

A Practitioner's Guide to Optimizing Response Time



Optimizing

Oracle Performance



O'REILLY®

*Cary Millsap
with Jeff Holt*

Targeting the Right Diagnostic Data

Once you have correctly targeted the user actions for which the business most needs performance improvement, it is data collection time. Diagnostic data collection is the project phase in which the typical performance analyst really begins to feel a sense of progress. There are very few arguments in Oracle literature today about how one should go about collecting performance diagnostic data. However, there should be. The way that you collect your diagnostic data has a tremendous influence over a project's potential for success. In fact, unless you are exceptionally lucky, a performance improvement project cannot proceed beyond a botched data collection.

I hope that this chapter will surprise you. It describes a couple of very important flaws in the standard data collection procedures that are deeply institutionalized in the Oracle culture. In the hundreds of flawed Oracle performance improvement projects that my colleagues and I have helped repair, a contributing factor to failure in nearly every project was one or more errors in data collection. Unfortunately, virtually every document written about Oracle performance prior to the year 2000 leads its reader to make these errors. I believe that the commonsense examples described in this chapter will forever change your attitude toward diagnostic data collection.

Expectations About Data Collection

The whole point of data collection in a performance improvement project using Method R is to collect response time data for a *correctly targeted user action*. No more, no less. Unfortunately, many application designers have complicated the data collection job significantly by providing insufficient instrumentation in their applications.



Many companies, especially Oracle Corporation, are improving the response time instrumentation in newer application releases.

The data collection lessons you learn in this chapter will make data collection seem more difficult than you had probably expected. The benefit of doing it right is that

you will reduce your overall project costs and durations by eliminating expensive and frustrating trial-and-error analysis/response iterations.

A Method R performance improvement project proceeds much differently than a project that uses the conventional trial-and-error approach introduced as Method C in Chapter 1. Figure 3-1 illustrates the difference. A project practitioner typically begins to feel like he is making real progress when the targeting and data collection phases are complete and he enters the analysis/response phase of a performance improvement project. The Method C practitioner typically reaches this milestone (marked t_1 in Figure 3-1) before the Method R practitioner working on the same problem would (marked t_2). If you don't expect this, it can become a political sensitivity in a Method R project. The time between t_1 and t_2 is when your risk of losing commitment to the new method is at its greatest.

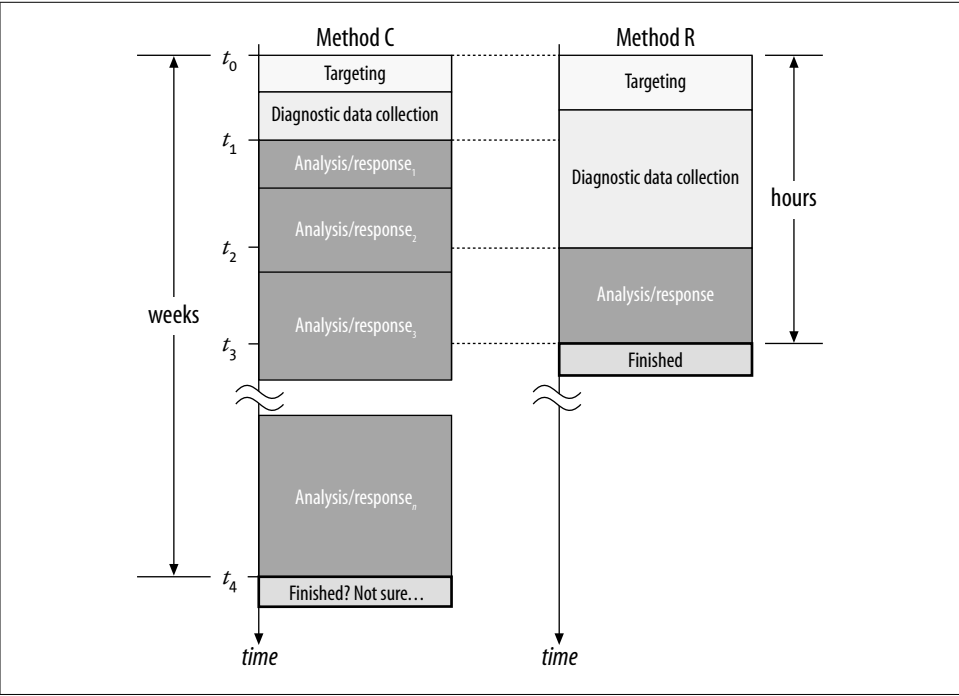


Figure 3-1. The targeting and diagnostic data collection phases of Method R consume more time than in conventional methods, but total project duration is typically much shorter

Finishing the data collection phase quickly is not the goal of a performance improvement project. The correct goal is to optimize the system with the smallest possible investment of resources. Method R is optimized for this goal. In fact, my colleagues and I created Method R specifically to help our customers fix performance improvement projects that had dragged on for weeks or even months without meaningful progress. In the overwhelming majority of Method R projects we've led, we've

been able to demonstrate how to achieve the optimization goal within one hour of obtaining correctly scoped diagnostic data. Once you have collected the *right* performance diagnostic data, Method R will require only a single analysis/response phase before you'll make progress against your targeted user action.

Method C practitioners spend most of their time in trial-and-error mode trying to determine the cause-effect relationship between the hundreds of possible problem causes and the business symptom that is most in need of improvement. A huge inefficiency of Method C is the need to perform, on average, several iterations of analysis and response activities before you'll stumble upon a symptom's root cause. Each iteration of analysis and response tends to consume more time than the prior one, because analysts usually try the easiest responses they can think of first, saving the more time-consuming and expensive tuning activities for later in the project after the cheaper ideas are discarded.

The final blow to Method C is that there's really no quantitative way to determine when you're "finished tuning." In many projects, Method C users never positively identify an actual contributory cause of a performance problem. Even in "successful" projects, practitioners spend weeks, months, or even years without really knowing whether a targeted performance problem has been truly perfected (*optimized*) or merely partially improved (*tuned*). The problem of not knowing whether a user action could be further tuned leads to a condition that Gaja Vaidyanatha and Kirti Deshpande cleverly call *Compulsive Tuning Disorder*, or CTD [Vaidyanatha and Deshpande (2001) 8]. I joke that CTD is a debilitating condition caused by *hope*. More specifically, CTD is caused by an absence of complete information that would allow you to prove conclusively whether the performance of a given user action has any room for improvement. Method R fills this information gap, eliminating the possibility of CTD.

The first time you use Method R, collecting the diagnostic data will probably be the most difficult phase of your project. For some applications, diagnostic data collection is a cake walk. For other applications, proper diagnostic data collection can legitimately become quite a difficult challenge. Chapter 6 describes which kinds of applications are easy and which are hard, and it illustrates some of the techniques that my colleagues and I have used to overcome various challenges. The good news is that once you've figured out how to collect good diagnostic data for a targeted user action in your application, the process will be much easier and less time-consuming on your next performance improvement project. Method C, on the other hand, will always suffer from the problem of multiple analysis/response iterations, regardless of where you are on the experience curve.

I believe that in the future, most application software vendors will make it very easy for users and analysts alike to collect precisely the diagnostic data that Method R requires. Newer releases of Oracle's E-Business Suite are simplifying the diagnostic data collection process, and everything I hear indicates that the Oracle release 10

kernel and application server software are headed in the same direction. If the dominance of methods analogous to Method R in other industries is any indication, then success in simplifying diagnostic data collection should practically universalize the adoption of Method R for Oracle performance improvement projects.

Different Methods for Different Performance Problems?

Could it be that conventional methods are more effective for “simple” performance tuning problems, and that Method R is more effective for “complex” ones? The problem with that question is this: How do you know whether a performance tuning problem is “simple” or “complex” without engaging in some kind of diagnostic data collection?

One approach that we considered during the construction of Method R was to collect very easy-to-obtain diagnostic data to use in deciding whether the more difficult-to-obtain diagnostic data were even necessary to collect. We found this approach to be sub-optimal. The problem with it is that there’s virtually no situation in which you can be *certain* about cause-effect relationships without the *correct* diagnostic data (and of course, sometimes the *correct* diagnostic data are difficult to obtain). The doubt and ambiguity that are admitted into a project by the analysis of easy-to-obtain diagnostic data rapidly deteriorate the efficiency of a performance improvement project. The thought-blocking fixations that I’ve seen caused by bad diagnostic data at many projects remind me of a wonderful quotation attributed to Cardinal Thomas Wolsey (1471–1530): “Be very, very careful what you put into that head, because you will never, ever get it out.”

A dominant goal during the construction of Method R was that it must be *deterministic*. Determinism is a key attribute that determines how teachable (or automate-able) a method can be. We wanted to ensure that any two people executing Method R upon a given performance problem would perform the same sequence of tasks, without having to appeal to experience, intuition, or luck to determine which step to take next. Our method achieves this by creating a single point of entry, and a well-defined sequence of if-then-else instructions at every decision point thereafter.

Data Scope

Good Oracle performance data collection requires good decision-making in two dimensions. You must collect data for the right *time scope* and the right *action scope*. Let’s begin by drawing a user action’s response time consumption as a sequence of chunks of time spent consuming various resources. Figure 3-2 shows the result. To keep it simple, our imaginary system consists of only three types of resource, called C, D, and S. Imagine that these symbols stand for CPU, disk, and serialization (such as the one-at-a-time access that the Oracle kernel imposes for locks, latches, and

certain memory buffer operations). In Figure 3-2, the time dimension extends in the horizontal direction.

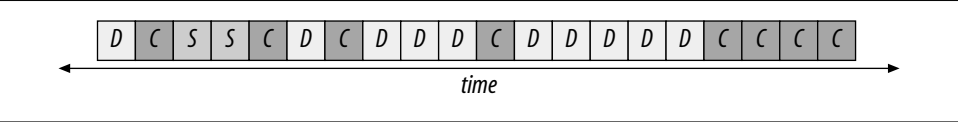


Figure 3-2. The consumption of three types of resource over the duration of a user action

We can denote a system that is executing several user actions at the same time by stacking such drawings vertically, as shown in Figure 3-3. In this drawing, the action dimension extends in the vertical direction.

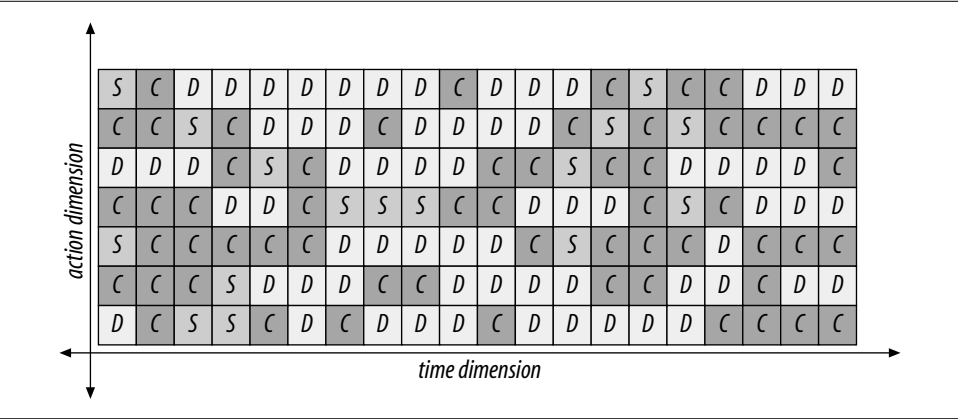


Figure 3-3. By adding a vertical dimension, this drawing depicts a system containing seven concurrent actions, each consuming three different types of resource through time

The following sections use this graphical notation to illustrate why the data collection methods that many Oracle experts have been teaching since the 1980s are actually what have been killing performance improvement projects all over the world.

Scoping Errors

In the system shown in Figure 3-3, imagine that the targeting process described in Chapter 2 has revealed the following: the most important performance problem for the business is that a user named Wallace endures an unacceptably long response time between times t_1 and t_2 , as shown in Figure 3-4.



In the following discussions, I shall use the mathematical notation for a *closed interval*. The notation $[a, b]$ represents the set of values between a and b , inclusive:

$$[a, b] = \{\text{all } x \text{ values for which } a \leq x \leq b\}$$

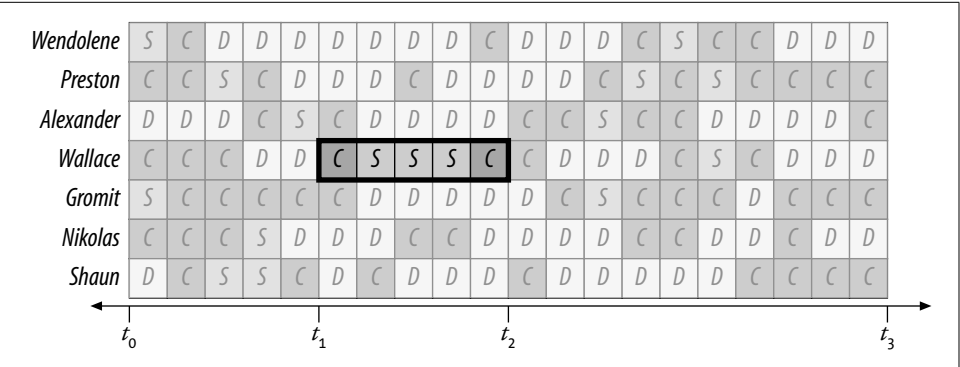


Figure 3-4. This system’s most important user, Wallace, experiences unacceptable performance in the time interval $[t_1, t_2]$

From the picture in Figure 3-4, it is easy to see that during the problem time interval, Wallace’s response time was consumed predominantly by S and secondarily by C, as shown in Table 3-1. Of course, repairing Wallace’s performance problem will require a reduction in time for Wallace’s action spent consuming either S, or C, or both. Amdahl’s Law indicates that any percentage reduction in consumption of S will have 1.5 times the response time impact that an equivalent percentage reduction in consumption of C will have, because the response time contribution of S is 1.5 times the size of the response time contribution of C.

Table 3-1. Resource profile for Wallace’s action for the time interval $[t_1, t_2]$

Resource	Elapsed time	Percentage of total time
S	3	60.0%
C	2	40.0%
Total	5	100.0%

Perhaps the most common data collection error is to collect data that are aggregated in both dimensions. Figure 3-5 shows what this mistake looks like. The heavy, dark line around all the blocks in the entire figure indicate that data were aggregated for all processes (not just Wallace’s), and for the whole time interval $[t_0, t_3]$ (not just $[t_1, t_2]$). Counting the time units attributable system-wide during the $[t_0, t_3]$ interval produces the resource profile shown in Table 3-2. As you can see, Wallace’s performance problem—which we know to have been too much time spent doing S—has been thoroughly buried by all of the irrelevant data that we collected. The result of the botched data collection will be a longer and probably less fruitful performance improvement project than we want.

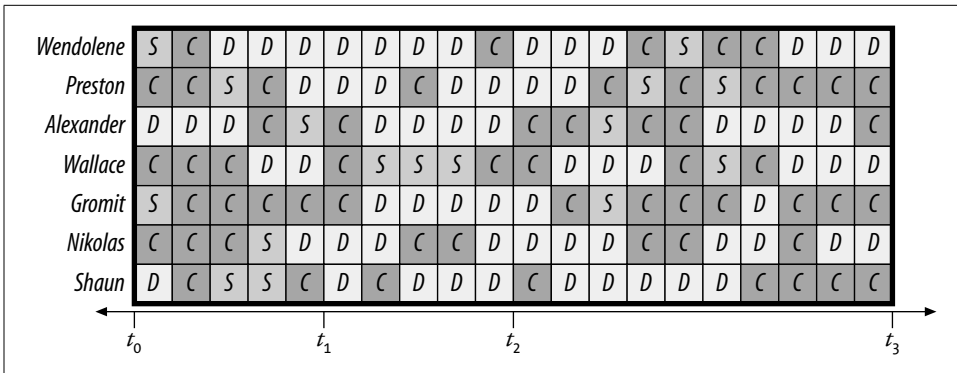


Figure 3-5. Collecting data that are improperly scoped on both the time and action dimensions will completely conceal the nature of Wallace’s problem in the time interval $[t_1, t_2]$

From the data shown in Table 3-2, you simply cannot see that *S* is Wallace’s principal problem root cause. It would actually be *irresponsible* to assume that *S* might be the root cause of Wallace’s problem.

Table 3-2. Resource profile for the entire system for the time interval $[t_0, t_3]$

Resource	Elapsed time	Percentage of total time
D	66	47.1%
C	58	41.4%
S	16	11.4%
Total	140	100.0%

Unfortunately, the deeply flawed data collection method illustrated here is the default behavior of *Statspack*, the *utlbstat.sql* and *utlestat.sql* script pair, and virtually every other Oracle performance tool created between 1980 and 2000. Of the most deeply frustrating performance improvement projects with which I’ve ever assisted, this type of data collection error is far and away the most common root cause of their failure.

The remedy to the data collection problem must be executed on *both* dimensions. Repairing the collection error in only one dimension is not enough. Observe, for example, the result of collecting the data shown in Figure 3-6. Here, the time scoping is done correctly, but the action scope is still too broad. The accompanying resource profile is shown in Table 3-3. Again, remember that you *know* the root cause of Wallace’s performance problem: it is a combination of *S* and *C*. But the data collected system-wide provides apparent “evidence” quite to the contrary, even though the data were collected for the correct time interval.

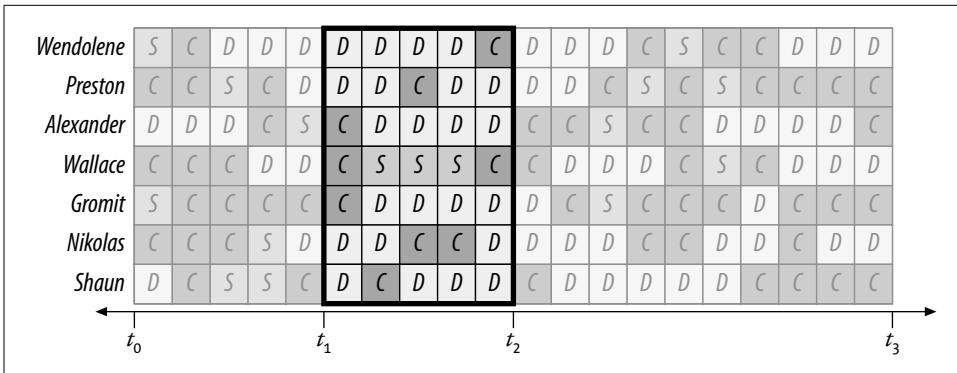


Figure 3-6. Collecting data that are scoped improperly on the action dimension also conceals the nature of Wallace’s performance problem, even though the data were collected for the correct time scope

Table 3-3. Resource profile for the entire system for the time interval $[t_1, t_2]$

Resource	Elapsed time	Percentage of total time
D	23	65.7%
C	9	25.7%
S	3	8.6%
Total	35	100.0%

Finally, examine the result of collecting data for the correct action scope but the wrong time scope, as shown in Figure 3-7. Table 3-4 shows the resource profile. Once again, presented with these data, even a competent performance analyst will botch the problem diagnosis job. Wallace’s problem is S and C, but you certainly wouldn’t figure it out by looking at Table 3-4.

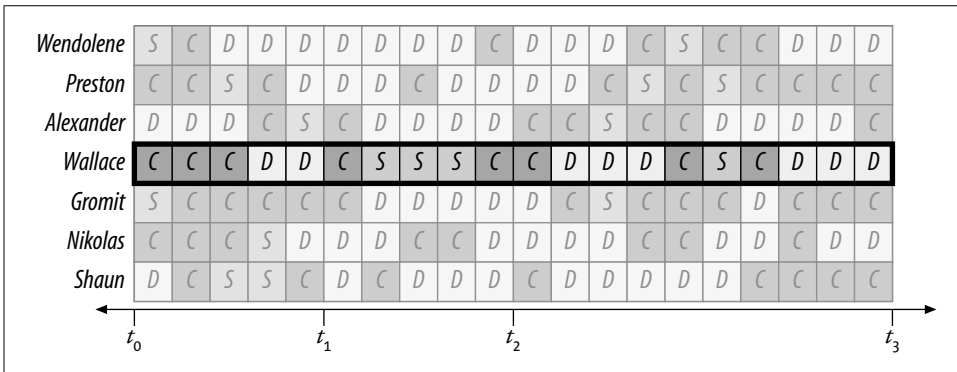


Figure 3-7. Collecting data that are scoped improperly on the time dimension also conceals the nature of Wallace’s performance problem, even though the data were collected for the correct action scope

Table 3-4. Resource profile for Wallace’s action for the time interval $[t_0, t_3]$

Resource	Elapsed time	Percentage of total time
<i>D</i>	8	40.0%
<i>C</i>	8	40.0%
<i>S</i>	4	20.0%
Total	20	100.0%

From this sequence of simple examples, it is easy to see why proper diagnostic data collection is so vital to a performance improvement project. The examples also clearly reveal the identity of the two dimensions along which you can assess whether or not a given diagnostic data collection can be deemed *proper*:

Reliable problem diagnosis cannot proceed unless the data collection phase produces response time data for exactly the right *time scope* and exactly the right *action scope*.

Long-Running User Actions

When you have a really long-running user action, do you need to collect performance diagnostic data for the whole thing? Perhaps you have an action that ran in ten minutes last week, but today it has already run for over four hours, and you’re wondering whether you should kill it. Do you have to restart the job in order to collect diagnostic data for it? Sometimes, I hear about batch jobs that run for several *days* before their users give up and terminate the jobs instead of letting them finish.* Do you really need to collect performance diagnostic data for the whole job?

The answer is no. Of course, collecting performance diagnostic data for some subset of an action’s performance problem duration introduces a type of time-scoping error, but it is actually useful to collect time-subset diagnostic data in some circumstances. For example:

- If a user action is supposed to run in n minutes, then collecting data for just $n + m$ minutes will reveal at least m minutes of response time that shouldn’t exist. For example, if a job is supposed to run in 10 minutes, then 25 minutes’ worth of diagnostic data will reveal at least 15 minutes of workload that shouldn’t exist.
- If a user action consists of a long sequence of repetitive tasks, then performance diagnostic data collected for a small number of the tasks will reveal the resources consumed by the whole action, as long as the tasks are homogeneous.

In Chapter 6, I discuss some collection errors that might occur if your data collection process begins in the midst of a database action. But in many cases, collecting time-subset diagnostic data can help you along your way.

* In some of these cases, I’ve been able to prove that if the job were left to run to completion, it would not be able to complete in our lifetimes.

“Too Much Data” Is Really Not Enough Data

It is tempting to say that the scoping problems in Tables 3-2 through 3-4 were the result of collecting “too much data.” However, the problem with these three resource profiles was not necessarily in what data were *collected*, it is more an issue of how the data were *aggregated*. Look again at Figure 3-5. There is plenty of information here to produce a correctly scoped resource profile. The problem with Table 3-2 is in how the data from Figure 3-5 were aggregated. The same can be said for Figures 3-6 and 3-7 and their resource profiles. The problem is not that the figures contain too *much* data, it’s that their corresponding resource profiles are aggregated incorrectly.

Poor aggregation is an especially big problem for projects that use SQL queries of Oracle V\$ fixed views as their performance diagnostic data sources. Oracle V\$ views by their nature provide data that are either aggregated for an entire instance since instance startup, or for an entire session since connection. For example, using V\$SYSSTAT or V\$SYSTEM_EVENT is *guaranteed* to produce the action scoping errors depicted in Tables 3-2 and 3-3. Even meticulous use of V\$SESSTAT and V\$SESSION_EVENT makes you prone to the type of time scoping error depicted in Table 3-4 (as you can see by experimenting with my *vprof* program described in Chapter 8).

When used with careful attention to time scope, Oracle’s V\$SESSTAT and V\$SESSION_EVENT views provide a high-level perspective of why a user action is taking so long. However, for the next level of your diagnosis, you’ll need to know details that V\$SESSTAT and V\$SESSION_EVENT can’t provide. For example, what if your preliminary analysis indicates that your targeted user action is spending most of its time waiting for the event called `latch free`? Then you’ll wish you had collected data from V\$LATCH (and perhaps V\$LATCH_CHILDREN) for the same time interval. But even if you had, you’ll notice that neither fixed view contains a session ID attribute, so collecting properly action-scoped data about latches on a busy system will be impossible.

The problem of acquiring secondary detail data from V\$ views is an extremely serious one. It’s by no means just a problem with V\$LATCH. What if the dominant consumer of response time had been CPU service? Then you need properly time- and action-scoped data at least from V\$SQL. What if the dominant consumer had been waits for `db file scattered read`? Then you need properly time- and action-scoped data at least from V\$FILESTAT. What if the problem had been waits for buffer busy waits? Then you need V\$WAITSTAT. In Oracle9i there are roughly 300 events that beg for details from any of dozens of V\$ fixed views. Even if you could query from all these V\$ views at exactly the right times to produce accurately time-scoped data, you’d still be left with aggregations whose values fall short of what you could acquire through other means.

Happily, there are at least three ways to acquire the drill-down data you need. The first doesn’t work very well. The second is expensive, but you might already have the capability. The third is available to you for the price of the book that you are holding. The following section explains.

Oracle Diagnostic Data Sources

There are at least three distinct ways to access Oracle's operational timing data:

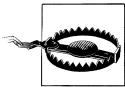
- Querying Oracle fixed views using SQL (fixed views are the views whose names begin with the prefix V\$, GV\$, or X\$).
- Polling Oracle shared memory segments directly to obtain the same V\$ data (that is, accessing the same V\$ data without using SQL).
- Activating Oracle's extended SQL trace facility to emit the complete historical timing activity of an Oracle session to a trace file.

Although V\$ data and extended SQL trace data look like quite different things, it's all the same data, just presented through different user interfaces. In Chapter 7, I describe where the base information comes from.

After devoting three years full-time to studying Method R and its data collection requirements, my personal opinion on the merits of these three approaches is as follows:

Querying V\$ data through SQL

Using SQL to acquire data from V\$ fixed views is an excellent way to compile information about *resource consumption* (that is, to acquire information about how many times various resources have been visited). See Tom Kyte's excellent example at <http://asktom.oracle.com/~tkyte/runstats.html> for more information. V\$ data are especially valuable during application development. Using SQL to acquire *timing* data through the V\$ fixed views, it's easy to get started experimenting with Oracle's operational timing data. But for several reasons listed in Chapter 8, the timing data you will obtain from this data source are unreliable for several problem types. Using SQL to acquire *timing* data from V\$ fixed views provides much less capability than the other two approaches.



One fixed view called X\$TRACE does provide a means to access extended SQL trace data through SQL. However, the X\$TRACE feature is presently undocumented, unsupported, and unstable. If Oracle Corporation fortifies the X\$TRACE facility in the future, it may render obsolete my pessimistic comments about drill-down with fixed view data. But as of Oracle release 9.2, the feature is not ready for production use.

Polling V\$ data directly from Oracle shared memory

If you already own a tool that allows you to properly manipulate the time scope and action scope of your diagnostic data, then high-frequency polling directly from shared memory is probably an excellent approach for you. High-frequency polling gives you diagnostic data that reliably help you solve many types of performance problem. However, attaching to shared memory and then storing gigantic masses of data requires either a lot of study and hard work, or a financial investment in a tool to do it for you. Such tools are expensive.

Activating the extended SQL trace facility

The extended SQL trace facility also offers outstanding diagnostic reliability, but without the research or investment pain that high-frequency polling requires. The principal disadvantage of using extended SQL trace is that you'll be able to collect diagnostic data only for those user actions that you can expect beforehand to exhibit suboptimal performance behavior. This can require some patience when a performance problem occurs only intermittently. With polling, you'll be able to construct properly scoped diagnostic data for any historical user action that you might like to analyze, but *only* if you have invested into enough un-aggregated diagnostic data storage. Extended SQL trace data provides an excellent low-cost substitute for high-frequency polling.

In Table 3-5, I've tried to translate my opinion into a numerical format for your convenience.



This technique of creating the illusion that a man's opinion can be manipulated arithmetically is something I picked up from reading *Car & Driver* magazine.

I believe that extended SQL trace data offers the best performance optimization value of the three diagnostic data sources identified in this chapter. In the past three years, my colleagues and I at *hotsos.com* have helped to diagnose and repair production performance problems in well over 1,000 cases using *only* extended SQL trace data. Our field testing has shown that, when used properly, the extended SQL trace feature is a *stunningly* reliable performance diagnostic tool.

Table 3-5. My opinion on the relative merits of the three Oracle operational timing data sources. Scores range from 1 to 10, with higher scores representing better performance in the named attribute

Attribute	Diagnostic data source		
	V\$ fixed views	Oracle shared memory	Extended SQL trace data
Ease of getting results now	9	1	8
Ease of storing the retrieved data	7	1	10
Ease of parsing the retrieved data	8	1	7
Minimal invasiveness upon Oracle kernel	2	10	7
Minimal invasiveness upon other resources	8	4	7
Capacity for historical drill-down analysis	1	8	7
Cost to develop tools to assist in analysis	9	1	6
Diagnostic reliability	3	9	9
Total	45	35	61

For More Information

Chapters 5 and 6 contain the information you will need to put extended SQL trace data to use, as soon as you're ready for it. Chapter 8 provides some guidance for you in the domain of Oracle's V\$ data sources. I do not discuss in this text how to obtain performance information directly from an Oracle SGA. Very few of the people who have figured out how to map the Oracle SGA are willing to talk about it publicly. Kyle Hailey is one professional who has figured it out and who has been willing to describe the process [Hailey (2002)]. As a technician, I find the subject of direct Oracle SGA memory access to be irresistible. However, as a practitioner and a student of optimization economics, I have found Oracle extended SQL trace data absolutely unbeatable.