The Method R Guide to

MASTERING ORACLE TRACE DATA

Second Edition

Cary V. Millsap

Method R Corporation · Southlake, Texas

Cary Millsap has been part of the Oracle community since 1989. He is the founder and president of Method R Corporation and a performance specialist at Accenture Enkitec Group. He is widely known in the Oracle community as a speaker, teacher, writer, and consultant. He is an Oracle ACE Director and a Founding Partner of the Oak Table Network. He wrote *Optimizing Oracle Performance* with Jeff Holt (for which they were named *Oracle Magazine Authors of the Year* in 2004). He is a co-author of *Oracle Insights: Tales of the Oak Table*, and he is published in *Communications of the ACM* and *ACM Queue*. Cary is also a designer and developer of the Method R software tools described in this book, which help professionals around the world manage Oracle performance. He blogs at carymillsap.blogspot.com and tweets at @CaryMillsap.

Method R Corporation, Southlake, Texas 76092
http://method-r.com
© 2011, 2016 by Cary V. Millsap
All rights reserved. First edition published 2011. Second edition 2016.
Revised August 4, 2016.

25 24 23 22 21 20 19 18 17 16 2 3 4 5

ISBN-13: 978-1518897887
ISBN-10: 1518897886

Editor, compositor, indexer, cover designer: Cary Millsap
Reviewers: Jeff Holt, Christian Antognini, Martin Berger, Guðmundur Jósepsson, Nancy Spafford, Neto from Brazil
Copy editor: Dan Gill

This book was prepared with the following software versions:

Method R Profiler version 6.1.0
Method R Tools version 4.1.0

Method R, Method R Workbench, Method R Profiler, Method R Tools, Method R Trace, and their respective logos are trademarks of Method R Corporation. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

The author has made his best efforts to prepare this material to be accurate and complete. The content is based upon final release software whenever possible; however portions of the material may be based upon pre-release versions supplied by software manufacturers. Method R Corporation makes no representation or warranties of any kind about the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this material.

Contents

Foreword by Christian Antognini *i*

Foreword by Martin Berger *iii*

Preface to the Second Edition *v*

Preface to the First Edition *vii*

Acknowledgments *ix*

List of Abbreviations *xi*

Software Used in this Book *xiii*

Part One

1 Thinking Clearly about Performance

An Axiomatic Approach 3
What Is Performance? 5
Performance as an Attribute 5
Response Time vs. Throughput 6
Percentile Specifications 7
Problem Diagnosis 9
The Sequence Diagram 10
The Profile 12
Latency, Call Count, and Mean 13
The Bottleneck 14
Amdahl's Law 15
Skew 17
Minimizing Risk 18
Efficiency 19
Load 20
Deterministic and Random Arrivals 20
Queueing Delay 20
Queueing Theory 21
Coherency Delay 22
Managing Load 23
Capacity Planning 26
Parallelism 26
Performance Testing 27
Measuring 28

Performance Is a Feature 29
Three Axioms of Performance 30
Method R 31
Eliminating Calls 33

2 Making Friends with the Oracle Database

Making Friends Begins with a Common Language 35
Where Software Speed Comes In 36
Feedback 37
Interpreting Your Trace File 38
Trace File Guided Tour 38
Optimizing the Query Program 41
Optimizing Begins with Measuring 43
On Specification Legitimacy 44
Trace File Guided Tour #2 45
Optimizing the Insert Program 46
Measuring Is Vital 49
Summary 50
Trace File: Slow Query 51
Trace File: Improved Query 52
Trace File: Slow Insert 53
Trace File: Improved Insert 53

3 Tracing Oracle

Performance and Tracing 55
Why You Should Trace 55
What Happens When You Trace 56
Tracing with ALTER SESSION and ALTER SYSTEM 57
Cleaning Up Pending Traces (ALTER SYSTEM) 58
Tracing from within Your Code 59
Tracing Oracle Application Express 60
Tracing Oracle E-Business Suite 61
Tracing from Another Session 62
Tracing by Task Name 63
Cleaning Up Pending Traces (DBMS_MONITOR) 64
Oracle User Session Handle Attributes 65
Setting the User Session Handle Attributes with PL/SQL 65
Setting the Attributes from SQL*Plus 66
Setting the Attributes without Extra Database Calls 66
Oracle DMS 68
Finding Your Trace File 70
Performance Intrusion 71
Tracing in Multiplexed Architectures 71

Tracing While You Develop 72

4 Oracle Extended SQL Trace Data

Trace File Guided Tour 75
Trace File Alphabet 77
Trace File Alphabet: Database Calls 77
Timer Precision 78
Trace File Alphabet: System Calls 78
Trace File Alphabet: Cursor Identification 79
Trace File Alphabet: Row Source Operations 80
Trace File Alphabet: Local Placeholder Bindings 81
Trace File Alphabet: Remote Placeholder Bindings 82
Trace File Alphabet: Transaction Control 83
Trace File Alphabet: Application Identification 83
Trace File Flow 83
Trace File Arithmetic: Call Durations 84
Trace File Arithmetic: Total Response Time 85
Trace File Arithmetic: Unaccounted-for Between Calls 86
Tracing Traps 86
Commercial Trace File Tools 87
Trace File: Hello World 88

5 Oracle Database Timed Event Reference

Where to Begin 91
Responding to Commonly Occurring Timed Events 92
What Does a Good Profile Look Like? 94
The 10-LIO Rule of Thumb 95
Oracle Timed Event Instrumentation 96
Tracing Oracle Kernel System Calls 96
Some Tricks 99
Oracle Timed Event Example 100
SQL*Net message from client 101
The Bug with SQL*Net message to client 102

Part Two

6 The Sample File

Method R Profiler Walk-Through 107
Fixing the Query 112
Customizing Method R Profiler 113

Method R Tools Walk-Through: mrls 114
Introduction to mrskew 115
Differences between Method R Profiler and mrskew 116
Method R Tools Walk-Through: mrskew 118
Conclusion 120
Exercises 120

7 Trace File Time Scoping

The Sloppy Trace 123
“Idle” Time 125
The First Rule of Using Trace Data 125
Finding and Eliminating the Time-Scope Problem 125
The Method R Tools Call Remover: mrcallrm 127
Diagnosing the Repaired Trace File 129
Conclusion 130
Exercises 130

8 Multiple Sessions in a Trace File

Weird Profiler Output 131
The Log File 132
Oracle’s trcsess Utility 133
Conclusion 135
Exercises 136

9 Array Fetch Size

Oracle Microseconds Problem 137
Performance Analysis 138
Predicting the Impact of a Repair 139
Changing the Array Size in Java 141
Are We Done Yet? 142
Conclusion 143
Exercises 143

10 Skew in Disk Read Latencies

The Profile 146
Skew in Read Call Durations 146
Skew in Read Call Sizes 147
Skew in File IDs 148
Skew in Object IDs 148
Skew in SQL Statements 149

Skew in Time of Execution 149
Conclusion 150
Exercises 151

11 Network I/O

The Trace File 153
Configuring Method R Profiler 154
Responding to the Profile 155
Testing the Hypothesis 156
The Result 157
Why the Upgrade Hurt Performance 157
Conclusion 158
Exercises 158

12 Missing Index

The First Statement 160
Skew in the LIO Counts 160
Root Cause 161
Proposed Remedy 163
Predicting the Impact 163
The Second Statement 164
Predicting the Impact 165
Conclusion 165
Exercises 166

13 Unaccounted-for Time

The “Before” Trace 167
Unaccounted-for Time 168
Proposed Remedy 169
Eliminating Calls 170
Predicting the Impact 171
The “After” Trace 173
Where Did We Go Wrong? 173
Conclusion 174
Exercises 175

14 SQL Sharing

How to Share SQL 177
Unsharable SQL 178
A Profile 178

Drilling In 180
Conclusion 180
Exercises 180

15 Listing Database Calls in Chronological Order

Parent-Child Relationships 184
Seeing Under the Covers 185
Conclusion 185
Exercises 186

16 Connection Pooling

Oceans and Islands 188
Oceans and Islands—Oh, and Rivers 188
Anatomy of the Experience ID 190
Generating Individual Profiles 191
The Difference between == and = 191
Diagnosing the Individual Profile 192
Isolating an Experience 192
Excerpting a Trace File 193
Conclusion 194
Exercises 194

17 Instrumenting Your Code

Instrumentation Example (SQL*Plus) 195
Instrumented Experience Duration Reporting 198
Instrumenting a Real Application 198
Instrumentation Example (Java) 200
Analyzing a Whole Trace File Directory 201
Drilling In 202
Why So Much Network I/O? 203
Predicting the Impact 204
The Value of Instrumentation 204
Finding Expensive Executions 205
Conclusion 207
Exercises 207

Part Three

18 Perspective

Why Is Proving So Important? 211
The Four Magic Questions about Performance 212
How Can You Avoid Performance Problems? 212
What Are You Supposed to See? 212
Where Do You Look? 213
How Do You Respond? 213
What You Get by Looking at Trace Files 214
Why You Should Trace in Development 216
Why You Should Trace in Testing 217
Why You Should Trace in Production 218

19 Why Trace?

Jon Adams 221
Christian Antognini 221
Tony Aponte 221
Martin Bach 221
Martin Berger 222
Stephen Butterworth 222
Wolfgang Breitling 222
Luca Canali 222
Ron Crisco 222
Dallas Deeds 223
Dominic Delmolino 223
Sergey Demin 223
Kirti Deshpande 223
Mark Farnham 223
Ken Ferlita 224
Iggy Fernandez 224
Lothar Flatz 224
Doug Gault 224
Dimitri Gielis 224
Alex Gorbachev 224
Tim Gorman 225
Jeff Gracz 225
Martin Haltmayer 225
Jeff Holt 225
Emile Hoogenkamp 225
Charles Hooper 226
Frits Hoogland 227

Lasse Jenssen 227
Randy Johnson 227
Guðmundur Jósepsson 227
Stefan Koehler 228
Niall Litchfield 228
Connor McDonald 228
Mary Elizabeth McNeely 228
Karen Morton 228
Arup Nanda 229
Neto from Brazil 229
Tanel Põder 229
Vince Poore 230
Kellyn Pot'Vin-Gorman 230
Jim Rice 230
Robyn Sands 230
Jože Senegačnik 231
Nancy Spafford 231
Jared Still 231
Patrik Stoss 231
Lars Johan Ulveseth 231
Graham Wood 232
Andrew Zitelli 232

Glossary 233

Bibliography 237

Index 241

Colophon 253

Foreword by Christian Antognini

In late 1999, through an Oracle Support note, I had my first encounter with extended SQL trace. From the beginning, it was clear that the information it provided was essential for understanding what an application is doing when it is connected to Oracle Database. At the same time, I was very disappointed that no tool was available for analyzing extended SQL trace files for the purpose of leveraging their content. But even worse, at that moment the widespread tuning methods were more similar to black art than computer science. As Cary would say… it was an age of darkness.

In 2003, when O'Reilly published *Optimizing Oracle Performance* (by Cary Millsap with Jeff Holt), the story changed. In fact, that book introduced a simple (but not simplistic) method, based on extended SQL trace, for optimizing the performance of Oracle Database applications: Method R. The four-step method was so concise that it could be summarized in eleven lines of text (page 20 of *Optimizing Oracle Performance*). Despite that, Method R was (and is) extremely powerful. It has been a game changer.

The remarkable achievement of the book you are reading is not only to give you a distilled and enhancement version of all the key concepts introduced by *Optimizing Oracle Performance*, but also to cover additional material about fundamental topics like instrumentation. A part that has been enormously enhanced is the one that covers the case studies. In fact, about half of the book is devoted to it. Case studies are very important because they make much easier for you to grasp, based on real examples, how to apply what you learned in the theoretical part of the book. In addition, for the users of the Method R Profiler and Method R Tools, the second part is very useful to realize how to fully take advantage of those powerful utilities.

Without any doubts, this book is the definitive guide to mastering Oracle trace data.

Christian Antognini

February 2016
Monte Carasso, Switzerland

Foreword by Martin Berger

Performance. That's what nearly everybody in every industry is searching for. Many fail. To many people, performance is a mystic secret, a Holy Grail. In the Grail stories from medieval times, the heroes were people we could never be. How they were selected and what tools they used were based on luck. And magic.

Like those ancient heroes, many people today seek better performance with methods based on stories they've read, experiences they've had or heard about, or checklists that have worked for their colleagues. But there is a better way. Cary Millsap has found it.

I got into contact with *The Method R Guide to Mastering Oracle Trace Data* in a course Cary taught in 2012. At that time I had already read several books about performance. I knew about Oracle's data dictionary views, SQL trace, and even quite a bit about Oracle internals, but still all this knowledge did not enable me to support application performance issues in any structured way.

Cary's "axiomatic" approach is different. He treats performance like a logical puzzle. Mathematical, but not out of reach. He shows you how to understand exactly what is happening in an Oracle system, which begins with measuring the right things. The most important thing he teaches is the repeatable process that ties it all together, that helps you make the right decisions.

With this basic change, he helps you find your Grail. There is no longer any need for magic. Or luck. With just a little work, we ourselves can be performance heroes.

Over the past three years I have managed to practice and extend my skills to translate Oracle trace files into answers for other DBAs, architects, developers, and even managers. That's the value that begins with this book.

Martin Berger

January 2016
Vienna, Austria

Preface to the Second Edition

When I wrote the first edition of this book in 2011, I needed a book for the course I was teaching. For a long time, I had been frustrated with idea that printouts of slides could pass for course material (Edward Tufte's work gave me the structure I needed to understand my frustration). Documents that help people learn need to consist of complete, fully-formed sentences, not just lists of prominent words. Most of what I needed had been written already, and published at method-r.com in the form of free downloads. Chapter 1 had reached almost 30,000 people this way.

In the nearly five years since publishing that first edition, I have taught its contents in dozens of classrooms and discussed the material with hundreds of people and learned a lot. One of the most important things I have learned is the profound value of *worked examples*. Worked examples are how our children learn our language, they are how people learn "unteachable" skills like chick sexing, and they are how my students learn how to master Oracle trace data. Worked examples generate more conversations among more students—and conversations that are far more interesting and enlightening—than any other technique for teaching that I've ever seen.

In the first edition, there were 31 pages of worked examples. This second edition gives you 101 pages of worked examples. I could not have done this without tools. In this book, I show you exactly which tools I use and how I use them. Just like I use a commercial tool—Adobe InDesign—to publish content like the book you're holding now, I use a commercial tool—Method R Workbench—to process my Oracle trace data. If you use Method R software too, then you'll be able to solve problems exactly as I show here. If you do not, then you will need to do your own digging, but you'll still know what you're looking for.

The second edition is over twice as big as the first, and it's not just the worked examples that are new. I have added important new sections to chapter 1 about latency, axioms of performance, Method R the method, and why eliminating calls is the cornerstone of optimizing a system. Chapter 3 got so big that I split it into two: one gives more detailed advice about tracing and finding your trace files, and the other provides a compact and complete reference for understanding the story your trace file is trying to tell you. The new chapter 5 is expanded to include new Oracle timed events, it shows what a good profile should look like, and it explains the 10-LIO rule of thumb that my colleagues and I have recommended since the early 2000s.

In the new part two, your favorite stories from the first edition are all still there, but they are bigger and more complete, exploring questions that were left open in the first edition. There are new stories, too, which explain how to deal with unaccounted-for time, multi-session trace files, and code that generates unsharable SQL. The new instrumentation chapter at the end of part two shows how to measure the response times your users are feeling, with precision that you've never had until now. The new part three gives the book more meaning than before. In chapter 18, I describe why I think trace files are still important in this modern age of ASH and AWR, and in a fun new chapter 19, a few dozen of my colleagues explain in their own words why trace files are still important to them.

One way I judge the quality of a project like this is by how much I learned while I worked on it. This one taught me a lot, thanks to all the students and conference attendees and reviewers and colleagues that have helped me along the way. I hope you enjoy the result.

Cary Millsap

April 2016
Irving, Texas

Preface to the First Edition

There are many options these days for measuring and monitoring Oracle Database application performance. This book is about the first one that ever made sense to me, and it's the one that endures in my mind to this day as the best foundation there is for understanding Oracle performance: the Oracle extended SQL trace feature.

In 1999, Jeff Holt and I embarked upon creating a method with a single starting point, so that no matter what your performance problem might be, you would always be able to solve it using a single, step-by-step method, with no need for guessing anywhere along the way. We studied many different ways of doing it. We embarked upon two projects in parallel: one to use Oracle fixed views (the ones whose names begin with `x$` and `v$` and `gv$`), and one to use Oracle extended SQL trace data. I chose the fixed view approach, because I knew that one was going to win, and I was the boss.

I was wrong. The limitations of using the fixed views were so numerous and so profound that after about a half a year's worth of dedicated effort, I abandoned the project and gave over to the side of the trace data. Trace files have two huge advantages:

- They are far easier to understand than the fixed views. Fixed views seem simple at first, but their omissions and inconsistencies soon make understanding what's in the fixed views themselves more difficult than solving the problem you're trying to diagnose. Trace files, on the other hand, look complicated at first, but their completeness and consistency pays off in the end. Trace files simply show the sequence of database calls and system calls that made up your response time; that's it. Everything Oracle Database does is laid out in the order that your calls have finished, complete with timing data. Once you know how to read a trace file, and especially *to process them with tools*, you're in a whole new world.
- They are easier to make available to the most important people in the performance formula: the developers who write the applications. Granting access upon the right Oracle fixed views to application developers is complicated, but it's nowhere near as complicated as trying to explain to a developer why a column called `seconds_in_wait` really isn't the number of seconds that a syscall has been waiting, that the value is updated every three seconds by the LGWR process regardless of whether the call being measured is complete, and …oh, never mind. Developers need to be thinking about developing software, not worrying about all the details of how the Oracle Database product works, and the idiosyncrasies of how those details are measured. Handing a trace file to a developer—or better yet, a *profile*, literally an invoice of response time—and saying, "Here's what is holding our code back today,…" It's a far better way to work.

This is not a book about discarding your Oracle fixed views or your monitoring tools that poll them. It's merely a book about how to look at Oracle performance another way, a way that has worked extraordinarily well for many of us. It's about new ways of thinking about performance using new tools for understanding aspects of your applications that you've never understood before.

Thank you for your interest in this book. I hope you enjoy the experience.

Cary Millsap

November 2011
Southlake, Texas

Acknowledgments

Every day when I go home, I'm grateful for the extraordinary people who live with me. The bulk of what I accomplish as a human being is through the love and grace of my wife and children—Mindy, Alexander, Nikolas, and Cathryne—and my mom Shirle (I miss you, Dad), my wife's parents Ted and Evelyn, and my sister-in-law Alison.

A lot of people have helped me with this work. If it weren't for Jeff Holt, then our profiler for Oracle wouldn't exist, and my personal understanding of Oracle and computer software in general (such as it is) would be far less complete. Martin Berger provided deeply insightful feedback that has improved both this book and our Method R software. Chris Antognini reviewed my work, answered technical questions, and stimulated software improvements. Lasse Jenssen helped improve the instrumentation sections for this second edition. Gummi Jósepsson added perspective and helped me make the whole book better. Sriram "Sri-Bob" Iyer and Doug Gault helped me write the new APEX tracing section. Christoph Ruepprich provided the trace file for the new chapter on multiple sessions per trace file. Jocke Treugut helped me optimize the "What lives here?" query in the sample file. Dimas Chbane helped me with the material about tracing Oracle EBS. Harold Palacio helped with the Java programming. Dan Gill helped me debug my writing. Nancy Spafford found the errors that all the rest of us had missed.

Ron Crisco has encouraged me and taught me many things about research and projects and progress and thinking and life. Ken Ferlita has written a lot of software for me and shared many conversations with me, helping me to learn and clarify my thinking. Karen Morton, whose time with us ended way too soon, always inspired me with her encouragement, her patience, and her clear mind. Kerry Osborne and Veronica Stigers welcomed my Method R colleagues and me into their workplace, this Accenture Enkitec Group, where inspiration and opportunity are everywhere; I am grateful to work among such a large and talented group of people who care about software performance. Thank you to Rachel Rutti, who introduced me to the work of Eli Goldratt, and to Jean Emerson, who introduced me to the work of Edward Tufte. Thank you to Kathy Sierra, who taught me the importance of using worked examples throughout the learning process. And thank you to Paul Brainard and my friends at Athletic Performance Enhancement (APE) for helping me stay healthy.

Other people who have helped or inspired me while preparing this book include: Alain Caron, Alex Gorbachev, Alyssa Freas, Andre van Winssen, Andrew Zitelli, Andy Rivenes, Anjo Kolk, Baron Schwartz, Bryn Llewellyn, Chet Justice, Carlos Sierra, Chris Date, Danny Menascé, Dave Ensor, Debra Lilley, Denny Chen, Dominic Delmolino, Don Knuth, Edward Tufte, Eli Goldratt, Emile Hoogenkamp, Frits Hoogland, Graham Wood, Jackie McIlroy, James Morle, Jared Still, Jimmy Brock, Jimmy Harkey, Jon Adams, Jon Bentley, Jonathan Lewis, Jorge Rimblas, Laura Nogaj, Mark Farnham, Mark Sweeney, Martin Bach, Mary Beth Mowery, Mauro Pagano, Michael Thomas, Mogens Nørgaard, Neil Gunther, Robyn Sands, Stefan Koehler, Steve Butterworth, Susie Campbell, Tanel Põder, Tim Hall, Tom Kyte, Tony Aponte, Toon Koppelaars, and my friends in the Oak Table Network.

And finally, thank you to Neto from Brazil, who provided prompt and exceptional feedback, and who wanted to be listed last for good luck.

List of Abbreviations

h	hour	3.600×10^{3} s = 3,600 s
min	minute	6×10^{1} s = 60 s
s	second	10^{0} s = 1 s
cs	centisecond	10^{-2} s = 0.01 s
ms	millisecond	10^{-3} s = 0.001 s
µs	microsecond	10^{-6} s = 0.000 001 s
	Oracle microsecond	1.024×10^{-6} s = 0.000 001 024 s
ns	nanosecond	10^{-9} s = 0.000 000 001 s
ps	picosecond	10^{-12} s = 0.000 000 000 001 s
fs	femtosecond	10^{-15} s = 0.000 000 000 000 001 s
as	attosecond	10^{-18} s = 0.000 000 000 000 000 001 s
EB	exabyte	10^{18} B = 1,000,000,000,000,000,000 B
PB	petabyte	10^{15} B = 1,000,000,000,000,000 B
TB	terabyte	10^{12} B = 1,000,000,000,000 B
GB	gigabyte	10^{9} B = 1,000,000,000 B
MB	megabyte	10^{6} B = 1,000,000 B
KB	kilobyte	10^{3} B = 1,000 B
B	byte	8×10^{0} b = 8 b
b	bit	10^{0} b = 1 b

Software Used in this Book

This book uses Unix-like commands like `ps`, `lsof`, `strace`, `grep`, `head`, `tail`, `nl`, `cut`, and so on.

The trace file manipulation utilities featured in this book are part of the Method R Workbench commercial software package, which is sold at http://method-r.com/store. Method R Workbench contains three Method R software products:

- **Method R Profiler** shows exactly how your programs spend your time, using color and structure to guide your understanding. With just a few clicks, you can answer the four magic questions about performance: How long? Why? What if? and What else?
- **Method R Tools** is a flexible and versatile utility suite that includes a trace file data mining tool, utilities for adjusting trace files, and more. With Method R Tools, you can answer questions about performance that you might never before have thought possible.
- **Method R Trace** is a zero-click trace file collector for users of the Oracle SQL Developer integrated development environment. It is the easiest way to create and securely retrieve perfect trace files, with no need to bother your database or system administrator.

Part One

The chapters in part one provide the background that you will need to understand how to optimize Oracle application performance using Oracle trace files. You'll learn the basics of computer software performance in a way that I hope you'll find refreshingly direct. You'll learn what kinds of performance information you should be looking for and why, and how Oracle trace files fit the requirement. Finally, you'll learn how to create trace files and how to understand the stories they tell.

1 Thinking Clearly about Performance

Throughout the brief history of our industry, many professionals have had great difficulty creating "high performance" as an attribute of complex computer systems. However, by understanding a few fundamental principles, the processes of solving and preventing performance problems become far simpler and more reliable. This chapter describes those principles, linking them together in a coherent journey covering the goals, the terms, the tools, and the decisions that you need to maximize your application's chance of having a long, productive, high-performance life. Examples in this chapter touch upon Oracle experiences, but the scope of the chapter is not restricted to Oracle products.

An Axiomatic Approach

When I joined Oracle Corporation in 1989, performance—what everyone called "Oracle tuning"—was difficult. Only a few people claimed they could do it very well, and those people commanded nice, high consulting rates. When circumstances thrust me into the "Oracle tuning" arena, I was quite unprepared. Recently, I've been introduced into the world of "MySQL tuning," and the situation seems very similar to what I saw in Oracle almost thirty years ago.

It reminds me a lot of the difficulties I had with algebra when I was about 13 years old. At that age, I had to appeal heavily to my "mathematical instincts" to solve equations like $3x + 4 = 13$. The problem with needing to appeal to one's mathematical instincts is this: what if you don't *have* any mathematical instincts? Does that mean that mathematics will be forever beyond your reach?

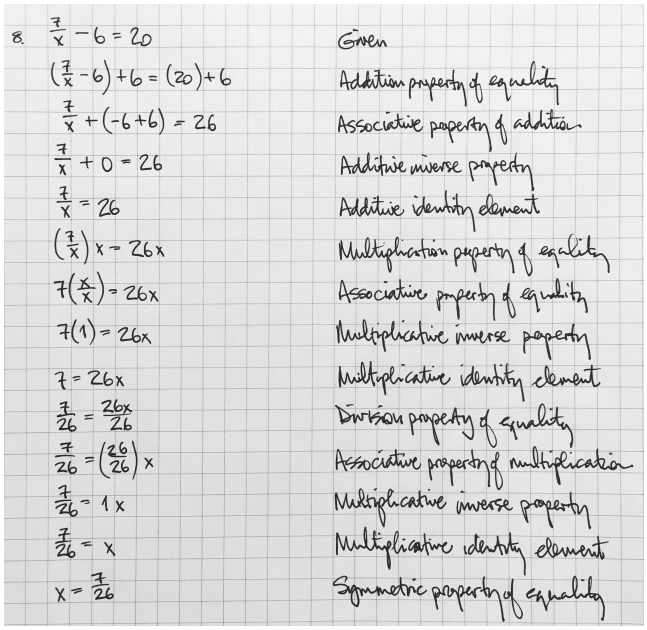
I can remember looking at a problem like "$3x + 4 = 13$; find x" and stumbling upon the answer $x = 3$ using trial and error. But what about a tougher problem like "$7/x - 6 = 20$; find x"? Then what?

The trial-and-error method of feeling my way through algebra problems worked—albeit sometimes slowly and uncomfortably—for easy equations, but it didn't scale as the problems got tougher. My problem was that I wasn't thinking clearly enough yet about algebra. I didn't understand yet how algebra worked and *why* it worked that way. My introduction at age fifteen to Jimmy Harkey put me on the road to solving that problem.

Mr. Harkey taught us what he called an *axiomatic approach* to solving algebra problems. He showed us a set of steps that worked every time (and he gave us plenty of homework to practice on).[1] In addition to working every time, by executing those steps, he made us *document*

[1] Millsap, "Axiomatic Approach."

our thinking as we worked. Not only were we thinking clearly, using a reliable and repeatable sequence of steps, we were *proving* to anyone who read our work that we were thinking clearly. Mr. Harkey already knew what x was. His chief concern was to understand what I was thinking while I worked through the problem. As a parent now who checks his children's homework several times a week, I wish more teachers taught better show-your-work habits. It's easier to debug my kids' thinking when their logging is switched on.



This is what homework and tests for Mr. Harkey looked like. He made us show each step, because Mr. Harkey cared more about the code path in our heads than our final answer. In essence, he required us to execute our work with logging switched on so that he could help us debug our thinking.

This was Mr. Harkey's axiomatic approach to algebra, geometry, trigonometry, and calculus: one small, logical, provable, easy-to-audit step at a time. Working this way for a few weeks earned me my first-ever feeling of confidence toward mathematics.

Naturally, I didn't realize it at the time, but of course *proving* was a skill that would be vital for my success in the world after school. In life, I've found that, of course, knowing things matters. But proving those things—to other people—matters more. Without good proving skills, it's difficult to be a good consultant, a good leader, or even a good employee.

My goal since the mid-1990s has been to create a similarly rigorous approach to Oracle performance optimization. Lately, I have expanded the scope of that goal beyond Oracle, to creation of an axiomatic approach to computer software performance optimization. My aim is to help you think clearly about how to optimize the performance of your computer software; …to help you really understand it.

What Is Performance?

If you google for the word *performance*, you get over a half a billion hits on concepts ranging from bicycle racing to the dreaded employee review process that many companies these days are learning to avoid. When I googled for *performance*, most of the top hits related to the subject of this chapter: **performance** is the time it takes for computer software to perform a task that you ask it to do.

And that's a great place to begin: the task. A **task** is a business-oriented unit of work. Tasks can nest: "Print Invoices" is a task; "Print One Invoice"—a subtask—is also a task. When a computer user talks about performance he usually means the time it takes for the system to execute some task. An individual execution of a task can result in a personal *experience* with that task. So, while a *task* corresponds to lines of code in your application and has a name like "Book Order," an **experience** corresponds to a single execution of a task. Each task implemented in a software application will typically be executed many times, resulting in many experiences. For example, the code to implement a "Book Order" task might be executed (and experienced) a million times per month.

Response time is the duration of an experience. For example, google.com reported that my search for the word *performance* had a response time of 0.24 s. The Google web page rendered that measurement right in my browser. This is evidence that Google values my perception of Google application performance. **Throughput** is the count of experiences that complete within a specified time interval. For example, "300 'Book Order' confirmations per second" is an expression of *throughput* for the "Book Order" task.

Performance as an Attribute

Have you ever heard someone say "the system is slow" or "the system is fast"? Let's explore what that means. Imagine an order processing system that consists of the following set of experiences. Is this system fast? Or is it slow?

	User	Task	Response time (s)	Requirement (s)	Fast/slow
1	fch	Book Order	1.019	2.0	Fast
2	rsm	Ship Order	3.528	5.0	Fast
3	djo	Pick Order	1.211	3.0	Fast
4	ffo	Book Order	2.716	2.0	Slow
5	fch	Book Order	1.917	2.0	Fast
6	ffo	Find Order	1.305	2.5	Fast
7	cra	Ship Order	14.287	5.0	Slow
8	jlm	Pick Order	3.278	3.0	Slow
9	tnp	Ship Order	1.911	5.0	Fast
10	ssw	Book Order	0.121	2.0	Fast

A list of ten user experiences on an order processing system. Is this system fast? Or slow? It's a bit of a dilemma.

It is a dilemma, isn't it? I've asked you to use just one word to characterize a ten-row table. The question automatically invites you to aggregate. So what do you say? Is the system fast, because it is fast for seven of the ten experiences? Or it is slow, because it is slow for the remaining three? How much more difficult would it be to characterize a system with 10 million experiences?

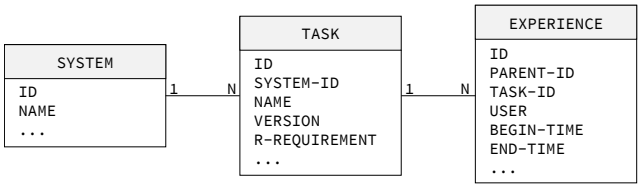
Perhaps you've noticed that the system is fast for user fch (experiences 1 and 5), and slow for user jlm (experience 8). So maybe it's easier to characterize a system as fast or slow for a given *user*. But that doesn't work either. Have a look at user ffo. She has experienced both a slow "Book Order" (experience 4) and a fast "Find Order" (experience 6). Ah, so maybe it's *tasks* that are fast or slow then. But no, you can't say that either, because "Book Order" is sometimes fast (experiences 1, 5, and 10) and sometimes slow (experience 4). Even if the "Book Order" experiences had all been fast (or all been slow), there's certainly no guarantee of performance homogeneity among all of the task's future executions.

So then, what can you say about the performance of this system? The answer is, not much, really. You *could*, of course, begin the surrogate measure game and start measuring something else about the system, like its CPU utilization or its memory pressure, but these aren't measures of *the time it takes your system to perform a task that someone has asked it to do.* They are not direct measures of *performance*.

The problem with the question of whether the system is fast or slow is the question itself. The attribute of *fast* or *slow* simply doesn't belong on the *system* entity. To the owner or user of a computer, the **system** is a set of *experiences*. The performance of those experiences is not necessarily consistent from one to the next, even for one user who runs the same task over and over. It doesn't make sense to talk about whether a system is fast or slow, because performance is not an attribute of a system…

Performance is an attribute of each individual *experience* with a system.

If we were to model performance relationally, the data model would look roughly like the figure below. Though the response time requirement might be an attribute of the *task*, the performance information itself—the begin and end times, which determine the response time—are unquestionably attributes of each *experience*.



The data model for the `system`, `task`, and `experience` entities makes it clear why performance—the time it takes for your system to perform a task you ask it to do—is an attribute of each individual experience.

Response Time vs. Throughput

Average throughput and average response time have a generally reciprocal type of relationship, but not exactly. The real relationship is more complex.

> ***Example:*** Imagine that you have measured your throughput at 10 executions per second for some benchmark. What, then, is your users' average response time? Is your average response time 1/10 = 0.1s per execution?

Imagine that your system processing this throughput had 10 parallel, independent, homogeneous service channels inside it (that is, it's a system with 10 independent, equally competent service providers inside it, each awaiting your request for service). Imagine further that each service channel is capable of processing a request for service in exactly 1 s. Then if 10 requests for service arrive simultaneously into the system, exactly 1 s later, 10 task executions will complete, each with a response time of exactly 1 s.

The relationship between average response time R and average throughput X is

$$R = \frac{N}{X} - Z,$$

where Z is the average duration between request arrivals into the system (called **think time**), and N is the number of service channels in the hardware configuration.[2]

Likewise, if you know your average response time R for a given task in your unit testing environment, you know how many service channels N you have, and you know your average think time Z, then you can predict your average throughput X as

$$X = \frac{N}{R + Z}.$$

Capacity planners use formulas like these to determine average throughput and response times,[3] but from models like these, you will not learn operational details about individual user experiences. You might learn from a model that your average "Book Order" throughput is 500 orders per second, but you won't learn that your actual "Book Order" throughput was only 117 orders per second between 2:00 P.M. and 3:00 P.M. last Wednesday. You might learn from a model that your average response time is 0.742 s per order, but you won't learn that Nancy had "Book Order" response times as high as 42.895 s from Golden, Colorado last Friday morning around 10:00 A.M. Models and formulas can inform you about average behavior, but many times understanding average behavior isn't enough. To manage performance well, you need to know actual throughput behavior and actual response time behavior, *as actual people experience them*. To know them both, you have to measure them both.

So, which is more important: response time, or throughput? In general, group managers care more about throughput than solo contributors do. For example, a sales clerk cares whether the response time of an afternoon report execution will require him to stay late after work, whereas the sales department manager cares additionally about whether the system can process all the orders that all her clerks will need to book today. In many circumstances, both response time and throughput are vital constraints.

> ***Example:*** A system owner has a business requirement that response time must be 1.0 s or less for a given task in 99% or more of executions (experiences), and the system must support a sustained throughput of 1,000 executions of the task within any arbitrary 10 min interval.

Percentile Specifications

In the prior section, I used the phrase "in 99% or more of executions" to qualify a response time expectation. Many people are more accustomed to statements like, "average response time must be *r* seconds or less." The percentile way of stating requirements maps better to the human experience.

> ***Example:*** Imagine that your response time tolerance is 1 s for some task that you execute on your computer every day. Further imagine that the lists of numbers shown below represent

[2] Jain, *Art of Performance Analysis*, 563.

[3] See, for example, the works of Jain, Menascé, Smith & Williams, and Gunther.

the measured response times of ten typical executions of that task. The average response time for each list is 1.000 s. Which one do you think you'd like better?

	Response time (s)	
	List A	List B
1	.924	.796
2	.928	.798
3	.954	.802
4	.957	.823
5	.961	.919
6	.965	.977
7	.972	1.076
8	.979	1.216
9	.987	1.273
10	1.373	1.320
mean	1.000	1.000

The average response time for each of these lists of response times is 1.000 s, yet the two lists depict clearly different user experiences.

Although the two lists have the same average response time, the lists represent significantly different experiences. In List A, 90% of response times were 1 s or less. In List B, only 60% of response times were 1 s or less. Stated in the opposite way, List B represents a set of user experiences of which 40% were unsatisfactory, but List A (having the same average response time as List B) represents only a 10% dissatisfaction rate.

In List A, the 90th percentile response time is 0.987 s. In List B, the 90th percentile response time is 1.273 s. These statements about percentiles are more informative than merely saying that each list represents an average response time of 1.000 s.

Percentile-based response time goals are more compelling requirement specifications because they match what people really care about. Users don't think about response times when they're good. But they notice and remember them when they're bad.

Users feel the variance, not the mean.[4]

A specification like, "Average response time for 'Book Order' must be less than 0.5 s" is not enough. You need something like, "The 90th percentile response time for 'Book Order' must be less than 1.0 s." An "average response time" specification is roughly the same as a 50th percentile specification. You probably need for your percentile to be at least 90 and possibly 99, or 99.9, or even more.

A percentile specification like this does a better job of measuring the variance that users feel, but it's still not quite enough.

Example: Continuing the previous example, with your 1 s response time tolerance, compare the familiar List A with the new List C shown here.

[4] GE, "Six Sigma."

	Response time (s)	
	List A	List C
1	.924	.091
2	.928	.109
3	.954	.134
4	.957	.136
5	.961	.159
6	.965	.172
7	.972	.185
8	.979	.191
9	.987	.207
10	1.373	8.616
mean	1.000	1.000

Both lists have identical average response times, and both yield the same success rate with respect to the specification, "90th percentile response time is less than 1 s," yet the two lists depict clearly different user experiences.

Again, these two lists depict identical average response times, and List C depicts a superior 90th percentile response time of 0.207 s (versus 0.987 s in List A). Yet, you might argue that, if users really do feel the variance and not the mean, then List C represents a worse overall response time experience, because the worst response time (8.616 s) is over *six times worse* than the worst response time in List A (1.373 s).

Here are depictions of two clearly different experiences, yet a simple 90th percentile specification wasn't enough to describe the distinction. A two-tier specification like the following resolves the problem:

"The 90th percentile response time for 'Book Order' must be less than 1.0 s, *and* the 99th percentile response time must be less than 2.0 s."

Stated in a way that perhaps more users can relate to:

Weak specification: "'Book Order' average response time must be less than 1.0 s."

Better specification: "'Book Order' must respond in less than 1.0 s in at least 90% of executions."

Best specification: "'Book Order' must respond in less than 1.0 s in at least 90% of executions, and in less than 2.0 s in at least 99% of executions."

Problem Diagnosis

The barrier to performance improvement is often the lack of a clear and useful problem statement. A good performance problem statement has two parts:

- *Current state*: a quantitative description of the way the system behaves now, expressed in terms of response time or throughput of one or more tasks.
- *Goal state*: a quantitative description of the way you want the system to behave, expressed in the same terms as the current state, and a quantitative description of the investment you're willing to make for the system to behave that way.

Bad problem statements typically match one of the following patterns:

- *Bad*: It doesn't name the task whose execution you're looking to improve. For instance, when a room full of people from all over the business is shouting, "Our whole [adjectives redacted] system is so slow we can't use it!" This statement provides no ability to focus; it gives you nowhere to begin, and it gives you no quantitative baseline from which to measure your progress.[5]
- *Bad*: It presumes the cause of the problem before the cause has been proven. Imagine stating your goal as "to upgrade to a higher-throughput storage device." Now imagine that your project successfully completes the upgrade, but the upgrade doesn't improve the performance of a key task that the business was hoping the upgrade would fix. In this case, the project is successful relative to its stated goal, but the real problem remains, because the real goal was not correctly identified.
- *Bad*: It does use the terms "response time" or "throughput," but only to quantify technical details about tasks instead of the tasks themselves. For instance, don't describe the throughput of disk I/O performance associated with a "Book Order" execution; talk about the throughput of the whole "Book Order" task itself.
- *Bad*: It is expressed indirectly in technical terms of "utilization," "latency," or "miss rate" instead of direct language that describes business results. Once I asked a CIO what his real performance goals were. He said, "I'd like the CPU utilization on my database server not to exceed 80% during our peak business hours." I told him that we could accomplish that goal by replacing his fast, high-end storage subsystems with really old, slow ones.[6] His real goal wasn't reduction of CPU utilization; it was to make his business task executions run faster. He figured that having leftover CPU capacity would ensure his real goal, but it's not necessarily so.

Say your real goal. Sometimes it takes a little work to figure out what your real goal really is. A good problem statement is expressed in non-technical language that *users* of systems use in their everyday work lives.

There's a problem, though, with trying to learn the current and goal states of a system's performance. What you want is something like this:

> "Response time of 'Book Order' is more than 20 s in many cases. We'll be happy when response time is 1 s or less in at least 95% of executions."

That sounds good in theory, but what if your user doesn't have a specific quantitative goal like "1 s or less in at least 95% of executions?" There are two quantities right there (1 and 95); what if your user doesn't know either one of them? Worse yet, what if your user *does* have specific ideas about his expectations, but those expectations are impossible to meet? How would you know what "possible" or "impossible" even is?

Let's work our way up to answering those questions. First, I want to introduce some tools.

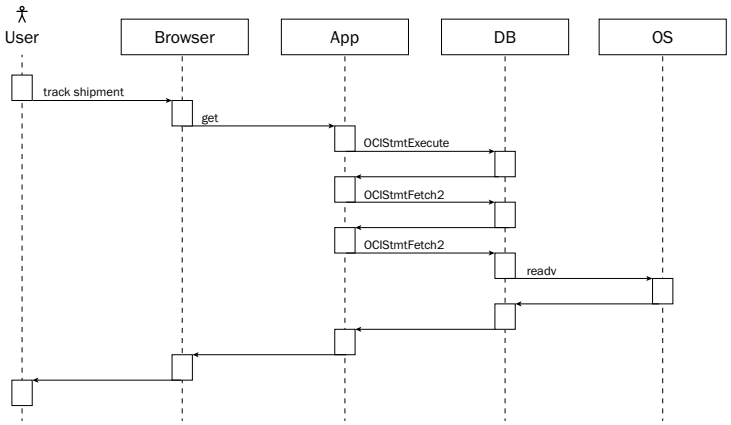
The Sequence Diagram

A **sequence diagram** is a type of graph specified in the Unified Modeling Language (UML), used to show the interactions between objects in the sequential order that those interactions occur. The sequence diagram is an exceptionally useful tool for visualizing response time. Here is a

[5] Millsap, "Now What?"

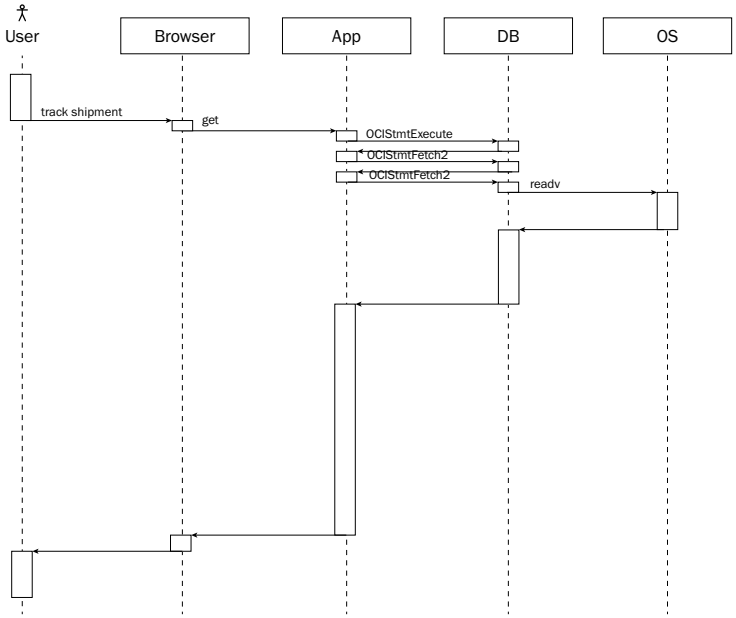
[6] To degrade I/O response times would cause processes to spend more time awaiting the results of read or write calls. When a process awaits a synchronous read or write call, it doesn't use CPU. Each I/O-consuming process would thus spend more time waiting for I/O calls to complete, causing many of its previously used CPU cycles in a given time interval to remain unused (while it waits for I/O). The net effect would be reduced CPU utilization.

standard UML sequence diagram for a simple application system composed of a browser, an application server, a database, and its host operating system:



This UML sequence diagram shows the interactions among a user, a browser, an application server, a database, and the database's host operating system.

Imagine now drawing the sequence diagram to scale, so that the distance between each "request" arrow coming in and its corresponding "response" arrow going out is proportional to the duration spent servicing the request, as follows:

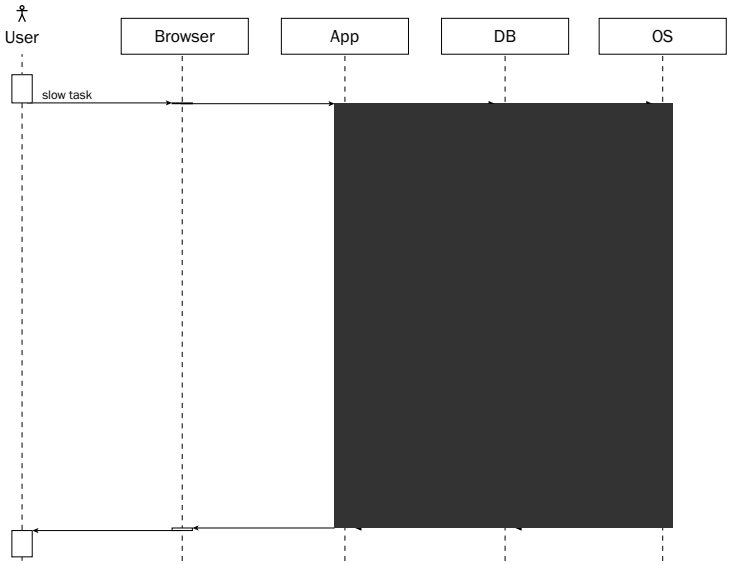


A UML sequence diagram drawn so that vertical distance in the diagram is proportional to the response time consumed.

With a sequence diagram like this, you have a good graphical representation of how the components represented in your diagram are spending your user's time. You can "feel" the relative contribution to response time by looking at the picture.

Sequence diagrams are just right for helping people conceptualize how their response is consumed on a given system, as one tier hands control of the task to the next. Sequence diagrams also work well to show how simultaneous threads of processing work in parallel. Sequence diagrams are good tools for analyzing performance outside of the information technology business, too.[7]

But there's a problem with sequence diagrams. Imagine that the task you're supposed to fix has a response time of 77,148 s (that's 21 h 25 min 48 s). In that roughly 21.4 h, running that task causes your application server to execute over 10,000 database calls, and your database to execute over 10,000,000 operating system calls. Here is what your sequence diagram for that task would look like:



A UML sequence diagram is incapable of showing what we need to see for an experience during which the application server executed over 10,000 database calls, and the database server executed over 10,000,000 operating system calls.

There are so many request and response arrows between the application and database tiers that you can't see any of the detail. Such a diagram would be nearly useless. Printing the sequence diagram on a very long scroll isn't a useful solution, because the resulting visual inspection would be just too difficult. The sequence diagram is a good tool for conceptualizing flow of control, but to think clearly about response time, you need something else.

The Profile

The sequence diagram doesn't scale well. To deal with tasks that have huge call counts, you need a convenient aggregation of the sequence diagram so that you can understand the most

[7] Millsap, "Global Entry."

important patterns in how your time has been spent. A **profile** is a tabular decomposition of response time, typically listed in descending order of component response time contribution. Here is a profile that matches the sequence diagram shown previously:

```
CALL-NAME                                          DURATION       %       CALLS      MEAN       MIN        MAX
-------------------------------------------------  -------------  ------  ---------  --------  --------  ---------
db file sequential read                            59,081.406102   76.6%  10,013,394  0.005900  0.000010  15.853019
log buffer space                                    6,308.758563    8.2%       9,476  0.665762  0.000004   1.010092
free buffer waits                                   4,688.730190    6.1%     200,198  0.023420  0.000004   1.021281
EXEC                                                4,214.190000    5.5%      36,987  0.113937  0.000000   5.400000
log file switch completion                          1,552.471890    2.0%       1,853  0.837815  0.000006   1.013093
db file parallel read                                 464.976815    0.6%       7,641  0.060853  0.000030   8.964706
log file switch (checkpoint incomplete)               316.968886    0.4%         351  0.903045  0.000003   1.014777
rdbms ipc reply                                       244.937910    0.3%       2,737  0.089491  0.000001   2.010042
undo segment extension                                140.267429    0.2%       1,411  0.099410  0.000001   0.108950
log file switch (private strand flush incomplete)     112.680587    0.1%         134  0.840900  0.002781   1.000239
17 others                                              23.367228    0.0%      58,126  0.000402  0.000000   5.045291
-------------------------------------------------  -------------  ------  ---------  --------  --------  ---------
TOTAL (27)                                         77,148.755600  100.0%  10,332,308  0.007467  0.000000  15.853019
```

The two defining characteristics of a profile are:

- The bottom-line duration (77,148.755 600 s in this example) must match the user response time experience that you're trying to diagnose.
- The durations in the table must sum to exactly this value; no more (that is, no overlaps) and no less (that is, no gaps).

If your profile meets these two criteria, then you have an unambiguous account of exactly the response time you want to analyze. From the data shown in this profile, you know that 76.6% of your user's response time is consumed by 10,013,394 `db file sequential read` calls.

A profile shows you where your code has spent your time and—sometimes even more importantly—where it has *not*. There is tremendous value in not having to guess about these things. With a profile, you can begin to formulate the answer to the question, "How long *could* this task run?" By now, you know this is an important question in the first step of the problem diagnosis process.

> **A profile shows where your code has spent its time and can answer the question, "How long *could* this task run?"**

Latency, Call Count, and Mean

In a profile like the one shown here, each duration (`DURATION`) is a sum of measured *latencies*, measured from the caller's perspective.

```
CALL-NAME                  DURATION       %   CALLS      MEAN       MIN        MAX
-------------------------  --------  ------  ------  --------  --------  --------
SQL*Net message from client  2.370973   49.7%  20,007  0.000119  0.000047  0.060656
PARSE                        1.842653   38.6%   5,001  0.000368  0.000000  0.070101
EXEC                         0.410591    8.6%   5,002  0.000082  0.000000  0.010015
SQL*Net message to client    0.065746    1.4%  20,007  0.000003  0.000001  0.030441
db file sequential read      0.055300    1.2%       2  0.027650  0.019335  0.035965
log file sync                0.017142    0.4%       3  0.005714  0.001956  0.012949
latch free                   0.005987    0.1%      19  0.000315  0.000008  0.002533
-------------------------  --------  ------  ------  --------  --------  --------
TOTAL (7)                    4.768392  100.0%  50,041  0.000095  0.000000  0.070101
```

A call's **latency** is the duration that elapses between when the caller makes the call and when the caller receives the response. The **call count** (`CALLS`) is a measured quantity as well, computed in the obvious manner of counting the number of calls that have been measured. The minimum (`MIN`) and maximum (`MAX`) values are also measured quantities.

The **mean** (or **average**) latency (`MEAN`), however, is a derived quantity. It is the arithmetic mean of the measured call latencies, computed as sum of the call latencies divided by the call count. It is not necessarily true that a subroutine's mean (or average) latency is equal to any actual subroutine call's latency.

> ***Example:*** Five calls to a subroutine consumed durations {0.2, 0.2, 0.2, 0.2, 4.2}s. The subroutine's total duration is 5.0s. The subroutine's average latency is 5.0s/5calls = 1.0s/call, but no individual subroutine call latency was 1.0s.

The Bottleneck

The resource that dominates the response time for a task execution is that execution's **bottleneck**. The bottleneck for an execution (for an experience) thus is simply the resource listed in the top line of that execution's profile.

> ***Example:*** Consider the following profile, which describes a 1,957.47s experience running a payroll application batch job. Which resource is the bottleneck?

```
CALL-NAME                      DURATION       %    CALLS      MEAN       MIN       MAX
-----------------------------  ------------  ------  -------  --------  --------  --------
SQL*Net message from client     984.010000   50.3%   95,161  0.010340  0.000000  0.310000
SQL*Net more data from client   418.820000   21.4%    3,345  0.125208  0.000000  0.270000
db file sequential read         279.340000   14.3%   45,084  0.006196  0.000000  0.050000
EXEC                            136.880000    7.0%   67,888  0.002016  0.000000  1.320000
PARSE                            74.490000    3.8%   10,098  0.007377  0.000000  0.090000
FETCH                            37.320000    1.9%   57,217  0.000652  0.000000  0.130000
latch free                       23.690000    1.2%   34,695  0.000683  0.000000  0.080000
log file sync                     1.090000    0.1%      506  0.002154  0.000000  0.050000
SQL*Net more data to client       0.830000    0.0%   15,982  0.000052  0.000000  0.020000
log file switch completion        0.280000    0.0%        3  0.093333  0.080000  0.110000
enqueue                           0.250000    0.0%      106  0.002358  0.000000  0.020000
SQL*Net message to client         0.240000    0.0%   95,161  0.000003  0.000000  0.010000
buffer busy waits                 0.220000    0.0%       67  0.003284  0.000000  0.020000
db file scattered read            0.010000    0.0%        2  0.005000  0.000000  0.010000
SQL*Net break/reset to client     0.000000    0.0%        2  0.000000  0.000000  0.000000
-----------------------------  ------------  ------  -------  --------  --------  --------
TOTAL (15)                    1,957.470000  100.0%  425,317  0.004602  0.000000  1.320000
```

> The bottleneck for this experience is `SQL*Net message from client`.

It is important to define *bottleneck* in terms of an individual task execution. Many people regard the *bottleneck* as an attribute of another entity. It is especially common for people to ask, for example, "What is the system's bottleneck?" The answer is that different task executions on a given system often have different bottlenecks, so to identify one of those experiences' bottlenecks as the *system*'s bottleneck is often confounding.

> ***Example:*** During the 1,957.47s payroll batch job experience profiled in the prior example, a system monitoring tool had reported that, while this job was running, the "system's bottleneck" was overwhelmingly the `latch free` resource. This diagnosis led an IT department to work for months toward reducing `latch free` calls on this system, with the top priority of improving this payroll job's performance. However, no amount of `latch free` reduction on this system was ever going to make an appreciable improvement to the response time of this payroll job. This is abundantly evident from the job's profile, where you can see that `latch free` calls have contributed only 1.2% of the job's total response time.

In this example, `latch free` might have been the aggregate bottleneck among all the task executions on the system, but this top-priority payroll job was experiencing a different bottleneck entirely. The very notion that the *bottleneck* is an attribute of a whole system helped to prevent this system's managers from seeing their real network configuration problem.

Amdahl's Law

Profiling helps you think clearly about performance. Even if Gene Amdahl hadn't done the work that he did back in 1967, you'd have probably come up with the gist of Amdahl's law yourself after the first few profiles you looked at. Amdahl's law implies that a task execution's performance improvement is proportional to how much that execution uses the thing you improved.

> **Performance improvement is proportional to how much a program uses the thing you improved.**

For example, if the thing you're trying to improve contributes only 5% to your task's total response time, then the maximum impact you'll be able to make is 5% of your total response time. This means that the closer to the top of a profile that you work (assuming that the profile is sorted in descending response time order), the bigger the benefit potential you'll have for your overall response time.

This doesn't mean that you always work a profile in top-down order, though, because you also need to consider the costs of the remedies you're considering.[8]

> ***Example:*** In the following profile, you can see, for each row in the profile, how much time you think you can save by implementing the best remedy and how much you think each remedy will cost to implement.

	Response time (s)		Expected potential response time improvement %	Expected relative difficulty (cost) of remedy
1	1,748.229	70.8%	34.5%	1,000,000
2	338.470	13.7%	12.3%	1
3	152.654	6.2%		+∞
4	97.855	4.0%	4.0%	10
5	58.147	2.4%	0.1%	1,000
6	48.274	2.0%	1.6%	1
7	23.481	1.0%		+∞
8	0.890	0.0%		+∞
Total	2,468.000	100.0%	52.5%	

This profile shows the potential for improvement and the corresponding cost of improvement for each line item.

> What remedy action would you implement first? Amdahl's law says that implementing the repair that will reduce response time by 34.5% (of 2,468 s) has the greatest potential benefit. But if that remedy is truly 1,000,000 times more expensive than the remedy reducing response time by 12.3%, then focusing on the second remedy may yield better net benefit.
>
> The key to your response is to have both parts of the two-part problem statement that I describe in "Problem Diagnosis" on page 9: you need to know your current state, *and* you need to know the goal state. The current state here is easy: the response time is 2,468.000 s

[8] Millsap, "Diagnosing before Resolving."

for the experience we're looking at. But the goal state? I've given you no clue. Here's why it matters...

Case 1: Imagine that your biggest customer will fire you—causing your entire business to end forever—if you don't make this 2,468 s task execute in at least 15% less time. What do you do? It's easy: you attack the "dirt cheap" remedies, which will yield a (12.3 + 4.0 + 1.6)% = 17.9% response time improvement. That's enough to satisfy your customer, and you'll do it without spending much.

Case 2: Imagine that your biggest customer will fire you—causing your entire business to end forever—if you don't make this 2,468 s task execute in at least 35% less time. What do you do? You should assign the easy work on line 2 to an assistant, and then, in parallel—*immediately*—get cracking on the expensive remedy action on line 1 of the profile. You cannot reach the 35%-better goal without accomplishing at least part of the highest-cost (1,000,000) improvement.

Which remedy action should you implement first? The deciding factor is your goal state. To make wise investment decisions, you must know what you need, and you must know how much you're willing to pay for it.

A tremendous value of the profile is that you can learn exactly how much improvement you should expect for a proposed investment. It opens the door to making better decisions about what remedies to implement first. Your predictions give you a yardstick for measuring your own performance as an analyst. And finally, it gives you a chance to showcase your cleverness and intimacy with your technology as you find more efficient remedies for better-than-expected response time reductions, at lower-than-expected costs.

What remedy action you implement first really boils down to how much you trust your cost estimates. Do your cost estimates really take into account the risks that the proposed improvement may inflict upon the system? For example, it may seem inexpensive to change that parameter or drop that index, but does that change potentially disrupt the good performance behavior of something out there that you're not even considering right now? Reliable cost estimation is another area in which your technological skills pay off.

Another factor worth considering is the political capital that you can earn by creating small victories. Maybe cheap, low-risk improvements won't amount to much overall response time improvement, but there's value in establishing a track record of small improvements that exactly fulfill your predictions about how much response time you'll save for the slow task. A track record of prediction and fulfillment ultimately—especially in the area of software performance, where myth and superstition have reigned at many locations for decades—gives you the credibility you need to influence your colleagues (your peers, your managers, your customers, ...) to let you perform increasingly expensive remedies that may produce bigger payoffs for the business.

A word of caution, however: don't get careless as you propose bigger, costlier, riskier remedies. Credibility is fragile. It takes a lot of work to build it up but only one careless mistake to bring it down.

Skew

When you work with profiles, you will encounter problems like this one:

> ***Example:*** Here is the profile you saw earlier on page 13:

```
CALL-NAME                                          DURATION       %       CALLS      MEAN       MIN        MAX
-------------------------------------------------  -------------  ------  ---------  --------  --------  ---------
db file sequential read                           59,081.406102   76.6%  10,013,394  0.005900  0.000010  15.853019
log buffer space                                   6,308.758563    8.2%       9,476  0.665762  0.000004   1.010092
free buffer waits                                  4,688.730190    6.1%     200,198  0.023420  0.000004   1.021281
EXEC                                               4,214.190000    5.5%      36,987  0.113937  0.000000   5.400000
log file switch completion                         1,552.471890    2.0%       1,853  0.837815  0.000006   1.013093
db file parallel read                                464.976815    0.6%       7,641  0.060853  0.000030   8.964706
log file switch (checkpoint incomplete)              316.968886    0.4%         351  0.903045  0.000003   1.014777
rdbms ipc reply                                      244.937910    0.3%       2,737  0.089491  0.000001   2.010042
undo segment extension                               140.267429    0.2%       1,411  0.099410  0.000001   0.108950
log file switch (private strand flush incomplete)    112.680587    0.1%         134  0.840900  0.002781   1.000239
17 others                                             23.367228    0.0%      58,126  0.000402  0.000000   5.045291
-------------------------------------------------  -------------  ------  ---------  --------  --------  ---------
TOTAL (27)                                        77,148.755600  100.0%  10,332,308  0.007467  0.000000  15.853019
```

> This profile shows that 10,013,394 read calls had consumed 59,081 s of response time. How much response time would you eliminate if you could eliminate half of those `db file sequential read` calls? If you eliminate half of those calls, then won't you eliminate half of the time consumed by those calls?

The answer is, *not necessarily.* The response time you'll save is not always proportional to the number of calls you eliminate. Consider this simpler example for a moment:

> ***Example:*** Four calls to a subroutine consumed four seconds. How much unwanted response time would you eliminate if you could eliminate half of those calls?
>
> The answer depends upon the response times of the individual calls that you could eliminate. You might have assumed that each of the call durations was the average (4 s)/(4 calls) = 1 s/call. But nowhere in the problem statement did I tell you that the call durations were uniform.
>
> Imagine the following two possibilities, where each list represents the latencies of the four subroutine calls:
>
> A = {1.0, 1.0, 1.0, 1.0}
> B = {3.7, 0.1, 0.1, 0.1}
>
> In list A, the latencies are uniform, so no matter which half (two) of the calls you eliminate, you'll reduce total response time to 2 s. However, in list B, it makes a tremendous difference which two calls you eliminate. If you eliminate the first two calls, then the total response time will drop to 0.2 s (a 95% reduction). However, if you eliminate the final two calls, then the total response time will drop to 3.8 s (only a 5% reduction), which is probably not enough of an improvement for a user to notice.

Skew is a non-uniformity in a list of values. The possibility of skew is what prohibits you from providing a precise answer to the question about eliminating half of your `db file sequential read` calls. Let's look again:

> ***Example:*** How much time will you conserve if you eliminate half of the 10,013,394 `db file sequential read` calls?

```
CALL-NAME                                          DURATION       %       CALLS      MEAN       MIN        MAX
-------------------------------------------------  -------------  ------  ---------  --------  --------  ---------
db file sequential read                           59,081.406102   76.6%  10,013,394  0.005900  0.000010  15.853019
log buffer space                                   6,308.758563    8.2%       9,476  0.665762  0.000004   1.010092
free buffer waits                                  4,688.730190    6.1%     200,198  0.023420  0.000004   1.021281
EXEC                                               4,214.190000    5.5%      36,987  0.113937  0.000000   5.400000
log file switch completion                         1,552.471890    2.0%       1,853  0.837815  0.000006   1.013093
db file parallel read                                464.976815    0.6%       7,641  0.060853  0.000030   8.964706
log file switch (checkpoint incomplete)              316.968886    0.4%         351  0.903045  0.000003   1.014777
rdbms ipc reply                                      244.937910    0.3%       2,737  0.089491  0.000001   2.010042
undo segment extension                               140.267429    0.2%       1,411  0.099410  0.000001   0.108950
log file switch (private strand flush incomplete)    112.680587    0.1%         134  0.840900  0.002781   1.000239
17 others                                             23.367228    0.0%      58,126  0.000402  0.000000   5.045291
-------------------------------------------------  -------------  ------  ---------  --------  --------  ---------
TOTAL (27)                                        77,148.755600  100.0%  10,332,308  0.007467  0.000000  15.853019
```

Without knowing anything about skew, the only answer you can provide is, "Somewhere between 50.066 970 s and 59,031.339 132 s."[9] That is the most precise correct answer you can return. Imagine, however, that you had some additional information that shows a histogram of the call durations, like this:

```
RANGE {min ≤ e < max}              DURATION       %       CALLS       MEAN        MIN        MAX
----------------------------  -------------  ------  ----------  ---------  ---------  ---------
 1.     0.000000     0.000001
 2.     0.000001     0.000010
 3.     0.000010     0.000100     199.445978    0.3%   9,346,059   0.000021   0.000010   0.000099
 4.     0.000100     0.001000      21.420428    0.0%     108,351   0.000198   0.000100   0.000999
 5.     0.001000     0.010000     612.513248    1.0%     106,319   0.005761   0.001000   0.009999
 6.     0.010000     0.100000  11,193.505611   18.9%     314,869   0.035550   0.010000   0.099999
 7.     0.100000     1.000000  26,057.804096   44.1%     130,471   0.199721   0.100002   0.999717
 8.     1.000000    10.000000  20,804.497660   35.2%       7,308   2.846811   1.000184   9.900656
 9.    10.000000   100.000000     192.219083    0.3%          17  11.307005  10.242772  15.853019
10.   100.000000 1,000.000000
11. 1,000.000000           +∞
----------------------------  -------------  ------  ----------  ---------  ---------  ---------
TOTAL (11)                    59,081.406102  100.0%  10,013,394   0.005900   0.000010  15.853019
```

Then you could formulate much more precise best-case and worst-case estimates. Specifically, if you had information like this, then you'd know how important it is to target the highest-latency calls and leave the low-latency ones alone. Eliminating the wrong 93% of the calls will eliminate only 0.3% of your response time, but eliminating the right 5% of your calls will eliminate 98.5% of your response time.

The ability to find skew gives you the precious ability to *target* your efforts to maximize your efficiency as an optimizer.

Minimizing Risk

A couple of sections back, I mentioned the risk that repairing the performance of one task can damage the performance of another. It reminds me of something that happened to me once in Denmark. It's a quick story:

Scene: A kitchen table in Måløv, Denmark. A dozen people from all over the world sat around the table, working on their laptop computers and conducting various conversations.

Cary: Guys, I'm burning up. Would you mind if I opened the window for a little bit to let some cold air in?

Friend: Why don't you just take off your [redacted] sweater?

The End.

It was at this point that I realized, I was wearing a new sweater that I had bought shortly beforehand. It was a quality multi-layer garment made of Gore-Tex®, Thinsulate®, and Icelandic lambswool, with a zippered collar that went up to my ears. Very warm indeed. Exceptionally well suited for the out-of-doors. Where I was not.

There's a general principle at work here that people who optimize know. When everyone is happy except for you, make sure you're locally efficient before you go messing around with the global stuff that affects everyone else, too. In other words:

> **Match the scope of your solution to the scope of your problem.**

[9] The lower bound is half of the call count times the minimum measured latency: (10,013,394 calls / 2) × 0.000 010 sec/call) = 50.066 970 s. The upper bound is the total call duration minus the lower bound: 59,081.406 102 s − 50.066 970 s = 59,031.339 132 s.

This principle is why I flinch whenever someone proposes to change a system's SQL*Net packet size, when the problem is really a couple of badly-written Java programs that make too many database calls (and hence too many network I/O calls). If everybody is getting along fine except for the user of one or two programs, then the safest solution to the problem is a change whose scope is localized to just those one or two programs.

Efficiency

One of the most difficult problems that a performance analyst can face is the dreaded "everything is slow" dilemma. If everyone on the entire system is suffering, then what should you focus on first? The task that is most constricting the business is the one you should focus on first.[10] The way to begin is to ensure that the program is working as efficiently as it can. **Efficiency** is the inverse of how much of a task execution's total service time can be eliminated without adding capacity, and without sacrificing required business function.

In other words, efficiency is an inverse measure of waste. Here are some examples of waste that commonly occur in the database application world:

> ***Example:*** A middle tier program creates a distinct SQL statement for every row it inserts into the database. It executes 10,000 database prepare calls when it could have accomplished the job with one prepare call.
>
> ***Example:*** A middle tier program makes 100 database fetch calls (and thus 100 network I/O calls) to fetch 994 rows. It could have fetched 994 rows in 10 fetch calls (and thus 90 fewer network I/O calls).
>
> ***Example:*** A SQL statement[11] touches the database buffer cache 7,428,322 times to return a 698-row result set. An extra filter predicate could have returned the 7 rows that the user really wanted to see, with only 52 touches upon the database buffer cache.

Certainly, if a system has some global problem that creates inefficiency for broad groups of tasks across the system (e.g., missing index, badly set parameter, poorly configured hardware), then you should fix it. But don't tune a system to accommodate programs that are inefficient.[12] There is a lot more leverage in curing the program inefficiencies themselves. Even if the programs are commercial, off-the-shelf applications, it will benefit you better in the long run to work with your software vendor to make your programs efficient, than it will to try to optimize your system to be as efficient as it can with inherently inefficient workload.

Improvements that make your program more efficient can produce tremendous benefits for everyone on the system. It's easy to see how top-line reduction of waste helps the response time of the task being repaired. What many people don't understand as well is that making one program more efficient creates a side-effect of performance improvement for other programs on the system that have no apparent relation to the program being repaired. It happens because of the influence of *load* upon the system.

---

[10] Goldratt, *The Goal*.

[11] My choice of article adjective here is a dead giveaway that I was introduced to SQL through the Oracle community.

[12] Admittedly, sometimes you need a tourniquet to keep from bleeding to death. But don't use a stopgap measure as a permanent solution. Address the inefficiency.

Load

Load is competition for a resource induced by concurrent task executions. Load is the reason that the performance testing done by software developers doesn't catch all the performance problems that show up later in production.

One measure of load is *utilization*. **Utilization** is resource usage divided by resource capacity for a specified time interval. As utilization for a resource goes up, so does the response time a user will experience when requesting service from that resource. Anyone who has ridden in an automobile in a big city during rush hour has experienced this phenomenon. When the traffic is heavily congested, it takes longer to get where you're going. There are two reasons that systems get slower as load increases: queueing delay, and coherency delay.

Deterministic and Random Arrivals

The process through which requests arrive into a system influences how that system will perform. Such an arrival process is said to be **deterministic** if and only if the timing of its arrivals can be predicted with certainty. An arrival process that cannot be predicted with certainty is said to be **random**.

> ***Example:*** Several years' worth of data suggest that the arrival rate of customers to your favorite restaurant at lunchtime is 542 customers per hour. Although you have confidence in predicting how many customers will arrive at lunchtime tomorrow, you do not know exactly when each customer will arrive; thus, the arrival process into the restaurant is *random*.

> ***Example:*** A busy superhighway has a traffic signal on its on-ramp that allows exactly one car to enter the highway every 5 s. During peak times, drivers queue behind the traffic signal in an unpredictable way, but the rate at which cars are allowed onto the highway is completely predictable; thus, the arrival process onto the highway is *deterministic*.

Determinism makes systems easier to manage. If you can control exactly when your next service request is coming, then you can plan your activity so that none of your capacity goes wasted and yet your system will never get hit with more workload than it can handle. However, on a system with *random* arrivals, the arrivals tend to clump up, which makes managing your capacity much more difficult. There will be periods when no requests enter the system, and then there will be periods when dozens of requests all arrive, inconveniently, at the same time. During the lulls, capacity will go unused—wasted—but during the spikes, users will have to wait for resources that are busy serving other requests.

Queueing Delay

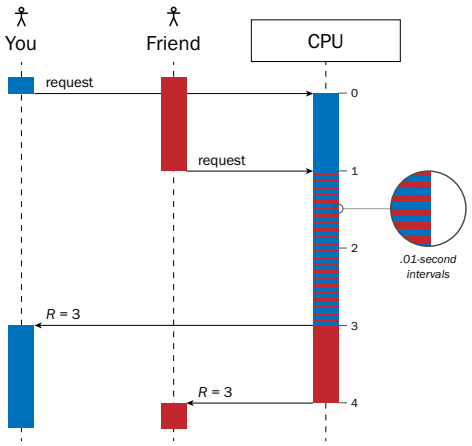
When you are on an unloaded system, where there's nobody else to wait on, each response time will consist exclusively of *service time*. **Service time** is the duration that a task execution spends consuming a given resource, measured in time per execution (e.g., seconds per click). Service times usually fluctuate. You can describe service times using averages and percentiles ("Percentile Specifications" on page 7).

> ***Example:*** It usually takes 3 min to cook a hamburger, but sometimes it takes 2.7 min, sometimes 3.4 min. If the mean time it takes to cook a hamburger is 3 min, then perhaps the 95th percentile cook time is 3.71 min. You can find out by measuring and recording all your hamburger cook times.

When you are on an unloaded system, your response time is all service time. However, as others join you on the system, your response times will begin to include not just service times, but also the time you'll spend waiting in line behind other requests. **Queueing delay** is the duration that a request spends enqueued at a given resource, awaiting its opportunity to consume that resource.

Like service time, queueing delay is measured in time per task execution. Queueing delay varies with load.

> ***Example:*** Imagine that you and one friend are the only users on a typical computer with one single-core CPU. Each of you will execute a request requiring 2 s of CPU time. You make your request first, and she makes hers 1 s later, while yours is running. For the first second, you'll get the CPU's full attention, but once you friend's request arrives, the operating system will begin time-slicing CPU time between the two of you, probably in 0.01 s increments (most schedulers use a CPU quantum of 1/100th of a second). While you are using the CPU for 0.01 s, your friend will wait in the CPU run queue. While she uses the CPU for the next 0.01 s, you'll wait.
>
> The scheduler will switch the two of you back and forth until your request is finished, and then, with your request out of the way, she'll get the CPU's full attention for her final second of processing. (The operating system scheduler code would continue to run every 0.01 s, but since there would be nobody in the run queue, your friend's process would never get preempted.) After all this, your response time would be 3 s: that's 2 s of service time, and 1 s of queueing delay, as shown in the figure below (the blue thread on the sequence diagram). And so would hers (the red thread).



Two people competing for CPU capacity on a preemptive multitasking system. Waiting each 1/100th of a second while your friend gets her time slice degrades your response time.

While service time fluctuations for a given unit of work are usually small, load-induced fluctuations in queueing delay can be huge, dwarfing service times by an order of magnitude or more. It is thus important to know how much you can expect queueing delays to change as load changes. This is where mathematical models enter the conversation.

Queueing Theory

In the early 1900s, the Danish mathematician Agner Erlang developed models to determine how many circuits the Copenhagen Telephone Company would need to provide acceptable telephone service.[13] Erlang's "M/M/m" model (also called "M/M/c") also helps predict how modern com-

[13] Wikipedia, "Erlang."

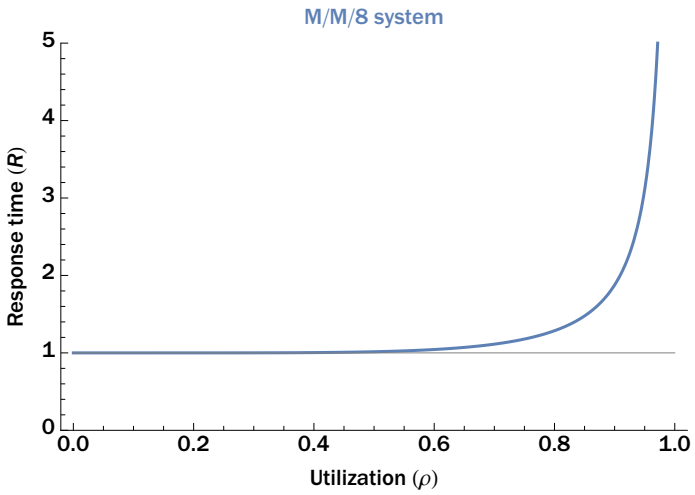
puting systems will perform. Most systems you use won't be exactly M/M/m, but nevertheless, most systems behave in a manner that the model helps to explain.

The lowercase m in "M/M/m" is the number of *service channels* in the system.[14] A **service channel** is a resource that shares a single queue with other such resources, like a clerk at a multi-clerk airport ticket counter or a CPU in a symmetric multiprocessor computer. M/M/m models only those systems whose service channels are homogeneous, parallel, and independent. Specifically, the service channel independence assumption omits *coherency delays* from the model. M/M/m represents response time (R) as consisting of just the two components, *service time* (S) and *queueing delay* (Q):

$$R = S + Q.$$

For most real-life Oracle-based applications, M/M/m will not be able to give you high-precision response time predictions. But it will be able to help you understand—and explain—why your performance behaves as it does as your loads change.

> ***Example:*** The following graph explains how it feels to use an 8-channel M/M/m system under different load conditions (think of an 8-lane highway or an 8-CPU computer). At low utilization values, your response times are so nice that it feels like nobody else is on the system. As load ramps up, you sense a slight, gradual degradation in response time. That gradual degradation doesn't hurt much, but as load continues to ramp up, response time begins to degrade in a manner that is neither slight nor gradual. Ultimately, the degradation becomes quite unpleasant and, in fact, hyperbolic.



This curve shows response time as a function of utilization for an M/M/m system with 8 service channels.

Coherency Delay

It's a reasonably safe bet that no matter what computer application system you're thinking of right now, it does *not* meet the M/M/m "service channels are independent" assumption. *Coherency delay* is a measure of the service channel interdependencies. **Coherency delay** is the

[14] Wikipedia, "Kendall Notation"; or Millsap & Holt, *Optimizing Oracle Performance*, 243.

duration that a request spends communicating and coordinating access to a shared resource. Like response time, service time, and queueing delay, coherency delay is measured in time per task execution.

Neil Gunther's *Universal Scalability Law* (USL) is a mathematical model for predicting coherency delay.[15] You'll see forms of coherency delay in your profiles. In Oracle, timed events like the following are examples of coherency delay:

```
enqueue
log file sync
buffer busy waits
latch free
```

You can't model coherency delays like these with M/M/m, because M/M/m assumes that all m of your service channels are parallel, homogeneous, and independent. The model assumes that after your request waits politely in a FIFO queue for long enough that all the requests that arrived ahead of it have exited the queue for service, it'll be your turn to be serviced. However, coherency delays don't work like that.

> ***Example:*** Imagine an HTML data entry form in which one button labeled "Update" executes a SQL `update` statement, and another button labeled `Save` executes a SQL `commit` statement. An application built like this would almost guarantee abysmal performance. That's because the design makes it possible—quite likely, actually—for a user to click Update, look at his calendar, realize uh-oh he's late for lunch, and then go to lunch for two hours before clicking `Save` later that afternoon.
>
> The impact to other tasks on this system that wanted to update the same row would be devastating. Each task would necessarily wait for a lock on the row (or, on some systems, worse: a lock on the row's page) until the locking user decided to go ahead and click `Save`. ...Or until a database administrator killed the user's session, which of course would have unsavory side effects to the person who had thought he had updated a row.

In the example, the amount of time a task would wait on the lock to be released has nothing to do with how busy the system is. It would be dependent upon random factors that exist outside of the system's various resource utilizations. That's why you can't model this kind of thing in M/M/m. It is also why you can never assume that a performance test executed in a unit testing type of environment is sufficient for a making a go/no-go decision about insertion of new code into a production system.

Managing Load

The goal of having a big system is to get lots of work done. But, if you crank the load up too much, the response time degradation will make everyone on the system unhappy. So, then, how much load is too much? At what utilization value is the optimal load? This question is relevant for every resource on your system that is affected by load. When you say "utilization," most people first think of CPU, but, for example, disk and network I/O performance also suffers under loads that are too strenuous, so you need to understand the optimal loads for these resources as well.

On resources with deterministic arrivals, your optimal resource utilization is 100%. That is, if you can *completely* plan your load, then you can fill every cycle of capacity that you have. ...But don't overfill, or of course you'll create queueing problems. On resources with random arrivals, it's more difficult. You would love to run at 100% utilization, but you need to leave some headroom for spikes in utilization caused by the randomness in your request arrivals. So another way to ask the optimal load question is, "How much headroom should I leave?" If you leave too much, then you're wasting expensive capacity, but if you leave too little, then performance will suffer. So, what's the right amount?

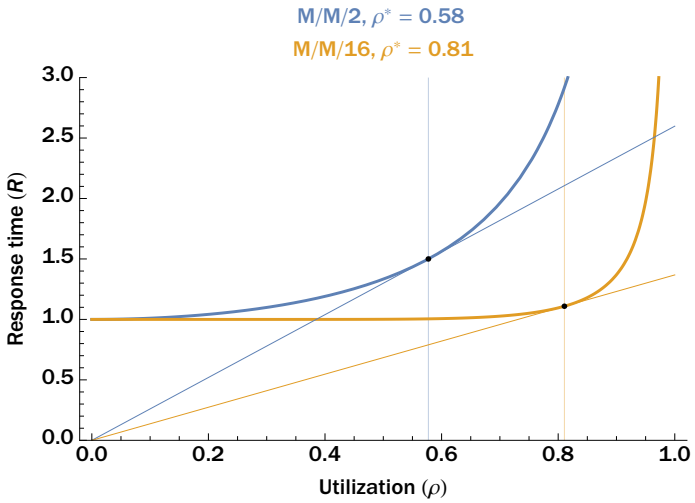
[15] Gunther, "Universal Law."

In the past, I have written that the values in the following table are good starting points for defining an operational utilization ceiling for a given resource:

Service channel count (m)	M/M/m knee utilization
1	50%
2	58%
4	67%
8	74%
16	81%
32	86%
64	89%
128	92%

M/M/m knee values for common values of m.

These values represent the **knee** in an M/M/m queueing system defined as the utilization value at which response time divided by throughput is at its minimum.[16] The knee defined this way is a starting point for defining an operational utilization ceiling for each resource on your system. At utilizations greater than the M/M/m knee utilization, response times are more prone to unbearable response time fluctuations stimulated by small peaks in utilization.



Operating at utilizations less than the knee utilizations (ρ^*) on these two M/M/m systems would produce response times that are more stable than if you were to operate at greater utilizations.

However, the M/M/m knee utilization is not necessarily your best utilization ceiling:

- The M/M/m knee utilization value might be too big. Your system probably isn't a perfect M/M/m system. It probably has scalability defects like *coherency delays*, which the model

[16] Vernon, "CS 547."

doesn't take into account. If these imperfections cause your system to perform more poorly than the model predicts, then you'll want a utilization ceiling that is *smaller* than the knee value.

- The M/M/m knee utilization value might be too small. Perhaps your users' response time tolerances are forgiving enough that you could run your system above your M/M/m knee utilization value without anyone perceiving a performance compromise. In this case, you might want a utilization ceiling that is *larger* than the knee value.

  Example: A system owner who is willing to tolerate mean response times of $R \leq 1$s per execution of a given task should be able to run his system at higher sustained utilizations than the owner of an identical system who is willing to tolerate mean response times of no more than $R \leq 0.1$s per execution of the same task.

Any useful utilization ceiling must be derived from inquiries about response time requirements, not solely from mathematical observations about your system.[17]

Keeping your utilization below some ceiling is not a goal in and of itself, it's something you do because you want to meet the real goal of providing consistent performance experiences to your system's users. Those experiences are the most important things to measure ("Percentile Specifications" on page 7). To that end, you should define your operational utilization ceiling for a given resource (CPU, disk drive, network interface, etc.) as follows:

1. Use the resource's knee value in the table on page 24 as a starting point. Your best operational utilization ceiling value may be larger or smaller than this number.
2. If your response times fail to meet your requirements, then adjust your utilization ceiling downward.[18]
3. If your response times are consistently better than your requirements, then you may adjust your utilization ceiling upward.

   Example: Imagine that your system has a single quad-core CPU, and that your business requires 95% of task executions to respond faster than your users' tolerances for the tasks they run on the system. A single quad-core CPU system is not exactly M/M/4, but you can begin with a utilization ceiling value of 67% (choosing the $m = 4$ entry from the knee utilization table). If more than 95% of your response time experiences are better than your users' tolerances, then you may adjust your ceiling upward from 67%.

   If fewer than 95% of your users' response time experiences meet their tolerances, then you must adjust your ceiling downward from 67%. If running your system at ever lower utilizations (by removing or deferring load) doesn't improve your response times, then your problem is not load induced. You'll need either to improve the efficiency of the tasks that are not performing satisfactorily or upgrade your hardware.

It's one thing to claim that you have a utilization ceiling value. It's another thing entirely to enforce it.

Example: It's easy to say that your utilization ceiling is 80%, but it's much more difficult to refuse to run a new financial report during a busy period when your CPU utilization is standing at 79.8%. The owner of the system may point at the spare 20.2% of capacity and demand that you run the report. If you have a record of historical response times (such as suggested by the schema shown on page 6) and the utilization values that correspond to those response times, you'll be able to make a much better informed decision about it.

It is okay to measure utilization in addition to response times, but measuring utilization *instead of* response times will lead you to mistakes. If you care about response times, then you need to be measuring response times.

---

[17] Gunther, "Knees and Queues."

[18] How? By renegotiating your requirements, by some combination of rearranging your job schedules, optimizing your business processes, optimizing your software, refusing to admit load onto your system that exceeds your system's ability to meet your required service levels, or upgrading your hardware.

Capacity Planning

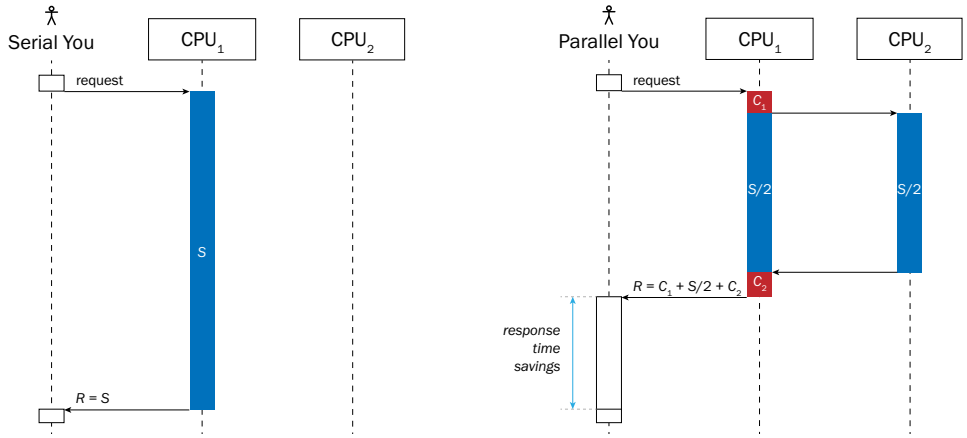
Using wisely chosen utilization ceilings can simplify your capacity planning process. It works like this:

1. If you keep your utilizations less than your ceilings, your system will behave roughly linearly: no big hyperbolic surprises.
2. Your goal capacity for a given resource is thus the amount at which you can comfortably run your tasks at peak times without driving utilization beyond your ceiling.
3. However, if you're letting your system run any of its resources beyond their ceiling utilizations, then you have performance problems (whether you're aware of them or not).
4. If you have performance problems, then you don't need to be spending your time with mathematical models; you need to be spending your time fixing those problems by either rescheduling load, eliminating load, or increasing capacity.

That's how capacity planning fits into the performance management process.

Parallelism

What can you do when you can't make a task run fast enough? One option is to *parallelize* it. The Oracle Database, for example, makes it relatively easy to convert a long-running process into a whole fleet of processes that all work on different pieces of a problem at the same time. Parallelism works like this:



The system on the left shows serial execution of a task; on the right, a 2-way parallel execution of the same task. In the parallel execution, there is some extra coordination overhead shown in red. Here, there are the same number of blue pixels in both cases; this is the best-case scenario.

The point of parallelizing a task is to reduce that task's response time by exploiting unused capacity. It creates catastrophic performance problems if there is no unused capacity to parallelize into.

Example: A task, executed serially, has been measured to run in 30 min on an unloaded computer with two quad-core CPUs. Using 8 parallel processes, it runs in 7 min on an unloaded computer. Running the task in parallel will save 23 min of response time when there's no other load on the system.

However, if the system is already busy running other load, then adding the parallelized task—which, on Oracle, can add 8 × 2 + 1 = 17 or more[19] new processes—may overwhelm the system for the duration of the task's execution, which, because of queueing delays induced by the overwhelming load, may be far more than the 7 min you might have expected.

Parallelism helps only if you have spare capacity available to parallelize into.

When you don't have any spare capacity available to parallelize into, then using parallel execution features is just an easy way to add load to your system, which, as you've seen, can quickly lead to catastrophic (literally hyperbolic) performance consequences.

Parallel operations can reduce response time, but they don't reduce load. Parallelization in fact puts *more* load on a system than a corresponding serial execution. There are two reasons for this:

- There is extra effort required to partition the workload among the parallel processes and then harvest the work done by those processes, and
- The code path executed by parallel processes is often less efficient than the code path executed by a serial process.

  Example: A parallel execution on an Oracle system typically uses a different, higher load execution plan than a serial execution of the same SQL statement.

Parallelism *redistributes* work horizontally in the sequence diagram, across more resources. Parallelization thus increases concurrency on a system while the parallel work is executing. This directly increases the *load* on a system, by cramming more total work into less total time. Parallelism can be a fine idea for reducing response times on big systems with few users. It tends to be a worse idea on busy systems. Even systems with small user counts get busy really quickly if each user is allowed to parallelize his work.

Performance Testing

Understanding the impact of call counts (which, collectively, make up the *load* on your system)—how they lead to queueing delays and coherency delays—brings us to a difficult question. How can you be confident that unforeseen performance problems won't wreck your production implementation? You can model. You can test. However, nothing you do will assure you that you'll avoid every performance problem. It is extremely difficult to create models and tests in which you'll foresee *all* your production problems in advance of actually encountering those problems in production.[20]

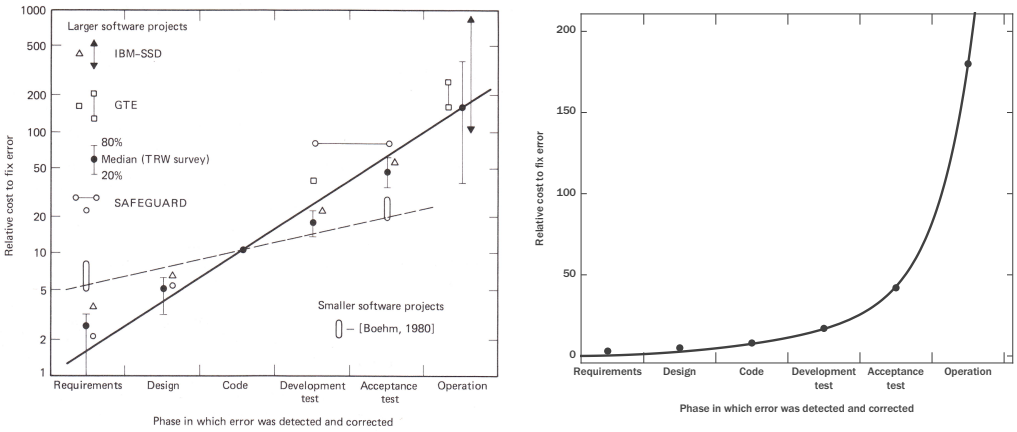
[19] Baskan, "Multiple Parallelizers."

[20] Computer Measurement Group is a network of professionals who study these problems.

Some people allow the apparent futility of this observation to justify not testing at all. Don't get trapped in that mentality. The following points are certain:

- You'll catch a lot more problems if you try to catch them prior to production than if you don't try at all. As Boehm has argued, it is absolutely worth considerable effort to catch problems early in the software development life cycle.[21]
- You'll never catch all your problems in pre-production testing. That's why you need a reliable and efficient method for solving the problems that leak through your pre-production testing processes.

Somewhere in the middle between "no testing" and "complete production emulation" is the right amount of testing. The right amount of testing for aircraft manufacturers is probably more than the right amount of testing for companies that sell baseball caps. But don't skip performance testing altogether. At the very least, your performance test plan will make you a more competent diagnostician (and clearer thinker) when it comes time to fix the performance problems that will inevitably occur during production operation.



Boehm showed that the cost to fix software errors increases exponentially throughout the software life cycle. The plot at right shows Boehm's "Median (TRW survey)" data on a linear vertical scale, which emphasizes the data's exponential nature.

Measuring

People feel throughput, and they feel response time. Throughput is often easy to measure, because all you have to do is count results over a specific time interval. Measuring response time is usually more difficult. Of course, it's easy enough to measure a user experience with a stopwatch, but it considerably more difficult to obtain and persist a precise account of the details—the call counts and the latencies—of why a given response time is as large as it is.

Unfortunately, people tend to measure what's easy to measure, which is not necessarily what they *should* be measuring. It's a bug—a downside of human adaptability. When it's not easy to measure what we need to measure, we tend to turn our attention to measurements that are easy

[21] Boehm, *Software Engineering Economics*, 40.

to get. When you measure something that's not really what you want, but it seems like it might be useful enough, and it's easier to obtain, you're using what's called a **surrogate measure**.

A crime scene investigator would prefer to see a video recording of the homicide, but the video is not available, so he measures the location, the pose, and the body temperature of the victim instead. A database performance analyst would love to see a detailed account of how much time each line of code has consumed, but the trace file is not available, so she measures call counts and utilization statistics instead.

Surrogate measures are a compromise inspired by better measures being difficult to obtain. I'm particularly impatient with surrogate measures in the business of computer software because, in the computer itself, we have at our disposal the most flexible tool ever created to help us solve the problem. To me, being forced to use surrogate measures for software performance is as ridiculous as using a backup camera that doesn't show when there's a child standing right behind your car.[22] It's a matter of good design—of caring enough to do a difficult job well.

Surrogate measures suck.

Suck is an ugly word, but it's a word fit for the task. Surrogate measures suck because they are unreliable—sometimes they work, sometimes they don't. It would actually be better if surrogate measures never worked, because then you would know never to trust them. The problem is that surrogate measures in fact *sometimes* work, and therein lies their evil. When surrogate measures do work, they inspire your confidence. And then they betray you.

This betrayal comes in two forms. First, surrogate measures can imply that something is a problem when it's not. That's what statisticians call *type I error*, the false positive.[23] Second, surrogate measures can imply that your system is okay when it's not. That's what statisticians call *type II error*, the false negative. I've seen each type of error waste years of people's time.

When it comes time to assess the specifics of a real system, your success is at the mercy of how good the measurements are that your system allows you to obtain. I've been fortunate to work in the Oracle market segment, where the software vendor at the center of our universe participates actively in making it possible to measure systems the right way. Getting application software developers to use the tools that Oracle offers is another story, but at least the capabilities are there in the product.

Performance Is a Feature

Performance is a software application feature, just like when my bug tracking system automatically converts strings like `case 1234` into hyperlinks.[24] Performance, like any other feature, doesn't just "happen"; it has to be designed and built. To do performance well, you have to think about it, study it, design it, write extra code for it, test it, and support it.

However, like many other features, you can't know exactly how performance is going to work out while it's still early in the project when you're writing, studying, designing, and creating the application. For many applications (arguably, for the vast majority), performance is *completely*

[22] Millsap, "Messed-Up App."

[23] Wikipedia, "Type I and Type II Errors."

[24] FogBugz, which is software that I enjoy using, does this.

unknown until the production phase of the software development life cycle. What this leaves you with is this:

> **Since you can't know how your application is going to perform in production, you need to write your application so that it's easy to fix performance in production.**

As David Garvin has taught us, it's much easier to manage something that's easy to measure.[25] Writing an application that's easy to fix in production begins with an application that's easy to measure in production.

Most times, when I mention the concept of production performance measurement, people drift into a state of worry about the measurement intrusion effect of performance instrumentation. They immediately enter a mode of data collection compromise, leaving only surrogate measures on the table. Won't software with extra code path to measure timings be slower than the same software without that extra code path?

I like an answer that Tom Kyte gave once in response to this question.[26] He estimated that the measurement intrusion effect of the Oracle Database's extensive performance instrumentation is *negative* 10% or less, where *or less* means *or better*, as in −20% or −30%. He went on to explain to a now-vexed questioner that the Oracle Database product is at least 10% faster now because of the knowledge that Oracle Corporation has gained from its performance instrumentation code, more than making up for any "overhead" the instrumentation might have caused.

I think that vendors tend to spend too much time worrying about how to make their measurement code path efficient without figuring out how first to make it effective. It lands squarely upon the idea that Donald Knuth wrote about in 1974 when he wrote that "premature optimization is the root of all evil."[27] The software designer who integrates performance measurement into his product is much more likely to create a fast application and—more importantly—an application that will become faster over time.

Three Axioms of Performance

At the beginning of this chapter, I described how Jimmy Harkey's axiomatic approach helped me master a complex subject: algebra. An axiomatic approach helps you in two important ways. It makes you better at working out complex problems, and it makes you better at explaining to other people why they should be motivated to participate in your proposed solution. I devoted the remainder of the chapter to discussing first principles about performance. I defined several terms and, I hope, helped you to understand the reasons that systems behave the way they do. I described ways to envision how a system spends your time. I showed you how to keep from being tricked by surprises beneath the data you're considering. I explained why systems behave the way they do as stresses on those systems increase.

Now it is time to tie these first principles together to create a repeatable and teachable method for optimizing the performance of your system. I will begin with three axioms of performance. I encourage you to consider these axioms carefully, just like Mr. Harkey considered the axioms of

[25] Garvin, "Learning Organization."

[26] Kyte, "Couple of Links..."

[27] Knuth, "Structured Programming," 268.

algebra with his high school students. It shouldn't take long to convince yourself of their reliability:

- **Axiom 1: To improve a system, you must measure what matters.** This is Eli Goldratt's central point in *The Goal*. Measuring what matters prevents many terrible mistakes. For example, if you measure what matters, you never fall prey to the mistaken idea that you can optimize a system by merely optimizing its individual components (its CPU, its memory, its disks, its network, etc.).
- **Axiom 2: You cannot guess where your system spends its time; you must measure it.** It's not just a few of us who are bad guessers, it's *everybody*; even the best programmers in history. It's Brian Kernighan, co-creator of the UNIX operating system and the C programming language, who "would have never guessed the hot spot of [his] program, but the profiler led him right to it."[28] It's Don Knuth, who wrote that "the universal experience of programmers who have been using measurement tools has been that their intuitive guesses [about which code paths are the most critical] fail."[29] It's all of us.
- **Axiom 3: The fastest call is no call.** Nils-Peter Nelson of Bell Labs wrote, "The fastest I/O is no I/O,"[30] but performance is not just about I/O, even on gigantic databases. Performance is about any code path that consumes *time*.

These axioms seem to be embedded in the DNA of great performance analysts. You'll see them echoed in the works of Don Knuth, Brian Kernighan, Jon Bentley, Tony Hoare, Edsger Dijkstra, Niklaus Wirth, Kirk McKusick, Michael Kay, Eli Goldratt, Dave Ensor, Anjo Kolk, Tom Kyte, Jonathan Lewis, and many others.

Method R

In 1999, Jeff Holt and I set out to create the optimal method for optimizing a system. We weren't just looking for a way to make systems faster; we were looking for the *fastest* way to make systems faster. We wanted a method with no wasted time, no wasted steps. We were looking not just to optimize systems, but to optimize the businesses that used these systems. Most of all, we wanted to optimize *ourselves*. We started with an analysis of the problems that people were having when they tried to improve the performance of their systems.

The first problem we found is that people were focusing on the wrong things. This is a violation of axiom 1. If performance is your problem, then utilization statistics, hit and miss ratios, subsystem performance metrics, ... These are the wrong things. Since a performance experience is always either a throughput or response time experience, you first need to express your performance problem as either a throughput problem, or a response time problem. In either case, your next step should be to measure how your program spends its time, axiom 2. A profile shows that.

A profile is like an invoice for response time. An invoice is easy to analyze. There are only two ways that a total price can be too high: (i) some quantity is too large, or (ii) some unit price is too high. Of course, the problem may be because of multiple quantities that are too large, or multiple prices that are too high, but quantity and price are fundamentally the only two problems you can have. Likewise, there are only two ways that a response time can be too big: (i) some call count is too large, or (ii) some call latency is too high. Axiom 3 reminds us that reducing call counts is the best way to reduce response time. It is in fact also the best way to reduce call latencies.

[28] Bentley, *More Programming Pearls*, 8.

[29] Knuth, "Structured Programming," 268.

[30] Bentley, *More Programming Pearls*, 62.

We called our algorithm *Method R*—the *R* stands for response time. Method R gets its relevance from Eli Goldratt, its profiler focus from Jon Bentley, and its Oracle measurement mechanics from Anjo Kolk. **Method R** is defined as follows:

```
while your system performance is not economically optimal
    identify the experience that you most need to improve
    profile the experience to measure its response time (R)
    optimize the experience
        if your program makes unnecessary calls
            eliminate your program's unnecessary calls
        if your calls are not executing at unconstrained hardware speed
            eliminate other programs' unnecessary calls
        else (your calls are executing at unconstrained hardware speed)
            upgrade your hardware
```

> Method R: an algorithm for optimizing a system based on response time and business priority.

In the years since Jeff and I first published this algorithm,[31] there have been a lot of interesting conversations about how to define Method R's terminating condition of economic optimality. In times of crisis, people usually interpret it to mean "safe from immediate danger." In rosier times, and especially during the development of a new application, a better definition is (i) your software makes no unnecessary calls, and (ii) calls run at hardware speed.[32]

When you are deciding which experience you most need to improve, remember the lesson from the example on page 15. You need to know your goal state so that you can factor in the anticipated cost of a proposed remedy action before you embark upon it.

Method R contains three terminal instructions: eliminate calls, eliminate calls, and upgrade. The one about upgrading is easy for your non-technical colleagues to understand.[33] Eliminating calls is more powerful and more subtle. Eliminating calls can take many forms.

Example: If the big daily 2 P.M. backup is slowing down your program, then eliminate calls from your program's time window by rescheduling the backup to 2 A.M.

Example: If that SAN is serving up 100 ms read calls because it is so busy hosting the disk I/O calls of 42 other systems, then eliminate calls by moving the data for some of those systems to another SAN.

Example: If your Oracle database buffer cache is so small that the block your program just read is already aged out of the cache, then eliminate unnecessary LIO calls that other programs are making while your program is running. If you can't, then eliminate some of your program's PIO calls by making your buffer cache bigger.

Example: If your query reads every block in a huge table to find and retrieve a handful of rows, then eliminate LIO calls by creating an index.

Example: If your software fetches only one row per network round trip, then eliminate database calls and network I/O calls by changing your code.

Example: If your application connects to Oracle and disconnects for every user experience, then eliminate connection management calls by changing your code.

---

[31] Millsap & Holt, *Optimizing Oracle Performance*, 20.

[32] Millsap, "Fast Enough."

[33] There is an interesting case in "Why the Upgrade Hurt Performance" on page 157 that you'll want to see before you recommend your next hardware upgrade.

Example: If your one part of your application exchanges millions of rows per day with another part of your application through a software service bus, one row at a time, then eliminate calls by changing your software architecture.

Eliminating Calls

Whether you are a system architect, a software developer, or a database administrator, …whether you're designing a new system or fixing an old one, *call count* is king. Eliminating calls is the secret to high-performance software. The reason? Because response time is not linear; it is proportional to the *squares* of your call counts.

Here's why. An experience's duration—its response time—is the sum of the durations of its profile lines. Each profile line's duration (d) is the product of that line's call count (c) and its mean call latency (μ):

$$d = c\mu$$

This simple formula makes it appear that d and c have a linear relationship; for example, it looks like doubling c would double d. But as you've seen in "Queueing Theory" on page 21, the real world doesn't behave that way. As c grows larger, d grows hyperbolically. Why? The secret is that μ (the mean latency) is itself influenced by c. As c grows larger, so does μ, because of queueing delays and coherency delays. Think of that four-lane highway where your speed slows down as the lanes become more and more crowded with competing vehicles. So what we really have is a formula where μ can be replaced by a function of c:

$$d = cf(c)$$

Seeing the formula expressed this way, you can better sense the influence of c upon d. You can see that d is proportional to c^2.

Everything I've said in this chapter—and everything I'll say in the remainder of this book—will point you toward this goal of eliminating calls. To eliminate calls, you must first know how to measure the calls that your system is making. But how? Particularly in Oracle? Here are the words that, in 2000, reset the focus of my whole career.

> **"A stethoscope is a simple tool that revolutionized the practice of medicine: it gave physicians an effective way to monitor the human body. A profile can do the same thing for your programs."**
>
> **—Jon Bentley, Bell Labs**