Oracle Insights: Tales of the Oak Table

MOGENS NØRGAARD,
JAMES MORLE, DAVE ENSOR, TIM GORMAN,
KYLE HAILEY, ANJO KOLK, JONATHAN LEWIS, CONNOR MCDONALD,
CARY MILLSAP, DAVID RUTHVEN, GAJA KRISHNA VAIDYANATHA

Apress®

Dear Reader,

Of all the heroes I get to meet in and around the OakTable network, Cary is the biggest. He is not "just" a friend of mine; he is a very dear friend of mine. To be in a position where I can introduce him, and his various claims to fame (he would never claim them himself), makes me feel incredibly privileged.

I first noticed his name on a number of (still legendary) papers that he wrote while working inside Oracle. His VLDB paper, his RAID paper (with the footnote about stripping), his Storage paper, and so on and on and on. All excellent papers, all well researched, all attempting to enforce good ideas and kill off bad ones. Oh, and not to be forgotten: the Optimal Flexible Architecture, which has become the standard for installing Oracle anywhere, first presented as a paper at IOUG in Miami in 1991. I was at the conference, but missed the paper. Few probably realized what this OFA idea would lead to.

The former Jonathan Lewis, a.k.a. Dave Ensor (who recently retired again), once remarked to me that "Cary is perhaps the best person Oracle ever had to disprove a theory."

The first time we met was at an EOUG conference in Vienna in 1996. He was in the presenters' lounge, and I went over and introduced myself. He glanced at me, shook my hand, and said some polite, American sounds, and then continued his work.

After perhaps 10 minutes he came over and said "Wait a minute—what did you say your name was?" So I repeated it, and then there was much joy and celebration over the next couple of days, because we had been communicating a lot across that intra-Oracle 2000-member mail list known as HELPKERN, and in the process generating some really interesting mail threads. Now, after several years of that, we met each other. (I also met his beautiful wife Mindy, and I felt envious of Cary for being younger, smarter, richer, more beautiful, and especially for having such a wonder of a wife—and that last part of my envy lasted right up until I met Anette. The rest of them are still with me.)

The next day we both had a seminar before the conference. Cary ran his VLDB seminar, and all of six people or so showed up for it. In the next room I was presenting some performance ragtag stuff or other for 30 or more attendees. Oh, and two of Cary's attendees left him around lunch to join my crowd. I don't think Cary has ever forgotten that experience, but in case he has, it felt good to share it with the rest of the Planet (Cary will usually talk about the Planet instead of the Earth—there's probably a scientific reason for it). Cary's obsession with math and laws of nature are evident in everything he does. In fact, OakTable members rarely talk about exponential curves—we call them Cary Curves. I think if it was legal he would speak in Perl, actually.

Some time later I attended a class of his (VLDB, I think) in Dallas in order to become a certified presenter of it in other parts of the world than Texas. By that time Cary had created the famous System Performance Group and he needed help in other parts of the World (sorry, Planet), because his team of 50–70 people could only just cover the need in the US region. So I volunteered my Premium Services outfit in the middle of one of Cary's sentences, he said yes, and thus we became the third SPG-associated group, along with the huge Mark Large in the UK and the other Mark in Australia. The contacts and knowledge we gained from that are still valuable to this date, and are the basis for several businesses around the World (Planet).

Then came the time when Cary (now a vice president) left Oracle to found Hotsos along with Jeff (also from SPG) and Gary Goodman (formerly of Oracle). One evening Jeff and Cary called me from a client site and needed to know about this strange tracing trick that could show you what the session was waiting for. It took a couple of hours before we got it to work over the phone, and the full credit for this should go to Miracle's own Johannes Djernæs, who finally got it to work for Jeff and Cary.

The rest is history. Many people knew about this way of tracing: Anjo had written the YAPP paper and put it to good use; I had, as usual, just talked for years about it; Kyle had actually used it; and so forth. But Cary? He couldn't just do like everyone else. Oh no. Cary had to create a method, a company, and a tool around it. His and Jeff's book is the only performance book I'll bring to the deserted island. There, in solitude and peace, I might finally find the time to understand that chapter about queuing theory that contains more Greek letters than the story of Odysseus.

Cary has a workshop the size of my garden (everything is big in Texas) where he builds beautiful things in wood (including an Oak Table, of course) and finds excuses for buying big machines that will take your arm off if you're not paying attention. He also used to train with the US Olympic team in Taekwondo, as a black belt. And as a boy he used to kill American box turtles (because they ate fish in the fish pond). The American box turtle is now an endangered species, and I'll leave it up to the reader to make the connection. By pure coincidence I'm currently the proud owner of three American box turtles, but I haven't dared tell Cary where I keep them yet.

Cary introduced me (during a seminar in Denmark many moons ago) to the notion of a mental break. It is not what you think. It is instead an interruption in a technical presentation of something completely unrelated (how insects die, or how Cary's table was constructed) to take your mind off the real topic in order to be able to absorb yet another 42 minutes of hard science. Ladies and gentlemen, dear readers—it is my great honor to present to you the one, the only, my dear friend, Cary.

—Mogens Nørgaard

Extended SQL Trace Data

By Cary Millsap

WAY BACK IN 1992, Oracle introduced a new means of extracting detailed response time information from a database: the extended SQL trace mechanism. In 1995, I began to learn much more about this feature when my colleagues at Oracle Corporation used it to remove the guesswork from Oracle performance improvement projects. The magic of this new extended SQL trace feature was that it allowed analysts to predict the exact response time impact of a proposed system change. At the time, this predictive ability was revolutionary. Predictability and the resulting notion of "fully informed decision making" has been the foundation of my career ever since.

In 1999, I left Oracle Corporation to work full time creating a new performance improvement method that I hoped would far surpass the reliability and accuracy of traditional Oracle methods. In our book, *Optimizing Oracle Performance* (Millsap and Holt, O'Reilly & Associates, 2003), Jeff Holt and I have published the full technical details of our new method and the extended SQL trace tool itself. Here, I describe the story behind Oracle's extended SQL trace capability, its history, the people who helped develop it, and how and why you might consider using it.

Before the Microscope

In the days before the microscope, the fight against infectious diseases was brutally difficult. In the fourteenth century, for example, bubonic plague ravaged Europe and Asia. Between 1347 and 1352, the Black Death (as the disease came to be called) killed 25 million Europeans—one third of the region's total population.[1]

---

1. Source: `http://www.byu.edu/ipt/projects/middleages/LifeTimes/Plague.html`.

Medieval medicine provided no cure for the plague, but men and women of the time tried everything they could think of to combat the disease. The dominant response to plague symptoms was bloodletting practiced with extraordinary aggressiveness. When a patient lost consciousness from too much blood loss, the "caregiver" would revive the patient with cold water. When the patient reawakened, the bleeding would continue.

The list of responses to plague symptoms would sound like comedy if not for the magnitude of the suffering that motivated them. Wives nudged their husbands in their sleep, prompting a roll from one side of the body to the other to "prevent the liver from heating too much on one side." Scents were big. People would inhale virtually anything they could think of that might have a chance of "replacing" the plague-bearing air in their bodies with plague-free air, from aromatic spices to their own sewage pits. Neighbors exterminated cats and those cats' masters on the premise that perhaps it was witchcraft at the root of the epidemic. There wasn't much that a medieval neighbor wouldn't try, if he thought he could save himself or his family from the horrors he had seen befall other families.

Medieval medicine failed to stop the plague because medieval medicine failed to identify the plague's root cause. The root cause, we know today, was a bacterium called *Yersinia pestis*, which was transported and inoculated into humans with exceptional efficiency by fleas (… turns out that cat killing probably caused positive harm). Medieval "doctors" didn't know that *Y. pestis* existed. They didn't even know that bacteria existed—they couldn't *see* them. You can't see plague bacteria without a microscope, and the microscope wouldn't be invented until a Dutchman by the name of Anton van Leeuwenhoek did the job late in the seventeenth century.

The history of Oracle performance analysis looks a lot like the history of humanity's war on infectious disease. The nouns are different, but the story is familiar. Instead of people, it's Oracle systems that die before their time. Instead of leeches, cat killing, and vapor wafting, it's hit ratios, database rebuilds, and disk rebalancing. Instead of *Yersinia pestis*, it's `latch free`, `buffer busy waits`, `SQL*Net message from client`, and the hundreds of other code paths in which an Oracle system can consume response time. And instead of the microscope, it's *extended SQL trace data.*

Our story even has its own Dutch guy. We will get to that in a little bit.

The Dark Ages

In the days prior to Oracle version 7, Oracle professionals taught only one way to "tune Oracle." That way was to look at a variety of data sources, usually beginning with the output of a program pair called `bstat.sql` and `estat.sql`, and including anything else you could find, such as output from `sar`, `vmstat`, `glance`, `top`, and whatever other operating system tools you could get your hands on.

The game was to look at the output from all these sources and construct a theory that could explain the root cause of a given performance problem. You would address that root cause and then retry the slow program, in the hope that it would be sufficiently fast now that you could declare success, and move on to the next item of business. Often, however, you'd find that the thing you fixed had no bearing on the problem at hand, or worse, perhaps you'd find that your attempted repair had exacerbated the problem. If this happened, then you could do a good job only if you had kept good enough notes to allow you to undo your change, so your system would be at least no worse off for your efforts.

This was "tuning." It was a process akin to forensic pathology: like detectives who searched for blood droplets and clothing fibers, you'd search for numbers or combinations of numbers that looked unusual. The number that always looked unusual was the famous *database buffer cache hit ratio*. The database buffer cache hit ratio was a measure of what proportion of database buffer accesses could be serviced without the Oracle kernel having to issue an operating system read call. The formula with which people computed the cache hit ratio (chr) was

$$chr = \frac{requests - reads}{requests} = 1 - \frac{reads}{requests}$$

No matter what its value was, the number always looked suspicious. For example, a value of 0.1 would have been considered unconscionably low. Nobody wanted to think that 90% of the Oracle blocks used by his system involved the participation of the disk subsystem.[2] Of course, when your database buffer cache hit ratio is too low, the prescribed response was to increase the size of the database buffer cache, which in those days meant incrementing the value of the db_block_buffers Oracle instance parameter.

But low cache hit ratios weren't the only problem. Slow systems could have high cache hit ratios, too. Some people knew why, but most didn't seem to. Most people assumed that if a system had a high buffer cache hit ratio and the system was still slow, then the buffer cache hit ratio simply must not be high *enough*. So, even people with cache hit ratios in excess of 0.99 would presume that the way to remedy their problem was, of course, to increase the size of the database buffer cache.

---

2. Of course, this isn't exactly what a cache hit ratio meant, but it's what a lot of people were *taught* that it meant. The first important distinction between this statement and actual fact is that many systems have an OS buffer cache between the Oracle kernel and the disk subsystem, so many I/O calls that seem, from the Oracle kernel's perspective, to be "physical" are in fact serviced from memory.

The problem is that this technique often didn't work. Increasing the size of the database buffer cache made a system faster if its buffer cache was too small to begin with. But on many systems, increasing the size of the buffer cache didn't produce any improvement in end-user performance, even if it did happen to improve the database buffer cache hit ratio.

Some people knew better. In the early 1990s, two of my Oracle mentors, Willis Ranney and Dave Ensor, taught me that a really high database buffer cache hit ratio value—a value, say, in excess of 0.95—was usually an indication of a serious performance problem that demanded attention. The argument went like this . . .

Imagine that a given result set could be produced by any of three different methods: A, B, and C. These methods use different join orders and access methods, but they all return the same answer. Furthermore, imagine that use of these methods produced the operational statistics shown in Table 5-1. Which method presents the best way to produce the result set?

Table 5-1. Database Buffer Cache Hit Ratios for Various Numbers of Requests and Reads

Method	Database Buffer Cache Hit Ratio
A	0.7
B	0.998
C	0.99999

The naïve answer is that method C is the best, for the simple reason that it produces the largest buffer cache hit ratio of the three methods. But using only the information shown in Table 5-1, it is impossible to determine which method is the best. Table 5-2 shows why.

Table 5-2. Database Buffer Cache Hit Ratios for Various Numbers of Requests and Reads

Method	Requests	Reads	Database Buffer Cache Hit Ratio	Response Time (Seconds)
A	10	3	0.7	0.031
B	1,000	2	0.998	0.070
C	100,000	1	0.99999	5.010

The far superior answer is that method A is the best, for a couple of reasons. First, method A produces the best *response time*, which is a metric that is very well connected to a user's perception of how the system is performing. Second, method A does far less work than either of the other two methods. That "requests" count means something. Each time an Oracle kernel process requests the access of a block from the database buffer cache, the process executes potentially thousands of CPU instructions, many of which are serialized.

The argument continued that if a system showed a database buffer cache hit ratio value, say, in excess of 0.95, it was a pretty good indication that there were one or more SQL statements resembling method C that should be converted to method A. Doing this would have two effects: it would make the system perform better, and it would reduce the value of the buffer cache hit ratio.

This argument was, at the time, revolutionary, and history has borne out that the argument was entirely accurate.[3] The argument caused a horrible problem, though: The principal system metric that Oracle professionals used to measure the performance of an Oracle system didn't work. "Low" numbers were supposed to be bad, and "high" numbers were supposed to be good, but "low" numbers weren't always bad, and "high" numbers weren't always good. Worse yet, the values of "low" and "high" were completely subjective.

It was a mess. And the problem was even worse than I've let on. It wasn't just the database buffer cache hit ratio metric that was flawed; almost *every* system metric we had was flawed in a similar manner. Is a latch miss ratio of 95% a problem? It might be, but if a latch has been acquired only 20 times in the past two months, and the 19 misses have cost a grand total of 2.718 seconds of end-user response time over that period, then who cares? Is CPU utilization of 90% a problem? It might be, but if there's batch workload queued up for processing, and response time fluctuations aren't bothering the users, then letting a CPU sit idle would be a waste of good capacity. Similar problems existed for every single metric we looked at.

The only solution we had was to look at larger, more elaborate collections of statistics. Most of us tried to do our jobs by looking at dozens of pages, each containing hundreds of operational statistics. We'd try to single out our favorites. If one time we had made a system go faster by noticing too "high" a ratio of `table scan blocks gotten` to `logons cumulative`, we would check that ratio every time we analyzed a new system. If one time we had discovered that creating a certain index made things faster, we'd check for the presence of that index.

Basically, we would fix the things we knew how to fix, and then we'd pray that what we had done would somehow cause the things people were complaining about to run faster. It was akin to looking in the best-lit spot in the parking lot for the car keys you lost, regardless of where you dropped them. The "method" we were taught for "tuning" Oracle databases worked only when the problem was something that we knew in advance how to fix.

---

3. One of the reasons that the many people were slow to realize the truth of this argument on their own was the widespread and nearly debilitating myth that reading a block from the Oracle database buffer cache is "10,000 times faster" than reading a block from a disk. The correct factor is on the order of 10 or 100, not 10,000 (Millsap, "Why You Should Focus on LIOs Instead of PIOs," Hotsos, 2001).

The Enlightenment

The beginning of the enlightenment for me came when I was in charge of Oracle Corporation's System Performance Group in 1995. One of the more pleasant aspects of my work in those days, as it still is today, was keeping in contact with the leader of the Premium Support Services group of Oracle Denmark, one Mr. Mogens Nørgaard.[4] On one particular day, my dear friend[5] was telling me the news of a new thing in Oracle that people were calling the *wait interface*.[6]

Mogens explained to me on the telephone that this wait interface thing was a lot like watching someone buy groceries. Instead of guessing how long someone had spent in the store by investigating the contents of her shopping cart, the wait interface was like a timer that could tell you exactly how long the shopper had spent choosing potatoes, how long she had spent choosing orange juice, and so on. A mutual friend named Anjo Kolk, a Dutchman, also from Oracle Corporation, had written a very nice paper about it called "Description of Oracle7 wait events and enqueues." Mogens would of course send me a copy, and I of course would read it.

I was interested because my friend seemed to think it was important, but, frankly, I thought the news wasn't as interesting as Mogens seemed to think it was. I certainly wouldn't have guessed during that conversation that in the year 2000, I would give the next four years of my career over to the thing he was describing.

At about the same time, a young consultant in my group named Virag Saksena had begun sending client engagement reports that represented a *huge* advance in optimization effectiveness. Virag had always impressed me with his effectiveness as an Oracle optimizer. He had already established himself with me as one of the most efficient SQL query optimizers on the planet. But in 1995, Virag was doing work that took Oracle "tuning" to a completely different plane.

---

4. It looks like the name Mogens would be pronounced \mō´g∂ns\, but it's not. The Danes swallow their *g*'s, so the name is pronounced \mōns\. That is, *Mogens* and *groans* rhyme perfectly.

5. Mogens is now not just my "dear friend," but my "award-winning dear friend" as well. Mogens has kidded me relentlessly about Americans' propensity to give so many awards to each other that everyone can advertise that they are "award-winning." Since *Oracle Magazine* named Mogens "2003 Oracle Educator of the Year" he doesn't kid me so much about it anymore.

6. Happily, as of version 10, Oracle Corporation no longer calls this feature the *wait interface*, but rather the *timed event* interface. This is a much more appropriate name for the feature. The word *wait* already has a specific technical definition in the field of queueing theory, and the "wait" statistics that the Oracle kernel produces are definitely not *waits* in the queueing theory sense (Millsap and Holt, *Optimizing Oracle Performance*, O'Reilly & Associates, 2003. pp. 239–242).

In the early 1990s, I fancied myself a decent Oracle "tuning expert." I did what most "tuning experts" in those days did. It's only slightly unfair to say that what we did was to follow these steps:

1. Listen politely as the client explained what the problem was. Take notes to prove that we're listening carefully.

2. Fix the ten or so problems that we knew how to fix.

3. Pray that the ten things we had "fixed" would provide some relief for the client's most important complaints.

4. Document the miracles we had performed, with *before* and *after* statistics, explain what changes we had made, and create a list of open issues that would have to be resolved later.

5. Send the invoice.

It usually took me three or four days to get through step 2, and then I would spend the remainder of the week praying and explaining. As I said, when I was doing this for a living, I had thought I was pretty good.

I wasn't.

What Virag was doing was *completely* different. He was listening politely, and then he was focusing his efforts on the client's most important complaints. He was making statements like this:

If you do x, *then you will reduce the response time of the thing you're complaining about by* y *seconds. Doing* x *will cost approximately* z.

This was completely revolutionary. He was providing exactly the information that a decision-maker would need to make a fully informed economic decision about whether to do x. Here was, for the first time in my career, a consultant who was informing his client about both the costs *and benefits* to be expected from an action. And he wasn't guessing. He was creating a spectacular track record of accurate predictions. When people did x, response times really did drop by y seconds, just like he'd said.

Before Virag, I had seen consultants specify only costs (which consultants are well-trained at estimating) and *guesses* about benefits:

Do x, *and we think it will help, but we can't really guarantee anything.*[7] *Doing* x *will cost approximately* z.

---

7. Of course, a consultant spends a lot of time figuring out ways to express such disappointments as this without sacrificing one's professional dignity.

To make Virag's predictions even more remarkable to me, he was making them on his *first day of a project*. Let me recap: I had thought I was pretty good, and I was often guessing even on Friday. Virag was making accurate and very detailed cost/benefit predictions about performance on Monday. I was blown away. I had seen the future.

Virag's extraordinary efficiency was an inspiration that would help drive my research into what I, and everyone else, might be able to extract from the Oracle extended SQL trace data that he was using.

The History of SQL Trace

Let me switch gears for a while and tell the story of Oracle's SQL trace feature, which would eventually become the *extended* SQL trace feature used today.

The Oracle extended SQL trace feature exists today because, fortunately, Oracle customers aren't the only ones who have to wrestle with Oracle application performance problems. The men and women who built the Oracle kernel fight the same problems too, both in customer situations and in competitive benchmarks. The following quote is from Juan Loaiza, an Oracle kernel architect, and at the time of this writing a vice president at Oracle Corporation (you can read the full story in Chapter 2, where Juan tells of the motives for inventing SQL trace and then extending it to the state in which we know it today):

> *I've always thought that diagnosing performance issues boils down to figuring out where the time is going in the system. If you can attribute the time correctly, then the source of the problem becomes obvious.*

Certainly our professional community is indebted to Juan Loaiza and his team for giving us the microscope to see how the Oracle kernel is spending *all* of our users' time.

Version 5

SQL trace is at least as old as Oracle version 5, which was released in 1986. The syntax to turn it on and off was simple:

```
select trace('sql',1) from dual
...
select trace('sql',0) from dual
```

Apparently, not too many people knew about SQL trace in Oracle version 5, and probably fewer than that actually used it. In an old Oracle internal document called "Optimizing," Oracle described the trace function as follows (quoted verbatim from the source):

> *The Kernel provides a trace function to provide information about internal operations.*
>
> *The trace function is essentially a debug aid for Oracle Software Development.*
>
> *—It is **not documented**.*
>
> *—It is **not supported**.*
>
> *—[It] will not be a function in ORACLE version 6.*

By today's standards, the version 5 SQL trace function didn't do a whole lot. All you could get from the 'sql' trace output was a stream of PARSING IN CURSOR sections that looked something like this:

```
====================
PARSING IN CURSOR 3:
   "select tab$pid,tab$rba,tab$tbl,tab$type,tab$sowner,tab$sname"
   " from sys.tables where tab$owner=:1 and tab$name=:2"
====================
PARSING IN CURSOR 4:
   "select idx$cky from indexes['1.f.1'] where idx$tbl=:1 and id"
   "x$cky is not null"
====================
PARSING IN CURSOR 1:
   "select * from dept "
====================
```

There were other trace functions besides 'sql'. With the trace function, an Oracle user could see information about access paths, sorting operations, and memory consumption.

Version 6

In Oracle version 6, SQL tracing became a core, documented feature for everyone to use. In version 6, Oracle introduced the syntax that also works in versions 7, 8, 9, and 10:

```
alter session set sql_trace=true
...
alter session set sql_trace=false
```

SQL trace was a significant step forward in power for the Oracle performance analyst. It allowed you to see the sequence of SQL statements that an application was executing, and it allowed you to see how much load each of these statements placed upon the database. Raw trace data was ugly, as you can see from Listing 5-1.

Listing 5-1. An Excerpt from Raw SQL Trace Data

```
=====================

PARSING IN CURSOR #1 len=103 dep=0 uid=5 oct=3 lid=5 tim=564568615 hv=3329006097 ad
='54a12578'
select count(*)
from (select n from dummy connect by n > prior n start with n=1)
where rownum < 100000
END OF STMT
PARSE #1:c=4,e=18,p=2,cr=34,cu=1,mis=1,r=0,dep=0,og=3,tim=564568615
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=3,tim=564568615
FETCH #1:c=381,e=393,p=0,cr=100000,cu=0,mis=0,r=1,dep=0,og=3,tim=564569008
FETCH #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=0,tim=564569008
```

However, it didn't matter too much because Oracle Corporation provided a tool called `tkprof` that formatted raw trace data into a more convenient form for people to use (Listing 5-2[8]).

Listing 5-2. An Excerpt from `tkprof` *Output*

```
select count(*)
from (select n from dummy connect by n > prior n start with n=1)
where rownum < 100000
```

---

8. Listing 5-2 shows the form of modern `tkprof` output (the case shown here was generated using Oracle version 8.1.6.1.0). The format of `tkprof` output has changed a little bit since version 6, but the content remains materially the same as it was in the version 6 days.

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	3.81	3.93	0	100000	0	1
total	4	3.82	3.93	0	100000	0	1

```
Misses in library cache during parse: 1
Optimizer goal: RULE
Parsing user id: 5
```

In addition to the standard ratio-based stuff, the better Oracle "tuning" courses taught students how to interpret tkprof output. The rules for using tkprof output were pretty simple:

1. Reduce the numbers that appear in the count column. That is, eliminate all the database calls you could. The most powerful way to accomplish this was by manipulating whatever application code was making database calls with SQL.

2. Reduce the numbers that appear in the query and current columns (derived from the cr and cu fields in the raw trace data, respectively). That is, eliminate all the requests for blocks from the buffer cache that you could. Sometimes doing this was as simple as creating or dropping indexes. Sometimes you had to manipulate the text of the SQL itself to accomplish this.

3. If the query + current sum was at its bare minimum, and disk was nonzero, then consider using a bigger database buffer cache. That is, eliminate all the "physical" read requests that you could, but only after eliminating all of the memory accesses that you could.

Of course, these guidelines evolved over time as well. At the first Oracle "tuning" course I ever attended, our instructor taught that the way to tune SQL with tkprof is to do whatever it took to make the disk column's value be zero. We now know that the problem with this strategy is that it helps to promote a very serious type of performance problem: a SQL statement can work entirely within the confines of the database buffer cache and still be horrendously inefficient. Tables 6-1 and 6-2 show how. However, this advice was the "golden rule" of internal Oracle training courses at the time of Oracle version 6.

When bad SQL or bad schema design, missing indexes, too much parsing, or under-use of Oracle's set processing capabilities were the problem, SQL trace data

was the microscope you needed to find the problem. But for some problems, SQL trace data wasn't enough. Standard SQL trace data is inadequate when the elapsed time of a SQL statement greatly exceeds the amount of CPU capacity the statement has consumed. For example, what do you do when you see this in your `tkprof` output?

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.02	0	0	0	0
Execute	1	0.03	0.03	0	0	0	0
Fetch	20	0.13	22.95	11	39	0	20
total	22	0.17	23.00	11	39	0	20

This table says that the query consumed 23.00 seconds of response time, but it *explains* only 0.17 seconds of that time. What are you supposed to believe about the other 22.83 seconds? The answer, when you're using standard SQL trace, is that *you just don't know.*

Of course, not knowing doesn't keep people from guessing. The most popular guess was the usual suspect of all performance problems throughout the late twentieth century: disk I/O. This guess was certainly right some of the time, but it was wrong some of the time, too. The problem is that when your problem is *not* CPU consumption and it's *not* disk I/O, what then?

Listing 5-3 shows an example of the problem. In this example, the statement in question is a simple UPDATE statement that updates the value associated with a specified primary key value. The update manipulated only one row (r=1), yet the execution phase of the command consumed 10.56 seconds of elapsed time (e=1056). What took so long? It certainly wasn't CPU service, because the statement consumed only about 0.02 seconds of CPU time (c=2). The "it must be disk I/O" argument doesn't hold much weight in this case, because the raw trace data indicates that the kernel issued no OS read calls during the update (p=0). How can you find why this update consumed more than 10 seconds?

Listing 5-3. Raw Level-1 Trace Data Showing That Elapsed Time Greatly Exceeding the CPU Service Duration

```
======================
PARSING IN CURSOR #1 len=31 dep=0 uid=73 oct=6 lid=73 tim=27139911 hv=1169598682 ad
='68f56dc8'
update c set v1='y' where key=1
END OF STMT
PARSE #1:c=0,e=902,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=4,tim=27139911
EXEC #1:c=2,e=1056,p=0,cr=3,cu=3,mis=0,r=1,dep=0,og=4,tim=27140969
```

Version 7

Oracle version 7 brought the solution to the problem of what to do when the value of e for a database call is much greater than the value of c. In version 7, Oracle introduced the feature that is the subject of this chapter: *extended* SQL trace. With extended SQL trace enabled, the Oracle kernel can tell you *everywhere* it spends your time. The prevailing hypothesis of the version 6 era would have been that the time was lost in disk I/O. However, Listing 5-4 shows exactly where that missing time from Listing 5-3 actually went: the Oracle kernel waited four times—for a grand total of approximately 10 seconds—on Oracle enqueue events. In other words, the update was blocked on the acquisition of a lock.

Listing 5-4. Raw Level-8 Trace Data Shows Where the Time Went

```
=====================
PARSING IN CURSOR #1 len=31 dep=0 uid=73 oct=6 lid=73 tim=27139911 hv=1169598682 ad
='68f56dc8'
update c set v1='y' where key=1
END OF STMT
PARSE #1:c=0,e=902,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=4,tim=27139911
WAIT #1: nam='enqueue' ela= 307 p1=1415053318 p2=393254 p3=35925
WAIT #1: nam='enqueue' ela= 307 p1=1415053318 p2=393254 p3=35925
WAIT #1: nam='enqueue' ela= 307 p1=1415053318 p2=393254 p3=35925
WAIT #1: nam='enqueue' ela= 132 p1=1415053318 p2=393254 p3=35925
EXEC #1:c=2,e=1056,p=0,cr=3,cu=3,mis=0,r=1,dep=0,og=4,tim=27140969
WAIT #1: nam='SQL*Net message to client' ela= 0 p1=1413697536 p2=1 p3=0
```

Do you think you would have guessed that? You might have. After all, this was an UPDATE statement, the kind of thing that might block on a lock. But it could very well have been something else. In Oracle version 7.3.4, it could have been any of 106 things, to be exact, because in version 7.3.4, Oracle had instrumented 106 distinct code paths in the Oracle kernel.

So, maybe you would have guessed that the dominant contributor of the update's response time was time spent blocked on a lock, but there would be about a 1% chance you'd have been right. But even if you had guessed that the problem was a lock, you couldn't have proven it. How much would you have been willing to wager? A few thousand dollars of your company's budget? Your career? And could you have guessed the lock type and lock mode? The trace data contains this information, too. It was an EXCLUSIVE mode TX lock, which you can determine by properly interpreting the value p1=1415053318, using Oracle Metalink documents such as 55541.999 and 62354.1 for guidance.

The Oracle version 7 syntax for activating extended SQL trace is, again, the
ALTER SESSION statement:

```
alter session set events '10046 trace name context forever, level 12'
...
alter session set events '10046 trace name context off'
```

The level number for the extended SQL tracing event (event number 10046) is
a binary encoding of three values, as shown in Table 5-3. You can combine the lev-
els to combine effects, so using level 12 creates the effect of using both level 4 and
level 8 at the same time. Curiously, manipulating the decimal 2-bit doesn't appear
to have any effect.

Table 5-3. Extended SQL Tracing Levels

Binary Digit	Decimal Value	Description
0000	0	No tracing
0001	1	Standard SQL tracing
0100	4	Standard SQL tracing plus bind variable information
1000	8	Standard SQL tracing plus timed event information
1100	12	Standard SQL tracing plus bind variable and timed event information

Using an ALTER SESSION command is well and good if you have read and write
access to the application code whose performance you're trying to diagnose. But
what happens if you're trying to diagnose an application that you've purchased
from a software vendor? If the application vendor wasn't thoughtful enough to
give you the option to trace the application code (maybe a menu option), then
what do you do?

One answer is to activate tracing with an ALTER SYSTEM command. However,
this approach can cause more problems than it solves. Tracing a whole system
can fill disks very rapidly, making it difficult to identify which trace data you want
to study and which you want to discard.

A standard Oracle package, DBMS_SYSTEM, provides a much more elegant
answer. The procedure DBMS_SYSTEM.SET_EV allows an Oracle session to activate
extended SQL tracing in another Oracle session. The syntax is

```
dbms_system.set_ev(sid, serial, 10046, 12, ")
...
dbms_system.set_ev(sid, serial, 10046, 0, ")
```

With `SET_EV`, you could trace *any* Oracle session on your system. However, there was a drawback to using `SET_EV`: the procedure was (and still is) officially unsupported. This drawback was more of a political issue than a technical one. The justification for not supporting `SET_EV` was entirely rational. First, you probably don't want to grant the execute privilege on `DBMS_SYSTEM` to just anyone; there's stuff in there that you don't want most of your users being able to run. Second, misuse of `SET_EV` can cause serious harm to your system. For example, typing the event number `10004` by accident instead of `10046` could cause application of the Oracle kernel event that will simulate a control file crash . . . probably not what you wanted to do.

I'm not much of a rule breaker in real life, but the rule not to use `SET_EV` is, in my opinion, a rule truly worth breaking. For many, the feature is just too valuable not to use. There seem to be at least as many Oracle Metalink articles encouraging the use of extended SQL trace (see especially note 39817.1) as there are articles discouraging it. It appears that extended SQL trace is well supported, just not the means for turning it on.

Version 8

In Oracle version 8, Oracle Corporation supported the extended SQL trace slightly more, with the distribution of the `/rdbms/admin/dbmssupp.sql` script. This script creates the package called `DBMS_SUPPORT`, which insulates users from the dreaded effects of misusing `SET_EV`. The `DBMS_SUPPORT` package makes it very easy to activate extended SQL trace for a given session:

```
dbms_support.start_trace_in_session(sid, serial, true, true)
...
dbms_support.stop_trace_in_session(sid, serial)
```

An inconvenient aspect of `DBMS_SUPPORT` is the following note in the file `dbmssupp.sql`:

```
Rem    NOTES
Rem     This package should only be installed when requested by Oracle
Rem     Support. It is not documented in the server documentation.
Rem     It is to be used only as directed by Oracle Support.
```

Alas, still unsupported; however, Metalink note 62294.1 definitely provides encouragement to users of `DBMS_SUPPORT`.

Aside from the improved access mechanism, extended SQL trace data didn't change much in version 8. It really didn't need to, because it worked so well in version 7. The number of wait events did grow from 106 in Oracle version 7 to 215 in Oracle version 8.

A couple of Oracle bugs shook some people's confidence in SQL trace from time to time. Bug 1210242 affected cursor sharing when tracing was activated. Bug 2425312 caused the Oracle kernel to neglect to emit important information about remote procedure calls into the trace output. But with the appropriate patches installed, SQL trace in Oracle version 8 worked accurately and reliably.

Version 9

Oracle version 9 brought the first major changes to trace data that had occurred in a long time. Changes included the following:

- *Improvement of timing statistics resolution:* Oracle improved its output timer resolution from 1-centisecond (0.01-second) units to 1-microsecond (0.000001-second) units. With the finer resolution, we were able to learn even more about short-duration database calls and wait events.[9]

- *Option to include wait event information in the* tkprof *report:* Finally, tkprof began processing the extended SQL trace information. Not too bad for an officially unsupported feature.

- *Even more instrumented wait events:* The Oracle kernel developers instrumented many more code paths in the kernel, as the wait event count rose to 399 in version 9.2.0.4.

- *Inclusion of row-source workload and timing statistics:* This helped analysts determine how much time each row-source operation in a statement's execution plan was responsible for contributing to total response time for the statement.

---

9. It probably sped up the Oracle kernel a little bit, too. Most underlying operating systems had long since provided microsecond timing data to the Oracle kernel, but only in version 9 did the Oracle kernel cease integer-dividing the microsecond part of the OS timing statistics by 10,000 to convert the data from microseconds to centiseconds.

The inclusion of row-source statistics in version 9.2.0.2 created a bit of a brief plunge into darkness for SQL trace users. Since version 6, Oracle has emitted one or more lines beginning with the token STAT in the trace data when a cursor with an execution plan closes. You can easily assemble these lines into an execution plan, as shown in Listings 5-5 and 5-6.

Listing 5-5. STAT *Lines Show Statistics About a Cursor's Row-Source Operations*

```
STAT #3 id=1 cnt=0 pid=0 pos=0 obj=0 op='FILTER '
STAT #3 id=2 cnt=1068 pid=1 pos=1 obj=16351 op='TABLE ACCESS FULL ACCOUNT '
STAT #3 id=3 cnt=1067 pid=1 pos=2 obj=0 op='CONCATENATION '
STAT #3 id=4 cnt=1067 pid=3 pos=1 obj=19512 op='TABLE ACCESS BY INDEX ROWID
 CUSTHASPRODUCT '
STAT #3 id=5 cnt=213128430 pid=4 pos=1 obj=22903 op='INDEX RANGE SCAN '
STAT #3 id=6 cnt=1067 pid=3 pos=2 obj=19512 op='TABLE ACCESS BY INDEX ROWID
 CUSTHASPRODUCT '
STAT #3 id=7 cnt=1160025328 pid=6 pos=1 obj=22903 op='INDEX RANGE SCAN '
```

Listing 5-6. The Execution Plan Denoted by the STAT *Lines in Listing 5-5*

```
Rows returned  Row-source operation (object id)
-------------  ------------------------------------------------------
            0  FILTER
        1,068    TABLE ACCESS FULL ACCOUNT (16351)
        1,067    CONCATENATION
        1,067      TABLE ACCESS BY INDEX ROWID CUSTHASPRODUCT (19512)
  213,128,430        INDEX RANGE SCAN (22903)
        1,067      TABLE ACCESS BY INDEX ROWID CUSTHASPRODUCT (19512)
1,160,025,328        INDEX RANGE SCAN (22903)
```

These STAT lines are enormously helpful to you, the performance analyst, because they tell you (a) what steps the Oracle kernel has used to access your data and (b) how much work each step has done. However, the study of performance is the study of *time*, and you cannot determine how long something took by counting how many times it happened. While it is nice to see how much work is being done by each row-source operation, it would be nicer to see how much *time* each row-source operation consumed.

This is the new feature that Oracle introduced in version 9.2.0.2. A new STAT line looks like this:

```
STAT #1 id=5 cnt=23607 pid=4 pos=1 obj=0 op='NESTED LOOPS  (cr=1750 r=156 w=0 time=
1900310 us)'
```

Notice that the row-source operation name has been augmented with a parenthetical list of new statistics (compare this sample to the lines in Listing 5-5). The `time=1900310 us` field in this example tells you that this `NESTED LOOPS` row-source operation consumed 1.900310 seconds of wall time.

Time consumption by row-source operation is wonderful information to have. The problem was that the means by which the Oracle kernel obtained this information in the first release of the feature was so resource intensive that using SQL trace (either standard or extended) became unbearably expensive in many cases. Using SQL trace in Oracle versions 9.2.0.2 through 9.2.0.4 caused some SQL statements to consume five or more times the response time they would have consumed by using SQL trace in earlier versions of Oracle.

This is Oracle bug 3009359. Fortunately, Oracle Corporation responded rapidly to reports of this problem with a patch present in 9.2.0.5. The world of SQL trace was rocked only briefly before order was restored. With the patch, extended SQL trace continued to provide faithful and valuable service.

Version 10

There's a big functional problem with the whole version 6/7/8/9 tracing model. In the version 5 and 6 days, when the architects at Oracle designed the SQL tracing model, there was really only one kind of Oracle application: client-server. In the Oracle client-server model, an Oracle user application process connected to one and only Oracle server *session*, which was executed within one and only one Oracle server *process*, resulting in one and only one trace file. For this model, the design of SQL tracing made perfect sense. If you wanted to trace user Nancy's functional user action, you would activate tracing for the one and only one Oracle session to which Nancy's program connected. You would find the trace file and analyze it. The world was good.

However, the Oracle application architecture model has diverged significantly since the 1980s. Oracle's *Multi-Threaded Server* (MTS) feature[10] was one of the first complications to the tracing model, because this feature allowed a single user action to engage the services of potentially many Oracle server processes. With MTS, the trace of a single user action commonly results in the creation of two or more trace files whose contents have to be knitted together either by hand or with some kind of custom software tool. With MTS, SQL trace is still a perfectly usable feature; you just have to do a little more work to piece all the data together correctly.

Oracle's *parallel execution* (PX) features provide another complication because, again, a single user action will engage the services of several Oracle

---

10. Which, incidentally, has an odd name because it's not multithreaded, as Dave Ensor and James Morle explain in Chapters 1 and 8, respectively.

server processes. The trace of a parallel degree *n* user action will result in the creation of one trace file in the user dump destination directory, and up to 2*n* more trace files in the background dump destination directory. Again, SQL tracing (both standard and extended) still works great for PX operations, but it takes a good bit more work to piece everything together because the content of these files has to be knitted together by hand or with some kind of custom software tool.

The advent of multiplexing environments takes the problem to a whole new level. Multi-tier computing has evolved to accommodate the scalability requirements of today's online applications to which literally *millions* of users might connect. Today, it is possible for a single functional user action to speckle evidence of its database workload across dozens of trace files, distributed across dozens of computers. The biggest problem is that, because you can control tracing only at the Oracle *session*, when you try to trace Nancy's workload, you necessarily trace other users' workload as well—for all the users who are sharing Nancy's Oracle session (or perhaps even *sessions*, plural). How in the world are you going to trace a user action, then, without getting either a whole lot more trace data or a whole lot less than you really want?

The Oracle version 10[11] *end-to-end tracing* model promises to solve this difficult technical problem, and tracing's nagging political problem to boot. That's right, in version 10, the extended SQL trace function is fully documented and fully supported (and the event count is up to 808 in version 10.1.0.2.). Furthermore, in version 10, "is tracing" becomes an attribute of a specified *functional* unit of work. The package DBMS_MONITOR provides the syntax for activating and deactivating the trace:

```
dbms_monitor.serv_mod_act_trace_enable(service, module, action, true, true)
...
dbms_monitor.serv_mod_act_trace_disable(service, module, action)
```

With version 10, Oracle provides a new tool called trcsess, which does the work of knitting trace files together for you, so that you can see a linear sequential record of the work done by an individual user action.

Of course, it is the application's responsibility to identify its functional units of work in such a manner that DBMS_MONITOR can find them. In the old days, good application developers used the set methods in DBMS_APPLICATION_INFO to identify the diagnosable components of their applications. The problem with DBMS_APPLICATION_INFO is that, for example, setting an application's MODULE and ACTION attributes requires the execution of two stored procedure calls. Calling these procedures integrates extra database call overhead into the application.

---

11. Officially, Oracle version 10 is called *Oracle Database 10g*, but I'll refer to it as *version 10* in the same spirit in which I operate when I don't call Oracle version 6 *ORACLE V6*.

Oracle version 10 provides changes to the *Oracle Call Interface* (OCI) that allow an application developer to piggyback an application's identifying information onto database calls that the application already makes.[12]

If it all works correctly, the end-to-end tracing model of version 10 is exactly what we have been waiting for: the ability to answer Nancy's question, "What took so long?" for any application's functional unit of work running on a system of any arbitrary architectural complexity.[13]

The Revolution

The lineage of work beginning with Juan Loaiza and continuing through Anjo Kolk, Mogens Nørgaard, and Virag Saksena has had a profound influence upon my career and the careers of many others. In late 1999 when Gary Goodman and I founded this company called Hotsos that now feeds our families, we created a business that would set out to revolutionize the practice of Oracle performance problem diagnosis. The problem we saw was that the world's approach to Oracle performance optimization required a lot of experience, intuition, and luck to be successful. Of course, any process that requires experience to succeed is expensive, and any process that requires intuition and luck to succeed is impossible to reproduce and, of course, just as difficult to teach. We had seen hundreds of good people lead failed performance improvement projects in the 1990s.

What we sought was a performance optimization method for Oracle systems that produced economic value by being quicker, more reliable, and—perhaps most difficult of all—*teachable*. We found the most important attribute of such a method to be *determinism*. We wanted to create a method that began with the same step every time, and that progressed reliably using exactly the same decision-making criteria through every step, completely irrespective of the experience level or intuitive development of the practitioner. For a method to be a success, any two people from any two cultural backgrounds must be able to achieve the same desired end result with the properly executed method. Guesswork is out of bounds.

Jeff Holt and I worked full time on this assignment as a technical problem for more than a year. We spent an enormous amount of time pursuing the acquisition of performance diagnostic data from traditional sources inside the database, such as the `V$SESSTAT` and `V$SESSION_WAIT` fixed views. The problems with using data

---

12. See, for example, the descriptions of the attributes `OCI_ATTR_MODULE`, `OCI_ATTR_CLIENT_INFO`, and `OCI_ATTR_ACTION` in the "User Session Handle Attributes" section in Appendix A of the *Oracle Call Interface Programmer's Guide* for Oracle Database 10*g* Release 1.

13. At the time of this writing, the production version of Oracle Database 10*g* has just been released, so we're still trying to figure out exactly how it all works.

from these sources eventually overwhelmed us. I've described several of these problems in detail in my book (Millsap and Holt, *Optimizing Oracle Performance*, pp. 177–187). One dominant factor was simply the lack of drill-down capability: it is impossible to provide adequate detail-mining capability using V$ data without either forcing extra iterations of the data collection process or implementing an expensive-to-build and expensive-to-maintain SGA-attach mechanism. Having said that, if you know what you're doing and have the time to dedicate to it, an SGA-attach mechanism can provide very useful information. In fact, in the very next chapter, Kyle Hailey talks about his experiences in developing just such a mechanism.

This realization stopped the show for our pursuit of using the traditional V$ data sources. However, we learned very quickly that Oracle's extended SQL trace data suffered from no such restriction. Notably, in our research since 1999, we have learned to appreciate the following benefits of extended SQL trace data:

- *It's cheap:* The Oracle extended SQL trace feature is inexpensive both in terms of acquisition cost (it's part of the core Oracle database product) and operational cost (used properly, the feature is far less invasive than a data acquisition tool that samples your V$ data structures using SQL or attaches directly to an instance's SGA).

- *It's reliable:* Extended SQL trace bugs are extremely rare. On the rare occasion when we have encountered a bug in the interface, Oracle Corporation has been swift to provide a repair. Bug 3009359 is an excellent example.

- *It's complete:* There are several things you can learn from extended SQL trace data that are impossible to find in the V$ data sources, including the ability to view response time contribution by individual database calls (for example, parse, execute, fetch); the ability to determine recursive relationships among database calls; and the ability to derive the amount of response time consumed by process preemption. These days, the only operational timing data that my colleagues and I need in most performance improvement projects is extended SQL trace data.

Extended SQL trace simply works better at diagnosing performance problems than anything else we've seen. It is the critical link between the user's performance experience and the question that performance analysts have always needed to answer: "What *took* so long?" As simple as that question seems, the ability to answer it in Oracle is revolutionary.[14]

---

14. And still impossible to answer in other databases such as SQL Server.

Diagnosis Beyond the Database

Certainly not all Oracle system performance problems have their root causes within the Oracle database. Yet Oracle's extended SQL trace feature is very much a database-centric way to measure the response times. How much good does the feature do you when a performance problem has its root cause outside the Oracle kernel? A very curious benefit of Oracle's extended SQL trace data is its ability to inform you of actions that are going on *outside* of the database. It seems counter-intuitive that such problems could be fully diagnosable with only data collected from the Oracle database tier. But it works.

For example, application coding problems like the one shown in Listing 5-7 are trivial to find in extended SQL trace data. In this example, an application executes a parse call inside a loop.

Listing 5-7. Bad Application Code That Parses Too Much, and the Trace Stream That Results

```
/* BAD application code that parses too many times */
foreach value in (set_with_N_elements) {
    sql = concat('select ... where col=', quote(value));
    handle = parse(sql);
    execute(handle);
}

/* The extended SQL trace stream that results */
PARSE #1:...
WAIT #1: nam='SQL*Net message to client' ...
WAIT #1: nam='SQL*Net message from client' ...
EXEC #1:...
WAIT #1: nam='SQL*Net message to client' ...
WAIT #1: nam='SQL*Net message from client' ...
...
PARSE #1:...
WAIT #1: nam='SQL*Net message to client' ...
WAIT #1: nam='SQL*Net message from client' ...
EXEC #1:...
WAIT #1: nam='SQL*Net message to client' ...
WAIT #1: nam='SQL*Net message from client' ...
...
PARSE #1:...
WAIT #1: nam='SQL*Net message to client' ...
WAIT #1: nam='SQL*Net message from client' ...
EXEC #1:...
```

```
WAIT #1: nam='SQL*Net message to client' ...
WAIT #1: nam='SQL*Net message from client' ...
...
```

Applications that do this type of thing are slow, and they scale very poorly. With extended SQL trace data, it's easy to see why the code in Listing 5-8 is so much more efficient than the code in Listing 5-7. Not only does the good code save the database from the bother of having to process $N-1$ parse calls, the good code also saves the network from having to process $N-1$ network round-trips between the application client and the database server.

Of course, competent interpretation of standard SQL trace data would have revealed the possibility that excessive parsing might be causing a performance problem. However, standard SQL trace data cannot tell you *how much* of an impact fixing the problem might yield. The extraordinary value of the *extended* SQL trace stream is its *response time* data that allows you to determine exactly how much end-user response time you might save by fixing the problem.

This information is hugely important. Without it, you cannot evaluate the merit of a proposed repair without actually trying it first. Eliminating the trial and error is the magic that Virag Saksena brought to my group in 1995. Virag didn't just say that an activity like reducing the number of parse calls was a "good idea." With extended SQL trace data, he was able to tell us exactly *how* good an idea it was. In cases where he could see that a proposed repair would yield a poor return on investment, he would steer his clients clear of wasting their resources on it. In cases where a proposed repair would create extraordinary *positive* impact, he was more effective than most consultants at arguing his ideas into production, because he could demonstrate the merits of those ideas by using simple language that technicians and users alike could understand.

Listing 5-8. Good Application Code That Parses Once, and the Trace Data That Results

```
/* GOOD application code that parses once */
sql = 'select ... where col=:v';
handle = parse(sql);
foreach value in (set_with_N_elements) {
    execute(handle, value);
}

/* The extended SQL trace stream that results */
PARSE #1:...
WAIT #1: nam='SQL*Net message to client' ...
WAIT #1: nam='SQL*Net message from client' ...
```

```
EXEC #1:...
WAIT #1: nam='SQL*Net message to client' ...
WAIT #1: nam='SQL*Net message from client' ...
EXEC #1:...
WAIT #1: nam='SQL*Net message to client' ...
WAIT #1: nam='SQL*Net message from client' ...
EXEC #1:...
WAIT #1: nam='SQL*Net message to client' ...
WAIT #1: nam='SQL*Net message from client' ...
```

The New "Step One"

In the old days, step one of a performance improvement project was to guess which system metrics to collect (or perhaps it was to collect them all and then guess which ones to pay attention to). These days, an extended SQL trace of a business's most important functional action is the first diagnostic data collection I perform. The feature is so informative that usually it's the *only* runtime diagnostic data collection I perform. Perhaps contrary to many people's expectation, I do not even use other tools to identify which parts of the system to trace. Instead, I construct a list of user actions that the *customer* has identified as being unsatisfactorily slow and collect extended SQL trace data for those transactions. This top-down focus on *business* priority (instead of letting the system tell you what it thinks is wrong with itself) is the foundation of our new performance improvement method (Millsap and Holt, *Optimizing Oracle Performance*).

Using extended SQL as a primary diagnostic data source has worked phenomenally well in the field, even in circumstances where I would have thought, in 1995, that it couldn't work at all. Extended SQL trace is what my colleagues and I turn to, regardless of what expensive database monitoring software is on site when we land. With extended SQL trace, we're able to solve problems from a surprisingly broad domain of root causes, including the following:

- *Data design mistakes:* Lack of constraint declarations, poor entity relationship design

- *Application mistakes:* Excessive parsing, inadequate use of array processing, overuse of locks, unnecessary memory buffer access serialization

- *Query mistakes:* Poorly written SQL, faulty indexing strategies

- *Operational mistakes:* Poorly collected optimizer statistics, faulty purge processes, poorly sized caches, poor batch management strategies, data density issues

- *Capacity shortages:* Insufficient CPU, memory, disk, or network capacity for the required workload

- *Network configuration mistakes:* Poor protocol selection, faulty devices

- *Disk configuration mistakes:* Poorly distributed I/O patterns, faulty devices, poorly configured RAID architectures

The reason the feature works so well—and for so many different problem types—is simple:

1. A piece of workload has a performance problem if and only if its run time is excessive.

2. Proper use of Oracle's extended SQL trace feature shows you where a given piece of workload has spent *almost all* of its time.

3. Therefore, performance problems cannot hide from Oracle's extended SQL trace data.

The only gap in this logic is the *almost* in step 2. Extended SQL trace is not quite airtight. For example, even in version 10.1.0.2, there is no Oracle timed event that covers the total amount of time that an Oracle kernel process spends writing its trace data to the trace file (although I think this feature would be easy for Oracle to add).[15] But the evolution of the extended SQL trace feature has narrowed the gaps in Oracle timing data to a hair's breadth.

Extended SQL trace is, in my considered opinion, the most important performance diagnostic feature in any Oracle system. It is the microscope that has brought the management of Oracle system performance into the scientific age.

---

15. On first blush, the problem of instrumenting writes to the trace file seems like a recursive problem, but it's easy to avoid the recursion. If the Oracle kernel would store an aggregation of all the individual durations spent making `write()` calls to its trace file, it could merely print some kind of `WAIT #0: nam='writing to trace'...` event to the trace file upon the `close()` call for the trace file descriptor. Hotsos has actually implemented this solution as an add-on that doesn't require any modification to Oracle code.

References

Millsap, Cary. "Why you should focus on LIOs instead of PIOs" Hotsos (`http://www.hotsos.com`), 2001.

Millsap, Cary, and Jeff Holt. *Optimizing Oracle Performance*. Sebastopol, CA: O'Reilly & Associates, 2003.