

*A Practitioner's Guide to Optimizing Response Time*



*Optimizing*

# Oracle Performance



**O'REILLY®**

*Cary Millsap  
with Jeff Holt*

---

# Targeting the Right User Actions

One of the first steps in any project is to figure out what the project is supposed to accomplish. The formal written result of figuring out the project's goal is called the project's *specification*. An Oracle performance improvement project, like all sorts of other projects, needs a specification. Otherwise, you have nothing that you can use to measure the success or failure of your project.

Many Oracle performance improvement projects are crippled from their beginnings with poor specifications. You've probably seen the cartoon in which a programmer's manager says, "You start coding. I'll go find out what they want." A lot of people try to "tune their systems" without ever really knowing what they're out to accomplish. On the other hand, there's no need for a system to languish for months while analysts try to construct the "ultimate" project specification, charging time and materials rates while they inch forward. Constructing a good specification for an Oracle performance improvement project should usually consume no more than a couple of hours.

The aim of this chapter is to help you get your performance improvement project started on the right foot, so that your project will optimize the economic value of a system. I'll explore some bad project specifications and explain why they hurt the projects they were supposed to help. I'll describe some specifications that have worked well, resulting in projects that have quickly created great positive economic impact to their systems. Throughout the chapter, I'll list some attributes that have distinguished good specifications from bad ones.

## Specification Reliability

A project specification can be called "reliable" only if any project that successfully fulfills the letter of that specification also fulfills the specification's true intent.

Unfortunately, the most commonly used specifications for performance improvement projects are unreliable. Examples of specifications include:

- Distribute disk I/O as uniformly as possible across many disk drives.
- Ensure that there is at least  $x\%$  of unused CPU capacity during peak hours.
- Increase the database buffer cache hit ratio to at least  $x\%$ .
- Eliminate all full-table scans from the system.

Each of these specifications is unreliable because the letter of each specification can be accomplished without actually producing a desired impact upon your system. There is a simple game that enables you to determine whether you have a reliable specification or not:

To establish whether or not the specification for a performance improvement project is reliable, ask yourself the question: “Is it possible to achieve the stated goal (the specification) of such a project without actually improving system performance?”

One easy way to get the game going is to imagine the existence of an evil genie. Is it possible for an evil genie to adhere to the letter of your “wish” (the project specification) while producing a project result that actually contradicts your obvious underlying goal? If the evil genie can create a system on which she could meet your project specification but still produce an unsatisfactory performance result, then the project specification has been proved unreliable.

The evil genie game is a technique employed in thought experiments by René Descartes in the 1600s and, more recently, by Elizabeth Hurley’s character in the film *Bedazzled*. Here’s how the evil genie game can play out for the bad specifications listed earlier:

*Distribute disk I/O as uniformly as possible across many disk drives*

This specification is a perfectly legitimate goal for trying to prevent performance problems when you are configuring a new system, but it is an unreliable specification for performance *improvement* projects. There are many systems on which making significant improvement to disk I/O performance will cause either negligible or even negative performance impact.

For example, imagine a system in which each of the most important business processes needing performance repair consumes less than 5% of the system’s total response time performing disk I/O operations. (We have hundreds of trace files that fit this description at [hotsos.com](http://hotsos.com).) On such a system, no amount of I/O “tuning” can create meaningful response time improvement of more than 5%. Since distributing disk I/O uniformly across many disk drives can result in a system without meaningfully improved performance, this specification is unreliable.

*Ensure that there is at least  $x\%$  of unused CPU capacity during peak hours*

There are several ways that an evil genie could accomplish this goal without helping the performance of your system. One way is to introduce a horrific disk I/O bottleneck, such as by placing the entire database on one gigantic disk drive

with excessively poor I/O-per-second capacity. As more and more user processes stack up in the disk I/O queue, much CPU capacity will go unused. Since increasing the amount of unused CPU can result in worse performance, this specification is unreliable.

*Increase the database buffer cache hit ratio to at least x%*

This one's easy: simply use Connor McDonald's innovative demonstration that I include in Appendix C. The application will show you how to increase your database buffer cache hit ratio to as many nines as you like, by adding CPU-wasting unnecessary workload. This additional wasted workload will of course degrade the performance of your system, but it will "improve" your buffer cache hit ratio. Connor's application is, of course, a trick designed to demonstrate that it is a mistake to rely on the buffer cache hit ratio as a measure of system "goodness." (I happen to know that Connor is definitely *not* evil, although I have on occasion noticed him exhibit behavior that is at least marginally genie-like.)

There are subtler ways to degrade a system's performance while "improving" its cache hit ratio. For example, SQL "tuners" often do it when they engage in projects to eradicate TABLE SCAN FULL row source operations (discussed again in the next specification I'll show). Another way an evil genie could improve your cache hit ratio in a way that harms performance is to reduce all your array fetch sizes to a single row [Millsap (2001b)]. Because it is so easy to increase the value of your buffer cache hit ratio in ways that degrade system performance, this specification is particularly unreliable.

*Eliminate all full-table scans from the system*

Unfortunately, many students of SQL performance optimization learn early the *untrue* rule of thumb that "all full-table scans are bad." An evil genie would have an easy time concocting hundreds of SQL statements whose performance would degrade as TABLE SCAN FULL row source operations were eliminated [Millsap (2001b); (2002)]. Because eliminating full-table scans can actually degrade performance, the action is an unreliable basis for a performance improvement project specification.

The cure for unreliable performance improvement specifications is conceptually simple. Just say what you mean. But of course, by the same logic, golf is simple: just hit the ball into the hole every time you swing. The problem in curing unreliable performance improvement specifications is to figure out how to specify what you really mean in a manner that doesn't lead to other errors. For example, a performance specification that comes closer to saying what you really mean is this one:

Make the system go faster.

However, even this specification is unreliable. I've seen dozens of projects with this specification result in ostensible success but practical failure. For example, a consultant finds, by examining V\$SQL, a batch job that consumes four hours. He "tunes" it so that it runs in 30 minutes. This is a project success; the consulting engagement summary says so. However, the success was meaningless. The batch program was

already as fast as it needed to be, because it ran in an otherwise vacant eight-hour batch window. The expensive input into performance improvement (the consultant's fee) produced no positive value to the business.

Worse yet, I've seen analysts make some program *A* go faster, but at the expense of making another vastly more important program *B* go slower. Many systems contain process interdependencies that can cause this situation. On these systems, "tuning" the wrong program not only consumes time and money to execute the tuning project, it results in actual *degradation* of a system's value to the business (see the section "Case 1: Misled by System-Wide Data" in Chapter 12 for an example).

This "make the system go faster" specification is just too vague to be useful. In my service line management role at Oracle Corporation, I had many discussions about how to specify projects—the whole idea of packaged services requires contract-quality specification of project goals. Most participants in those meetings understood very quickly that "make the system go faster" is too vague. What I find remarkable today is that most of these people saw the vagueness in entirely the wrong place.

Most people identify the *go faster* part of the specification as the root of the problem. People commonly suggest that "make the system go faster" is deficient because the statement doesn't say, numerically, how *much* faster. In my Oracle meetings, explorations of how to improve "make the system go faster" generally led to discussion of various ways to measure actual and perceived speeds, ways to establish "equivalency" metrics such as count-based utilization measures (like hit ratios), and so on. Of course, the search for "equivalency" measures finds a dead end quickly because—if you execute the evil genie test correctly—such presumed equivalency measures are usually unreliable.

Figuring out how much faster a system "needs to go" often leads into expensive project rat holes. (An exception is when an analyst has found the maximum allowable service time for an operation by using a model like the queueing theory one that I describe in Chapter 9.) When our students today discuss the "make the system go faster" spec, it usually takes very little leading for students to realize that the real problem is actually hidden in the word *system*. For example, consider the following commonly suggested "improvements" to the original "make the system go faster" specification:

- Make the system go 10% faster.
- Make the system complete all business functions in less than one second.

First of all, each specification expressed in this style is susceptible to the same evil genie tricks as the original spec. But by adding detail, we've actually weakened the original statement. For example:

*Make the system go 10% faster*

Do you really expect that *every* business transaction on the system can go 10% faster? Even those that perform only a couple of Oracle logical I/O calls (LIOs)

to begin with? On the other hand, is 10% really enough of an improvement for an online query that consumes seventeen minutes of response time?

*Make the system complete all business functions in less than one second*

Is it really good enough for a single-row fetch via a primary key to consume 0.99 seconds of response time? On the other hand, is it really reasonable to expect that an Oracle application should be able to emit a 72-page report in less than one second?

Do these two formats actually lead to an improvement of the original “make the system go faster” specification? They do not. A bigger problem is actually the lack of definition for the word “system.”

## The System

What is *the system*? Most database and system administrators interpret the term much differently than anyone else in the business does. To most database and system administrators, *the system* is a complex collection of processes and shared memory and files and locks and latches, and all sorts of technical things that can be measured by looking at “V\$ tables” and operating system utilities and maybe even graphical system monitoring dashboards. However, *nobody* else in the business sees a system this way. A user thinks of *the system* as the collection of the few forms and batch jobs in that user’s specific job domain. A manager thinks of *the system* as a means for helping improve the efficiency of the business. To users and managers, the redness, yellowness, or greenness of your dashboard dials is completely and utterly irrelevant.

Here’s a simple test to determine for yourself whether I’m telling the truth. Try to imagine yourself as a user who has just waited two hours past your reporting deadline this morning because your “fifteen-minute report” required three full hours to run. Try to imagine your reaction to a database administrator who would say the following words in front of your colleagues during a staff meeting: “There was absolutely nothing wrong with the system while your report was running, because all our dashboard dials were green during the entire three-hour period.”

Please remember this when you are acting in the role of performance analyst: a *system* is a collection of end-user programs. An end-user is watching each of these programs attentively. (If no one is watching a particular program attentively, then it should be running only during off-peak time periods, or perhaps not at all.) The duration that each program requires to deliver a requested chunk of business value is that program’s response time. The response time of an individual user action is practically the only performance metric that your business cares about. Hence:

Response time for an end-user action is the first metric that you should care about.

## Economic Constraints

When you eliminate the ambiguity of the word “system,” you take one big step closer to a foolproof goal:

Improving the performance of program A during the weekday 2:00 p.m. to 3:00 p.m. window is critical to the business. Improve the performance of A as much as possible for this time period.

But is this specification evil genie–proof? Not yet. Imagine that the average run time of program A is two minutes. Suppose that the evil genie could reduce the response time from two minutes to 0.25 seconds. Great... But at a cost of \$1,000,000,000. Oops. Maybe improving response time only to 0.5 seconds would have been good enough and would only have cost \$2,000. The specification omits any mention of an economic constraint.

There is an optimization project specification that I believe may actually be evil genie–proof. It is the optimization goal described by Eli Goldratt in [Goldratt (1992), 49]:

Make money by increasing net profit, while simultaneously increasing return on investment, and simultaneously increasing cash flow.

This specification gives us the ultimate acid test by which to judge any other project specification. However, it does fall prey to the same “hit the ball into the hole on every swing” lack of detail that I discussed earlier.

## Making a Good Specification

Let’s stop fooling around with faulty project specifications and start constructing some good ones. It shouldn’t take you more than a couple of hours to create a good specification for most performance improvement projects. Here’s how:

1. Identify the user actions that the *business* needs you to optimize, and identify the contexts in which those actions are important.
2. Prioritize these user actions into buckets of five.
3. For each of the actions in your top bucket, determine *whom* you can observe executing the action in its suboptimal context and *when* you can make the observation.

## User Action

In this book, I try to make a careful distinction between *user actions*, *programs*, and *Oracle sessions*. A *user action* is exactly what it sounds like: an action executed by a user. Such an action might be the entry of a field in a form or the execution of one or more whole programs. A user action is defined as some unit of work whose output and whose performance have meaning to the business. The notion of *user action* is especially important during project specification because the user action is precisely the unit of work that has business meaning.

A *program* is of course a sequence of computer instructions that carries out some business function. A user action might be a program, a part of a program, or multiple programs. An *Oracle session* is a specific sequence of database calls that flow through a connection between a user process and an Oracle instance. A program can initiate zero or more Oracle sessions, and in some configurations, more than one program can share a single Oracle session. The notion of an *Oracle session* is important during data collection because the Oracle kernel keeps track of performance statistics at the Oracle session level.



Oracle does make a distinction between a *connection* (a communication pathway) and a *session*. You can be connected to Oracle and not have any sessions. On the other hand, you can be connected and have many simultaneous sessions on that single connection.

## Identifying the Right User Actions and Contexts

The first step in your specification is to identify the user actions that the *business* needs you to optimize. If you mess up this step, it is likely that your performance improvement project will fail. It is vital for you to obtain a list of *specific user actions*. The ones you select should be the ones that are the most important in the business's pursuit of net profit, return on investment, and cash flow.

I emphasize “that the *business* needs you to optimize” because you are specifically *not* looking for a database administrator's opinion about performance at this point. One of the most common mistakes that Oracle performance analysts make is that they consult their V\$ views to learn where their system needs “tuning.” Your V\$ views can't tell you. I'll describe in Chapter 3 some of the technical reasons why it's unreliable to consult your V\$ views for this information.

Finding out what your business needs is usually easy. It is almost never the result of a long goal-definition project. It is almost always the result of asking a business leader who speaks in commonsense language, “If we could make one program faster by the end of work today, which program would you choose?” The following examples illustrate the type of response that you're looking for:

- We manufacture disk drives. We have a warehouse full of disk drives that are ready to ship. We receive hundreds of telephone calls each morning from angry customers who placed orders with us over two weeks ago, demanding to know the status of their shipments. At any given time, there is an average of over two dozen empty FedEx trucks parked at our loading dock. If you go down to the loading dock, you can see that our packers and the truck drivers are sitting on boxes drinking coffee right now. They can't load the boxes on the trucks because the program that prints shipping labels is too slow. Our business's most important performance problem is the program that prints shipping labels.



- We're spending too much on server license and maintenance fees. We have 57 enterprise-class servers in our shop, and we need to cut that number to ten or fewer. We already house 80% of our enterprise data on one large storage area network (SAN). However, our total CPU workload that is presently distributed across 57 servers is probably too large to fit onto ten machines. Our business's most important performance problem is eliminating enough unnecessary CPU workload so that we can perform the server consolidation effort and ditch about fifty of our servers.

The hardest part is usually gaining access to the right people in the business to get the information you need. You might have to dig a little bit for your list. The following techniques can help:

*Ask your boss where the performance risks are*

Steer him away from answers that refer to technical components of the database. Force the conversation into the domain of user language. Ask which user is giving him the most flak about system performance, and then book a lunch with the user. The loudest user is not necessarily the one with the business's most critical problem, but understanding that user's problems are probably a good start.

*Take a user to lunch*

Buy him a sandwich, and ask down-to-earth questions like, "If I could make something you use faster today, what would you want it to be?"

*Find a sales forecast for your business*

Consider which application processes are going to be the most important ones to facilitate your company's planned sales growth. Are those processes running as efficiently as they can?

If you get stuck in your conversations with people with whom you're trying to identify user actions that are important to the business, ask them which actions fit into these categories:

- Actions that are business critical
- Actions that run a long time
- Actions that are run extremely often
- Actions that consume a lot of capacity of a resource you're trying to conserve

In addition to identifying which user actions require optimization, you need to identify the *context* in which those actions are important. For example:

- Is the action always slow?
- Is it slow only at a particular time of day (week, month, or year)?
- Is it slow only when it runs at the same time as some other program(s)?
- Is it slow only when the number of connected users exceeds some threshold?

- Is it slow only after some other program runs (upload, delete, etc.)?

Without context, you run the risk that you'll collect performance diagnostic data for the "problem" action and then find after all your effort that there's apparently nothing wrong with it. You have to identify how to find the user action when it is performing at its worst. Otherwise, you're not going to be able to see the problem. This concept is so important that I'll say it again:

You have to identify how to find the user action when it is performing at its *worst*.

In this step, it is usually important to select more than one user action, especially in situations where many users perceive many different performance problems. This is true even in situations where the number-one system performance problem has a priority that clearly exceeds everything else on the system. The reasons for this advice come from the experience of using the method many times:

- Because cost is a factor in net benefit, the business net benefit of improving, for example, user action #3 may actually exceed the business net benefit of improving user action #1.
- Producing significant improvement quickly in *any* of a system's top five most important performance problems can create a significant political advantage, including factors like project team morale and project sponsor confidence.
- You might not know how to improve performance for user action #1. But fixing, for example, user action #3 may eliminate so much unnecessary workload that #1 becomes a non-issue.
- You can't tell which performance improvement action will produce the greatest net benefit to the business until you can see a high-level cost-benefit analysis for the user actions in your top-five bucket.

## Prioritizing the User Actions

Once you have constructed the list of candidate user actions, you need to rank the importance of their improvement to the business. Everything you do later will require that you have chosen the most important actions to optimize first. Business prioritization is vital for several reasons, including:

*The most important actions will get fixed the soonest*

This is the most important reason. Quite simply, if you don't optimize the most important business processes first, then you're not optimizing.

*Trade-off decisions will always favor more important user actions*

On occasion, you may find that an optimization for one user action inflicts a performance penalty upon another. This happens frequently when the optimization strategy you choose is to increase the capacity of some component. However, because I hope to convince you to increase capacity only when necessary (that is, *rarely*), such trade-offs should be rare.

### *Less important user actions enjoy collateral benefits*

The term *collateral damage* has been introduced into our language by discussions of accidents that occur during wartime. The opposite of collateral damage is *collateral benefit*—a benefit yielded serendipitously by attending to something else. Collateral performance benefits occur frequently on computer systems in which we eliminate huge amounts of unnecessary work.

It's easy to over-analyze at this stage, but there's actually no need to spend much time here. All you need are rough categories. I recommend grouping your user actions into prioritized buckets of no fewer than five. This way, you won't be tempted to obsess over the precise ranking of actions that are close in importance. For example, if you have ten important problem user actions, then create no more than two groups of five. If you have more than ten problem actions (I've visited sites whose lists numbered in excess of fifty), then I suggest partitioning your list into three parts:

1. The five most important user actions (your first bucket).
2. The five next most important user actions (your second bucket).
3. The remainder of the important user actions you've listed (the union of your third and subsequent buckets)

Be especially wary of executing any prioritization task with the participation of large groups. Every user, of course, will try to convince you that his actions are the very most supremely important actions on the entire system. And of course, every action on the system cannot take top priority. Most of the time that you might spend negotiating whether a user action belongs in one group or another could be invested more wisely in other steps of the method. If you find that the whole prioritization task is consuming more than just a few minutes, then step back and just make some sensible decisions. Assure the users whose actions don't fall into the top priority class that they haven't lost anything; you'll attend to their problems too.

## **Determining Who Will Execute Each Action and When**

The final step in the construction of a good spec for your performance improvement project is the specification of how you'll be able to find each targeted action when it next runs in its targeted context. This information will allow you to find the programs implementing those actions so that you can measure their performance.

Often, the success of a diagnostic data collection effort will be determined by your ability to establish simple human contact with a person who will execute the slow action and answer the following simple questions:

- When is the next time that this person expects for the action to exhibit the performance problem?
- How can you watch?

The answers to these questions unambiguously define the parameters you'll use for your diagnostic data collection process, which I describe in Chapter 3.

If you have a tool that constantly monitors the appropriate performance statistics for every individual user action on your system, then predicting who will run a problem program and when it will happen becomes unnecessary. The luxury of having such data for every user action on your system will allow you to respond to a complaint about an action in the recent past instead of having to predict their occurrences in the imminent future. Such tools are expensive, but they do exist.

If you do not own such a tool, then you'll have to be more selective in which diagnostic data you'll want to collect, and the step described in this section will be essential. For you, I hope that Chapters 6 and 8 will provide significant value.

## Specification Over-Constraint

I've discussed the reliability problems introduced by specifications that are too vague. Equally devastating is the specification that is too precise. Many specifications that go into too much detail actually conflict with the optimization goal. A specification that requires some specific program performance improvement *and* a 10-point improvement in the database buffer cache hit ratio might actually be impossible to achieve. It is entirely possible that improving the performance of a specified program might result in a dramatically lower system-wide cache hit ratio. (See [Millsap (2001b)] for an example.)

Another fun example occurred several years ago when I was an Oracle Corporation employee. A performance specification required that, on a particular client-server application form, navigation from one field to the next must occur within 0.5 seconds. The specification further required for the client system to be in Singapore and for the server system to be in Chicago. Furthermore, the specification required that we could not modify the prepackaged application, which made an average of six synchronous database calls across the wide-area network (WAN) per field.

The objective as stated in the specification was unachievable, because the specification is *over-constrained*; it in fact conflicts with the physical laws of our universe. There is no way that six round-trip network transmissions can occur between Singapore and Chicago within the span of half a second. Even if we could eliminate all components of response time except for the theoretically smallest amount of time required for the data transmission at its fastest theoretically possible rate (that is, if we could ignore the time consumed by cable, hubs, routers, the database, and so on), executing six round-trip communications per field will require *at least* 0.6 seconds per field.

*Proof:* Assume that all practical influences other than the speed of light have no effect upon performance of field-to-field navigation. The speed of light in a vacuum is approximately 299,792,458 meters per second. The distance along the Earth's surface

from Singapore to Chicago is approximately 15,000,000 meters. Therefore, the distance traversed by six round-trips for each field is  $2 \times 6 \times 15,000,000$  meters, or approximately 180,000,000 meters per field. Obeying the relationship  $d = rt$ , we find that  $t = d/r \approx 0.6$  seconds per field. Reintroducing all of the practical influences upon performance that we have ignored up to now will only degrade performance further. Therefore, the requirement specification cannot be met. *QED*.

There is *no way* to meet this specification without relaxing at least one of its constraints. The most important constraint to eliminate first was the requirement that each field must execute an average of six round-trips between the client and the database server. The most important task of the existing performance improvement project was to show the proof of why any project with the given specification was doomed to failure. Until this proof became known, people on the project had continued to waste time and money in pursuit of an unattainable goal.

Good projects don't come from bad project specifications. Whether the problem is sloppy targeting or a specification that is utterly unattainable, you cannot afford to base your performance improvement project upon a faulty specification.