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

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.

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

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.

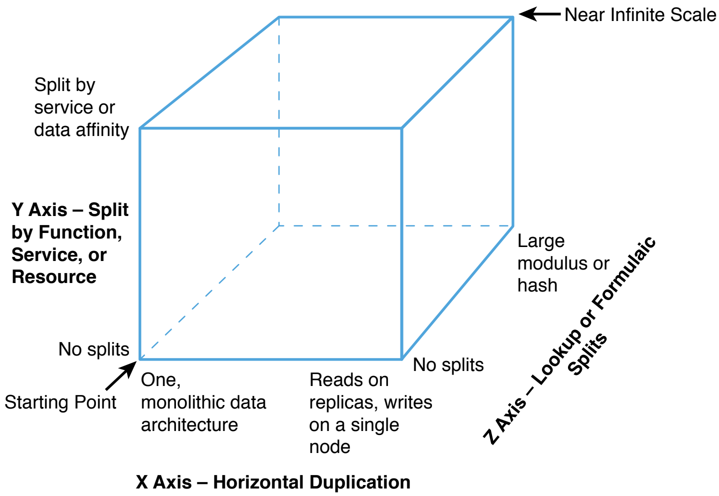

Figure 2.1 AKF Scale Cube

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.

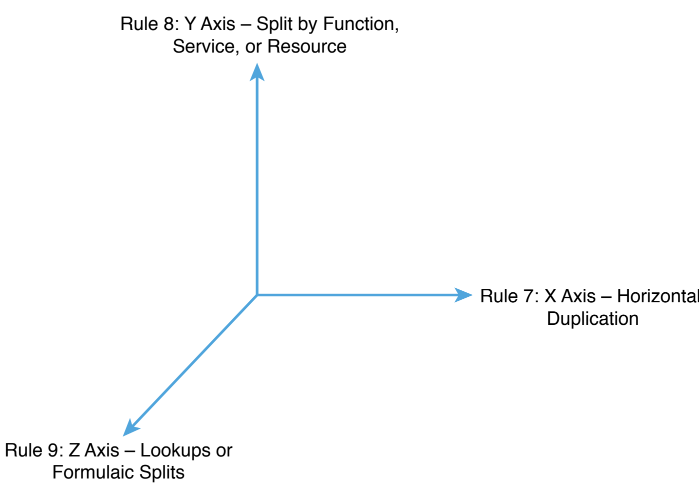

Figure 2.2 Three axes of scale

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.

Index

T

Z