DOI:10.1145/1810891.1810909

Article development led by acmqueue
queue.acm.org

Improving the performance of complex software is difficult, but understanding some fundamental principles can make it easier.

BY CARY MILLSAP

Thinking Clearly About Performance, Part 1

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 high consulting rates. When circumstances thrust me into the "Oracle tuning" arena, I was quite unprepared.

Recently, I've been introduced to the world of "MySQL tuning," and the situation seems very similar to what I saw in Oracle more than 20 years ago.

It reminds me a lot of how difficult beginning algebra seemed when I was about 13 years old. At that age, I had to appeal heavily to trial and error to get through. I can remember looking at an equation such as $3x + 4 = 13$ and basically stumbling upon the answer, $x = 3$.

The trial-and-error method worked—albeit slowly and uncomfortably—for easy equations, but it didn't scale as the problems got tougher—for example, $3x + 4 = 14$. Now what? My problem was that I wasn't thinking clearly yet about algebra. My introduction at age 15 to teacher James R. Harkey put me on the road to solving that problem.

In high school Mr. Harkey taught us what he called an *axiomatic* approach to solving algebraic equations. He showed us a set of steps that worked every time (and he gave us plenty of homework to practice on). In addition, by executing those steps, we necessarily *documented* our thinking as we worked. Not only were we thinking clearly, using a reliable and repeatable sequence of steps, but we were also *proving* to anyone who read our

work that we were thinking clearly. Our work for Mr. Harkey is illustrated in Table 1.

This was Mr. Harkey's axiomatic approach to algebra, geometry, trigonometry, and calculus: one small, logical, provable, and auditable step at a time. It's the first time I ever really got 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 been expanding the scope of that goal beyond Oracle to: "Create an axiomatic approach to computer software performance optimization." I've found that not many people like it when I talk like that, so let's say it like this: "My goal is to help you think clearly about how to optimize the performance of your computer software."

What is Performance?

Googling the word *performance* results in more than a half-billion hits on concepts ranging from bicycle racing to the dreaded employee review process that many companies these days are learning to avoid. Most of the top hits relate to the subject of this article: the time it takes for computer software to perform whatever task you ask it to do.

And that's a great place to begin: the task, a business-oriented unit of work. Tasks can nest: "print invoices" is a task; "print one invoice"—a subtask—is also a task. For a computer user, *performance* usually means the time it takes for the system to execute some task. *Response time* is the execution duration of a task, measured in time per task, such as "seconds per click." For example, my Google search for the word *performance* had a response time of 0.24 seconds. The Google Web page rendered that measurement right in my browser. That is evidence to me that Google values my perception of Google performance.

Some people are interested in another performance measure: *throughput*, the count of task executions that complete within a specified time interval, such as "clicks per second." In general, people who are responsible for the performance of *groups* of people worry more about throughput than does the person who works in a solo contributor role. For example, an individual accountant is usually more concerned about whether the response time of a daily report will require that accountant to stay late after work. The manager of a group of accounts is additionally concerned about whether the system is capable of processing all the data that all of the accountants in that group will be processing.

Response Time versus Throughput

Throughput and response time have a generally reciprocal type of relationship, but not exactly. The real relationship is subtly complex.

Example 1. *Imagine that you have measured your throughput at 1,000 tasks per second for some benchmark. What, then, is your users' average response time? It's tempting to say that the average response time is 1/1,000 = .001 seconds per task, but it's not necessarily so.*

Imagine that the system processing this throughput had 1,000 parallel, independent, homogeneous service channels (that is, it's a system with 1,000 independent, equally competent service providers, each awaiting your request for service). In this case, it is possible that each request consumed exactly 1 second.

Now, you can know that average response time was somewhere between 0 and 1 second per task. You cannot derive response time exclusively *from a throughput measurement, however; you have to measure it separately (I carefully include the word exclusively in this statement, because there are mathematical models that can compute response time for a given throughput, but the models require more input than just throughput).*

The subtlety works in the other direction, too. You can certainly flip this example around and prove it. A scarier example, however, will be more fun.

Example 2. *Your client requires a new task that you're programming to deliver a throughput of 100 tasks per second on a single-CPU computer. Imagine that the new task you've written executes in just .001 seconds on the client's system. Will your program yield the throughput the client requires?*

It's tempting to say that if you can run the task once in just one thousandth of a second, then surely you'll be able to run that task at least 100 times in the span of a full second. And you're right, if the task requests are nicely serialized, for example, so that your program can process all 100 of the client's required task executions inside a loop, one after the other.

But what if the 100 tasks per second come at your system at random, from 100 different users logged into your client's single-CPU computer? Then the gruesome realities of CPU schedulers and serialized resources (such as Oracle latches and locks and writable access to buffers in memory) may restrict your throughput to quantities much less than the required 100 tasks per second. It might work; it might not. You cannot derive throughput exclusively from a response time measurement. You have to measure it separately.

Response time and throughput are not necessarily reciprocals. To know them both, you need to measure them both. Which is more important? For a given situation, you might answer legitimately in either direction. In

Table 1. The axiomatic approach as taught by Mr. Harkey.

3.1x + 4	= 13	problem statement
3.1x + 4 − 4	= 13 − 4	subtraction property of equality
3.1x	= 9	additive inverse property, simplification
3.1x / 3.1	= 9 / 3.1	division property of equality
x	≈ 2.903	multiplicative inverse property, simplification

many circumstances, the answer is that both are vital measurements requiring management. For example, a system owner may have a business requirement not only that response time must be 1.0 second or less for a given task in 99% or more of executions but also that the system must support a sustained throughput of 1,000 executions of the task within a 10-minute interval.

Percentile Specifications

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

Example 3. *Imagine that your response time tolerance is 1 second for some task that you execute on your computer every day. Imagine further that the lists of numbers shown in Table 2 represent the measured response times of 10 executions of that task. The average response time for each list is 1.000 second. Which one do you think you would like better?*

Although the two lists in Table 2 have the same average response time, the lists are quite different in character. In list A, 90% of response times were one second or less. In list B, only 60% of response times were one second or less. Stated in the opposite way, list B represents a set of user experiences of which 40% were dissatisfactory, 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 .987 seconds; in list B, it is 1.273 seconds. These statements about percentiles are more informative than merely saying that each list represents an average response time of 1.000 second.

As GE says, "Our customers feel the variance, not the mean."[1] Expressing response-time goals as percentiles makes for much more compelling requirement specifications that match with end-user expectations: for example, the "Track Shipment" task must complete in less than .5 seconds in at least 99.9% of executions.

Problem Diagnosis

In nearly every performance problem I've been invited to repair, the stated problem has been about response time: "It used to take less than a second to do *X*; now it sometimes takes 20+." Of course, a specific statement like that is often buried under veneers of other problems such as: "Our whole [adjectives deleted] system is so slow we can't use it."[2]

Just because something happened often for me doesn't mean it will happen for you. The most important thing to do first is state the problem clearly, so you can think about it clearly.

A good way to begin is to ask, what is the goal state that you want to achieve? Find some specifics that you can *measure* to express this: for example, "Response time of *X* is more than 20 seconds in many cases. We'll be happy when response time is one second or less in at least 95% of executions." That sounds good in theory, but what if your user doesn't have such a specific quantitative goal? This particular goal has two quantities (1 and 95); what if your user doesn't know either one of them? Worse yet, what if your user *does* have specific ideas, but those expectations are impossible to meet? How would you know what "possible" or "impossible" even is?

Let's work our way through those questions.

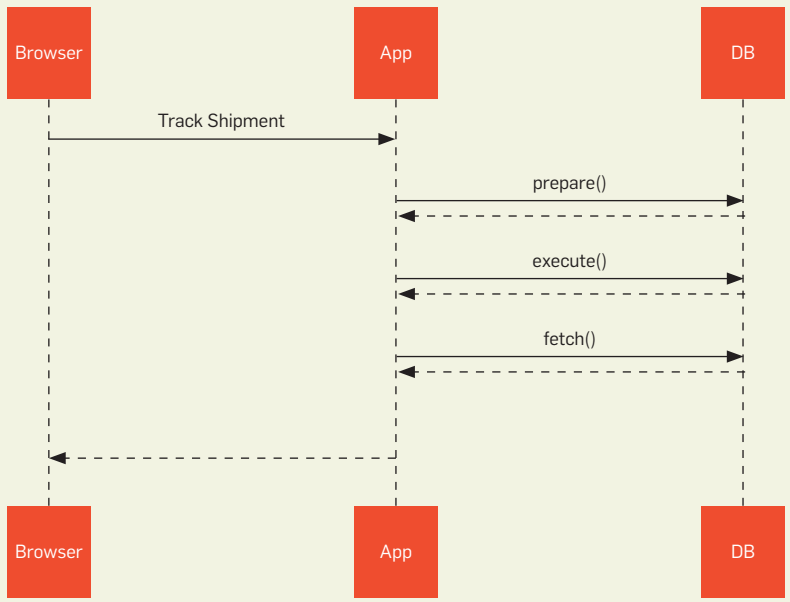
The Sequence Diagram

A *sequence diagram* is a type of graph specified in UML (Unified Modeling Language), 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. Figure 1 shows a standard UML sequence diagram for a simple application system composed

Table 2. The average response time for each of these two lists is 1.000 second.

	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

Figure 1. This UML sequence diagram shows the interactions among a browser, an application server, and a database.



of a browser, application server, and a database.

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 are proportional to the duration spent servicing the request. I have shown such a diagram in Figure 2. This is 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.

Figure 2. A UML sequence diagram drawn to scale, showing the response time consumed at each tier in the system.

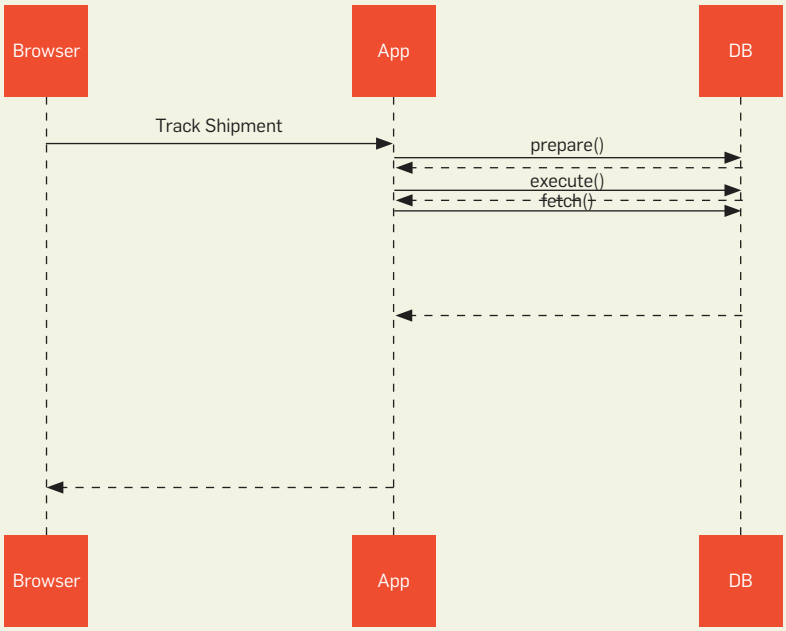
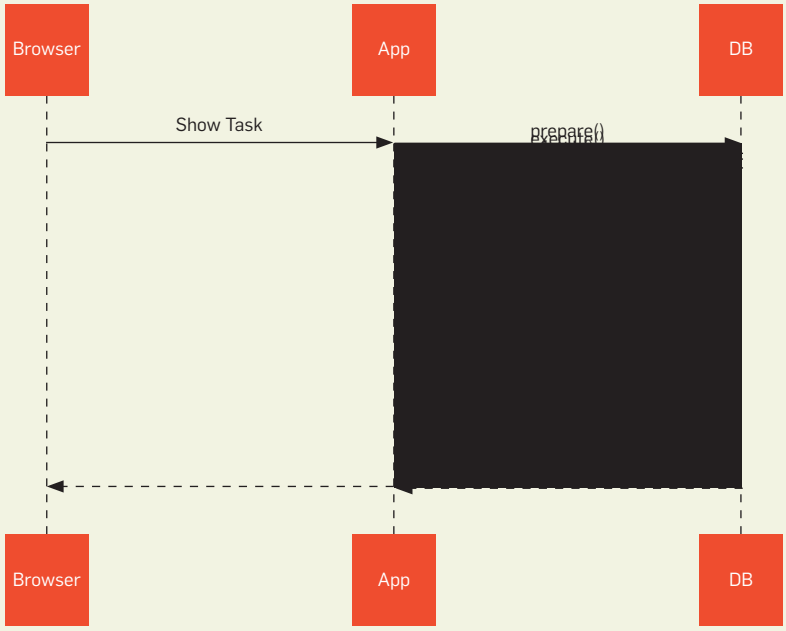


Figure 3. This UML sequence diagram shows 322,968 database calls executed by the application server.



Sequence diagrams are just right for helping people conceptualize how their responses are 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 processing threads work in parallel, and they are good tools for analyzing performance outside of the information technology business.[1]

The sequence diagram is a good conceptual tool for talking about performance, but to think clearly about performance, you need something else. Here's the problem. Imagine the task you're supposed to fix has a response time of 2,468 seconds (41 minutes, 8 seconds). In that period of time, running that task causes your application server to execute 322,968 database calls. Figure 3 shows what the sequence diagram for that task would look like.

There are so many request and response arrows between the application and database tiers that you can't see any of the detail. Printing the sequence diagram on a very long scroll isn't a useful solution, because it would take weeks of visual inspection before you would be able to derive useful information from the details you would see.

The sequence diagram is a good tool for conceptualizing flow of control and the corresponding flow of time. To think clearly about response time, however, you need something else.

The Profile

The sequence diagram does not scale well. To deal with tasks that have huge call counts, you need a convenient aggregation of the sequence diagram so that you understand the most important patterns in how your time has been spent. Table 3 shows an example of a *profile*, which does the trick. A profile is a tabular decomposition of response time, typically listed in descending order of component response time contribution.

Example 4. *The profile in Table 3*

is rudimentary, but it shows exactly where your slow task has spent your user's 2,468 seconds. With the data shown here, for example, you can derive the percentage of response time contribution for each of the function calls identified in the profile. You can also derive the average response time for each type of function call during your task.

A profile shows *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.

From the data shown in Table 3, you *know* that 70.8% of your user's response time is consumed by `DB:fetch()` calls. Furthermore, if you can drill down in to the individual calls whose durations were aggregated to create this profile, you can know how many of those `App:await _ db _ netIO()` calls corresponded to `DB:fetch()` calls, and you can know how much response time each of those consumed. With a profile, you can begin to formulate the answer to the question, "How long should this task run?"... which, by now, you know is an important question in the first step (section 0) of any good problem diagnosis.

Amdahl's Law

Profiling helps you think clearly about performance. Even if Gene Amdahl had not given us Amdahl's Law back in 1967, you would probably have come up with it yourself after the first few profiles you looked at.

Amdahl's Law states: Performance improvement is proportional to how much a program uses the thing you improved. 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 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 *cost* of the remedies you'll be executing.[3]

Example 5. *Consider the profile in Table 4. It's the same profile as in Table 3, except here you can see how much time you think you can save by implementing the best remedy for each row in the profile, and you can see how much you think each remedy will cost to implement.*

Which remedy action would you implement first? Amdahl's Law says that implementing the repair on line 1 has the greatest potential benefit of saving about 851 seconds (34.5% of 2,468 seconds). If it is truly "super expensive," however, then the remedy on line 2 may yield better a net benefit—and that is the constraint to which you really need to optimize—even though the potential for response time savings is only about 305 seconds.

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 much better decisions about what remedies to implement first. Your predictions give you a yardstick for measuring your own performance as an analyst. Finally, it gives you a chance to showcase your cleverness and intimacy with your technology as you find more efficient remedies for reducing response time more than expected, at lower-than-expected costs.

What remedy action you implement first really boils down to how much you trust your cost estimates. Does "dirt cheap" really take into account the risks that the proposed improvement may inflict upon the system? For example, it may seem dirt cheap 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 thinking about 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

Table 3. This profile shows the decomposition of a 2,468.000-second response time.

	Function Call	R (sec)	Calls
1	`DB: fetch()`	1,748.229	322,968
2	`App: await _ db _ netIO()`	338.470	322,968
3	`DB: execute()`	152.654	39,142
4	`DB: prepare()`	97.855	39,142
5	Other	58.147	89,422
6	`App: render _ graph()`	48.274	7
7	`App: tabularize()`	23.481	4
8	`App: read()`	0.890	2
	Total	**2,468.000**	

Table 4. This profile shows the potential for improvement and the corresponding cost (difficulty) of improvement for each line item from Table 2.

	Potential improvement % and cost of investment	R (sec)	R (%)
1	34.5% super expensive	1,748.229	70.8%
2	12.3% dirt cheap	338.470	13.7%
3	Impossible to improve	152.654	6.2%
4	4.0% dirt cheap	97.855	4.0%
5	0.1% super expensive	58.147	2.4%
6	1.6% dirt cheap	48.274	2.0%
7	Impossible to improve	23.481	1.0%
8	0.0% dirt cheap	0.890	0.0%
	Total	**2,468.000**	

Table 5. A skew histogram for the 322,968 calls from Table 2.

	Range {min ≤ e < max}		R (sec)	Calls
1	0	0.000001	0.000	0
2	0.000001	0.00001	0.002	397
3	0.00001	0.0001	0.141	2,169
4	0.0001	0.001	31.654	92,557
5	0.001	0.01	389.662	180,399
6	0.01	0.1	1,325.870	47,444
7	0.1	1	0.900	2
	Total		**1,748.229**	**322,968**

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 rack up successes and propose ever-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 repeatedly encounter sub-problems such as this:

Example 6. *The profile in Table 3 revealed that 322,968* `DB: fetch()` *calls had consumed 1,748.229 seconds of response time. How much unwanted response time would be eliminated if you could eliminate half of those calls? The answer is almost never, "Half of the response time." Consider this far simpler example for a moment:*

Example 7. *Four calls to a subroutine consumed four seconds. How much unwanted response time would be eliminated if you could eliminate half of those calls? The answer depends upon the response times of the individual calls that we could eliminate. You might have assumed that each of the call durations was the average 4/4 = 1 second, but nowhere did the statement tell you that the call durations were uniform.*

Imagine the following two possibilities, where each list represents the response times of the four subroutine calls:

```
A = {1, 1, 1, 1}
B = {3.7, .1, .1, .1}
```

In list A, the response times are uniform, so no matter which half (two) of the calls you eliminate, you will reduce total response time to two seconds. In list B, however, it makes a tremendous difference which two calls are eliminated. If you eliminate the first two calls, then the total response time will drop to .2 seconds (a 95% reduction). If you eliminate the final two calls, then the total response time will drop to 3.8 seconds (only a 5% reduction).

Skew is a nonuniformity in a list of values. The possibility of skew is what prohibits you from providing a precise answer to the question I asked at the beginning of this section. Let's look again:

Example 8. *The profile in Table 3 revealed that 322,968* `DB: fetch()` *calls had consumed 1,748.229 seconds of response time. How much unwanted response time would you eliminate by eliminating half of those calls? Without knowing anything about skew, the most precise answer you can provide is, "Somewhere between 0 and 1,748.229 seconds."*

Imagine, however, that you had the additional information available in Table 5. Then you could formulate much more precise best-case and worst-case estimates. Specifically, if you had information like this, you would be smart to try to figure out how specifically to eliminate the 47,444 calls with response times in the .01- to .1-second range.

Summary

In Part 1, I have tried to link together some of the basic principles that I have seen people trip over in my travels as a software performance analyst. In Part 2, I will describe how competition for shared resources influences performance by covering the concepts of *efficiency*, *load*, *queuing delay*, and *coherency delay*. I will also explain how to think clearly about performance during the design, build, and test phases of an application, so that you'll be much more likely to create fast software that can become even faster throughout its production lifespan. C

Related articles on queue.acm.org

You're Doing It Wrong
Poul-Henning Kamp
http://queue.acm.org/detail.cfm?id=1814327

Performance Anti-Patterns
Bart Smaalders
http://queue.acm.org/detail.cfm?id=1117403

Hidden in Plain Sight
Bryan Cantrill
http://queue.acm.org/detail.cfm?id=1117401

References

1. General Electric Company. What is Six Sigma? The roadmap to customer impact; http://www.ge.com/sixsigma/SixSigma.pdf.
2. Millsap, C. My whole system is slow. Now what? 2009; http://carymillsap.blogspot.com/2009/12/my-whole-system-is-slow-now-what.html.
3. Millsap, C. On the importance of diagnosing before resolving. 2009; http://carymillsap.blogspot.com/2009/09/on-importance-of-diagnosing-before.html.
4. Millsap, C. Performance optimization with Global Entry. Or not? 2009; http://carymillsap.blogspot.com/2009/11/performance-optimization-with-global.html.

Cary Millsap is the founder and president of Method R Corporation (http://method-r.com), a company devoted to software performance. He is the author (with Jeff Holt) of *Optimizing Oracle Performance* (O'Reilly) and a co-author of *Oracle Insights: Tales of the Oak Table* (Apress). He is the former vice president of Oracle Corporation's System Performance Group and is also an Oracle ACE Director and a founding partner of the Oak Table Network, an informal association of well-known "Oracle scientists." He blogs at http://carymillsap.blogspot.com, and he tweets at http://twitter.com/CaryMillsap.