are an e-commerce player, we might split users by their geography to make more efficient use of our available inventory in distribution centers, and to give the fastest response time on the e-commerce Web site. Or maybe we create partitions of data that allow us to evenly distribute users based on the recency, frequency, and monetization of their purchases. Or, if all else fails, maybe we just use some modulus or hash of a user identification (userid) number that we've assigned the user at signup.

Why would we ever decide to partition similar things? For hyper-growth companies, the answer is easy. The speed with which we can answer any request is at least partially determined by the cache hit ratio of near and distant caches. This speed in turn indicates how many transactions we can process on any given system, which in turn determines how many systems we need to process a number of requests. In the extreme case, without partitioning of data, our transactions might become agonizingly slow as we attempt to traverse huge amounts of monolithic data to come to a single answer for a single user. Where speed is paramount and the data to answer any request is large, designing to split different things (Rule 8) and similar things (Rule 9) becomes a necessity.

Splitting similar things obviously isn't just limited to customers, but customers are the most frequent and easiest implementation of Rule 9 within our consulting practice. Sometimes we recommend splitting product catalogs, for instance. But when we split diverse catalogs into items such as lawn chairs and diapers, we often categorize these as splits of different things. We've also helped clients shard their systems by splitting along a modulus or hash of a transaction ID. In these cases, we really don't know anything about the requestor, but we do have a monotonically increasing number upon which we can act. These types of splits can be performed on systems that log transactions for future reference as in a system designed to retain errors for future evaluation.

Summary

We maintain that three simple rules can help you scale nearly everything. Scaling along the X, Y, and Z axes each has its own set of benefits. Typically X axis scaling has the lowest cost from a design and software development perspective; Y and Z axis scaling is a little more challenging to design but gives you more flexibility to further fully separate your services, customers, and even engineering teams. There are undoubtedly more ways to scale systems and platforms, but armed with these three rules, few if any scale-related problems will stand in your way:

- **Scale by cloning**—Cloning or duplicating data and services allows you to scale transactions easily.
- **Scale by splitting different things**—Use nouns or verbs to identify data and services to separate. If done properly, both transactions and data sets can be scaled efficiently.
- **Scale by splitting similar things**—Typically these are customer data sets. Set customers up into unique and separated shards or swim lanes (see Chapter 9 for the definition of *swim lane*) to enable transaction and data scaling.

Notes

1. Edgar F. Codd, "A Relational Model of Data for Large Shared Data Banks," 1970, www.seas.upenn.edu/~zives/03f/cis550/codd.pdf.
2. Wikipedia, "Third Normal Form," http://en.wikipedia.org/wiki/Third_normal_form.
3. Wikipedia, "Brooks' Law," https://en.wikipedia.org/wiki/Brooks'_law.

3

Design to Scale Out Horizontally

Within our practice, we often tell clients, "Scaling up is failing up." What does that mean? In our minds, it is clear: we believe that within hyper-growth environments it is critical that companies plan to scale in a horizontal fashion—what we describe as scaling out. Most often this is done through the segmentation or duplication of workloads across multiple systems. The practice or implementation of that segmentation often looks like one of the approaches we described in Chapter 2, "Distribute Your Work." When hyper-growth companies do not scale out, their only option is to buy bigger and faster systems. When they hit the limitation of the fastest and biggest system provided by the most costly provider of the system in question, they are in big trouble. Ultimately, this is what hurt eBay in 1999, and we still see it more than a decade later in our business and with our clients today. The constraints and problems with scaling up aren't only physical issues. Often they are caused by a logical contention that bigger and faster hardware simply can't solve. Before we dig into this topic, let's hear from a CTO who learned this lesson through the implementation of firewalls.

Chris Schremser is the CTO at ZirMed, a revenue cycle management company that empowers healthcare organizations to simultaneously optimize their revenue and their patient population (customer) health with a comprehensive end-to-end platform. Because the ZirMed system deals with sensitive patient data (PII) covered under the Health Insurance Portability and Accountability Act (HIPAA), Chris and his team designed the data center's network with very restrictive firewall policies. Communication between tiers (Web server to application server) was required to pass through firewalls. Chris recalled, "Our original design called for two very large firewalls that had a perspective on every subnet that we have in our product. When we were very small, that worked very well. We were able to just continually add hardware whenever we got close to capacity thresholds. Buy bigger. We would always buy bigger. We started off with a very, very small set of devices in the early days, and then we just continued to scale up. And the final addition, before we changed the model, was two very large devices that were carrier-grade devices. They were blade chassis with four blades per chassis."

The two devices were set up in high-availability (HA) mode, allowing for what the vendor claimed to be seamless failover between the pair. Unfortunately, ZirMed's product relies on maintaining state during sessions, and this session state did not fail over gracefully between the firewall pair. Chris continued, "So, the firewall vendor would always tell us that it's a seamless failover because we have a high-speed network between the chassis, and we're maintaining all the sockets and sessions. But their idea of seamless and my idea of seamless are two very different things because we have applications that failed over every time we'd have one of these events—sockets would stop connecting for a little bit; Web servers would error out trying to connect back to the database. In fact, one of the things that we came to understand is that the vendor had a relatively aggressive failover policy. If the device was running in a single-chassis configuration, it only rebooted at the very last step of panic, whereas if it was running in a pair, it would be a little bit more aggressive, and when something weird was happening on the primary, it would go ahead and fail over to the secondary. So in operating in a two-node HA pair, the rate of failure per device would be higher than in a single-node system. That in turn would cause our applications an inordinate amount of pain."

ZirMed was plagued with other firewall issues because of the complexity of their implementation. "We use a content addressable storage system and it relies on multicast across all the nodes to tell it who's got what content," Chris explained. "We had tried to segment that off, and we thought we had actually segmented that off. Turns out we had missed that, and all that UDP [User Datagram Protocol] traffic was passing through the firewall. It was overwhelming the firewall, and so connections wouldn't even make it up into the blades, but we couldn't even see this—we had to actually call the vendor in to see this. Because it's so complex, with a chassis and a lot of blades and a lot of distribution, we really lost a lot of visibility into how the system was working."

Chris and his team started having debates about buying another very large pair of firewalls but from a different vendor, but they had a bit of an "aha moment." Chris reflected, "When we build software, we know that we have to scale out, not up. Why are we not applying those exact same methodologies in our hardware purchases? And just that one simple question that the team asked completely changed how we thought. And so, in the end, our implementation went from being bigger, bigger, bigger that does all this protection into now very targeted, smaller firewalls."

In ZirMed's new design they implemented four pairs of firewalls. While slightly more complex from a management perspective, the devices themselves are much simpler. Here again we see this troubling notion of "complexity." When used one way, more devices equals more complexity—or as we prefer to indicate, more devices to manage and oversee. But when seen from another perspective, more devices equals lower complexity—lower rates of failure overall and fewer incidents to manage. "Now, we've got much smaller devices, which vendors know how to build very, very well," Chris pointed out. "They're very simple. They're very well understood. It allows us to also be more intelligent about what runs on these devices, and more importantly, the design now allows for four to become six or eight or ten or 12, or whatever we need. We can just keep adding individual devices, carving off pieces of the network. Sure, we have to make some firewall rule adjustments, but it allows us to scale out at the hardware layer instead of scaling up."

The benefits of scaling out and not up are many. Chris concluded, "We've already seen tremendous wins, and wins in ways that we didn't even understand. One of the things that we are planning on doing is to push down rule creation for our development environment into the application engineering teams. They don't have to be firewall experts. They can go ahead and create those with some review process because we know before they get promoted to production they are going to be reviewed by a networking engineer, and because we're blocking external traffic at the wide-area network pair of firewalls, we haven't increased the risk. It has allowed us to not only be flexible in our growth, but be flexible in our provisioning, flexible in terms of empowering our teams."

Now that we've heard how Chris learned the importance of scaling out instead of up, let's discuss some rules about this topic. In this chapter we are going to discuss some thoughts around designing your systems to scale horizontally, or *out*, rather than *up*, using commodity hardware, and even how this concept can translate into data centers.

Rule 10—Design Your Solution to Scale Out, Not Just Up

> ### Rule 10: What, When, How, and Why
>
> **What:** *Scaling out* is the duplication or segmentation of services or databases to spread transaction load and is the alternative to buying larger hardware, known as *scaling up*.
>
> **When to use:** Any system, service, or database expected to grow rapidly or that you would like to grow cost-effectively.
>
> **How to use:** Use the AKF Scale Cube to determine the correct split for your environment. Usually the horizontal split (cloning) is the easiest.
>
> **Why:** Allows for fast scale of transactions at the cost of duplicated data and functionality.
>
> **Key takeaways:** Plan for success and design your systems to scale out. Don't get caught in the trap of expecting to scale up only to find out that you've run out of faster and larger systems to purchase.

What do you do when faced with rapid growth of customers and transactions on your systems and you haven't built them to scale to multiple servers? Ideally you'd investigate your options and decide you could either buy a larger server or spend engineering time to enable the product to run on multiple servers. Having the ability to run your product on multiple servers through all tiers is *scaling out*. Continuing to run your systems on larger hardware at any tier is *scaling up*. In your analysis, you might come to the decision through an ROI calculation that it is cheaper to buy the next-larger server rather than spend the engineering resources required to change the application. While we would applaud the analytical approach to this decision, for high-growth companies and products it's probably flawed, and in our experience it is often flawed in moderate-growth companies as well.

The most common flaw in such calculations is a failure to extend the analysis into multiple years of technology refreshes for the underlying hardware and infrastructure. Moving from a machine with two 64-bit octa-core processors to one with four octa-core processors will likely cost proportionally exactly what you get in improved computational resources (roughly two times the cost and likely something slightly less than two times the improvement subject to Amdahl's Law). The fallacy comes in as we continue to purchase larger servers with more processors. This curve of cost to computational processing is a power law in which the cost begins to increase disproportionally to the increase in processing power provided by larger servers (see Rule 11). Assuming that your company continues to succeed and grow, you will continue to travel up the curve in costs for bigger systems. While you may budget for technology refreshes over time, you will be forced to purchase systems at an incredibly high price point relative to the cheaper systems you could purchase if you had built to scale horizontally. Overall, your total capital expenditures increase significantly. Of course the costs to solve the problem with engineering resources will also likely increase due to the increased size of the code base and complexity of the system, but this cost should be linear. Thus your analysis in the beginning should have resulted in a decision to spend the time up front changing the code to scale out.

Using an online pricing and configuration utility from one of the large server vendors, the graph in Figure 3.1 shows the cost of seven servers, each configured as closely as possible to one another (RAM, disk, and so on) except for the increasing number of processors and cores per processor. Admittedly the computational resource from two quad-core processors is not exactly equivalent to a single octa-core, but for this cost comparison it is close enough. Notice the exponential trend line that fits the data points.

In our experience with more than 400 clients, this type of analysis almost always results in the decision to modify the code or database to scale out instead of up. The repetitive costs of these larger servers through the refresh cycles of multiple technologies compound the problem and seal the deal. That's why it's an AKF Partners' belief that scaling up is failing up. Eventually you will get to a point where either the cost becomes uneconomical or there is no bigger hardware made. For example, we had a client who did have some ability to split their customers onto different systems but continued to scale up with their database hardware. They eventually topped out with six of the largest servers made by their preferred hardware vendor. Each of these systems cost more than $3 million, totaling nearly $20 million in hardware cost. Struggling to continue to scale as their customer demand increased, they undertook a project to horizontally scale their databases. They were able to replace

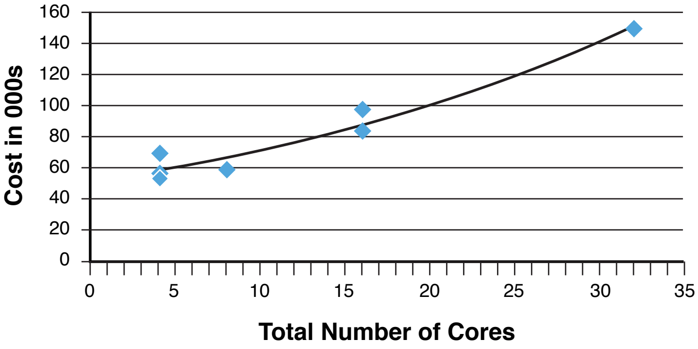


Figure 3.1 Cost per core

each of those large servers with four much smaller servers costing $350,000 each. In the end they not only succeeded in continuing to scale for their customers but realized a savings of almost $10 million. The company continued to use the old systems until they ultimately failed with age and could be replaced with newer, smaller systems at lower cost.

Most applications are either built from the start to allow them to be run on multiple servers or can be easily modified to accommodate this. For most SaaS applications this is as simple as replicating the code onto multiple application servers and putting them behind a load balancer. The application servers need not know about each other, but each request gets handled by whichever server gets sent the request from the load balancer. If the application has to keep track of state (see Chapter 10, "Avoid or Distribute State," for why you will want to eliminate this), a possible solution is to allow session cookies from the load balancer to maintain affinity between a customer's browser and a particular application server. Once the customer has made an initial request, the server responding to that request will continue to handle that customer until the session has ended.

For a database to scale out often requires more planning and engineering work, but as we explained in the beginning this is almost always effort well spent. In Chapter 2, we covered three ways in which you can scale an application or database. These are identified on the AKF Scale Cube as X, Y, and Z axes corresponding to replicating (cloning), splitting different things (services), and splitting similar things (customers).

"But wait!" you cry. "Intel's cofounder Gordon Moore predicted in 1965 that the number of transistors that can be placed on an integrated circuit will double every two years." That's true. Moore's Law has amazingly held true for over 50 years now. The problem with this is that this "law" cannot hold true forever, as Gordon Moore admitted in a 2005 interview.[1] Additionally, if yours is a true hyper-growth company, you are growing faster than just doubling customers or transactions every two years. You might be doubling every quarter. Furthermore, Amdahl's Law hints that you are not likely to get the full benefit of Moore's Law unless you spend a great deal of time optimizing your solution to be parallelized. Relying on Moore's Law to scale your system, whether it's your application or database, is likely to lead to failure.

Rule 11—Use Commodity Systems (Goldfish Not Thoroughbreds)

Rule 11: What, When, How, and Why

What: Use small, inexpensive systems where possible.

When to use: Use this approach in your production environment when going through hyper-growth and adopt it as an architectural principle for more mature products.

How to use: Stay away from very large systems in your production environment.

Why: Allows for fast, cost-effective growth. Allows you to purchase the capacity you need rather than spending for unused capacity far ahead of need.

Key takeaways: Build your systems to be capable of relying on commodity hardware, and don't get caught in the trap of using high-margin, high-end servers.

Hyper-growth can be a lonely place. There's so much to learn and so little time to do that learning. But rest assured, if you follow our advice, you'll have lots of friends—lots of friends that draw power, create heat, push air, and do useful moneymaking tasks—computers. And in our world, the world of hyper-growth, we believe that a lot of little low-cost "goldfish" are better than a few big high-cost "thoroughbreds."

One of my favorite lines from an undergraduate calculus book is "It should be intuitively obvious to the casual observer that <insert some totally nonobvious statement here>." This particular statement left a mark on me, primarily because what the author was presenting was neither intuitive nor obvious to me at the time. It might not seem obvious that having more of something, like many more "smaller" computers, is a better solution than having fewer, larger systems. In fact, more computers probably mean more power, more space, and more cooling. The reason more and smaller is often better than less and bigger is twofold and described later in this chapter.

Your equipment provider benefits from selling you into their highest-margin products. For nearly every provider, the products with the highest margins tend to be the largest products with the largest number of processors. Why is this so? Many companies rely on faster, bigger hardware to do their necessary workloads and are simply unwilling to invest in scaling their own infrastructure. As such, the equipment manufacturers can hold these companies hostage with higher prices and achieve higher margins. But there is an interesting conundrum within this approach as these faster, bigger machines aren't really capable of doing more work compared to an equivalent number of processors in smaller systems. On a per-CPU basis, these machines have less horsepower and are less efficient than smaller systems. As you add CPUs, each CPU does slightly less work than it would in a single-CPU system (regardless of cores). There are many reasons for this, including the inefficiency of scheduling algorithms for multiple processors, conflicts with memory bus access speeds, structural hazards, data hazards, and so on. In virtualized environments, the overhead of managing virtual machines increases with the size of the physical host. The sweet spot appears to be two- or four-socket physical hosts.

Think carefully about what we just said. You are paying more on a CPU basis but actually doing less per CPU. The vendor is nailing you twice!

When confronted with the previous information, your providers will likely go through the relatively common first phase of denial. The wise ones will quickly move on and indicate that your total cost of ownership will go down as the larger units draw less aggregate power than the smaller units. In fact, they might say, you can work with one of their partners to partition (or *virtualize*) the systems to get the benefit of small systems and lower power drain. This brings us to our second point: We must do some math.

It might, in fact, be the case that the larger system will draw less power and save you money. As power costs increase and system costs decrease, there is no doubt that there is a "right size" system for you that maximizes power, system cost, and computing power. But your vendors aren't the best source of information for this. You should do the math on your own. It is highly unlikely that you should purchase the largest system available because that math almost never works. To figure out what to do with the vendors' arguments, let's break them down into their component parts.

The math is easy. Look at power cost and unit power consumption as compared to an independent third-party benchmark on system utilization. We can find the right system for us that still fits the commodity range (in other words hasn't been marked up by the vendor as a high-end system) and maximizes the intersection of computing power with minimal power and space requirements. Total cost of ownership, in nearly all cases and when considering all costs, typically goes down.

On the topic of virtualization, remember that no software comes for free. There are many reasons to *virtualize* (or in the old language *domain* or *partition*) systems. But you never virtualize a system into four separate domains and end up with more system processing power and throughput than if you had just purchased four systems equivalent to the size of each domain. Remember that the virtualization software has to use CPU cycles to run and that it's getting those cycles from somewhere. Again, there are many reasons to virtualize, but greater system capacity in a larger domained system as compared to smaller equivalently sized systems is a fallacy and is not one of them.

What are the other reasons we might want to use commodity systems as compared to more costly systems? We are planning to scale aggressively and there are economies to our rate of scaling. We can more easily negotiate for commodity systems. While we might have more of them, it is easier to discard them and work on them at our leisure than the more expensive systems that will demand time. While this may seem counterintuitive, we have been successful in managing more systems with less staff in the commodity (goldfish) world than in the costly system (thoroughbred) world. We pay less for maintenance on these systems, we can afford more redundancy, and they fail less often due to having fewer parts (CPUs, for example) on a per-unit basis.

And, ultimately, we come to why we call these things "goldfish." At scale, these systems are very inexpensive. If they "die," you are probably incented to simply throw them away rather than investing a lot of time to fix them. "Thoroughbreds," on the other hand, represent a fairly large investment and will take time to maintain and fix. Ultimately, we prefer to have many little friends rather than a few big friends.

Rule 12—Scale Out Your Hosting Solution

Rule 12: What, When, How, and Why

What: Design your systems to have three or more live data centers to reduce overall cost, increase availability, and implement disaster recovery. A data center can be an owned facility, a colocation, or a cloud (IaaS or PaaS) instance.

When to use: Any rapidly growing business that is considering adding a disaster recovery (cold site) data center or mature business looking to optimize costs with a three-site solution

How to use: Scale your data per the AKF Scale Cube. Host your systems in a "multiple live" configuration. Use IaaS/PaaS (cloud) for burst capacity, new ventures, or as part of a three-site solution.

Why: The cost of data center failure can be disastrous to your business. Design to have three or more as the cost is often less than having two data centers. Consider using the cloud as one of your sites, and scale for peaks in the cloud. Own the base; rent the peak.

> **Key takeaways:** When implementing disaster recovery, lower your cost by designing your systems to leverage three or more live data centers. IaaS and PaaS (cloud) can scale systems quickly and should be used for spiky demand periods. Design your systems to be fully functional if only two of the three sites are available, or N-1 sites available if you scale to more than three sites.

The data center has become one of the biggest pain points in scaling for rapidly growing companies. This is because data centers take a long time to plan and build out and because they are often one of the last things that we think about during periods of rapid growth. And sometimes that "last thing" that we think about is the thing that endangers our company most.

Building and operating data centers requires core competencies that rarely overlap with a technology team; avoid owning a data center until the scale of your company is large enough to achieve the cost savings possible through building and operating your own data centers. Make use of colocation providers and cloud providers while you grow; let them bear the risk of data center operations as well as the timelines to build them. The minimum efficient scale to achieve cost savings with an owned data center is approximately 3MW of IT capacity—about 9,000 two-socket servers. The time needed to build such a data center is 12 months for experienced companies using standardized designs and equipment to 24 months for a new design and the accompanying equipment selection process. The capital required ranges from $30 million to $50 million depending upon the level of facility redundancy specified. Building and operating data centers is not a simple task. Avoid it until your company is large enough for it to make good business sense.

This rule is a brief treatment of the "how" and "why" to split up data centers for rapid growth.

First, let's review a few basics. For the purposes of fault isolation (which helps create high availability) and transaction growth, we are going to want to segment our data using both the Y and Z axes of scale presented in Rules 8 and 9, respectively. For the purposes of high availability and transaction growth, we are going to want to replicate (or clone) data and services along the X axis as described in Rule 7. Finally, we are going to assume that you've attempted to apply Rule 40 (see Chapter 10, "Avoid or Distribute State") and that you either have a stateless system or can design around your stateful needs to allow for multiple data centers. It is this segmentation, replication, and cloning of data and services as well as *statelessness* that form the building blocks for us to spread our data centers across multiple sites and geographies. Standardize system configuration, code deployment, and monitoring to enable seamless growth between colocation sites and cloud sites.

If we have sliced our data properly along the Z axis (see Rule 9), we can now potentially locate data closer to the users requesting that data. If we can slice data while maintaining multitenancy by individual users, we can choose data center locations that are near our end users. If the "atomic" or "granular" element is a company, we might also locate next to the companies we serve (or at least the largest employee bases of those companies if it is a large company).

Let's start with three data centers. Each data center is the "home" for roughly 33% of our data. We will call these data sets A, B, and C. Each data set in each data center has its data replicated in halves, 50% going to each peer data center. Assuming a Z axis split (see Rule 9) and X axis (see Rule 7) replication of data, 50% of data center A's customers would exist in data center B, and 50% would exist in data center C. In the event of any data center failure, 50% of the data and associated transactions of the data center that failed will move to its peer data centers. If data center A fails, 50% of its data and transactions will go to data center B and 50% to data center C. This approach is depicted in Figure 3.2. The result is that you have 200% of the data necessary to run the site in aggregate, but each site contains only 66% of the necessary data as each site contains the copy for which it is a master (33% of the data necessary to run the site) and 50% of the copies of each of the other sites (16.5% of the data necessary to run the site for a total of an additional 33%).

To see why this configuration is better than the alternative, let's look at some math. Implicit in our assumption is that you agree that you need at least two data centers to stay in business in the event of a geographically isolated disaster. If you have two data centers labeled A and B, you might decide to operate 100% of your traffic out of data center A and leave data center B for a warm standby. In a hot/cold (or active/passive) configuration you would need 100% of your computing and network assets in both data centers to include 100% of your Web and application servers, 100% of your database servers, and 100% of your network equipment. Power needs would be similar and Internet connectivity would be similar. You probably keep slightly more than 100% of the capacity necessary to serve your peak demand in each location to handle surges in demand. So let's say that you keep 110% of your needs in both locations. Anytime you buy additional servers for one place, you have to buy them for the other. You may also decide to connect the data centers with your own dedicated circuits for the purpose of secure replication of data. Running live out of both sites would help you in the event of a major catastrophe because only 50% of your transactions would initially fail until you transfer that traffic to the alternate site, but it won't help you from a budget or financial perspective. A high-level diagram of the data centers may look as depicted in Figure 3.3.

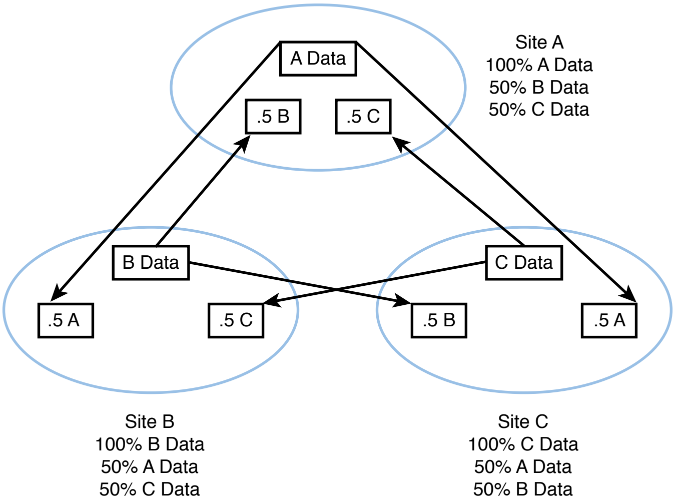


Figure 3.2 Split of data center replication

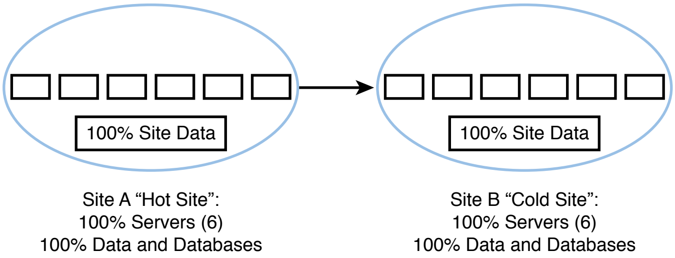


Figure 3.3 Two-data-center configuration, "hot" and "cold" sites

But with three live sites, our costs go down. This is because for all nondatabase systems we only really need 150% of our capacity in each location to run 100% of our traffic in the event of a site failure. For databases, we still need 200% of the storage, but that cost stays with us no matter what approach we use. Power and facilities consumption should also be at roughly 150% of the need for a single site, though obviously we will need slightly more people, and there's probably slightly more overhead than 150% to handle three sites versus one. The only area that increases disproportionately is the network interconnects because we need two additional connections (versus one) for three sites versus two. Our new data center configuration is shown in Figure 3.4, and the associated comparative operating costs are listed in Table 3.1. Any one of the three sites could be a cloud site, which can also reduce the increased staffing needed to operate from three sites.

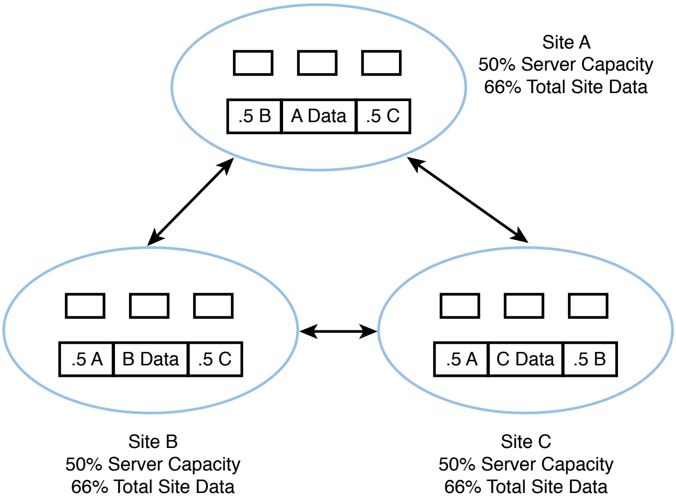


Figure 3.4 Three-data-center configuration, three hot sites

Table 3.1 **Cost Comparisons**

Site Configuration	Network	Servers	Databases	Storage	Network Site Connections	Total Cost
Single Site	100%	100%	100%	100%	0	100%
2-Site "Hot" and "Cold"	200%	200%	200%	200%	1	200%
2-Site Live/Live	200%	200%	200%	200%	1	200%
3-Site Live/Live/Live	150%	150%	150%	200%	3	~166%

As we've shown, choosing to operate with three live sites instead of two reduces hardware costs by 25% (from 200% of peak need to 150%). One could extrapolate this and think a business needing 50 servers for peak demand should buy 51 servers and host them in 51 different colocations. From an academic perspective, this does minimize hardware spend, but the network and management costs of trying to use so many sites make the idea ludicrous.

One great benefit of a three-site configuration is the ability to leverage idle capacity for the creation of testing zones (such as load and performance tests) and the ability to leverage these idle assets during spikes in demand. These spikes can come at nearly any time. Perhaps we get some exceptional and unplanned press, or maybe we just get some incredible viral boost from an exceptionally well-connected individual or company. The capacity we have on hand for a disaster starts getting traffic, and we quickly order additional capacity. Voilà!

As we've hinted, running three or more sites comes with certain drawbacks. While the team gains confidence that each site will work because all of them are live, there is some additional operational complexity. We believe that, while some additional complexity exists, it is not significantly greater than attempting to run a hot and cold site. Keeping two sites in sync is tough, especially when the team likely doesn't get many opportunities to prove that one of the two sites would actually work if needed. Constantly running three sites is a bit tougher, but not significantly so.

Network transit costs also increase at a fairly rapid pace even as other costs ultimately decline. For a fully connected graph of sites, each new site (N+1) requires N additional connections where N is the previous number of sites. Companies that handle this cost well typically negotiate for volume discounts and play third-party transit providers off of each other for reduced cost. An additional option is to haul traffic to a peering site to optimize cost and performance, particularly if traffic peaks are irregular or are significantly larger than baseline traffic. This requires careful analysis of the benefits possible versus the cost to haul traffic to a peering location.

Of course with the elasticity benefits of IaaS, you need not "own" everything. One option is to "rent" each of your three data centers in various geographic locations (such as Amazon AWS's "regions"). Alternatively, you might decide to employ a hybrid approach, keeping a mix of colocation facilities and owned data centers, then augmenting them with dynamic scaling into the public cloud on a seasonal or daily basis, depending on the demand.

Finally, we expect to see an increase in employee and employee-related costs with a multiple live site model. If our sites are large, we may decide to colocate employees near the sites rather than relying on remote-hands work. Even without employees on site, we will likely need to travel to the sites from time to time to validate setups, work with third-party providers, and so on. Potential staff increases can be mitigated if one site is cloud and thus requires minimal on-site work. Standardization of configurations, deployment tools, and monitoring will help optimize staff size as the number of servers overall grows. While you are performing your cost calculations, remember that use of multiple data centers has other benefits, such as ensuring that those data centers are close to end customers to reduce page load times. The "Multiple Live Site Considerations" sidebar summarizes the benefits, drawbacks, and architectural considerations of a multiple live site implementation.

Multiple Live Site Considerations

Multiple live site benefits include

- Higher availability as compared to a hot and cold site configuration
- Lower costs compared to a hot and cold site configuration
- Faster customer response times if customers are routed to the closest data center for dynamic calls
- Greater flexibility in rolling out products in an SaaS environment
- Greater confidence in operations versus a hot and cold site configuration
- Fast and easy "on-demand" growth for spikes using spare capacity in each data center, particularly if PaaS/IaaS/cloud is part of the overall solution

Drawbacks or concerns of a multiple live site configuration include

- Greater operational complexity
- Likely a small increase in headcount
- Increase in travel and network costs

Architectural considerations in moving to a multiple live site environment include

- Eliminating the need for state and affinity wherever possible
- Routing customers to the closest data center if possible to reduce dynamic call times
- Investigating replication technologies for databases and state if necessary

Rule 13—Design to Leverage the Cloud

Rule 13: What, When, How, and Why

What: This is the purposeful utilization of cloud technologies to scale on demand.

When to use: When demand is temporary, spiky, and inconsistent and when response time is not a core issue in the product. Consider when you are "renting your risk"—when future demand for new products is uncertain and you need the option of rapid change or walking away from your investment. Companies moving from two active sites to three should consider the cloud for the third site.

How to use:

- Make use of third-party cloud environments for temporary demand, such as seasonal business trends, large batch jobs, or quality assurance (QA) environments during testing cycles.
- Design your application to service some requests from a third-party cloud when demand exceeds a certain peak level. Scale in the cloud for the peak, then reduce active nodes to a basic level.

Why: Provisioning of hardware in a cloud environment takes a few minutes as compared to days or weeks for physical servers in your own colocation facility. When used temporarily, this is also very cost-effective.

Key takeaways: Design to leverage virtualization in all sites and grow in the cloud to meet unexpected spiky demand.

Cloud computing is part of the IaaS offering provided by many vendors such as Amazon.com, Google, IBM, and Microsoft Corporation. Vendor-provided clouds have four primary characteristics: pay by usage, scale on demand, multiple tenants, and virtualization. Third-party clouds are generally composed of many physical servers that run a hypervisor software, allowing them to emulate smaller *virtual* servers. For example, an eight-processor machine with 32GB of RAM might be divided into four machines, each allowed to utilize two processors and 8GB of RAM.

Customers can *spin up* or start using one of these virtual servers and are typically charged according to how long they use it. Pricing is different for each of the vendors providing these services, but typically the break-even point for using a virtual server versus purchasing a physical server is around 12 months. This means that if you are using the server 24 hours a day for 12 months, you will exceed the cost of purchasing the physical server. When these virtual servers can be started and stopped based on demand, cost savings are possible. Thus, if you need this server for only six hours per day for batch processing, your break-even point is extended for upward of 48 months.

While cost is certainly an important factor in your decision to use a cloud, another distinct advantage of the cloud is that provisioning of the hardware typically takes minutes as compared to days or weeks with physical hardware. The approval process required in your company for additional hardware and the steps of ordering, receiving, racking, and loading a server can easily take weeks. In a cloud environment, additional servers can be brought into service in minutes. Additionally, your colocation site may not have capacity ready for new hardware, thus increasing lead time while new racks are built out.

The two ideal ways that we've seen companies make use of third-party cloud environments is when demand is either temporary or inconsistent. Temporary demand can come in the form of nightly batch jobs that need intensive computational resources for a couple of hours or from QA cycles that occur for a couple of days each month when testing the next release. Inconsistent demand can come in the form of promotions or seasonality such as Cyber Monday.

One of our clients makes great use of a third-party cloud environment each night when they process the day's worth of data into their data warehouse. They spin up hundreds of virtual instances, process the data, and then shut the instances down, ensuring that they pay only for the amount of computational resources that they need. Another of our clients uses virtual instances for their QA engineers. They build a machine image of the software version to be tested, and then as QA engineers need a new environment or a refreshed environment, they allocate a new virtual instance. Because they use virtual instances for their QA environment, the dozens of testing servers don't remain unused the majority of the time. Yet another of our clients uses a cloud environment for ad serving when their demand exceeds a certain point. A data store is synchronized every few minutes, so the ads served from the cloud are nearly as up-to-date as those served from the colocation facility. This particular application can handle a slight delay in the synchronization of data because serving an ad when requested, even if not absolutely the best ad, is still much better than not serving the ad because of scaling issues.

Another attractive attribute of the cloud is that it is ideal for "renting risk." Let's say that your company is a new startup and you are uncertain if there will be demand for your product. Alternatively, yours may be an established company with plans for new products to address your existing market or a new market. Perhaps you are adding functionality to your existing product in the form of separately deployable services to your existing solution but are uncertain if your customers will adopt that functionality. Each of these cases represents risk—risk that the customer will not adopt whatever you have built. In these cases, it makes sense to rent the capacity associated with that risk so that you can walk away from it without eating the expense of the hardware.

Think about your system and what parts are most ideally suited for a cloud environment. Often there are components, such as batch processing, testing environments, or surge capacity, that make sense to put in a cloud. Cloud environments allow for scaling on demand with very short notice.

Summary

While scaling up is an appropriate choice for slow- to moderate-growth companies, those companies whose growth consistently exceeds Moore's Law will find themselves hitting the computational capacity limits of high-end, very expensive systems with little notice. Nearly all the high-profile service failures about which we've all read have been a result of products simply outgrowing their "britches." We believe it is always wise to plan to scale out early so that when the demand comes, you can easily split up systems. Follow our rules of scaling out both your systems and your data centers, leveraging the cloud for unexpected demand, and relying on inexpensive commodity hardware, and you will be ready for hyper-growth when it comes!

Notes

1. Manek Dubash, "Moore's Law Is Dead, Says Gordon Moore," *TechWorld*, April 13, 2005, www.techworld.com/news/operating-systems/moores-law-is-dead-says-gordon-moore-3576581/.

4

Use the Right Tools

You may never have heard of Abraham Maslow, but there is a good chance that you know of his "Law of the Instrument," otherwise known as Maslow's Hammer. Paraphrased, it goes something like "When all you have is a hammer, everything looks like a nail." There are at least two important implications of this "law."

The first is that we all tend to use instruments or tools with which we are familiar to solve the problems before us. If you are a C programmer, you will likely try to solve a problem or implement requirements within C. If you are a database administrator (DBA), there is a good chance that you'll think in terms of how to use a database to solve a given problem. If your job is to maintain a third-party e-commerce package, you might try to solve nearly any problem using that package rather than simpler solutions that might require a two- to three-line interpreted shell script.

The second implication of this law really builds on the first. If within our organizations we consistently bring in people with similar skill sets to solve problems or implement new products, we will very likely get consistent answers built with similar tools and third-party products. The problem with such an approach is that while it has the benefit of predictability and consistency, it may very well drive us to use tools or solutions that are inappropriate or suboptimal for our task. Let's imagine we have a broken sink. Given Maslow's Hammer, we would beat on it with our hammer and likely cause further damage. Extending this to our topic of scalability, why would we want to use a database when just writing to a file might be a better solution? Why would we want to implement a firewall if we are going to block only certain ports and we have that ability within our routers?

We're going to hear from two technology executives with more than four decades of experience between them. One executive has spent the majority of his career leading teams implementing systems required to scale for hundreds of millions of users. The other executive started his career implementing large-scale systems and most recently has been providing scalable infrastructure for technologists. Each of them shares his unique perspective on the importance of using the right tools.

James Barrese has spent the majority of his multidecade career at eBay and the now-spun-off PayPal. Between these two companies he has held a variety of jobs, ranging from architect to VP of technology to his current position as the CTO and SVP of payment services business at PayPal. He has also served on several boards and is often asked for advice by tech companies. He has been around Silicon Valley long enough that he's seen all different types of organizations and technologists that make them up. When the authors asked him how he's seen teams struggle to use the right tools, he started off with some sound advice: "Using the right tool for the right job at the right time in an organization's lifecycle is critical. This is a balancing act that requires judgment, especially in a large organization. Some teams suffer from always chasing the 'next cool tool.' Their infrastructure ends up being littered with a myriad of different tools, none of them hardened, robust, or able to be supported at scale. On the flip side, some organizations get good at just one thing, and they take that one thing way too far."

James continued with a story: "For example, I've seen a situation where we had a very large organization that was really good at scaling one of the leading enterprise databases to a massive scale. Unfortunately, every challenge was solved using that enterprise-grade database technology. While it was a solution that was supportable by the organization, and scalable, the result was that the tool got to be overused. Things like simple frequent reads, temporal updates, and experimental new apps all went through a very high-end and costly infrastructure. Additionally, as the business grew, it required ever-higher levels of availability and scale. This in turn required more high-end expensive hardware and more expensive database licenses, as well as processes that were slower than ideal. This one system was carrying all the weight of everything the organization wanted to do. While it worked, and the execution risk for projects was lower, this is a classic example of overusing a tool."

When asked how teams get themselves into this overuse situation, James explained, "We have all seen the waves of advances in PaaS solutions, caching solutions, NoSQL data solutions, as well as efficient big-data tools and infrastructure. These tools present modern approaches to solving problems more effectively than older tools often can. Unfortunately for many organizations, a lack of experimentation and adoption of these newer technologies has led to tool lock-in and overuse. This situation of overuse of an infrastructure tool is common in many organizations. It's understandable that an organization that is growing quickly and moving fast doesn't have a lot of time or the ability to inject significant risks in meeting important business objectives. Additionally, introducing new tools comes with the trade-off of making existing tools better. This can lead an organization down a blind alley of always building up a core infrastructure while never experimenting and finding better tools that might solve problems more elegantly, efficiently, and with better results."

James concluded with some advice for organizations struggling with the proper use of tools: "It's critical that every organization avoid getting trapped in this innovator's dilemma. While a portion of the R&D portfolio needs to go to critical projects, and making existing tools better, a set portion always needs to be isolated for proactive analysis, piloting, and adoption of new tool capabilities. It's important that the teams owning core tools are also the teams that are innovating with new advances and new technologies. This will set an organization up to be leading, innovating, and cost-effectively solving problems, with the right tools being used to solve the right problems, and will make the company more successful in the long term."

Now let's hear from our other technology executive. Chris Lalonde was one of the cofounders of ObjectRocket, a DBaaS (database as a service, a specialized category of PaaS or platform as a service) offering MongoDB in the cloud, which was acquired by

Rackspace in 2013. He's currently the GM of data stores at Rackspace and has over 18 years of experience in a variety of senior technologist roles with companies including Bullhorn, Quigo, and eBay. Chris and his ObjectRocket team have been integral in the migration to nonrelational databases for many companies, and as such he has many stories of their successes and failures. One story regarding a customer that didn't understand the persistent data storage tools, their trade-offs, and when to use each of them is Chris's (least) favorite. Chris recalled, "We had a customer who was moving from a relational database to a nonrelational database and they had several challenges. One challenge was their rationalization of what could and couldn't be done inside MongoDB versus what could and couldn't be done inside of their previous database, Oracle. This was one of those cases where they really needed the right tool for the right job because they were looking for the flexibility of a nonrelational platform. What they didn't understand were the side effects and trade-offs of such a platform."

Chris provided us with an analogy: "When I'm talking about a relational database versus a nonrelational database, I'll use the example of a doctor's office. Doctors' offices work like a nonrelational database. You go in and you say, 'I'm here for my checkup,' and the receptionist goes in and pulls your health file from the filing cabinet and hands that to the doctor. As you go through your checkup, the doctor just adds things to the end of your file. When you're done, they take the file and they put it back in the cabinet. That's almost exactly how document stores like MongoDB work from a data perspective. Of course it's a bit simplified, but that's essentially how it works. If we take the same scenario and use a nonrelational database, the way it would work is you go into the office and tell the receptionist that you're here for your appointment, and the receptionist would go back to the first filing cabinet and pull out your name from an ID file. Then he'd go to the next filing cabinet and pull out your age from an age file, and then he'd go to the next filing cabinet and pull out your height, on and on. Then he'd take all those files together and hand them to the doctor and say, 'Here's your information.' The doctor would have to sort through them all, and if the doctor wanted to update it, he'd have to create a whole new filing cabinet. Some people just don't get that there are cases where having a whole bunch of filing cabinets is great, but then there are cases when it's not so great. Imagine if you have a lot of patients and a small office; pretty soon you have no space to see patients!

"So, this one customer," Chris continued, "had multiple terabytes worth of data sitting in a relational database and they pushed it all into a nonrelational database. The first thing that happened was that the data almost doubled in size. That was kind of a surprise for them. Next they began using some of the advantages of the nonrelational database such as the flexibility of changing data structures and the ease of identifying where relationships may exist through correlation within objects. But they still wanted to do some analytics that were both easy and fast in the relational database due to the support of the relational schema. These analytics were disadvantaged in response time within a nonrelational system. This customer went from a relational database where they were missing agility and flexibility but had the ability to make some of these very specific queries to a nonrelational world where they enjoyed flexibility and agility but then started missing that ability to query across data structures efficiently. This is a perfect example of why people need to understand what they are trying to get out of their data before they lock themselves into a specific tool. I often say that a carpenter doesn't build a house with just a hammer. The carpenter needs a saw and a screwdriver and other things. Engineers specifically, and maybe people in general, tend to fixate on using what they know. Going back to the doctor's office analogy, if you want to ask how many of the patients are males over six feet tall and you are using a nonrelational database, you've got to pull every file out of the whole thing and search through them one at a time. That's going to take a lot of effort no matter how efficient you are in sorting through all those files."

Chris tried to put this in context. "The reality of the world that we live in is that there are more technologies today than there have ever been, and data is growing at a faster rate than ever before. The advantages that those technologies can give businesses are massive, but not understanding the trade-offs can kill a business. The company that I spoke of was trying to take advantage of the agility of a nonrelational database; that's great, but they failed to understand that they had given up some of the analytical capability, such as multitable complex joins, that they used in a relational database. Understanding these differences is just good engineering; it can make the difference between companies that prosper and those that fail. Maybe what you're doing is good enough right now, but the problem is that people aren't thinking about scale—'What do I have to do to make this go 10x or 100x?' Why is that important? Well, I'll give you an example. AOL took five years to get to its first million customers; eBay took about three years to get to its first million customers; Instagram took nine weeks to get to its first million customers. The pace of growth has gotten so dramatically faster today that technologists are thrown into the storm of growth with little warning, and if those technologists have only one tool in their toolbox or if they don't know how to scale their data 1,000x, they probably won't have a job or a business for long."

Chris concluded, "There is no perfect database. There's no perfect data store. They all have trade-offs, and that's kind of the thing that everybody needs to wrap their head around. People have their biases for whatever reason, but the honest answer is that when you're writing or reading data from or to some kind of storage mechanism, there are a couple of core choices that you have to make that ultimately determine the characteristics of the database. One solution may take twice as much storage space, generally be a bit slower, but give you significantly greater flexibility. Conversely, another choice may give you less storage to worry about, be faster for many things, but constrain you in what you can do with it. Knowing these differences or having an expert to help you understand these trade-offs is critical for designing a modern application and really is a business advantage. It is kind of like understanding the difference in saws. A chainsaw, a jigsaw, and a hacksaw are all saws, but engineers made design decisions that help them cut the most efficiently through trees, lumber, or metal respectively. Can you cut down an oak tree with a hacksaw? Probably, but I wouldn't recommend it. Picking the right tool for the right job is really like that and honestly is probably more critical. If you need to cut down a tree, go for the chainsaw, that'd be my advice; but a lot of people only seem to have hacksaws."

Both James and Chris have different perspectives on using the right tools, one from an implementation perspective and the other from that of a service provider. However, both agree on the importance of using the right tools and the pitfalls of not doing so. Next we'll cover the rules about using databases, firewalls, and log files appropriately, but as you can imagine, this can extrapolate to any number of technologies, organizational structures, and processes. Don't get locked into only what you are familiar with; spend the time to learn new things and be open to them.

Rule 14—Use Databases Appropriately

Rule 14: What, When, How, and Why

What: Use relational databases when you need ACID properties to maintain relationships between your data and consistency. For other data storage needs consider more appropriate tools such as NoSQL DBMSs.

When to use: When you are introducing new data or data structures into the architecture of a system.

How to use: Consider the data volume, amount of storage, response time requirements, relationships, and other factors to choose the most appropriate storage tool. Consider how your data is structured and your products need to manage and manipulate data.

Why: An RDBMS provides great transactional integrity but is more difficult to scale, costs more, and has lower availability than many other storage options.

Key takeaways: Use the right storage tool for your data. Don't get lured into sticking everything in a relational database just because you are comfortable accessing data in a database.

Relational database management systems (RDBMSs), such as Oracle and MySQL, are based on the relational model introduced by Edgar F. Codd in his 1970 paper "A Relational Model of Data for Large Shared Data Banks."[1] Most RDBMSs provide two huge benefits for storing data. The first is the guarantee of transactional integrity through ACID properties; see Table 2.1 in Chapter 2, "Distribute Your Work," for definitions. The second is the relational structure within and between tables. To minimize data redundancy and improve transaction processing, the tables of most OLTP databases are normalized to third normal form, where all records of a table have the same fields, nonkey fields cannot be described by only one of the keys in a composite key, and all nonkey fields must be described by the key. Within the table each piece of data is highly related to other pieces of data. Between tables there are often relationships known as foreign keys. While these are two of the major benefits of using an RDBMS, these are also the reasons for their limitations in terms of scalability.

Because of this guarantee of ACID properties, an RDBMS can be more challenging to scale than other data stores. When you guarantee consistency of data and you have multiple nodes in your RDBMS cluster, such as with MySQL NDB, synchronous replication is used to guarantee that data is written to multiple nodes upon commitment.

With Oracle Real Application Clusters (RAC) there is a central database, but ownership of areas of the DB is shared among the nodes, so write requests have to transfer ownership to that node and reads have to hop from requestor to master to owner and back. Eventually you are limited by the number of nodes that data can be synchronously replicated to or by their physical geographical location.

The relational structure within and between tables in the RDBMS makes it difficult to split the database through such actions as sharding or partitioning. See Chapter 2 for rules related to distributing work across multiple machines. A simple query that joined two tables in a single database must be converted into two separate queries with the joining of the data taking place in the application to split tables into different databases.

The bottom line is that data that requires transactional integrity or relationships with other data is likely ideal for an RDBMS. Data that requires neither relationships with other data nor transactional integrity might be better suited for other storage systems. Let's talk briefly about a few of the alternative storage solutions and how they might be used in place of a database for some purposes to achieve better, more cost-effective, and more scalable results. The benefits will vary based on the type of NoSQL DBMS solution you implement. Each NoSQL database has varying characteristics, so you will want to choose the best for your data structure and product deployment needs.

One often overlooked storage system is a file system. Perhaps this is thought of as unsophisticated because most of us started programming by accessing data in files rather than databases. Once we graduated to storing and retrieving data from a database, we never looked back. File systems have come a long way, and many are specifically designed to handle very large amounts of files and data. Some of these include Google File System (GFS), MogileFS, and Ceph. File systems are great alternatives when you have a "write once, read many" system. Put another way, if you don't expect to have conflicting reads and writes over time on a structure or object and you don't need to maintain a large number of relationships, you don't really need the transactional overhead of a database; file systems are a great choice for this kind of work.

The next set of alternative storage strategies is termed NoSQL. Technologies that fall into this category are often subdivided into key-value stores, extensible record stores, and document stores. There is no universally agreed-upon classification of technologies, and many of them could accurately be placed in multiple categories. The label of NoSQL may lead some to believe that they are interchangeable. The differences, changing rapidly with open-source projects, will influence whether or not you are using your DBMS choices appropriately. We've included some example technologies in the following descriptions, but this is not to be considered gospel. Given the speed of development on many of these projects, the classifications are likely to become even more blurred in the future.

Key-value stores include technologies such as Memcached, Redis, and Amazon DynamoDB and Simple DB. These products have a single key-value index for data and that is stored in memory. Some have the capability to write to persistent storage or in memory such as the Amazon DBs. Some products in this subcategory use synchronous replication across nodes, and others are asynchronous. These offer significant scaling and performance by using a simplistic data store model, the key-value pair, but this is also a significant limitation in terms of what data can be stored. Additionally, the key-value stores that rely on synchronous replication still face the limitations that RDBMS clusters do, which are limits on the number of nodes and their geographical locations.

Extensible record stores (ERSs), sometimes called wide column stores or table-style DBMSs, include technologies such as Google's proprietary Bigtable and Facebook's, now open-source, Cassandra and the open-source HBase. Cassandra is often considered to be the most widely used, but the technology is based on ideas from Google's Bigtable, considered to be the origin of the ERS model. All of these products use a row and column data model that can be split across nodes. Rows are split or sharded on primary keys, and columns are broken into groups and placed on different nodes. This method of scaling is similar to the X and Y axes in the AKF Scale Cube, shown in Figure 2.1 in Chapter 2, where the X axis split is read replicas and the Y axis is separating the tables by services supported. In these products row sharding is done automatically, but column splitting requires user definitions, similar to how it is performed in an RDBMS. These products use an asynchronous replication providing eventual consistency. This means that eventually, which may take milliseconds or hours, the data will become consistent across all nodes.

Document stores include technologies such as MongoDB, CouchDB, Amazon's DynamoDB, and Couchbase. The data model used in this category is called a "document" but is more accurately described as a multi-indexed object model. The multi-indexed object (or "document") can be aggregated into collections of multi-indexed objects (typically called "domains"). These collections or "domains" in turn can be queried on many different attributes. Document store technologies do not support ACID properties; instead, they use asynchronous replication, providing an eventually consistent model.

NoSQL solutions limit the number of relationships between objects or entities to a few. It is this reduction of relationships that allows for the systems to be distributed across many nodes and achieve greater scalability while maintaining transactional integrity and read-write conflict resolution. As we have pointed out, this often comes at the cost of immediate consistency. You can tune consistency and latency in many of the NoSQL solutions with trade-offs, but immediate consistency is not possible as with an RDBMS. Strong consistency is possible and can be configured for all of your reads, but this will consume more resources and increased latency will occur as a typical GetItem will look across nodes for the latest update.

As is so often the case, and as you've probably determined while reading the preceding text, there is a trade-off between scalability and flexibility within these systems. The degree of relationship between data entities ultimately drives this trade-off; as relationships increase, flexibility also increases. This flexibility comes at an increase in cost and a decrease in the ability to easily scale the system. Figure 4.1 plots RDBMS, NoSQL, and file system solutions against both the costs (and limits) to scale the system and the degree to which relationships are used between data entities. Figure 4.2 plots flexibility against the degree of relationships allowed within the system. The result is clear: relationships engender flexibility but also create limits to our scale. As such, we do not want to overuse relational databases but rather choose a tool appropriate to the task at hand to engender greater scalability of our system.

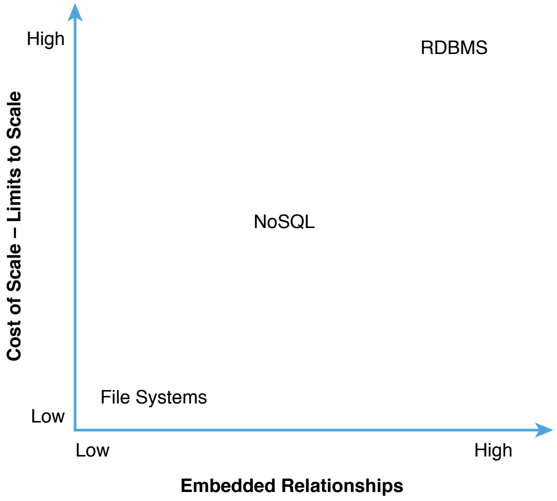


Figure 4.1 Cost and limits to scale versus relationships

Another data storage alternative that we are going to cover in this rule is Google's MapReduce.[2] At a high level, MapReduce has both a Map and a Reduce function. The Map function takes a key–value pair as input and produces an intermediate key–value pair. The input key might be the name of a document or a pointer to a piece of a document. The value could be content consisting of all the words within the document itself. This output is fed into a reducer function that uses a program that groups the

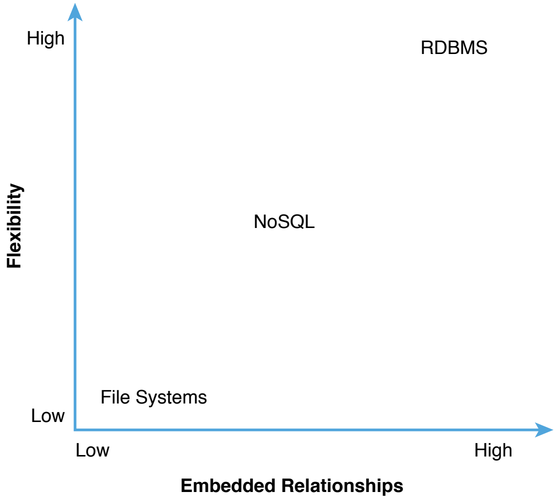


Figure 4.2 Flexibility versus relationships

words or parts and appends the values for each into a list. This is a rather trivial program that sorts and groups the functions by key. The huge benefit of this technology is the support of distributed computing of very large data sets across many servers.

An example technology that combines two of our data storage alternatives is Apache's Hadoop. This was inspired by Google's MapReduce and Google File System, both of which were mentioned previously. Hadoop provides the benefits of a highly scalable file system with distributed processing for storage and retrieval of the data.

So now that we've covered a few of the many options that might be preferable to a database when storing data, what data characteristics should you consider when making this decision? As with the myriad of options available for storage, there are numerous characteristics that should be considered. A few of the most important ones are the degree of relationships needed between elements, the rate of growth of the solution, and the ratio of reads to writes of the data (and potentially whether data is updated). Finally, we are interested in how well the data monetizes (that is, is it profitable?) because we don't want the cost of the system to exceed the value we expect to achieve from it.

The degree of relationships between data is important because it drives flexibility, cost, and time of development of a solution. As an example, imagine the difficulty of storing a transaction involving a user's profile, payment, purchase, and so on in a key-value store and then retrieving the information piecemeal such as through a report of purchased items. While you can certainly do this with a file system or NoSQL alternative, it may be costly to develop and time-consuming in delivering results back to a user.

The expected rate of growth is important for a number of reasons. Ultimately this rate impacts the cost of the system and the response times we would expect for some users. If a high degree of relationships is required between data entities, at some point we will run out of hardware and processing capacity to support a single integrated database, driving us to split the database into multiple instances.

Read and write ratios are important as they help drive an understanding of what kind of system we need. Data that is written once and read many times can easily be put on a file system coupled with some sort of application, file, or object cache. Images are great examples of systems that typically can be put on file systems. Data that is written and then updated, or with high write-to-read ratios, is better off within NoSQL or RDBMS solutions.

These considerations bring us to another cube, Figure 4.3, where we've plotted the three considerations against each other. Note that as the X, Y, and Z axes increase in value, the cost of the ultimate solution increases. Where we require a high degree of relationships between systems (upper right and back portion of Figure 4.3), rapid growth, and resolution of read and write conflicts, we are likely tied to several smaller RDBMS systems at relatively high cost for both our development and for the systems, maintenance, and possibly licenses for the databases. If growth and size are small but relationships remain high and we need to resolve read and write conflicts, we can use a single monolithic database (with high-availability clustering).

Relaxing relationships slightly allows us to use NoSQL alternatives at any level of reads and writes and with nearly any level of growth. Here again we see the degree to which relationships drive our cost and complexity, a topic we explore later in Chapter 8,

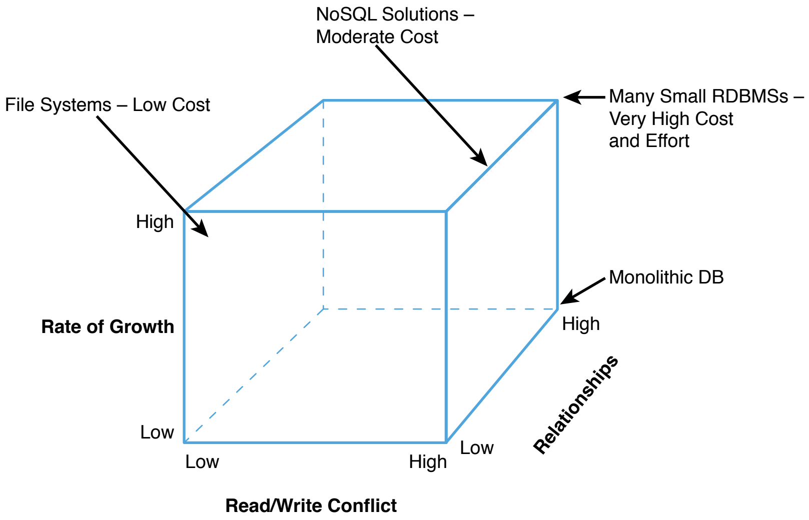


Figure 4.3 Solution decision cube

"Database Rules." Cost is lower for these NoSQL alternatives. Finally, where relationship needs are low and read-write conflict is not a concern, we can get into low-cost file systems to provide our solutions.

Monetization value of the data is critical to understand because as many struggling startups have experienced, storing terabytes of user data for free on class A storage is a quick way to run out of capital. A much better approach might be using tiers of data storage; as the data ages in terms of access date, continue to push it off to cheaper and slower-access storage media. We call this the *Cost-Value Data Dilemma*, which is where the value of data decreases over time and the cost of keeping it increases over time. We discuss this dilemma more in Rule 47 in Chapter 12, "Miscellaneous Rules," and describe how to solve the dilemma cost-effectively.

Rule 15—Firewalls, Firewalls Everywhere!

Rule 15: What, When, How, and Why

What: Use firewalls only when they significantly reduce risk, and recognize that they cause issues with scalability and availability.

When to use: Always.

How to use: Employ firewalls for critical PII, PCI (Payment Card Industry) compliance, and so on. Don't use them for low-value static content.

Why: Firewalls can lower availability and cause unnecessary scalability chokepoints.

Key takeaways: While firewalls are useful, they are often overused and represent both availability and scalability concerns if not designed and implemented properly.

The decision to employ security measures should ultimately be viewed through the lens of profit maximization. Security in general is an approach to reduce risk. Risk in turn is a function of both the probability that an action will happen and the impact or damage the action would cause should it happen. Firewalls help to manage risk in some cases by reducing the probability that an event will happen. They do so at some additional capital expense, some impact on availability (and hence either transaction revenue or customer satisfaction), and often an additional concern for scalability: the creation of a difficult-to-scale chokepoint in either network traffic or transaction volume. Unfortunately, far too many companies view firewalls as an all-or-nothing approach to security. They overuse firewalls and underuse other security approaches that would otherwise make them even more secure. We can't overstate the impact of firewalls on availability. In our experience, failed firewalls are the number-two driver of site downtime next to failed databases. As such, this rule is about reducing them in number. Remember, however, that there are many other things that you should be doing for security while you look to eliminate any firewalls that are unnecessary or simply burdensome.

In our practice, we view firewalls as perimeter security devices meant to increase both the perceived and actual cost of gaining entry to a product. In this regard, they serve a similar purpose to the locks you have on the doors to your house. In fact, we believe that the house analogy is appropriate to how one should view firewalls, so we'll build on that analogy here.

There are several areas of your house that you don't likely lock up; for example, you probably don't lock up your front yard. You probably also leave certain items of relatively low value in front of your house, such as hoses and gardening implements. You may also leave your vehicle outside even though you know it is more secure in your garage given how quickly most thieves can bypass vehicle security systems. More than likely you have locks and maybe deadbolts on your exterior doors and potentially smaller privacy locks on your bathrooms and bedrooms. Other rooms of your house, including your closets, probably don't have locks on them. Why the differences in our approaches?

Certain areas outside your house, while valuable to you, simply aren't of significant enough value for someone else to steal them. You really value your front yard but probably don't think someone's going to come with a shovel and dig it up to replant elsewhere. You might be concerned with someone riding a bicycle across it and destroying the grass or the sprinkler head, but that concern probably doesn't drive you to incur the additional cost of fencing it (other than a decorative picket fence) and destroying the view for both you and others in the neighborhood.

Your interior doors really have locks only for the purpose of privacy. Most interior doors don't have locks meant to keep out an interested and motivated intruder. We don't lock and deadbolt most of our interior doors because such locks present more of a hassle to us as occupants of the house, and the additional hassle isn't worth the additional security they provide.

Now consider your product. Several aspects, such as static images, CSS files, JavaScript, and so on, are important to you but don't really need high-end security. In many cases, you likely look to deliver these attributes via an edge cache or content delivery network (CDN) outside your network anyway (see Chapter 6, "Use Caching Aggressively"). As such, we shouldn't subject these objects to an additional network hop (the firewall), the associated lower overall availability, and the scale-limiting attributes of an additional network chokepoint. We can save some money and reduce the load on our firewalls simply by ensuring that these objects are delivered from private IP addresses and have access only via ports 80 and 443.

Returning to the value and costs of firewalls, let's explore a framework by which we might decide when and where to implement them. We've indicated that firewalls cost us in the following ways: there is a capital cost to purchase them, they create an additional item that needs to be scaled, and they represent an impact on our availability as a firewall is one more device on the critical route of any transaction that can fail and cause problems. We've also indicated that they add value when they are used to deter or hinder the efforts of those who would want to steal from us or harm our product. Table 4.1 shows a matrix indicating some of the key decision criteria for us in implementing firewalls.

The first thing that you might notice is that we've represented value to the bad guy and cost to firewall as having a near-inverse relationship. While this relationship won't always be true, in many of our clients' products it is the case. Static object references tend to represent a majority of object requests on a page and often are the heaviest elements of the pages. As such they tend to be costly to firewall given the transaction rate and throughput requirements. They are even more costly when you consider that they hold very little value to a potential bad guy. Given the high cost in terms of potential availability impact and capital relative to the likelihood that they are the focus of a bad guy's intentions, it makes little sense for us to invest in their protection. We'll just ensure that they are on private IP space (for example, 10.X.Y.Z addresses or the like) and that the only traffic that gets to them is requests for ports 80 and 443.

On the flip side, we have items like credit cards, bank account information, and Social Security numbers. These items have a high perceived value to our bad guy. They are also less costly to protect relative to other objects as they tend to be requested less frequently than many of our objects. We absolutely should lock these things away!

In the middle are all the other requests that we service within our platform. It probably doesn't make a lot of sense to ensure that every search a user performs goes through a firewall. What are we protecting? The actual servers themselves? We can protect our assets well

Table 4.1 **Firewall Implementation Matrix**

Value to "Bad Guy"	Cost to Firewall	Examples	Firewall Decision
Low	High	CSS, static images, JavaScript	No
Low	Medium	Product catalogs, search services	No
Medium	Medium	Critical business functions	Maybe
High	Low	Personally identifiable information (e.g., Social Security numbers, credit cards), password reset information	Yes

against attacks such as DDOS attacks with packet filters, routers, and carrier relationships. Other compromises can be thwarted by limiting the ports that access these systems. If there isn't a huge motivation for a bad guy to go after the services, let's not spend a lot of money and decrease our availability by pretending that they are the crown jewels.

In cases where you must use firewalls, we recommend isolating them by swim lane if your budget allows. As we have discussed, firewalls and load balancers are common failure points but are expensive. Ideally, each swim lane will have its own firewalls and load balancers to minimize the impact of failure, which we'll address further in Rule 36 (Chapter 9, "Design for Fault Tolerance and Graceful Failure").

Many companies are now implementing SDNs (software-defined networks), which decouple control and management of networking devices, like firewalls, from the hardware itself. This approach centralizes administration of firewall rules, making it simpler to dynamically manage which traffic must pass through. In conjunction with NVF (network function virtualization), you can build a firewall solution using commodity hardware that can be clustered, ideally having a positive impact on availability. However, it is important to keep in mind that any extra hop reduces availability and increases latency regardless of implementation.

In summation, don't assume that everything deserves the same level of protection. The decision to employ firewalls is a business decision focused on decreasing risk at the cost of decreasing availability and increasing capital costs. Too many companies view firewalls as a unary decision—if something exists within our site, it must be firewalled—when in fact firewalls are just one of many tools you might employ to help decrease your risk. Not everything in your product is likely deserving of the cost and impact on availability that a firewall represents. As with any other business decision, this one should be considered in the light of these trade-offs, rather than just assuming a cookie-cutter approach to your implementation. Given the nature of firewalls, they can easily become the biggest bottleneck in your product from a scale perspective.

Rule 16—Actively Use Log Files

Rule 16: What, When, How, and Why

What: Use your application's log files to diagnose and prevent problems.

When to use: Put a process in place that monitors log files and forces people to take action on issues identified.

How to use: Use any number of monitoring tools from custom scripts to Splunk or the ELK framework to watch your application logs for errors. Export these and assign resources to identify and solve the issue.

Why: The log files are excellent sources of information about how your application is performing for your users; don't throw this resource away without using it.

Key takeaways: Make good use of your log files, and you will have fewer production issues with your system. When issues do occur, you will be able to address them more quickly.

In the spirit of using the right tools for the job, one of the tools that is likely in all our toolboxes but often gets overlooked is log files. Fortunately new tools are emerging that allow teams to consume and monitor log files in new ways and without much effort. Unless you've purposely turned off logging on your Web or application servers, almost all varieties come with error and access logs. Apache has error and access logs, Tomcat has java.util.logging or Log4j logs, and WebSphere has SystemErr and SystemOut logs. These logs can be incredibly valuable tools for identifying and troubleshooting problems when an incident occurs. You can also get ahead of incidents before they happen by using log files to proactively debug your application. Logs provide powerful insights into the performance and errors occurring within your application that might prevent it from scaling. To best use this tool there are a few simple but important steps to follow.

The first step in using log files is to aggregate them. As you probably have dozens or perhaps even hundreds of servers, you need to pull this data together to use it. If the amount of data is too large to pull together, there are strategies such as sampling, pulling data from every nth server, that can be implemented. Another strategy is to aggregate the logs from a few servers onto a log server that can then transmit the semi-aggregated logs into the final aggregation location. As shown in Figure 4.4, dedicated log servers can aggregate the log data to then be sent to a data store. This aggregation is generally done through an out-of-band network that is not the same network used for production traffic. What we want to avoid is impacting production traffic from logging, monitoring, or aggregating data. Advances in distributed compute and data storage allow you to consume and perform processing on large data files like logs in a cost-effective manner using an environment that is isolated from your application.

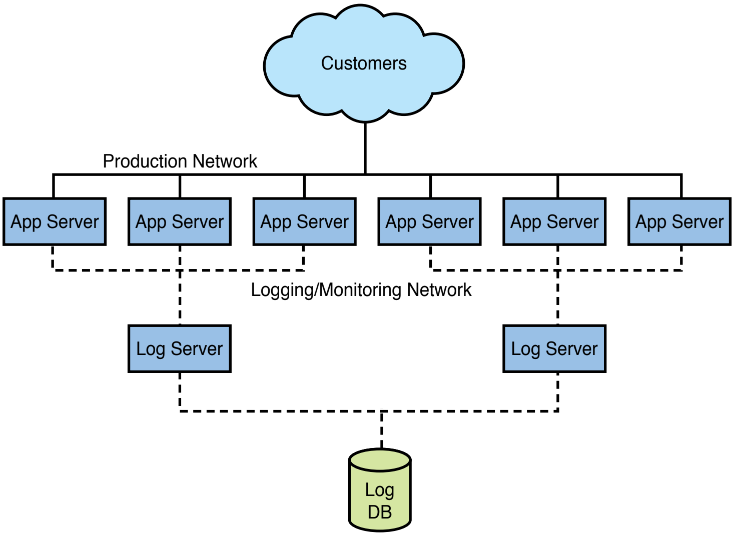


Figure 4.4 Log aggregation

The next step is to monitor these logs. Surprisingly many companies spend the time and computational resources to log and aggregate but then ignore the data. While you can just use log files during incidents to help restore service, this isn't optimal. A preferred use is to monitor these files with automated tools. This monitoring can be done through custom scripts such as a simple shell script that greps the files, counting errors and alerting when a threshold is exceeded. More sophisticated tools such as Cricket or Cacti include graphing capabilities. A tool that combines the aggregation and monitoring of log files is Splunk.

Tools like Splunk and frameworks like ELK (Elasticsearch, Logstash, Kibana) are solutions that combine the aggregation and monitoring of log files. The ELK framework is a set of separate open-source solutions that can work synchronously together to provide a comprehensive monitoring stack that is easily integrated into your systems environment. You begin by implementing Logstash, an application similar to Extract, Transform, and Load (ETL) that extracts data from a variety of inputs (i.e., log files), performs transformations, and can then push the transformed data into Elasticsearch. Elasticsearch is a distributed data store and search engine that indexes large amounts of data to make it quickly and easily searchable. Doing so enables you to monitor metrics like mean response time of a service, max and min values over a given time period, and counts of error types. The final component, Kibana, is a data visualization tool that displays your metrics in graphical and tabular presentation modes as they are being generated in real time. Through Kibana, you can quickly see if metrics are falling out of bounds of the expected results through use of statistical process controls (see Rule 49, "Design Your Application to Be Monitored," in Chapter 12) and set up alerts that notify you when they do. This output could help you identify issues even before you see them propagate into your business output metrics, allowing you to get ahead of an incident management situation.

In addition to helping you during an incident, log files are a vital resource for debugging errors in your application. Once you've aggregated the logs and monitored them for errors, the final step is to take action to fix the problems. This requires assigning engineering and QA resources to identify common errors as being related to individual problems. It is often the case that one bug in an application flow can result in many different error manifestations. The same engineers who identified the bug might also be assigned to fix it, or other engineers might be assigned the task.

We'd like to have the log files completely free of errors, but we know that's not always possible. While it's not uncommon to have some errors in application log files, you should establish a process that doesn't allow them to get out of control or ignored. Some teams periodically, every third or fourth release, clean up all miscellaneous errors that don't require immediate action. These errors might be something as simple as missing redirect configurations or not handling known error conditions in the application.

We must also remember that logging comes at some cost. Not only is there a cost in keeping additional data, but very often there is a cost in terms of transaction response times. We can help mitigate the former by summarizing logs over time and archiving and purging them as their value decreases (see Rule 47). We can help minimize the latter by logging in an asynchronous fashion. Ultimately we must pay attention to our costs for logging and make a cost-effective decision about both how much to log and how much data to keep.

We hope we've convinced you that log files are an important tool in your arsenal for monitoring and incident management, as well as debugging errors in your application. By simply using a tool that you likely already have, you can greatly improve your customer experience and scalability of your application.

Summary

Using the right tool for the job is important in any discipline. Just as you wouldn't want your plumber to bring only a hammer into your house to fix your pipes, your customers and investors don't want you to bring a single tool to solve problems with diverse characteristics and requirements. Avoid falling prey to Maslow's Hammer and bring together diverse teams capable of thinking of different solutions to problems. A final word of caution on this topic is that each new technology introduced requires another skill set to support. While the right tool for the job is important, don't overspecialize to the point that you have no depth of skills to support your systems.

Notes

1. Edgar F. Codd, "A Relational Model of Data for Large Shared Data Banks," 1970, www.seas.upenn.edu/~zives/03f/cis550/codd.pdf.
2. Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Google Research Publications, http://research.google.com/archive/mapreduce.html.

5

Get Out of Your Own Way

You might not have heard of Confinity, but you probably know their hallmark product, PayPal. Today the name is synonymous with sending money all over the world via the Internet, but the application was originally designed to allow individuals to "beam" sums of money between handheld devices such as Palm Pilots face-to-face.[1] Confinity was founded by Max Levchin, Peter Thiel, and Luke Nosek in December 1998. At the company's launch party, venture capital firm Nokia Ventures used the software to beam $3 million to Confinity CEO Thiel's Palm Pilot in about 5 seconds. Thiel is quoted as saying, "Most transactions take place between people in the real world, away from the desktop."[2] While this was true in the late 1990s, since 2000 three-quarters of retail sales growth has occurred through online channels. Excluding cars, gasoline, and groceries, online sales account for 16% of retail and continue to expand at an annual pace of about 15%.[3]

In March 2000, Confinity merged with the online banking company X.com, founded by Elon Musk, and after a corporate restructuring the company adopted the name PayPal, Inc. While the first version of the software was designed to transfer money between people face-to-face, the PayPal team quickly realized the enormous growth of online commerce. They scrapped the mobile service offering and concentrated on online payments. However, one concept required in face-to-face transactions, but not when payments are sent between people in different locations, is that of simultaneous reconciliation.

As you might imagine, in the early days of the Internet people were hesitant to trust online payments. If someone standing in front of you "beamed" you money but it didn't immediately show up on your device, you might get a little concerned about the veracity of the payment system. However, two people conducting a transaction on eBay or another e-commerce site were rarely (probably never) in front of their home computers and at the same time on the phone with each other waiting for a payment to go through. Additionally, banks have used the notion of a pending transaction for decades. Online banking has allowed us to see these pending transactions as they occur and later as they clear.

From the authors' time running PayPal engineering and architecture and our time since as users of the product, PayPal never let go of the concept of the synchronous transfer of money from one account to another. In the database that stored account information, this required that the two balance adjustments of the payer and payee accounts occur within the same transaction. Whether bound by the constraints and context of the original company, or simply a result of our own shortsightedness, it's something we wish we could go back in time and change. This constraint caused no end of scaling issues that could have been simply solved by placing accounts in a number of smaller databases and using the pending transaction while processes ensured that money was moved accurately from one account to another. The cost of scaling the solution would have gone down over time (and paid for itself multifold), and the overall availability of the product would have gone up significantly.

Jim Gray, a computer scientist who received the Turing Award, is credited with coming up with the four properties that define a reliable transaction system, aka relational database. These became the acronym that many of us know today as ACID—atomicity, consistency, isolation, and durability. Relational databases ensure atomicity whereby in a transaction involving two or more discrete pieces of information, either all of the pieces are committed or none are committed. In PayPal's scenario, both the credit to the payee's account and the debit from the payer's accounts had to occur together or the transaction failed and neither account was modified. We often refer to this type of restriction as a temporal constraint because the logic constrains that our payee and payer account changes occur simultaneously rather than asynchronously. This constraint is often associated with one of the other ACID properties: consistency. The concept of consistency requires that a transaction either create a new and valid state of data or return all data to its original state before the transaction began. When operating in a single relational database, consistency is easily achieved because the database engine ensures this. In the early 2000s PayPal was experiencing exponential growth. In one quarter of 2001, transactional revenues totaled $13.2 million, which was more than in the entire previous year.[4] This fantastic growth rate continued for years. Even into 2006 the year-over-year growth rate of total payment volume (TPV) was up over 36%.[5] Maintaining all of these transactions on a single database instance became a scaling nightmare.

The PayPal engineering and architecture teams worked tirelessly for years ensuring that the single transaction database could continue to function. They tried a myriad of solutions, including the dreaded two-phase commit (2PC) that resulted in the unmitigated 24.0 disaster (see Chapter 8, "Database Rules"), ultimately relying on Y axis splits (see Rule 8 in Chapter 2, "Distribute Your Work") to pull nontransactional services off of the primary database. The enormous number of developer and architect hours, days, months, and years that have been spent on this scale challenge would probably be frightening to calculate. All of this work, outages, and scaling problems because the business refused to relax the temporal constraint that was ingrained in its DNA from the first product release. Would PayPal's customers have cared? While it is possible a few individuals might have noticed and of course complained, it's very likely that this would not have had a noticeable impact on growth, revenue, or TPV. To answer this concretely would have required only a simple experiment where transactions were artificially marked as pending for some time period. A simple sharding solution by customer account (Z axis split; see Rule 9 in Chapter 2) along with a pending transaction likely would have enabled PayPal's persistent storage tier to scale very simply and very cost-effectively. This is a prime example of how we sometimes put scaling constraints on ourselves without truly understanding the impact or nonimpact on our customers. Again, if only we could go back in time and correct the mistakes and lessons from which we teach . . .

If we started off our consulting engagements with the comment "You need to get out of your own way in order to scale," our clients would undoubtedly argue that they are NOT the roadblock to scaling their system. As our experiences at PayPal demonstrate, this unfortunately is often the case. In many cases our clients are hampering their own scale because of decisions they have made in the past. In fact, not making the necessary changes to fix these constraints is essentially making the same decision every day. If you disagree with that statement, consider the financial tool of sunk cost analysis. This is a methodology that investors use to determine if they should stay in a particular investment despite what they've already invested and potentially lost in that investment. Economists argue that rational investors should not let previous investments (sunk costs) influence their decision to remain in the investment. Those investors who do are not assessing that decision exclusively on its own merits. In this chapter, we're going to cover three rules about decisions you might have made with regard to how your system works. If you've constrained your system because of decisions you've made regarding state, redirects, or data consistency, consider making different decisions and changing your system.

Rule 17—Don't Check Your Work

> ### Rule 17: What, When, How, and Why
>
> **What:** Avoid checking and rechecking the work you just performed or immediately reading objects you just wrote within your products.
>
> **When to use:** Always (see rule conflict in the following explanation).
>
> **How to use:** Never read what you just wrote for the purpose of validation. Store data in a local or distributed cache if it is required for operations in the near future.
>
> **Why:** The cost of validating your work is high relative to the unlikely cost of failure. Such activities run counter to cost-effective scaling.
>
> **Key takeaways:** Never justify reading something you just wrote for the purpose of validating the data. Trust your persistence tier to notify you of failures, and read and act upon errors associated with the write activity. Avoid other types of reads of recently written data by storing that data locally.

Carpenters and woodworkers have an expression: "Measure twice and cut once." You might have learned such a phrase from a high school wood shop teacher—one who might have been missing a finger. Missing digits aside, the logic behind such a statement is sound and based on experience through practice. It's much better to validate a measurement before making a cut, as a failed measurement will potentially increase production waste by creating a useless board of the wrong size. We won't argue with such a plan. Instead, we aim to eliminate waste of a different kind: the writing and subsequent immediate validation of the just-written data, analogous to a post-cut measurement in carpentry.

We've been surprised over the last several years at how often we find ourselves asking our clients, "What do you mean you are reading and validating something that you just wrote?" Sometimes clients have a well-thought-out reason for their actions, though we have yet to see one with which we agree. More often than not, the client cops a look that reminds us of a child who just got caught doing something he or she shouldn't be doing. The claims of those with well-thought-out (albeit in our opinion value-destroying) answers are that their application requires an absolute guarantee that the data not only be written but also be written correctly. Keep in mind that most of our clients have SaaS or commerce platforms; they aren't running nuclear power facilities, sending people into space, controlling thousands of passenger-laden planes in flight, or curing cancer.

Fear of failed writes and calculations has long driven extra effort on the part of many a developer. This fear, perhaps justified in the dark ages of computing, was at least partially responsible for the fault-tolerant computer designs developed by both Tandem and Stratus in the late 1970s and early 1980s, respectively. The primary driver of these systems was to increase mean time to failure (MTTF) within systems through "redundant everything," including CPUs, storage, memory, memory paths, storage paths, and so on. Some models of these computers necessarily compared results of computations and storage operations along parallel paths to validate that the systems were working properly. One of the authors of this book developed applications for an aging Stratus minicomputer, and in the two years he worked with it, the system never identified a failure in computation between the two processors, or failure writes to memory or disk.

Today those fears are much less well founded than they were in the late 1970s through the late 1980s. In fact, when we ask our clients who first write something and then attempt to immediately read it how often they find failures, the answer is fairly consistent: "Never." And the chances are that unless they fail to act upon an error returned from a write operation, they will never experience such an event. Sure, corruption happens from time to time, but in most cases that corruption is identified during the actual write operation. Writing and then reading a result doubles the transactions on your systems and as a result halves the number of total transactions you may perform to create value. This in turn decreases margins and profitability. A better solution is to simply read the return value of the operation you are performing and trust that it is correct, thereby increasing the number of value-added transactions you can perform. As a side note here, the most appropriate protection against corruption is to properly implement high availability and have multiple copies of data around such as a standby database or replicated storage (see Chapter 9, "Design for Fault Tolerance and Graceful Failure"). Ideally you will ultimately implement multiple live sites (see Chapter 3, "Design to Scale Out Horizontally," Rule 12).

Of course not every "write then immediately read" activity is a result of an overzealous engineer attempting to validate what he or she has just written. Sometimes it's the result of end users immediately requesting the thing they just wrote. The question we ask here is why these clients (and by that we mean applications and/or browsers) don't store frequently used (including written) data locally. If you just wrote something and you know you are likely to need it again, just keep it around locally. One common example of such a need is during a registration flow for most products. Typically there is a stage at which one wants to present to the user the data you are about to commit to the permanent registration “record.” Another one might be the purchase flow embedded within most shopping cart systems on commerce sites. Regardless of the case, if the information you are writing is going to be needed in the near future, it makes sense to keep it around, to cache it. See Chapter 6, “Use Caching Aggressively,” for more information on how and what to cache. One nifty trick here in providing users with data they may immediately need is to simply write said data to the screen of the client (again an application or browser) directly rather than requesting the data again. Or pass said data through the URI and use it in subsequent pages.

The point to which all the preceding paragraphs are leading is that doubling your activity reduces your ability to scale cost-effectively. In fact, it doubles your cost for those transactions. So while you may be engineering a solution to avoid a couple of million in risk associated with failed writes, you may be incurring tens of millions of dollars in extra infrastructure to accomplish it. Rarely, and in our experience never, does this investment in engineering time and infrastructure overcome the risk it mitigates. Reading after writing is bad in most cases because it not only doubles your cost and limits your scalability, it rarely returns value in risk mitigation commensurate with the costs. There are no doubt cases where it is warranted, though those are far fewer in number than is justified by many technology teams and businesses.

The observant reader may have identified a conflict in our rules. Storing information locally on a system might be indicative of state and certainly requires affinity to the server to be effective. As such, we’ve violated Rule 40 (see Chapter 10, “Avoid or Distribute State”). At a high level, we agree, and if forced to make a choice we would always develop a stateless application over ensuring that we don’t have to read what we just wrote. That said, our rules are meant to be nomothetic or “generally true” rather than idiographic or “specifically true.” You should absolutely try not to duplicate your work and absolutely try to maintain a largely stateless application. Are these two statements sometimes in conflict? Yes. Is that conflict resolvable? Absolutely!

The way we resolve such a conflict in rules is to take the 30,000-foot approach. We want a system that does not waste resources (like reading what we just wrote) while we attempt to be largely stateless for reasons we discuss in Chapter 10. To do this, we decide to never read for the singular purpose of validation. We also agree that there are times when we might desire affinity for speed and scale versus reading what we just wrote. This means maintaining some notion of state, but we limit these to transactions where it is necessary for us to read something that we just wrote. While this approach causes a violation of our state rules, it makes complete sense as we are attempting to introduce state in a limited set of operations where it actually decreases cost and increases scalability as opposed to how it often does just the opposite.

As with any rule, there are likely exceptions. What if you exist in a regulatory environment that requires absolutely 100% of all writes of a particular piece of data be verified to exist, be encrypted, and be backed up? We’re not certain such an environment exists, but if it did there are almost always ways to meet requirements such as these without blocking for an immediate read of data that was just written. Here is a bulleted checklist of questions you can answer and steps you can take to eliminate reading what you just wrote and blocking the user transaction to do so:

- **Regulatory/legal requirement**—Is this activity a regulatory or legal requirement? If it is, are you certain that you have read it properly? Rarely does a requirement spell out that you need to do something "in line" with a user transaction. And even if it does, the requirement rarely (probably never) applies to absolutely everything that you do.
- **Competitive differentiation**—Does this activity provide competitive differentiation? Careful—"Yes" is an all-too-common and often incorrect answer to this question. Given the low rate of failures you would expect, it is hard to believe that you will win by correctly handling the .001% of failures that your competitors will have by not checking twice. Put that on your marketing literature and see how much traction it brings.
- **Asynchronous completion**—If you have to read after writing for the purpose of validation due to either a regulatory requirement (doubtful but possible) or competitive differentiation (beyond doubtful; see the preceding point), consider doing it asynchronously. Write locally and do not block the transaction. Handle any failures to process by re-creating the data from logs, reapplying it from a processing queue, or worst case asking the user for it again in the very small percentage of cases where you lose it. If the failure is in copying the data to a remote backup for high availability, simply reapply that record or transaction. Never block the user under any scenario pending a synchronous write to two data sources.
- **Fool the user**—If the data simply needs to be displayed to the user in the same or subsequent screen, use the nifty trick of writing it to the screen directly or passing information through the URI. This eliminates the subsequent database read and gives the user the desired effect at much lower cost.

Rule 18—Stop Redirecting Traffic

Rule 18: What, When, How, and Why

What: Avoid redirects when possible; use the right method when they are necessary.

When to use: Always.

How to use: If you must use redirects, consider server configurations instead of HTML or other code-based solutions.

Why: Redirects in general delay the user, consume computation resources, are prone to errors, and can negatively affect search engine rankings.

Key takeaways: Use redirects correctly and only when necessary.

There are many reasons that you might want to redirect traffic. A few of these include tracking clicks on content or an advertisement, misspelled domains (for example, a*fk*partners.com instead of a*kf*partners.com), aliasing or shortening URLs (for example, akfpartners.com/news instead of akfpartners.com/news/index.php), or changing domains (for example, moving the site from akf-consulting.com to akfpartners.com). There is even a design pattern called Post/Redirect/Get (PRG) that is used to avoid some duplicated form submissions. Essentially this pattern calls for the POST operation on a form submission to redirect the browser, preferably with an HTTP 303 response. All these and more are valid reasons for redirecting users from one place to another. However, like any good tool redirection can be used improperly, like trying to use a screwdriver instead of a hammer, or too frequently, such as splitting a cord of wood without sharpening your ax. Either problem ends up with less-than-desirable results. Let's first talk a little more about redirection according to the HTTP standard.

According to RFC 2616, Hypertext Transfer Protocol,[6] there are several redirect codes, including the more familiar "301 Moved Permanently" and the "302 Found for Temporary Redirection." These codes fall under the Redirection 3xx heading and refer to a class of status code that requires further action to be taken by the user agent to fulfill the request. The complete list of 3xx codes is provided in the sidebar "HTTP 3xx Status Codes."

HTTP 3xx Status Codes

- **300 Multiple Choices**—The requested resource corresponds to any one of many representations and is being provided so that the user can select a preferred representation.
- **301 Moved Permanently**—The requested resource has been assigned a new permanent URI, and any future references to this resource *should* use the URI returned.
- **302 Found**—The requested resource resides temporarily under a different URI, but the client *should* continue to use the Request-URI for future requests.
- **303 See Other**—The response to the request can be found under a different URI and *should* be retrieved using a GET method. This method exists primarily for the PRG design pattern to allow the output of a POST to redirect the user agent.
- **304 Not Modified**—If the client has performed a conditional GET request and access is allowed, but the document has not been modified, the server *should* respond with this status code.
- **305 Use Proxy**—The requested resource *must* be accessed through the proxy given by the Location field.
- **306 (Unused)**—This status code is no longer used in the specification.
- **307 Temporary Redirect**—The requested resource resides temporarily under a different URI.

So, we've agreed that there are many valid reasons for using redirects, and the HTTP standard even has multiple status codes that allow for various types of redirects. What then is the problem with redirects? The problem is that they can be performed in numerous ways, some better than others in terms of resource utilization and performance, and they can easily get out of hand. Let's examine a few of the most popular methods of redirecting users from one URI to another and discuss the pros and cons of each.

The simplest way to redirect users from one page or domain to another is to construct an HTML page that requests they click on a link to proceed to the real resources they are attempting to retrieve. The page might look something like this:

```
<html><head></head><body>
[em]<p>Please click <a href="http://www.akfpartners.com/
techblog">here for your requested page</a></p>
</body></html>
```

The biggest problem with this method is that it requires users to click again to retrieve the real page they were after. A slightly better way to redirect with HTML is to use the meta tag "refresh" and automatically send the user's browser to the new page. The HTML code for that would look like this:

```
<html><head>
[em]<meta http-equiv="Refresh" content="0;
url=http://www.akfpartners.com/techblog" />
</head><body>
[em]<p>In case your page doesn't automatically refresh, click
<a href="http://www.akfpartners.com/techblog">here for your
requested page</a></p>
</body></html>
```

With this we solved the user interaction problem, but we're still wasting resources by requiring our Web server to receive a request and respond with a page back to the browser that must parse the HTML code before the redirection. Another more sophisticated method of handling redirects is through code. Almost all languages allow for redirects; in PHP the code might look like this:

```
<?
Header( "HTTP/1.1 301 Moved Permanently" );
Header( "Location: http://www.akfpartners.com/techblog" );
?>
```

This code has the benefit of not requiring the browser to parse HTML but rather redirect through an HTTP status code in a header field. In HTTP, header fields contain the operating parameters of a request or response by defining various characteristics of the data transfer. The preceding PHP code results in the following response:

```
HTTP/1.1 301 Moved Permanently
Date: Mon, 11 Oct 2010 19:39:39 GMT
Server: Apache/2.2.9 (Fedora)
X-Powered-By: PHP/5.2.6
Location: http://www.akfpartners.com/techblog
Cache-Control: max-age=3600
Expires: Mon, 11 Oct 2010 20:39:39 GMT
Vary: Accept-Encoding,User-Agent
Content-Type: text/html; charset=UTF-8
```

We've now improved our redirection by using HTTP status codes in the header fields, but we're still requiring our server to interpret the PHP script. Instead of redirecting in code, which requires either interpretation or execution, we can request the server to redirect for us with its own embedded module. In the Apache Web server, two primary modules are used for redirecting, `mod_alias` and `mod_rewrite`. The `mod_alias` is the easiest to understand and implement but is not terribly sophisticated in what it can accomplish. This module can implement `alias`, `aliasmatch`, `redirect`, or `redirectmatch` commands. Following is an example of a `mod_alias` entry:

```
Alias /image /www/html/image
Redirect /service http://foo2.akfpartners.com/service
```

The `mod_rewrite` module compared to the `mod_alias` module is sophisticated. According to Apache's own documentation, this module is a "killer one"[7] because it provides a powerful way to manipulate URLs, but the price you pay is increased complexity. An example rewrite entry for redirecting all requests for *artofscale.com* or *www.artofscale.com* URLs to *theartofscalability.com* permanently (301 status code) follows:

```
RewriteEngine on
RewriteCond %{HTTP_HOST} ^artofscale.com$ [OR]
RewriteCond %{HTTP_HOST} ^www.artofscale.com$
RewriteRule ^/?(.*)$
"http\:\/\/theartofscalability\.com\/$1" [R=301,L]
```

To add to the complexity, Apache allows the scripts for these modules to be placed in either the .htaccess files or the httpd.conf main configuration file. However, using the .htaccess files should be avoided in favor of the main configuration files primarily because of performance.[8] When configured to allow the use of .htaccess files, Apache looks in every directory for .htaccess files, thus causing a performance hit, whether you use them or not! Also, the .htaccess file is loaded every time a document is requested instead of once at startup, like the httpd.conf main configuration file.

We've now seen some pros and cons of redirecting through different methods, which we hope will guide us in how to use redirection as a tool. The last topic to cover is making sure we're using the right tool in the first place. Ideally we want to avoid redirection completely. A few of the reasons to avoid redirection when possible are that it always delays users from getting the resource they want, it takes up computational resources, and there are many ways to break redirection, hurting user browsing or search engine rankings.

A few examples of ways that redirects can be wrong come directly from Google's page on why URLs are not followed by its search engine bots.[9] These include redirect errors, redirect loops, too-long URLs, and empty redirects. You might think that creating a redirect loop would be difficult, but it is much easier than you think, and while most browsers and bots stop when they detect the loop, trying to service those requests takes up a ton of resources, not to mention creating a miserable user experience.

As we mentioned in the beginning of this rule, there are certainly times when redirection is necessary, but with a little thought there are ways around many of these. Take click tracking, for example. There are certainly all types of business needs to keep track of clicks, but there might be a better way than sending the user to a server to record the click in an access log or application log and then sending the user to the desired site. One alternative is in the browser to use the onClick event handler to call a JavaScript function. This function can request a 1x1 pixel through a PHP or other script that records the click. The beauty of this solution is that it doesn't require the user's browser to request a page or receive back a page or even a header before it can start loading the desired page.

When it comes to redirects, make sure you first think through ways that you can avoid them. Using the right tool for the job as discussed in Chapter 4, "Use the Right Tools," is important, and redirects are specialized tools. Once those options fail, consider how best to use the redirect tool. We covered several methods and discussed their pros and cons. The specifics of your application will dictate the best alternative.

Rule 19—Relax Temporal Constraints

> ### Rule 19: What, When, How, and Why
>
> **What:** Alleviate temporal constraints in your system whenever possible.
>
> **When to use:** Anytime you are considering adding a constraint that an item or object must maintain a certain state between a user's actions.
>
> **How to use:** Relax constraints in the business rules.
>
> **Why:** The difficulty in scaling systems with temporal constraints is significant because of the ACID properties of most RDBMSs.
>
> **Key takeaways:** Carefully consider the need for constraints such as items being available from the time a user views them until the user purchases them. Some possible edge cases where users are disappointed are much easier to compensate for than not being able to scale.

In the domains of mathematics and machine learning (artificial intelligence) there is a set of constraint satisfaction problems (CSPs) where the state of a set of objects must satisfy certain constraints. CSPs are often highly complex, requiring a combination of heuristics and combinatorial search methods to be solved.[10] Two classic puzzles that can be modeled as CSPs are Sudoku and the map coloring problem. The goal of Sudoku is to fill each nine-square row, each nine-square column, and each nine-square box with the numbers 1 through 9, with each number used once and only once in each section. The goal of a map coloring problem is to color a map so that regions sharing a common border have different colors. Solving this involves representing the map as a graph where each region is a vertex and an edge connects two vertices if the corresponding regions share a border.

A more specific variety of the CSP is a temporal constraint satisfaction problem (TCSP), which is a representation where variables denote events, and constraints represent the possible temporal relations between them. The goals are ensuring consistency among the variables and determining scenarios that satisfy all constraints. Enforcing what is known as local consistency on the variables ensures that the constraints are satisfied for all nodes, arcs, and paths within the problem. While many problems within machine learning and computer science can be modeled as TCSPs, including machine vision, scheduling, and floor plan design, use cases within SaaS systems can also be thought of as TCSPs.

An example of a temporal constraint within a typical SaaS application would be purchasing an item in stock. There are time lapses between a user viewing an item, putting it in the shopping cart, and purchasing it. One could argue that for the absolute best user experience, the state of the object, specifically whether or not the object is available, would remain consistent throughout this process. To do so would require that the application mark the item as "taken" in the database until the user browses off the page, abandons the cart, or makes the purchase.

This is pretty straightforward until we get a lot of users on our site. It's not uncommon for users to view 100 or more items before they add anything to their cart. One of our clients claims that users look at more than 500 search results before adding a single item to their cart (no, this wasn't a lingerie site). In this case our application probably needs several read replicas of the database to allow many more people to search and view items than to purchase them. Herein lies the problem; most RDBMSs aren't good at keeping all the data completely consistent between nodes. Even though read replicas or slave databases can be kept within seconds of each other in terms of consistent data, certainly there will be edge cases when two users want to view the last available inventory of a particular item. We'll come back and solve this problem, but first let's talk about why databases make this difficult.

In Chapter 2 and Chapter 4 we spoke about ACID properties of RDBMSs (refer to Table 2.1). The one property that makes scaling an RDBMS in a distributed manner difficult is consistency. The CAP Theorem, also known as the Brewer Theorem, so named after computer scientist Eric Brewer, states that three core requirements exist when designing applications in a distributed environment, but it is impossible to simultaneously satisfy all three requirements. These requirements are expressed in the acronym CAP:

- **Consistency**—The client perceives that a set of operations has occurred all at once.
- **Availability**—Every operation must terminate in an intended response.
- **Partition tolerance**—Operations will complete, even if individual components are unavailable.

What has been derived as a solution to this problem is called BASE, an acronym for architectures that solve CAP and stands for "basically available, soft state, and eventually consistent." By relaxing the ACID properties of consistency we have greater flexibility in how we scale. A BASE architecture allows for the databases to become consistent, eventually. This might be minutes or even just seconds, but as we saw in the previous example, even milliseconds of inconsistency can cause problems if our application expects to be able to "lock" the data.

The way we would redesign our system to accommodate this eventual consistency would be to relax the temporal constraint. The user just viewing an item would not guarantee that it is available. The application would "lock" the data when the item is placed into a shopping cart, and this would be done on the primary write copy or master database. Because we have ACID properties, we can guarantee that if our transaction completes and we mark the record of the item as "locked," then that user can continue through the purchase confident that the item is reserved. Other users viewing the item may or may not have it available for them to purchase.

Another area in which temporal constraints are commonly found in applications is the transfer of items (money) or communications between users. The PayPal example with which we opened this chapter is a great example of such a constraint. Guaranteeing that user A gets the money, message, or item in his or her account as soon as user B sends it is easy on a single database. Spreading out the data among several copies of the data makes this consistency much more difficult. The way to solve this is to not expect or require the temporal constraint of instant transfer. More than likely it is totally acceptable that user A waits a few seconds before seeing the money that user B sent. The reason is simply that most dyads don't synchronously transfer items in a system. Obviously synchronous communication such as chat is different.

It is easy to place temporal constraints on your system because at first glance it appears that it would be the best customer experience to do so. However, before doing so consider the long-term ramifications of how difficult that system will be to scale because of the constraint.

Summary

We offered three rules in this chapter that deal with decisions you might make that can constrain your system's ability to scale. Start by not double-checking yourself. You employ expensive databases and hardware to ensure that your systems properly record transactions and events. Don't expect them *not* to work. We all have the need for redirection at times, but excessive use of this tool causes all types of problems from user experience to search engine indexing. Don't allow redirects to proliferate in your environment unchecked. Finally, consider the business requirements that you place on your system. Temporal constraints of items and objects make it difficult and expensive to scale. Carefully consider the real costs and benefits of these decisions.

Notes

1. Karlin Lillington, "PayPal Puts Dough in Your Palm," *Wired*, July 27, 1999, www.wired.com/1999/07/paypal-puts-dough-in-your-palm/.
2. Karlin Lillington, "It's Now Beam Me a Loan, Scotty," *Irish Times*, www.irishtimes.com/business/it-s-now-beam-me-a-loan-scotty-1.212019.
3. Marco Kesteloo and Nick Hodson, "2015 Retail Trends," Strategy&, www.strategyand.pwc.com/perspectives/2015-retail-trends.
4. Eric M. Jackson, *The PayPal Wars: Battles with eBay, the Media, the Mafia, and the Rest of Planet Earth* (Los Angeles: World Ahead Publishing, 2004).
5. Chris Skinner, "Celebrating PayPal's Centenary," http://thefinanser.com/2007/04/celebrating-paypals-centenary.html.

6. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, Networking Working Group Request for Comments 2616, "Hypertext Transfer Protocol—HTTP/1.1," June 1999,

   www.ietf.org/rfc/rfc2616.txt.

7. Apache HTTP Server Version 2.4 Documentation, "Apache Module mod_rewrite,"

   https://httpd.apache.org/docs/current/mod/mod_rewrite.html.

8. Apache HTTP Server Version 2.4 Documentation, "Apache HTTP Server Tutorial: .htaccess Files,"

   https://httpd.apache.org/docs/current/howto/htaccess.html.

9. Google Webmaster Central, Webmaster Tools Help, "URLs Not Followed Errors,"

   www.google.com/support/webmasters/bin/answer.py?answer=35156.

10. Wikipedia, "Constraint Satisfaction Problem,"

    http://en.wikipedia.org/wiki/Constraint_satisfaction_problem.

This page intentionally left blank

6

Use Caching Aggressively

In the world of business, it is often said that "Cash is King." In the world of technology, "Cache is King." By caching at every level from the browser through the cloud, one can significantly increase the ability to scale. Similar to the theme of Rule 17, "Don't Check Your Work" (see Chapter 5, "Get Out of Your Own Way"), caching also helps to minimize the amount of work your system does. Caching prevents you from needing to look up, create, or serve the same data over and over again. This chapter covers seven rules that will help guide you on the appropriate type and amount of caching for your application.

Scaling static content, such as text and images that don't change very often, is elementary. A number of rules in this book cover how to make static content highly available and scalable at low cost through the use of caches. Dynamic content, or content that changes over time, is not so elementary to serve quickly and scale out. Lon Binder, currently the CTO of Warby Parker, a lifestyle brand founded in 2010 offering designer eyewear, has been the CTO at several companies and has learned many lessons using caches to scale dynamic content. "If you make the entire site static HTML, it's easy to make it go fast," explained Lon. "But if you make the site dynamic, the content can be richer, more up-to-date—which makes for a better functional experience for the end user. But it's also much more challenging to make it performant, so that was the conflict and so this is where caching came in."

Prior to Warby Parker, Lon was at a company where he had between 7,000 and 8,000 landing pages. Some of these were very frequently changing pages that were personalized to the users. Some pages had large media elements such as multimegabyte video and hero images on product detail pages. Some image files were over 50MB because of the requirements for zoom-in capabilities.

To solve latency and scale issues, the first thing Lon's team did was to add a content distribution network; they chose Akamai. Lon stated, "It was really simple to just take all of our static assets and push them there [Akamai] and let them handle caching closest to the user. And then we could expire [the objects] using their typical cache expiration tools when we published. Expiring objects was part of our deploy process." For parts of the page that were personalized, such as at the top of the page where it might say, "Hi, Mike, welcome back," Lon's team used an Ajax call to replace the static content on the client side with the user's name. For bigger sections of the site, the team used iframes to replace content dynamically. All of this got Lon's team pretty far in terms of scaling and performance of the pages; however, they knew they needed to go farther.

Next Lon's team started profiling the application to understand what was causing slow load times. The team found some database queries that were being called frequently, requesting the same data over and over again. By tuning the database buffer size, they were able to reduce the execution time of the queries and keep the result set in cache.

Another challenge Lon and his team faced was that product detail pages changed on a regular basis. Merchandisers would submit work requests to the software team, who then changed the HTML for these product pages and deployed code. Each deployment would then invalidate any number of caches, causing high load on servers and slower response times to end customers. Lon's team closely monitored the cache hit/miss ratios. Determining that most of the changes were really small, Lon decided to build a small content management system that allowed the merchandisers to use a proprietary markup language to change information displayed, such as the price of the product, the SKU, and the content descriptions, without affecting the overall layout, the Cascading Style Sheets (CSS), or other assets, leaving the cache intact. This approach not only reduced the load on his development team but allowed assets to continue to be cached, thereby eliminating the increases in server load and customer response time.

Lon also considered caching pages served from the content management service from the content delivery network. The CDN could cache the pages and Lon's team could expire them through an API as marketers changed content. But the cost of caching full static pages was fairly high. Instead, Lon decided to pre-render sections of the page and then assemble them at the time of request. Lon recalled, "At the top of the page you have a view of the different images related to a product. This stuff didn't change that much, except for perhaps the price. So we would change the layout of the page in such a way where we could grab a huge chunk of that HTML, pre-render it, and cache a CLOB [character large object] in memory using Memcached. Memcached would distribute it across our cluster of application servers, making it super-easy. We proactively expired that cache based on business changes to data. We wouldn't proactively pre-render it, so the first request still took some time. Every call but the first call went really, really fast. One of the reasons that netted out to be faster is because we were able to get rid of those Ajax calls that I mentioned when we were doing the full reverse proxy caching. Removing that second request to the Web server was nice for the client side."

Lon and his team also attacked business objects. The team initially used an open-source object-relational mapping (ORM) tool called Entity Engine and then later moved to Hibernate. Eventually they scrapped the entire ORM and built a data access layer so they could optimize their queries better (ORMs are notoriously bad at generating queries). Above the data access layer they built a business access layer. As Lon described it, "Those business objects were like compositions. And those are where we had a caching rule. So what would happen, just to give you a sense of this, is a user would request a particular product. A product was a business object that might contain data from 20 different tables or views. So we'd bring these together, form a business object, and then cache that business object. We'd have a nice fully composed concept that was really meaningful to the business. The cost of assembling them is a little high, but the cost of caching them is relatively low because they really are only a bunch of text. A fully composed product model, there's not that much in there, maybe a few hundred bytes. So we ended up having this business engine that sat in our business access layer that would be able to proactively expire these business objects. And they were really great. We had a lot of debates about what things belonged in there. For example, does it ever make sense to cache a customer object? This was a very hotly debated topic for our organization because if you're a customer, while you're on the site we reuse your customer object in your session quite a bit, but it's just within your session."

Lon and his team learned a lot about caching and took the concept much further than most organizations. The result was incredibly fast page-rendering times even with very media-rich and dynamic content. Lon recapped, "We had all these levels of cache, and the thing that I'm most proud about is at that time—this is going back a few years—our site was performing faster than Amazon.com with substantially richer imagery with a ton of media on the site, huge images, high resolution, lots of video content, and lots of user-generated community content. It was an extremely dynamic site with very, very fast load times. You would get the HTML back in about 150 milliseconds and even the most complex pages would be under 700 milliseconds. So we were very, very happy with the performance."

A word of caution is warranted here before we get into these rules. As with any system implementation or major modification, the addition of caching, while often warranted, may create new challenges within your system. Multiple levels of caching can make it more difficult to troubleshoot problems in your product. For this reason you should design the caching to be monitored (as discussed in Rule 49; see Chapter 12, "Miscellaneous Rules"). While caching is a mechanism that often engenders greater scalability, it also needs to be engineered to scale well. Developing a caching solution that scales poorly will create a scalability chokepoint within your system and lead to lower availability down the road. The cache failures can have a catastrophic impact on the availability of your site as services soon get overloaded. Thus, you should ensure that you've designed the caches to be highly available and easily maintained. Finally, caching is an art best performed with deep experience. Look to hire engineers with past experience to help you with your caching initiatives.

Rule 20—Leverage Content Delivery Networks

Rule 20: What, When, How, and Why

What: Use CDNs (content delivery networks) to offload traffic from your site.

When to use: When speed improvements and scale warrant the additional cost.

How to use: Most CDNs leverage DNS to serve content on your site's behalf. Thus you may need to make minor DNS changes or additions and move content to be served from new subdomains.

Why: CDNs help offload traffic spikes and are often economical ways to scale parts of a site's traffic. They also often substantially improve page download times.

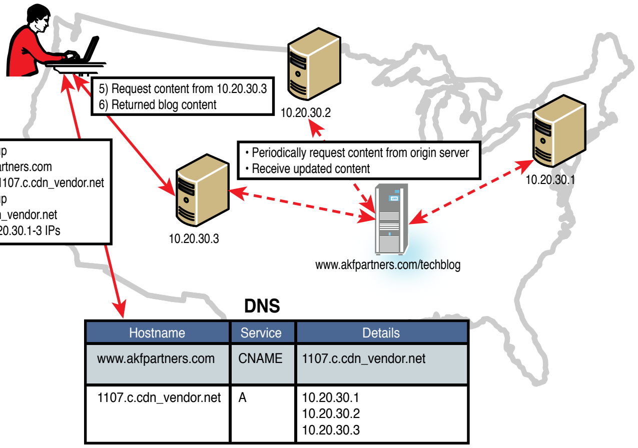
Key takeaways: CDNs are a fast and simple way to offset spikiness of traffic as well as traffic growth in general. Make sure you perform a cost-benefit analysis and monitor the CDN usage.

The easiest way to keep from being crushed under an avalanche of user demand is to avoid the avalanche altogether. There are two ways to achieve this. The first is by going out of business or never entering into business. A much better way, and one consistent with the theme of this book, is to have someone or something else handle much of the avalanche on your behalf. This is where content delivery networks (CDNs) come in.

CDNs are a collection of computers, called *nodes* or *edge servers*, connected via a network, called a *backbone*, that have duplicate copies of their customers' data or content (images, Web pages, and so on) on them. By strategically placing edge servers on different Tier 1 networks and employing a myriad of technologies and algorithms, the CDN can direct requests to nodes that are optimally suited to respond. This optimization could be based on such things as the fewest network hops, highest availability, or fewest requests. The focus of this optimization is most often the reduction of response times as perceived by the end user, requesting person, or service.

How this works in practice can be demonstrated best by an example; see Figure 6.1. Let's say the AKF blog was getting so much traffic that we decided to employ a CDN. We would set up a CNAME in DNS that pointed users requesting www.akfpartners.com/techblog to 1107.c.cdn_vendor.net (see the DNS table in Figure 6.1). The user's browser would then query DNS for akfpartners.com (step 1), receive the CDN domain name back (step 2), perform another DNS lookup on the CDN domain (step 3), receive IPs associated with 1107.c.cdn_vendor.net (step 4), and route and receive the request for our blog content to one of those IPs (steps 5–6). The content of our blog would be cached on the CDN servers, and periodically the CDN servers would query the origin or originating server, in this case our server hosting our blog, for updates.

As you can see in our example, the benefit of using a CDN in front of your own blog server is that the CDN takes all the requests (possibly hundreds or thousands per hour)



5) Request content from 10.20.30.3
6) Returned blog content

1) DNS lookup
www.akfpartners.com
2) Returned 1107.c.cdn_vendor.net
3) DNS lookup
1107.c.cdn_vendor.net
4) Return10.20.30.1-3 IPs

• Periodically request content from origin server
• Receive updated content

Hostname	Service	Details
www.akfpartners.com	CNAME	1107.c.cdn_vendor.net
1107.c.cdn_vendor.net	A	10.20.30.1 10.20.30.2 10.20.30.3

Figure 6.1 CDN example

and requests from your server only when checking for updated cache. This requires you to purchase fewer servers, less power, and smaller amounts of bandwidth, as well as requiring fewer people to maintain that infrastructure. Of course, this improvement in availability and response time isn't free—it typically comes at a premium to your public peering (Internet peering) traffic costs. Often CDN providers price on either the 95th percentile of peak traffic (like many transit providers) or total traffic delivered. Rates drop on a per-traffic-delivered basis as the traffic increases. As a result, analysis of when to convert to a CDN rarely works on a cost-only basis. It's necessary to factor in the reduction in response time to end users, the likely resulting increase in user activity (faster response often elicits more transactions), the increase in availability of your site, and the reduction in server, power, and associated infrastructure costs. In most cases, we've found that clients with greater than 10M of average revenues are better served by implementing CDNs than by continuing to serve that traffic themselves.

You might think that all this caching sounds great for static Web sites, but how does this help your dynamic pages? To start with, even dynamic pages have static content. Images, JavaScript, CSS, and so on are all usually static, which means they can be cached in a CDN. The actual text or content generated dynamically is usually the smallest portion of the page. Second, CDNs are starting to enable dynamic page support. Akamai offers a service called Dynamic Site Accelerator[1] that is used to accelerate and cache dynamic pages. Akamai was one of the companies, along with Oracle, Vignette, and others, that developed Edge Side Includes,[2] a markup language for assembling dynamic Web content on edge servers.

Whether you have dynamic or static pages on your site, consider adding a CDN to the mix of caches. This layer provides the benefits of faster delivery, typically very high availability, and less traffic on your site's servers.

Rule 21—Use `Expires` Headers

> ### Rule 21: What, When, How, and Why
>
> **What:** Use `Expires` headers to reduce requests and improve the scalability and performance of your system.
>
> **When to use:** All object types need to be considered.
>
> **How to use:** Headers can be set on Web servers or through application code.
>
> **Why:** The reduction of object requests increases the page performance for the user and decreases the number of requests your system must handle per user.
>
> **Key takeaways:** For each object type (image, HTML, CSS, PHP, and so on), consider how long the object can be cached and implement the appropriate header for that time frame.

It is a common misconception that you can control how pages are cached by placing meta tags, such as `Pragma`, `Expires`, or `Cache-Control`, in the `<HEAD>` element of the page. See the following code for examples. Unfortunately, meta tags in HTML are recommendations for how the browser should treat a page, but many browsers do not pay attention to them. Even worse, because proxy caches don't inspect the HTML, they do not abide by these tags at all.

```
<META HTTP-EQUIV="EXPIRES" CONTENT="Mon, 22 Aug 2011
11:12:01 GMT">
<META HTTP-EQUIV="Cache-Control" CONTENT="NO-CACHE">
```

HTTP headers, unlike meta tags, provide much more control over caching. This is especially true with regard to proxy caches because they do pay attention to headers. These headers cannot be seen in the HTML and are generated dynamically by the Web server or the code that builds the page. You can control them by configurations on the server or in code. A typical HTTP response header could look like this:

```
HTTP Status Code: HTTP/1.1 200 OK
Date: Thu, 21 Oct 2015 20:03:38 GMT
Server: Apache/2.2.9 (Fedora)
X-Powered-By: PHP/5.2.6
Expires: Mon, 26 Jul 2016 05:00:00 GMT
Last-Modified: Thu, 21 Oct 2015 20:03:38 GMT
Cache-Control: no-cache
Vary: Accept-Encoding, User-Agent
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8
```

A couple of the most pertinent headers for caching are the Expires and Cache-Control. The Expires entity-header field provides the date and time after which the response is considered stale. To mark a response as "never expires," the origin server should send a date one year from the time of response. In the preceding example, notice that the Expires header identifies the date 26 July 2016 with a time of 05:00 GMT. If today's date was 26 June 2016, the page requested would expire in approximately one month and should be refreshed from the server at that time.

The Cache-Control general-header field is used to specify directives that, in accordance with RFC 2616 Section 14 defining the HTTP 1.1 protocol, must be obeyed by all caching mechanisms along the request/response chain.[3] There are many directives that can be issued under the header, including public, private, no-cache, and max-age. If a response includes both an Expires header and a max-age directive, the max-age directive overrides the Expires header, even if the Expires header is more restrictive. Following are the definitions of a few of the Cache-Control directives:

- **public**—The response may be cached by any cache, shared or nonshared.
- **private**—The response message is intended for a single user and must not be cached by a shared cache.
- **no-cache**—A cache must not use the response to satisfy a subsequent request without revalidation with the origin server.
- **max-age**—The response is stale if its current age is greater than the value given (in seconds) at the time of a request.

There are several ways to set HTTP headers, including through a Web server and through code. In Apache 2.2 the configurations are set in the httpd.conf file. Expires headers require the mod_expires module to be added to Apache.[4] There are three basic directives for the Expires module. The first tells the server to activate the module, ExpiresActive. The next directive is to set the Expires header for a specific type of object such as images or text, ExpiresByType. The last directive is a default for how to handle all objects not specified by a type, ExpiresDefault. See the following code for an example:

```
ExpiresActive On
ExpiresByType image/png "access plus 1 day"
ExpiresByType image/gif "modification plus 5 hours"
ExpiresByType text/html "access plus 1 month 15 days 2 hours"
ExpiresDefault "access plus 1 month"
```

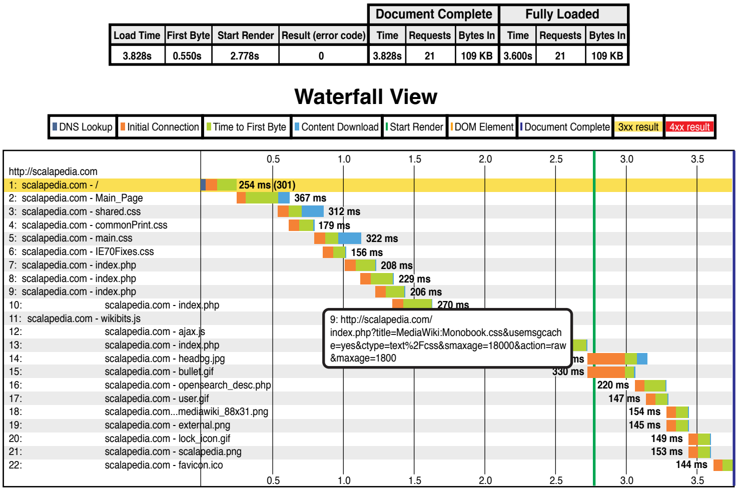
The other way to set HTTP Expires as well as Cache-Control and other headers is in code. In PHP this is pretty straightforward by using the header() command to send a raw HTTP header. This header() command must be called before any output is sent, either by HTML tags or from PHP. See the following sample PHP code for setting headers. Other languages have similar methods of setting headers.

```
<?php
header("Expires: 0");
header("Last-Modified: " . gmdate("D, d M Y H:i:s") . " GMT");
header("cache-control: no-store, no-cache, must-revalidate");
header("Pragma: no-cache");
?>
```

Our final topic deals with configuring Web servers for the optimization of performance and scale. *Keep-alives*, or HTTP persistent connections, allow for the reuse of TCP connections for multiple HTTP requests. In HTTP/1.1 all connections are considered persistent, and most Web servers default to allow keep-alives. According to the Apache documentation, the use of keep-alives has resulted in a 50% reduction in latency for HTML pages.[5] The default setting in Apache's httpd.conf file is KeepAlive On, but the default KeepAliveTimeOut is set at only 5 seconds. The benefit of longer time-out periods is that more HTTP requests do not have to establish, use, and break down TCP connections, but the benefit of short time-out periods is that the Web server threads will not be tied up for servicing other requests. Finding a balance between the two based on the specifics of your application or site is important.

As a practical example, we ran a test on one of our sites using webpagetest.org, the open-source tool developed by AOL for testing Web pages. The configuration was a simple MediaWiki running on an Apache HTTP Server v 2.2. Figure 6.2 shows the results from the test on the wiki page with the keep-alives turned off and the Expires headers not set. The initial page load was 3.8 seconds, and the repeat view was 2.3 seconds.

In Figure 6.3, the results are shown from the test on the wiki page with the keep-alives turned on and the Expires headers set. The initial page load was 2.6 seconds, and the repeat view was 1.4 seconds. This is a reduction in page load time of 32% for the initial page load and 37% for the repeat page load!



				Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
3.828s	0.550s	2.778s	0	3.828s	21	109 KB	3.600s	21	109 KB

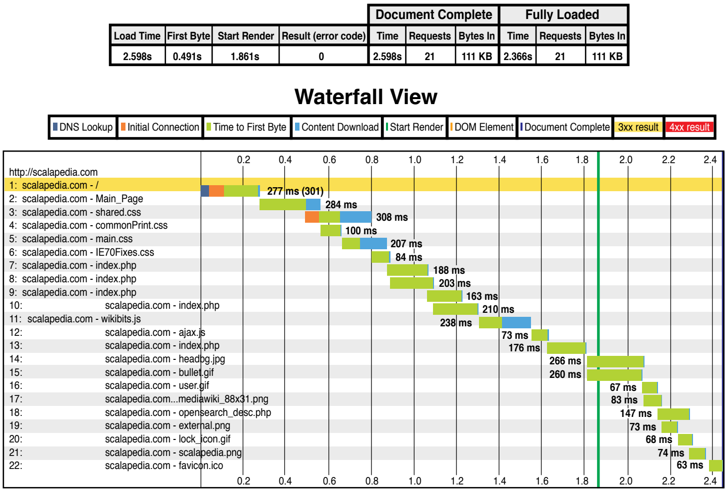
Waterfall View

DNS Lookup	Initial Connection	Time to First Byte	Content Download	Start Render	DOM Element	Document Complete	3xx result	4xx result

http://scalapedia.com	
1: scalapedia.com - /	254 ms (301)
2: scalapedia.com - Main_Page	367 ms
3: scalapedia.com - shared.css	312 ms
4: scalapedia.com - commonPrint.css	179 ms
5: scalapedia.com - main.css	322 ms
6: scalapedia.com - IE70Fixes.css	156 ms
7: scalapedia.com - index.php	208 ms
8: scalapedia.com - index.php	229 ms
9: scalapedia.com - index.php	206 ms
10: scalapedia.com - index.php	270 ms
11: scalapedia.com - wikibits.js	
12: scalapedia.com - ajax.js	
13: scalapedia.com - index.php	
14: scalapedia.com - headbg.jpg	ms
15: scalapedia.com - bullet.gif	330 ms
16: scalapedia.com - opensearch_desc.php	220 ms
17: scalapedia.com - user.gif	147 ms
18: scalapedia.com...mediawiki_88x31.png	154 ms
19: scalapedia.com - external.png	145 ms
20: scalapedia.com - lock_icon.gif	149 ms
21: scalapedia.com - scalapedia.png	153 ms
22: scalapedia.com - favicon.ico	144 ms

9: http://scalapedia.com/ index.php?title=MediaWiki:Monobook.css&usemsgcache=yes&ctype=text%2Fcss&smaxage=18000&action=raw&maxage=1800

Figure 6.2 Wiki page test (keep-alives off and no `Expires` headers)



				Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
2.598s	0.491s	1.861s	0	2.598s	21	111 KB	2.366s	21	111 KB

Waterfall View

DNS Lookup	Initial Connection	Time to First Byte	Content Download	Start Render	DOM Element	Document Complete	3xx result	4xx result

http://scalapedia.com	
1: scalapedia.com - /	277 ms (301)
2: scalapedia.com - Main_Page	284 ms
3: scalapedia.com - shared.css	308 ms
4: scalapedia.com - commonPrint.css	100 ms
5: scalapedia.com - main.css	207 ms
6: scalapedia.com - IE70Fixes.css	84 ms
7: scalapedia.com - index.php	188 ms
8: scalapedia.com - index.php	203 ms
9: scalapedia.com - index.php	163 ms
10: scalapedia.com - index.php	210 ms
11: scalapedia.com - wikibits.js	238 ms
12: scalapedia.com - ajax.js	73 ms
13: scalapedia.com - index.php	176 ms
14: scalapedia.com - headbg.jpg	266 ms
15: scalapedia.com - bullet.gif	260 ms
16: scalapedia.com - user.gif	67 ms
17: scalapedia.com...mediawiki_88x31.png	83 ms
18: scalapedia.com - opensearch_desc.php	147 ms
19: scalapedia.com - external.png	73 ms
20: scalapedia.com - lock_icon.gif	68 ms
21: scalapedia.com - scalapedia.png	74 ms
22: scalapedia.com - favicon.ico	63 ms

Figure 6.3 Wiki page test (keep-alives on and `Expires` headers set)

Rule 22—Cache Ajax Calls

> ### Rule 22: What, When, How, and Why
>
> **What:** Use appropriate HTTP response headers to ensure cacheability of Ajax calls.
>
> **When to use:** Every Ajax call except for those absolutely requiring real-time data that are likely to have been recently updated.
>
> **How to use:** Modify `Last-Modified`, `Cache-Control`, and `Expires` headers appropriately.

> **Why:** Decrease user-perceived response time, increase user satisfaction, and increase the scalability of your platform or solution.
>
> **Key takeaways:** Leverage Ajax and cache Ajax calls as much as possible to increase user satisfaction and increase scalability.

For those unfamiliar with common Web development terms, think of Ajax as one of the "approaches" behind some of those drop-down menus that start to offer suggestions as you type, or the map services that allow you to zoom in and out of maps without making additional round-trip calls to a distant server. If handled properly, Ajax not only makes for wonderfully interactive user interfaces, it helps us in our scalability endeavors by allowing the client to handle and interact with data and objects without requiring additional server-side work. However, if handled improperly, Ajax can actually create some unique scalability constraints by significantly increasing the number of requests our servers need to handle. Make no mistake about it, while these requests might be asynchronous from the perspective of the browser, a huge burst in a short period of time may very well flood our server farms and cause them to fail.

Ajax is an acronym for Asynchronous JavaScript and XML. While often referred to as a technology, it's perhaps best described as a group of techniques, languages, and approaches employed on the browser (or client side) to help create richer and more interactive Web applications. While the items within this acronym are descriptive of many Ajax implementations, the actual interactions need not be asynchronous and need not make use of XML only as a data interchange format. JSON may take the place of XML, for instance. JavaScript, however, is almost always used.

Jesse James Garrett is widely cited as having coined the term Ajax in 2005 in his article "Ajax: A New Approach to Web Applications."[6] In a loose sense of the term, Ajax consists of standards-based presentation leveraging CSS and DHTML, interaction and dynamic display capabilities facilitated by the Document Object Model (or DOM), a data interchange and manipulation mechanism such as XML with XSLT or JSON, and a data retrieval mechanism. Data retrieval is often (but not absolutely necessarily) asynchronous from the end user perspective. JavaScript is the language used to allow everything to interact within the client browser. When asynchronous data transfer is used, the XMLHttpRequest object is used. The purpose of Ajax is to put an end to the herky-jerky interactions described by our first experiences with the Internet, where everything was a request-and-reply interaction. With this background behind us, we move on to some of the scalability concerns associated with Ajax and finally discuss how our friend caching might help us solve some of these concerns.

Clearly we all strive to create interfaces that improve user interaction and satisfaction with the aim of increasing revenues, profits, and stakeholder wealth. Ajax is one method by which we might help facilitate a richer and more real-time experience for our end users. Because it can help eliminate what would otherwise be unnecessary round trips for interactions within our browser, user interactions can happen more quickly. Users can zoom in or zoom out without waiting for server responses, drop-down menus can be prepopulated based on previous entries, and users typing query strings into search bars can start to see potential search strings in which they might be interested to better guide their exploration. The asynchronous nature of Ajax can also help us load mail results into a client browser by repetitively fetching mail upon certain user actions without requiring the user to hit a "next page" button.

But some of these actions can also be detrimental to cost-effective scale of our platforms. Let's take the case of a user entering a search term for a specific product on a Web site. We may want to query a product catalog to populate suggested search terms for users as they type in search terms. Ajax could help with such an implementation by using each successive keystroke to send a request to our servers, return a result based on what was typed thus far, and populate that result in a drop-down menu without a browser refresh as the user types. Or the returned result may be the full search results of an as-yet-uncompleted string as the user types! Examples of both implementations can be found in many search engines and commerce sites today. But allowing each successive keystroke to ultimately result in a search query to a server can be both costly and wasteful for our back-end systems. A user typing "Beanie Baby," for instance, may cause 11 successive searches to be performed where only one is absolutely necessary. The user experience might be fantastic, but if the user types quickly, as many as eight to ten of those searches may never actually return results before he or she finishes typing.

Another way to achieve the same result without a 10x increase in traffic involves caching. With a little engineering, we can cache the results of previous Ajax interactions within the client browser and potentially within our CDNs (Rule 20), page caches (Rule 23), and application caches (Rule 24). Let's first look at how we can make sure that we leverage the cache in the browser.

Three key elements of ensuring that we can cache our content in the browser are the `Cache-Control` header, the `Expires` header, and the `Last-Modified` header of our HTTP response. Two of these we discussed in detail in relation to Rule 21. For `Cache-Control` we want to avoid the `no-store` option, and where possible we want to set the header to `public` so that any proxies and caches (such as a CDN) in between our endpoints (clients) and our servers can store result sets and serve them up to other requests. Of course we don't want private data sets to be public, but where possible we certainly want to leverage the high degree of caching that "public" offers us.

Our goal is to eliminate round trips to decrease both user-perceived response time and server load. As such, the `Expires` header of our response should be set far enough out into the future that the browser will cache the first result locally and read from it with subsequent requests. For static or semistatic objects, such as profile images or company logos, this might be set days or more into the future. Some objects might have greater temporal sensitivity, such as the reading of a feed of friends' status updates. In these cases, we might set `Expires` headers out by seconds or maybe even minutes to give the sense of real-time behavior while reducing overall load.

The `Last-Modified` header helps us handle conditional GET requests. In these cases, consistent with the HTTP/1.1 protocol, the server should respond with a 304 status if the item in cache is appropriate or still valid. The key to all these points is, as the "Http" portion of the name XMLHttpRequest implies, that Ajax requests behave (or should behave) the same as any other HTTP request and response. Our knowledge of these requests will aid us in ensuring that we increase the cacheability, usability, and scalability of all the systems that enable these requests.

While the previous approaches will help when we have content that we can modify in the browser, the problem becomes a bit more difficult when we use expanding search strings such as those we might find when a user interacts with a search page and starts typing a search string. There simply is no simple solution to this particular problem. But using `public` as the argument in the `Cache-Control` header will help to ensure that all similar search strings are cached in intermediate caches and proxies. Therefore, common beginnings of search strings and common intermediate search strings have a good chance of being cached somewhere before we get them. This particular problem can be generalized to other specific objects within a page leveraging Ajax. For instance, systems that request specific objects, such as an item for sale in an auction or a message in a social networking site or an e-mail system, should use specific message IDs rather than relative offsets when making requests. Relative names such as "page=3&item=2" which identifies the second message in the third page of a system can change, causing coherency and consistency problems. A better term would be "id=124556" representing an atomic item that does not change and can be cached for this user or future users when the item is public.

Cases where we have a static or even semidynamic set of items (e.g., limited or context-sensitive product catalog) are easy to solve. We can fetch these results, asynchronously from the client perspective, and both cache them for later use by the same client or perhaps more importantly ensure that they are cached by CDNs and intermediate caches or proxies for other clients performing similar searches.

We close this rule by giving an example of a bad response to an Ajax call and a good response. The bad response may look like this:

```
HTTP Status Code: HTTP/1.1 200 OK
Date: Thu, 21 Oct 2015 20:03:38 GMT
Server: Apache/2.2.9 (Fedora)
X-Powered-By: PHP/5.2.6
Expires: Mon, 26 Jul 1997 05:00:00 GMT
Last-Modified: Thu, 21 Oct 2015 20:03:38 GMT
Pragma: no-cache
Vary: Accept-Encoding,User-Agent
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8
```

Using our three topics, we notice that our `Expires` header occurs in the past. We are missing the `Cache-Control` header completely, and the last modified header is consistent with the date that the response was sent; together, these force all GETs to grab new content. A more easily cached Ajax result would look like this:

```
HTTP Status Code: HTTP/1.1 200 OK
Date: Thu, 21 Oct 2015 20:03:38 GMT
Server: Apache/2.2.9 (Fedora)
X-Powered-By: PHP/5.2.6
Expires: Sun, 26 Jul 2020 05:00:00 GMT
```

```
Last-Modified: Thu, 31 Dec 1970 20:03:38 GMT
Cache-Control: public
Pragma: no-cache
Vary: Accept-Encoding,User-Agent
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8
```

In this example, we set the `Expires` header to be well into the future, set the `Last-Modified` header to be well into the past, and told intermediate proxies that they can cache and reuse the object for other systems through `Cache-Control: public`.

Rule 23—Leverage Page Caches

> ### Rule 23: What, When, How, and Why
>
> **What:** Deploy page caches in front of your Web services.
>
> **When to use:** Always.
>
> **How to use:** Choose a caching solution and deploy.
>
> **Why:** Decrease load on Web servers by caching and delivering previously generated dynamic requests and quickly answering calls for static objects.
>
> **Key takeaways:** Page caches are a great way to offload dynamic requests, decrease customer response time, and scale cost-effectively.

A *page cache* is a caching server you install in front of your Web servers to offload requests for both static and dynamic objects from those servers. Other common names for such a system or server are reverse proxy cache, reverse proxy server, and reverse proxy. We use the term *page cache* deliberately, because whereas a proxy might also be responsible for load balancing or SSL (Secure Sockets Layer) acceleration, we are solely focused on the impact that these caching servers have on our scalability. When implemented, the proxy cache looks like Figure 6.4.

Page caches handle some or all of the requests until the pages or data that is stored in them is out of date or until the server receives a request for which it does not have the data. A failed request is known as a *cache miss* and might be a result of either a full cache with no room for the most recent request or an incompletely filled cache having either a low rate of requests or a recent restart. The cache miss is passed along to the Web server, which answers and populates the cache with the request, either replacing the least-recently-used record or taking up an unoccupied space.

There are three key points we emphasize in this rule. The first is that you should implement a page (or reverse proxy) cache in front of your Web servers and in doing so you will get a significant scalability benefit. Web servers that generate dynamic content do significantly less work as calculated results (or responses) are cached for the appropriate time. Web servers that serve static content do not need to look up that content, and you need fewer of them. We will, however, agree that the benefit of a page cache for static content isn't nearly as great as the benefit for dynamic content.

The second point is that you need to use the appropriate HTTP headers to ensure the greatest (but also business-appropriate) cache potential of your content and results.

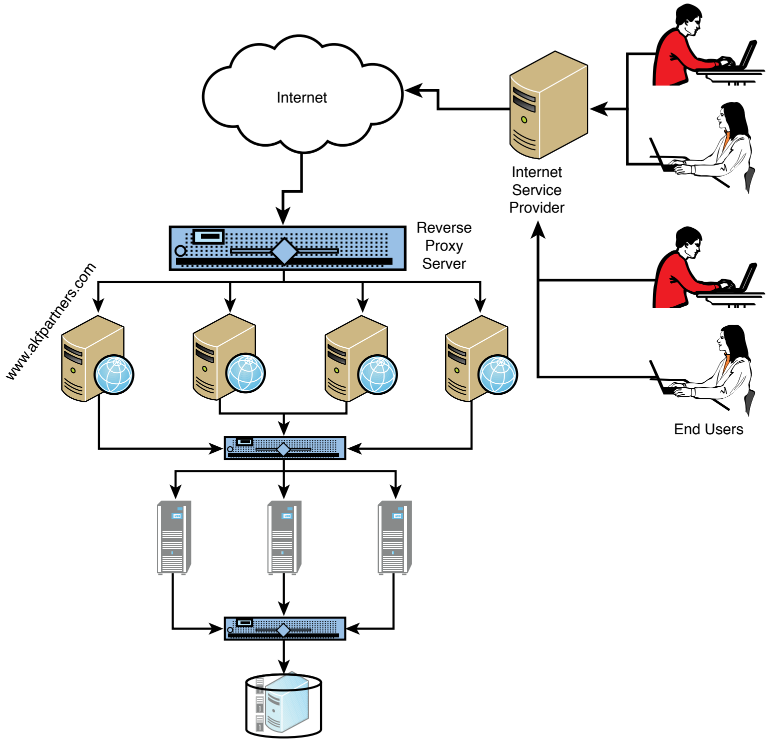


Figure 6.4 Proxy cache

For this, refer to our brief discussion of the `Cache-Control`, `Last-Modified`, and `Expires` headers in Rules 21 and 22. Section 14 of RFC 2616 has a complete description of these headers, their associated arguments, and the expected results.[7]

Our third point is that where possible you should include another HTTP header from RFC 2616 to help maximize the cacheability of your content. This new header is known as the `ETag`. The `ETag`, or entity tag, was developed to facilitate the method of `If-None-Match` conditional GET requests by clients of a server. `ETags` are unique identifiers issued by the server for an object at the time of first request by a browser. If the resource on the server side is changed, a new `ETag` is assigned to it. Assuming appropriate support by the browser (client), the object and its `ETag` are cached by the browser, and subsequent `If-None-Match` requests by the browser to the Web server will include the tag. If the tag matches, the server may respond with an HTTP 304 Not Modified response. If the tag is inconsistent with that on the server, the server will issue the updated object and its associated `ETag`.

The use of an `ETag` is optional, but to help ensure greater cacheability within page caches as well as all other proxy caches throughout the network transit of any given page or object, we highly recommend their use.

Rule 24—Utilize Application Caches

Rule 24: What, When, How, and Why

What: Make use of application caching to scale cost-effectively.

When to use: Whenever there is a need to improve scalability and reduce costs.

How to use: Maximize the impact of application caching by analyzing how to split the architecture first.

Why: Application caching provides the ability to scale cost-effectively but should be complementary to the architecture of the system.

Key takeaways: Consider how to split the application by Y axis (Rule 8) or Z axis (Rule 9) before applying application caching in order to maximize the effectiveness from both cost and scalability perspectives.

This isn't a section on how to develop an application cache. That's a topic for which you can get incredible, and free, advice by performing a simple search on your favorite Internet search engine. Rather we are going to make two basic but important points:

- The first is that you absolutely must employ application-level caching if you want to scale in a cost-effective manner.
- The second is that such caching must be developed from a systems architecture perspective to be effective long term.

We'll take for granted that you agree wholeheartedly with our first point and spend the rest of this rule on our second point.

In both Rule 8 and Rule 9 (see Chapter 2, "Distribute Your Work"), we hinted that the splitting of a platform (or an architecture) functionally by service or resource (Y axis, Rule 8), or by something you know about the requester or customer (Z axis, Rule 9), could pay huge dividends in the cacheability of data to service requests. The question is which axis or rule to employ to gain what amount of benefit. The answer to that question likely changes over time as you develop new features or functions with new data requirements. The implementation approach, then, needs to change over time to accommodate the changing needs of your business. The process to identify these changing needs, however, remains the same. The learning organization needs to constantly analyze production traffic, costs per transaction, and user-perceived response times to identify early indications of bottlenecks as they arise within the production environment and feed that data to the architecture team responsible for making changes.

The key question to answer here is what type of split (or refinement of a split) will garner the greatest benefit from a scalability and cost perspective. It is entirely possible that through an appropriate split implementation, and with the resulting cacheability of data within the application servers, 100 or even 100,000 servers can handle double, triple, or even 10x the current production traffic. To illustrate this, let's walk through a quick example of a common e-commerce site, a fairly typical SaaS site focused on servicing business needs, and a social networking or social interaction site.

Our e-commerce site has a number of functions, including search, browse, image inspection (including zooming), account update, sign-in, shopping cart, checkout, suggested items, and so on. Analysis of current production traffic indicates that 80% of our transactions across our most heavily used functions, including searching, browsing, and suggested products, occur across less than 20% of our inventory. Here we can leverage the Pareto Principle and deploy a Y axis (functional) split for these types of services to leverage the combined high number of hits on a comparatively small number of objects by our entire user base. Cacheability will be high, and our dynamic systems can benefit from the results delivered pursuant to similar earlier requests.

We may also find out that we have a number of power users—users with frequent requests. For these user-specific functions, we can decide to employ a Z axis split for user-specific functionality such as sign-in, shopping cart, account update (or other account information), and so on. While we can hypothesize instinctively, it is clearly valuable to get real production data from our existing revenue-producing site to help inform our decisions.

As another example, let's imagine that we have an SaaS business that helps companies handle customer support operations through hosted phone services, e-mail services, chat services, and a relationship management system. In this system, there are a great number of rules unique to any given business. On a per-business basis, it might require a great deal of memory to cache these rules and the data necessary for a number of business operations. If you've immediately jumped to the conclusion that a customer-oriented or Z axis split is the right approach, you are correct. But we also want to maintain some semblance of multitenancy within both the database and the application. How do we accomplish this and still cache our heaviest users to scale cost-effectively? Our answer, again, is the Pareto Principle. We can take the 20% of our largest businesses that might represent 80% of our total transaction volumes (such a situation exists with most of our customers) and spread them across several swim lanes of database splits. To gain cost leverage, we take 80% of our smaller users and sprinkle them evenly across all these swim lanes. The theory here is that the companies with light utilization are going to experience low cache hit rates whether or not they exist in their own swim lanes. As such, we might as well take our larger customers and allow them to benefit from caching while gaining cost leverage from our smaller customers. Those smaller customer experiences won't be significantly different unless we host them on their own dedicated systems, which we know runs counter to the cost benefits we expect in an SaaS environment.

Our last example deals with a social networking or interaction site. As you might expect, we are again going to apply the Pareto Principle and information from our production environment to help guide our decisions. Social networks often involve a small number of users with an incredibly skewed percentage of traffic. Sometimes these users are active consumers, sometimes they are active producers (destinations where other people go), and sometimes they are both.

Our first step might be to identify whether there is a small percentage of information or subsites that have a disproportionately high percentage of the "read" traffic. Such nodes within our social network can help guide us in our architectural considerations and might lead us to perform Z axis splits for these producers such that their nodes of activity are highly cacheable from a read perspective. Assuming the Pareto Principle holds true (as it typically does), we've now serviced nearly 80% of our read traffic with a small number of servers (and potentially page/proxy caches; see Rule 23). Our shareholders are happy because we can service requests with very little capital intensity.

What about the very active producers of content and/or updates within our social network? The answer may vary depending on whether their content also has a high rate of consumption (reads) or sits mostly dormant. In the case where these users have both high production (write/update) rates and high consumption (read) rates, we can just publish their content directly to the swim lane or node in which it is being read. If read and write conflicts start to become a concern as these "nodes" get hot, we can use read replication and horizontal scale techniques (the X axis or Rule 7), or we can start to think about how we order and asynchronously apply these updates over time (see Chapter 11, "Asynchronous Communication and Message Buses"). As we continue to grow, we can mix these techniques. If we still have troubles, after caching aggressively from the browser through CDNs to page and application caches (the rules in this chapter), we can continue to refine our splits. Maybe we enforce a hierarchy within a given user's updates and start to split them along content boundaries (another type of Y axis split—Rule 8), or perhaps we just continue to create read replicas of data instances (X axis—Rule 7). We may identify that the information read has a unique geographic bias (as is the case with some types of news) and we begin to split the data along geolocation-determined boundaries by request, which is something we know about the requester and therefore another type of Z axis split (Rule 9).

With any luck, you've identified a pattern in this rule. The first step is to hypothesize as to likely usage and determine ways to split to maximize cacheability. After implementing these splits in both the application and supporting persistent data stores, evaluate their effectiveness in production. Further refine your approach based on production data, and iteratively apply the Pareto Principle and the AKF Scale Cube (Rules 7, 8, and 9) to refine and increase cache hit rates. Lather, rinse, repeat.

Rule 25—Make Use of Object Caches

> ### Rule 25: What, When, How, and Why
>
> **What:** Implement object caches to help scale your persistence tier.
>
> **When to use:** Anytime you have repetitive queries or computations.
>
> **How to use:** Select any one of the many open-source or vendor-supported solutions and implement the calls in your application code.
>
> **Why:** A fairly straightforward object cache implementation can save a lot of computational resources on application servers or database servers.
>
> **Key takeaways:** Consider implementing an object cache anywhere computations are performed repeatedly, but primarily this is done between the database and application tiers.

Object caches are data stores (usually in memory) that store a hashed summary of each item. These caches are used primarily for caching data that may be computationally expensive to regenerate, such as the result set of complex database queries. A hash function is a mathematical function that converts a large and variable-sized amount of data into a small hash value.[8] This hash value (also called a hash sum or checksum) is usually an integer that can be used as an index in an array. This is by no means a full explanation of hash algorithms as their design and implementation constitute a domain unto itself, but you can test several of these on Linux systems with `cksum`, `md5sum`, and `sha1sum` as shown in the following code. Notice how variable lengths of data result in consistent 128-bit hashes.

```
# echo 'AKF Partners' | md5sum
90c9e7fd09d67219b15e730402d092eb[em][em]-
# echo 'Hyper Growth Scalability AKF Partners' | md5sum
faa216d21d711b81dfcddf3631cbe1ef[em][em]-
```

There are many different varieties of object caches such as the popular Memcached, Apache's OJB, and NCache just to name a few. As varied as the choice of tools are the implementations. Object caches are most often implemented between the database and the application to cache result sets from SQL queries. However, some people use object caches for results of complex application computations such as user recommendations, product prioritization, or reordering advertisements based on recent past performance. The object cache in front of a database tier is the most popular implementation because often the database is the most difficult and most expensive to scale. If you have the ability to postpone the split of a database or the purchase of a larger server, which is *not* a recommended approach to scaling, by implementing an object cache this is an easy decision. Let's talk about how to decide when to pull the trigger and implement an object cache.

Besides the normal suspects of CPU and memory utilization by the database, one of the most telling pieces of data that indicates when your system is in need of an object cache is the Top SQL report. This is a generic name for any report generated (or tool used) to show the most frequent and most resource-intensive queries run on the database. Oracle's Enterprise Manager Grid Control has a Top SQL Assessment built in for identifying the most resource-intensive SQL statements. Besides identifying and prioritizing the improvement of slow-running queries, this data can also be used to show which queries could be eliminated from the database by adding caching. There are equivalent reports or tools either built in or offered as add-ons for all the popular databases.

Once you've decided you need an implementation of an object cache, you need to choose one that best fits your needs. A word of caution for those engineering teams that at this point might be considering building a home-grown solution. There are more than enough production-grade object cache solutions to choose from. As an example, Facebook uses more than 800 servers supplying more than 28 terabytes of memory for its system.[9] While there are possible reasons that might drive you to make a decision to build an object cache instead of buying/using an open-source product, this decision should be highly scrutinized.

The next step is to actually implement the object cache, which is generally straightforward. Memcached supports clients for many different programming languages such as Java, Python, and PHP. In PHP the two primary commands are `get` and `set`. In the following example you can see that we connect to the Memcached server. If that fails, we just query the database through a function we call `dbquery`, not shown in the example. If the Memcached connection succeeds, we attempt to retrieve the `$data` that is associated with a particular `$key`. If that `get` fails, we query the `db` and set the `$data` into Memcached so that the next time we look for that data it is in Memcached. The `false` flag in the `set` command is for compression and the `90` is for the expiration time in seconds.

```
$memcache = new Memcache;
If ($memcache->connect('127.0.0.1', 11211)) {
[em][em]If ($data = $memcache->get('$key')) {
[em]} else {
[em][em][em][em]$data = dbquery($key);
[em][em][em][em]$memcache->set('$key',$data, false, 90);
[em][em]}
} else {
[em][em]$data = dbquery($key);
}
```

The final step in implementing the object cache is to monitor it for the cache hit rate. This ratio is the number of times the system requests an object that is in the cache compared to the total number of requests. Ideally this ratio is 85% or better, meaning that requests for objects not in cache or expired in cache occur only 15% or less of the time. If the cache hit ratio drops, you need to consider adding more object cache servers.

Rule 26—Put Object Caches on Their Own "Tier"

Rule 26: What, When, How, and Why

What: Use a separate tier in your architecture for object caches.

When to use: Anytime you have implemented object caches.

How to use: Move object caches onto their own servers.

Why: The benefits of a separate tier are better utilization of memory and CPU resources and having the ability to scale the object cache independently of other tiers.

Key takeaways: When implementing an object cache, it is simplest to put the service on an existing tier such as the application servers. Consider implementing or moving the object cache to its own tier for better performance and scalability.

In Rule 25 we covered the basics of implementing an object cache. We left off with you monitoring the object cache for cache hit ratio, and when this drops below ~85% we suggested that you consider expanding the object cache pool. In this rule, we're going to discuss where to implement the object cache pool and whether it should reside on its own tier within your application architecture.

Many companies start with the object cache on the Web or application servers. This is a simple implementation that works well to get people up and running on an object

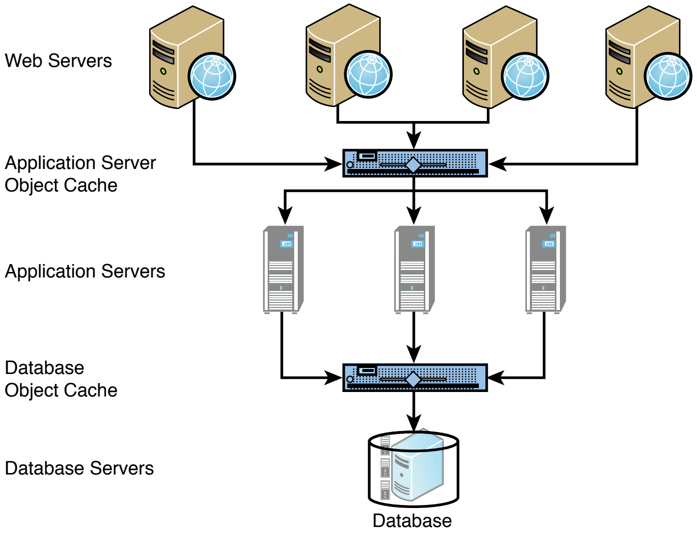


Figure 6.5 Object cache

cache without an investment in additional hardware or virtual instances if operating within a cloud. The downside to this is that the object cache takes up a lot of memory on the server, and it can't be scaled independently of the application or Web tier when needed.

A better alternative is to put the object cache on its own tier of servers. This would be between the application servers and the database, if using the object cache to cache query result sets. If caching objects created in the application tier, this object cache tier would reside between the Web and application servers. See Figure 6.5 for a diagram of what this architecture would look like. This is a logic architecture in that the object cache tier could be a single physical tier of servers that are used for database object caching as well as application object caching.

The advantage of separating these tiers is that you can size the servers appropriately in terms of memory and CPU requirements. Furthermore, you can scale the number of servers in this pool independently of other pools. Sizing the server correctly can save quite a bit of money since object caches typically require a lot of memory—most all store the objects and keys in memory—but require relatively low computational processing power. You can also add servers as necessary and have all the additional capacity utilized by the object cache rather than splitting it with an application or Web service.

Summary

In this chapter, we offered seven rules for caching. We have so many rules dedicated to this one subject because there are a myriad of caching options to consider, but also because caching is a proven way to scale a system. By caching at every level from the browser through the network all the way through your application to the databases, you can achieve significant improvements in performance as well as scalability.

Notes

1. Akamai, "Dynamic Site Accelerator," 2014,

   www.akamai.com/us/en/multimedia/documents/product-brief/dynamic-site-accelerator-product-brief.pdf.

2. Mark Tsimelzon et al., W3C, "ESI Language Specification 1.0,"

   www.w3.org/TR/esi-lang.

3. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, Networking Group Request for Comments 2616, June 1999, "Hypertext Transfer Protocol—HTTP/1.1,"

   www.ietf.org/rfc/rfc2616.txt.

4. Apache HTTP Server Version 2.4, "Apache Module mod_expires,"

   http://httpd.apache.org/docs/current/mod/mod_expires.html.

5. Apache HTTP Server Version 2.4, Apache Core Features, "KeepAlive Directive,"

   http://httpd.apache.org/docs/current/mod/core.html# keepalive.

6. Jesse James Garrett, "Ajax: A New Approach to Web Applications," Adaptive Path.com, February 18, 2005,

   http://adaptivepath.org/ideas/ajax-new-approach-web-applications/.

7. Fielding et al., Hypertext Transfer Protocol—HTTP/1.1, "Header Field Definitions,"

   www.w3.org/Protocols/rfc2616/rfc2616-sec14.html.

8. Wikipedia, "Hash Function,"

   http://en.wikipedia.org/wiki/Hash_function.

9. Paul Saab, "Scaling Memcached at Facebook," December 12, 2008,

   www.facebook.com/note.php?note_id=39391378919& ref=mf.

7

Learn from Your Mistakes

Research has long supported the position that we learn more from our failures than from our successes. But we can only truly learn from our failures if we foster an environment of open, honest communication and fold in lightweight processes that help us repeatedly learn and get the most from our mistakes and failures. Rather than emulate the world of politics, where failures are hidden from others and as a result bound to be repeated over time, we should strive to create an environment in which we share our failures as antipatterns to best practices. To be successful, we need to learn aggressively by watching customers and treating each failure as a learning opportunity, rely on organizations like QA appropriately, expect systems to fail, and design for those failures appropriately. The following story illustrates how one company was able to learn both deeply and broadly.

Intuit Inc. is an American software company that develops financial and tax preparation software and related services for small businesses, accountants, and individuals. As of 2015, Intuit was a highly successful global company with a market cap of almost $26 billion, revenues of $4.19 billion, and over 8,500 employees. Intuit was founded in 1983 by Scott Cook and Tom Proulx, and its first products were all desktop- and laptop-based products. By the mid-2000s consumer behavior started to change—users began to expect the products they once ran on their systems to instead be delivered as a service online (software as a service) and eventually via mobile device. Identifying these trends early, Intuit, over a number of years, developed Web and mobile versions of many of its products, including Quicken, TurboTax, and QuickBooks. Through both of these transitions, SaaS and mobile, Intuit made a point of learning rapidly, but they used much different approaches each time. The following stories exemplify Intuit's learnings in both their SaaS migration and mobile journey.

While Intuit's business shifted toward SaaS, their knowledge of hosting their own SaaS products was limited. The information technology group, led by the CTO, Tayloe Stansbury, was a growing team of individuals who were responsible for the infrastructure upon which software solutions would run. "We had a fairly major data center outage in 2010 that was started by a PDU [power distribution unit] failure," said Stansbury. "A repairman came in to fix that and managed to knock out the sister PDU while he was fixing the first one. We were completely without power in the data center, and one of the storage units didn't quiesce properly, causing a complete data loss. It took almost 24 hours to restore that data and get all of the applications running again."

Fortunately, this happened outside of any of Intuit's peaks, such as payroll and tax season. (Intuit makes a sizable portion of its revenue across a number of product lines during seasonal peaks such as the US tax preparation season.) As Tayloe recounted, "The amount of customer damage was not small, but it could have been much, much worse." It served as a good reminder that had the failure happened during peak season for a product line, it would have been much more damaging for the company. In a sense it was the perfect warning to make everybody in the organization aware of the missing knowledge. In relatively quick order, there were a number of deep-dive meetings with each of the teams to look at how they could get their failover architecture or active/active architecture in place. However, as Tayloe started to look at how Intuit could have understood and addressed these issues earlier, he discovered that a lot of the people who were driving these efforts didn't have the experience to develop the proper SaaS architecture and furthermore didn't know the right questions to ask.

As an example, one of the application leaders had asked her team about failover capacity. She was told that they had failover capability but because she didn't follow up and ask for more details, she didn't realize that the failover capacity was just a handful of QA servers. The good news was that these servers resided in another data center. The bad news was that they were meant only for testing, were not scaled to meet production traffic, and had never run the full set of software and services. That particular leader believed, based on the answers to her questions, that they were protected. "She didn't ask the second-, third-, and fourth-level questions and make sure they practiced the failover and back again," stated Tayloe. Realizing that his leadership team was missing this knowledge of scaling large SaaS systems, he set out to remedy the situation.

Over the next two years, Tayloe recruited directors and VPs for the engineering teams who had the knowledge and experience to architect and develop highly available SaaS systems. It wasn't a case where everyone was replaced, but Tayloe ensured that he brought in enough new people with the right skills that the entrenched team could be changed. Enough of the new skills were brought in that there was critical mass of new knowledge and culture. This approach worked well, and within a short time period the teams were diligently working on new active/active, highly available architectures.

The mobile transition also started with the Intuit team learning that they did not have the right skill set. As Intuit had grown up in the desktop space, they—not surprisingly—had a lot of desktop engineers. They did not have a bunch of mobile developers. Tayloe recounted, "While every other company in the valley was scrambling to build mobile, it was kind of a piece of cake for us, once we got the company as a whole tilted over to the idea that you really could do serious financial processing, financial systems, on mobile devices and that they weren't just for games." Tayloe's solution to the problem was to repurpose desktop engineers into mobile developers.

Tayloe stated, "It turns out that building for desktop and building for mobile really are quite isomorphic to each other. We were able to repurpose desktop engineers by putting them through about a month of training on either Android or iOS and get them productive on building high-quality mobile apps right after that. You can release several times a day to the Web. Every time you want to release to a client device you've got to package that up, and customers adopt it when they adopt it and you can't do it too frequently. It happened that we had a bunch of engineers who were used to building for that environment.

You lose a little bit of screen real estate, you gain some device sensors that you didn't have on the desktop, and otherwise it's kind of the same." Tayloe's point is that sometimes missing knowledge must be gained from outside sources, and sometimes you can retrain existing staff. These decisions require technology leaders to apply their expertise and judgment.

The Intuit stories serve to teach us that organizations must learn both deeply and broadly. Depth of learning comes from asking "why" multiple times until the answers stop coming and causes are clearly identified. Breadth of learning comes not only from looking at the technical and architectural fixes necessary to make a better product, but also from identifying what we need to do with training, people, organizations, and the processes we employ. Incidents like data center failures are costly, and we must learn deeply and broadly from them in order to prevent similar failures.

Rule 27—Learn Aggressively

> **Rule 27: What, When, How, and Why**
>
> **What:** Take every opportunity, especially failures, to learn and teach important lessons.
>
> **When to use:** Be constantly learning from your mistakes as well as your successes.
>
> **How to use:** Watch your customers or use A/B testing to determine what works. Employ a postmortem process and hypothesize failures in low-failure environments.
>
> **Why:** Doing something without measuring the results or having an incident without learning from it are wasted opportunities that your competitors are taking advantage of. We learn best from our mistakes—not our successes.
>
> **Key takeaways:** Be constantly and aggressively learning. The companies that learn best, fastest, and most often are the ones that grow the fastest and are the most scalable. Never let a good failure go to waste. Learn from every one and identify the architecture, people, and process issues that need to be corrected.

Do people in your organization think they know everything there is to know about building great, scalable products? Or perhaps your organization thinks it knows better than the customer. Have you heard someone say that customers don't know what they want? Although it might be true that customers can't necessarily articulate what they want, that doesn't mean they don't know it when they see it. Failing to learn continuously and aggressively, meaning at every opportunity, will leave you vulnerable to competitors who are willing to constantly learn.

Our continuing research on social contagion (also known as viral growth) of Internet-based products and services has revealed that organizations that possess a learning culture are far more likely to achieve viral growth than those that do not. In case you're not familiar with the terms *social contagion* and *viral growth*, the term *viral* derives from epidemiology (the study of health and illness in populations) and is used in reference to Internet-based companies to explain how things spread from user to user. The exponential growth of users is known as viral growth and implies the intentional sharing of information by people. In nature most people do not intentionally spread viruses, but on the Internet they do in the form of information or entertainment, and the resulting spread is similar to that of a virus. Once this exponential growth starts, it is possible to accurately predict its rate because it follows a power law distribution until the product reaches a point of nondisplacement. Figure 7.1 shows the growth in cumulative users for a product achieving viral growth (solid line) and one that just barely misses the tipping point by less than 10%.

The importance of creating a culture of learning cannot be underestimated. Even if you're not interested in achieving viral growth but want to produce great products for your customers, you must be willing to learn. There are two areas in which learning is critical. The first, as we have been discussing, is from the customers. The second is from the operations of the business/technology. We discuss each briefly in turn. Both rely on excellent listening skills. We believe that we were given two ears and one mouth to remind us to listen more than we talk.

Focus groups are interesting because you get an opportunity to sit down with your customers and hear what they think. The problem is that they, like most of us, can't really know how they will react to a product until they get to see and feel it in their own living room/computer. Not to delve too deeply into the philosophical realm, but this in part is caused by what is known as *social construction*. Put very simply, we make meaning of everything (and we do mean everything—it's been argued that we do this for reality itself) by labeling things with the meaning that is most broadly held within our social groups. While we can form our own opinions, they are most often just reflections or built on what others believe. So, how do you get around this problem of not being able to trust what customers say? Launch quickly and watch your customers' reactions.

Watching your customers can be done in a number of ways. Simply keeping track of usage and adoption of new features is a great start. The more classic A/B testing is even better. This is when you segment your customers into group A and group B randomly and allow group A to have access to one version of the product and group B access to

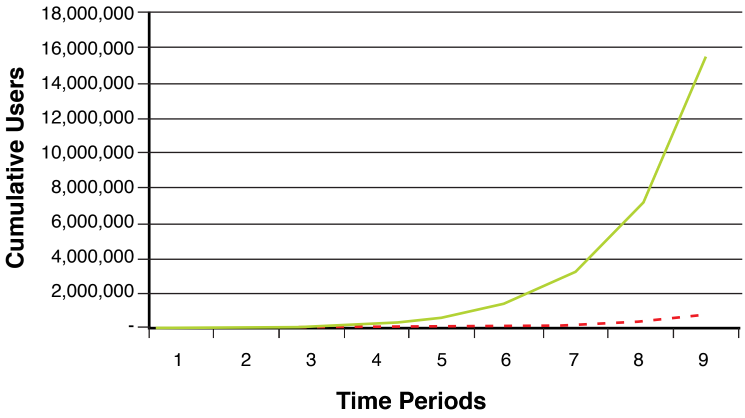


Figure 7.1 Viral growth

the other version. By comparing results, such as abandonment rates, time spent on site, conversion rates, and so on, you can decide which version performs better. Obviously some forethought must be put into the metrics that are going to be measured, but this is a great and fairly accurate way to compare product versions.

The other areas in which you must constantly learn if you want to achieve scalability are technology and business operations. You must never let incidents or problems pass without learning from them. Every site issue, outage, or downtime is an opportunity to learn how to do things better in the future.

Many of us, when discussing world events at social gatherings, have likely uttered sentences something to the effect of "We never seem to learn from history." But how many of us truly apply that standard to ourselves, our inventions, and our organizations within our work? There exists an interesting paradox within our world of highly available and highly scalable technology platforms: those systems are built to fail less often, and as a result the organizations have less opportunity to learn. Inherent to this paradox is the notion that each failure of process, systems, or people offers us an opportunity to perform a postmortem of the event for the purpose of learning. A failure to leverage these precious events to improve our people, processes, and architecture dooms us to continuing to operate exactly as we do today, which in turn means a failure to improve. A failure to improve, when drawn on a business contextual canvas of hyper-growth, becomes a painting depicting business failure. Too many things happen in our business when we are growing quickly to believe that a solution that we designed for x scale will be capable of supporting a business 10x the scale.

The world of nuclear power generation offers an interesting insight into this need to learn from our mistakes. In 1979, the TMI-2 reactor at Three Mile Island experienced a partial core meltdown, creating the most significant nuclear power accident in US history. This accident became the basis of several books, at least one movie, and two important theories on the source and need for learning in environments in which accidents are rare but costly.

Charles Perrow's Normal Accident Theory hypothesizes that the complexity inherent in modern coupled systems makes accidents inevitable.[1] The coupling inherent in these systems allows interactions to escalate rapidly with little opportunity for humans or control systems to interact successfully. Think back to how often you might have watched your monitoring solution go from all "green" to nearly completely "red" before you could respond to the first alert message.

Todd LaPorte, who developed the theory of high reliability organizations, believes that even in the case of an absence of accidents from which an organization can learn, there are organizational strategies to achieve higher reliability.[2] While the authors of these theories do not agree on whether their theories can coexist, they share certain elements. The first is that organizations that fail often have greater opportunities to learn and grow than those that do not, assuming of course that they take the opportunity to learn from them. The second, which follows from the first, is that systems that fail infrequently offer little opportunity to learn, and as a result in the absence of other approaches the teams and systems will not grow and improve.

Having made the point that learning from and improving after mistakes is important, let's depart from that subject briefly to describe a lightweight process by which we can learn and improve. For any major issue that we experience, we believe an organization should attack that issue with a postmortem process that addresses the problem in three distinct but easily described phases:

- **Phase 1: Timeline**—Focus on generating a timeline of the events leading up to the issue or crisis. Nothing is discussed other than the timeline during this first phase. The phase is complete once everyone in the room agrees that there are no more items to be added to the timeline. We typically find that even after we've completed the timeline phase, people continue to remember or identify timeline-worthy events in the next phase of the postmortem.
- **Phase 2: Issue identification**—The process facilitator walks through the timeline and works with the team to identify issues. Was it OK that the first monitor identified customer failures at 8:00 a.m. but that no one responded until noon? Why didn't the auto-failover of the database occur as expected? Why did we believe that dropping the table would allow the application to start running again? Each and every issue is identified from the timeline, but no corrections or actions are allowed to be made until the team is done identifying issues. Invariably, team members will start to suggest actions, but it is the responsibility of the process facilitator to focus the team on issue identification during Phase 2.
- **Phase 3: State actions**—Each item should have at least one action associated with it. For each issue on the list, the process facilitator works with the team to identify an action, an owner, an expected result, and a time by which it should be completed. Using the SMART principles, each action should be specific, measurable, attainable, realistic, and timely. A single owner should be identified, even though the action may take a group or team to accomplish.

No postmortem should be considered complete until it has addressed the people, process, and architecture issues responsible for the failure. Too often we find that clients stop at "a server died" as a root cause for an incident. Hardware fails, as do people and processes, and as a result no single failure should ever be considered the "true root cause" of any incident. The real question for any failure of scalability or availability is to ask, "Why didn't the holistic system act more appropriately?" If a database fails due to load, "Why didn't the organization identify the need earlier?" What process or monitoring should have been in place to help the organization find the issue? Why did it take so long to recover from the failure? Why isn't the database split up such that any failure has less of an impact on our customer base or services? Why wasn't there a read replica that could be quickly promoted as the write database? In our experience, you are never finished unless you can answer the question "why" at least five times to cover five different potential problems. Asking "why" five times is common but somewhat arbitrary, and you should continue to ask it until nothing new is revealed. It is a process for identifying multiple causes or contributing factors. Rarely do we see a failure caused by a single root cause.

Keep incident logs and review them periodically to identify repeating issues and themes.

Now that we've discussed what we should do, let's return to the case where we don't have many opportunities to develop such a system. Weick and Sutcliffe have a solution for organizations lucky enough to have built platforms that scale effectively and fail infrequently.[3] Their solution, as modified to fit our needs, is described as follows:

- **Preoccupation with failure**—This practice is all about monitoring our product and our systems and reporting errors in a timely fashion. Success, they argue, narrows perceptions and breeds overconfidence. To combat the resulting complacency, organizations need complete transparency into system faults and failures. Reports should be widely distributed and discussed frequently such as in a daily meeting to discuss the operations of the platform.
- **Reluctance to simplify interpretations**—Take nothing for granted and seek input from diverse sources. Don't try to box failures into expected behavior and act with a healthy bit of paranoia. The human tendency here is to explain small variations as being "the norm," whereas they can easily be your best early indicator of future failure.
- **Sensitivity to operations**—Look at detail data at the minute level. Include the usage of real-time data and make ongoing assessments and continual updates of this data.
- **Commitment to resilience**—Build excess capability by rotating positions and training your people in new skills. Former employees of eBay operations can attest that database administrators (DBAs), systems administrators (SAs), and network engineers used to be rotated through the operations center to do just this. Furthermore, once fixes are made, the organization should be quickly returned to a sense of preparedness for the next situation.
- **Deference to expertise**—During crisis events, shift the leadership role to the person possessing the greatest expertise to deal with the problem. Consider creating a competency around crisis management such as a "technical duty officer" in the operations center.

Never waste an opportunity to learn from your mistakes, as they are your greatest source of opportunity to make positive change. Put a process, such as a well-run postmortem, in place to extract every ounce of learning that you can from your mistakes. If you don't take time to perform a postmortem on the incident, get to the *real* root cause, and put that learning back into the organization so that you don't have that same failure again, you are bound to repeat your failures. Our philosophy is that while mistakes are unavoidable, making the same mistake twice is unacceptable. If a poor-performing query doesn't get caught until it goes into production and results in a site outage, we must get to the real root cause and fix it. In this case the root cause goes beyond the poorly performing query and includes the process and people that allowed it to get to production. If you have a well-designed system that fails infrequently, even under extreme scale, practice organizational "mindfulness" and get close to your data to better identify future failures easily. It is easy to be lured into a sense of complacency in these situations, and you are well served to hypothesize and brainstorm on different failure events that might happen. The key here is to learn from everything—mistakes as well as successes.

Rule 28—Don't Rely on QA to Find Mistakes

> ### Rule 28: What, When, How, and Why
>
> **What:** Use QA to lower the cost of delivered products, increase engineering throughput, identify quality trends, and decrease defects—*not* to increase quality.
>
> **When to use:** Whenever you can, get greater throughput by hiring someone focused on testing rather than writing code. Use QA to learn from past mistakes—always.
>
> **How to use:** Hire a QA person anytime you get greater than one engineer's worth of output with the hiring of a single QA person.
>
> **Why:** Reduce cost, increase delivery volume/velocity, decrease the number of repeated defects.
>
> **Key takeaways:** QA doesn't increase the quality of your system, as you can't test quality into a system. If used properly, it can increase your productivity while decreasing cost, and most importantly it can keep you from increasing defect rates faster than your rate of organizational growth during periods of rapid hiring.

Rule 28 has an ugly and slightly misleading and controversial title meant to provoke thought and discussion. Of course it makes sense to have a team responsible for testing products to identify defects. The issue is that you shouldn't rely solely on these teams to identify all your defects any more than airlines rely on flight attendants for safe landings of their planes. At the heart of this view is one simple fact: You can't test quality into your system. Testing only identifies issues that you created during development, and as a result it is an identification of value that you destroyed and can recapture. It is rare that testing, or the group that performs it, identifies untapped opportunities that might create additional value.

Don't get us wrong—QA definitely has an important role in an engineering organization. It is a role that is even more important when companies are growing at an incredibly fast rate and need to scale their systems. The primary role of QA is to help identify product problems at a lower cost than having engineers perform the same task. Two important derived benefits from this role are to increase engineering velocity and to increase the rate of defect detection.

These benefits are achieved similarly to the fashion in which the Industrial Revolution reduced the cost of manufacturing and increased the number of units produced. By pipelining the process of engineering and allowing engineers to focus primarily on building products (and of course unit-testing them), less time is spent per engineer in the setup and teardown of the testing process. Engineers now have more time per day to focus on building applications for the business. Typically we see both output per hour and output per day increase as a result of this. Cost per unit drops as a result of higher velocity at static cost. Additionally, the headcount cost of a great QA organization typically is lower on a per-head basis than the cost of an engineering organization, which further reduces cost. Finally, as the testing organization is focused and incented to identify defects, they don’t have any psychological conflicts with finding problems within their own code (as many engineers do) or the code of a good engineering friend who sits next to them.

> ### When to Hire a QA Person
>
> You should hire a QA person anytime you can get one or more engineer’s worth of productivity out of hiring someone in QA. The math is fairly simple. If you have 11 engineers, and each of them spends roughly 10% of his or her time on testing activities that could be done by a single QA person, then by hiring a single QA person you can get 1.1 engineer’s worth of productivity back. Typically that person will also come at lower cost than an engineer, so you get 1.1 engineer’s worth of work at .8 or .9 the cost of an engineer.

None of this argues against pairing engineers and QA personnel together as in the case of well-run agile processes. In fact, for many implementations, we recommend such an approach. But the division of labor is still valuable and typically achieves the goals of reducing cost, increasing defect identification, and increasing throughput.

But the greatest as-of-yet-unstated value of QA organizations arises in the case of hyper-growth companies. It’s not that this value doesn’t exist within static companies or companies of lower growth, but it becomes even more important in situations where engineering organizations are doubling (or more) in size annually. In these situations, standards are hard to enforce. Engineers with longer tenure within the organization simply don’t have time to keep up with and enforce existing standards and even less time to identify the need for new standards that address scale, quality, or availability needs. In the case where a team doubles year over year, beginning in year three of the doubling, half of the existing “experienced” team has only a year or less of company experience!

That brings us to why this rule is in the chapter on learning from mistakes. Imagine an environment in which managers spend nearly half of their time interviewing and hiring engineers and in which in any given year half of the engineers (or more) have less than a full year with the company. Imagine how much time the existing longer-tenured engineers will spend trying to teach the newer engineers about the source code management system, the build environments, the production environments, and so on. In such an environment too little time is spent validating that things have been built correctly, and the number of mistakes released into QA (but ideally not production) increases significantly.

In these environments, it is QA’s job to teach the organization what is happening from a quality perspective and where it is happening such that the engineering organization can adapt and learn. QA then becomes a tool to help the organization learn what mistakes it is making repeatedly, where those mistakes lie, and ideally how the organization can keep from making them in the future. QA is likely the only team capable of seeing the recurring problems.

Newer engineers, without the benefit of seeing their failures and the impacts of those failures, will likely not only continue to make them, but the approaches that led to these failures will become habit. Worse yet, they will likely train those bad habits in the newly hired engineers as they arrive. What started out as a small increase in the rate of defects will become a vicious cycle. Everyone will be running around attempting to identify the root cause of the quality nightmare, when the nightmare was bound to happen and is staring them in the face: a failure to learn from past mistakes!

Techniques such as code reviews and test-driven development (TDD) help engineers create quality code, identifying defects before they reach QA. Conducting code reviews one on one with peers helps engineers find and fix mistakes overlooked in the initial development process. Test-driven development is a technique that has the engineer develop the automated test case that defines the new feature or function, then produce the minimum amount of code to pass the test. TDD improves code quality while increasing productivity. These techniques help build quality into the software early in the process and reduce rework.

QA must work to identify where a growing organization is having recurring problems and create an environment in which those problems are discussed and eliminated. And here, finally, is the most important benefit of QA: it helps an organization learn from engineering failures. QA helps mitigate risk. Defects are created by engineers. Understanding that they can't test quality into the system, and unwilling to accept a role as a safety screen, like being behind a catcher in baseball to stop uncaught balls, the excellent QA organization seeks to identify systemic failures in the engineering team that lead to later quality problems. This goes beyond the creation of burn-down charts and find/fix ratios; it involves digging into and identifying themes of problems and their sources. Once these themes are identified, they are presented along with ideas on how to solve the problems.

Rule 29—Failing to Design for Rollback Is Designing for Failure

Rule 29: What, When, How, and Why

What: Always have the ability to roll back code.

When to use: Ensure that all releases have the ability to roll back, practice it in a staging or QA environment, and use it in production when necessary to resolve customer incidents.

How to use: Clean up your code and follow a few simple procedures to ensure that you can roll back your code.

Why: If you haven't experienced the pain of not being able to roll back, you likely will at some point if you keep playing with the "fix-forward" fire.

Key takeaways: Don't accept that the application is too complex or that you release code too often as excuses that you can't roll back. No sane pilot would take off in an airplane without the ability to land, and no sane engineer would roll code that he or she could not pull back off in an emergency.

To set the right mood for this next rule we should all be gathered around a campfire late at night telling scary stories. The story we're about to tell you is your classic scary story, including the people who hear scary noises in the house but don't get out. Those foolish people who ignored all the warning signs were us.

It was the week of October 4, 2004. The PayPal engineering team had just completed arguably its most complex cycle of development focused on major architectural changes. These changes were meant to split up what was largely a monolithic database named "CONF." CONF contained the accounts of every PayPal user and was integral to the sending and receiving of money synchronously on the PayPal system.

For years, PayPal hadn't needed to consider separating the monolithic database. While transaction and user account growth was spectacular (PayPal consistently ranked in the top five fastest-growing solutions by user account through 2004), Moore's Law allowed the company to stall significant splits in its database architecture by purchasing bigger and faster systems. In 2004 the company hosted CONF on a Solaris Sun Fire 15K capable of hosting 106 1.35GHZ processors on 18 system boards or "uniboards." Even with all of this comparatively large processing power (for the time), the company was burning through the capacity of the system (then Sun's—now Oracle's—largest system available). While Sun had announced an even larger server, the E25K, the senior technology executives (eBay SVP of technology and CTO Marty Abbott, PayPal CTO Chuck Geiger, and PayPal VP of engineering and architecture Mike Fisher) felt they could no longer rely on vertical scale for future growth.

Because the PayPal product started with functionality for transferring money on early mobile devices, the PayPal product owners felt strongly that any transfer of money between accounts needed to happen in "real time." For example, if Marty wanted to send $50 to Mike, Marty's account had to be immediately debited and the funds immediately credited to Mike's account for the transaction to be successful. A failure in debit or credit would necessitate a rollback of the transaction. Whether such an approach was truly a competitive differentiator, it was absolutely a departure from typical industry practice in payments at other institutions such as traditional banks. These solutions performed the debit and credit of money in separate transactions. The debit occurs first on Marty's account. This may be a debit of the amount immediately, or a notification of a pending transaction. A notification to Mike Fisher's account may or may not happen in the temporal vicinity of the debit to Marty's account. If some notification happens, it is typically a separate process that indicates an inbound pending transaction. Some future process then performs any final debits and credits—usually not rolled into a synchronous transaction but rather multiple asynchronous transactions. We'll discuss how the PayPal team approached solving this problem in Chapter 8, "Database Rules." For now, let's join Chuck Geiger, PayPal's CTO at the time, and hear what he had to say about the implementation.

"PayPal had, since its early days, always rolled new code out and planned to 'fix forward' any mistakes found in production. This approach had never caused significant problems for the business with any release prior to 24.0. It did, however, stand in stark contrast to the then parent company eBay's process of always ensuring that a release could be rolled back at any time after introduction to production," explained Chuck.

"The PayPal engineering team felt that the cost to be able to always roll back any release simply exceeded the value of such a process. The thought process was that the cost applied to any release was high on a per-release basis, and that the probability that something would need to be rolled back instead of being fixed forward was low. My experience now tells me that there are two failures in this type of analysis, but I'll get to that in a minute. When 24.0 rolled out, it was an unmitigated disaster. We simply could not process transactions during peak US daytime hours. This continued for at least three business days as we tried to fix the problems associated with the release. That was three days of failure to perform for our users."

eBay issued this apology[4] to its users on October 14, 2004:

> Many of you have experienced problems using PayPal over the last few days. First we want to apologize for the way this has impacted your buying and selling activity and, for many of you, your businesses.
>
> As we communicated earlier, the recent problems accessing the site and using PayPal functions were the result of unforeseen problems with new code that was launched on Friday 10/8 to upgrade the site architecture. While account data and personal information were never compromised, many users experienced difficulties using the PayPal functions to perform payment and shipping activities. Technology teams from both PayPal and eBay worked around the clock to bring the service back to normal as soon as possible.
>
> PayPal functionality has been restored and PayPal users, on and off eBay, are able to resume normal trading activities. Going forward, our technical teams will continue to focus on ensuring that we provide the highest standard of reliability and safety on the PayPal platform.
>
> Thank you for your patience during this process. Providing outstanding service is our top priority at PayPal. Although we didn't meet your expectations—or our own—in the past few days, we look forward to the opportunity to regain your faith in our service.
>
> Sincerely,
>
> Meg Whitman, President and CEO, eBay
> Matt Bannick, Senior Vice President, Global Online Payments, eBay
> and General Manager, PayPal
> Marty Abbott, Senior Vice President, Technology, eBay

Chuck continued, "As if the actual site problem weren't enough, through a dedicated effort to be consistent with eBay's ability to roll back we determined that the cost of enabling rollback for any release wasn't high. Within two weeks we created an approach to allow us to modify database schemas in advance of software changes and thereby prove backward compatibility. And here we come to the failures in analysis I had mentioned: (1) The cost of being able to roll back from any release is almost always less than you think it will be, and (2) the value of being able to roll back is immeasurable. It will take you only one significant failure to feel as I do—but I urge you not to take that risk."

The following bulleted points provided us and many other teams since then the ability to roll back. As you'd expect, the majority of the problem with rolling back is in the database. By going through the application to clean up any outstanding issues and then adhering to some simple rules, every team should be able to roll back.

- **Database changes must only be additive**—Columns or tables should only be added, not deleted, until the next version of code is released that deprecates the dependency on those columns. Once these standards are implemented, every release should have a portion dedicated to cleaning up the last release's data that is no longer needed.
- **Database changes scripted and tested**—The database changes that are to take place for the release must be scripted ahead of time instead of applied by hand. This should include the rollback script. The two reasons for this are that (1) the team needs to test the rollback process in QA or staging to validate that they have not missed something that would prevent rolling back, and (2) the script needs to be tested under some amount of load condition to ensure that it can be executed while the application is using the database.
- **Restricted SQL queries in the application**—The development team needs to disambiguate all SQL by removing all `SELECT *` queries and adding column names to all `UPDATE` statements.
- **Semantic changes of data**—The development team must not change the definition of data within a release. An example would be a column in a ticket table that is currently being used as a status semaphore indicating three values such as assigned, fixed, or closed. The new version of the application cannot add a fourth status until code is first released to handle the new status.
- **Wire on/wire off**—The application should have a framework added that allows code paths and features to be accessed by some users and not by others, based on an external configuration. This setting can be in a configuration file or a database table and should allow for both role-based access as well as access based on random percentage. This framework allows for beta testing of features with a limited set of users and allows for quick removal of a code path in the event of a major bug in the feature, without rolling back the entire code base.

We learned a painful but valuable lesson that left scars so deep we never pushed another piece of code that couldn't be rolled back. Even though we moved on to other positions with other teams, we carried that requirement with us. As you can see, the preceding guidelines are not overly complex but rather straightforward rules that any team can apply and have rollback capability going forward. The additional engineering work and additional testing to make any change backward compatible have the greatest ROI of any work you can do.

Summary

This chapter has been about learning. Learn aggressively, learn from others' mistakes, learn from your own mistakes, and learn from your customers. Be a learning organization and a learning individual. The people and organizations that constantly learn will always be ahead of those that don't. As Charlie "Tremendous" Jones, the author of nine books and numerous awards, said, "In ten years you will be the same person you are today except for the people you meet and the books you read." We like to extend that thought to say that an organization will be the same tomorrow as it is today except for the lessons learned from its customers, itself, and others.

Notes

1. Charles Perrow, *Normal Accidents* (Princeton, NJ: Princeton University Press, 1999).
2. Todd R. LaPorte and Paula M. Consolini, "Working in Practice but Not in Theory: Theoretical Challenges of 'High-Reliability Organizations,'" *Journal of Public Administration Research and Theory*, Oxford Journals, http://jpart.oxfordjournals.org/content/1/1/19.extract.
3. Karl E. Weick and Kathleen M. Sutcliffe, "Managing the Unexpected," http://high-reliability.org/Managing_the_Unexpected.pdf.
4. "A Message to Our Users—PayPal Site Issues Resolved," http://community.ebay.com/t5/Archive-Technical-Issues/A-Message-To-Our-Users-PayPal-Site-Issues-Resolved/td-p/996304.

8

Database Rules

As we discussed in Chapter 7, "Learn from Your Mistakes," we believe that an incident is a terrible thing to waste. In a sense, at the conclusion of any incident, we've paid some price. This price may be lost revenue associated with lost transactions, or in subscription model products it may be in an impact on customer satisfaction and perhaps future customer churn. As such, we need to reap the maximum value of that incident and learn everything we possibly can from it. The PayPal 24.0 incident described in Chapter 7 is an example of an incident with many hidden lessons.

Recall from our discussion in that chapter that the PayPal approach (circa 2004) of immediate synchronous debits and credits was inconsistent with industry practices and thought by the product team to be a competitive differentiator relative to the industry. It's very simple to perform a debit and credit of separate accounts when all the accounts for a given transaction exist on the same physical and logical database. In fact, the transaction will rarely fail—or will fail in its entirety quickly due to physical (host-based) problems or logical (database or application) problems. But ensuring this level of transaction integrity across two separate servers is a much more difficult problem altogether. Applying the traditional approach, of handling the transactions separately with some notion of a "pending" transaction (each leg of the transaction happening asynchronously relative to the other), makes the otherwise difficult problem of the accounts existing on separate databases comparatively simple. Requiring that the transaction continue to happen synchronously and in real time is a difficult problem to solve.

The PayPal architecture team attempted to solve the problem in release 24.0 by applying a computer science approach known as a two-phase commit (2PC). Two-phase commit consists of a commit request (sometimes known as voting) phase and a second phase to perform the commit. During the voting phase, the process attempts to lock the necessary attributes (in this case user accounts), and only if locks are successful does it proceed to a commit. Assuming a successful lock, the process performs the commit. If any phase fails, the entire process rolls back.

The engineers were aware of the well-known disadvantages of the 2PC protocol and approach, namely, that the solution requires blocking a transaction through database locks and that when distributed across systems it may result in a phenomenon known as deadlock (the halting of all processing against certain attributes with locks that cannot be resolved easily). To reduce the probability of such an event, the initial approach was to limit the number of accounts split from CONF in order to reduce the number of transactions that would require a 2PC.

Unfortunately, this risk mitigation step did not have the desired effect. After 24.0 was released in the evening hours, even with relatively low traffic, the PayPal site started to have problems. Many transactions were slow to complete, and others seemed to block in their entirety. Users started to complain, and the monitors displaying payment volume by minute dipped precipitously—to rates of less than 50% of expected payment volumes. The PayPal system was broken.

Engineers worked feverishly to figure out how to resolve the situation and still maintain the initial split of accounts that would help pave the way for future scalability across multiple databases. But alas, the engineering and architecture teams could not figure out how to get the 2PC protocol to function at a speed fast enough to keep transactions from blocking, even when a relatively small number of transactions required the 2PC approach. The initial changes released late in the week of October 4 were eliminated, and accounts were migrated back to CONF. As the authors like to tell our clients, there are always solutions other than 2PC—solutions that are almost guaranteed to result in higher availability, faster response times, and significantly greater scalability.

In Chapter 4, "Use the Right Tools," we discussed Maslow's Hammer (aka the Law of the Instrument), which put simply is an overreliance, to a fault, on a familiar tool. We discussed that one common example of overuse is the relational database. Recall that relational databases typically give us certain benefits outlined by an acronym called ACID, described in Table 8.1.

ACID properties are really powerful when we need to split up data into different entities, each of which has some number of relationships with other entities within the database. They are even more powerful when we want to process a large number of transactions through these entities and relationships: transactions consisting of reads of the data, updates to the data, the addition of new data (inserts or creates), and removal of certain data (deletes). While we should always strive to find more lightweight and faster ways to perform transactions, sometimes there simply isn't an easy way around using a relational database, and sometimes the relational database is the best option for our implementation given the flexibility it affords. Whereas Rule 14 (Chapter 4) argued against using databases where they are not necessary, this chapter, when used in conjunction with the rules of Chapter 2 ("Distribute Your Work"), helps us make the most of databases without causing major scalability problems within our architecture.

Table 8.1 **ACID Properties of Databases**

Property	Explanation
Atomicity	All of the operations in the transaction will complete, or none will.
Consistency	The database will be in a consistent state when the transaction begins and ends.
Isolation	The transaction will behave as if it is the only operation being performed upon the database.
Durability	Upon completion of the transaction, the operation will not be reversed.

Rule 30—Remove Business Intelligence from Transaction Processing

Rule 30: What, When, How, and Why

What: Separate business systems from product systems and product intelligence from database systems.

When to use: Anytime you are considering internal company needs and data transfer within, to, or from your product.

How to use: Remove stored procedures from the database and put them in your application logic. Do not make synchronous calls between corporate and product systems.

Why: Putting application logic in databases is costly and represents scale challenges. Tying corporate systems and product systems together is also costly and represents similar scale challenges as well as availability concerns.

Key takeaways: Databases and internal corporate systems can be costly to scale due to license and unique system characteristics. As such, we want them dedicated to their specific tasks. In the case of databases, we want them focused on transactions rather than product intelligence. In the case of back-office systems (business intelligence), we do not want our product tied to their capabilities to scale. Use asynchronous transfer of data for business systems.

We often tell our clients to steer clear of stored procedures within relational databases. One of their first questions is typically "Why do you hate stored procedures so much?" The truth is that we don't dislike stored procedures. In fact, we've used them with great effect on many occasions. The problem is that stored procedures are often overused within solutions, and this overuse sometimes causes scalability bottlenecks in systems that would otherwise scale efficiently and almost always results in a very high cost of scale. Given the emphasis on databases, why didn't we put this rule in the chapter on databases? The answer is that the reason for our concerns about stored procedures is really the need to separate business intelligence and product intelligence from transaction processing. In general, this concept can be further abstracted to "Keep like transactions together (or alternatively separate unlike transactions) for the highest possible availability and scalability and best possible cost." Those are a lot of words for a principle, so let's first return to our concern about stored procedures and databases as an illustration of why this separation should occur.

Databases tend to be one of the most expensive systems or services within your architecture. Even in the case where you are using an open-source database, in most cases the servers upon which these systems exist are attached to a relatively high-cost storage solution (compared to other solutions you might own), have the fastest and largest number of processors, and have the greatest amount of memory. Often, in mature environments, these systems are tuned to do one thing well—perform relational operations and commit transactions to a stable storage engine as quickly as possible. The cost per compute cycle on these systems tends to be higher than that of the remainder of the solutions or services within a product's architecture (for example, application servers or Web servers). These systems also tend to be the points at which certain services converge and the defining points for a swim lane. In the most extreme sense, such as in a young product, they might be monolithic in nature and as a result the clear governor of scale for the environment.

Using stored procedures and business logic on the database also makes replacement of the underlying database down the road much more challenging. If you decide to change to an open-source or a NoSQL solution, you will need to develop a plan to migrate the stored procedures or replace the logic in the application. Although tools have been developed to help migrate stored procedures, if you do not have to deal with them in the first place the migration to another solution becomes much easier.

For all these reasons, using this expensive compute resource for business logic makes very little sense. Each transaction will only cost more as the system is more expensive to operate. The system is likely also a governor to our scale, so why would we want to steal capacity by running other than relational transactions on it? For all these reasons, we should limit these systems to database (or storage-related or NoSQL) transactions to allow the systems to do what they do best. In so doing we can both increase our scalability and reduce our cost for scale.

Stored procedures also represent a bit of a concern for our product development processes. Many companies are attempting to completely automate their testing in order to reduce time to market and decrease their cost of quality. The approach these companies take is very much ground up, starting with unit testing and ending with completely automated regression test suites. The most advanced companies perform all of this within continuous integration environments that run unit, system, and regression tests continuously (hence the term *continuous integration*). But in many cases it is not as easy to test stored procedures as it is the other code that developers write. As such, in many cases it can become an impediment in advancing to best-in-class product development lifecycle steps such as continuous integration and continuous deployment (sometimes also called continuous delivery).

Using the database as a metaphor, we can apply this separation of dissimilar services to other pieces of our architecture. We very likely have back-office systems that perform functions like e-mail sending and receiving (non-platform-related), general ledger and other accounting activities, marketing segmentation, customer support operations, and so on. In each of these cases, we may be enticed to simply bolt these things onto our platform. Perhaps we want a transaction purchased in our e-commerce system to immediately show up in our CFO's enterprise resource planning (ERP) system. Or maybe we want it to be immediately available to our customer support representatives in case something goes wrong with the transaction. Similarly, if we are running an advertising platform we might want to analyze data from our data warehouse in real time to suggest even better advertising. There are a number of reasons why we might want to mix our business-process-related systems with our product platform. We have a simple answer: Don't do it.

Ideally we want these systems to scale independently relative to their individual needs. When these systems are tied together, each of them needs to scale at the same rate as the system making requests of them. In some cases, as was the case with our database performing business logic, the systems may be more costly to scale. This is often the case with ERP systems that have licenses associated with CPUs to run them. Why would we possibly want to increase our cost of scale by making a synchronous call to the ERP system for each transaction? Moreover, why would we want to reduce the availability of our product platform by adding yet another system in series as we discuss in Rule 38 (Chapter 9, "Design for Fault Tolerance and Graceful Failure")?

Just as product intelligence should not be placed on databases, business intelligence should not be tied to product transactions. There are many cases where we need that data resident within our product, and in those cases we should do just that—make it resident within the product. We can select data sets from these other systems and represent them appropriately within our product offering. Often this data will be best served with a new or different representation, sometimes of a different normal form. Very often we need to move data from our product back to our business systems such as in the case of customer support systems, marketing systems, data warehouses, and ERP systems. In these cases, we will also likely want to summarize and/or represent the data differently. Furthermore, to increase our availability we will want these pieces of data moved asynchronously back and forth between the systems. ETL systems are widely available for such purposes, and there are even open-source tools available to allow you to build your own ETL processes.

And remember that asynchronous does not mean "old" or "stale" data. For instance, you can select data elements over small time intervals and move those between your systems. Additionally, you can always publish the data on some sort of message bus for use on these other systems. The lowest-cost solution is batch extraction, but if temporal constraints don't allow such cost-efficient movement, then message buses are absolutely an appropriate solution. Just remember to revisit our rules on message buses and asynchronous transactions in Chapter 11, "Asynchronous Communication and Message Buses."

Finally, by separating product and business intelligence in your platform, you can also separate the teams that build and support those systems. If a product team is required to understand how their changes impact all related business intelligence systems, it will likely slow down their pace of innovation as it significantly broadens the scope when implementing and testing product changes and enhancements.

Rule 31—Be Aware of Costly Relationships

> ### Rule 31: What, When, How, and Why
>
> **What:** Be aware of relationships in the data model.
>
> **When to use:** When designing the data model, adding tables/columns, or writing queries, consider how the relationships between entities will affect performance and scalability in the long run.
>
> **How to use:** Think about database splits and possible future data needs as you design the data model.
>
> **Why:** The cost of fixing a broken data model after it has been implemented is likely 100x as much as fixing it during the design phase.
>
> **Key takeaways:** Think ahead and plan the data model carefully. Consider normalized forms, how you will likely split the database in the future, and possible data needs of the application.

In our personal lives we all generally strive to establish and build relationships that are balanced. Ideally we put into a relationship roughly the same as what we get out. When a personal relationship becomes skewed in one person's favor, the other person may become unhappy, reevaluate the relationship, and potentially end it. Although this book isn't about personal relationships, the same cost = benefit balance that exists in our personal relationships is applicable to our database relationships.

Database relationships are determined by the data model, which captures the cardinality and referential integrity rules of the data. To understand how this occurs and why it is important, we need to understand the basic steps involved in building a data model that results in the data definition language (DDL) statements that are used to actually create the physical structure to contain the data, that is, tables and columns. While there are all types of variations on this process, for a relational model the first step generally is to define the entities.

An entity is anything that can exist independently such as a physical object, event, or concept. Entities can have relationships with each other, and both the entity and the relationship can have attributes describing them. Using the common grammar analogy, entities are nouns, relationships are verbs, and attributes are adjectives or adverbs, depending on what they modify.

Entities are single instances of something, such as a customer's purchase order, which can have attributes such as an order ID and total value. Grouping the same types of entities together produces an entity set. In our database the entity is the equivalent of the row, and the entity set is the table. The unique attribute that describes the entity is the primary key of the table. Primary keys enforce entity integrity by uniquely identifying entity instances. The unique attributes that describe the relationship between entities are the foreign keys. Foreign keys enforce referential integrity by completing an association between two entities of different entity sets. Most commonly used to diagram entities, relationships, and attributes are entity relationship diagrams (ERDs). ERDs show the cardinality between entity sets, one-to-one, one-to-many, or many-to-many relationships.

Once the entities, relationships, and attributes are defined and mapped, the last step in the design of the data model is to consider normalization. The primary purpose of normalizing a data model is to ensure that the data is stored in a manner that allows for insert, update, select, and delete (aka CRUD: create, read, update, delete) with data integrity. Non-normalized data models have a high degree of data redundancy, which means that the risk of data integrity problems is greater. Normal forms build upon each other, meaning that for a database to satisfy the requirements for second normal form it first must satisfy those for first normal form. The most common normal forms are described in the sidebar. If a database adheres to at least the third normal form, it is considered normalized.

Normal Forms

Here are the most common normal forms used in databases. Each higher normal form implies that it must satisfy lower forms. Generally a database is said to be in normal form if it adheres to third normal form.

- **First normal form**—Originally, as defined by Codd,[1] the table should represent a relation and have no repeating groups. While "relation" is fairly well defined by Codd, the meaning of "repeating groups" is a source of debate. Controversy exists over whether tables are allowed to exist within tables and whether null fields are allowed. The most important concept is the ability to create a key.
- **Second normal form**—Nonkey fields cannot be described by only one of the keys in a composite key.
- **Third normal form**—All nonkey fields must be described by the key.
- **Boyce-Codd normal form**—Every determinant is a candidate key.
- **Fourth normal form**—A record type should not contain two or more multivalued facts.
- **Fifth normal form**—Every nontrivial join dependency in the table is implied by the candidate keys.
- **Sixth normal form**—No nontrivial join dependencies exist.

An easy mnemonic for the first three normal forms is "1—The Key, 2—The Whole Key, and 3—Nothing but the Key."

As you have probably figured out by now, the relationships between entities dramatically affect how efficiently the data is stored, retrieved, and updated. They also play a large role in scalability as these relationships define how we are able to split or shard our database. If we are attempting to perform a Y axis split of our database by pulling out the order confirmation service, this might prove problematic if the order entity is extensively related to other entities. Trying to untangle this web of relationships is difficult after the fact. It is well worth the time spent up front in the design phase to save you 10x or 100x the effort when you need to split your databases.

One last aspect of data relationships that is important to scalability is how we join tables in our queries. This, of course, is also very much dictated by the data model but also by our developers who are creating reports, new pages in our applications, and so on. We won't attempt to cover the steps to query optimization in detail here, but suffice it to say that new queries should be reviewed by a competent DBA who is familiar with the data model and should be analyzed for performance characteristics prior to being placed into the production environment.

You have probably noticed that there is a relationship between a desire for increased data integrity through normalization and the degree to which relationships must be used in a database. The higher the normal form, the greater the number of potential relationships as we create tables specific to such things as repeating values. What was once taught as a law years ago in database design (moving up in normal form is good) is now seen as more of a trade-off in high-transaction system design. This trade-off is similar to the trade-offs between risk and cost, cost and quality, time and cost, and so on; specifically, a decrease in one side typically implies an increase in the other. Often to increase scale, we look to reduce normal forms. Remember as we discussed in Chapter 4, when scale is desired and ACID properties aren't necessary for your product, a NoSQL solution may be appropriate.

When SQL queries perform poorly because of the requirements to join tables, there are several alternatives. The first is to tune the query. If this doesn't help, another alternative is to create a view, materialized view, summary table, and so on that can preprocess the joins. Another alternative is to not join in the query but rather pull the data sets into the application and join in memory in the application. While this is more complex, it removes the processing of the join from the database, which is often the most difficult to scale, and puts it in the application server tier, which is easier to scale out with more commodity hardware. A final alternative is to push back on the business requirements. Often our business partners will come up with different solutions when it is explained that the way they have requested the report requires a 10% increase in hardware but the removal of a single column may make the report trivial in complexity and nearly equivalent in business value.

Rule 32—Use the Right Type of Database Lock

> ### Rule 32: What, When, How, and Why
>
> **What**: Be cognizant of the use of explicit locks and monitor implicit locks.
>
> **When to use**: Anytime you employ relational databases for your solution.
>
> **How to use**: Monitor explicit locks in code reviews. Monitor databases for implicit locks and adjust explicitly as necessary to moderate throughput. Choose a database and storage engine that allow flexibility in types and granularity of locking.
>
> **Why**: Maximize concurrency and throughput in databases within your environment.
>
> **Key takeaways**: Understand the types of locks and manage their usage to maximize database throughput and concurrency. Change lock types to get better utilization of databases, and look to split schemas or distribute databases as you grow. When choosing databases, ensure that you choose one that allows multiple lock types and granularity to maximize concurrency.

Locks are a fact of life within a database; they are the way in which databases allow concurrent users while helping to ensure the consistency and isolation components of the ACID properties of a database. But there are many different types of database locks, and even different approaches to implementing them. Table 8.2 offers a brief and high-level overview of different lock types supported in many different open-source and third-party proprietary database management systems. Not all of these locks are supported by all databases, and the lock types can be mixed. For instance, a row lock can be either explicit or implicit.

Table 8.2 **Lock Types**

Type of Lock	Description
Implicit	Implicit locks are those generated by the database on behalf of a user to perform certain transactions. These are typically generated when necessary for certain DML (data manipulation language) tasks.
Explicit	These are locks defined by the user of a database during the course of his or her interaction with entities within the database.
Row	Row-level locking locks a row in a table of a database that is being updated, read, or created.
Page	Page-level locking locks the entire page that contains a row or group of rows being updated.
Extent	Typically, these are locks on groups of pages. They are common when database space is being added.
Table	This locks an entire table (an entity within a database).
Database	This locks the entirety of entities and relationships within a database.

If you research DB locks, you will find many other types of locks. There are, depending on the type of database, key and index locks that work on the indices that you create over your tables. You may also find a discussion of column locking where some columns in certain rows and certain tables are locked. To our knowledge, few if any databases actually support this type of locking, and if it is supported it isn't used very frequently within the industry.

While locking is absolutely critical to the operations of a database to facilitate both isolation and consistency, it is obviously costly. Typically databases allow reads to occur simultaneously on data, while blocking all reads during the course of a write (an update or insertion) on an element undergoing an operation. Reads then can occur very fast, and many of them can happen at one time, whereas typically a write happens in isolation. The finer the granularity of the write operation, such as in the case of a single row, the more of these can happen within the database or even within a table at a time. Increasing the granularity of the object being written or updated, such as updating multiple rows at a time, may require an escalation of the type of lock necessary.

The size or granularity of lock to employ ultimately impacts the throughput of transactions. When updating many rows at a single time within a database, the cost of acquiring multiple row locks and the competition for these rows might result in fewer transactions per second than just acquiring a larger lock of a page, extent, or table. But if too large a lock is grabbed, such as a page when updating only a small number of rows, then transaction throughput will decrease while the lock is held.

Often a component of the database (commonly called an *optimizer* within many databases) determines what size of element should be locked in an attempt to allow maximum concurrency while ensuring consistency and isolation. In most cases, initially allowing the optimizer to determine what should be locked is your best course of action. This component of the database has more knowledge about what is likely to be updated than you do at the time of the operation. Unfortunately, these systems are bound to make mistakes, and this is where it is critical that we monitor our databases in production and make changes to our DML to make it more efficient as we learn from what happens in our production environments.

Most databases allow performance statistics to be collected that allow us to understand the most common locking conditions and the most common events causing transactions to wait before being processed. By analyzing this information historically, and by monitoring these events aggressively in the production environment, we can identify when the optimizer is incorrectly identifying the type of lock it should use and force the database to use an appropriate type of locking. For instance, if through our analysis we identify that our database is consistently using table locking for a particular table and we believe we would get greater concurrency out of row-level locking, we might be able to force this change.

Perhaps as important as the analysis of what is causing bottlenecks and what type of locking we should employ is the notion of determining if we can change the entity relationships to reduce contention and increase concurrency. This of course brings us back to the concepts we discussed in Chapter 2. We can, for instance, split our reads across multiple copies of the database and force writes to a single copy as in the X axis of scale (Rule 7). Or we can split up our tables across multiple databases based partially on contention such as in the Y axis of scale (Rule 8). Or we may just reduce table size by pulling out certain customer-specific data into multiple tables to allow the contention to be split across these entities such as in the Z axis of scale (Rule 9).

Finally, where we employ databases to do our work we should try to ensure that we are choosing the best solution. As we've said many times, we believe that you can scale nearly any product or service using nearly any set of technologies. That said, most decisions we make would have an impact on either our cost of operating our product or our time to market. There are, for example, some database storage engine solutions that limit the types of locks we can employ within the database and as a result limit our ability to tune our databases to maximize concurrent transactions. MySQL is an example where the selection of a storage engine such as MyISAM can limit you to table-level locks and, as a result, potentially limit your transaction throughput. When using something such as MySQL and the MyISAM storage engine, scaling really needs to be considered using the X, Y, and Z axes of scale.

Rule 33—Pass on Using Multiphase Commits

Rule 33: What, When, How, and Why

What: Do not use a multiphase commit protocol to store or process transactions.

When to use: Always pass or alternatively never use multiphase commits.

How to use: Don't use them; split your data storage and processing systems with Y or Z axis splits.

> **Why:** A multiphase commit is a blocking protocol that does not permit other transactions to occur until it is complete.
>
> **Key takeaways:** Do not use multiphase commit protocols as a simple way to extend the life of your monolithic database. They will likely cause it to scale even less and result in an even earlier demise of your system.

Multiphase commit protocols, which include the popular two-phase commit (2PC) and three-phase commit (3PC), are specialized consensus protocols. The purpose of these protocols is to coordinate processes that participate in a distributed atomic transaction to determine whether to commit or abort (roll back) the transaction.[2] Because of these algorithms' capability to handle system-wide failures of the network or processes, they are often looked to as solutions for distributed data storage or processing.

The basic algorithm of 2PC consists of two phases. The first phase, the voting phase, is where the master storage or coordinator makes a "commit request" to all the cohorts or other storage devices. All the cohorts process the transaction up to the point of committing and then acknowledge that they can commit or vote "yes." Thus begins the second phase or completion phase, where the master sends a commit signal to all cohorts that begin the commit of the data. If any cohorts should fail during the commit, then a rollback is sent to all cohorts and the transaction is abandoned. An example of this protocol is shown in Figure 8.1.

So far this protocol probably sounds pretty good since it provides atomicity of transactions within a distributed database environment. Hold off on your judgment just a short while longer. In the example in Figure 8.1, notice that the app server initiated the transaction, step A. Then all the 2PC steps started happening and had to complete, step B, before the master database could acknowledge back to the app server that indeed that transaction was completed, step C. During that entire time the app server thread was held up

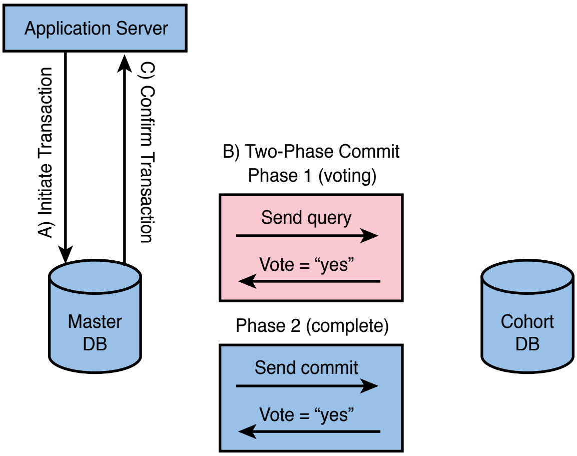


Figure 8.1 2PC example

waiting for the SQL query to complete and the database to acknowledge the transaction. This example is typical of almost any consumer purchase, registration, or bidding transaction on the Web where you might try to implement 2PC. Unfortunately, locking up the app server for that long can have dire consequences. While you might think either that you have plenty of capacity on your app servers or that you can scale them out pretty cost-effectively since they are commodity hardware, the locking also occurs on the database. Because you're committing all rows of data—assuming you have row-level locking capabilities because it's even worse for block level—you are locking up all those rows until everything commits and gives the "all clear." Furthermore, if the coordinator fails permanently, some of the cohorts will never resolve their transactions, leaving locks in place. You can see how such scenarios would create a costly failure of your systems.

As we discussed in the introduction of this chapter, we've implemented (or rather failed to implement) 2PC on a large scale, and the results were disastrous and entirely due to the lock and wait nature of the approach. Our database could initially handle thousands of reads and writes a second prior to the 2PC implementation. After introducing 2PC for just a fraction of the calls (less than 2%), the site completely locked up before processing a quarter of the total number of transactions it could previously handle. While we could have added more application servers, the database was not able to process more queries because of locks on the data.

While 2PC might seem like a good alternative to actually splitting your database by a Y or Z axis split (Rules 8 and 9), think again. Pull apart (or separate) database tables the smart way instead of trying to extend the life of your monolithic database with a multiphase commit protocol.

Rule 34—Try Not to Use `Select for Update`

> ### Rule 34: What, When, How, and Why
>
> **What:** Minimize the use of the `FOR UPDATE` clause in a `SELECT` statement when declaring cursors.
>
> **When to use:** Always.
>
> **How to use:** Review cursor development and question every `SELECT FOR UPDATE` usage.
>
> **Why:** Use of `FOR UPDATE` causes locks on rows and may slow down transactions.
>
> **Key takeaways:** Cursors are powerful constructs that when properly used can actually make programming faster and easier while speeding up transactions. But `FOR UPDATE` cursors may cause long-held locks and slow transactions. Refer to your database documentation to determine whether you need to use the `FOR READ ONLY` clause to minimize locks.

When leveraged properly, cursors are powerful database control structures that allow us to traverse and process data within some result set defined by the query (or operation) of the cursor. Cursors are useful when we plan to specify some set of data and "cursor through" or process the rows in the data set in an iterative fashion. Items within the data set can be updated, deleted, or modified or simply read and reviewed for other processing.

The real power of the cursor is as an extension of the capability of the programming language, as many procedural and object-oriented programming languages don't offer built-in capabilities of managing data sets within a relational database. One potentially troublesome approach in very high-transaction systems is the FOR UPDATE clause in SELECT cursors as we often are not in control of how long the cursor will be active. The resulting lock on records can cause slowdowns or even near-deadlock scenarios in our product.

In many databases, when the cursor with a FOR UPDATE clause is opened, the rows identified within the statement are locked until either a commit or a rollback is issued within the session. The COMMIT statement saves changes and a ROLLBACK cancels any changes. With the issuing of either statement, the locks associated with the rows in the database are released. Additionally, after issuing the commit or rollback, you lose your position within the cursor and will not be able to execute any more fetches against it.

Pause for a second now and think back to Rule 32 and our discussion of database locks. Can you identify at least two potential problems with the "select for update" cursor? The first problem is that the cursor holds locks on rows within the database while you perform your actions. Granted, in many cases this might be useful, and in some smaller number of cases it either might be unavoidable or may be the best approach for the solution. But these locks are going to potentially cause other transactions to block or wait while you perform some number of actions. If these actions are complex or take some time, you may stack up a great number of pending transactions. If these other transactions also happen to be cursors expecting to perform a "select for update," we may create a wait queue that simply will not be processed within our users' acceptable time frame. In a Web environment, impatient users waiting on slowly responding requests may issue additional requests with the notion that perhaps the subsequent requests will complete more quickly. The result is a disaster of cascading failures; our systems come to a halt as pending requests stack up on the database and ultimately cause our Web servers to fill up their TCP ports and stop responding to users.

The second problem is the mirror image of our first problem and was hinted at previously. Future cursors desiring a lock on one or more rows that are currently locked will wait until other locks clear. Note that these locks don't necessarily need to be placed by other cursors; they can be explicit locks from users or implicit locks from the RDBMS. The more locking that we have going on within the database, even while some of it is likely necessary, the more likely it is that we will have transactions backing up. Very long-held locks will engender slower response times for frequently requested data. Some databases, such as Oracle, include the optional keyword NOWAIT that releases control back to the process to perform other work or to wait before trying to reacquire the lock. But if the cursor must be processed for some synchronous user request, the end result to the user is the same—a long wait for a client request.

Be aware that some databases default to "for update" for cursors. In fact, the American National Standards Institute (ANSI) SQL standard indicates that any cursor should default to FOR UPDATE unless it includes the clause FOR READ ONLY on the DECLARE statement. Developers and DBAs should refer to their database documentation to identify how to develop cursors with minimal locks.

Rule 35—Don't Select Everything

> ### Rule 35: What, When, How, and Why
>
> **What:** Don't use `Select *` in queries.
>
> **When to use:** Always use this rule (or, put another way, never select everything).
>
> **How to use:** Always declare what columns of data you are selecting or inserting in a query.
>
> **Why:** Selecting everything in a query is prone to break things when the table structure changes and it transfers unneeded data.
>
> **Key takeaways:** Don't use wildcards when selecting or inserting data.

This is a pretty simple and straightforward rule. For most of us the first SQL we learned was

```
Select * from table_name_blah;
```

When it returned a bunch of data, we were thrilled. Unfortunately, some of our developers either never moved beyond this point or regressed to it over the years. Selecting everything is fast and simple but really never a good idea. There are several problems with this that we'll cover, but keep in mind that this mentality of selecting unnamed data can be seen in another DML statement, `Insert`.

There are two primary problems with the `Select *`. The first is the probability of data mapping problems, and the second is the transfer of unnecessary data. When we execute a select query, we're often expecting to display or manipulate that data, and to do so requires that we map the data into some type of variable. In the following code example there are two functions, `bad_qry_data` and `good_qry_data`. As the names should give away, `bad_qry_data` is a bad example of how you can map a query into an array, and `good_qry_data` shows a better way of doing it. In both functions, we are selecting the values from the table `bestpostpage` and mapping them into a two-dimensional array. Since we know there are four columns in the table, we might feel that we're safe using the `bad_qry_data` function. The problem is that when the next developer needs to add a column to the table, he or she might issue a command such as this:

```
ALTER TABLE bestpostpage ADD remote_host varchar(25) AFTER id;
```

The result is that your mapping from column 1 is no longer the `remote_ip` but instead is now `remote_host`. A better solution is to simply declare all the variables that you are selecting and identify them by name when mapping.

```
function bad_qry_data() {
[em]$sql = "SELECT * "
[em][em]. "FROM bestpostpage ".
[em]"ORDER BY insert_date DESC LIMIT 100";
[em]$qry_results = exec_qry($sql);
[em]$i = 0;
[em]while($row = mysql_fetch_array($qry_results)) {
[em][em]$ResArr[$i]["id"] = $row[0];
[em][em]$ResArr[$i]["remote_ip"] = $row[1];
[em][em]$ResArr[$i]["post_data"] = $row[2];
[em][em]$ResArr[$i]["insert_date"] = $row[3];
[em][em]$i++;
[em]} // while
[em]return $ResArr;
} //function qry_data

function good_qry_data() {
[em]$sql = "SELECT id, remote_ip, post_data, insert_date "
[em][em]. "FROM bestpostpage "
[em][em]. "ORDER BY insert_date DESC LIMIT 100";
[em]$qry_results = exec_qry($sql);
[em]$i = 0;
[em]while($row = mysql_fetch_assoc($qry_results)) {
[em][em]$ResArr[$i]["id"] = $row["id"];
[em][em]$ResArr[$i]["remote_ip"] = $row["remote_ip"];
[em][em]$ResArr[$i]["post_data"] = $row["post_data"];
[em][em]$ResArr[$i]["insert_date"] = $row["insert_date"];
[em]$i++;
[em]} // while
[em]return $ResArr;
} //function qry_data
```

The second big problem with `Select *` is that usually you don't need all the data in all the columns. While the actual lookup of additional columns isn't resource consuming, all that additional data being transferred from the database server to the application server can add up to a significant amount when that query gets executed dozens or even hundreds of times per minute for different users.

Lest you think this is all about the much-maligned `Select` statement, `Insert` can fall prey to the exact same problem of unspecified columns. The following SQL statement is perfectly valid as long as the column count of the table matches the number of values being entered. This will break when an additional column is added to the table, which might cause an issue with your system but should be caught early in testing.

```
INSERT INTO bestpostpage VALUES (1, '10.97.23.45', 'test
data', '2010-11-19 11:15:00');
```

A much better way of inserting the data is to use the actual column names, like this:

```
INSERT INTO bestpostpage (id, remote_ip, post_data,
insert_date) VALUES (1, '10.97.23.45', 'test data',
'2010-11-19 11:15:00');
```

As a best practice, do not get in the habit of using `Select` or `Insert` without specifying the columns. Besides wasting resources and being likely to break or potentially even corrupt data, it also prevents you from rolling back. As we discussed in Rule 29 (Chapter 7), building the capability to roll back is critical to both scalability and availability.

Summary

In this chapter we discussed rules that will help your database scale. Ideally, we'd like to avoid the use of relational databases because they are more difficult to scale than other parts of systems, but sometimes their use is unavoidable. Given that the database is often the most difficult part of the application to scale, particular attention should be paid to these rules. When the rules presented in this chapter are combined with rules from other chapters such as Chapter 2, you should have a strong base of dos and don'ts to ensure that your database scales.

Notes

1. E. F. Codd, "A Relationship Model of Data for Large Shared Data Banks," 1970, www.seas.upenn.edu/~zives/03f/cis550/codd.pdf.
2. Wikipedia, "Two-Phase Commit Protocol," http://en.wikipedia.org/wiki/Two-phase_commit_protocol.

9

Design for Fault Tolerance and Graceful Failure

Chances are that you've heard of the UK newspaper *The Guardian*. You may know it for breaking famous stories such as the Edward Snowden leaks, or the 2011 phone-hacking scandal associated with Rupert Murdoch's News International. What you probably don't know is that the online product and technology teams have won a number of awards themselves and are considered in many circles to be among the best product teams within the United Kingdom.

Guardian News and Media, part of the Guardian Media Group, is wholly owned by Scott Trust Limited, which exists to secure the financial and editorial independence of *The Guardian* in perpetuity. The group has as one of its core goals a significant growth of online readership. To be successful, the team believes it needs to build a superior online media delivery platform.

Enter Tanya Cordrey, formerly chief digital officer of *The Guardian* (until mid-2015); Grant Klopper, director of engineering; and Graham Tackley, director of data technology, then members of the *Guardian* product and engineering executive team. We interviewed Grant and asked him to tell the *Guardian* story.

"The situation in 2008 was a bit odd. The solution we were running *The Guardian* on was a very 'enterprisey' set of software—a very complex code base separated into a number of services coupled with a monolithic database. It worked well for quite some time, but with Internet growth you just never know what will happen. We have one really big article and traffic can double or more from previous peaks in a day. The problem is that when something breaks, everything starts to break. It's like the entire product had one giant fuse—and when that fuse blew, it would just take everything with it.

"It's important to understand," Grant continued, "how critical our online strategy is to our success. Newspaper circulations—I'm talking about printed newspapers—have been declining for quite some time. We have to make up for that with readership online. Failing to do so means we may not live up to the goal of *The Guardian* trust to ensure independent journalism in perpetuity. We have to be great. But in 2008, the solution we had wasn't so great. Big-traffic days could bring down absolutely everything. Furthermore, small failures in software or hardware could have a similar effect. One site, but operating on one fuse.

"Tanya Cordrey reached out to her former eBay colleagues, now helping companies as the growth consulting firm AKF Partners. With AKF's help we started redesigning aspects of the site to add fault isolation zones or 'swim lanes.' Basically, we wanted to take our single fuse in our fuse box and ensure that we had multiple fuses. Anytime one would break, only portions of the site would fail. This fault isolation approach allows us to separate software and database components into independent swim lanes, each of which is completely independent of other swim lanes. Different content areas of our site can be delivered from different swim lanes. If one of them fails, like the weather section, we can continue to deliver time-critical news. Furthermore, we can spend differently on high availability for each of the swim lanes. Swim lanes like news can get significantly greater redundancy than solutions of lower importance like weather. In the old solution, everything had the same availability and same costs; now we can make trade-offs to make news even more highly available without significantly increasing our costs."

Grant continued, "And the results have been nothing short of spectacular. If you went back to 2008 here, you would see that the editorial staff was always concerned about whether our Web site would make it through any given big-traffic day. Fast-forward to today and we just had 14 million unique readers for our coverage of the November 2015 terrorist acts in Paris. The editorial staff here doesn't think twice about the online solution; they know it will just work.

"And it's not just our Web site that receives this level of attention. When you operate a product on the Internet, you rely on any number of tools and unique infrastructure to help you deliver your product." Grant concluded, "We now apply these same concepts to the way in which we monitor, release solutions, et cetera. Everything that is critical has its own set of swim lanes such that a failure in our tooling won't bring all of our other tools and monitors down."

Grant and *The Guardian*'s needs for fault isolation and fault containment are not unique. Our experience is that most engineering teams are very good at, and are very focused on, delivering systems that work properly. Additionally, most engineers understand that it is impossible to create defect-free solutions and that as a result it is impossible to create a system that will not fail. Even with this in mind, very few engineers spend any significant time outlining and limiting the "blast radius" of any given failure.

When a system is under incredible load, say during peak times of demand, even small failures of certain features can back up transactions and bring the whole product to its knees.

In our business, availability and scalability go hand in hand. A product that isn't highly available really doesn't need to scale, and a site that can't scale won't be highly available when the demand comes. Because we can't keep systems from failing, we have to spend time trying to limit the impact of failures across our systems. This chapter offers several approaches to limiting the effect of failures, reducing the frequency of failures, and in general increasing the overall availability of the products you produce.

Rule 36—Design Using Fault-Isolative "Swim Lanes"

Rule 36: What, When, How, and Why

What: Implement fault isolation zones or *swim lanes* in your designs.

When to use: Whenever you are beginning to split up persistence tiers (e.g., databases) and/or services to scale.

> **How to use:** Split up persistence tiers and services along the Y or Z axis and disallow synchronous communication or access between fault-isolated services and data.
>
> **Why:** Increase availability and scalability. Reduce both incident identification and resolution. Reduce time to market and cost.
>
> **Key takeaways:** Fault isolation consists of eliminating synchronous calls between fault isolation domains, limiting asynchronous calls and handling synchronous call failure, and eliminating the sharing of services and data between swim lanes.

Our terminology in splitting up services and data is rich with confusing and sometimes conflicting terms. Different organizations often use words such as *pod*, *pool*, *cluster*, and *shard*. Adding to this confusion is that these terms are often used interchangeably within the same organization. In one context, a team may use *shard* to identify groupings of services and data, whereas in another it only means the separation of data within a database. Given the confusion and differentiation in usage of the existing terms, we created the term *swim lane* in our practice to try to hammer home the important concept of fault isolation. While some of our clients started adopting the term to indicate fault-isolative splits of services or customer segmentation in production, its most important contribution is in the design arena. Table 9.1 is a list of common terms, their most common descriptions, and an identification of how and when they are used interchangeably in practice.

Table 9.1 **Types of Splits**

Split Name	Description
Pod	Pods are self-contained sets of functionality containing app servers, persistent storage (such as a database or other persistent and shared file system), or both. Pods are most often splits along the Z axis, as in a split of customers into separate pods. *Pod* is sometimes used interchangeably with the term *swim lane*. It has also been used interchangeably with the term *pool* when referring to Web or application services.
Cluster	Clusters are sometimes used for Web and application servers in the same fashion as a “pool,” identified next. In these cases a cluster refers to an X axis scale of similar functionality or purpose configured such that all nodes or participants are “active.” Often a cluster will share some sort of distributed state above and beyond that of a pool, but this state can cause scalability bottlenecks under high transaction volumes. Clusters might also refer to active/passive configuration where one or more devices sit “passive” and become “active” on the failure of a peer device.
Pool	Pools are servers that group similar functionality or potentially separate groups of customers. The term typically refers to front-end servers, but some companies refer to database service pools for certain characteristics. Pools are typically X-axis-replicated (cloned) servers that are demarcated by function (Y axis) or customer (Z axis).

Continues

Table 9.1 **Types of Splits (*Continued*)**

Split Name	Description
Shard	Shards are horizontal partitions of databases or search engines. Horizontal partitioning means the separation of data across database tables, database instances, or physical database servers. Shards typically occur along the Z axis of scale (for instance, splitting up customers among shards), but some companies refer to functional (Y axis) splits as shards as well.
Swim lane	S*wim lane* is a term used to identify a fault isolation domain. Synchronous calls are never allowed across swim lane boundaries. Put another way, a swim lane is defined around a set of synchronous calls. The failure of a component within one swim lane does not affect components in other swim lanes. As such, no component is shared across swim lanes. A swim lane is the smallest boundary across which no synchronous calls occur.

From our perspective, the most important differentiation among these terms is that most are focused on a division of labor or transactions but only one is focused solely on limiting the propagation of failure. Whereas *pool*, *shard*, *cluster*, and *pod* might refer to either how something is implemented in a production environment or how one might divide or scale customers or services, *swim lane* is an architectural concept around creating fault isolation domains. A fault isolation domain is an area in which, should a physical or logical service fail to work appropriately, whether that failure is a slow response time or an absolute failure to respond, the only services affected are those within the failure domain. Swim lanes extend the concepts provided within shards and pods by extending the failure domain to the front door of your services—the entry into your data center. At the extreme it means providing separate Web, application, and database servers by function or fault isolation zone. At its heart, a swim lane is about both scalability and availability, rather than just a mechanism by which one can scale transactions.

We borrowed the concept from CSMA/CD (carrier sense multiple access with collision detection—commonly referred to as Ethernet), where fault isolation domains were known as collision domains. To offset the effects of collisions in the days before full duplex switches, Ethernet segments would contain collisions such that their effects weren't felt by all attached systems. We felt the term *swim lane* was a great metaphor for fault isolation as in swimming pools the lines between lanes of swimmers help keep those swimmers from interfering with each other during the course of their swim. Similarly, "lines" between groupings of customers, or functionality across which synchronous transactions do not occur, can help ensure that failures in one lane don't adversely affect the operations of other lanes.

The benefits of fault-isolative swim lanes go beyond the notion of increasing availability through the isolation of faults. Because swim lanes segment customers and/or functionality shared across customers, when failures occur you can more quickly identify the source. If you've performed a Z axis segmentation of your customers from your Web servers through your persistence tier, a failure that is unique to a single customer will quickly be isolated to the set of customers in that swim lane. You'll know you are looking for a bug or issue that is triggered by data or actions unique to the customers in that swim lane. If you've performed a Y axis segmentation and the "shopping cart" swim lane has a problem, you'll know immediately that the problem is associated with either the code, the database, or the servers constituting that swim lane. Incident detection and resolution as well as problem detection and resolution both clearly benefit from fault isolation.

Other benefits of fault isolation include better scalability, faster time to market, and lower cost. Because we focus on partitioning our systems, we begin to think of scaling horizontally, and hence our scalability increases. If we've separated our swim lanes by the Y axis of scale, we can separate our code base and make more efficient use of our engineers as discussed in Chapter 2, "Distribute Your Work." As such, we get better engineering throughput and therefore lower cost per unit developed. And if we are getting greater throughput, we are obviously delivering our product to market faster. Ultimately all of these benefits allow us to handle the "expected but unexpected": those things that we know will happen sooner or later but for which we cannot clearly identify the impact. In other words, we know things are going to break; we just don't know what will break or when it will happen. Fault isolation allows us to more gracefully handle these failures. Table 9.2 summarizes the benefits of fault isolation (or swim lanes).

Having discussed why we should swim-lane or fault-isolate our product offerings, we turn our attention to the more important question of how to achieve fault isolation. We rely on four principles that both define swim lanes and help us in designing them. The first is that nothing is shared between swim lanes. We typically exempt major network components such as inbound border routers and some core routers but include switches unique to the service being fault isolated. It is also common to share some other

Table 9.2 **Fault Isolation Benefits**

Area	Benefit
Availability	Availability is increased because a failure within one failure domain does not impact other services (Y axis) or other customers (Z axis), depending on how the swim lane is architected.
Incident detection	Incidents are detected faster because fewer components or services need to be investigated during an event. Isolation helps identify what exactly is failing.
Scalability	Horizontal scale is achieved when fault-isolated services can grow independently of each other.
Cost	Development cost is reduced through higher engineer throughput achieved from focus and specialization.
Time to market	As throughput increases, time to market for functions decreases.

devices such as a very large switched storage area network or load balancers in a smaller site. Wherever possible, and within your particular cost constraints, try to share as little as possible. Databases and servers should never be shared. Because swim lanes are partially defined by a lack of sharing, the sharing of servers and databases is always the starting point for identifying where swim lane boundaries truly exist. Due to the costs of network gear and storage subsystems, these are sometimes shared across swim lanes during the initial phases of growth.

The second principle of swim lanes is that no synchronous calls happen between swim lanes. Because synchronous calls tie services together, the failure of a service being called can propagate to all other systems calling it in a synchronous and blocking fashion. Therefore, it would violate the notion of fault isolation if a failure of a service we hoped to be in one swim lane could cause the failure of a service in another swim lane.

The third principle limits asynchronous calls between swim lanes. While asynchronous calls are much less likely than synchronous calls to propagate failures across systems, there still exists an opportunity to reduce our availability with these calls. Sudden surges in requests may make certain systems slow, such as in the case of messages being generated subsequent to a denial-of-service attack. An overwhelming number of these messages may back up queues, start to fill up TCP ports, and even bring database processing of synchronous requests to a standstill if not properly implemented. Thus, we try to limit the number of these transactions crossing swim lane boundaries.

The last principle of swim lanes addresses how to implement asynchronous transmissions across swim lane boundaries when they are absolutely necessary. Put simply, anytime we are going to communicate asynchronously across a swim lane, we need the ability to "just not care" about the transaction. In some cases, we may time out the transaction and forget about it. Potentially we are just "informing" another swim lane of some action, and we don't care to get a response at all. In all cases we should implement logic to "wire off" or "toggle off" the communication based on a manual switch, an automatic switch, or both. Our communications should be able to be switched off by someone monitoring the system and identifying a failure (the manual switch) and should sense when things aren't going well and stop communicating (the automatic switch).

Thinking back to the preface to this book, recall that Rick Dalzell mentioned Amazon's first split was a separation of its store functionality from its fulfillment functionality. If for some reason the Amazon storefront went down, Amazon could continue to fulfill orders it had already received. If fulfillment went down, the store could continue to receive and queue up orders. Clearly the two needed to communicate with each other, but not "synchronously" from a customer perspective. Work could be passed from the front end to the back end in the form of orders to be fulfilled and from the back end to the front end in terms of order status updates. Such a split offers the benefits of both fault isolation and greater levels of scalability as each can be scaled independently.

These principles are summarized in Table 9.3.

Table 9.3 **Fault Isolation Principles**

Principle	Description
Share nothing	Swim lanes should not share services. Some sharing of network gear such as border routers and load balancers is acceptable. If necessary, storage area networks can be shared. Databases and servers should never be shared.
No synchronous calls between swim lanes	Synchronous calls are never allowed across swim lane boundaries. Put another way, a swim lane is the smallest unit across which no synchronous calls happen.
Limit asynchronous calls between swim lanes	Asynchronous calls should be limited across swim lanes. They are permitted, but the more calls that are made, the greater the chance of failure propagation.
Use time-outs and wire on/wire off with asynchronous calls	Asynchronous calls should be implemented with time-outs and the ability to turn off the call when necessary due to failures in other services. See Rule 39.

How about the case where we want fault isolation but need synchronous communication or access to another data source? In the former case, we can duplicate the service that we believe we need and put it in our swim lane. Payment gateways are one example of this approach. If we were to swim-lane along a Z axis by customer, we probably don’t want each of our customer swim lanes synchronously (blocking) calling a single payment gateway for a service like checkout. We could simply implement N payment gateways where N is the degree of customer segmentation or number of customer swim lanes.

What if there is some shared information to which we need access in each of these swim lanes such as in the case of login credentials? Maybe we have separated authentication and sign-in/login into its own Y axis split, but we need to reference the associated credentials from time to time on a read-only basis within each customer (Z axis) swim lane. We often use read replicas of databases for such purposes, putting one read replica in each swim lane. Many databases offer this replication technology out of the box and even allow you to slice it up into smaller pieces, meaning that we don’t need to duplicate 100% of the customer data in each of our swim lanes. Some customers cache relevant information for read-only purposes in distributed object caches within the appropriate swim lane.

One question that we commonly get is how to implement swim lanes in a virtualized server world. Virtualization adds a new dimension to fault isolation—the dimension of logical (or virtual) in addition to physical failures. If virtualization is implemented primarily to split up larger machines into smaller ones, you should continue to view the physical server as the swim lane boundary. In other words, don’t mix virtual servers from different swim lanes on the same physical device. Some of our customers, however, have such a variety of product offerings with varying demand characteristics throughout the year that they rely on virtualization as a way of flexing capacity across these product offerings. In these cases, we try to limit the number of swim lanes that are mixed on a virtual server. Ideally, you would flex an entire physical server in and out of a swim lane rather than mix swim lanes on that server.

> **Swim Lanes and Virtualization**
>
> When using virtualization to carve larger servers into smaller servers, attempt to keep swim lanes along physical server boundaries. Mixing virtual servers from different swim lanes on one physical server eliminates many of the fault-isolative benefits of a swim lane.

Rule 37—Never Trust Single Points of Failure

> **Rule 37: What, When, How, and Why**
>
> **What:** Never implement and always eliminate single points of failure.
>
> **When to use:** During architecture reviews and new designs.
>
> **How to use:** Identify single instances on architectural diagrams. Strive for active/active configurations.
>
> **Why:** Maximize availability through multiple instances.
>
> **Key takeaways:** Strive for active/active rather than active/passive solutions. Use load balancers to balance traffic across instances of a service. Use control services with active/passive instances for patterns that require singletons.

In mathematics, singletons are sets that have only one element {A}. In programming parlance, the singleton pattern is a design pattern that mimics the mathematical notion and restricts the instantiation of a class to only one object. This design pattern is useful for coordination of a resource but is often overused by developers in an effort to be expeditious—more on this later. In system architecture, the singleton pattern, or more aptly the singleton *antipattern*, is known as a single point of failure (SPOF). This is when there exists only one instance of a component within a system that when it fails will cause a system-wide incident.

SPOFs can be anywhere in the system from a single Web server or single network device, but most often the SPOF in a system is the database. The reason for this is that the database is often the most difficult to scale across multiple nodes and therefore gets left as a singleton. In Figure 9.1, even though there are redundant login, search, and checkout servers, the database is a SPOF. What makes it worse is that all the service pools are reliant on that single database. While any SPOF is bad, the bigger problem with a database as a SPOF is that if the database slows down or crashes, all service pools with synchronous calls to that database will also experience an incident.

We have a mantra that we share with our clients, and it is simply "Everything fails." This goes for servers, storage systems, network devices, and data centers. If you can name it, it can fail, and we've probably seen it happen. While most people think of data

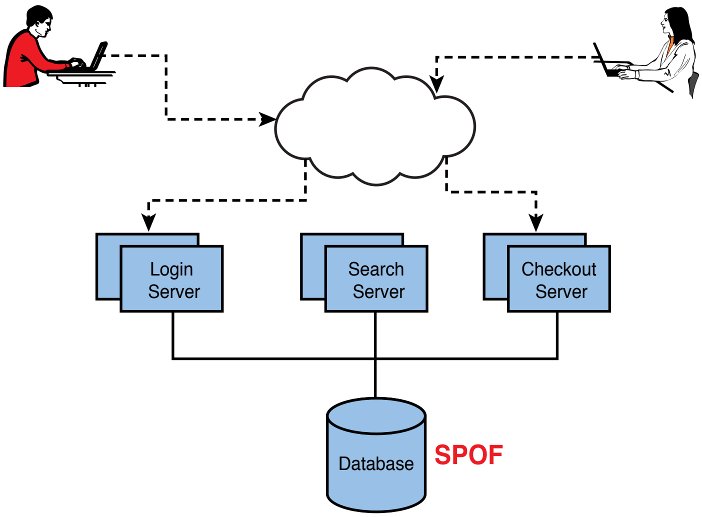


Figure 9.1 Database SPOF

centers as never failing, we have personal experience with more than a dozen data center outages in as many years. The same goes for highly available storage area networks. While they are remarkably more reliable than the old SCSI disk arrays, they still can and do fail.

The solution to most SPOFs is simply requisitioning another piece of hardware and running two or more of every service by cloning that service as described in our X axis of scale. Unfortunately, this isn't always so easy. Let's retrace our steps to the programming singleton pattern. While not all singleton classes will prevent you from running a service on multiple servers, some implementations absolutely will prevent you from this without dire consequences. As a simplified example, if we have a class in our code that handles the subtraction of funds from a user's account, this might be implemented as a singleton to prevent unpleasant things from happening to a user's balance such as going negative. If we place this code on two separate servers without additional controls or semaphores, it is possible that two simultaneous transactions will attempt to debit a user's account, which could lead to erroneous or undesired conditions. In this case we need to either fix the code to handle this condition or rely on an external control to prevent it. While the most desirable solution is to fix the code so that the service can be implemented on many different hosts, often we need an expeditious fix to remove a SPOF. As the last focus of this rule, we'll discuss a few of these quick fixes next.

The first and simplest solution is to use an active/passive configuration. The service would run actively on one server and passively (not taking traffic) on a second server. This hot/cold configuration is often used with databases as a first step in removing a SPOF. The next alternative would be to use another component in the system to control the access to data. If the SPOF is the database, a master/slave configuration can be set up, and the application can control access to the data with writes/updates going to the master and reads/selects going to the slave. In addition to mitigating the SPOF, introducing read-only replicas of databases with high read-to-write ratios will reduce the load on your master database and allow you to take advantage of more cost-effective hardware as discussed in Rule 11, "Use Commodity Systems (Goldfish Not Thoroughbreds)" (see Chapter 3, "Design to Scale Out Horizontally"). A last configuration that can be used to fix a SPOF is a load balancer. If the service on a Web or application server is a SPOF and cannot be eliminated in code, the load balancer can often be used to fix a user's request to only one server in the pool. This is done through session cookies, which are set on the user's browser and allow the load balancer to redirect that user's requests to the same Web or application server each time, resulting in a consistent state.

We covered several alternative solutions to SPOFs that can be implemented quickly when code changes cannot be made in a timely manner. While the best and final solution should be to fix the code to allow for multiple instances of the service to run on different physical servers, the first step is to eliminate the SPOF as soon as possible. Remember, "Everything fails," so don't be surprised when your SPOF fails.

Rule 38—Avoid Putting Systems in Series

> ### Rule 38: What, When, How, and Why
>
> **What:** Reduce the number of components that are connected in series.
>
> **When to use:** Anytime you are considering adding components.
>
> **How to use:** Remove unnecessary components, collapse components, or add multiple parallel components to minimize the impact.
>
> **Why:** Components in series are subject to the multiplicative effect of failure.
>
> **Key takeaways:** Avoid adding components to your system that are connected in series. When it is necessary to do so, add multiple versions of that component so that if one fails, others are available to take its place.

Components in electrical circuits can be connected in a variety of ways; the two simplest are series and parallel. Circuits in series have components (these may be capacitors, resistors, or other components) that are connected along a single path. In this type of circuit the current flows through every component, and the resistance and voltage are additive. Figure 9.2 shows two circuits, one with three resistors and one with three batteries, and the resulting resistance and voltage. Notice that in this diagram if any of the components fails, such as a resistor blows, then the entire circuit fails.

Figure 9.3 shows two parallel circuits, the top one with three resistors (and a voltage source or capacitor) and the bottom one with three batteries. In this circuit, the total resistance is calculated by summing the reciprocals of each resistor and then taking the reciprocal of that sum. The total resistance by definition must be less than the smallest resistance. Notice also that the voltage does not change, but instead the batteries contribute only a fraction of the current, which has the effect of extending their useful life.

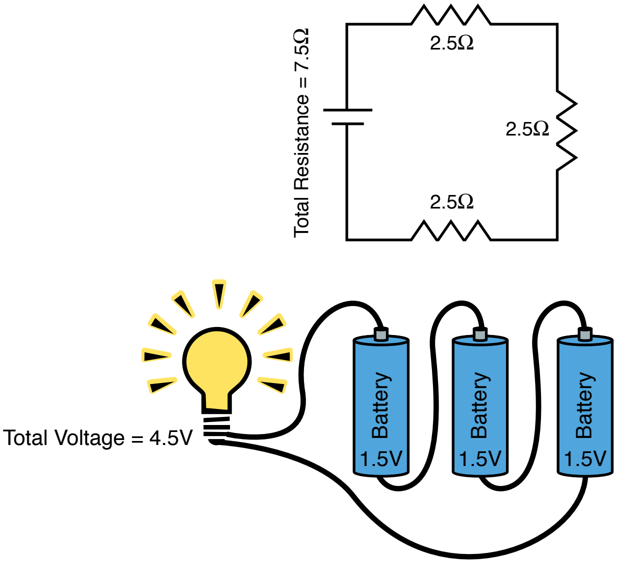


Figure 9.2 Circuits in series

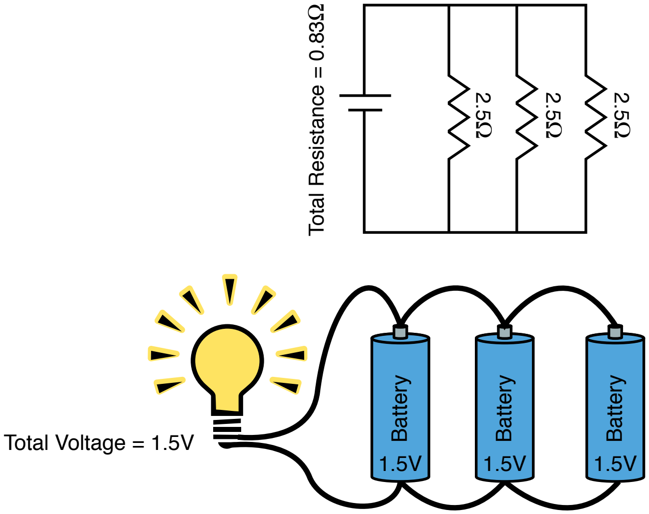


Figure 9.3 Circuits in parallel

Notice that in these circuits the failure of a component does not cause a failure across the entire circuit.

The similarities between the architecture of your system and that of a circuit are many. Like the circuit, your system consists of varied components—Web and application servers, load balancers, databases, and network gear—all of which can be connected in parallel or in series. As a simple example, let's take a static Web site that has a lot of traffic. You could put ten Web servers all with the same static site on them to serve traffic. You could either use a load balancer to direct traffic or assign all ten separate IP addresses that you associate with your domain through DNS. These Web servers are connected in parallel just like the batteries in Figure 9.3. The total current or amount of traffic that one Web server has to handle is a fraction of the total, and if one Web server fails, the site remains available as there are still nine other Web servers.

As an example of a more typical architecture in series, let's add some layers. If we take a standard three-tier site that has a single Web server, one application server, and one database server, we have an architecture connected in series. For a request to be fulfilled, the Web server must accept it and pass a request to the application server, which queries the database. The application server then receives the data back, manipulates the data, and sends it back to the Web server, which finally fulfills the request to the customer. If any component in this circuit or architecture breaks, the entire system is down.

This brings us back to real-world architecture. Almost always there are going to be requirements to have components in series. When you take into consideration the load balancer, the Web and application tier, the database, the storage system, and so on, many components are required to keep your system running. Certainly adding components in parallel, even when tiers are connected in series, helps reduce the risk of total system failure caused by a component failure. Multiple Web servers spread the traffic load and prevent a system failure if only one Web server fails. On the Web and application tiers most people readily acknowledge this concept. Where most people overlook this problem is in the database and network layers. If Web and application servers connected in parallel all are connected in series to a single database, we can have a single component result in catastrophic failure as discussed in the previous rule, "Never Trust Single Points of Failure." This is why it is important to pay attention to the rules in Chapter 2 about splitting your database and in Chapter 3 about scaling horizontally.

In regard to the network components, we often see architectures that pay a lot of attention to connecting servers in parallel but completely ignore the network devices, especially firewalls. It is not uncommon to see firewalls inside and outside the network; see Rule 15 (Chapter 4, "Use the Right Tools") for further discussion regarding firewalls. In this case we have traffic going through a firewall, through a load balancer, through a firewall, through a switch, then to a Web server, an application server, a database server, and all the way back. There are at least seven components in series. So what's the big deal about adding another component if you're already going through a half-dozen?

Items in series have a *multiplicative effect* on risk of failure. As a simple example, if we have two servers in series, each with 99.9% availability or uptime, then our total availability of the system cannot be greater than 99.9% × 99.9% = 99.8%. If we add a third component, in series, at the same three-nines availability of 99.9%, we get an even lower availability of 99.9% × 99.9% × 99.9% = 99.7%. The more components that are placed in series, the lower the system's availability. Table 9.4 lists some simple calculations that demonstrate the lowered availability and the resulting increase in downtime per month.

Table 9.4 **Components in Series at 99.9% Availability**

No. of Components in Series	Total Availability	Minutes of Downtime per Month
1	99.9%	43.2
2	99.8%	86.4
3	99.7%	129.5
4	99.6%	172.5
5	99.5%	215.6
6	99.4%	258.6
7	99.3%	301.5
8	99.2%	344.4
9	99.1%	387.2

For every component (at 99.9% availability) that we add to the system in series, we are adding ~43 minutes of downtime per month. However, for every pair of components (at 99.9% availability each) that we add together in parallel, the downtime added drops down to less than a minute (~26 seconds) per month. This improvement is even more drastic when you parallelize components that have lower individual availability.

Because your system, just like most circuits today, is much more complicated than a simple series or parallel connection, the exact calculations for your expected availability are much more complicated than our simple example. However, what you can take away from this is that components in series significantly increase a system's risk of experiencing downtime. You can of course mitigate this by reducing the items in series or adding multiple numbers of those components, in parallel.

Rule 39—Ensure That You Can Wire On and Off Features

Rule 39: What, When, How, and Why

What: Create a framework to disable and enable features of your product.

When to use: For functionality that is newly developed, noncritical, or dependent on a third party, consider using a wire-on/wire-off framework.

How to use: Develop shared libraries to allow automatic or on-demand enabling and disabling of features. See Table 9.5 for recommendations.

Why: Turning off broken functionality or noncritical functionality in order to protect and preserve critical functionality is important for end users.

Key takeaways: Implement wire-on/wire-off frameworks whenever the cost of implementation is less than the risk and associated cost. Work to develop shared libraries that can be reused to lower the cost of future implementation.

We introduced the notion of what we call wire-on/wire-off frameworks in Chapter 7, "Learn from Your Mistakes," while discussing rollbacks and mentioned it again in this chapter while we were discussing fault isolation as a method of design. Ultimately these types of frameworks help to ensure that your systems can either fail gracefully (in the event of self-diagnosing frameworks) or operate with certain functionality disabled by human intervention. Sometimes companies refer to similar functionality as "feature toggles" or "circuit breakers."

There have been several approaches to wire on/wire off in the past, each with certain benefits and drawbacks. The approach to enabling and disabling services probably depends on the capabilities of your engineering team, your operations team, and the business criticality of the service in question. Table 9.5 covers some of these approaches.

Table 9.5 isn't meant to be an all-encompassing list of the possibilities for enabling and disabling functionality. In fact, many companies blend some of these options. They

Table 9.5 **Feature Wire-On/Wire-Off Approaches**

Approach	Description	Pro	Con
Automatic markdown based on time-outs	When calls to third parties become slow, the application logic will mark this functionality as "off" for a period of time or until manual intervention re-enables it.	Fastest way to mark down a service that might bring several other services down when slow or unavailable.	Sensitive to "false failures" or incorrect identification of a failing service. When coupled with auto markup, may cause a "pinging" effect of the service. Every service needs to make its own decision.
Stand-in service	Replace a service with an auto responder with a dummy response that indicates service unavailable or a cached response of "likely good data."	Easy to implement, at least on the service side. May allow user determination of failure.	Each calling service needs to understand the "failure" response. May be slower to mark down and may require user intervention to mark up.
Manual markdown command	Administrators manually send a signal/ issue a command to stop displaying the functionality that is performing slowly.	Allows manual determination of failed service.	Slower than automatic markdown. Also, if services have TCP ports full due to slow or failed service, the command may not work as desired.
Config file markdown	Change configuration variable in a file to indicate the "wire off" of a feature.	Doesn't rely on request/reply communication as in a command.	Likely requires restart of a service to reload configurations in memory.

Continues

Table 9.5 **Feature Wire-On/Wire-Off Approaches (*Continued*)**

Approach	Description	Pro	Con
File markdown	Presence of a file (or absence of a file) indicates if the functionality is available or not.	Doesn't rely on request/reply communication as in a synchronous command. Might not require service restart.	May slow down processing as each request requires "polling" for files.
Runtime variable	Read at startup as an argument to the program for daemon-like processes.	Similar to config file.	Similar to config file.

may read in variables from a database or a file at startup that direct the application code to display or not display certain sets of functionality. An example of this was implemented at PayPal when they first implemented internationalization. Some countries allowed only a limited set of payment functionality based on banking or money transfer regulations. Depending on the geographical location from which the user was accessing the site, he or she might see only a subset of the functionality provided on the main site.

Equally important issues to tackle when considering wire-on/wire-off frameworks for features/functionality are the decisions about where and when they should be used. Clearly implementing the framework represents additional work and as a result additional cost to the business. Let's start with the (unlikely and probably incorrect) position that certain features will never fail. If we could tell which features would never fail, we would never want to implement this functionality for those features because it represents a cost with no return. With that starting point, we can identify where this investment has value or provides the business with a return. Any feature that has a high rate of usage (high throughput) and whose failure will impact other important features on the site is a candidate. Another candidate is any feature that is undergoing fairly significant change in a given release. The idea in both of these areas is that the cost of implementing wire on/wire off is less than the risk (where risk is a function of both the likelihood of a failure and the impact of that failure) to our business. If the development of a feature with this functionality costs an extra $1,000 and an unhandled failure of the feature might cost us $10,000 in business, is the cost justified?

When done well, engineering teams can reduce the cost of implementing a wire-on/wire-off framework by implementing a set of shared libraries to be used across functions. This approach doesn't reduce the cost of implementing the framework to zero for any new development, but it does help to reduce the cost for future generations of framework-enabled features.

We recommend implementing wire-on/wire-off frameworks for a couple of key reasons. First, when the functionality is new or actively being developed, it is more likely to have bugs. Having the ability to mark down or turn off broken functionality is highly desirable. Second, if the functionality is not critical to the service you are providing, you may want the ability to mark down or turn off noncritical functions. When computational resources become constrained—perhaps you have a memory leak that puts your application into full garbage collection—being able to turn off noncritical functionality to preserve the more critical features is nice to have. Third, calls to third parties often have to be made synchronously. When the vendor's API begins to respond slowly, having the ability to turn off that functionality, preventing it from slowing your entire application or service, is very desirable. To be clear, we don't believe that everything should be capable of being enabled/disabled or wired on/off. Such an approach is costly and ill advised, but well-run teams should be able to identify risky and shared components in order to implement the appropriate safeguards.

Summary

We believe that availability and scalability go hand in hand. A product that isn't highly available doesn't need to scale because users will soon stop coming. A site that can't scale won't be highly available when the demand comes because the site will be slow or completely down. Because of this you can't work on one without thinking about the other. This chapter offered four rules that help ensure that your site stays highly available as well as continues to scale. Don't let a focus on scalability cause you to forget how important availability is to your customer.

10

Avoid or Distribute State

We met Rick Dalzell in the preface of this book as he described some of the journey and challenges of getting Amazon's commerce platform to scale to the transaction volumes it handles today. Rick and the Amazon team learned a lot about the costs associated with maintaining state in large distributed systems. Put simply, Rick's hard-earned advice is to avoid state wherever possible. "Everyone thinks of Amazon as one of the world's largest stateless engines," Rick stated. "In truth, there are notions of state built into the system. Order workflow, for instance, needs to have a notion of state as an order progresses from the shopping cart through fulfillment and finally shipping. But we learned early on that state also has a very high cost and that the cost associated with it increases significantly in a highly distributed system.

"When a request comes into the Amazon.com Web site, that request may in turn be farmed out to tens or hundreds of other processes, each of which does its work in a largely asynchronous fashion. Imagine the cost and complexity associated with each of these processes maintaining some notion of state. Coordination would become mind-boggling. The costs associated with compute time and storage for the state would increase significantly. Now, imagine transferring that state across each of these services. Now you are in this crazy game of synchronizing process states. Or you try to get smart and attempt to force consistent state between processes through two-phase commit.[1] All of these things are really, really costly in small systems; imagine their cost across hundreds, thousands, tens of thousands, and so on, servers and services.

"We tried all the ways you can think of to synchronize and manage state, and then we just decided we needed to avoid it. Here's the other thing: when you have a stateful system and something fails, now you have some real problems. How do you recover that state? How long are you willing to wait to allow it to fully recover?

"We learned this thing the hard way—that state is bad and should be avoided if at all possible. Luckily, we learned it during the early days of business on the Internet. It was painful to learn those things back then, but we had a bit more flexibility. These days, you simply can't fail and you can't be down. Business and consumer expectations have increased to near-unobtainable levels. Sometimes you can't get around having [state], such as in an order workflow, but wherever you can, you should avoid it."

Session and state destroy the ultimate value promised by multitenancy within Internet (SaaS, commerce, and so on) applications. The more data we retain that is associated with a user's interactions, the fewer users we can house on a system at any given time. In the desktop world, this was rarely a concern as we often had a lot of power and memory available for a single user at any given time. In the multitenant world, the goal is to house

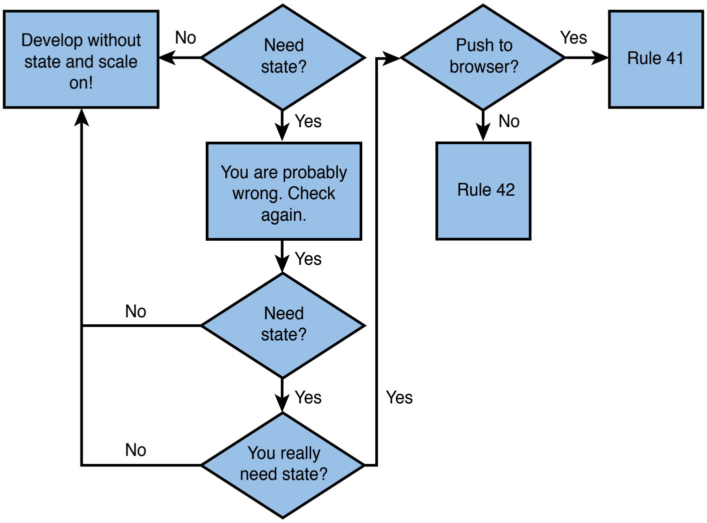


Figure 10.1 Decision flowchart for implementing state in a Web application

as many users as possible on a single system while still delivering a stellar user experience. As such, we strive to eliminate any approach that will limit the degree of tenancy on any system. State and session cost us in terms of both memory and processing power and as a result are enemies of our cost-effective scale goals.

While we would prefer to avoid state at all cost, it is sometimes valuable to our business. Indeed, the very nature of some applications (such as the order workflow system that Rick described) requires user state. If state is necessary, we need to implement it in a fault-tolerant, highly available, and cost-effective way such as distributing it to our end users (Rule 41) or positioning it on a special service within our infrastructure (Rule 42).

Figure 10.1 depicts diagrammatically both our feelings on state and how to approach decisions about how to implement state.

Rule 40—Strive for Statelessness

Rule 40: What, When, How, and Why

What: Design and implement stateless systems.

When to use: During design of new systems and redesign of existing systems.

How to use: Choose stateless implementations whenever possible. If stateful implementations are warranted for business reasons, refer to Rules 41 and 42.

Why: The implementation of state limits scalability, decreases availability, and increases cost.

Key takeaways: Always push back on the need for state in any system. Use business metrics and multivariate (or A/B) testing to determine whether state in an application truly results in the expected user behavior and business value.

Paradoxically it is both a sad and an exciting day when our applications grow beyond the ability of a single server to meet our transaction-processing needs. It is exciting because our business is growing and sad because we embark upon a new era of development requiring new skills to scale our systems. We can sometimes rely on session replication technologies to help us scale, but these approaches also have limits that are quickly outgrown. As we described in Chapter 2, "Distribute Your Work," you will soon find yourself replicating too much information in memory across too many application servers. Very likely you will need to perform a Y or Z axis split.

Many of our clients simply stop at these splits and rely on affinity maintained through a load balancer to handle session and state needs. Once a user logs in, or starts some flow specific to a pool of application servers, he or she maintains affinity to that application server until the function (in the case of a Y axis split where different pools provide different functions) or session (in the case of a Z axis split where customers are segmented into pools) is complete. This is an adequate approach for many products where growth has slowed or where customers have more relaxed availability requirements. Maintaining affinity drives increased cost; capacity planning can become troublesome when several high-volume or long-running sessions become bound to a handful of servers, and availability for certain customers will be impacted when the application server on which they are running fails. While it is possible to rely on session replication to create another host to which we might move in the event of a system failure, this approach is costly and requires duplicate memory consumption as well as increased system capacity.

Ultimately, the solution that serves a majority of our hyper-growth clients the best is to eliminate the notion of state wherever possible. We prefer to start discussions on the topic of state with "Why do you need it at all?" Our clients are often taken aback, and the typical response is "Well, that's the way it's always been and we need to know what just happened to make the next move." They are often at a loss when pressed to back up their responses with data showing the efficacy of state in terms of revenue, increased transaction volume, and so on. Granted, there are certain solutions that require state (e.g., workflow processes), but more often than not, state is a luxury and a costly one at that.

Never underestimate the power of "simple and easy" in an application as an effective weapon against "rich and complex." Craigslist won the local classifieds war against eBay with a largely text-based and stateless application. eBay, while striving to stay as stateless as possible, had a competing classifieds product with a significantly greater number of features and "richness" years ahead of rival Craigslist. Yet simple won the day. Not convinced? How about Google against all comers in the search market? While others invested in rich interfaces, Google at least initially built on the concept that your last search was the only thing that mattered, and what you really want is good search results. No state, no session, no nonsense.

The point is that session and state cost money, and you should implement them only where there is a clear competitive advantage backed up by operating metrics (determined through A/B or multivariate analysis). Session and state require memory and generally require greater code complexity, meaning longer-running transactions.

This reduces the number of transactions we can handle per second per server, which increases the number of systems that we need. The systems may also need to be larger and more costly given the memory requirements to house state on or off the systems. Potentially this requires developing "state farms" as we describe later in this chapter, which means more devices. In turn, more devices mean more space, power, and cooling or in the virtual world more cloud resources for which we are paying. Remember that every server (or virtual machine) we need costs us three times over, as we need to provide space for it, cool it, and power it. Cloud resources have the same costs; they are just passed on to us in a bundle.

It is best to always question the need for state in any application or service. Establish a strongly worded principle—something along the lines of "We develop stateless applications." Be clear that state distribution (the movement of state to the browser or distributed state server or cache) is not the same as being stateless. Rules 41 and 42 exist to allow us to create rich business functionality where there is a clear competitive advantage, displayed through operating metrics that drive revenue and transactions, not as an alternative to this rule.

Rule 41—Maintain Sessions in the Browser When Possible

Rule 41: What, When, How, and Why

What: Try to avoid session data completely, but when needed, consider putting the data in users' browsers.

When to use: Anytime that you need session data for the best user experience.

How to use: Use cookies to store session data in the users' browsers.

Why: Keeping session data in the users' browsers allows the user request to be served by any Web server in the pool and reduces storage requirements.

Key takeaways: Using cookies to store session data is a common approach and has advantages in terms of ease of scale. One of the most concerning drawbacks is that unsecured cookies can easily be captured and used to log in to people's accounts.

If you have to keep sessions for your users, one way to do so is in the users' browsers. There are several pros and cons of this approach. One benefit of putting the session data in a browser is that your system doesn't have to store the data. As explained in Rule 42, keeping session data within the system can amount to a large overhead of storage and retrieval. Not having to do this relieves the system of a large burden in terms of storage and workload. A second benefit of this approach is that the browser's request can be serviced by any server in the pool. As you scale your Web servers along the X axis (horizontally) with the session data in the browser, any server in the pool can handle the request.

Of course, everything has trade-offs. One of the cons of storing session state in the browser is that the data must be transferred back and forth between the browser and the servers that need this data. Moving this data back and forth for every request can be expensive, especially if the amount of data starts to become significant. Be careful not to dismiss this last statement too quickly. While your session data might not be too large now, let a couple of dozen developers have access to storing data in cookies, and after a couple of code releases you will be wondering why the pages are loading so slowly. Another very serious con that was brought to light by the Firefox plug-in Firesheep is that session data can be easily captured on an open WiFi network and used to nefariously log in to someone else's account. With the aforementioned plug-in, session cookies from most of the popular sites such as Google, Facebook, Twitter, and Amazon, just to name a few, can be compromised. A little later we suggest a way to protect your users' cookies against this type of hack or attack, commonly called *sidejacking*.

Storing cookies in browsers is simple and straightforward. In PHP, as shown in the following example, it is as simple as calling `setcookie` with the parameters of the cookie name, value, expiration, path, domain, and secure (whether it should be set only over HTTPS). To destroy the cookie when you're done with it, just set the same cookie but with the expire parameter set to `time()-3600`, which sets the expiration time to one hour ago.

```
setcookie("SessionCookie", $value, time()+3600, '/', '.akf
partners.com', true);
```

Some sessions are stored in multiple cookies, and other session data is stored in a single cookie. One factor to consider is the maximum size of cookies. According to RFC 2965, browsers should support cookies at least up to 4KB in size and up to 20 cookies from the same domain.[2] Most browsers, however, support these as maximums. To our earlier point, the larger the cookie, the slower your pages will load since this data has to be transmitted back and forth with each request.

Now that we're using cookies to support our sessions and we're keeping them as small as possible so that our system can scale, the next question is how do we protect them from being sidejacked? Obviously you can transmit your pages and cookies all in HTTPS. The SSL protocol used for HTTPS requires encrypting and decrypting all communication and requests. While this might be a requirement for a banking site, it doesn't make sense for a news or social networking site. Instead we recommend a method using at least two cookies. One cookie is an authorization cookie that is requested via HTTPS on each HTTP page using a JavaScript call such as the following. This allows the bulk of the page (content, CSS, scripts, and so on) to be transferred by unsecure HTTP but a single authorization cookie to be transferred via HTTPS.[3]

```
<script type="text/javascript" src="https://verify.akfdemo.
com/authenticate.php"></script>
```

For ultimate scalability we recommend avoiding sessions altogether. However, this isn't always possible. In these cases we recommend storing the session data in the user's browser. When implementing this, it is critical to maintain control of the size of the cookie data. Excessive amounts of data slow the performance of the page load as well as the Web servers on the system.

Rule 42—Make Use of a Distributed Cache for States

> ### Rule 42: What, When, How, and Why
>
> **What:** Use a distributed cache when storing session data in your system.
>
> **When to use:** Anytime you need to store session data and cannot do so in users' browsers.
>
> **How to use:** Watch for some common mistakes such as a session management system that requires affinity of a user to a Web server.
>
> **Why:** Careful consideration of how to store session data can help ensure that your system will continue to scale.
>
> **Key takeaways:** Many Web servers or languages offer simple server-based session management, but these are often fraught with problems such as user affiliation with specific servers. Implementing a distributed cache allows you to store session data in your system and continue to scale.

Per our recommendations in Figure 10.1, we hope that you've taken your time in arriving at the conclusion to maintain state in your application or system and in deciding that you cannot push that state out to the end user. It is a sad, sad day that you've come this far, and you should hang your head in shame that you were not enough of an engineer to figure out how to develop the system without state or without the ability to allow the end users to maintain state.

We are kidding, of course, as we have already acknowledged that there are some systems that must maintain state and even a small number where that state is best maintained within the service, application, or infrastructure of your product. Conceding that point, let's move on to a few rules for what *not* to do when you maintain state within your application.

First and foremost, stay away from state systems that require affinity to an application or Web server. If the server dies, all session information (including state) on that server will likely be lost, requiring those customers (potentially numbering into the thousands) to restart whatever process they were in. Even if you persist the data in some local or network-enabled storage, there will be an interruption of service and the user will need to start again on another server.

Second, don't use state or session replication services such as those within some application servers or third-party "clustering" servers. As stated previously in this chapter, such systems simply don't scale well as modifications to session need to be propagated to multiple nodes. Furthermore, in choosing to do this type of implementation we are creating a new concern for scalability in how much memory we use on our systems.

Third, when choosing a session cache or persistence engine, locate that cache away from the servers performing the actual work. While a small improvement, it helps with availability because when you lose a server, you either lose the cache associated with that server or the service running on it and not both. Creating a cache (or persistent) tier also allows us to scale that tier based on the cache accesses alone rather than needing to accommodate both the application service and the internal and remote cache services.

Distributed Session/State Cache Don'ts

Here are three approaches to avoid in implementing a cache to manage session or state:

- **Don't** implement systems that require affinity to a server to function properly.
- **Don't** use state or session replication to create duplicates of data on different systems.
- **Don't** locate the cache on the system doing the work (this doesn't mean you shouldn't have a local application cache—just that session information is best handled in its own tier of servers).

Abide by the rules of what *not* to do and the choice of what to do becomes straightforward. We strive to be agnostic in our approaches to such matters, and as such we care more about designs and less about the implementation details such as which open-source caching or database solution you might want to implement. We do feel strongly that there is rarely a need to develop the caching solution yourself. With all of the options from distributed object caches like Memcached to open-source and third-party databases, it seems ludicrous that you would implement your own caching solution for session information.

This brings us to the question of what to use for a cache. The question really comes down to reliability and persistence versus cost. If you expect to keep session or state information for quite some time, such as in the case of a shopping cart, you may decide that for some or all of your session information you are going to rely on a solution that allows lengthy and durable persistence. In many cases databases are used for these implementations. Clearly, however, a database will cost you more per transaction than a simpler solution such as a nonpersisting distributed object cache.

If you don't need persistence and durability, you can choose from one of many object caches. Refer to Chapter 6, "Use Caching Aggressively," for a discussion of object caches and their uses. In some cases, you may decide that you want both the persistence of a database and the relative low cost for performance of a cache in front of that database. Such an implementation gives you the persistence of a database while allowing you to scale the transactions more cost-effectively through a cache that front-ends that database.

Distributed Session/State Cache Considerations

Here are three common implementations for distributed caches and some notes on their benefits and drawbacks:

- Database-only implementations are the most costly overall but allow all data to be persisted and handle conflicts between updates and reads very well in a distributed environment.
- Nonpersistent object caches are fast and comparatively inexpensive, but they do not allow data to be recovered upon failure and aren't good for implementations with long periods between accesses by users.
- Hybrid solutions with databases providing persistence and caches providing cost-effective scale are great when persistence is required and low relative cost is preferred.

Summary

Our first recommendation is to avoid state at all costs, but we understand that session data is sometimes a necessity. If state is necessary, try to store the session data in the users' browsers (Rule 41). Doing so eliminates the need to store data in your system and allows for the servicing of requests by any Web server in the pool. If this is not possible, we recommend making use of a distributed caching system for session data (Rule 42). Following these rules will help ensure that your system continues to scale.

Notes

1. Authors' note: The reader is directed to Rule 33 in Chapter 8, "Database Rules," where we advise against this.
2. D. Kristol and L. Montulli, Networking Working Group Request for Comments 2965, "HTTP State Management Mechanism," October 2000,

   www.ietf.org/rfc/rfc2965.txt.
3. This solution was developed by Randy Wigginton, as explained and demonstrated on our blog,

   http://akfpartners.com/techblog/2010/11/20/slaying-firesheep/.

11 Asynchronous Communication and Message Buses

We met Chris Schremser and his company, ZirMed, a revenue cycle management company, in Chapter 3, "Design to Scale Out Horizontally." We now revisit Chris to hear about another lesson that he and his team learned about asynchronous communication. Chris began, "We process healthcare transactions between parties—healthcare providers and payers. There are really two implementations of that. One is, the day before a patient arrives for a scheduled visit, a physician's office will upload a whole big bunch of patient [records] and make sure that they have the current insurance on file, and they understand whether there are copays or things like that. The second implementation is, as a patient who wasn't scheduled comes into the office, the doctor's office will do a real-time eligibility benefit verification. This is a well-understood transaction in the healthcare space. And so we have an API that allows our partners, practice management systems or hospital information systems, to make this call as a patient presents to them in the office in real time to then determine whether they need to do any up-front payment collection. This is funny because, generally speaking, our application is very much written following asynchronous design patterns. One day, we saw a bunch of these API calls for real-time eligibility benefit verification start timing out. But not just the API calls started timing out; we saw kind of our entire application, various portions of our entire application, just really having performance problems."

The ZirMed system is designed such that their Web servers interact with an IBM MQ implementation through an internal, restricted API that allows them to talk to the MQ server. When ZirMed's external API receives a request, it makes an internal API call that puts a message on the queue and gets a correlation ID. The application then continues all of the other transaction processing for that call. While the call appears to be asynchronous, the one call that interacts with the MQ server itself is a synchronous call.

Chris explained, "Now, this design has been in place for ten years, and lots of pieces of the application integrate with this particular Web service that talks to the MQ server in this synchronous model. What we were seeing was that it had reached the maximum number of requests it could handle. It was clear very quickly that the payer endpoint was a problem. Payers, for this eligibility benefit verification, try to return in under 10 seconds. Generally speaking, they were very good at that. But, on this particular day, four of the major payers were slow. And by slow, I mean instead of returning a response in 10 seconds, they were returning a response in 50 seconds. We found a single transaction chokehold in our system that had been running for a long time. And so because we didn't understand that we had this single chokepoint, all of the systems that use this single chokepoint went slow. The cause of this slowness was an inordinately large number of real-time benefit verifications, and four of our payers way upstream from us were having issues. Our system was impacted because of a bad design choice, because we have one synchronous method that had been sitting around for so many years we had completely forgotten about it. And so some upstream event that shouldn't have had any impact on us at all caused our entire system to have a really bad day."

Chris and his team quickly fixed this problem and began an investigation to ensure that other synchronous calls were not lurking in their system. Chris concluded, "You want to talk about the epitome of how a bad design choice that seemed very innocuous all those years ago cascades down into a problem where we just didn't see it. We had no idea it was coming at us." The lesson here is that a single overlooked synchronous call that works well for years can cause a site-wide outage when processing slows down enough to exceed capacity.

Asynchronous communication between applications and services has been both the savior and the downfall of many platforms. When implemented properly, asynchronous communication is absolutely a valuable rung in the ladder of near-infinite scale. When implemented haphazardly, it merely hides the faults and blemishes of a product and is very much akin to putting "lipstick on a pig."

As a rule, we favor asynchronous communication whenever possible. As we discuss in this chapter, this favorable treatment requires that one not only communicates in an asynchronous fashion but actually develops the application to be asynchronous in behavior. This means, in large part, moving away from request/reply protocols—at least those with temporal constraints on responses. At the very least it requires aggressive time-outs and exception handling when responses are required within a specified period. These defense mechanisms can be difficult to implement even for the most experienced teams.

As the most often preferred implementation of asynchronous communication, the message bus is often underimplemented. In our experience, it is often thrown in as an afterthought without the appropriate monitoring or architectural diligence. As we discuss later in this chapter, recent platform-as-a-service offerings can increase the likelihood of a "quick and dirty" implementation. The result is often delayed catastrophe; as critical messages back up, the system appears to be operating properly until the entire bus grinds to a halt or crashes altogether. A critical portion of the product infrastructure is down. The site goes "off the air." The purpose of this chapter is to keep such brown-, gray-, or blackouts from happening.

Rule 43—Communicate Asynchronously as Much as Possible

> ### Rule 43: What, When, How, and Why
>
> **What:** Prefer asynchronous over synchronous communication whenever possible.
>
> **When to use:** Consider for all calls between services and tiers. Implement whenever possible and definitely for all noncritical calls.
>
> **How to use:** Use language-specific calls to ensure that requests are made in a nonblocking fashion and that the caller does not stall (block) awaiting a response.
>
> **Why:** Synchronous calls stop the entire program's execution waiting for a response, which ties all the services and tiers together, resulting in cascading latency or failures.
>
> **Key takeaways:** Use asynchronous communication techniques to ensure that each service and tier is as independent as possible. This allows the system to scale much farther than it would if all components were closely coupled together.

In general, asynchronous calls, no matter whether they are within a service or between two different services, are much more difficult to implement than synchronous calls. The reason is that asynchronous calls often require coordination to communicate back to the service that first sent a message an indication that the request has been completed. If you're firing and forgetting, there is no requirement for communication or coordination back to the calling method. This can easily be done in a variety of ways, including something as simple as the following PHP function, which makes use of the ampersand & to run the process in the background:

```
function asyncExec($filename, $options = '') {
[em][em]exec("php -f {$filename} {$options} >> /dev/null &");
}
```

However, firing and forgetting is not always an option. Often the calling method wants to know when the called method is complete. The reason for this could be that other processing has to happen before results can be returned. We can easily imagine a scenario in an e-commerce platform where shipping fees need to be recalculated along with crediting discount codes. Ideally we'd like to perform these two tasks simultaneously instead of having to calculate the shipping, which might require a third-party call to a vendor, and then processing the discount codes on the items in the shopping cart. But we can't send the final results back to the user until both are complete.

In most languages there are mechanisms designed to allow for the coordination and communication between the parent method and the asynchronous child method called *callbacks*. In C/C++, this is done through function pointers; in Java, it is done through object references. There are many design patterns that use callbacks, such as the delegate design pattern and the observer design pattern, each of which can be difficult to implement properly. But why go to all this hassle to call other methods or services asynchronously?

We go through the hassle of making some of our calls asynchronously because when all the methods, services, and tiers are tied together through synchronous calls, a slowdown or failure in one causes a delayed but nevertheless cascading failure in the entire system. As we discussed in reference to Rule 38 (Chapter 9, "Design for Fault Tolerance and Graceful Failure"), putting all your components in series has a multiplicative effect on failure. We covered this concept with availability, but it also works for the risk of a bug per KLOC (thousand lines of code). If methods A, B, and C have a 99.99% chance of being bug free and one calls the other, which calls the other, all synchronously, the chance of a bug affecting that logic stream of the system is 99.99% × 99.99% × 99.99% = 99.97%.

The same concept of reducing the risk of propagating failures was covered in Rule 36 (Chapter 9). In that rule we covered the idea of splitting a system's pools into separate swim lanes for different sets of customers, the benefit being that if there is a problem in one swim lane, it will not propagate to the other customers' lanes, which minimizes overall impact on the system. Additionally, fault detection is much easier because there are multiple versions of the same code that can be compared. This ability to more easily detect faults when the architecture has swim lanes also applies to modules or methods that have asynchronous calls.

Asynchronous calls prevent the spreading of failures or slowdowns, and they aid in the determination of where the bug resides when there is a problem. Most people who have had a database problem have seen it manifest itself in the app or Web tier because a slow query causes connections to back up, which in turn causes sockets to remain open on the app server—sometimes all of the sockets available on the app server. The database monitoring might not complain, but the application monitoring will. In this case you have synchronous calls between the app and database servers, and the problem becomes more difficult to diagnose.

Of course you can't have all asynchronous calls between methods and tiers in your system, so the real question is which ones should be made asynchronous. To start with, calls that must be implemented in synchronous fashion should have time-outs that allow for gracefully handling errors or continued processing when a synchronously called method or service fails or times out. The way to determine which calls are asynchronous candidates is to analyze each call based on criteria such as the following:

- **External API/third party**—Is the call to a third party or external API? If so, it absolutely should be made into an asynchronous call. Way too many things can go wrong with external calls to make them synchronous. You do not want the health and availability of your system tied to a system that you can't control and into which you have limited visibility.
- **Long-running processes**—Is the process being called notorious for being long running? Are the computational or I/O requirements significant? If so, such calls are great candidates for asynchronous calls. Often the more problematic issues are with slow-running processes rather than outright failures.
- **Error-prone/overly complex methods that change frequently**—Is the call to an overly complex method or service that gets changed frequently? The greater the number of large or complex changes, the more likely there is to be a bug

within the called service. Avoid tying critical code to code that is complex and needs to be changed frequently. That is asking for an increased number of failures.

- **Temporal constraint**—When a temporal constraint does not exist between two processes, consider firing and forgetting the child process. This might be the scenario when a new registrant receives a welcome e-mail or when logging events in your system. While your system should care if the e-mail doesn't go out or the event is not logged, the results of the registration page back to the user should not be stalled waiting for it to be sent.

These are just a few of the most important criteria to use in determining whether a call should be made asynchronously. A full set of considerations is left as an exercise for the reader and the reader's team. While we could list another ten of these criteria, as the numbers increase they become more specific to particular systems. Also, going through the exercise with your team of developers for an hour will make everyone aware of the pros and cons of using synchronous and asynchronous calls, which is more powerful in terms of following this rule and thus scaling your systems than any list that we could provide.

Rule 44—Ensure That Your Message Bus Can Scale

> ### Rule 44: What, When, How, and Why
>
> **What:** Message buses can fail from demand like any other physical or logical system. They need to be scaled.
>
> **When to use:** Anytime a message bus is part of your architecture.
>
> **How to use:** Employ the Y and Z AKF axes of scale.
>
> **Why:** To ensure that your bus scales to demand.
>
> **Key takeaways:** Treat message buses like any other critical component of your system. Scale them ahead of demand using either the Y or Z axis of scale.

One of the most common failures we identify within technology architectures is a giant single point of failure often dubbed the *enterprise service bus* or message bus. While the former is typically a message bus on steroids that often includes transformation capabilities and interaction APIs, it is also more likely to have been implemented as a single artery through the technology stack replete with aged messages clinging to its walls like so much cholesterol. When asked, our clients most often claim the asynchronous nature of messages transiting this bus as a reason why the required amount of time wasn't spent on architecting the bus to make it more scalable or highly available. While it is true that applications designed to be asynchronous are often more resilient to failure, and while these apps also tend to scale more efficiently, they are still prone to high-demand bottlenecks and failure points. The good news is that the principles you have learned so far in this book will as easily resolve the scalability needs of a message bus as they will solve the needs of a database.

Asynchronous systems tend to scale more easily and tend to be more highly available than synchronous systems. This attribute is due in large part to the component systems and services being capable of continuing to function in the absence or tardiness of certain data. But these systems still need to offload and accept information to function. When the system or service that allows them to "fire and forget" or to communicate but not block on a response becomes slow or unavailable, they are still subject to the problems of having logical ports fill up to the point of system failure. Such failures are absolutely possible in message buses, as the "flesh and blood" of these systems is still the software and hardware that run any other system. While in some cases the computational logic that runs the bus is distributed among several servers, systems and software are still required to allow the passing and interpretation of messages sent over the bus.

Having dispelled the notion that message buses somehow defy the laws of physics that bind the rest of our engineering endeavors, we can move on to how to scale them. We know that one of anything, whether it is physical or logical, is a bad idea from both an availability and a scalability perspective, so we need to split it up. As you may have already surmised from our previous hint, a good approach is to apply the AKF Scale Cube to the bus. In this particular case, though, we can remove the X axis of scale (see Rule 7 in Chapter 2, "Distribute Your Work") because cloning the bus probably won't buy us much. By simply duplicating the bus infrastructure and the messages transiting the bus, we would potentially raise our availability (if one bus fails, the other could continue to function), but we would still be left with a scale problem. Potentially we could send 1/Nth the messages on each of the N buses that we implement, but then all potential applications would need to listen to all buses. We still potentially have a reader congestion problem. What we need is a way to separate or differentiate our messages by something unique to the message or data (the Y axis—Rule 8, Chapter 2) or something unique to the customer or user (the Z axis—Rule 9, Chapter 2). Figure 11.1 is a depiction of the AKF's three axes of scale repurposed to message queues.

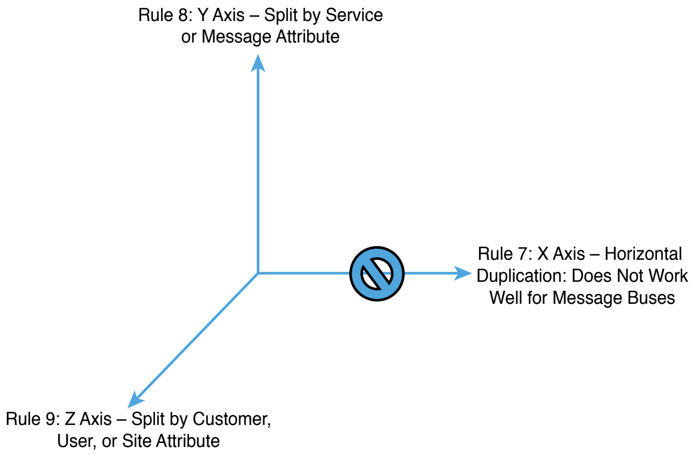

Rule 8: Y Axis – Split by Service or Message Attribute
Rule 7: X Axis – Horizontal Duplication: Does Not Work Well for Message Buses
Rule 9: Z Axis – Split by Customer, User, or Site Attribute

Figure 11.1 AKF Scale Cube for message buses

Having discarded the X axis of scale, let's further investigate the Y axis of scale. There are several ways in which we might discriminate or separate messages by attribute. One easy way is to dedicate buses to particular purposes. For a commerce site, we may choose a resource-oriented approach that transits customer data on one bus (or buses), catalog data on another, purchase data on another, and so on. We may also choose a services-oriented approach and identify affinity relationships between services and implement buses unique to affinity groups. "Hold on," you cry, "if we choose such an approach of segmentation, we lose some of the flexibility associated with buses. We can't simply hook up some new service capable of reacting to all messages and adding new value in our product."

Of course, you are absolutely correct. Just as the splitting of databases reduces the flexibility associated with having all of your data commingled in a single place for future activity, so does the splitting of a service bus reduce flexibility in communication. But remember that these splits are to enable the greater good of enabling hyper-growth and staying in business! Do you want to have a flat business that can't grow past the limitations of your monolithic bus or be wildly successful when exponentially increasing levels of demand come flooding into your site?

We have other Y axis options as well. We can look at things we know about the data such as its temporal qualities. Is the data likely to be needed quickly, or is it just a "for your information" piece of data? This leads us to considerations of quality of service, and segmenting by required service level for any level of data means that we can build buses of varying quality levels and implied cost to meet our needs. Table 11.1 summarizes some of these Y axis splits, but it is by no means meant to be an all-encompassing list.

Returning to Figure 11.1, we now apply the AKF Z axis of scale to our problem. As previously identified, this approach is most often implemented by splitting buses by customer. It makes the most sense when your implementation has already employed a Z axis split,

Table 11.1 **AKF Y Axis Splits of Message Bus**

Attribute Split	Pro	Con
Temporal	Monitoring for failures to meet response times is easy—just look for the oldest message against an absolute standard.	Not all messages are created equal. Some may be small and fast but not necessary for critical function completion.
Service	Only connects systems that need to communicate with each other.	Reduction in flexibility with various nodes connected in affinity fashion.
Quality of service	Costs to scale and make any bus highly available can scale in accordance with the importance of the message.	Likely to still need a way to scale buses with a lot of traffic for either highly important or unimportant messages.
Resource	Similar types of data (rather than the services) share a bus. Simple logical implementation.	May require some services to listen for infrequent messages on a bus.

because each of the swim lanes or pods can have a dedicated message bus. In fact, this would need to be the case if we truly wanted fault isolation (see Chapter 9). That doesn't mean that we can't leverage one or more message buses to communicate asynchronously between swim lanes. But we absolutely do not want to rely on a single shared infrastructure among the swim lanes for transactions that should complete within the swim lane.

The most important point in determining how to scale a message bus is to ensure that the approach is consistent with the approach applied to the rest of the technology architecture. If, for instance, you have scaled your architecture along customer boundaries using the AKF Z axis of scale, it makes the most sense to put a message bus that has been architected for Y axis scalability in each of these pods of customers. If you have split up services or resources via the Y axis of scale, it makes sense that the message buses should follow a similar pattern. If you have split up services and resources along the Y and Z axes and need only one method for the amount of message traffic you experience, the Z axis should most likely trump the Y axis to allow for greater fault isolation.

All of the previous recommendations apply directly to message buses over which you have complete control. What if you want to leverage PaaS solutions that may offer your organization time-to-market or cost advantages? The good news is that many of the PaaS solutions have considered the scalability and availability needs of their users. The bad news is that you still need to architect your solution to meet the needs of *your* customers and work within the constraints of the vendor offering these services. Many PaaS queuing services are built to scale across multiple servers in multiple data centers or availability zones. They may even promise "unlimited scalability." It is vitally important that the messaging infrastructure you implement use these capabilities but is not solely reliant on them for scalability or availability. Simply put, the vendor should not create your architecture. Ask yourself, "How does this platform fail and scale?" and ensure that you've used the Y and Z axis concepts to implement a well-architected service.

PaaS solutions for asynchronous communication can be easy to implement and cost-effective. They can also be poorly chosen, poorly implemented, and costly if key considerations are not taken into account at the outset. Consider aspects such as short and long polling to ensure proper latency of messages, batch-style requests to optimize your costs, and the need for ordered delivery of messages when implementing a PaaS solution.

Rule 45—Avoid Overcrowding Your Message Bus

Rule 45: What, When, How, and Why

What: Limit bus traffic to items of higher value than the cost to handle them.

When to use: On any message bus.

How to use: Value and cost-justify message traffic. Eliminate low-value, high-cost traffic. Sample low-value/low-cost and high-value/high-cost traffic to reduce the cost.

Why: Message traffic isn't "free" and presents costly demands on your system.

Key takeaways: Don't publish everything. Sample traffic where possible to ensure alignment between cost and value.

Nearly anything, if done to excess, can have severe and negative consequences. Physical fitness, for example, if taken to an extreme over long periods of time, can actually depress the immune system of the body and leave the person susceptible to viruses. Such is the case with publishing absolutely everything that happens within your product on a single (or, if you follow Rule 43, several) message bus. The trick is to know which messages have value, determine how much value they have, and determine whether that value justifies the cost of publishing the message at volume.

Why, having just explained how to scale a message bus, are we so interested in how much information we send to this now nearly infinitely scalable system? The answer is the cost and complexity of our scalable solution. While we are confident that following the advice in Rule 44 will result in a scalable solution, we want that solution to scale within certain cost constraints. We often see our clients publishing messages for nearly every action taken by every service. In many cases, this publication of information is duplicative of data that their application also stores in some log file locally (as in a Web log). Very often they will claim that the data is useful in troubleshooting problems or in identifying capacity bottlenecks (even while it may actually create some of those bottlenecks). In one case we've even had a client claim that we were the reason they published everything on the bus! This client claimed that they took our advice of "Design Your Systems to Be Monitored" (see Rule 49 in Chapter 12, "Miscellaneous Rules") to mean "Capture Everything Your System Does." The rise of robust data processing and analytics platforms has at least partially fueled this desire to use a message bus as the primary delivery mechanism for all types and sizes of data.

Let's start with the notion that not all data has equivalent value to your business. Clearly in a for-profit business, the data that is necessary to complete revenue-producing transactions is more important in most cases than data that helps us analyze transactions for future actions. Data that helps us get smarter about what we do in the future is probably more important than data that helps us identify bottlenecks (although the latter is absolutely very important). Clearly most data has some "option value" in that we might find a use for it later, but this value is lower than the value of data that has a clear and meaningful impact on our business today. In some cases, having a small sample of data gives us nearly as much value as having all of it, as in the case of statistically significant sampling of lower-value data in a high-transaction system.

In most systems and especially across most message buses (except when we segment by quality of service in Rule 44), data has a somewhat consistent cost. Even though the value of a transaction or data element (datum) may change by the type of transaction or even the value of the customer, the cost of handling that transaction remains constant. This runs counter to how we want things to work. Ideally we want the value of any element of our system to significantly exceed the cost of that element or in the worst case do no more than equal the cost. Figure 11.2 shows a simple illustration of this relationship and explains the actions a team should take with regard to the data.

The upper left quadrant of Figure 11.2 is the best possible case—a case in which the value of the data far exceeds the cost of sending it across the bus. In commerce sites clear examples of these transactions would be shopping cart transactions. These are clearly

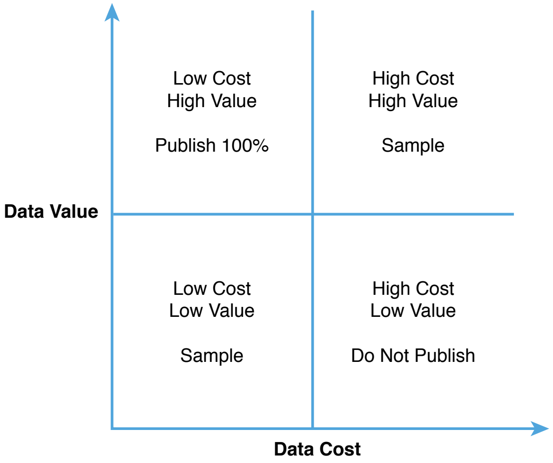


Figure 11.2 Cost/value relationship of data and corresponding message bus action

valuable, and we'd gladly use them to justify scaling out our message bus and increasing our costs. The lower right quadrant is an area in which we would likely just discard the data altogether until such time that it is deemed more valuable than the cost to transport it. A potential case might be where someone changes his or her profile picture on a social networking site (assuming that the profile picture change actually took place without a message being produced). In this case the data we would have sent through the message bus indicating that this event occurred could easily be derived through other means such as storing the last-updated time for the picture in our database. One potential downside is that we may not be able to act on the event quickly, and it's important to understand the value of this delay and include it in the calculation of value.

One consideration when discussing the value of data on message buses is that not all message buses are equal. Implementations range from open source like Apache Kafka and RabbitMQ to PaaS such as AWS Kinesis and SQS. Open-source solutions can potentially be a pitfall as the cost of sending messages can be seen as free. However, this overlooks the cost of the hardware or virtual machines on which the message bus runs and the cost of the engineers who have to monitor and maintain the bus.

The rate at which we publish something has an impact on its cost on the message bus. As we increase the demand on the bus, we increase the cost of the bus(es) as we need to scale the bus to meet the new demand. Sampling allows us to reduce the cost of those transactions, and some cases that we've described previously may still allow us to retain 100% of the value of those transactions. The act of sampling serves to reduce the cost of the transaction. Once the cost is reduced, the value of the data may exceed the new cost, thereby allowing us to keep some portion of the data. Reducing the cost of the transaction means we can reduce the size and complexity of our message bus(es) as we reduce the total number of messages being sent.

We can apply these techniques to many use cases of messaging systems. One popular implementation of message buses is an event-based architecture that enables massive streams of information to be divided into small messages and consumed by any number of systems for individualized use. Consider the example of anti-fraud services for payment systems that consume activities as messages and can in near real time execute complex algorithms and data enrichment processes. While it may appear to be an easy decision to capture all events on the message bus or queue, consider if it's absolutely necessary. Would sampling the events allow you to generate the same value, or do you need to analyze every event?

The overall message here is that just because you have implemented a message bus doesn't mean that you have to use it for everything. There will be a strong urge to send more messages than are necessary, and you should fight that urge. Always remember that not every piece of data is created equally in terms of value, but its cost is likely equal to that of its peers. Don't be easily swayed by the promise of cheap storage and the ability to seemingly process "everything." Use the technique of sampling to reduce the cost of handling data, and throw away (or do not publish) those things of low value. We return to the notion of value and cost in Rule 47 (Chapter 12) when we discuss storage.

Summary

This chapter is about asynchronous communication, and while it is the preferred method of communication, it is generally more difficult, more expensive (in terms of development and system costs), and can actually be done to excess. We started the chapter by providing an overview of asynchronous communication and then offered a few of the most critical guidelines for when to implement asynchronous communication. We then followed up with two rules dealing with message buses, which are one of the most popular implementations of asynchronous communication.

In Rules 43 and 44, we covered how to scale a message bus and how to avoid overcrowding it. As we mentioned in the introduction to this chapter, the message bus, while often the preferred implementation of asynchronous communication, is also often underimplemented. Being thrown in as an afterthought without the appropriate monitoring or architectural diligence, it can turn out to be a huge nightmare instead of an architectural advantage.

Pay attention to these rules to ensure that the communication within and between services can scale effectively as your system grows.

This page intentionally left blank

12

Miscellaneous Rules

Everyone is aware of the peaks and valleys inherent in demand in both online and brick-and-mortar commerce. Black Friday (the day after Thanksgiving) was a name coined in the 1960s to mark the kickoff to the holiday shopping season, "Black" referring to retailers moving from operating at a loss (in the red) to generating a profit (in the black). For years the online analog to Black Friday was Cyber Monday, the first Monday following Thanksgiving, and it represented the peak day of transactions for e-commerce companies. Whether Black Friday and Cyber Monday are really the days when most retailers turn a profit for the year is moot. But one thing is true for nearly all retailers: Black Friday and Cyber Monday are immensely important to them, and for those days their capacity must be clearly planned and transactions carefully monitored. As any e-commerce veteran can tell you, the analysis and planning for these peak days are critical to the success of a Web store. Failure to meet demand can not only significantly hurt sales and profits, it may land you on the pages of *Fortune* magazine or, worse yet, ensure that you get an unwelcome segment on CNN or CNBC.

As difficult as e-commerce capacity planning and monitoring can be, it pales in comparison to the complexity of events that financial services solutions must endure. Whereas e-commerce solutions may have seasonal peaks, financial services have these easily identified and plannable events, as well as "flash" events pursuant to exogenous unplanned variables completely outside their control. Daily market openings and closings, earnings announcements (typically quarterly), annual index rebalancing (Russell Index), options expirations (quad witching day), and known federal report distributions or trade group meetings (e.g., US Federal Reserve Board, aka "The Fed," meetings on interest rates, consumer confidence reports) are all examples of planned events that generate bursts in volume. External events that may trigger demand surges on financial services solutions include geopolitical catastrophes such as the bombing of the World Trade Center on 9/11 and rumors regarding the internal operations of a company. To show just how sensitive the markets can be in reacting to events, generating unanticipated demand on financial systems, recall that the markets moved 1% in reaction to the Associated Press having its Twitter account hacked to falsely report an attack on the President of the United States in April of 2013.[1]

But don't take our word for it; listen to Brad Peterson's war stories and lessons learned. Brad has experience from both sides of the financial services industry. For a number of years he was the EVP and CIO of Charles Schwab, a company providing services to individual and institutional traders, and is now the EVP and CIO of NASDAQ, a leading provider of trading, clearing, exchange technology, listing, information, and public company services across six continents. Prior to Schwab and NASDAQ, Brad worked with us as eBay's VP and CIO. His experience in trade placement, trade execution, and commerce give him a one-of-a-kind perspective on capacity planning within volatile trading solutions and the need to monitor solutions in order to react quickly to known and unforeseen events.

"Consider the phrase 'flash crash' or 'market storm,'" said Brad with a chuckle. "The fact that 'flash' or 'storm' is part of the phrase helps indicate just how difficult a world it is in which our teams operate. It happens in a moment's notice, and the cause, at least in terms of surviving the storm, is irrelevant at the time. In the case of August 24, 2015, the popular opinion is that it was a herd mentality of traders rushing to protect themselves—a 'panic buy' of protection. But again, that really doesn't matter in the heat of the moment. What matters is that the Dow dropped 1,100 points at open. You simply can't foresee that; you can't identify a date on which that is going to happen. Furthermore, if you want to provide a service and do it profitably, you can't just have an infinite number of servers waiting to be used. Infinite capacity is infinite cost. Infinite cost is infinite loss. So what do you do?"

Brad broke the solution down into two important variables. "The first is that you need cost-effective capacity on demand to handle these unpredictable events with unpredictable volume. In the old world order before infrastructure as a service, large companies like us would have vendors supply racked equipment that we paid for only when we first lit them up and used them. Doing so helped us handle demand surges without having to pay for the capacity before we needed it. Unfortunately, once you've started using these systems, you start depreciating them and the amortization affects your bottom line as the systems sit mostly idle outside of peak demand periods. Additionally, you have to be a large company with a lot of purchasing power over your providers to get that kind of deal. In the new world order of elastic consumption-based economics, we can rent burst capacity from infrastructure-as-a-service providers. Of course, we have to plan for that in advance and ensure that our architectures support the capability to burst compute into the cloud."

The second variable in the solution is monitoring for events. Brad continued, "Most companies in my experience monitor everything you'd expect—CPU utilization, memory, heap sizes, network utilization, and so forth. And these things are all important, but they don't provide the signals that indicate you are about to experience something out of the ordinary. If you want to handle one of the unpredictable events, you need to sense that something is off kilter as the market starts to change. You need to see that the first and second derivatives of trade requests and executions with respect to time are changing abnormally with respect to past behavior. These are the most important demand-related signals a financial services industry can watch, and our solutions must be developed to monitor such activities and alert upon them in real time. Business activities, not the underlying system activities, are the best predictors that something is about to cause a problem. These signals, not the rest of the noise of monitoring, are your indication that you need to do something right now. Besides monitoring these key business metrics in real time, our network operations centers monitor national and international news. We pay attention to how our business is doing, how our systems are doing, and what events outside of our control are likely to drive changes in our business and systems capacity."

Brad emphasized the importance of distinguishing between the signal intelligence and the noise of everything else his team monitors. Quality "signals" closely related to business value creation help Brad and executives at other companies identify not only changes in the markets but significant deviations in human behavior that may be indicative of underlying system problems or new exogenous variables for which they must prepare. We'll return to this key insight in Rule 49, "Design Your Application to Be Monitored."

Rule 46—Be Wary of Scaling through Third Parties

> ### Rule 46: What, When, How, and Why
>
> **What:** Scale your own system; don't rely on vendor solutions to achieve scalability.
>
> **When to use:** Whenever considering whether to use a new feature or product from a vendor.
>
> **How to use:** Rely on the rules in this book for understanding how to scale, and use vendor-provided products and services in the most simplistic manner possible.
>
> **Why:** Three reasons for following this rule: Own your destiny, keep your architecture simple, and reduce your total cost of ownership. Understand that the customer holds you, not the vendor, responsible for the scale and availability of your product.
>
> **Key takeaways:** Do not rely on vendor products, services, or features to scale your system. Keep your architecture simple, keep your destiny in your own hands, and keep your costs in control. All three of these rules can be violated when relying on a vendor's proprietary scaling solution.

As you climb the management track in technology organizations, you invariably start to attend vendor meetings and eventually find yourself being solicited by vendors almost constantly. In a world where global IT spending of over $3.5 trillion was projected to fall 5.5% in 2015,[2] you can safely bet that vendors are recruiting the best salespeople possible and are working their hardest to sell you their products and services. These vendors are often sophisticated in their approaches and truly consider their relationships with clients to be long-term. Vendors approach these relationships hoping to extract higher revenues and profits from each client with which they work. The salespeople interacting with you are expected to increase their book of business by increasing their total client base and increasing value achieved on a per-client basis. This is all great business and we don't fault the vendors for trying, but we do want to caution you as a technologist or business leader to be aware of the pros and cons of relying on vendors to help you scale. We're going to cover three reasons that you should avoid relying on vendors to scale.

First, we believe that you should want to keep the fate of your company, your team, and your career in your own hands. Looking for vendors to relieve you of this burden will likely result in a poor outcome because to a vendor you're just one of many customers. A vendor will never respond to your crisis in the same fashion and time frame in which you will respond. As a CTO or technology leader, if the vendor you vetted and selected fails, causing downtime for your business, you are just as responsible as if you had implemented the vendor's solution yourself. Every product has bugs; even vendor-provided solutions have bugs. Most software patches are 99% bug fixes with the major versions reserved for new features. Vendors also have to consider the relative priority of your problem compared with the problems of all their other customers, just as you do with the solutions you produce.

This position should not be taken to imply that you should do everything yourself such as writing your own database or firewall. Use vendors for things that they can do better than you and that are not part of your core competency. Ultimately we are talking about ensuring that you can split up your application and product to allow it to scale, because scaling should be a core competency of yours. The key differentiator here is that the ability to scale is your responsibility. Conversely, building the best database is rarely your responsibility.

Our next point on this topic is that, as with most things in life, simpler is better. We teach a simple cube (see Chapter 2, "Distribute Your Work") to understand how to build scalable architectures. The more complex you make your system, the more you are likely to suffer from availability issues. More complex systems are more difficult and more costly to maintain. Clustering technologies are much more complex than straightforward log shipping for creating read replicas. Recall Rules 1 and 3 from this book (see Chapter 1, "Reduce the Equation"): "Don't Overengineer the Solution" and "Simplify the Solution Three Times Over." Complex math problems should be made simple by reducing the equation into simpler problems. The same is true of your architecture—simple, nonmonolithic pieces (like a vendor-provided self-clustering database) are more complex, not less. They place monoliths in your architecture instead of reducing the architecture to easily solved components.

Third, let's talk about the real total cost of trying to scale through third-party vendors. One of our architectural principles, which should be one of yours as well, is that the most cost-effective way to scale is to be vendor neutral. Locking yourself into a single vendor gives the vendor the upper hand in negotiations. We're going to pick on database vendors for a bit, but this discussion applies to almost all technology vendors. The reason database companies build additional features into their systems is that their revenue streams need to grow faster than just the adoption of new customers would normally allow. The way to achieve this is through a technique called up-selling, which involves getting existing customers to purchase additional features or services.

One of the most prevalent add-on features for databases is clustering. It's a perfect feature in that it supposedly solves a problem that customers that are growing rapidly need to solve—scalability of the customer's platform. Additionally, it is proprietary, meaning that once you start using one vendor's clustering service, you can't just switch to another's solution. If you're a CTO of a hyper-growth company that needs to continue producing new features for your customers and might not be that familiar with scaling architectures, when a vendor waltzes in and tells you that they have a solution to your biggest, scariest problem, you're anxious to jump on board with them. And often the vendor will make the jump very easy by throwing in this additional feature for the first-year contract. The reason vendors do this is that they know this is the hook. If you start scaling with their technology solution, you'll be reluctant to switch, and they can increase prices dramatically when you have very few alternatives.

For these three reasons—control of your own destiny, additional complexity, and total cost of ownership—we implore you to consider scaling without relying on vendors. If you do choose a vendor for part of your solution, exercise caution in how you implement their product or service. Think about how you would replace the vendor one to three years down the road. The rules in this book should more than adequately arm you and your team with the knowledge to get started scaling in a simple but effective manner.

Rule 47—Purge, Archive, and Cost-Justify Storage

> ### Rule 47: What, When, How, and Why
>
> **What:** Match storage cost to data value, including removing data of value lower than the cost to store it.
>
> **When to use:** Apply to data and its underlying storage infrastructure during design discussions and throughout the lifecycle of the data in question.
>
> **How to use:** Apply recency, frequency, and monetization analysis to determine the value of the data. Match storage costs to data value.
>
> **Why:** Not all data is of similar value to the business, and in fact data value often declines (or less often increases) over time. Therefore, we should not have a single storage solution with equal cost for all our data.
>
> **Key takeaways:** It is important to understand and calculate the value of your data and to match storage costs to that value. Don't pay for data that doesn't have a stakeholder return.

Storage has been getting cheaper, faster, and denser in the same fashion as processors. As a result, some companies and many organizations within companies just assume that storage is virtually free. In fact, a marketing professional asked us in 2002 why we were implementing mail file size maximum constraints while companies like Google and Yahoo! were touting free unlimited personal e-mail products. Our answer was twofold and forms the basis for this rule. First, the companies offering these solutions expected to make money off their products through advertising, whereas it was unclear how much additional revenue the marketing person was committing to while asking for more storage. Second, and more importantly, while the marketing professional considered a decrease in cost to be virtually "free," the storage in fact still costs money in at least three ways: the storage itself cost money to purchase, the space it occupied cost us money (or lost opportunity relative to higher-value services where we owned the space), and the power and HVAC to spin and cool the drives was increasing rather than decreasing in cost on a per-unit basis.

Discussing this point with our marketing colleague, we came upon a shared realization and a solution to the problem. The realization was that not every piece of data (or e-mail), whether it be used for a product or to run an IT back-office system, is of equivalent value. We hinted at this concept in Rule 44, “Ensure That Your Message Bus Can Scale” (see Chapter 11, “Asynchronous Communication and Message Buses”). Order history in commerce systems provides a great example of this concept; the older the data, the less meaningful it is to our business and our customers. Our customers aren’t likely to go back and view purchase data that is ten years, five years, or possibly even two years old, and even if they are, the frequency of that viewing is likely to decay over time. Furthermore, that data isn’t likely as meaningful to our business in determining product recommendations as more recent purchase information (except in the case of certain durable goods that get replaced in those intervals like vehicles). Given this reduction in value both to the customer and to our business, why would we possibly want to store it on systems that cost us the same as storage for more recent data?

The solution was to apply a marketing concept known as RFM, which stands for *recency, frequency, and monetization*, analysis. Marketing gurus use this technique to make recommendations to people or send special offers to keep high-value customers happy or to “reactivate” those that haven’t been active recently. The concept is extensible to our storage needs (or, as the case may be, our storage woes). Many of our clients tell us that the fastest-growing portion of their budget and in some cases the largest single component of their budget is storage. We’ve applied this RFM technique within their businesses to help both mature their understanding of the data residing on their storage subsystems and ultimately solve the problem through a tiered storage archival and purge strategy.

First we need an understanding of the meaning of the terms constituting the acronym RFM. *Recency* accounts for how recently the data item in question has been accessed. This might be a file in your storage system or rows within a database. *Frequency* speaks to how frequently that data is accessed. Sometimes this is captured as the mean period between access and the rough inverse of this—the number of accesses over some time interval. *Monetization* is the value that a specific piece of data has to your business in general. When applied to data, these three terms help us calculate overall business value and access speeds. As you might expect, we are moving toward applying our proprietary cube to yet another problem! By matching the type of storage to the value of the data in an RFM-like approach, we might have a cube that has high-cost storage mapped to high-value data in the upper right and an approach to delete and/or archive data in the bottom left. The resulting cube is shown in Figure 12.1.

The X axis of our repurposed cube addresses the frequency of access. The axis moves from data that is never (or very infrequently) accessed to that which is accessed constantly or always. The Y axis of the cube identifies recency of access and has low values of never to high values of the data being accessed right now. The Z axis of the cube deals with monetization, from values of no value to very high value. Using the cube as a method of analysis, we can plot potential solutions along the multiple dimensions of the cube. Data in the lower left and front portion of the cube has no value and is never accessed, meaning that we should purge this data if regulatory conditions allow us to do so. Why would we incur a cost for data that won’t return value to our business? The upper right and back portion of our three-dimensional cube identifies the most valuable pieces of

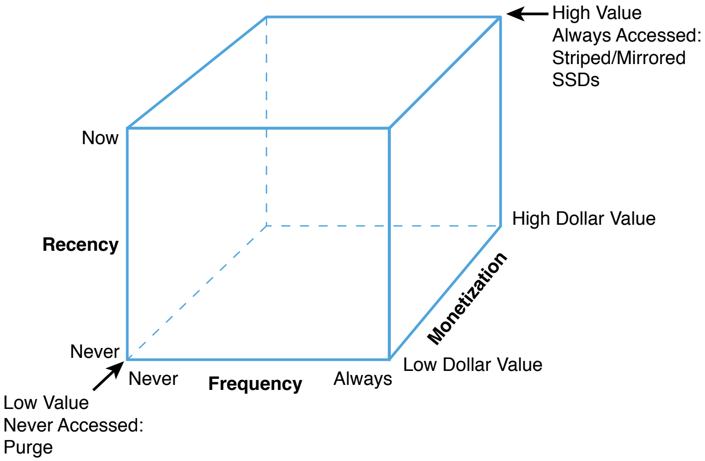


Figure 12.1 AKF Scale Cube applied to RFM storage analysis

business data. We strive to store these on the solutions with the highest reliability and fastest access times such that the transactions that use them can happen quickly for our clients. Ideally we would cache this data somewhere as well as having it on a stable storage solution, but the underlying storage solution might be the fastest solid-state disks (SSDs) that current technology supports. These disks might be striped and mirrored for access speed and high availability.

The product of our RFM analysis might yield a score for the value of the data overall. Maybe it's as simple as a product, or maybe you'll add some magic of your own that actually applies some dollar value to the resulting score. If we employed this dollar value score to a value curve that matched the resulting RFM value to the cost of a solution to support it, we might end up with a diagram similar to Figure 12.2.

We purge very low-value data, just as we did in our analysis of the cube of Figure 12.1. Low-value data goes on low-cost systems with slow access times. If you need to access it, you can always do it offline and e-mail a report or whatever to the requester. High-value systems might go on very fast but relatively costly SSDs or some storage area network equivalent. The curve of Figure 12.2 is purely illustrative and was not developed with actual data. Because data varies in value across businesses, and because the cost of the solutions to support this data changes over time, you should develop your own solution. Some data might need to be kept for some period of time due to regulatory or legal reasons. In these cases we look to put the data on the cheapest solution possible to meet our legal/regulatory requirements while not making the data dilutive in terms of shareholder value.

Many commerce companies and some financial services institutions apply this concept to older data that is meant primarily for historical record keeping. Your online bank provider may put all historical transactions in a read-only file system, removing it from the transactional database. You can view your past transactions, but it is unlikely that you

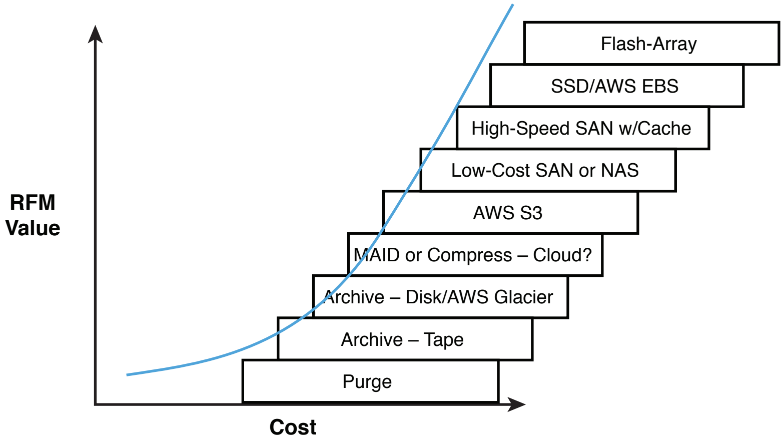


Figure 12.2 RFM value, cost, and solution curve

can make changes to them, update them, or cancel them. After some period of time, say 90 days or a year, they remove your access to the transactions online and you have to request them via e-mail. Similarly, commerce sites often remove the history of orders that have been shipped and received from a database and keep it for some period of time on inexpensive storage. The data will not change and is meant only to help you check on old purchases or reorder items of interest.

It is important to keep in mind that data ages, and the RFM cube recognizes this fact. As a result, it isn't a one-time analysis but rather a living process. As data ages and decays in value, we want to move it to storage whose cost is consistent with that declining value. As such, we need to have processes and procedures to archive data or to move it to lower-cost systems. In rare cases, data may actually increase in value with age, and so we may also need systems to move data to higher-cost (more reliable and faster) storage over time. Make sure you address these needs in your architecture.

Rule 48—Partition Inductive, Deductive, Batch, and User Interaction (OLTP) Workloads

Rule 48: What, When, How, and Why

What: Partition and fault-isolate unique workloads to maximize overall availability, scalability, and cost effectiveness.

When to use: Whenever architecting solutions that are composed of analytics (inductive or deductive) and product (batch or user interactive) solutions.

How to use: Ensure that solutions supporting the four basic types of workloads (induction, deduction, batch, and user interactive/OLTP) are fault isolated from each other and that each exists in its own fault isolation zone.

Why: Each of these workloads has unique demand and availability requirements. Furthermore, each can impact the availability and response time of the other. By separating these into distinct fault isolation zones, we can ensure that they do not conflict with each other, and each can have an architecture that is cost-effective for its unique needs.

Key takeaways: Induction is the process of forming hypotheses from data. Deduction is the process of testing hypotheses against data to determine validity. Induction and deduction solutions should be separated for optimum performance and availability. Similarly batch, user interaction, and analytics workloads should be separated for maximum availability, scalability, and cost effectiveness. Separate analytics solutions into those meant for induction and deduction. Choose the right solution for each.

We like to segment product and business workloads into four broad categories: induction, deduction, product batch, and user product interactive (OLTP). Batch processing is typically launched on a scheduled basis and is often relatively long running, addressing a number of records or documents and performing complex calculations or validation checks. User interactive transactions (aka OLTP) are user-initiated requests of a product or service and tend to be fast-responding transactions such as search, checkout, add_comment, add_to_cart, and so on. Induction and deduction are best defined through a brief story.

The Broken Windows Theory gets its name from a 1982 *Atlantic Monthly* article.[3] The article claimed that the existence of broken windows within any neighborhood invites vandalism. Once a cycle of vandalism begins, it is both difficult to stop and will likely escalate to other crimes. A corollary to this theory is that by focusing on smaller crimes such as vandalism, cities can reduce overall crime rates. New York City's Mayor Giuliani's seemingly successful Zero Tolerance Program was based on this theory. Crime rates in New York dropped over a ten-year period following the implementation of this program. Malcolm Gladwell, in his book *Tipping Point*,[4] used New York's implementation of the Broken Windows Theory as further indication of the correctness of the theory.

The authors of the *Atlantic Monthly* article, Kelling and Wilson, were puzzled why the assignment of police officers to walking beats (which were meant to increase the perception of police presence and order) didn't result in a reduction in crime as originally hypothesized. Based on some existing psychological research, they hypothesized that the evidence of crime (such as vandalism) was the best indicator of future crimes. Such evidence, they reasoned, trumped any deterrence provided by a perception of police presence or of active policing. Having formed a hypothesis, they moved to test this hypothesis through several social experiments. Their tests, and subsequent tests and validations such as the New York experiment, illustrate the process of deduction. Deduction is the testing of a hypothesis against available data with the intent of proving or disproving its validity. Deduction starts with a generalized view of relationships between data elements and ends with additional data or information (valid or invalid relationships).

Enter the self-described "Rogue Economist" Steven D. Levitt and his coauthor Stephen J. Dubner, both of *Freakonomics* fame. While the two authors didn't deny that the Broken Windows Theory could explain some drop in crime, they did cast significant doubt on the approach as the primary explanation for crime rates dropping. Crime rates dropped nationally during the same ten-year period in which New York pursued its Zero Tolerance

Program. This national drop in crime occurred both in cities that practiced Broken Windows and in those that did not. Furthermore, crime rates dropped irrespective of either an increase or a decrease in police spending. The explanation, therefore, argued the authors, could not primarily be broken windows. The most likely explanation and most highly correlated variable was a reduction in the pool of potential criminals. Roe v. Wade legalized abortion, and as a result there was a significant decrease in the number of unwanted children, a disproportionately high percentage of whom would grow up to be criminals.

Levitt and Dubner started with a process of induction rather than deduction. Induction starts with a number of observations or data elements and attempts to create a generalization (or hypothesis) regarding their relationships. Whereas deduction starts with a hypothesis (e.g., "A change in independent variable X causes a corresponding change in dependent variable Y," or "Decreasing vandalism decreases crime overall"), induction begs the question of what data elements are related in an attempt to form a hypothesis (e.g., "What independent variables account for a change in dependent variable Y?" or "What are the variables that may best account for the change in crime rates?").

Induction and deduction aren't mutually exclusive. In fact, they are meant to support each other in a cyclic fashion. The Broken Windows Theory authors weren't necessarily wrong; they just didn't have the best answer. It's highly probable that they are right, but to a lesser degree than Levitt and Dubner. Induction not only helps us increase the probability of having the best answer, it helps to reduce the existence of the dreaded Type 1 error—an error in which we incorrectly support our hypothesis as being correct when it is in fact not correct.

The Broken Windows/*Freakonomics* comparison not only helps to define induction and deduction, it helps to show that both of them should be present in your analytics solutions. Furthermore, the comparison of induction and deduction serves to illustrate a vast difference in requirements to be successful. Referring back to our initial discussion of batch versus user interactive characteristics, you can probably see a similar differentiation. Induction looks much more like batch processing in its attributes; we need to look at significantly more data and likely run for significantly longer periods of time than deduction. Similarly, deduction shares similar attributes with OLTP in that it is fast compared to induction and typically requires significantly less data.

We've discussed that data size and run times/response times are likely to vary among induction, deduction, batch, and OLTP transactions. Each of these is likely to also have varying degrees of impact on the business when they fail. In most scenarios, OLTP transactions provide the primary utility to which the consumer subscribes. As such, ensuring that the "front door" is always open to allow customers to transact is typically the most important of the four transaction types. Batch transactions are also typically very important, though they often can have a bit of a delay in processing time without the same impact on overall customer utility. Deduction systems usually are third in the priority of business importance, as the continual testing for relationships (such as in the continual testing of fraud models) in an analytics system may be important for the end product solution. Induction systems are typically the least important in terms of availability because if our quest for new understanding between variables is delayed a day, it is not likely to have nearly the impact as the other three systems.

This hierarchy of business value as well as the performance and data characteristics of the four systems argues for the need to architect four distinct and fault-isolated (from each other) environments wherever possible. We know from experience that slow-running transactions operating on very large data sets often compete and interfere with the timely completion of transactions that operate on smaller data sets and would otherwise complete quickly. Therefore, we should split systems that support induction from deduction and those that support batch from OLTP. Additionally, because the value (and hence criticality) of each of these is likely different, we do not want to subject the architecture of the least important (but nevertheless valuable) solution to the costs of the most important. Recovery time objectives (RTOs) and recovery point objectives (RPOs) of OLTP systems may be an order of magnitude smaller (in other words, need to be recovered faster with less data loss) than those of systems supporting induction. We may want only seconds of downtime with our primary user-facing product, whereas perhaps we can support a few hours of downtime on a system meant to support induction or pattern-seeking activities.

In summary, we believe that properly developed analytics (or "big data") systems recognize the need for both induction and deduction. When properly developed, these solutions recognize that induction and deduction are so different in behavior, desired response times, and need for availability that they should be developed on separate fault-isolated solutions built to the needs of the process in question. Batch and OLTP processing should be similarly separated when possible, resulting in four unique environments for batch, OLTP, induction, and deduction. In implementing such a fault-isolated environment, we can increase availability through reduced workload contention, increase the scalability of each solution unique to the demands placed on it, and reduce the overall cost of all solutions by implementing availability architectures consistent with their individual business needs.

Rule 49—Design Your Application to Be Monitored

Rule 49: What, When, How, and Why

What: Think about how you will need to monitor your application as you are designing it.

When to use: Anytime you add or change modules of your code base.

How to use: Build hooks into your system to record transaction times.

Why: Having insight into how your application is performing will help answer many questions when there is a problem.

Key takeaways: Adopt as an architectural principle that your application must be monitored. Additionally, look at your overall monitoring strategy to make sure you are first answering the question "Is there a problem?" and then the "Where?" and "What?"

When it comes to monitoring, most SaaS companies start by installing one of the open-source monitoring tools such as Cacti, Ntop, or Nagios, just to name a few. This is a great way to check in on network traffic or the servers' CPU and memory but requires someone to pay attention to the monitors. The next stage for most companies is to set up an automatic alerting system, which is a nice step forward. The problem with this scenario is that if you follow these steps, by now you're at a point where you are paging at least one person in the middle of the night when a server starts consuming too much memory. If your reaction is "Great!" then let's consider the questions "Is there a problem with your site that is affecting your customers?" and "How large is the impact?" The reality is that with the types of monitors we've described here, you just don't know. These monitors would not have helped Brad Peterson during the flash crashes and market storms, and they won't help you understand whether you are having an incident.

Just because a server has high CPU or memory utilization does not mean that your customers are having any issue with your site at all. And while reacting to every bump in the night on your system is better than ignoring them, the best solution is to actually know the impact of that bump on your customers to determine the most appropriate response. The way to achieve this is to monitor your system from the perspective of a business metric. For example, if you have an e-commerce site, you might want to monitor the number of items put into shopping carts or the total value of purchases per time period (second, minute, 10 minutes, and so on). For an auction site you might want to monitor the number of items listed or the number of searches performed per time period. The correct time period is the one that smooths out the data points enough that the normal variation doesn't obscure real issues. When you plot these business metrics on a graph against the data from a week ago (week over week), you can easily start to see when there is a problem.

Figure 12.3 shows a graph of new account signups for a site. The solid line represents data from last week, and the dotted line represents data from this week. Notice the drop starting around 9:00 a.m. and lasting until 3:00 p.m. From this graph it is obvious that there was a problem. If the cause of this problem had been a network issue with your ISP, monitoring your servers would not have caught it; system CPU and memory would have been fine during these six hours because very little processing was taking place on them. The next step after plotting this data is to put in an automated check that compares today's values against last week's and alerts when it is out of statistical significance.[5]

Monitoring business metrics also puts business context directly into the hands of technologists and engineers who are building the systems. By displaying graphs of business metrics, technologists not only can easily see what is driving their business and when it is impacted by a problem but also have near-real-time insight into how their changes (code, network, or systems) impact those metrics. From the previous example, if your team introduced a code change during a low-traffic period that impacts only certain signup use cases, you may not have noticed the magnitude of the problem until more customers visited the site.

Once you know there is a problem affecting your customers, you can react appropriately and start asking the other questions that monitoring is designed to answer. These

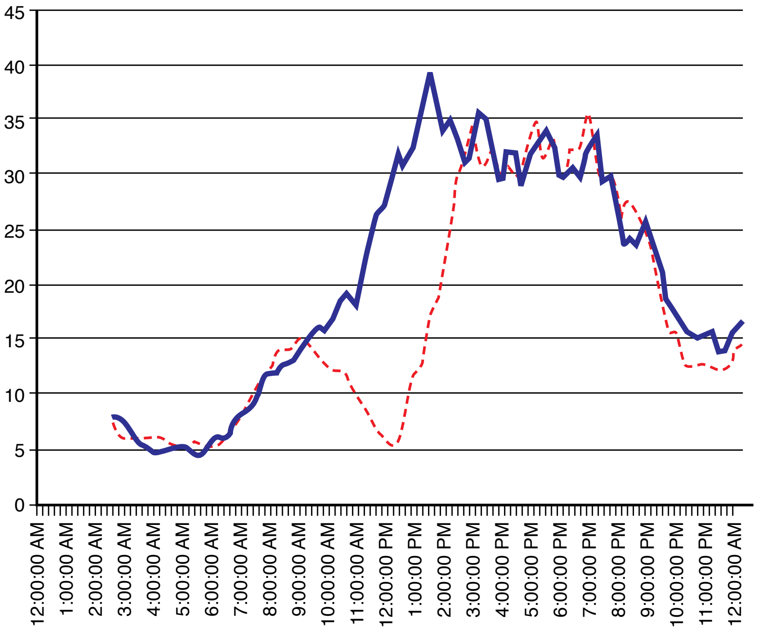


Figure 12.3 Monitoring business metrics

questions include "Where is the problem?" and "What is the problem?" Figure 12.4 shows two triangles. The one on the left represents the scope of the question being asked, and the one on the right represents how much data is required to answer that question. Answering the question "Is there a problem?" doesn't require much data but is very large

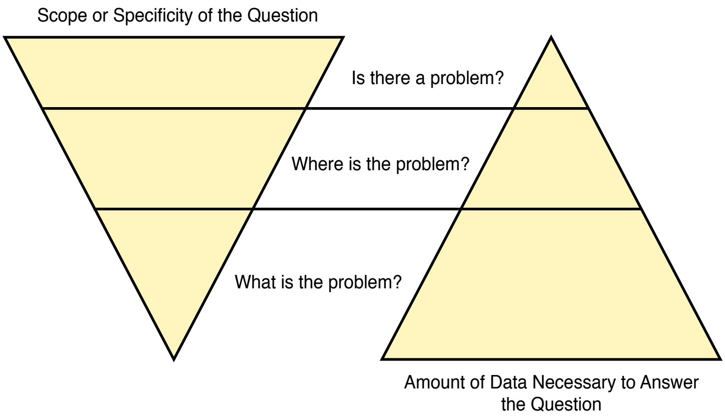

Scope or Specificity of the Question
Is there a problem?
Where is the problem?
What is the problem?
Amount of Data Necessary to Answer the Question

Figure 12.4 Monitoring scope versus amount of data

in terms of scope. This as we previously discussed is best answered by monitoring business metrics. The next question—"Where is the problem?"—requires more data, but the scope is smaller. This is the level at which monitoring of the application will help provide an answer. We cover this in more detail later in the chapter. The last question—"What is the problem?"—requires the most data but is the most narrow in scope. This is where Nagios, Cacti, and other tools can be used to answer the question.

Rule 16 (see Chapter 4, "Use the Right Tools") covered the importance of trapping exceptions, logging them, and monitoring the logs. We're going to expand on the concept by discussing how you not only should catch errors and exceptions but also should adopt as an architectural principle the concept of "design to be monitored." This simply stated is the idea that your application code should make it easy to place hooks in for watching the execution of transactions such as SQL, APIs, remote procedure calls (RPCs), or method calls. Some of the best monitored systems have asynchronous calls before and after a transaction to record the start time, transaction type, and end time. These are then posted on a bus or queue to be processed by a monitoring system. Tracking and plotting this data can yield all types of insights into answering the question "Where is the problem?"

Once you've mastered answering these three questions—"Is there a problem?" "Where is the problem?" and "What is the problem?"—there are a couple of advanced monitoring questions that you can start to ask. The first is "Why is there a problem?" This question usually gets asked during the postmortem process as discussed in Rule 27 (see Chapter 7, "Learn from Your Mistakes"). When performing continuous deployments, addressing this problem requires a much faster cycle than a typical postmortem. Your organization must integrate learning from answering "Why?" into the next hour's code release. Insights from answering this question might include adding another test to the smoke or regression test to ensure that future bugs similar to this one get caught before being deployed.

A final question that monitoring can help answer is "Will there be a problem?" This type of monitoring requires the combination of business monitoring data, application monitoring data, and hardware monitoring data. Using statistical tools such as control charts or machine learning algorithms such as neural nets or Bayesian belief networks, this data can be analyzed to predict whether a problem is likely to occur. A final step to this and perhaps the holy grail of monitoring would be for the system to take action when it thinks a problem will occur and fix itself. Considering that today most automatic failover monitors mess up and fail over inappropriately, we know that automatic or self-healing systems are a long way off.

While predicting problems sounds like a fun computer science project, don't even think about it until you have all the other steps of monitoring in place and working well. Start monitoring from the customer's perspective by using business metrics. This will start you off on the appropriate level of response to all the other monitoring.

Rule 50—Be Competent

> ### Rule 50: What, When, How, and Why
>
> **What:** Be competent, or buy competency in, for each component of your architecture.
>
> **When to use:** For any solution delivered online.

How to use: For each component of your product, identify the team responsible and level of competency with that component.

Why: To a customer, every problem is *your* problem. You can't blame suppliers or providers. You provide a service—not software.

Key takeaways: Don't confuse competence with build-versus-buy or core-versus-context decisions. You can buy solutions and still be competent in their deployment and maintenance. In fact, your customers demand that you do so.

Maybe you think that this particular rule goes without saying. "Of course we are competent in what we do—how else could we remain in business?" For the purpose of this rule, we are going to assume that you have an Internet offering—some sort of SaaS platform, e-commerce offering, financial services offering, or some other solution delivered over the Internet.

How well does your team really understand the load balancers that you use? How often are you required to get outside help to resolve problems or figure out how to implement something on those load balancers? What about your databases? Do your developers or DBAs know how to identify which tables need indices and which queries are running slowly? Do you know how to move tables around on file systems to reduce contention and increase overall capacity? How about your application servers? Who is your expert with those?

Perhaps your answer to all these questions is that you don't really need to do those things. Maybe you've read books, including at least one other that these authors have written, that indicate you should identify the things in which you have value-producing differentiation capabilities and specialize in those areas. The decision that something is "noncore" or that one should buy versus build (as in the case of a build-versus-buy decision) should not be confused with whether your team should be competent in the technology that you buy. It absolutely makes sense for you to use a third-party or open-source database, but that doesn't mean that you don't have to understand and be capable of operating and troubleshooting that database.

Your customers expect you to deliver a service to them. To that end, the development of unique software to create that service is a means to an end. You are, at the end of the day, in the service business. Make no mistake about that. It is a mindset requirement that when not met has resulted in the deterioration and even death of companies. Friendster's focus on the "F-graph," the complex solution that calculated relationships within the social network, was at least one of the reasons Facebook won the private social network race. At the heart of this focus was an attitude held within many software shops—a focus that the challenging problem of the F-graph needed to be solved. This focus led to a number of outages within the site, or very slow response times as systems ground to a halt while attempting to compute relationships in near real time. Contrast this with a focus on a service, where availability and response time are more important than any particular feature. Software is just the means for providing that service.

But in our world you also need more than just software. Infrastructure is also important to help us get transactions processed on time and in a highly available manner. Just as we might focus too much on the solution to a single problem, so might we overlook the other components within our architecture that deliver that service. If we must be competent in our software to deliver a service, so must we be competent in everything else that we employ to deliver that service. Our customers expect superior service delivery. They don't understand and don't care that you didn't develop and aren't an expert in the particular component of your architecture that is failing. The world of SaaS has redefined the traditional mindset of shipping software and requires removing the divide not only between software and infrastructure teams but also between the pieces you build and the pieces you buy or partner with to deliver the whole solution.

So it is that while we don't need to develop every piece of our solution (in fact we should not develop every piece), we do need to understand each piece. For anything we employ, we need to know that we are using it correctly, maintaining it properly, and restoring it to service promptly when it fails. We can do this by developing those skills within our own team or by entering into relationships to help us. The larger our team and the more we rely on the component in question, the more important it is that we should have some in-house expertise. The smaller our team and the less important the component, the more willing we should be to outsource the expertise. But in relying on partners for help, the relationship needs to go beyond that provided by most suppliers of equipment. A simple way to get this started is to ensure that every component has an "owner," whether that is a person or a team. Perform a quick review of all your services and assign owners where they are missing. The provider of the service needs to have "skin in the game." Put another way, a provider needs to feel your pain and the pain of the customer when your service fails. You can't be caught in a wait queue for second-level support while your customers scream at you for service restoration.

Summary

This chapter is a mix of rules that don't fit well in other chapters but are extremely important. Starting with a warning to avoid letting vendors provide scalability solutions through their products and continuing with advice about keeping business logic in the most appropriate place, monitoring appropriately, and finally being competent, we covered a wide variety of topics. While all of these rules are important, perhaps no other rule than Rule 50, "Be Competent," brings it all together. Understanding and implementing these 50 rules is a great way to ensure that you and your team are competent when it comes to ensuring that your systems will scale.

Notes

1. Edmund Lee, "AP Twitter Account Hacked in Market-Moving Attack," Bloomberg Business, April 24, 2013,

   www.bloomberg.com/news/articles/2013-04-23/dow-jones-drops-recovers-after-false-report-on-ap-twitter-page.

2. “Gartner Says Worldwide IT Spending to Decline 5.5 Percent in 2015,” Gartner Newsroom, June 30, 2015,

   www.gartner.com/newsroom/id/3084817.

3. George L. Kelling and James Q. Wilson, “Broken Windows: The Police and Neighborhood Safety,” *The Atlantic*, March 1982,

   www.theatlantic.com/magazine/archive/1982/03/broken-windows/304465/.

4. Malcolm Gladwell, *The Tipping Point: How Little Things Can Make a Big Difference* (Boston: Little, Brown, 2000).

5. Meaning that the data is unlikely to have occurred because of chance. Wikipedia, “Statistical Significance,”

   http://en.wikipedia.org/wiki/Statistical_significance.

This page intentionally left blank

13

Rule Review and Prioritization

In addition to being a handy aggregation of the rules for future reference, this chapter introduces a method by which these rules may be analyzed for application in your architecture. If you are building something from scratch, we highly recommend the inclusion of as many of the 50 rules as makes sense in your product. The amount of scale required in your service should be taken into consideration, as creating scale that isn't required is a waste of company assets and should be avoided. If you are attempting to redesign your existing system in an evolutionary fashion for greater scale, the method of risk-benefit analysis represented herein may help you prioritize the application of these rules in your reengineering efforts.

A Risk-Benefit Model for Evaluating Scalability Projects and Initiatives

Before we begin describing a risk-benefit model, let's first review why we are interested in scalability. The desire to have a highly available and usable (to some specified or implied quality-of-service metric) product or service, even under conditions of moderate to extreme growth, is why we invest in making our product scalable. If we weren't interested in what would happen to our product's availability or quality of service as demand increases on our site, we wouldn't be interested in attempting to scale it. This unfortunately is the approach most of us think that state governments take in implementing their Department of Motor Vehicles services. The government simply doesn't appear to care that people will line up and wait for hours during periods of peak demand. Nor do they care that the individuals providing the service couldn't seem to care less about the customer. The government knows it offers a service that customers must use if they want to drive, and any customers who are dissatisfied and leave will simply come back another day. But most of the products that we build won't have this state-sanctioned and -protected monopoly. As a result, we must be concerned about our availability and hence our scalability.

It is within this context of concern over availability and scalability that we represent our concept of risk. The risk of an incident caused by the inability to scale manifests itself as a threat to our quality of service or availability. One method of calculating risk is to look at the probability that a problem will happen multiplied by its impact should it happen (or move from risk to issue). Figure 13.1 shows this method of risk decomposition.

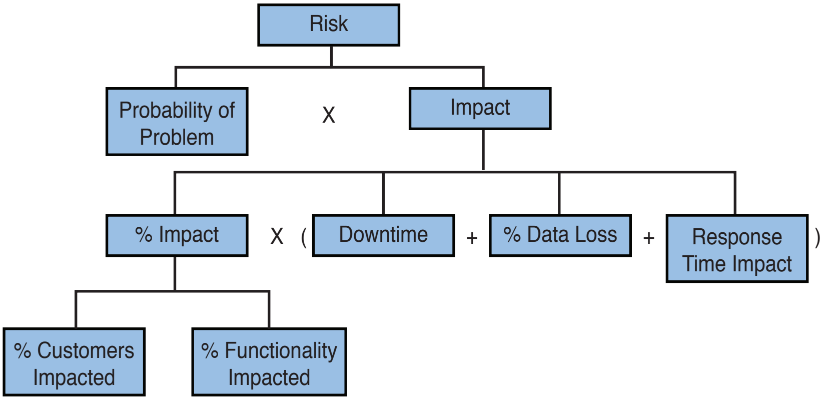


Figure 13.1 Scalability and availability risk composition

The impact can be broken down into components of downtime (the amount of time your service is unavailable), data loss, and response time degradation. Percentage impact might be further broken down into the percentage of the customers impacted and the percentage of the functionality impacted. There is certainly further decomposition possible; for instance, some customers may represent significantly greater value on either a license or transaction fee basis. Furthermore, downtime may have a different multiplier applied to it than response time; data loss may trump both of these.

This model isn't meant to hold true for all businesses. Rather it is meant to demonstrate a way in which you can build a model to help determine which things you should focus on in your business. One quick note before we proceed with how to use the model: We highly recommend that businesses calculate actual impact on revenue when determining the system's actual availability. For instance, if you can demonstrate that you lost 10% of your expected revenue in a given day, you should say your availability was 90%. Wall clock time is a terrible measure of availability as it treats every hour of every day equivalently, and most businesses don't have an equal distribution of traffic or revenue-producing transactions. A simple way to calculate this is by monitoring a set of transactions that represent the usage of your service. For example, an e-commerce service might monitor items added to carts or checkouts. By comparing today's usage over yesterday's or against usage on the same day last week, you can quickly determine the impact of your service's downtime. Now back to our discussion.

The terminal nodes (nodes without children) of the decomposition graph of Figure 13.1 are leaves within our risk tree: Probability of a Problem, % Customers Impacted, % Functionality Impacted, Downtime, % Data Loss, and Response Time Impact. We can see that many of our rules map to these leaves. For instance, Rule 1 is really about decreasing the probability of a problem happening by ensuring that the system is easily understood and therefore less likely to be problematic. It may also decrease downtime as the solution is likely to be easier to troubleshoot and resolve. Using a bit of subjective analysis, we may decide that this rule has a small (or low) benefit to impact and a medium change to the probability of a problem happening. The result of this may be that the rule has an overall medium impact on risk (low + medium = ~medium).

We want to take just a minute to explain how we arrived at this conclusion. The answer is that we used a simple High, Medium, and Low analysis. There's no rocket science or magic here; we simply used our experience with how the rule might apply to the tree we developed for our particular view of risk and scalability. So while the answer was largely subjective, it was informed by 70 years of combined experience within our partnership. While you can certainly invest in creating a more deterministic model, we are not convinced that it will yield significantly better results than having a team of smart people determine how these rules impact your risk.

We are now going to assume that a change in risk serves as a proxy for benefit. This is a fairly well-understood concept within business, so we won't spend a great deal of time explaining it. Suffice it to say that if you can reduce your risk, you reduce the likelihood of a negative impact on your business and therefore increase the probability of achieving your goals. What we need to do now is determine the cost of this risk reduction. Once we know that, we can take our benefit (risk reduction) and subtract our costs to develop the solution. This combination creates a prioritization of work. We suggest a simple High, Medium, and Low categorization for cost. High cost might be anything more than $10 million for a very large company to more than $10,000 for a very small company. Low cost could be anything less than $1 million for a very large company or verging on zero for a very small company. This is often derived from the cost of developer days. For example, if your average developer's salary and benefits cost the business $100,000 per year, then for each day a developer needs to work on a project, assuming ~250 days of work per year, you add $400 to the cost.

Our prioritization equation then becomes risk reduction minus cost equals priority or (R − C = P). Table 13.1 shows how we computed the nine permutations of risk and cost

Table 13.1 **Risk Reduction, Cost, and Benefit Calculation**

Risk Reduction	Cost	Resulting Benefit/Priority
High	High	Medium—3
High	Medium	High—2
High	Low	Very High—1
Medium	High	Low—4
Medium	Medium	Medium—3
Medium	Low	High—2
Low	High	Very Low—5
Low	Medium	Low—4
Low	Low	Medium—3

and the resulting priority. The method we chose was simple. The benefit for any equation where risk and cost were equivalent was set to Medium and the priority set to the midrange of 3. Where risk reduction was two levels higher than cost, the benefit was set to Very High and the priority set to 1. Where risk reduction was two levels lower than cost (Low risk, High cost), the benefit was set to Very Low and priority set to 5. Differences of one were either Low (risk reduction Low and cost Medium) with a priority score of 4 or High (risk reduction Medium and cost Low) with a priority score of 2. The projects with the lowest priority score have the highest benefit and are the first things we will do.

Using the previous approach, we now will rate each of our 50 rules. The sidebars from each chapter for each rule are repeated in the following section with the addition of our estimation of risk reduction, cost, and the calculated benefit/priority. As we mentioned earlier, the way we arrived at these values was the result of our experience of more than 70 years (combined) with more than 400 companies and growing in our consultancy practice at AKF Partners. Your particular risk reduction, cost, and benefit may vary, and we encourage you to calculate and prioritize these yourselves. Our estimates should be good for smaller action-oriented companies that simply want to know what to do today. Note that these estimates are to "rework" an existing solution. The cost of designing something from scratch will be significantly different—typically a lower cost but with equivalent benefit.

For the sake of completeness, there are many other approaches to determining cost and benefit. You may, for instance, replace our notion of risk reduction with the ability to achieve specific KPIs (key performance indicators) within your company. The previous method would be appropriate if you had a KPI regarding the scalability and availability of your product (something all Web-enabled businesses should have). If you are a business with contractual obligations, another approach might be to determine the risk of not meeting specific SLAs (service level agreements) outlined within your contracts. Many other possibilities exist. Just choose the right approach to prioritize for your business and get going!

50 Scalability Rules in Brief

Following is a review of the 50 scalability rules within this book. It serves as a quick reference to each rule, including what the rule is, when to use it, how to use it, why to use it, and the key takeaways. In addition, we have evaluated each of the rules consistent with our preceding discussion along the dimensions of risk reduction, cost to implement, and the resulting benefits and priority of implementation.

Rule 1—Don't Overengineer the Solution

> **What:** Guard against complex solutions during design.
>
> **When to use:** Can be used for any project and should be used for all large or complex systems or projects.
>
> **How to use:** Resist the urge to overengineer solutions by testing ease of understanding with fellow engineers.

Why: Complex solutions are excessively costly to implement and are expensive to maintain long term.

Key takeaways: Systems that are overly complex limit your ability to scale. Simple systems are more easily and cost-effectively maintained and scaled.

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 2—Design Scale into the Solution (D-I-D Process)

What: An approach to provide JIT (just-in-time) scalability.

When to use: On all projects; this approach is the most cost-effective (resources and time) to ensure scalability.

How to use:

- Design for 20x capacity.
- Implement for 3x capacity.
- Deploy for roughly 1.5x capacity.

Why: D-I-D provides a cost-effective, JIT method of scaling your product.

Key takeaways: Teams can save a lot of money and time by thinking of how to scale solutions early, implementing (coding) them a month or so before they are needed, and deploying them days before the customer rush or demand.

Risk reduction: Low

Cost: Low

Benefit and priority: Medium—3

Rule 3—Simplify the Solution Three Times Over

What: Used when designing complex systems, this rule simplifies the scope, design, and implementation.

When to use: When designing complex systems or products where resources (engineering or computational) are limited.

How to use:

- Simplify scope using the Pareto Principle.
- Simplify design by thinking about cost effectiveness and scalability.
- Simplify implementation by leveraging the experience of others.

Why: Focusing just on "not being complex" doesn't address the issues created in requirements or story and epoch development or the actual implementation.

Key takeaways: Simplification needs to happen during every aspect of product development.

Risk reduction: Low

Cost: Low

Benefit and priority: Medium—3

Rule 4—Reduce DNS Lookups

What: Reduce the number of DNS lookups from a user perspective.

When to use: On all Web pages where performance matters.

How to use: Minimize the number of DNS lookups required to download pages, but balance this with the browser's limitation for simultaneous connections.

Why: DNS lookups take a great deal of time, and large numbers of them can amount to a large portion of your user experience.

Key takeaways: Reduction of objects, tasks, computation, and so on is a great way of speeding up page load time, but division of labor must be considered as well.

Risk reduction: Low

Cost: Low

Benefit and priority: Medium—3

Rule 5—Reduce Objects Where Possible

What: Reduce the number of objects on a page where possible.

When to use: On all Web pages where performance matters.

How to use:

- Reduce or combine objects but balance this with maximizing simultaneous connections.
- Look for opportunities to reduce weight of objects as well.
- Test changes to ensure performance improvements.

Why: The number of objects impacts page download times.

Key takeaways: The balance between objects and methods that serve them is a science that requires constant measurement and adjustment; it's a balance among customer usability, usefulness, and performance.

Risk reduction: Low

Cost: Low

Benefit and priority: Medium—3

Rule 6—Use Homogeneous Networks

What: Ensure that switches and routers come from a single provider.

When to use: When designing or expanding your network.

How to use:

- Do not mix networking gear from different OEMs for switches and routers.
- Buy or open-source for other networking gear (firewalls, load balancers, and so on).

Why: Intermittent interoperability and availability issues simply aren't worth the potential cost savings.

Key takeaways: Heterogeneous networking gear tends to cause availability and scalability problems. Choose a single provider.

Risk reduction: High

Cost: High

Benefit and priority: Medium—3

Rule 7—Design to Clone or Replicate Things (X Axis)

What: Typically called horizontal scale, this is the duplication of services or databases to spread transaction load.

When to use:

- Databases with a very high read-to-write ratio (5:1 or greater—the higher the better).
- Any system where transaction growth exceeds data growth.

How to use:

- Simply clone services and implement a load balancer.
- For databases, ensure that the accessing code understands the difference between a read and a write.

Why: Allows for fast scale of transactions at the cost of duplicated data and functionality.

Key takeaways: X axis splits are fast to implement, are low cost from a developer effort perspective, and can scale transaction volumes nicely. However, they tend to be high cost from the perspective of operational cost of data.

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 8—Design to Split Different Things (Y Axis)

What: Sometimes referred to as scale through services or resources, this rule focuses on scaling by splitting data sets, transactions, and engineering teams along verb (services) or noun (resources) boundaries.

When to use:

- Very large data sets where relations between data are not necessary.
- Large, complex systems where scaling engineering resources requires specialization.

How to use:

- Split up actions by using verbs, or resources by using nouns, or use a mix.
- Split both the services and the data along the lines defined by the verb/noun approach.

Why: Allows for efficient scaling of not only transactions but also very large data sets associated with those transactions. Also allows for the efficient scaling of teams.

Key takeaways: Y axis or data/service-oriented splits allow for efficient scaling of transactions, large data sets, and can help with fault isolation. Y axis splits help reduce the communication overhead of teams.

Risk reduction: Medium

Cost: Medium

Benefit and priority: Medium—3

Rule 9—Design to Split Similar Things (Z Axis)

What: This is very often a split by some unique aspect of the customer such as customer ID, name, geography, and so on.

When to use: Very large, similar data sets such as large and rapidly growing customer bases or when response time for a geographically distributed customer base is important.

How to use: Identify something you know about the customer, such as customer ID, last name, geography, or device, and split or partition both data and services based on that attribute.

Why: Rapid customer growth exceeds other forms of data growth, or you have the need to perform fault isolation between certain customer groups as you scale.

Key takeaways: Z axis splits are effective at helping you to scale customer bases but can also be applied to other very large data sets that can't be pulled apart using the Y axis methodology.

Risk reduction: High

Cost: High

Benefit and priority: Medium—3

Rule 10—Design Your Solution to Scale Out, Not Just Up

What: *Scaling out* is the duplication or segmentation of services or databases to spread transaction load and is the alternative to buying larger hardware, known as *scaling up*.

When to use: Any system, service, or database expected to grow rapidly or that you would like to grow cost-effectively.

How to use: Use the AKF Scale Cube to determine the correct split for your environment. Usually the horizontal split (cloning) is the easiest.

Why: Allows for fast scale of transactions at the cost of duplicated data and functionality.

Key takeaways: Plan for success and design your systems to scale out. Don't get caught in the trap of expecting to scale up only to find out that you've run out of faster and larger systems to purchase.

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 11—Use Commodity Systems (Goldfish Not Thoroughbreds)

What: Use small, inexpensive systems where possible.

When to use: Use this approach in your production environment when going through hypergrowth and adopt it as an architectural principle for more mature products.

How to use: Stay away from very large systems in your production environment.

Why: Allows for fast, cost-effective growth. Allows you to purchase the capacity you need rather than spending for unused capacity far ahead of need.

Key takeaways: Build your systems to be capable of relying on commodity hardware, and don't get caught in the trap of using high-margin, high-end servers.

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 12—Scale Out Your Hosting Solution

What: Design your systems to have three or more live data centers to reduce overall cost, increase availability, and implement disaster recovery. A data center can be an owned facility, a colocation, or a cloud (IaaS or PaaS) instance.

When to use: Any rapidly growing business that is considering adding a disaster recovery (cold site) data center or mature business looking to optimize costs with a three-site solution

How to use: Scale your data per the AKF Scale Cube. Host your systems in a "multiple live" configuration. Use IaaS/PaaS (cloud) for burst capacity, new ventures, or as part of a three-site solution.

Why: The cost of data center failure can be disastrous to your business. Design to have three or more as the cost is often less than having two data centers. Consider using the cloud as one of your sites, and scale for peaks in the cloud. Own the base; rent the peak.

Key takeaways: When implementing disaster recovery, lower your cost by designing your systems to leverage three or more live data centers. IaaS and PaaS (cloud) can scale systems quickly and should be used for spiky demand periods. Design your systems to be fully functional if only two of the three sites are available, or N-1 sites available if you scale to more than three sites.

Risk reduction: High

Cost: High

Benefit and priority: Medium—3

Rule 13—Design to Leverage the Cloud

What: This is the purposeful utilization of cloud technologies to scale on demand.

When to use: When demand is temporary, spiky, and inconsistent and when response time is not a core issue in the product. Consider when you are "renting your risk"—when future demand for new products is uncertain and you need the option of rapid change or walking away from your investment. Companies moving from two active sites to three should consider the cloud for the third site.

How to use:

- Make use of third-party cloud environments for temporary demand, such as seasonal business trends, large batch jobs, or QA environments during testing cycles.
- Design your application to service some requests from a third-party cloud when demand exceeds a certain peak level. Scale in the cloud for the peak, then reduce active nodes to a basic level.

Why: Provisioning of hardware in a cloud environment takes a few minutes as compared to days or weeks for physical servers in your own colocation facility. When used temporarily, this is also very cost-effective.

Key takeaways: Design to leverage virtualization in all sites and grow in the cloud to meet unexpected spiky demand.

Risk reduction: Low

Cost: Medium

Benefit and priority: Low—4

Rule 14—Use Databases Appropriately

What: Use relational databases when you need ACID properties to maintain relationships between your data and consistency. For other data storage needs consider more appropriate tools such as NoSQL DBMSs.

When to use: When you are introducing new data or data structures into the architecture of a system.

How to use: Consider the data volume, amount of storage, response time requirements, relationships, and other factors to choose the most appropriate storage tool. Consider how your data is structured and your products need to manage and manipulate data.

Why: An RDBMS provides great transactional integrity but is more difficult to scale, costs more, and has lower availability than many other storage options.

Key takeaways: Use the right storage tool for your data. Don't get lured into sticking everything in a relational database just because you are comfortable accessing data in a database.

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 15—Firewalls, Firewalls Everywhere!

What: Use firewalls only when they significantly reduce risk, and recognize that they cause issues with scalability and availability.

When to use: Always.

How to use: Employ firewalls for critical PII, PCI compliance, and so on. Don't use them for low-value static content.

Why: Firewalls can lower availability and cause unnecessary scalability chokepoints.

Key takeaways: While firewalls are useful, they are often overused and represent both availability and scalability concerns if not designed and implemented properly.

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 16—Actively Use Log Files

What: Use your application's log files to diagnose and prevent problems.

When to use: Put a process in place that monitors log files and forces people to take action on issues identified.

How to use: Use any number of monitoring tools from custom scripts to Splunk or the ELK framework to watch your application logs for errors. Export these and assign resources to identify and solve the issue.

Why: The log files are excellent sources of information about how your application is performing for your users; don't throw this resource away without using it.

Key takeaways: Make good use of your log files, and you will have fewer production issues with your system. When issues do occur, you will be able to address them more quickly.

Risk reduction: Low

Cost: Low

Benefit and priority: Medium—3

Rule 17—Don't Check Your Work

What: Avoid checking and rechecking the work you just performed or immediately reading objects you just wrote within your products.

When to use: Always.

How to use: Never read what you just wrote for the purpose of validation. Store data in a local or distributed cache if it is required for operations in the near future.

Why: The cost of validating your work is high relative to the unlikely cost of failure. Such activities run counter to cost-effective scaling.

Key takeaways: Never justify reading something you just wrote for the purpose of validating the data. Trust your persistence tier to notify you of failures, and read and act upon errors associated with the write activity. Avoid other types of reads of recently written data by storing that data locally.

Risk reduction: Low

Cost: Medium

Benefit and priority: Low—4

Rule 18—Stop Redirecting Traffic

What: Avoid redirects when possible; use the right method when they are necessary.

When to use: Always.

How to use: If you must use redirects, consider server configurations instead of HTML or other code-based solutions.

Why: Redirects in general delay the user, consume computation resources, are prone to errors, and can negatively affect search engine rankings.

Key takeaways: Use redirects correctly and only when necessary.

Risk reduction: Low

Cost: Low

Benefit and priority: Medium—3

Rule 19—Relax Temporal Constraints

What: Alleviate temporal constraints in your system whenever possible.

When to use: Anytime you are considering adding a constraint that an item or object must maintain a certain state between a user's actions.

How to use: Relax constraints in the business rules.

Why: The difficulty in scaling systems with temporal constraints is significant because of the ACID properties of most RDBMSs.

Key takeaways: Carefully consider the need for constraints such as items being available from the time a user views them until the user purchases them. Some possible edge cases where users are disappointed are much easier to compensate for than not being able to scale.

Risk reduction: High

Cost: Low

Benefit and priority: Very High—1

Rule 20—Leverage Content Delivery Networks

What: Use CDNs (content delivery networks) to offload traffic from your site.

When to use: When speed improvements and scale warrant the additional cost.

How to use: Most CDNs leverage DNS to serve content on your site's behalf. Thus, you may need to make minor DNS changes or additions and move content to be served from new subdomains.

Why: CDNs help offload traffic spikes and are often economical ways to scale parts of a site's traffic. They also often substantially improve page download times.

Key takeaways: CDNs are a fast and simple way to offset spikiness of traffic as well as traffic growth in general. Make sure you perform a cost-benefit analysis and monitor the CDN usage.

Risk reduction: Medium

Cost: Medium

Benefit and priority: Medium—3

Rule 21—Use `Expires` Headers

What: Use `Expires` headers to reduce requests and improve the scalability and performance of your system.

When to use: All object types need to be considered.

How to use: Headers can be set on Web servers or through application code.

Why: The reduction of object requests increases the page performance for the user and decreases the number of requests your system must handle per user.

Key takeaways: For each object type (image, HTML, CSS, PHP, and so on), consider how long the object can be cached and implement the appropriate header for that time frame.

Risk reduction: Low

Cost: Low

Benefit and priority: Medium—3

Rule 22—Cache Ajax Calls

What: Use appropriate HTTP response headers to ensure cacheability of Ajax calls.

When to use: Every Ajax call except for those absolutely requiring real-time data that are likely to have been recently updated.

How to use: Modify `Last-Modified`, `Cache-Control`, and `Expires` headers appropriately.

Why: Decrease user-perceived response time, increase user satisfaction, and increase the scalability of your platform or solution.

Key takeaways: Leverage Ajax and cache Ajax calls as much as possible to increase user satisfaction and increase scalability.

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 23—Leverage Page Caches

What: Deploy page caches in front of your Web services.

When to use: Always.

How to use: Choose a caching solution and deploy.

Why: Decrease load on Web servers by caching and delivering previously generated dynamic requests and quickly answering calls for static objects.

Key takeaways: Page caches are a great way to offload dynamic requests, decrease customer response time, and scale cost-effectively.

Risk reduction: Medium

Cost: Medium

Benefit and priority: Medium—3

Rule 24—Utilize Application Caches

What: Make use of application caching to scale cost-effectively.

When to use: Whenever there is a need to improve scalability and reduce costs.

How to use: Maximize the impact of application caching by analyzing how to split the architecture first.

Why: Application caching provides the ability to scale cost-effectively but should be complementary to the architecture of the system.

Key takeaways: Consider how to split the application by Y axis (Rule 8) or Z axis (Rule 9) before applying application caching in order to maximize the effectiveness from both cost and scalability perspectives.

Risk reduction: Medium

Cost: Medium

Benefit and priority: Medium—3

Rule 25—Make Use of Object Caches

What: Implement object caches to help scale your persistence tier.

When to use: Anytime you have repetitive queries or computations.

How to use: Select any one of the many open-source or vendor-supported solutions and implement the calls in your application code.

Why: A fairly straightforward object cache implementation can save a lot of computational resources on application servers or database servers.

Key takeaways: Consider implementing an object cache anywhere computations are performed repeatedly, but primarily this is done between the database and application tiers.

Risk reduction: High

Cost: Low

Benefit and priority: Very High—1

Rule 26—Put Object Caches on Their Own "Tier"

What: Use a separate tier in your architecture for object caches.

When to use: Anytime you have implemented object caches.

How to use: Move object caches onto their own servers.

Why: The benefits of a separate tier are better utilization of memory and CPU resources and having the ability to scale the object cache independently of other tiers.

Key takeaways: When implementing an object cache, it is simplest to put the service on an existing tier such as the application servers. Consider implementing or moving the object cache to its own tier for better performance and scalability.

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 27—Learn Aggressively

What: Take every opportunity, especially failures, to learn and teach important lessons.

When to use: Be constantly learning from your mistakes as well as your successes.

How to use: Watch your customers or use A/B testing to determine what works. Employ a postmortem process and hypothesize failures in low-failure environments.

Why: Doing something without measuring the results or having an incident without learning from it are wasted opportunities that your competitors are taking advantage of. We learn best from our mistakes—not our successes.

Key takeaways: Be constantly and aggressively learning. The companies that learn best, fastest, and most often are the ones that grow the fastest and are the most scalable. Never let a good failure go to waste. Learn from every one and identify the architecture, people, and process issues that need to be corrected.

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 28—Don't Rely on QA to Find Mistakes

What: Use QA to lower the cost of delivered products, increase engineering throughput, identify quality trends, and decrease defects—*not* to increase quality.

When to use: Whenever you can get greater throughput by hiring someone focused on testing rather than writing code. Use QA to learn from past mistakes—always.

How to use: Hire a QA person anytime you get greater than one engineer's worth of output with the hiring of a single QA person.

Why: Reduce cost, increase delivery volume/velocity, decrease the number of repeated defects.

Key takeaways: QA doesn't increase the quality of your system, as you can't test quality into a system. If used properly, it can increase your productivity while decreasing cost, and most importantly it can keep you from increasing defect rates faster than your rate of organizational growth during periods of rapid hiring.

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 29—Failing to Design for Rollback Is Designing for Failure

What: Always have the ability to roll back code.

When to use: Ensure that all releases have the ability to roll back, practice it in a staging or QA environment, and use it in production when necessary to resolve customer incidents.

How to use: Clean up your code and follow a few simple procedures to ensure that you can roll back your code.

Why: If you haven't experienced the pain of not being able to roll back, you likely will at some point if you keep playing with the "fix-forward" fire.

Key takeaways: Don't accept that the application is too complex or that you release code too often as excuses that you can't roll back. No sane pilot would take off in an airplane without the ability to land, and no sane engineer would roll code that he or she could not pull back off in an emergency.

Risk reduction: High

Cost: Low

Benefit and priority: Very High—1

Rule 30—Remove Business Intelligence from Transaction Processing

What: Separate business systems from product systems and product intelligence from database systems.

When to use: Anytime you are considering internal company needs and data transfer within, to, or from your product.

How to use: Remove stored procedures from the database and put them in your application logic. Do not make synchronous calls between corporate and product systems.

Why: Putting application logic in databases is costly and represents scale challenges. Tying corporate systems and product systems together is also costly and represents similar scale challenges as well as availability concerns.

Key takeaways: Databases and internal corporate systems can be costly to scale due to license and unique system characteristics. As such, we want them dedicated to their specific tasks. In the case of databases, we want them focused on transactions rather than product intelligence. In the case of back-office systems (business intelligence), we do not want our product tied to their capabilities to scale. Use asynchronous transfer of data for business systems.

Risk reduction: High

Cost: Medium

Benefit and priority: High—2

Rule 31—Be Aware of Costly Relationships

What: Be aware of relationships in the data model.

When to use: When designing the data model, adding tables/columns, or writing queries, consider how the relationships between entities will affect performance and scalability in the long run.

How to use: Think about database splits and possible future data needs as you design the data model.

Why: The cost of fixing a broken data model after it has been implemented is likely 100x as much as fixing it during the design phase.

Key takeaways: Think ahead and plan the data model carefully. Consider normalized forms, how you will likely split the database in the future, and possible data needs of the application.

Risk reduction: Low

Cost: Low

Benefit and priority: Medium—3

Rule 32—Use the Right Type of Database Lock

What: Be cognizant of the use of explicit locks and monitor implicit locks.

When to use: Anytime you employ relational databases for your solution.

How to use: Monitor explicit locks in code reviews. Monitor databases for implicit locks and adjust explicitly as necessary to moderate throughput. Choose a database and storage engine that allow flexibility in types and granularity of locking.

Why: Maximize concurrency and throughput in databases within your environment.

Key takeaways: Understand the types of locks and manage their usage to maximize database throughput and concurrency. Change lock types to get better utilization of databases, and look to split schemas or distribute databases as you grow. When choosing databases, ensure that you choose one that allows multiple lock types and granularity to maximize concurrency.

Risk reduction: High

Cost: Low

Benefit and priority: Very High—1

Rule 33—Pass on Using Multiphase Commits

What: Do not use a multiphase commit protocol to store or process transactions.

When to use: Always pass or alternatively never use multiphase commits.

How to use: Don't use them; split your data storage and processing systems with Y or Z axis splits.

Why: A multiphase commit is a blocking protocol that does not permit other transactions to occur until it is complete.

Key takeaways: Do not use multiphase commit protocols as a simple way to extend the life of your monolithic database. They will likely cause it to scale even less and result in an even earlier demise of your system.

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 34—Try Not to Use `Select for Update`

What: Minimize the use of the `FOR UPDATE` clause in a `SELECT` statement when declaring cursors.

When to use: Always.

How to use: Review cursor development and question every `SELECT FOR UPDATE` usage.

Why: Use of `FOR UPDATE` causes locks on rows and may slow down transactions.

Key takeaways: Cursors are powerful constructs that when properly used can actually make programming faster and easier while speeding up transactions. But `FOR UPDATE` cursors may cause long-held locks and slow transactions. Refer to your database documentation to determine whether you need to use the `FOR READ ONLY` clause to minimize locks.

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 35—Don't Select Everything

What: Don't use `Select *` in queries.

When to use: Always use this rule (or, put another way, never select everything).

How to use: Always declare what columns of data you are selecting or inserting in a query.

Why: Selecting everything in a query is prone to break things when the table structure changes and it transfers unneeded data.

Key takeaways: Don't use wildcards when selecting or inserting data.

Risk reduction: High

Cost: Low

Benefit and priority: Very High—1

Rule 36—Design Using Fault-Isolative "Swim Lanes"

What: Implement fault isolation zones or *swim lanes* in your designs.

When to use: Whenever you are beginning to split up persistence tiers (e.g., databases) and/or services to scale.

How to use: Split up persistence tiers and services along the Y or Z axis and disallow synchronous communication or access between fault-isolated services and data.

Why: Increase availability and scalability. Reduce both incident identification and resolution. Reduce time to market and cost.

Key takeaways: Fault isolation consists of eliminating synchronous calls between fault isolation domains, limiting asynchronous calls and handling synchronous call failure, and eliminating the sharing of services and data between swim lanes.

Risk reduction: High

Cost: High

Benefit and priority: Medium—3

Rule 37—Never Trust Single Points of Failure

What: Never implement and always eliminate single points of failure.

When to use: During architecture reviews and new designs.

How to use: Identify single instances on architectural diagrams. Strive for active/active configurations.

Why: Maximize availability through multiple instances.

Key takeaways: Strive for active/active rather than active/passive solutions. Use load balancers to balance traffic across instances of a service. Use control services with active/passive instances for patterns that require singletons.

Risk reduction: High

Cost: Medium

Benefit and priority: High—2

Rule 38—Avoid Putting Systems in Series

What: Reduce the number of components that are connected in series.

When to use: Anytime you are considering adding components.

How to use: Remove unnecessary components, collapse components, or add multiple parallel components to minimize the impact.

Why: Components in series are subject to the multiplicative effect of failure.

Key takeaways: Avoid adding components to your system that are connected in series. When it is necessary to do so, add multiple versions of that component so that if one fails, others are available to take its place.

Risk reduction: Medium

Cost: Medium

Benefit and priority: Medium—3

Rule 39—Ensure That You Can Wire On and Off Features

What: Create a framework to disable and enable features of your product.

When to use: For functionality that is newly developed, noncritical, or dependent on a third party, consider using a wire-on/off framework.

How to use: Develop shared libraries to allow automatic or on-demand enabling and disabling of services. See Table 9.5 for recommendations.

Why: Turning off broken functionality or noncritical functionality in order to protect and preserve critical functionality is important for end users.

Key takeaways: Implement wire-on/wire-off frameworks whenever the cost of implementation is less than the risk and associated cost. Work to develop shared libraries that can be reused to lower the cost of future implementation.

Risk reduction: Medium

Cost: High

Benefit and priority: Low—4

Rule 40—Strive for Statelessness

What: Design and implement stateless systems.

When to use: During design of new systems and redesign of existing systems.

How to use: Choose stateless implementations whenever possible. If stateful implementations are warranted for business reasons, refer to Rules 41 and 42.

Why: The implementation of state limits scalability, decreases availability, and increases cost.

Key takeaways: Always push back on the need for state in any system. Use business metrics and multivariate (or A/B) testing to determine whether state in an application truly results in the expected user behavior and business value.

Risk reduction: High

Cost: High

Benefit and priority: Medium—3

Rule 41—Maintain Sessions in the Browser When Possible

What: Try to avoid session data completely, but when needed, consider putting the data in users' browsers.

When to use: Anytime that you need session data for the best user experience.

How to use: Use cookies to store session data in the users' browsers.

Why: Keeping session data in the users' browsers allows the user request to be served by any Web server in the pool and reduces storage requirements.

Key takeaways: Using cookies to store session data is a common approach and has advantages in terms of ease of scale. One of the most concerning drawbacks is that unsecured cookies can easily be captured and used to log in to people's accounts.

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 42—Make Use of a Distributed Cache for States

What: Use a distributed cache when storing session data in your system.

When to use: Anytime you need to store session data and cannot do so in users' browsers.

How to use: Watch for some common mistakes such as a session management system that requires affinity of a user to a Web server.

Why: Careful consideration of how to store session data can help ensure that your system will continue to scale.

Key takeaways: Many Web servers or languages offer simple server-based session management, but these are often fraught with problems such as user affiliation with specific servers. Implementing a distributed cache allows you to store session data in your system and continue to scale.

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 43—Communicate Asynchronously as Much as Possible

What: Prefer asynchronous over synchronous communication whenever possible.

When to use: Consider for all calls between services and tiers. Implement whenever possible and definitely for all noncritical calls.

How to use: Use language-specific calls to ensure that requests are made in a nonblocking fashion and that the caller does not stall (block) awaiting a response.

Why: Synchronous calls stop the entire program's execution waiting for a response, which ties all the services and tiers together, resulting in cascading latency or failures.

Key takeaways: Use asynchronous communication techniques to ensure that each service and tier is as independent as possible. This allows the system to scale much farther than it would if all components were closely coupled together.

Risk reduction: High

Cost: Medium

Benefit and priority: High—2

Rule 44—Ensure That Your Message Bus Can Scale

What: Message buses can fail from demand like any other physical or logical system. They need to be scaled.

When to use: Anytime a message bus is part of your architecture.

How to use: Employ the Y and Z AKF axes of scale.

Why: To ensure that your bus scales to demand.

Key takeaways: Treat message buses like any other critical component of your system. Scale them ahead of demand using either the Y or Z axis of scale.

Risk reduction: High

Cost: Medium

Benefit and priority: High—2

Rule 45—Avoid Overcrowding Your Message Bus

What: Limit bus traffic to items of higher value than the cost to handle them.

When to use: On any message bus.

How to use: Value and cost-justify message traffic. Eliminate low-value, high-cost traffic. Sample low-value/low-cost and high-value/high-cost traffic to reduce the cost.

Why: Message traffic isn't "free" and presents costly demands on your system.

Key takeaways: Don't publish everything. Sample traffic where possible to ensure alignment between cost and value.

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 46—Be Wary of Scaling through Third Parties

What: Scale your own system; don't rely on vendor solutions to achieve scalability.

When to use: Whenever considering whether to use a new feature or product from a vendor.

How to use: Rely on the rules in this book for understanding how to scale, and use vendor-provided products and services in the most simplistic manner possible.

Why: Three reasons for following this rule: Own your destiny, keep your architecture simple, and reduce your total cost of ownership. Understand that the customer holds you, not the vendor, responsible for the scale and availability of your product.

Key takeaways: Do not rely on vendor products, services, or features to scale your system. Keep your architecture simple, keep your destiny in your own hands, and keep your costs in control. All three of these can be violated when relying on a vendor's proprietary scaling solution.

Risk reduction: High

Cost: Low

Benefit and priority: Very High—1

Rule 47—Purge, Archive, and Cost-Justify Storage

What: Match storage cost to data value, including removing data of value lower than the cost to store it.

When to use: Apply to data and its underlying storage infrastructure during design discussions and throughout the lifecycle of the data in question.

How to use: Apply recency, frequency, and monetization analysis to determine the value of the data. Match storage costs to data value.

Why: Not all data is of similar value to the business, and in fact data value often declines (or less often increases) over time. Therefore, we should not have a single storage solution with equal cost for all our data.

Key takeaways: It is important to understand and calculate the value of your data and to match storage costs to that value. Don't pay for data that doesn't have a stakeholder return.

Risk reduction: Medium

Cost: Medium

Benefit and priority: Medium—3

Rule 48—Partition Inductive, Deductive, Batch, and User Interaction (OLTP) Workloads

What: Partition and fault-isolate unique workloads to maximize overall availability, scalability, and cost effectiveness.

When to use: Whenever architecting solutions that are composed of analytics (inductive or deductive) and product (batch or user interactive) solutions.

How to use: Ensure that solutions supporting the four basic types of workloads (induction, deduction, batch, and user interactive/OLTP) are fault isolated from each other and that each exists in its own fault isolation zone.

Why: Each of these workloads has unique demand and availability requirements. Furthermore, each can impact the availability and response time of the other. By separating these into distinct fault isolation zones, we can ensure that they do not conflict with each other, and each can have an architecture that is cost-effective for its unique needs.

Key takeaways: Induction is the process of forming hypotheses from data. Deduction is the process of testing hypotheses against data to determine validity. Induction and deduction solutions should be separated for optimum performance and availability. Similarly batch, user interaction, and analytics workloads should be separated for maximum availability, scalability, and cost effectiveness. Separate analytics solutions into those meant for induction and deduction. Choose the right solution for each.

Risk reduction: High

Cost: Medium

Benefit and priority: High—2

Rule 49—Design Your Application to Be Monitored

What: Think about how you will need to monitor your application as you are designing it.

When to use: Anytime you add or change modules of your code base.

How to use: Build hooks into your system to record transaction times.

Why: Having insight into how your application is performing will help answer many questions when there is a problem.

Key takeaways: Adopt as an architectural principle that your application must be monitored. Additionally, look at your overall monitoring strategy to make sure you are first answering the question “Is there a problem?” and then the “Where?” and “What?”

Risk reduction: Medium

Cost: Low

Benefit and priority: High—2

Rule 50—Be Competent

What: Be competent, or buy competency in, for each component of your architecture.

When to use: For any solution delivered online.

How to use: For each component of your product, identify the team responsible and level of competency with that component.

Why: To a customer, every problem is *your* problem. You can’t blame suppliers or providers. You provide a service—not software.

Key takeaways: Don’t confuse competence with build-versus-buy or core-versus-context decisions. You can buy solutions and still be competent in their deployment and maintenance. In fact, your customers demand that you do so.

Risk reduction: High

Cost: Low

Benefit and priority: Very High—1

A Benefit/Priority Ranking of the Scalability Rules

As you would expect, the distribution of rules is fairly normal but shifted toward the high end of benefit and priority. There are of course no rules that were ranked Very Low since they would not have made the cut for inclusion in the list. The following sections group the 50 rules by benefit/priority for ease of reference.

Very High—1

Rule 19	Relax Temporal Constraints
Rule 25	Make Use of Object Caches
Rule 29	Failing to Design for Rollback Is Designing for Failure
Rule 32	Use the Right Type of Database Lock
Rule 35	Don’t Select Everything
Rule 46	Be Wary of Scaling through Third Parties
Rule 50	Be Competent

High—2

Rule 1	Don’t Overengineer the Solution
Rule 7	Design to Clone or Replicate Things (X Axis)
Rule 10	Design Your Solution to Scale Out, Not Just Up
Rule 11	Use Commodity Systems (Goldfish Not Thoroughbreds)
Rule 14	Use Databases Appropriately
Rule 15	Firewalls, Firewalls Everywhere!
Rule 22	Cache Ajax Calls
Rule 26	Put Object Caches on Their Own “Tier”
Rule 27	Learn Aggressively
Rule 28	Don’t Rely on QA to Find Mistakes
Rule 30	Remove Business Intelligence from Transaction Processing
Rule 33	Pass on Using Multiphase Commits
Rule 34	Try Not to Use `Select for Update`
Rule 37	Never Trust Single Points of Failure
Rule 41	Maintain Sessions in the Browser When Possible
Rule 42	Make Use of a Distributed Cache for States
Rule 43	Communicate Asynchronously as Much as Possible
Rule 44	Ensure That Your Message Bus Can Scale
Rule 45	Avoid Overcrowding Your Message Bus
Rule 48	Partition Inductive, Deductive, Batch, and User Interaction (OLTP) Workloads
Rule 49	Design Your Application to Be Monitored

Medium—3

Rule 2	Design Scale into the Solution (D-I-D Process)
Rule 3	Simplify the Solution Three Times Over
Rule 4	Reduce DNS Lookups
Rule 5	Reduce Objects Where Possible
Rule 6	Use Homogeneous Networks
Rule 8	Design to Split Different Things (Y Axis)
Rule 9	Design to Split Similar Things (Z Axis)
Rule 12	Scale Out Your Hosting Solution
Rule 16	Actively Use Log Files
Rule 18	Stop Redirecting Traffic
Rule 20	Leverage Content Delivery Networks

Rule 21	Use `Expires` Headers
Rule 23	Leverage Page Caches
Rule 24	Utilize Application Caches
Rule 31	Be Aware of Costly Relationships
Rule 36	Design Using Fault-Isolative “Swim Lanes”
Rule 38	Avoid Putting Systems in Series
Rule 40	Strive for Statelessness
Rule 47	Purge, Archive, and Cost-Justify Storage

Low—4

Rule 13	Design to Leverage the Cloud
Rule 17	Don’t Check Your Work
Rule 39	Ensure That You Can Wire On and Off Features

Very Low—5

N/A

Summary

This chapter was a summary of the 50 rules in this book. Additionally, we provided a method by which these rules can be prioritized for a generic Web-based business looking to re-architect its platform in an evolutionary fashion. The prioritization does not mean as much for a business just starting to build its product or platform because it is much easier to build in many of these rules at relatively low cost when you are building something from scratch.

As with any rule there are exceptions, and not all of these rules will apply to your specific technology endeavors. For instance, you may not employ traditional relational databases, in which case our database rules will not apply to you. In some cases, it does not make sense to implement or employ a rule due to cost constraints and the uncertainty of your business. After all, as many of our rules imply, you don’t want to overcomplicate your solution, and you want to incur costs at an appropriate time so as to maintain profitability. Rule 2 is a recognition of this need to scale cost-effectively; where you can’t afford the time or money to implement a solution today, at least spend some amount of comparatively cheap time deciding how the solution will look when you do implement it. One example might be to wait to implement a scalable (fault-isolated and Y or Z axis-scaled) message bus. If you don’t implement the solution in code and systems infrastructure, you should at least discuss how you will make such an implementation in the future if your business becomes successful.

Similarly, there are exceptions with our method of prioritizing these rules. We have applied a repeatable model that encapsulates our collective learning across many companies. Because the result is, in a fashion, an average, it is subject to the same problem as all averages: an attempt to describe an entire population is going to be wrong for many specific data points. Feel free to modify our mechanism to fit your specific needs.

A great use for this chapter is to select a number of rules that fit your specific needs and codify them as architectural principles within your organization. Use these principles as the standards employed within software and infrastructure reviews. The exit criteria for these meetings can be complete adherence to the set of rules that you develop. Architectural reviews and joint architectural development sessions can similarly employ these rules to ensure adherence to principles of scalability and availability.

Whether your system is still in the design phase on the whiteboard or ten years old with millions of lines of code, incorporating these rules into your architecture will help improve its scalability. If you're an engineer or an architect, make use of these rules in your designs. If you are a manager or an executive, share these rules with your teams. We wish you the best of luck with all your scalability projects.

This page intentionally left blank

Index

This page intentionally left blank