Mastering Performance with Oracle Extended SQL Trace

Cary Millsap
Method R Corporation, Southlake, Texas, USA
http://twitter.com/CaryMillsap
cary.millsap@method-r.com

Revised 2011/02/23
The most recent update of this paper is available free of obligation at http://method-r.com/downloads.

Oracle's extended SQL trace data stream contains a linear sequential record of every database call and every operating system call that the Oracle kernel executes in response to the code that you write. The trace file contains timings that enable you to precisely measure how long your code takes to run and why. The detailed information about individual database calls and operating system calls is vital to building scalable applications (as well as troubleshooting them in production). The information in there is a gold mine, and some of it is available nowhere else.

TABLE OF CONTENTS

1. Performance and Tracing 1
2. Why You Should Trace 2
3. What Happens When You Trace 2
4. Controlling the Trace 2
5. Session Tracing 3
6. End-to-End Tracing 4
7. End-to-End Prerequisites 4
8. Finding Your Trace File 5
9. Performance Intrusion 6
10. Tracing in Multiplexed Architectures 6
11. Tracing While You Develop 7
12. Trace File Guided Tour 7
13. Trace File Alphabet 9
14. Trace File Alphabet: DB Calls 9
15. Trace File Alphabet: OS Calls 10
16. Trace File Flow 10
17. Trace File Arithmetic: Part 1 10
18. Trace File Arithmetic: Part 2 11
19. Trace File Arithmetic: Part 3 11
20. Tracing Traps 11
21. Commercial Trace File Tools 12
22. Acknowledgments 12
23. About the Author 12
24. Revision History 13
25. Appendix: "Hello World" Trace 14

1 PERFORMANCE AND TRACING

You click a button on a web form. It's a button that you click all the time. Normally, the system you're using responds to that click almost instantaneously, but now… Now, it's taking more than 5 seconds every time you do it, and sometimes it takes more than 10. The obvious and very simple question is: *why*?

The vast majority of people who experience a problem like this never find out the answer.

Think for a minute about what the answer ought to look like. If you're a IT professional, then you've probably seen so many reports that the simple answer doesn't even occur to you. Here's what *I* want: I want a report telling me where *my* time went.

I don't want to see how busy the whole system was when my click took 5 seconds; I don't want to see all the "bad" SQL that executed at or near the time when I had my problem; and I don't want to see all the different kinds of waiting that the system was doing when I clicked. I just want to see where *my* time went, untainted by information about anything not directly relevant to *my* response time.

With Oracle, you can have it. The feature that gives it to you is included with every release and every edition of the Oracle Database, from Express Edition (XE) to Enterprise Edition (EE), since 1992. You don't have to

© 2011 Method R Corporation. All rights reserved.

buy an Option or a Pack to use it. It's called *extended SQL trace*.

2 WHY YOU SHOULD TRACE

Oracle extended SQL trace helps you diagnose, it helps you learn, and it helps you debug:[1]

- System administrators, diagnosticians, and trouble-shooters will find it invaluable for solving problems with programs for which the source is not readily available (or not easily navigable) since they do not need to be recompiled in order to trace them.
- Students, hackers, and the overly-curious will find that a great deal can be learned about a system and its system calls by tracing even ordinary programs.
- And programmers will find trace data very useful for bug isolation, sanity checking, and performance optimization. Some problems cannot be solved without it.

What makes tracing so good compared to the other performance information you can get from Oracle is that with tracing, you get a linear sequential record of everything your Oracle kernel process does while it's trying to compute the answer you've asked for. You don't have to worry about gaps in your data (like with Oracle's Active Session History) or figuring out how to isolate your process's work in the huge picture of everything the system is doing right now (like with some of Oracle's *v$* fixed views and all of the GUI tools that use them).

With trace data, you see every database call your application makes to the Oracle kernel, and you see every system call your kernel makes to its host operating system. And it's the perfect tool for developers, because you don't have to have loads of complicated permissions to use it. I believe you can learn more about Oracle performance with trace data in a day than you can learn studying *v$* data for a month, because trace data so directly maps to the reality of how the user, the application code, and the database interact.

Trace files are the foundation of almost everything you need to know about Oracle performance.

3 WHAT HAPPENS WHEN YOU TRACE

Tracing is an attribute of an *Oracle session*.[2] For each session with trace enabled, any *Oracle kernel process*[3] that executes instructions for that session writes trace data for that session to the process's currently open trace file. Some facts:

- Tracing is a session attribute.
- But Oracle kernel processes are what write your trace data.
- Since more than one process can execute instructions for a given Oracle session (for example, by using Oracle Parallel Execution features, or by enabling Oracle Shared Server), your trace data can appear in more than one file.
- The more exotic your architecture (for example, the more tiers you use, the more pooling features you use), the more difficult it may be to map your trace data to a specific user's request.

Exhibit 1 shows a simple case in which a user traces an Oracle session connected through *sqlplus*, causing an Oracle kernel process to write trace data to a *.trc* file.

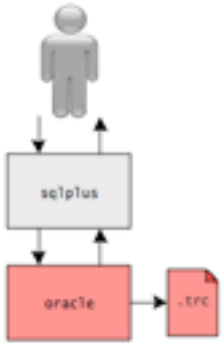

Exhibit 1. The Oracle kernel process, to which a user connects through some type of application, is the process that writes the trace file.

4 CONTROLLING THE TRACE

You can trace any Oracle session on your system if you have permission to execute the standard Oracle PL/SQL package called *dbms_monitor*. This package gives

---

[1] Oracle extended SQL trace is useful for the same kinds of reasons that the Unix *strace* tool is useful. My description here is paraphrased from the *strace* manual page at *http://linuxcommand.org/man_pages/strace1.html*.

[2] An Oracle session is an entity with a unique *sid-serial#* combination within an Oracle instance, which you can see in *v$session*.

[3] An Oracle kernel process is an operating system process (which you can inspect with, for example, a Linux command like *ps*, *lsof*, or *strace*) that executes the instructions required by an Oracle session.

© 2011 Method R Corporation. All rights reserved.

you a lot of control over which sessions you trace. Here are some of its procedures:

session_trace_enable
: Enable tracing for either the database session from which the procedure is called, or a specific session identified by its session identifier and serial number.

database_trace_enable
: Enable tracing for a specific instance or a whole database.

Oracle provides a *disable* procedure to match each *enable* procedure.

When you use *dbms_monitor*, I recommend that you use the default value of `waits=>true`. If you use the value `waits=>false`, you'll be using SQL trace, but not extended SQL trace. You'll get a trace of the Oracle database calls your application makes, but not the OS calls that Oracle makes. Without a record of those OS calls in your trace file, you'll not be able to account for potentially significant amounts of response time.

I also recommend that you use the non-default value `binds=>true`. Setting `binds=>true` causes the Oracle Database to emit the values that the application bound to its placeholder variables into the trace data stream. For example, if someone has executed this SQL statement:

```
update salary set amount = :v1 where id = :v2
```

…then your trace data will show what values were assigned to *v1* and *v2*. This is invaluable information for creating reproducible test cases for solving problems. However, be aware of the data security implication of emitting bind data into a trace file; when you do it, you are copying potentially sensitive data from your database to your file system.

5 SESSION TRACING

One of the nice things about Oracle tracing is that you're not required to change an application in order to use it. Even if the application you're using is completely un-modifiable and was in no way designed to cooperate with tracing, you still have the following very powerful tools at your disposal:

dbms_monitor.session_trace_enable
: With *session_trace_enable*, you can trace any session on your system if you know its session id and serial number. You just have to be meticulous to enable the trace immediately before the user issues the slow request, and then disable the trace immediately after the slow request is fulfilled.

Triggers
: For fast-moving sessions that don't give you time to identify their session ids before they begin doing work (like that Pro*C program that runs for 20 seconds, but that should run in 2), triggers are a convenient tool for inserting code to enable tracing. You can, for example, create an *after logon* trigger whose body uses:

```
session_trace_enable(
  session_id => null,
  serial_num => null,
  …
)
```

This code will enable tracing for the session that's logging on. There's no need for a corresponding *before logoff* trace disablement, by the way, because logging off automatically disables the trace.

sys_context
: The *sys_context* function gives you dozens of hooks you can use inside of a trigger body to identify the session that's logging in. For example,

```
sys_context('USERENV','IP_ADDRESS')
```

…returns the IP address of the machine from which the client is connected. With *sys_context*, you can write PL/SQL code in your *after logon* trigger body that can decide to trace only logins originating from a specific IP address, only logins requested by certain client OS users, or logins matching any of the other criteria that *sys_context* makes available to you.

Running suspicious SQL in your own session
: There are lots of ways you can identify suspicious SQL statements: you'll find them in OEM, in your AWR or ADDM reports, or by querying *v$sqlstat*. Copy them and run them in SQL*Plus or Oracle SQL Developer. You can plug in whatever bind variable values you need and trace to your heart's content.[4]

dbms_monitor.database_trace_enable
: If all else fails, there's always the nuclear bomb option of tracing everything that happens over some time interval you choose. Beware, however: enabling trace for a whole instance or database

[4] …Subject, naturally, to constraints on whether you are permitted access to the objects manipulated by the statement, and so on. Also, please make sure that, before you execute a SQL statement that might change rows in the database, you do not have an *auto-commit* feature enabled.

© 2011 Method R Corporation. All rights reserved.

will only activate tracing for *new* sessions that log in after you enable the trace. It won't enable tracing for sessions that are already logged in.

6 END-TO-END TRACING

There are three major milestones in the history of tracing in the Oracle Database:

- In 1988, Oracle version 6 introduced *SQL trace*, which let us discover the start and end time of every database call an application executed.
- In 1993, Oracle version 7 introduced *extended* SQL trace, which let us discover the duration of every system call the database executed.
- In 2003, Oracle version 10 introduced *end-to-end* extended SQL trace, which let us activate tracing, not just for sessions whose session ids we knew, but for sessions with specific attribute values.

The third milestone—end-to-end tracing—makes it possible for you to specify the *intent* to trace a business task in advance of that task ever running. Then, the Oracle Database will automatically enable tracing for any session that executes the task you have specified.

The way you specify the standing order to trace certain tasks is with the following *dbms_monitor* procedures:

client_id_trace_enable
: Enable tracing for any database session whose client identifier matches the one you specify.

serv_mod_act_trace_enable
: Enable tracing for any database session whose service, module, and action names match the ones you specify.

For example, if you wanted to trace any user whose client id is `192.168.1.104`, you'd use

```
dbms_monitor.client_id_trace_enable(
  client_id => '192.168.1.104',
  waits     => true,
  binds     => true,
  plan_stat => 'ALL_EXECUTIONS'
)
```

…And that's that. No matter when the next person logged in from 192.168.1.104, his work would be traced. If you wanted to trace any business task whose service is called `FINAPPS`, whose module is called `GLPPOS`, and whose action is called `Concurrent Request`, you'd execute:

```
dbms_monitor.serv_mod_act_trace_enable(
  service_name => 'FINAPPS',
  module_name  => 'GLPPOS',
  action_name  => 'Concurrent Request'
  waits        => true,
  binds        => true,
  plan_stat    => 'ALL_EXECUTIONS'
)
```

…Then you'd be set to trace your next several GL Posting runs.

Before end-to-end tracing, the hardest thing about tracing was getting a trace file that represents exactly the business task that the user is experiencing right there at her keyboard. When you trace a session based on its session id, you almost always capture either more trace data than you really want, or not enough. That's because it takes you a little time to identify the session's id before you can execute the *session_trace_enable* call.

However, when you activate tracing for a specific client id or service-module-action name combination, you create a standing order to Oracle, instructing the kernel to begin tracing any session that changes its client id to your specific value, or whenever it changes its server-module-action name combination to your specific values. And then Oracle will stop tracing immediately when that session changes its identifying information to something different.

Pure magic.

7 END-TO-END PREREQUISITES

But there's a catch with end-to-end tracing. How can *dbms_monitor* know a session's client id or service-module-action names? The answer: your application has to provide those attribute values. If your application doesn't name its tasks or set the client id attribute for its sessions in a useful way, then you can't use the *dbms_monitor* end-to-end tracing features.

If you're using an off-the-shelf application, your only recourse may be to live without the end-to-end tracing features (though you can still use the techniques described in section 5) and lobby your vendor to instrument a future version of your application in the manner I'm about to describe. If you can change your application source code yourself, instrumenting for end-to-end tracing is not hard to do. The only thing you really have to do is have each session set what Oracle calls its *user session handle attributes*. There are several ways to do it.

One way you can set those session handle attributes for your code is to use these standard Oracle PL/SQL packages:

© 2011 Method R Corporation. All rights reserved.

dbms_session
: The *set_identifier* procedure lets you set the calling session's client identifier.

dbms_application_info
: The *set_action* procedure lets you set the calling session's action name, and the *set_module* procedure lets you set the calling session's module and action names together.

If you prefer a simpler approach to setting your session handle attributes, you can use the Instrumentation Library for Oracle (ILO), a free open-source project that Method R Corporation maintains at SourceForge. ILO gives you PL/SQL packages that make it dead simple for an application developer to define where in an application's code path each business task begins and ends. It sets all the "hooks" that your systems DBAs will need later to trace specific business tasks by name. ILO gives you several extra features as well, which are detailed in the project's documentation.

If it bothers you to insert additional database calls into your application (and if you're building an application that will serve thousands of database requests per second, it should), then you should choose another option. Some Oracle APIs allow you to pass service-module-action names and a client-id from the application to the database as an *attribute* of each of the database calls that you were already determined to make.

The Oracle user session handle attributes you'll need to set are these:

```
OCI_ATTR_CLIENT_IDENTIFIER
OCI_ATTR_SERVICE
OCI_ATTR_MODULE
OCI_ATTR_ACTION
```

You can set them, for example, with the Oracle Call Interface (OCI) *OCIAttrSet* function, or with the JDBC *setEndToEndMetrics* method. These are client-side function calls that don't require a round-trip to the database. Once you have made a call to set your user session handle attributes, each database call that your application code makes after that is marked with the identifying information that *dbms_monitor* needs.

Setting your Oracle user session handle attributes yourself enables you to instrument your application with no extra database calls at all, which means no extra network round-trips between the application and the database. It's the most scalable way to instrument your code.

To summarize, you can instrument your application code for end-to-end tracing by calling standard Oracle PL/SQL procedures (or calling them through someone else's package, like ILO), or you can do it with no extra database calls by setting session handle attributes directly with an Oracle API like OCI or JDBC.

If your application is not designed to set these attributes for your sessions, then you won't be able to use the *client_id_trace_enable* or *serv_mod_act_trace_enable* procedures in *dbms_monitor*. You'll have to use only the *session_trace_enable* and *database_trace_enable* procedures instead, until you can convince the people who built your application to instrument the application so that you can measure and manage its performance while you operate it. Until then, not only will you not be able to trace as flexibly, you also won't be able to group application performance statistics using other applications like Oracle Enterprise Manager.

8 FINDING YOUR TRACE FILE

Once you've traced your code, you'll need to find your trace file. The process running your Oracle kernel code writes your trace file to the operating system directory named by an Oracle instance parameter. In Oracle version 11, the Oracle kernel will write your files into a directory that 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 directories by using this statement:

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

In Oracle version 10 and prior, the *user_dump_dest* directory is probably where most of your trace files will be. If you use Oracle parallel execution features, then you'll find some of your trace files 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 on different versions of Oracle, but you can count on your trace file names containing at least some of the following elements:

- The string `ora`.

© 2011 Method R Corporation. All rights reserved.

- 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* within your Oracle session (with an *alter session* command), then the string value of that parameter.
- 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. Without access to your trace files, optimizing the code you write is going to be a *lot* more expensive for your company.

9 PERFORMANCE INTRUSION

One thing that everyone always wants to know is, "What is the performance penalty of tracing?" People get a little nervous about all those extra timer calls that their Oracle kernel processes must execute when you turn trace on.

What they don't realize is that their Oracle kernel process is making all those timer calls anyway, whether tracing is activated or not. How else could it publish the timing information you see in fixed views like *v$sesstat*, *v$session_event*, and *v$session_wait*? The actual performance intrusion of tracing is the cost of the additional I/O calls required to persist to your trace file the information Oracle is already collecting.

Most people should trace only a few sessions at a time. If you don't have specialized tools, you'll never in your life be able to sift through all the trace files that an Oracle Database system can generate in a good, busy hour of work. If you want to trace a lot of sessions at once (and there are legitimate reasons for wanting to), you can trace as many sessions as your hardware capacity will allow. If you trace infrequently, then you don't have to be too careful about your trace file destination. If you trace a lot of sessions simultaneously, then just make sure you write your trace files to a drive array that has all the storage capacity *and* throughput capacity you need, and then trace all you want.

Unless you're using a version of Oracle that has a trace-related performance bug (Oracle bug 3009359, introduced in version 9.2.0.2, was a famous one), your program will probably run with ±5% of the duration that it takes to run it without tracing.[5] If a program's response time gets more than 10% worse when you trace it, then you should check whether your I/O subsystem to which you're writing trace files is ok, check whether your session's setting of *statistics_level* is what you want it to be, and check your Oracle Support database for bugs.

People who ask about the performance intrusion of tracing generally ask because they're fearful that they'll get into trouble if they do anything that might make performance worse than it already is. However, your goal in tracing is to learn how much faster a program ought to be running. Isn't that information worth at least as much as a one-time response time penalty induced by a trace? I've seen people work on problems for months on end without trace data, when they could have cleared everything up with just a single one-time trace. Even if tracing imposed five times the performance penalty that it does, I wouldn't use it any less.

10 TRACING IN MULTIPLEXED ARCHITECTURES

I mentioned in section 3 that tracing is an attribute of the Oracle session. However, many systems these days have many layers of software between the user and the Oracle session, and many of those layers employ pooling technologies that conceal the user's identity from her Oracle session.

The solution, in general, is this: where there's a will, there's a way. If you are committed to tracing a specific user's specific business task, it can be done. Here are some of the tools you can use to do it:

User session handle attributes
: The service, module, action, and client id attributes are the right and proper cure for tracing in pooled architectures. When the application identifies its users to the Oracle database, the *dbms_monitor* package lets you enable tracing for exactly the tasks you want traced, and then tools like Oracle's *trcsess* help you do the work of concatenating the various fragments of trace data that will describe how your task spent its time.

Session isolation
: If your application doesn't set its session handle attributes that allow you to distinguish the business task you're interested in from the thousands of other business tasks running at the same time, then isolate your user into a configuration where you can make the distinction.

[5] That's right, plus or minus. Sometimes a program will run faster when you trace it. It sounds nice, but it's usually not what you want.

© 2011 Method R Corporation. All rights reserved.

For example, instead of your user connecting to her usual *http://www.app.com* site, where her task gets stripped of her identity as it enters some session pool or connection pool with a thousand other tasks, have her connect to *http://www.app.com:88*, where her task is the only task in a session pool or connection pool with a pool size of exactly one. You can trace every session connecting to the database through port 88, which, at the time you're doing your analysis, you know will be nobody else but the one user you're analyzing.

11 TRACING WHILE YOU DEVELOP

The most important time for a developer to trace is early in the software development lifecycle, where defects are cheaper and easier to fix than any other time in your project.[6] Fortunately, tracing is easiest in the development environment.

I can remember since my earliest days of working with Oracle having some kind of *sqlplus* script that will turn trace on or off when you call it. It could be as simple as two scripts *traceon.sql* and *traceoff.sql* that look like this:

```
$ cat traceon.sql
exec dbms_monitor.session_trace_enable(binds=>true);
$
$ cat traceoff.sql
exec dbms_monitor.session_trace_disable();
```

When you write your code, it works like this:

```
@traceon
-- Your code goes here
@traceoff
```

Nowadays, I use the MR Trace extension for Oracle SQL Developer. When MR Trace is enabled, every *Run Script* button click creates a trace file, which MR Trace automatically fetches for you from your database server to your workstation. It also lets you tag your local trace files to make them easy to find later.

The best thing a developer can do to promote high performance over an application's long lifespan is to make it easy to trace the application throughout its production lifecycle. Of course, you do this by coding the application to set its own user session handle attributes, as I've described in section 7.

Applications work better when the world works like this:

1. Application developers trace their SQL and PL/SQL code as they write it. They study their trace files to ensure that the code they're writing is efficient.
2. Developers become fluent in understanding what the trace files mean, which results in more efficient application code.
3. Application developers code applications to set their user session handle attributes.
4. Developers and database administrators participate together during integration testing to trace the application code (which is easy with the user session handle attributes all set) and study the trace files to ensure that the code is efficient under load and that there are no unwanted performance interdependencies among competing sessions.
5. Production system administrators (database administrators, for example) trace the application code periodically in production when the application is behaving well, to record a baseline of how each application task is supposed to perform (which is easy with the user session handle attributes all set).
6. During performance problem episodes, production operators trace problem business tasks (session handle attributes make this easy) and work together with application developers—who are already familiar with what the trace file for the given task normally looks like—to solve problems quickly.

With habitual tracing throughout a software lifecycle, applications tend to have fewer performance problems, and they tend to get fixed more quickly on the rare occasion when they do have a problem. They tend to evolve toward being fast, optimized systems that can last a long time. …Much like the Oracle Database, which is itself heavily performance instrumented.

12 TRACE FILE GUIDED TOUR

Once you have your trace file in hand, the real fun begins. Now you have a call-by-call description of what the Oracle Database did throughout your entire response time.

If your only interface to the Oracle Database has been SQL or PL/SQL executed through client software like *sqlplus* or Oracle SQL Developer, you may not know about the distinct steps that Oracle uses to process a

[6] Barry Boehm, 1981, *Software Engineering Economics*, Englewood Cliffs NJ: P T R Prentice Hall, p40.

© 2011 Method R Corporation. All rights reserved.

single statement. Exhibit 2 shows the steps from the perspective of the Oracle Call Interface.

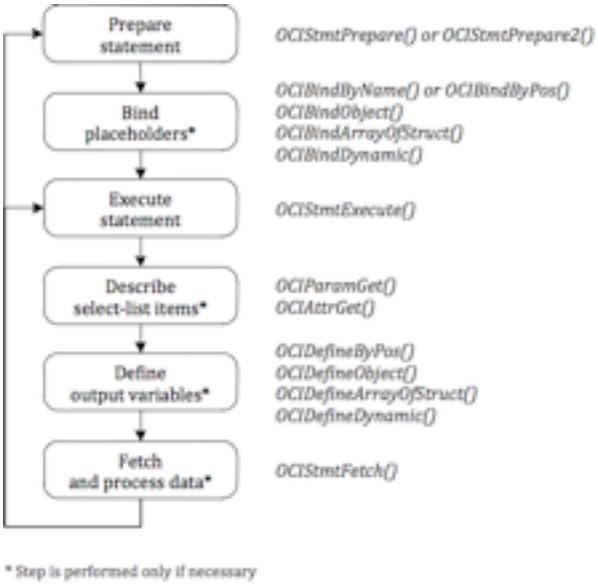

Exhibit 2. Steps in processing a SQL statement, from the OCI Programmer's Guide.

You can see this sequence of activity whenever you trace even a simple SQL statement like this:

```
select 'Hello world' from dual
```

An Oracle 11.2.0.2.0 trace file for an execution of this SQL statement is shown in its entirety in section 24. Here are some highlights.

On line 34, you can see the "Hello world" query itself. Lines 36–45 show how Oracle processed the query:

```
32. =====================
33. PARSING IN CURSOR #7285244 len=30 dep=0 uid=5 oct=3
    lid=5 tim=1294688002747560 hv=1604528588 ad='3d526cd8'
    sqlid='guthms1gu6afc'
34. select 'Hello world' from dual
35. END OF STMT
36. PARSE …,e=70,…
37. EXEC …,e=112,…
38. WAIT …'SQL*Net message to client' ela= 2
39. FETCH …,e=8,…,r=1,…
40. STAT …op='FAST DUAL (…)'
41. WAIT …'SQL*Net message from client' ela= 375
42. FETCH …,e=1,…,r=0,…
43. WAIT …'SQL*Net message to client' ela= 1
44. WAIT …'SQL*Net message from client' ela= 1627
45. XCTEND rlbk=0,…
```

Here, you can see the query being prepared (the *PARSE* line), which consumed 70 µs.[7] Next, it was executed (regarding Exhibit 2, there was no bind necessary, since my query didn't contain any placeholders), which consumed 112 µs. Then, on line 38, the Oracle kernel process issued an OS *write* call to send the result of the *prepare-execute* call pair back to the client (which Oracle calls *SQL*Net message to client*).

On line 39 is a *fetch* call, which consumed 8 µs and returned 1 row. Line 40 shows the execution plan that Oracle used to process my query. Line 41 shows the Oracle kernel blocked on an OS *read* call for 375 µs, awaiting another database call from the client (which Oracle calls *SQL*Net message from client*).

On line 42, the kernel has processed a second *fetch* call, consuming 1 µs and fetching no rows. Line 43 shows that the kernel issued another OS *write* call to send the zero-row result set back to the client (which of course informs the client that there are no more rows to be had from the query). Line 44 shows the kernel blocked for 1,627 µs on another OS *read* call, awaiting instructions for what to do next. Line 45 shows that the next action was a *commit*.

The remainder of the trace file (shown in its entirety in section 24) is interesting, because it shows something you might not have expected: the session inserted a row into *sys.aud$*. From this, I can tell that my instance is set up for session-level auditing. I didn't have to work my way down a checklist to figure that out; plus, my trace data will show me exactly the cost of that overhead upon my response time.

On line 48, you can see the SQL statement. Line 50 shows that the *prepare* call for that statement consumed 93 µs. On lines 51–156, you can see the values that were bound into the statement's placeholder variables (`:1`, `:2`, …, `:22`) within that statement. Line 157 shows that the *execute* call for the insert took 1,733 µs. Line 158 shows the execution plan for the insert, and line 159 shows that the *close* call for the insert took 5 µs. Line 160 shows that the OS call Oracle uses to implement its *log file sync* event consumed 153 µs, and the final two lines of the trace file show two more *close* calls, which consumed 9 µs and 30 µs.

I hope you can see from just this little tour how it is possible to use trace data to see exactly where your

[7] Call durations in trace files are expressed in microseconds, abbreviated µs. 1 µs = .000001 seconds, or 1/1,000,000th of a second.

© 2011 Method R Corporation. All rights reserved.

time has gone.[8] You just need to know how to read them. Let's begin with the alphabet.

13 TRACE FILE ALPHABET

Trace files look plenty ugly when you first meet them, but it's not too difficult to get a hang of what they mean. There are really only two types of lines in the trace file that tell you where time has gone:

Database call lines
: Database call lines begin with the keyword *PARSE*, *EXEC*, *FETCH*, *CLOSE*, *UNMAP*, or *SORT UNMAP*. Such a line indicates that an application client has made a database call, and that the Oracle Database has responded to it.

System call lines
: System call lines begin with the keyword *WAIT*. Such a line indicates that the Oracle kernel process has made an OS call, and that the OS has responded to it.

That's it; just two.

14 TRACE FILE ALPHABET: DB CALLS

With exception of the *CLOSE* call, which doesn't contain all the fields that the other database calls do, all contain the same fields. They look like this:

```
FETCH #12390720:c=8000,e=1734,p=1,cr=7,cu=0,mis=0,r=1,
dep=2,og=4,plh=3992920156,tim=1294438818783887
```

Here's what the fields mean:

`FETCH`
: This was a *fetch* call (not a *parse* or *exec* or some other kind).

`#12390720`
: The call's cursor handle id is 12390720. You can search backwards (toward the beginning) in your trace file to find the SQL or PL/SQL statement for which this call was made.

`c=8000`
: This call consumed 8,000 ±10,000 µs of CPU time. Yes, Oracle measured the call as having consumed [far] more CPU time than the call's actual elapsed duration.[9]

`e=1734`
: This call consumed 1,734 ±1 µs of elapsed time.

`p=1`
: This call obtained 1 Oracle block via OS read calls.

`cr=7`
: This call executed 7 "consistent mode" accesses upon the Oracle database buffer cache.

`cu=0`
: This call executed 0 "current mode" accesses upon the Oracle database buffer cache.

`mis=0`
: This call encountered 0 misses upon the Oracle library cache.

`r=1`
: This call returned 1 row.

`dep=2`
: There were 2 levels of database calls below this call in the database call stack. That is, this call is a child of another db call, which in turn is a child of another db call.

`og=4`
: This call's Oracle optimizer goal is *choose*. The four choices, in order, are *all*, *first*, *rule*, and *choose*.

`plh=3992920156`
: This call's execution plan hash value is 3992920156.

`tim=1294438818783887`
: This call completed at time 1,294,438,818,783,887 ±1 µs. The *tim* value is given in µs, but the answer to "µs since *what*?" varies by platform. All that really matters is for the *tim* values to be consistent within a given trace file, which they usually are.[10]

---

[8] You can see another trace file guided tour in "For Developers: Making Friends with the Oracle Database for Fast, Scalable Applications."

[9] For a detailed description of why the CPU consumption statistic is accurate to only ±10,000 µs , see *Optimizing Oracle Performance*, by Cary Millsap and Jeff Holt (O'Reilly 2003), pp161–165. There is a mistake in the book that I should clean up, however. Until very recently, I believed that Oracle Database kernels running on Solaris with microstate accounting turned on would result in *c* statistic values accurate to ±1 µs. However, since Oracle on Solaris (at least through 11*g*R2) uses the *times* system call instead of *getrusage*, even with microstate accounting available from the OS, Oracle's *c* statistic is accurate to only ±10,000 µs.

[10] They're most likely not to be consistent within a given file on Microsoft Windows systems, where *tim* values wrap (that is, get as large as they can go and then start over at 0) more frequently than on other systems.

© 2011 Method R Corporation. All rights reserved.

15 TRACE FILE ALPHABET: OS CALLS

All OS call lines resemble this:

```
WAIT #12397272: nam='db file sequential read' ela= 221
file#=1 block#=2735 blocks=1 obj#=423
tim=1294438818791494
```

Here's what the fields mean:

`WAIT`
: This call is an OS call, not a database call.[11]

`#12397272`
: The call's cursor handle id is 12397272.

`nam='db file sequential read'`
: This call was a read of database file that retrieved one or more blocks and stored them contiguously somewhere in memory. Oracle Corporation gives each OS call its kernel makes a distinct name, and the company documents the meanings of those names so you can more easily understand what's going on.

`ela= 221`
: This call consumed 221 ±1 µs of elapsed time.

`file#=1 block#=2735 blocks=1`
: This read call (probably a *pread* call—you can tell by using *strace*) read 1 block from Oracle file id 1, beginning at block id 2735. Each call has three parameters like these, which are named differently depending on the call name. These three parameters give you contextual information about what the call was meant to accomplish.

`obj#=423`
: This call operated upon the Oracle object with id 423. When a call operated upon no object in particular, the Oracle kernel will emit `obj#=-1`.

`tim=1294438818791494`
: This call completed at time 1,294,438,818,791,494 ±1 µs.

16 TRACE FILE FLOW

The Oracle kernel emits trace lines as calls complete. You have to know this to determine the parent-child relationships among calls. For example, what can you know about the following (simplified) trace data stream?

```
PARSE #2:…,e=1,…,tim=5
EXEC #2:…,e=2,…,tim=8
FETCH #2:…,e=4,…,tim=15
PARSE #1:…,e=15,…,tim=17
```

From just the data shown here, you can determine the start and end time of each call, and from that, you can work out the parent-child relationships among these calls:

```
tim  call
---  --------------
  2  PARSE #1 {
  4      PARSE #2 {
  5      }
  6      EXEC #2 {
  8      }
 11      FETCH #2 {
 15      }
 17  }
```

Before Oracle 10.2, when there were no *tim* values on the OS call lines, we inferred call parent-child hierarchy from the *dep* values on database call lines and from the lines' relative locations. We attributed each *dep=1* call as the child of the *dep=0* call that followed it in the trace data stream. With *tim* values on all our call lines, it is no longer necessary to do that. You can determine all the parent-child information you need now just by noticing at what time each call begins and ends.

17 TRACE FILE ARITHMETIC: PART 1

Now you know the trace file alphabet and how trace file lines flow forth at the completion of the database calls your client application makes and the OS calls your Oracle kernel makes. The next step is to understand a few details about how time works in Oracle trace files.

First, a database call's total duration (e) is approximately the sum of the time it spends using CPU (c) plus the duration of the OS calls made by the call. The equation is only approximate, primarily because of the ±10,000 µs on the CPU time measurement. Expressed as a precise equality, we have, for a given database call i:

$$e_i = c_i + \sum_{j \in \text{Children}(i)} ela_j + \Delta_i \qquad (1)$$

The term Δ_i there at the end is the time within the known duration of the call that is unaccounted for (unexplained, if you prefer) by Oracle. It can be positive or negative or zero. As much as you'd like for this unaccounted-for duration to be zero, it just won't be. However, in the aggregate, unaccounted-for time won't usually be a material contributor to total response time. When it is, it's an indication that:

[11] I've had good fun explaining why the word *wait* is the wrong word to use in this context. See for example my essay, "Dang it, people, they're syscalls, not 'waits'…"

© 2011 Method R Corporation. All rights reserved.

- Either you have forgotten to trace the OS calls (as with `waits=>false`);
- Or you have an un-patched Oracle bug;
- Or you have a process preemption problem (perhaps too much paging or swapping).

Here's how the arithmetic works out in a two-line trace file excerpt:

```
1. WAIT #124367516: nam='Disk file operations I/O'
   ela= 124 FileOperation=2 fileno=4 filetype=2 obj#=-1
   tim=1294702369858445
2. FETCH #124367516:c=0,e=386,p=0,cr=6,cu=0,mis=0,r=14,
   dep=0,og=1,plh=3956160932,tim=1294702369858666
```

Here, you have an OS call on line 1 that is a child of the database call on line 2. You can prove that by comparing *tim* and duration values:

```
              tim  call
----------------  ----------------------
1294702369858280  FETCH #124367516 {
1294702369858321      WAIT #124367516 {
1294702369858445      }
1294702369858666  }
```

For the database call (a *fetch*), we have $e = 386$, and $c = 0$. The sum of the OS call durations is simply 124. Using the equation above, this leaves an unaccounted-for duration of $\Delta = 262$. Here's what the call's response time looks like in the form of a profile table:

Duration (μs)		*Call*
262	67.9%	unaccounted-for within calls
124	32.1%	Disk file operations I/O
0	0.0%	CPU service, *fetch* calls
386	100.0%	Response time

Exhibit 3. Response time profile for the fetch database call, using Equation 1.

This is only a simple example. You can extend the same logic to database calls that are parents of many more OS calls, or even calls that are parents of one or more levels of other database calls.

18 TRACE FILE ARITHMETIC: PART 2

The second relationship you need to understand is that the total response time described by a trace file is simply the time elapsed from the beginning of the first call to the end of the final call. Remember that the Oracle kernel doesn't emit call begin times—only call end times. But you can compute the begin time of a call by subtracting its duration from its end time:

$$R = tim_{end} - \left(tim_{begin} - e_{begin} \right) \qquad (2)$$

Note that although Equation 2 refers to the duration of the first call as *e*, if the first call in the file is an OS call, you'll need to use its *ela* field.

Be aware that the values tim_{begin} and e_{begin} refer to fields associated with the first call to begin in the process being traced. Because the Oracle kernel emits trace lines when calls *end* (not when they begin), the first call to begin is not necessarily the first call listed in the trace file.

19 TRACE FILE ARITHMETIC: PART 3

The final category of response time to describe is the time that is unaccounted-for between the top-level database and OS calls. This one is simple:

$$\Delta_{between} = R - \sum_{i \in \text{TopLevelCalls}} e_i \qquad (3)$$

That is, the total between-call unaccounted-for duration is the total response time that we know has passed minus the time accounted for by top-level calls (both database and OS) in the trace file. So Equation 1 gives us most of the detail that you need to make up your profile. Equation 2 gives you the total response time that you need to account for. And Equation 3 gives you the label to which to assign the remainder of the unexplained duration. These are the three elements you need to create a complete response time profile.

20 TRACING TRAPS

When performance is the question, tracing is usually my answer. However, it's not always easy. Here are some of the pitfalls to avoid.

Data collection. By far, the most common problem we encounter in the field is the trace file whose response time doesn't match the response time of the end user's experience that you're trying to diagnose. We get a lot of trace files whose overall response times are dominated by either a small number of long-duration *SQL*Net message from client* calls or a small number of unaccounted-for gaps in the action. The most important thing to remember about collecting trace data is you want the response time of your trace file to exactly match the response time of your end user's experience.

Oracle microseconds. A microsecond is 1/1,000,000th of a second, or 1,000 nanoseconds. It may surprise you to learn that, on some platforms, an Oracle microsecond is 1,024 nanoseconds, or 1/976,563th of a second. Being off by 2.4% may not sound like too big of a problem, but if you want to

© 2011 Method R Corporation. All rights reserved.

regard an Oracle *tim* value as a number of microseconds that have elapsed since the Unix epoch (1970-01-01T00:00:00.000000Z in ISO 8601 form), you need to multiply your *tim* value by 1.024 before calculating the date and time it represents. Otherwise, for *tim* values generated in 2011, you will be off by almost a year.

Oracle bugs. The Oracle 11.1 era has not been kind to trace file users. Bug 7522002 causes database call time values to run off a 1,024-nanosecond-per-microsecond clock and OS call time values to run off a 1,000-nanosecond-per-microsecond clock. This causes the sequence of calls in a trace file to zigzag back and forth in time. This problem is correctible, but then bug 8342329 isn't.[12] It causes the Oracle kernel to emit the same *tim* value repeatedly for multiple distinct database calls, which is debilitating. Happily, these bugs in 11.1 have patches and are fixed in version 11.2.0.2.0.

21 COMMERCIAL TRACE FILE TOOLS

There's generally too much detail in a trace file to look at all at once. To make sense out of what's there requires software assistance. The most famous trace file software tool of all is Oracle Corporation's *tkprof* utility, which Oracle ships with the Oracle Database (all releases, all editions). *Tkprof* aggregates by SQL statement, and it allows you to sort those SQL statements in its output report by any of several criteria. This can work out ok, as long as:

- You care about inefficient SQL more than you care about response time (which is often a mistake).
- You sort by the right thing (`sort=prsela,exeela,fchela` is almost always the right thing).
- Your problem doesn't involve parent-child relationships among database calls (so-called recursive SQL).

I used *tkprof* extensively for the first ten years of my career. However, I spent a lot of time looking at raw trace data, too, because there's a lot of information inside a trace file that *tkprof* ignores, and the format of the *tkprof* output is not particularly friendly to my response time focus. So, I've helped to design and write a few software tools in the past ten years, including:

Method R Profiler
: The Method R Profiler is software that Jeff Holt and I wrote about in our book *Optimizing Oracle Performance* (O'Reilly 2003). It produces an HTML report that unambiguously explains the response time accounted in a trace file, no matter what the cause (whether SQL or not), using color to direct your attention.

mrskew
: The Method R skew analyzer is a data-mining tool for trace files. It can do nearly everything the Profiler can do, and more. With *mrskew*, you can group by pretty much anything in a trace file to drill with practically any analysis perspective you want. For example, you can group a whole directory full of trace files by SQL statement hash values, or database file ids, or hash latch address values, or library cache miss counts, or block counts per read call, or service-module-action names, or *prepare* call durations, …anything you like.

22 ACKNOWLEDGMENTS

Thank you to Jeff Holt, Ken Ferlita, Harold Palacio, and Ron Crisco at Method R Corporation for their help in preparing this work, and to Andrew Zitelli of Thales Raytheon Company for sharing his research with us.

23 ABOUT THE AUTHOR

Cary Millsap is the founder and president of Method R Corporation, a small business that builds and optimizes software all over the world (*http://method-r.com*). Cary designs and writes software and educational material. He is the author of Optimizing Oracle Performance (O'Reilly 2003), for which he and Method R colleague Jeff Holt were named *Oracle Magazine*'s Authors of the Year. He has presented at hundreds of conferences and courses worldwide, and he is also published in *Communications of the ACM*.

Cary spent the 1990s as a consultant and leader of consulting teams within Oracle Corporation. In 1999, Cary resigned as vice president of Oracle's System Performance Group to start a career as a business owner. Cary is 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* and tweets at *http://twitter.com/CaryMillsap*.

[12] See also patches 9415425 for Oracle 11.1.0.7.1 on Solaris and 9941786 for Oracle 11.1.0.7.4 on Solaris.

© 2011 Method R Corporation. All rights reserved.

24 REVISION HISTORY

2011/01/13
: Initial revision.

2011/02/23
: Incorporated discovery by Alex Gorbachev, researched by Jeff Holt and Andrew Zitelli, noting that even with Solaris microstate accounting, the *c* statistic is accurate only to ±10,000µs.

  Incorporated feedback from Mark Farnham into the footnote about tracing "to your heart's content."

© 2011 Method R Corporation. All rights reserved.

25 APPENDIX: “HELLO WORLD” TRACE

```
Trace file /opt/oracle/diag/rdbms/v11202/V11202/trace/V11202_ora_21089.trc
Oracle Database 11g Enterprise Edition Release 11.2.0.2.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options
ORACLE_HOME = /opt/oracle/product/11.2.0.2
System name:Linux
Node name:   rhel01.dev.method-r.com
Release:     2.6.18-92.1.13.el5xen
Version:     #1 SMP Thu Sep 4 04:20:55 EDT 2008
Machine:     i686
VM name:     Xen Version: 3.1 (PVM)
Instance name: V11202
Redo thread mounted by this instance: 1
Oracle process number: 30
Unix process pid: 21089, image: oracle@rhel01.dev.method-r.com (TNS V1-V3)


*** 2011-01-10 13:33:22.746
*** SESSION ID:(19.1247) 2011-01-10 13:33:22.746
*** CLIENT ID:() 2011-01-10 13:33:22.746
*** SERVICE NAME:(SYS$USERS) 2011-01-10 13:33:22.746
*** MODULE NAME:(SQL*Plus) 2011-01-10 13:33:22.746
*** ACTION NAME:() 2011-01-10 13:33:22.746

=====================
PARSING IN CURSOR #7285244 len=60 dep=0 uid=5 oct=47 lid=5 tim=1294688002745892 hv=1548041990 ad='3d51df50'
sqlid='31wf1v9f4ags6'
BEGIN dbms_monitor.session_trace_enable(binds=>true); END;
END OF STMT
EXEC #7285244:c=0,e=845,p=0,cr=0,cu=0,mis=0,r=1,dep=0,og=1,plh=0,tim=1294688002745889
WAIT #7285244: nam='SQL*Net message to client' ela= 2 driver id=1650815232 #bytes=1 p3=0 obj#=-1 tim=1294688002746391
WAIT #7285244: nam='SQL*Net message from client' ela= 931 driver id=1650815232 #bytes=1 p3=0 obj#=-1
tim=1294688002747354
CLOSE #7285244:c=0,e=55,dep=0,type=0,tim=1294688002747449
=====================
PARSING IN CURSOR #7285244 len=30 dep=0 uid=5 oct=3 lid=5 tim=1294688002747560 hv=1604528588 ad='3d526cd8'
sqlid='guthms1gu6afc'
select 'Hello world' from dual
END OF STMT
PARSE #7285244:c=0,e=70,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,plh=1388734953,tim=1294688002747559
EXEC #7285244:c=0,e=112,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,plh=1388734953,tim=1294688002747793
WAIT #7285244: nam='SQL*Net message to client' ela= 2 driver id=1650815232 #bytes=1 p3=0 obj#=-1 tim=1294688002747865
FETCH #7285244:c=0,e=8,p=0,cr=0,cu=0,mis=0,r=1,dep=0,og=1,plh=1388734953,tim=1294688002747910
STAT #7285244 id=1 cnt=1 pid=0 pos=1 obj=0 op='FAST DUAL  (cr=0 pr=0 pw=0 time=4 us cost=2 size=0 card=1)'
WAIT #7285244: nam='SQL*Net message from client' ela= 375 driver id=1650815232 #bytes=1 p3=0 obj#=-1
tim=1294688002748453
FETCH #7285244:c=0,e=1,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=0,plh=1388734953,tim=1294688002748493
WAIT #7285244: nam='SQL*Net message to client' ela= 1 driver id=1650815232 #bytes=1 p3=0 obj#=-1 tim=1294688002748520
WAIT #7285244: nam='SQL*Net message from client' ela= 1627 driver id=1650815232 #bytes=1 p3=0 obj#=-1
tim=1294688002750168
XCTEND rlbk=0, rd_only=1, tim=1294688002750230
=====================
PARSING IN CURSOR #7386176 len=447 dep=1 uid=0 oct=2 lid=0 tim=1294688002750383 hv=1097020010 ad='3d7e77d4'
sqlid='f711myt0q6cma'
insert into sys.aud$( sessionid,entryid,statement,ntimestamp#, userid,userhost,terminal,action#,returncode, logoff
$lread,logoff$pread,logoff$lwrite,logoff$dead, logoff$time,comment$text,spare1,clientid,sessioncpu,proxy$sid,user$guid,
instance#,process#,auditid,dbid) values(:1,:2,:3,SYS_EXTRACT_UTC(SYSTIMESTAMP),     :4,:5,:6,:7,:8,     :9,:10,:11,:12,
cast(SYS_EXTRACT_UTC(systimestamp) as date),:13,:14,:15,:16,:17,:18,     :19,:20,:21,:22)
END OF STMT
PARSE #7386176:c=0,e=93,p=0,cr=0,cu=0,mis=0,r=0,dep=1,og=4,plh=0,tim=1294688002750382
BINDS #7386176:
Bind#0
oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
kxsbbbfp=0070ac04  bln=22  avl=04  flg=05
value=63843
Bind#1
oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
kxsbbbfp=0070abe0  bln=24  avl=02  flg=05
```

© 2011 Method R Corporation. All rights reserved.

```
value=2
Bind#2
oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
kxsbbbfp=0070abbc  bln=24  avl=02  flg=05
value=1
Bind#3
oacdty=01 mxl=32(06) mxlc=00 mal=00 scl=00 pre=00
oacflg=18 fl2=0001 frm=01 csi=178 siz=32 off=0
kxsbbbfp=006df248  bln=32  avl=06  flg=09
value="SYSTEM"
Bind#4
oacdty=01 mxl=32(23) mxlc=00 mal=00 scl=00 pre=00
oacflg=18 fl2=0001 frm=01 csi=178 siz=32 off=0
kxsbbbfp=006df1c2  bln=32  avl=23  flg=09
value="rhel01.dev.method-r.com"
Bind#5
oacdty=01 mxl=32(05) mxlc=00 mal=00 scl=00 pre=00
oacflg=18 fl2=0001 frm=01 csi=178 siz=32 off=0
kxsbbbfp=635ffd04  bln=32  avl=05  flg=09
value="pts/0"
Bind#6
oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
kxsbbbfp=0070ab98  bln=24  avl=03  flg=05
value=101
Bind#7
oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
kxsbbbfp=0070ab74  bln=24  avl=01  flg=05
value=0
Bind#8
oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
kxsbbbfp=0070ab40  bln=24  avl=02  flg=05
value=44
Bind#9
oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
kxsbbbfp=0070ab1c  bln=24  avl=01  flg=05
value=0
Bind#10
oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
kxsbbbfp=0070aaf8  bln=24  avl=02  flg=05
value=4
Bind#11
oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
kxsbbbfp=0070aad4  bln=24  avl=01  flg=05
value=0
Bind#12
oacdty=01 mxl=32(00) mxlc=00 mal=00 scl=00 pre=00
oacflg=18 fl2=0001 frm=01 csi=178 siz=32 off=0
kxsbbbfp=00000000  bln=32  avl=00  flg=09
Bind#13
oacdty=01 mxl=32(06) mxlc=00 mal=00 scl=00 pre=00
oacflg=18 fl2=0001 frm=01 csi=178 siz=32 off=0
kxsbbbfp=635ffc98  bln=32  avl=06  flg=09
value="oracle"
Bind#14
oacdty=01 mxl=32(00) mxlc=00 mal=00 scl=00 pre=00
oacflg=18 fl2=0001 frm=01 csi=178 siz=32 off=0
kxsbbbfp=00000000  bln=32  avl=00  flg=09
Bind#15
oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
kxsbbbfp=0070aab0  bln=24  avl=02  flg=05
value=1
Bind#16
oacdty=02 mxl=22(00) mxlc=00 mal=00 scl=00 pre=00
oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
```

© 2011 Method R Corporation. All rights reserved.

```
kxsbbbfp=0070aa8c  bln=24  avl=00  flg=05
Bind#17
oacdty=01 mxl=32(00) mxlc=00 mal=00 scl=00 pre=00
oacflg=18 fl2=0001 frm=01 csi=178 siz=32 off=0
kxsbbbfp=00000000  bln=32  avl=00  flg=09
Bind#18
oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
kxsbbbfp=0070aa58  bln=24  avl=01  flg=05
value=0
Bind#19
oacdty=01 mxl=32(05) mxlc=00 mal=00 scl=00 pre=00
oacflg=18 fl2=0001 frm=01 csi=178 siz=32 off=0
kxsbbbfp=bfc1b840  bln=32  avl=05  flg=09
value="21089"
Bind#20
oacdty=01 mxl=32(00) mxlc=00 mal=00 scl=00 pre=00
oacflg=18 fl2=0001 frm=01 csi=178 siz=32 off=0
kxsbbbfp=00000000  bln=32  avl=00  flg=09
Bind#21
oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
kxsbbbfp=0070a7f8  bln=24  avl=06  flg=05
value=518865609
EXEC #7386176:c=0,e=1733,p=0,cr=1,cu=2,mis=0,r=1,dep=1,og=4,plh=0,tim=1294688002752264
STAT #7386176 id=1 cnt=0 pid=0 pos=1 obj=0 op='LOAD TABLE CONVENTIONAL  (cr=1 pr=0 pw=0 time=155 us)'
CLOSE #7386176:c=0,e=5,dep=1,type=0,tim=1294688002752341
WAIT #0: nam='log file sync' ela= 153 buffer#=5673 sync scn=1046841 p3=0 obj#=-1 tim=1294688002752632
CLOSE #7285244:c=0,e=9,dep=0,type=0,tim=1294688002752690
CLOSE #7248564:c=0,e=30,dep=0,type=0,tim=1294688002752740
```

© 2011 Method R Corporation. All rights reserved.