

# For Developers: Making Friends with the Oracle Database

Cary Millsap

Method R Corporation, Southlake, Texas, USA

cary.millsap@method-r.com

Revised 2009/02/09

*To many application developers, a database is just a “data store” with an API that they call when they need to persist an object. It’s an abstraction that makes sense from one perspective: in a world where you’re expected to write dozens of new features every day in Java, PHP, or C#, who has the time or the inclination to dive into what’s going on deep inside the Oracle Database? As a result of this abstraction, though, developers sometimes inflict unintended performance horrors upon their customers. The good news is that you can avoid most of these horrors simply by better understanding a bit more about what’s going on inside the Oracle kernel. The trick is knowing which details you need to study, and which you can safely learn later. This presentation describes, from a developer’s perspective, some of the most important code paths inside the Oracle kernel that can make the difference between an application that breaks down under load and one that can scale to thousands of users.*

## 1 INTRODUCTION

Since the early 1990s, my professional focus has been software performance. By that I mean *speed*. I’m a developer, too. In my career, I’ve written a lot of C code. These days, I write a lot of Perl. My aim in this paper is to give you a better understanding about what goes on inside the Oracle Database in response to the code you write. I’m going to show you examples of code written in Java, but what you’ll see applies equally regardless of whether your code is PHP, C#, Ruby, Python, Perl, C, or something else.

My goal is simple. If I can get you to understand a little more clearly what’s going on inside the Oracle Database kernel, you’ll write better, faster code. Although developers are often “taught” that they shouldn’t concern themselves with what goes on inside the database, that’s really not true. Not if you want to build big applications that run fast, anyway.

So my job is to help you make friends with Oracle. When we’re done, you’ll write faster code, and you’ll probably write *less* code, too, that’ll be easier to maintain. I’ll show you an example. I think that once you get to know your database a little better, you’ll be impressed with some of the things it does for you.

Making friends with new software is not altogether different from making a new *human* friend. The best way to get started is to learn your new friend’s language.

Now, you’re probably already at least a little bit familiar with the SQL language, which enables you to read and write data to the Oracle Database. Teaching you SQL is not my aim here. My aim is to teach you how to communicate about the *performance* of the application code you write. There are lots of tools that let you assess the performance of the client code you write. In this paper, I’ll acquaint you with a tool that Oracle Corporation provides, which will help you measure very easily how efficiently the code you write uses the Oracle Database.

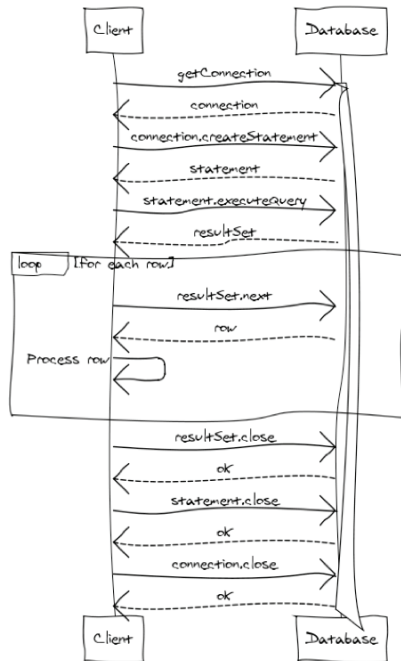


Exhibit 1. Sequence diagram for a multi-row database query.

## 2 WHERE SOFTWARE SPEED COMES IN

Let's take a look at how an application that uses Oracle might work. Exhibit 1 shows a rough UML sequence diagram that serves as a good conceptual model for how a business task that fetches rows from an Oracle database works.

It's a pretty simple model. First, the application does what it has to do to connect to Oracle and prepare a SQL statement for fetching the data. Then it fetches a row at a time, processing each row in turn. It does this until it has processed all the data that it needs. Then it cleans up after itself.

Mission accomplished, right? It's certainly a thrill the first time you make something like this happen: making data go from this complex and expensive "black box" into your application, then perhaps into an HTML page where anyone on the Internet can see it.

But what about the performance of the application? Maybe on your development system, the code you wrote is lightning fast. There are two questions you need to be able to answer:

1. Is your code as fast as it *should* be?
2. Will your code *scale*?

When I say *scale*, I mean it in the mathematical sense that means the rate at which a user's response times will *change* as some other factor in the system

changes. For example, for a system "not to scale well to large user counts" means that it slows down more aggressively than you want as more users log into the system. If it doesn't scale well to large data volumes, then that means it will slow down more aggressively than you want as people insert rows into your database.

The point of this paper is to give you some insight into how you can *measure* how efficiently your application is written, so that you can know whether your code is as fast as it should be and whether it will scale. Why guess? ...when you can *know*.

## 3 FEEDBACK

Probably the most important tool you need to make *learning* happen more efficiently is *feedback*. A little girl reaches for the hot stove. Mommy yells, "No!" That's feedback. The little boy standing close by wants to see what all the fuss is about, so he decides to touch the stove anyway. POW! More feedback. Feedback from the stove is even more memorable than the feedback from Mommy.

Feedback that is close in time to the behavior it measures is more valuable than feedback that happens much later than the behavior it measures. Imagine what would happen if, when the little boy touched the hot stove, it didn't hurt until three months later. Without immediate feedback, the boy might leave his hand on the stove for several seconds, and then what happens three months later would be absolutely horrifying. Three months later, a lot of the really good choices wouldn't be available anymore. Deferred feedback can be deadly. Imagine if there were a ten-second delay on what you could see through your car's windshield. See the story of the chemical element radium for an important historical example (Wikipedia 2008).

So, how can you shorten the feedback loop about the performance of the application you are writing today? With a feature that comes standard with every release and edition of the Oracle Database since version 7. The feature is called "extended SQL trace." I've written about its history in (Nørgaard, et al. 2004, 155-182).

When you're interested in generating your own trace files, you can see how to do that in section 17 near the end of this paper. Right now, let's take a look at the kinds of things you'll find in a trace file that will help you write better code.

## 4 INTERPRETING YOUR TRACE FILE

Trace files can be intimidating, especially when they contain hundreds of thousands of lines. But even the most complicated trace files are rooted in a surprisingly small number of fundamental principles, which I'll cover here.

Oracle trace files record only two basic categories of useful information about where your time has gone:

### Database calls

Each line that begins with the token `PARSE`, `EXEC`, or `FETCH` represents a single, completed database call executed by the Oracle kernel. A database call line tells you, with its *e* value, how many microseconds of elapsed time the call consumed. Its *c* value tells you approximately<sup>1</sup> how many microseconds of CPU time the call consumed.

### Operating system calls (OS calls)

Each line that begins with the token `WAIT` represents a single, completed operating system call (OS call) executed by the Oracle kernel.<sup>2</sup> An OS call line tells you, with its *ela* value, how many microseconds of elapsed time the call consumed.

The trick to understanding Oracle trace data is to learn what the various database calls and OS calls mean. Once you know how to figure this out, you'll be able to read for yourself an irrefutable play-by-play account of exactly what your code has spent its time doing, measured from the Oracle Database's perspective.

## 5 TRACE FILE GUIDED TOUR

Now let's take a look at the guts of an Oracle trace file. When we discuss Oracle trace file contents, you need to remember that it's the Oracle kernel process that writes to the trace file. The play-by-play you'll see in the Oracle trace data stream is the story told from the perspective of an Oracle *server* process like the one shown in Exhibit 2.

<sup>1</sup> The *c* statistic in Oracle trace data is only as accurate as the *getusage* function your OS provides. On systems that don't use *microstate accounting* (e.g., Solaris does), a *c* statistic is accurate to only  $\pm 10,000$   $\mu$ s. For more details, see chapter 7 of Cary Millsap and Jeff Holt, *Optimizing Oracle Performance* (Sebastopol, CA: O'Reilly, 2003).

<sup>2</sup> It's a little more complicated than that, because sometimes the Oracle kernel calls more than one OS call in the context of a single `WAIT` line. But it usually doesn't matter if you don't know this.

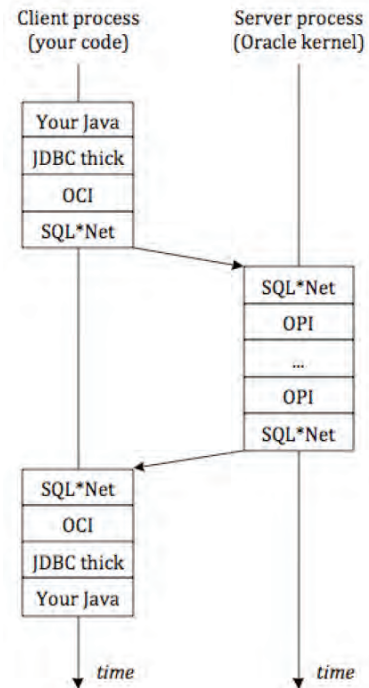


Exhibit 2. The Oracle two-tier, client-server architecture; the Oracle kernel process is the one that writes the trace data you'll be reading.

The following paragraphs contain a play-by-play analysis of an Oracle trace file produced by a Java program that consumed about 23 seconds of response time querying 142,517 rows out of an Oracle database. You can see more details from that trace file in section 19.1, near the end of this paper. The action within the trace file that we're really interested in begins at line 24:

```
24. =====
25. PARSE IN CURSOR #1 ...
26. select * from sla_run
27. END OF STMT
```

These lines reveal what future references to "#1" will mean: they'll be references to a cursor associated with the SQL statement "`select * from sla_run`".

```
28. PARSE #1:c=0,e=251,...
```

Line 28 is where the action really begins. This line reveals that the Oracle kernel executed an Oracle Program Interface (OPI) *parse* function upon the *select* statement shown on line 26. What I'm calling an OPI *parse* function is the server-side companion of the *OCIStmtPrepare* call, which the JDBC executed within our client process. To see details about what such a function does, consult (Oracle Corporation 2008). This *parse* function consumed 251  $\mu$ s (that is, 251 microseconds, or 0.000251 seconds) of response time.

29. BINDS #1:

There were no placeholders in the SQL statement to which values were to be bound; otherwise, we would have seen more information (more lines of trace data) here.

```
30. EXEC #1:c=0,e=84,...
31. WAIT #1: nam='SQL*Net message to client' ela= 2 ...
```

The Oracle kernel executed an OPI *exec* function (companion of *OCISmtExecute*) upon the cursor described in lines 24 through 27. The function consumed 84  $\mu$ s of response time. Then the Oracle kernel wrote some information back to its caller (the Java program) through its SQL\*Net interface. The 2  $\mu$ s duration reported here is *not* the duration of an OS *write* call, although you have a right to expect that it would be (Millsap, SQL\*Net 2005). It is actually only the duration of an OS timer call executed before the *write*. It's an Oracle bug, but not a very important one.

```
32. WAIT #1: nam='SQL*Net message from client' ela= 25227 ...
```

Then the Oracle kernel blocked upon an OS *read* call (the same way a *read* of your keyboard will block until you press the *Enter* key), passing approximately 25,227  $\mu$ s of end-user response time. This is actually the duration of the whole network round-trip from the Oracle kernel process to the client and back. It also includes all the time spent executing code on the client.

```
33. WAIT #1: nam='SQL*Net message to client' ela= 24 ...
34. FETCH #1:c=0,e=355,...,r=10,...
35. WAIT #1: nam='SQL*Net message from client' ela= 7851 ...
```

The kernel executed an OPI *fetch* function (companion of *OCISmtFetch*) upon the cursor executed in line 30. The fetch returned 10 rows in one network round-trip, consuming 355  $\mu$ s. You can regard the *SQL\*Net message to client* call as a tiny little 24  $\mu$ s of code path executed within the context of the fetch. After the fetch, the kernel blocked upon an OS *read* call for another 7,851  $\mu$ s awaiting the next call from the client.

```
36. WAIT #1: nam='SQL*Net message to client' ela= 2 ...
37. FETCH #1:c=0,e=84,...,r=10,...
38. WAIT #1: nam='SQL*Net message from client' ela= 1306 ...

39. WAIT #1: nam='SQL*Net message to client' ela= 2 ...
40. FETCH #1:c=0,e=81,...,r=10,...
41. WAIT #1: nam='SQL*Net message from client' ela= 1282 ...
```

(42,746 lines here are not shown)

```
42. WAIT #1: nam='SQL*Net message to client' ela= 2 ...
43. FETCH #1:c=0,e=86,...,r=7,...
44. WAIT #1: nam='SQL*Net message from client' ela= 2141 ...
```

Here, the same three-line pattern repeats thousands of times. The kernel returns ten rows per network round-trip to the client until finally there are only seven rows returned in the final attempted ten-row

fetch, at which point the client code path can detect that it has fetched all the rows that the kernel had to offer, and the query is finished.

Reading a trace file line-by-line is not tremendously difficult; it gets tedious, but not difficult. Another layer of difficulty folds in when you want to summarize the information presented by a block of trace file lines. The job is more complicated than you might have guessed by now, because there are gaps and overlaps in how the Oracle kernel emits its own timing data. You can learn about the details in (Millsap and Holt 2003).

You can often get a good general sense of what's going on in a trace file simply by aggregating *c* or *e* values for database calls and *ela* values for OS calls. I summarize trace data with a software tool that I helped design and develop, called the Method R Profiler.<sup>3</sup>

Summarizing the elapsed time consumed by database calls and OS calls yields the information shown in Exhibit 3.

Response time (seconds)		Call
20.070	85.8%	SQL*Net message from client
1.652	7.1%	CPU service, fetch calls
1.683	7.2%	all other
23.405	100.0%	Total response time

Exhibit 3. Response time profile of our 23-second query program execution.

The code spends 85.8% of its user's time moving rows across the network<sup>4</sup> and only 7.1% fetching rows out of the database. That's a pattern I want you to recognize when you see it, this issue of response time being dominated by network I/O. It is an important performance antipattern.<sup>5</sup>

## 6 OPTIMIZING THE QUERY PROGRAM

Whenever network I/O time dominates the response time of a program you've written, it's probably an easy opportunity for you to optimize your code. It's usually not hard to perform the optimization. The step that most people miss is to look at response time

<sup>3</sup> Visit <http://method-r.com/software/profiler-info> for details.

<sup>4</sup> Some of the 20.070 seconds of *SQL\*Net message from client* time is spent by client Java code path. If I wanted to test exactly how much time the client program is spending, I would profile the client-side Java code.

<sup>5</sup> Ironically, many Oracle authors teach that you should ignore all *SQL\*Net message from client* calls. I hope by now that you can see how awful of an idea that is.

decomposed this way to begin with. Once you see the problem, you can focus your energy on fixing it. So, here we go...

Did you flinch at all when you saw this line of the trace data that I showed you earlier?

```
34. FETCH #1: c=0, e=355, ..., r=10, ...
```

Notice that the fetch call returned not just one row to the application, but *ten*. Whenever your Java code asks the JDBC for *rowSource.next*, the JDBC cleverly grabs and buffers rows for you in batches of ten. The default array size is 10. Had the array size been set to 1 instead, can you guess what our response time would have looked like?

The answer is that the program's response time would have been about ten times worse. We tried it:

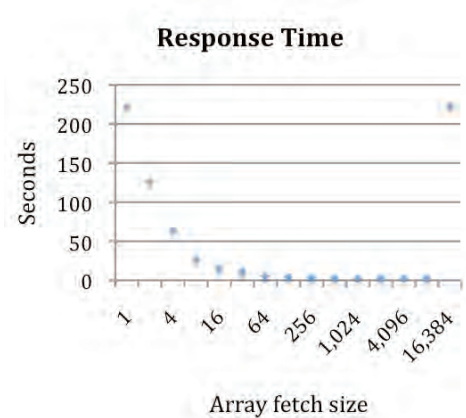
Array fetch size	Response time (seconds)
1	240.297
10	23.405

It's easy to understand why, once you've seen the trace data. If our program had fetched one row at a time instead of ten, it would have made ten times more network round-trips to fetch all the data. Thus, instead of the 20 seconds of response time you saw in Exhibit 3, you should expect 200 seconds of network I/O time. ...Which is almost exactly what did happen.

At this point, I hope you've begun to wonder about the following questions:

1. What if you were to make the array fetch bigger than 10? Would it improve response time?
2. Is there a point of diminishing returns, where the array fetch size can be too big?
3. How do you manipulate the array fetch size?

The answer to the first question is that yes, absolutely, increasing the array fetch size improves response time for our program. We tried several array fetch sizes, and you can see from the shape of the curve in Exhibit 4 that manipulating the array fetch size had a profound impact upon performance.



*Exhibit 4. Array fetch size has a profound impact upon our query's performance.*

In our program testing, we tried lots of array fetch sizes, and we found that the sweet spot for us (given our row sizes, our TCP/IP packet sizes, etc.) was roughly 2,048. That's where our response times settled in at about 2.4 seconds. That array fetch size yields about a 90% reduction in response time compared to the default setting of 10.

Array sizes larger than 2,048 didn't produce a performance benefit for us, in spite of consuming a lot more memory. For array fetch sizes larger than 4,096, response times actually degraded a bit. When we tried an array fetch size setting of 16,384, our Java program promptly died of an out-of-memory error.<sup>6</sup> The bathtub shape of the data in Exhibit 4 tells us that bigger is better to a point, at which too big becomes bad.

So how can you manipulate your array fetch size? Well, if you use the Sun JDBC, you can't. But if you use the Oracle JDBC, it's easy. You execute the *setFetchSize* method upon the statement object, like this:

```
statement.setFetchSize(size);
resultSet = statement.executeQuery(query);
while (resultSet.next()) {
    // do your business upon resultSet
}
```

Exhibit 5 shows how the program that used the array fetch size of 2,048 spent our 2.4 seconds.

<sup>6</sup> We used a maximum Java heap size of 64 MB by specifying `java -Xms 2m -Xmx 64m`.

Response time (seconds)		Call
0.890	37.1%	SQL*Net message from client
0.823	34.3%	SQL*Net more data to client
0.745	31.0%	CPU service, fetch calls
-0.058	-2.4%	all other
2.400	100.0%	Total response time

Exhibit 5. Response time profile of our improved 2.4-second query program execution.

Don't feel too unsettled by the negative number shown in the "all other" category of Exhibit 5. It is an artifact of the  $\pm 10,000\text{-}\mu\text{s}$  accuracy on the CPU statistic that I mentioned earlier. Sometimes, the *c* statistic over-accounts for elapsed duration. For example, a database call with *c*=10000, *e*=8000 would indicate:

Response time (microseconds)		Call
10,000	125.0%	CPU service, ...
-2,000	-25.0%	all other
8,000	100.0%	Total response time

The point is that by manipulating our array fetch size, we reduced the total time we spent waiting for network I/O calls from 20.070 seconds (Exhibit 3) to a much more bearable  $0.890 + 0.823 = 1.713$  seconds. We even reduced the amount of CPU time we spent in fetch calls, presumably because we're not making nearly as many fetch calls with the larger array fetch size.

The overall performance improvement is spectacular (23.405 seconds to 2.400 seconds) because we targeted the exact reason that the program spent so much of our time. Such is the principal beauty of profiling: it focuses our attention where it belongs, and it safely allows us to ignore everything that doesn't matter.

Section 19.2 shows some of the raw trace data for the improved program. Here's a brief walk-through of how that program progressed. The interesting action begins on line 24:

```

24. =====
25. PARSING IN CURSOR #1 ...
26. select * from sla_run
27. END OF STMT
28. PARSE #1: c=1000, e=121, ...
29. BINDS #1:
30. EXEC #1: c=0, e=57, ...
31. WAIT #1: nam='SQL*Net message to client' ela= 3 ...
32. WAIT #1: nam='SQL*Net message from client' ela= 5960 ...

```

This is the same pattern as you saw previously in the slow code. However, what comes next is very different:

```

33. WAIT #1: nam='SQL*Net message to client' ela= 6 ...
34. WAIT #1: nam='SQL*Net more data to client' ela= 149 ...
35. WAIT #1: nam='SQL*Net more data to client' ela= 7 ...
36. WAIT #1: nam='SQL*Net more data to client' ela= 9 ...
    (many more 'SQL*Net more data to client' calls happen here)
68. WAIT #1: nam='SQL*Net more data to client' ela= 10 ...
69. FETCH #1: c=11998, e=22206, ...r=2048, ...

```

Here, the Oracle kernel makes a *SQL\*Net message to client* call to ship results back to the client, but the result set is so big that it won't fit into a single network packet. So the kernel makes a number of *SQL\*Net more data to client* calls to fulfill the passage of data that it owes to the client. Finally, the kernel completes the *fetch* call, which you can see returned 2,048 rows in 22,206  $\mu\text{s}$ , after consuming approximately 11,998  $\mu\text{s}$  of CPU time.

The pattern repeats until the end of the trace file:

```

(2,859 lines here are not shown)
70. FETCH #1: c=6999, e=6837, ...r=1205, ...
71. WAIT #1: nam='SQL*Net message from client' ela= 18557 ...

```

...where the last 6,837- $\mu\text{s}$  *fetch* returns the final 1,205 rows.<sup>7</sup>

## 7 OPTIMIZING BEGINS WITH MEASURING

Here's a quick quiz. What have you learned so far?

- Your optimal Oracle array fetch size is 2,048.
- You should always check to make sure that you've optimized your array fetch size.
- You should always check to see where your response time is going before you optimize anything.

Choice (a) is a really poor one, because the optimal Oracle array fetch size for the next program you write is probably going to be different than the optimal array fetch size for the program we just analyzed. I haven't studied it in detail, but I believe that your optimal array fetch size is a function of several factors, including at least these:

How big are the rows you're returning?  
How big are your network packets?

I can easily imagine, for example, that if our rows had been 100 times larger than they were, then our optimal array fetch size might have been 100 times smaller. However, I've learned enough over the years to believe that any kind of a mathematical model is

<sup>7</sup> Earlier, I mentioned that this query returns 142,517 rows. Note that  $142,517 \% 2,048 = 1,205$  (using '%' as the modulus operator), so 1,205 is exactly the number of rows you should expect to be left, after several 2,048-row fetches, for the final fetch to grab.

useful only as a starting point, and that you should base your optimizations upon real operational data.

Choice (b) is therefore superior to choice (a), but what if network response time had accounted for only 8.58% of response time? Then the impact of successfully optimizing your network I/O would have been so diminished that it might not be worth your time invested into doing it.

Choice (c) is therefore superior to choice (b). Not every program you write will have response time dominated by network I/O. There are thousands of different ways your program can spend your user's time. Why use a checklist to attack everything that might possibly be slowing your code down when you have a tool at your disposal—the Oracle extended SQL trace data—to show you *exactly* how your code is spending its time? Why guess? ...when you can *know*.

Oracle's trace files give you a language for understanding the performance of the code you write.

## 8 ON SPECIFICATION LEGITIMACY

I've just described a process of analyzing and improving performance for a query that returns 142,517 rows. I didn't tell you anything about how the Java program fetching those rows was going to *use* those rows. I left that to your imagination. I merely hoped you would tacitly assume that someone needed all those rows. Developers can tend to do that.

I, of course, chose such an example because it's easier to showcase the problem of "too much network I/O" when you illustrate with a query that returns a lot of rows. However, in real life, it's fair to ask the question, "Why do you need 142,517 rows?" Or even, "Do you *really* want all those rows?"

Dan Tow<sup>8</sup> and I once had a dinner conversation in which we agreed on the following postulate:

No human *ever* wants to see more than 10 rows.

Our idea was that once you're presented with more than 10 rows to look at, all in one picture, you'd really rather see some kind of aggregation (count, sum, mean, ...whatever) of the data instead.<sup>9</sup>

So, the next time you're asked to improve the performance of a query that returns a jillion rows, at least ask the question whether the user using your program really, *really* wants to see a jillion rows in the first place. It's easier, cheaper, and more effective to

---

<sup>8</sup> You can find Dan at <http://www.singingsql.com>.

<sup>9</sup> Our corollary, of course, was, "Auditors are not human."

give a user less data—if that's what he really wants—than to make your program faster at returning stuff that people didn't really want to begin with.

## 9 TRACE FILE GUIDED TOUR #2

Let's look at another example of some trace data that reveals another performance antipattern you should know about. Section 19.3 shows an excerpt of an Oracle trace file for a sequence of 10,000 inserts, which consumed 32.706 seconds—about 33 seconds—of end-user response time.

First, before we look at the trace data, let me ask you this: Is 33 seconds a good response time for a program that inserts 10,000 rows into a table? Or is it a bad response time?

People's first response when I ask a question like that is usually either eyes-averted silence or "What kind of machine did it run on?" The truth is, unless you've recently encountered an application that did something similar to what this program did, you probably don't have any idea. There's nothing wrong with that. In a few minutes, you'll know the answer anyway.

So, let's walk through the lines and see what's going on. The interesting action begins at line 24:

```
24. =====
25. PARSING IN CURSOR #2 ...
26. insert into JAVA_TEST_TABLE values ('0')
27. END OF STMT
```

Lines 24 through 27 indicate that cursor "#2" refers to the SQL statement that inserts a row into *java\_test\_table*.

```
28. PARSE #2:...,e=678,...
29. BINDS #2:
```

These lines indicate that an OPI *parse* call has consumed 678  $\mu$ s of response time, and that there were no values bound to placeholders in the statement. (That makes sense, because there aren't any placeholders in the statement.)

```
30. EXEC #2:...,e=1245,...,r=1,...
```

The Oracle kernel executed an OPI *exec* function, which consumed 1,245  $\mu$ s of response time to insert one row.

```
31. XCTEND r1bk=0, rd_only=0
```

The kernel executed an OPI *commit* function (server-side companion to *OCITransCommit*), which committed everything this session has done thus far to the database.

```

32. WAIT #2: nam='SQL*Net message to client' ela= 2 ...
33. WAIT #2: nam='SQL*Net message from client' ela= 1154 ...

```

A network round-trip consumed 1,156  $\mu$ s of response time.

And then, in the following lines, you see the same pattern over and over...

```

34. =====
35. PARSING IN CURSOR #2 ...
36. insert into JAVA_TEST_TABLE values ('1')
37. END OF STMT
38. PARSE #2:...,e=302,...
39. BINDS #2:
40. EXEC #2:...,e=202,...,r=1,...
41. XCTEND rlbk=0, rd_only=0
42. WAIT #2: nam='log file sync' ela= 113 ...
43. WAIT #2: nam='SQL*Net message to client' ela= 2 ...
44. WAIT #2: nam='SQL*Net message from client' ela= 1918 ...
45. =====
46. PARSING IN CURSOR #2 ...
47. insert into JAVA_TEST_TABLE values ('2')
48. END OF STMT
49. PARSE #2:...,e=232,...
50. BINDS #2:
51. EXEC #2:...,e=146,...,r=1,...
52. XCTEND rlbk=0, rd_only=0
53. WAIT #2: nam='log file sync' ela= 107 ...
54. WAIT #2: nam='SQL*Net message to client' ela= 2 ...
55. WAIT #2: nam='SQL*Net message from client' ela= 2050 ...
(106,315 lines here are not shown)

```

However, you can see, on lines 42 and 53, a kind of OS call that I haven't shown you yet in this paper. The Oracle name *log file sync* refers to a synchronization event that takes place between the Oracle kernel process inserting the rows and a "background" Oracle process called the "redo log writer." The two *log file sync* calls shown here consumed 113  $\mu$ s and then 107  $\mu$ s of response time.

Summarizing the elapsed time consumed by database calls and OS calls (aggregating the *e* and *ela* values for the individual call types) yields the information shown in Exhibit 6.

Response time (seconds)		Call
14.637	44.8%	SQL*Net message from client
6.389	19.5%	log file sync
4.901	15.0%	CPU service, parse calls
0.943	2.9%	CPU service, execute calls
5.836	17.8%	all other
32.706	100.0%	Total response time

Exhibit 6. Response time profile of our 33-second insert program execution.

## 10 OPTIMIZING THE INSERT PROGRAM

Now let's revisit the question, "Is 33 seconds a good response time for this program?" Exhibit 6 gives enough data to prove that the answer is a resounding *no*. Here's why...

The whole point of our program is to insert 10,000 rows into a table. The Oracle function that does that work is represented in the trace data by the EXEC lines. This program spent only 2.9% of its time—less than a second!—actually putting rows into the database; yet I had to wait more than half a minute for the program to complete. The remainder of the time was spent (wasted?) doing other things.

Again, you can see in Exhibit 6, the dominant response time contributor fits the "excessive network I/O" antipattern. The trace file actually shows one network round-trip for every single row that's inserted into the database. The next step should be obvious to you by now: we need to find a way to insert more than one row per network round-trip.

Here's how the original Java code was written:

```

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;

for (Integer i : ilist) {
    String sql = String.format("insert into t values ('%d')", i);
    Statement st = conn.createStatement();
    st.executeUpdate(sql);
}

```

This code performs the extraordinarily nasty act of creating 10,000 (the size of *ilist*) distinct SQL statements within the Oracle Database. Yes, your database administrator might tell you not to worry, that he has a database parameter called *cursor\_sharing* that will take care of the problem.

But it won't.

...Because your problem isn't so much the 4.901 seconds of *parse* work that's being done on the Oracle Database server as it is the 14.637 seconds of work moving data across the network from your client code to your server code. Manipulating your Oracle Database's *cursor\_sharing* parameter may make your database administrators happy (some of the red lights on his dashboard will turn green), but it's not going to make your *users* happy, because their response times will still be intolerable.<sup>10</sup>

<sup>10</sup> The Oracle *cursor\_sharing* parameter lets your database administrator instruct the Oracle kernel to translate your SQL statements into what those statements might have looked like if you had used placeholders instead of literal values. It's a clever plan, but there are two big problems with it. First, it puts even *more* workload on your database server. More importantly, it does nothing to reduce the quantity of network I/O that is our example's dominant problem.

People can argue all they want about whether to make adjustments to the network or the database, but a good application developer can stop all the discussion by writing the code a different way. I'll show you how we re-wrote it shortly. Before I get into the code changes, let me show you the performance improvement we were able to achieve, which you can see in Exhibit 7. That should serve as some motivation for you to move forward.

Response time (seconds)		Call
0.083	55.0%	SQL*Net message from client
0.040	26.4%	log file sync
0.029	19.2%	CPU service, execute calls
-0.001	-0.5%	all other
0.151	100.0%	Total response time

Exhibit 7. Response time profile of our optimized 0.151-second insert program execution.

That's a 99.5% performance improvement, from 32.706 seconds to 0.151 seconds. More importantly, that's an improvement from "get up from your desk" slow to "click-flash" fast. And it frees up over 32 seconds' worth of capacity on your system that allows other work to take place without having to compete against *your* program for resources.

The changes we made to the code are all in response to what the profile in Exhibit 6 showed us. A simple top-down walk of that profile led us to ask the following questions:

- Do we really need to spend 14.637 seconds doing network communications?
- Do we really need to spend 6.389 seconds executing *log file sync* calls?
- Do we really need to spend 4.901 seconds preparing SQL?

Now let's *answer* those questions.

We've already seen the "excessive network I/O" antipattern in the query example I worked through for you earlier. It is related to the excessive amount of time we've spent preparing SQL (those PARSE calls that showed up in our raw trace data).

The answer here is, once again, to figure out how to process more than one row per database call. The general strategy we want to employ is to prepare our SQL statement only once, bind values into that statement to represent all our rows to be inserted, and then make sure we're bundling bunches of row insertions into each database call our code motivates.

You can do that with the Java Database Connectivity API from Sun (the Sun JDBC). There are good

resources on the Internet to show you how (Crawford, Farley and Flanagan 2005). We've found the Oracle JDBC to be a little bit easier to use, and our experience leads us to believe it's a little bit faster, too. Here's the code we used to produce our 0.151-second example:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;
import oracle.jdbc.OracleConnection;

((OracleConnection)conn).setDefaultExecuteBatch(1000);
conn.setAutoCommit(false);
String sql = String.format("insert into t values (?)");
PreparedStatement pst = conn.prepareStatement(sql);
for (Integer i : ilist) {
    pst.setInt(1, i);
    pst.executeUpdate();
}
conn.commit();
```

First, we imported the Oracle JDBC that's required to use the *setDefaultExecuteBatch* value to 1,000. Skipping over the *setAutoCommit* call for a moment, you can see that now instead of preparing thousands of distinct SQL statements inside the loop, we've prepared a single SQL statement outside the loop, and this statement uses the '?' character as a placeholder for the values to be inserted. Using (and reusing) a *PreparedStatement* will save 9,999 of the OPI parse calls that plagued the original 32-second program.

Inside the loop, we now have a *setInt* call that binds the value of *i* to the first (and only) '?' placeholder in the SQL statement, and then we have an *executeUpdate* call to do the insertion. Since the default execute batch size is now set to 1,000, this code will only make a real Oracle database call (and the associated network round-trip) once for every 1,000 calls.

So, with this code, we've saved thousands of network I/Os and reduced the amount of time the program spends parsing inside the database to practically zero. Magic.

The *log file sync* time represents another important performance improvement opportunity.<sup>11</sup> The repeated appearance of either *log file sync* calls or XCTEND lines in a trace file (in our case, we had both) is an indication of repeated database *commit* processing. Excessive commit processing can create a performance problem that accelerates as the amount of concurrent workload increases (so it's a scalability problem too), and it even creates a functional problem as well.

<sup>11</sup> In case you're wondering how you'll figure out what these event names mean when you're done reading this paper, I cover that in section 18.

Let's talk about the functional problem first. Imagine that you're responsible for running this 10,000-row insert once a day. Imagine that it clicks along just fine for weeks on end, but then one day, part-way into the program's execution, your system housing the Oracle database crashes. It happens.

So when someone gets the Oracle database restarted, one of the things you're going to have to do is figure out how many of the rows made it into the database before it crashed. In fact, you're going to have to figure out exactly *which* rows made it into the database before it crashed, because you're going to need to make sure that either you delete those rows so you can run your program again, or you're going to need to figure out exactly which rows you need to insert so that all 10,000 will be in the database when you're done. None can be missing, and none can be duplicated.

If you're like most application developers, you're only going to want to do this job once. It's not going to be a lot of fun figuring out which rows made it and which rows didn't. If you're lucky, there'll be some kind of batch id or date field that will allow you to pinpoint exactly which rows made it in. Maybe you'll be tempted to write some kind of clean-up functions to go with your application so that this kind of scenario will be easier for you (or your customers) to deal with if it ever happens again.

If you use Oracle differently, however, you won't have to do that.

If, instead of committing every single row to the database as your application inserts it, you waited until the end of all 10,000 insertions and then committed the whole transaction, your life would be *much* simpler. Then, if the Oracle system ever failed in the midst of your job, you'd know: either there are 10,000 new rows in the database (if your *commit* call succeeded) or there are 0 new rows in the database (if your *commit* call failed).

*Plus*, you'll save a lot of unnecessary work that consumes a lot of unnecessary time. Implementing the single-commit idea reduced our time spent waiting for *log file sync* events from 6.389 seconds (Exhibit 6) to 0.040 seconds (Exhibit 7). That's huge. *And* we eliminated over six seconds of labor that *other* processes on the system had to wait behind, which makes the whole system faster for everyone.

The decision of how often to commit is an important one. I've shown here why committing after every row can be bad. However, committing only once at the end of a multi-million-row insert may be bad, too, because of the stress upon the Oracle Database undo

management subsystem, and because maybe you don't want to have to restart a whole multi-million-row transaction after an instance failure. Maybe it would be better to commit in batches of 10,000 rows, or maybe even 10 rows. Part of your application design responsibility is to manage this tradeoff, which should include the task of using *measurements* instead of guesses about the costs of commit processing.

Section 19.4 shows some of the trace data for an execution of the new program. Notice that each EXEC line now processes 1,000 rows (*r=1000*) instead of just one. And notice that there is only one *log file sync* calls now, right at the end of the trace file.

## 11 AGAIN, MEASURING IS VITAL

So, what have you learned from this paper? It's easy to start thinking in absolutes about "best practices." I'm suspicious about most so-called best practices because of Clarke's Fourth Law: For every expert, there is an equal and opposite expert (Wikipedia 2008).

There is one "best practice," though, that I believe in deeply:

---

You should always *measure* your application's performance and target your optimization efforts at places where your code will benefit from it most.

---

As you now understand, the way I implement this practice in the Oracle world today is with the extended SQL trace feature. There are other ways to peek into what the Oracle kernel is doing,<sup>12</sup> but no other way gives you such a simple sequential listing of exactly where *all* of the time has gone for just the code path that you're interested in. If you learn how to use the extended SQL trace feature, you'll learn a lot more about Oracle, and you'll learn exactly what you *need* to know, right when you need to learn it.

I think one of the nicest things about Oracle tracing (and profiling in general) is that it focuses your attention where it needs to be right now. It doesn't leave you feeling like you have to know *everything* about Oracle all at once. It's also a feature that's available in development, testing, and production environments, so it's truly a single tool that you can use throughout your entire software lifecycle.

As a developer, you can make it really easy to trace your code. My team have tried to make it easy for

---

<sup>12</sup> Oracle's Active Session History (or ASH) is one name you might recognize.

developers to instrument code the right way by publishing a free, open source instrumentation library for Oracle called *ILO* (Method R Corporation 2008). *ILO* consists of a couple of Oracle PL/SQL packages with functions you can call from Java, PHP, or whatever language you're writing in. If you're curious about how good instrumentation works, studying the PL/SQL within *ILO* is a good place to start.

## 12 SUMMARY

*Speed* is a vital feature of good software. Speed doesn't just happen by accident. It is a feature that you have to design into good programs. Designing good performance into your code is extremely difficult to do without good feedback about the speed of your code as you're writing it.

Many Oracle application developers work under the assumption that the Oracle Database kernel is a "black box" that they shouldn't bother to understand. But writing code within the abstraction model that the Oracle Database is merely a "persistent data store" almost assures that you will write code that is slower and that wastes far more precious computing resources than it should.

The Oracle extended SQL trace mechanism provides the feedback that you need to write fast, efficient code for Oracle-based systems. Extended SQL tracing reveals where your time is being spent inside the Oracle Database kernel, all the way down to the individual database or operating system subroutine call.

With this paper, I hope I have illuminated the following key points for you:

- Oracle extended SQL trace data shows you exactly where your code spends your user's time.
- With trace data, you can determine whether your code is efficient.
- If your code can run faster, tracing shows you why and by how much. It allows you to predict the performance impact of changing your code, without so much trial and error.
- Tracing prevents you from wasting time "tuning" aspects of your program that won't result in appreciable response time benefits.
- Tracing prevents the feeling that you have to know "everything about Oracle" before you can write fast, provably efficient code.
- You can use performance feedback in your programs to create self-adjusting applications that

adapt to your software's true operational constraints.

- Not every perceived requirement is a legitimate requirement. Tracing allows you to measure objectively the performance cost of a feature, which in turn helps the business make better decisions about which software features it truly requires.
- Knowing how to measure performance, and knowing how to determine whether the performance you have observed is optimal are more important than your being able to remember lists of software performance "best practices."
- You can make your application easier to tune and debug by incorporating extended SQL tracing into your code. Ideally, you should regard the ability to measure performance at run-time as an important functional specification.

I hope this paper encourages you to instrument your code, gain access to your Oracle trace data, and study it for some of your own applications that mean something to you and that need to be fast.

I haven't covered everything you need to know here, of course. For example, I haven't discussed how connection management code paths prevent your application from scaling. I haven't told you any details about why excessive parse calls and excessive visits to the database buffer cache cause an application not to scale. But I have shown you enough to get you started upon a much more informed path toward writing scalable, high-performance Oracle-based applications. I've also included, in section 18 near the end of this paper, a list of call names that you'll encounter frequently as you look through trace files.

With this paper, you'll be able to come a long way in understanding what your trace files are trying to tell you. If you're interested in studying Oracle trace files in more detail, then I hope you will have a look at (Millsap and Holt, *Optimizing Oracle Performance 2003*), where Jeff Holt and I have written about the subject in depth.

## 13 ACKNOWLEDGMENTS

Thank you to Harold Palacio for writing and testing the Java code used in my examples, and to Karen Morton, Laura Nogaj, Mark Sweeney, Ron Crisco, and Jeff Holt for reviewing my work. I am indebted to Ken Ferlita and everyone else at Method R for their ideas, research, encouragement, and manifest contributions that make it possible for me to write in the first place.

To draw the UML sequence diagram shown in Exhibit 1, I used the nifty tool at <http://www.websequencediagrams.com/>.

## 14 BIBLIOGRAPHY

Crawford, William, Jim Farley, and David Flanagan. *An Introduction to JDBC, Part 3*. 2005.  
[http://www.onjava.com/pub/a/onjava/excerpt/javacnntut\\_2/index3.html](http://www.onjava.com/pub/a/onjava/excerpt/javacnntut_2/index3.html) (accessed 12 24, 2008).

Method R Corporation. *Instrumentation Library for Oracle (ILO)*. 2008.  
<http://sourceforge.net/projects/hotsos-ilo/>.

Millsap, Cary. 'SQL\*Net message to client' isn't what you might think it is. Nov 28, 2005.  
<https://portal.hotsos.com/newsletters/volume-i-issue-2/tips/sql-net-message-to-client-isn2019t-what-you-might-think-it-is/>.

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

Nørgaard, Mogens, et al. *Oracle Insights: Tales of the Oak Table*. Berkeley, CA: Apress, 2004.

Oracle Corporation. *My Oracle Support*.  
<http://metalink.oracle.com/CSP/ui/index.html>.

—. "Oracle Call Interface Programmer's Guide." *Oracle Corporation*. 2008.  
[http://download.oracle.com/docs/cd/B28359\\_01/apdev.111/b28395/toc.htm](http://download.oracle.com/docs/cd/B28359_01/apdev.111/b28395/toc.htm).

Wikipedia. *Clarke's three laws*. Dec 17, 2008.  
[http://en.wikipedia.org/wiki/Clarke%27s\\_three\\_laws](http://en.wikipedia.org/wiki/Clarke%27s_three_laws).

—. *Radium*. Dec 18, 2008.  
<http://en.wikipedia.org/wiki/Radium>.

## 15 ABOUT THE AUTHOR

Cary Millsap is well-known in the global Oracle community as a speaker, educator, consultant, and writer. He is the founder and president of Method R Corporation (<http://method-r.com>), a small company devoted to genuinely satisfying software performance. Method R offers consulting services, education courses, and software tools—including the Method R Profiler—that help you optimize your software performance.

Cary is the author (with Jeff Holt) of *Optimizing Oracle Performance* (O'Reilly), for which he and Jeff were named *Oracle Magazine's* 2004 Authors of the Year. He is also a contributor to *Oracle Insights: Tales of the Oak Table* (Apress). He is the former Vice President of

Oracle Corporation's System Performance Group, and a co-founder of Hotsos. Cary is also an Oracle ACE Director and a founding partner of the Oak Table Network, an informal association of "Oracle scientists" that are well known throughout the Oracle community. Cary blogs at <http://carymillsap.blogspot.com>.

## 16 REVISION HISTORY

2009/02/09: Added footnote 4. Thank you, Michael Thomas for pointing out this omission.

## 17 APPENDIX: CREATING AND FINDING ORACLE TRACE FILES

This section describes how to create and find your own Oracle extended SQL trace files.

### 17.1 Turning the Trace On and Off

You can turn tracing on or off by executing an Oracle function call from your application. For Oracle Database 10<sup>g</sup> and beyond, you can turn tracing on and off with calls to functions in the standard Oracle package called *dbms\_monitor*:

```
dbms_monitor.session_trace_enable(null,null,true,true)
dbms_monitor.session_trace_disable(null,null)
```

There are other ways to do it, too, but this one is simple and secure. It *will* require the person acting as your database administrator to grant you permission to execute the *dbms\_monitor* package.

If you're using an Oracle Database version prior to 10<sup>g</sup>, then ask your database administrator to install the Oracle package called *dbms\_support*. There are instructions about how to do it at the My Oracle Support web site (Oracle Corporation n.d.).

### 17.2 Finding Your Trace File

When you've traced your code, you'll need to find your trace file. The process running your Oracle kernel code writes your trace files to the operating system directory named by an Oracle instance parameter. In Oracle version 11, the Oracle kernel will write your trace files into the "Diag Trace" directory, which you can identify by using the following SQL statement:

```
select * from v$diag_info where name='Diag Trace'
```

In older versions of Oracle, you can identify your trace file directory by using this SQL statement:

```
select * from v$parameter where name in
('user_dump_dest', 'background_dump_dest')
```

The *user\_dump\_dest* directory is where most of your trace files are probably going to show up. If you use Oracle parallel execution features, then you'll find some of your trace data in the *background\_dump\_dest* directory.

Different ports of Oracle use different naming conventions for trace files. Your trace file names will probably look something like one of the following:

```
xe_ora_10840.trc
prod7_23389_ora.trc
ora_1492_delta1.trc
ORA01215.trc
```

```
fin1_ora_11297_POSTING.trc
MERKUR_S7_FG_ORACLE_013.trc
```

Trace files may look different on different platforms and different versions of Oracle, but you can count on your trace file names containing some or all of the following elements:

- The string «ora»;
- Your Oracle instance name;
- Your Oracle kernel process id (on Microsoft Windows, it will be your process's thread id);
- If you set a *tracefile\_identifier* in your Oracle session, then the string value of that parameter; and
- The suffix «.trc».

If you're writing code that will connect to an Oracle instance that someone else manages, you'll need to coordinate with that person to get permissions to read your trace files.<sup>13</sup> Without access to your trace files, optimizing the code you write is going to be a *lot* more expensive for your company.

## 18 APPENDIX: THE IMPORTANT CODE PATHS

You can learn a lot in a ten-minute session looking at Oracle trace data, but what happens when you see a call name that you don't understand? This section contains some very brief starter advice that will guide you through the most common code paths that you'll see described in your trace files.

For calls not listed here, the best online sources are probably Google and oracle.com (in that order). However, you'll learn more reliably by learning to use a tool like *Dtrace* or *strace* (or *truss*, *scstrace*, *tusc*, ...whatever OS call tracing tool your platform gives you) to see what OS call each Oracle call name maps to. Once you're to that point, it's easy: there is lots of reliable documentation for OS calls available all over the place, all the way down to the source code level for many operating systems.

Here are some Oracle kernel code paths you're going to be seeing over and over again:

### EXEC

If your execution has a query component, see the *FETCH* entry. If you are executing a PL/SQL package with no real time consuming SQL statements within it, then use the Oracle

---

<sup>13</sup> Method R Corporation sells an extension for Oracle SQL Developer that automates the process of acquiring Oracle trace files for application developers.

*dbms\_profiler* package to identify which lines of PL/SQL are costing you the most time.

#### *FETCH*

Don't visit the buffer cache (the value *cr + cu* from your Oracle extended SQL trace data) more than 10 times per row returned per data source. For example, a 4-table join that returns 3 rows should have  $cr + cu \leq 120$ .

#### *PARSE*

A good application never parses (that is, *prepares*) a given SQL statement more than once per session. A *great* application never parses a given SQL statement more than once per Oracle instance startup. Parsing too often prevents an application from scaling to large user counts, no matter how many CPUs your system might happen to have. It's because parsing is a software-serialized operation.

#### *buffer busy waits*

Happens to your program when it tries to change an in-memory block in the database buffer cache, but some other process is in the midst of modifying that buffer. Fix the problem by working with your database administrator to make the most competed-for database blocks less interesting to so many concurrent Oracle sessions.

#### *db file scattered read*

Indicates a read of two or more Oracle blocks in a single *readv* OS call. Conventional advice instructs your database administrator to make the database buffer cache bigger when he sees lots of this kind of OS call. Better advice is to write more efficient queries that follow the rule-of-ten advice explained in the *FETCH* section.

#### *db file sequential read*

Normally indicates the read of a single Oracle block with a *pread* OS call. If you can eliminate unnecessary buffer cache visits as directed in the *FETCH* section, you'll naturally eliminate *db file sequential read* calls as well.

#### *enqueue*

Indicates that your code is trying to change a row that another Oracle session has locked. If you wrote the program that's holding the lock, then rewrite it to hold the lock for a smaller total duration. For example, don't allow any end-user data entry opportunities to occur between an *insert*, *update*, *delete*, or *merge* SQL statement and its subsequent *commit*. If you didn't write the code that's holding the lock, then find out who did.

#### *latch\** (call names with the word *latch* in them)

You'll see *shared pool* or *library cache* latch waits (maybe both of them) when you write code that

makes too many parse calls. You'll see *cache buffers chains* or *cache buffers lru chain* latch waits when you write inefficient SQL that visits the database buffer cache too many times. Don't do those things.

Note that in Oracle version 10 and beyond, the names of many latch-related OS calls each contains the name of the latch, as in *latch: cache buffers chains*. Prior to version 10, all latch-related OS calls were recorded under the call name *latch free*, and the type of the latch was listed as the value of the *p2* field in the *WAIT* line.

#### *log file sync*

Indicates *commit* call processing, which you can easily abuse when, for example, you write applications that use auto-commit.

#### *SQL\*Net message from client*

Indicates a network round-trip. Don't make unnecessary network round-trips. See the *PARSE* section for how to eliminate unnecessary parse calls and the associated round-trips. It's almost never a good idea to write applications that process only one database row at a time.

If you're sloppy in how you collect your trace data for interactive applications, then some of the time included in your *SQL\*Net message from client* durations will be time that the end user spends regarding the data just presented to him. This is a false negative performance indicator that you should fix by being more careful about how you collect your trace data. Scope your trace data collecting to include *only* the time that your end user is waiting on the application to perform some operation.

#### *SQL\*Net more data to client*

This is what you'll see when you crank up your array fetch size for programs that *select* a lot of rows. It's like a *SQL\*Net message to client*, except it's in the context of a single data transfer that's already in progress.

#### *unaccounted-for*

If you use a commercial profiling tool like the Method R Profiler (also sold as the Hotsos Profiler), the presence of *unaccounted-for* time almost always indicates time that your process has spent preempted by the operating system. Fix it by making your program (and the programs it competes against for CPU time) use as little CPU as possible. Eliminate unnecessary parse calls, and ensure that your SQL visits the database buffer cache as little as possible.

## 19 APPENDIX: TRACE FILES

This section contains excerpts from the raw Oracle extended SQL trace data referenced in this paper.

### 19.1 Slow Query

The following trace file excerpt shows trace data for the 23.405-second query program.

```
1. /usr/lib/oracle/xe/app/oracle/admin/XE/udump/xe_ora_9024_METHODDR_TESTING_.trc
2. Oracle Database 10g Express Edition Release 10.2.0.1.0 - Production
3. ORACLE_HOME = /usr/lib/oracle/xe/app/oracle/product/10.2.0/server
4. System name: Linux
5. Node name: oracle01.dev.method-r.com
6. Release: 2.6.25.6-27.fc8
7. Version: #1 SMP Fri Jun 13 16:38:52 EDT 2008
8. Machine: i686
9. Instance name: XE
10. Redo thread mounted by this instance: 1
11. Oracle process number: 18
12. Unix process pid: 9024, image: oracleXE@oracle01.dev.method-r.com
13.
14. *** SERVICE NAME:(SYS$USERS) 2008-11-25 11:12:29.682
15. *** SESSION ID:(26.22686) 2008-11-25 11:12:29.682
16. =====
17. PARSING IN CURSOR #2 len=69 dep=0 uid=55 oct=42 lid=55 tim=1198860497736569 hv=3164292706 ad='30e088a4'
18. alter session set events '10046 trace name context forever, level 12'
19. END OF STMT
20. EXEC #2:c=1000,e=91,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=1198860497736561
21. XCTEND rlbk=0, rd_only=1
22. WAIT #2: nam='SQL*Net message to client' ela= 2 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198860497737042
23. WAIT #2: nam='SQL*Net message from client' ela= 985 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198860497738097
24. =====
25. PARSING IN CURSOR #1 len=21 dep=0 uid=55 oct=3 lid=55 tim=1198860497738418 hv=3808180571 ad='2d45e9d0'
26. select * from sla_run
27. END OF STMT
28. PARSE #1:c=0,e=251,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=1198860497738414
29. BINDS #1:
30. EXEC #1:c=0,e=84,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=1198860497738645
31. WAIT #1: nam='SQL*Net message to client' ela= 2 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198860497738727
32. WAIT #1: nam='SQL*Net message from client' ela= 25227 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198860497764024
33. WAIT #1: nam='SQL*Net message to client' ela= 24 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198860497764438
34. FETCH #1:c=0,e=355,p=0,cr=4,cu=0,mis=0,r=10,dep=0,og=1,tim=1198860497764514
35. WAIT #1: nam='SQL*Net message from client' ela= 7851 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198860497772417
36. WAIT #1: nam='SQL*Net message to client' ela= 2 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198860497772500
37. FETCH #1:c=0,e=84,p=0,cr=1,cu=0,mis=0,r=10,dep=0,og=1,tim=1198860497772568
38. WAIT #1: nam='SQL*Net message from client' ela= 1306 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198860497773924
39. WAIT #1: nam='SQL*Net message to client' ela= 2 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198860497773985
40. FETCH #1:c=0,e=81,p=0,cr=1,cu=0,mis=0,r=10,dep=0,og=1,tim=1198860497774051
41. WAIT #1: nam='SQL*Net message from client' ela= 1282 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198860497775378
(42,746 lines here are not shown)
42. WAIT #1: nam='SQL*Net message to client' ela= 2 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198860521138223
43. FETCH #1:c=0,e=86,p=0,cr=1,cu=0,mis=0,r=7,dep=0,og=1,tim=1198860521138298
44. WAIT #1: nam='SQL*Net message from client' ela= 2141 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198860521140722
```

### 19.2 Improved Query

The following trace file excerpt shows trace data for the improved 2.400-second insert program.

```
1. /usr/lib/oracle/xe/app/oracle/admin/XE/udump/xe_ora_28638_QUERY_2048.trc
2. Oracle Database 10g Express Edition Release 10.2.0.1.0 - Production
3. ORACLE_HOME = /usr/lib/oracle/xe/app/oracle/product/10.2.0/server
4. System name: Linux
5. Node name: oracle01.dev.method-r.com
6. Release: 2.6.25.6-27.fc8
7. Version: #1 SMP Fri Jun 13 16:38:52 EDT 2008
8. Machine: i686
9. Instance name: XE
10. Redo thread mounted by this instance: 1
11. Oracle process number: 18
12. Unix process pid: 28638, image: oracleXE@oracle01.dev.method-r.com
13.
```

```

14. *** SERVICE NAME:(SYS$USERS) 2008-12-10 09:51:02.061
15. *** SESSION ID:(26.27749) 2008-12-10 09:51:02.061
16.
17. PARSING IN CURSOR #2 len=69 dep=0 uid=55 oct=42 lid=55 tim=1200121349669845 hv=3164292706 ad='30d22bc0'
18. alter session set events '10046 trace name context forever, level 12'
19. END OF STMT
20. EXEC #2:c=0,e=59,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=1200121349669837
21. XCTEND rlbk=0, rd_only=1
22. WAIT #2: nam='SQL*Net message to client' ela= 1 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1200121349670096
23. WAIT #2: nam='SQL*Net message from client' ela= 869 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1200121349671014
24.
25. PARSING IN CURSOR #1 len=21 dep=0 uid=55 oct=3 lid=55 tim=1200121349671182 hv=3808180571 ad='2d7dd08c'
26. select * from sla_run
27. END OF STMT
28. PARSE #1:c=1000,e=121,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=1200121349671176
29. BINDS #1:
30. EXEC #1:c=0,e=57,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=1200121349671305
31. WAIT #1: nam='SQL*Net message to client' ela= 3 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1200121349671351
32. WAIT #1: nam='SQL*Net message from client' ela= 5960 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1200121349677368
33. WAIT #1: nam='SQL*Net message to client' ela= 6 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1200121349677583
34. WAIT #1: nam='SQL*Net more data to client' ela= 149 driver id=1952673792 #bytes=2001 p3=0 obj#=-1 tim=1200121349677942
35. WAIT #1: nam='SQL*Net more data to client' ela= 7 driver id=1952673792 #bytes=2002 p3=0 obj#=-1 tim=120012134968167
36. WAIT #1: nam='SQL*Net more data to client' ela= 9 driver id=1952673792 #bytes=2000 p3=0 obj#=-1 tim=120012134968404
37. WAIT #1: nam='SQL*Net more data to client' ela= 7 driver id=1952673792 #bytes=2006 p3=0 obj#=-1 tim=1200121349687867
38. WAIT #1: nam='SQL*Net more data to client' ela= 8 driver id=1952673792 #bytes=1998 p3=0 obj#=-1 tim=1200121349688893
39. WAIT #1: nam='SQL*Net more data to client' ela= 10 driver id=1952673792 #bytes=2000 p3=0 obj#=-1 tim=1200121349693321
40. WAIT #1: nam='SQL*Net more data to client' ela= 7 driver id=1952673792 #bytes=2000 p3=0 obj#=-1 tim=120012134969641
41. WAIT #1: nam='SQL*Net more data to client' ela= 3402 driver id=1952673792 #bytes=2002 p3=0 obj#=-1 tim=1200121349683312
42. WAIT #1: nam='SQL*Net more data to client' ela= 132 driver id=1952673792 #bytes=2000 p3=0 obj#=-1 tim=1200121349683800
43. WAIT #1: nam='SQL*Net more data to client' ela= 121 driver id=1952673792 #bytes=2002 p3=0 obj#=-1 tim=1200121349684161
44. WAIT #1: nam='SQL*Net more data to client' ela= 8 driver id=1952673792 #bytes=2001 p3=0 obj#=-1 tim=1200121349684409
45. WAIT #1: nam='SQL*Net more data to client' ela= 10 driver id=1952673792 #bytes=2000 p3=0 obj#=-1 tim=1200121349684666
46. WAIT #1: nam='SQL*Net more data to client' ela= 6 driver id=1952673792 #bytes=2002 p3=0 obj#=-1 tim=1200121349684920
47. WAIT #1: nam='SQL*Net more data to client' ela= 828 driver id=1952673792 #bytes=2000 p3=0 obj#=-1 tim=1200121349685966
48. WAIT #1: nam='SQL*Net more data to client' ela= 123 driver id=1952673792 #bytes=2002 p3=0 obj#=-1 tim=1200121349686428
49. WAIT #1: nam='SQL*Net more data to client' ela= 68 driver id=1952673792 #bytes=2000 p3=0 obj#=-1 tim=1200121349686696
50. WAIT #1: nam='SQL*Net more data to client' ela= 7 driver id=1952673792 #bytes=2001 p3=0 obj#=-1 tim=1200121349686943
51. WAIT #1: nam='SQL*Net more data to client' ela= 8 driver id=1952673792 #bytes=2003 p3=0 obj#=-1 tim=1200121349687223
52. WAIT #1: nam='SQL*Net more data to client' ela= 6 driver id=1952673792 #bytes=1999 p3=0 obj#=-1 tim=1200121349687468
53. WAIT #1: nam='SQL*Net more data to client' ela= 9 driver id=1952673792 #bytes=2002 p3=0 obj#=-1 tim=1200121349687736
54. WAIT #1: nam='SQL*Net more data to client' ela= 6 driver id=1952673792 #bytes=2000 p3=0 obj#=-1 tim=1200121349688054
55. WAIT #1: nam='SQL*Net more data to client' ela= 2047 driver id=1952673792 #bytes=2002 p3=0 obj#=-1 tim=1200121349690352
56. WAIT #1: nam='SQL*Net more data to client' ela= 11 driver id=1952673792 #bytes=2000 p3=0 obj#=-1 tim=1200121349690646
57. WAIT #1: nam='SQL*Net more data to client' ela= 7 driver id=1952673792 #bytes=2001 p3=0 obj#=-1 tim=1200121349690905
58. WAIT #1: nam='SQL*Net more data to client' ela= 7 driver id=1952673792 #bytes=2001 p3=0 obj#=-1 tim=1200121349691160
59. WAIT #1: nam='SQL*Net more data to client' ela= 8 driver id=1952673792 #bytes=2006 p3=0 obj#=-1 tim=1200121349691597
60. WAIT #1: nam='SQL*Net more data to client' ela= 8 driver id=1952673792 #bytes=1997 p3=0 obj#=-1 tim=1200121349691851
61. WAIT #1: nam='SQL*Net more data to client' ela= 7 driver id=1952673792 #bytes=2001 p3=0 obj#=-1 tim=1200121349692248
62. WAIT #1: nam='SQL*Net more data to client' ela= 7 driver id=1952673792 #bytes=2002 p3=0 obj#=-1 tim=1200121349692507
63. WAIT #1: nam='SQL*Net more data to client' ela= 5125 driver id=1952673792 #bytes=2003 p3=0 obj#=-1 tim=1200121349697850
64. WAIT #1: nam='SQL*Net more data to client' ela= 11 driver id=1952673792 #bytes=1997 p3=0 obj#=-1 tim=1200121349698254
65. WAIT #1: nam='SQL*Net more data to client' ela= 9 driver id=1952673792 #bytes=2005 p3=0 obj#=-1 tim=1200121349698541
66. WAIT #1: nam='SQL*Net more data to client' ela= 7 driver id=1952673792 #bytes=2003 p3=0 obj#=-1 tim=1200121349698782
67. WAIT #1: nam='SQL*Net more data to client' ela= 10 driver id=1952673792 #bytes=1999 p3=0 obj#=-1 tim=1200121349699020
68. WAIT #1: nam='SQL*Net more data to client' ela= 10 driver id=1952673792 #bytes=1997 p3=0 obj#=-1 tim=1200121349699464
69. FETCH #1:c=11998,e=22206,p=0,cr=16,cu=0,mis=0,r=2048,dep=0,og=1,tim=1200121349699630
(2,859 lines here are not shown)
70. FETCH #1:c=6999,e=6837,p=0,cr=16,cu=0,mis=0,r=1205,dep=0,og=1,tim=1200121352051111
71. WAIT #1: nam='SQL*Net message from client' ela= 18557 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1200121352069816

```

### 19.3 Slow Insert

The following trace file excerpt shows trace data for the slow 32.706-second insert program.

```

1. /usr/lib/oracle/xe/app/oracle/admin/XE/udump/xe_ora_9835_METHODR_TESTING_.trc
2. Oracle Database 10g Express Edition Release 10.2.0.1.0 - Production
3. ORACLE_HOME = /usr/lib/oracle/xe/app/oracle/product/10.2.0/server
4. System name: Linux
5. Node name: oracle01.dev.method-r.com
6. Release: 2.6.25.6-27.fc8
7. Version: #1 SMP Fri Jun 13 16:38:52 EDT 2008
8. Machine: i686

```

```

9. Instance name: XE
10. Redo thread mounted by this instance: 1
11. Oracle process number: 21
12. Unix process pid: 9835, image: oracleXE@oracle01.dev.method-r.com
13.
14. *** SERVICE NAME:(SYS$USERS) 2008-11-25 11:39:16.895
15. *** SESSION ID:(24.25822) 2008-11-25 11:39:16.895
16. =====
17. PARSING IN CURSOR #3 len=69 dep=0 uid=55 oct=42 lid=55 tim=1198862067281227 hv=3164292706 ad='30e088a4'
18. alter session set events '10046 trace name context forever, level 12'
19. END OF STMT
20. EXEC #3:c=0,e=71,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,tim=1198862067281220
21. XCTEND rlbk=0, rd_only=1
22. WAIT #3: nam='SQL*Net message to client' ela= 2 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198862067281755
23. WAIT #3: nam='SQL*Net message from client' ela= 3633 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198862067285447
24. =====
25. PARSING IN CURSOR #2 len=40 dep=0 uid=55 oct=2 lid=55 tim=1198862067286178 hv=223277221 ad='30d87524'
26. insert into JAVA_TEST_TABLE values ('0')
27. END OF STMT
28. PARSE #2:c=0,e=678,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,tim=1198862067286173
29. BINDS #2:
30. EXEC #2:c=2000,e=1245,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=1,tim=1198862067287560
31. XCTEND rlbk=0, rd_only=0
32. WAIT #2: nam='SQL*Net message to client' ela= 2 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198862067288169
33. WAIT #2: nam='SQL*Net message from client' ela= 1154 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198862067289370
34. =====
35. PARSING IN CURSOR #2 len=40 dep=0 uid=55 oct=2 lid=55 tim=1198862067289787 hv=1228865179 ad='2d6c3ea4'
36. insert into JAVA_TEST_TABLE values ('1')
37. END OF STMT
38. PARSE #2:c=0,e=302,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,tim=1198862067289781
39. BINDS #2:
40. EXEC #2:c=0,e=202,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=1,tim=1198862067290111
41. XCTEND rlbk=0, rd_only=0
42. WAIT #2: nam='log file sync' ela= 113 buffer#=737 p2=0 p3=0 obj#=-1 tim=1198862067290479
43. WAIT #2: nam='SQL*Net message to client' ela= 2 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198862067290546
44. WAIT #2: nam='SQL*Net message from client' ela= 1918 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198862067292503
45. =====
46. PARSING IN CURSOR #2 len=40 dep=0 uid=55 oct=2 lid=55 tim=1198862067292826 hv=2978669578 ad='2d75ac84'
47. insert into JAVA_TEST_TABLE values ('2')
48. END OF STMT
49. PARSE #2:c=0,e=232,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,tim=1198862067292821
50. BINDS #2:
51. EXEC #2:c=0,e=146,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=1,tim=1198862067293093
52. XCTEND rlbk=0, rd_only=0
53. WAIT #2: nam='log file sync' ela= 107 buffer#=739 p2=0 p3=0 obj#=-1 tim=1198862067293425
54. WAIT #2: nam='SQL*Net message to client' ela= 2 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198862067293477
55. WAIT #2: nam='SQL*Net message from client' ela= 2050 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1198862067295563
(106,315 lines here are not shown)

```

## 19.4 Improved Insert

The following trace file excerpt shows trace data for the improved 0.154-second insert program.

```

1. /usr/lib/oracle/xe/app/oracle/admin/XE/udump/xe_ora_3807_INSERT_TEST.trc
2. Oracle Database 10g Express Edition Release 10.2.0.1.0 - Production
3. ORACLE_HOME = /usr/lib/oracle/xe/app/oracle/product/10.2.0/server
4. System name: Linux
5. Node name: oracle01.dev.method-r.com
6. Release: 2.6.25.6-27.fc8
7. Version: #1 SMP Fri Jun 13 16:38:52 EDT 2008
8. Machine: i686
9. Instance name: XE
10. Redo thread mounted by this instance: 1
11. Oracle process number: 22
12. Unix process pid: 3807, image: oracleXE@oracle01.dev.method-r.com
13.
14. *** SERVICE NAME:(SYS$USERS) 2008-12-22 16:37:52.283
15. *** SESSION ID:(24.34335) 2008-12-22 16:37:52.283
16. =====
17. PARSING IN CURSOR #2 len=69 dep=0 uid=55 oct=42 lid=55 tim=1201157687776660 hv=3164292706 ad='28e47760'
18. alter session set events '10046 trace name context forever, level 12'
19. END OF STMT

```

```

20. EXEC #2:c=0,e=53,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=1201157687776652
21. XCTEND r1bk=0, rd_only=1
22. WAIT #2: nam='SQL*Net message to client' ela= 2 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1201157687776907
23. WAIT #2: nam='SQL*Net message from client' ela= 60441 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1201157687837388
24.
25. PARSING IN CURSOR #1 len=39 dep=0 uid=55 oct=2 lid=55 tim=1201157687837622 hv=2094431222 ad='28f82278'
26. Insert into JAVA_TEST_TABLE values (:1)
27. END OF STMT
28. PARSE #1:c=0,e=160,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=1201157687837613
29. BINDS #1:
30. kkscoacd
31. Bind#0
32. oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
33. oacflg=03 fl2=1000000 frm=01 csi=178 siz=24 off=0
34. kxsbbbfp=b7f27e2c bln=22 avl=01 flg=05
35. value=0
36. WAIT #1: nam='SQL*Net more data from client' ela= 27 driver id=1952673792 #bytes=3 p3=0 obj#=-1 tim=1201157687844067
37. WAIT #1: nam='SQL*Net more data from client' ela= 11 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1201157687844425
38. EXEC #1:c=7999,e=7941,p=0,cr=396,cu=499,mis=0,r=1000,dep=0,og=1,tim=1201157687845652
39. WAIT #1: nam='SQL*Net message to client' ela= 3 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1201157687845745
40. WAIT #1: nam='SQL*Net message from client' ela= 4885 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1201157687850661
41. BINDS #1:
42. kkscoacd
43. Bind#0
44. oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
45. oacflg=03 fl2=1000000 frm=01 csi=178 siz=24 off=0
46. kxsbbbfp=b7f27e2c bln=22 avl=02 flg=05
47. value=1000
48. WAIT #1: nam='SQL*Net more data from client' ela= 17 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1201157687851385
49. WAIT #1: nam='SQL*Net more data from client' ela= 14 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1201157687852578
50. EXEC #1:c=1999,e=2057,p=0,cr=131,cu=169,mis=0,r=1000,dep=0,og=1,tim=1201157687852764
(83 lines here are not shown)
51. EXEC #1:c=2999,e=2397,p=0,cr=236,cu=299,mis=0,r=1000,dep=0,og=1,tim=1201157687882375
52. WAIT #1: nam='SQL*Net message to client' ela= 2 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1201157687882418
53. WAIT #1: nam='SQL*Net message from client' ela= 1841 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1201157687884288
54. BINDS #1:
55. kkscoacd
56. Bind#0
57. oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
58. oacflg=03 fl2=1000000 frm=01 csi=178 siz=24 off=0
59. kxsbbbfp=b7f27e2c bln=22 avl=02 flg=05
60. value=9000
61. WAIT #1: nam='SQL*Net more data from client' ela= 17 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1201157687884646
62. WAIT #1: nam='SQL*Net more data from client' ela= 12 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1201157687885716
63. EXEC #1:c=1000,e=1539,p=0,cr=118,cu=153,mis=0,r=1000,dep=0,og=1,tim=1201157687885866
64. WAIT #1: nam='SQL*Net message to client' ela= 3 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1201157687885907
65. WAIT #1: nam='SQL*Net message from client' ela= 398 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1201157687886334
66. XCTEND r1bk=0, rd_only=0
67. WAIT #0: nam='log file sync' ela= 39869 buffer#=376 p2=0 p3=0 obj#=-1 tim=1201157687927436
68. WAIT #0: nam='SQL*Net message to client' ela= 4 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1201157687927533
69. WAIT #0: nam='SQL*Net message from client' ela= 1743 driver id=1952673792 #bytes=1 p3=0 obj#=-1 tim=1201157687929301
70. XCTEND r1bk=0, rd_only=1

```