How to find and fix your

Java APEX ADF OBIEE .NET SQL PL/SQL

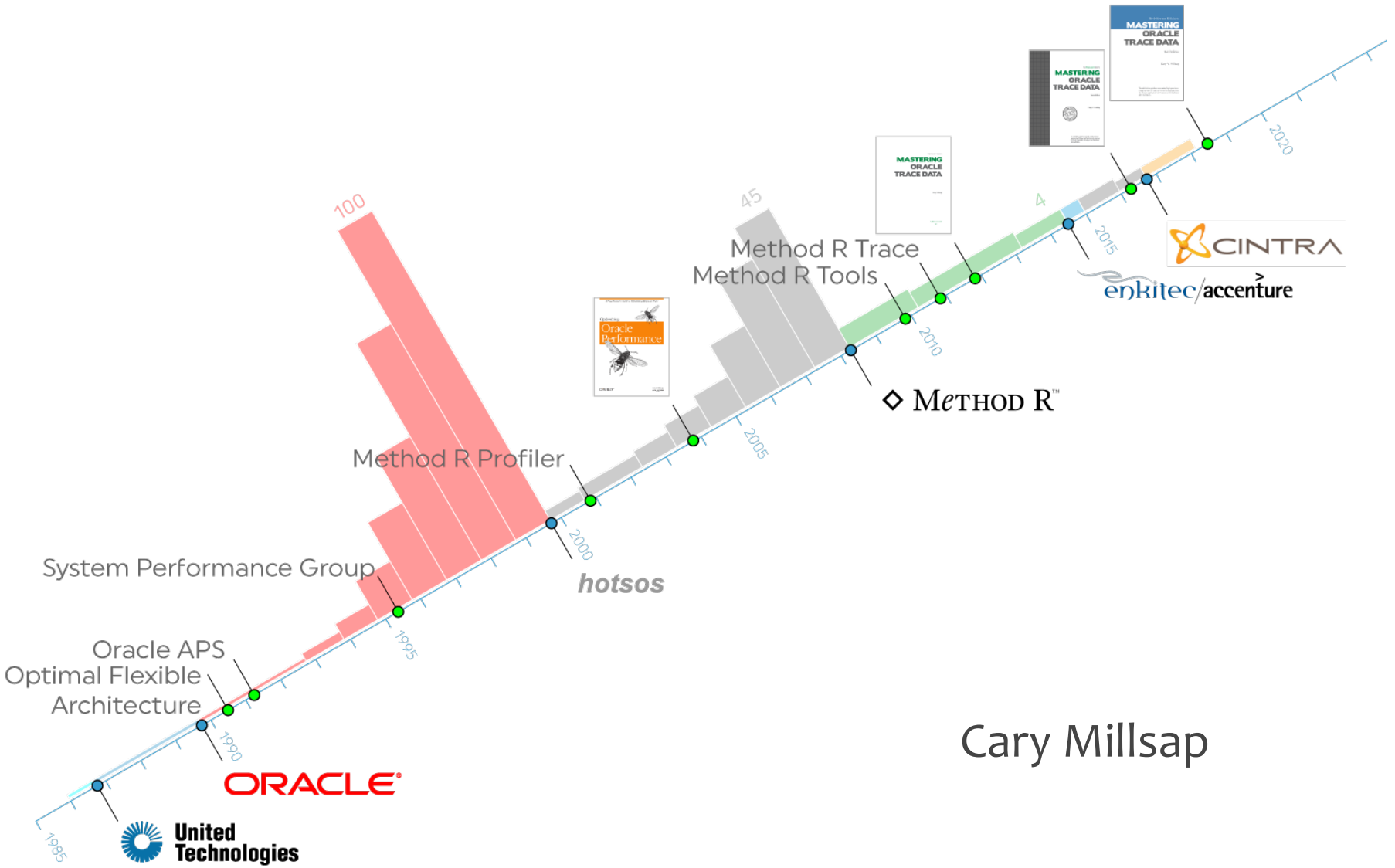
application performance problem

Cary Millsap
Cintra Software and Services · Method R Corporation

@CaryMillsap

UNYOUG Educational Workshop
Rochester Institute of Technology · Rochester, New York
Thursday 12 April 2018

© 2006, 2018 Cary Millsap



Cary Millsap

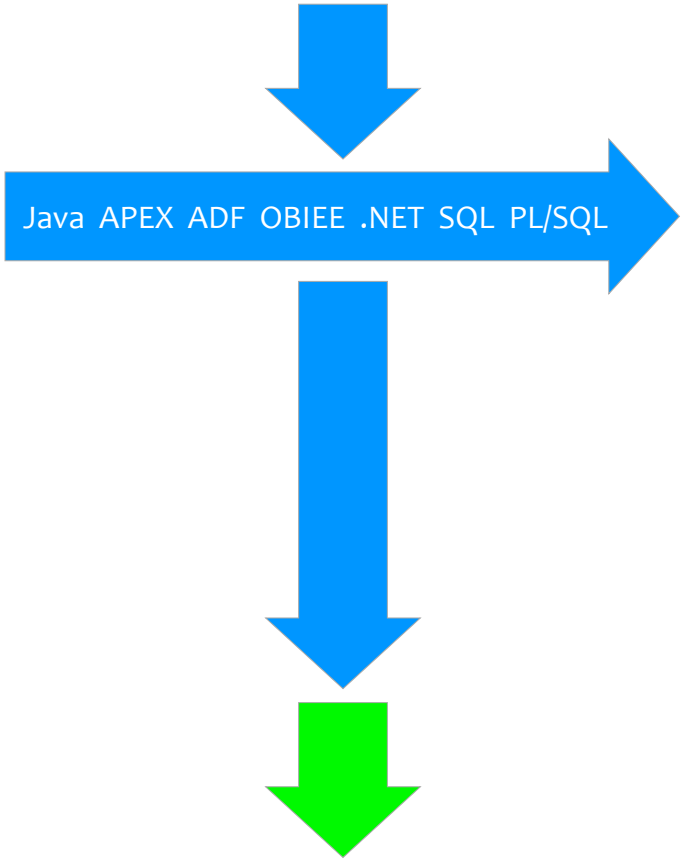
"What is the most common Oracle performance problem you see?"

Q

"What is the most common Oracle performance problem you see?"

A

... Assuming that other people's common problems must be **your** problem.


Java APEX ADF OBIEE .NET SQL PL/SQL

What is a **performance problem?**

WHAT
YOU HAVE
LEARNED,
UNLEARN
YOU MUST.

PERFORMANCE IS NOT
AN ATTRIBUTE OF A SYSTEM.

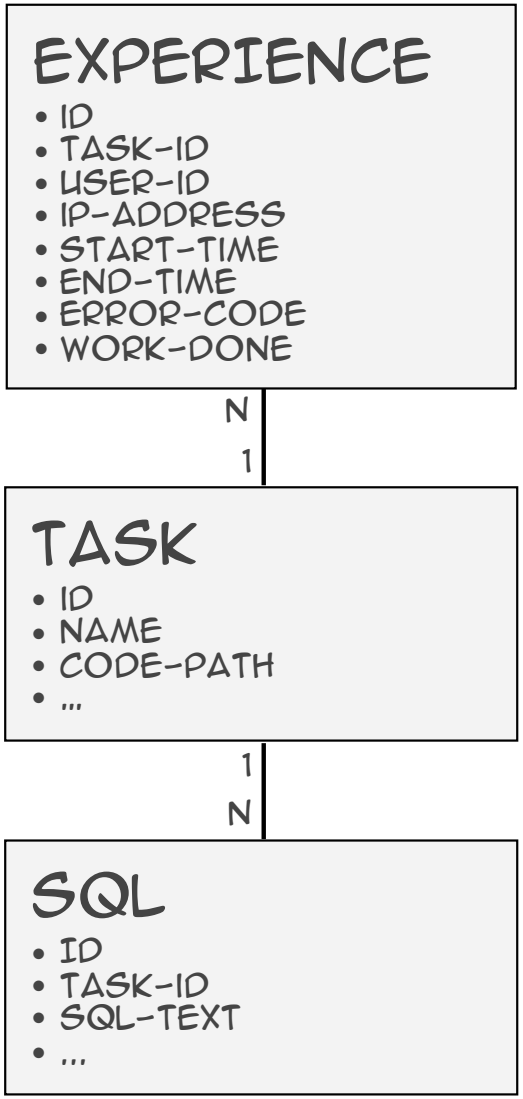
#define $PERF_{id}$ (($R_{id} \leq SLR_{id}$)?"FAST":"SLOW")

ID	USERNAME	OPERATION	R	SLR
1	FCHANG	OE BOOK	12.019	2.0
2	RSMITH	OE SHIP	3.528	5.0
3	DJOHNSON	OE PICK	1.211	5.0
4	FFORBES	OE BOOK	0.716	2.0
5	FCHANG	OE BOOK	1.917	2.0
6	LBUMONT	PA MTCH	1.305	2.0

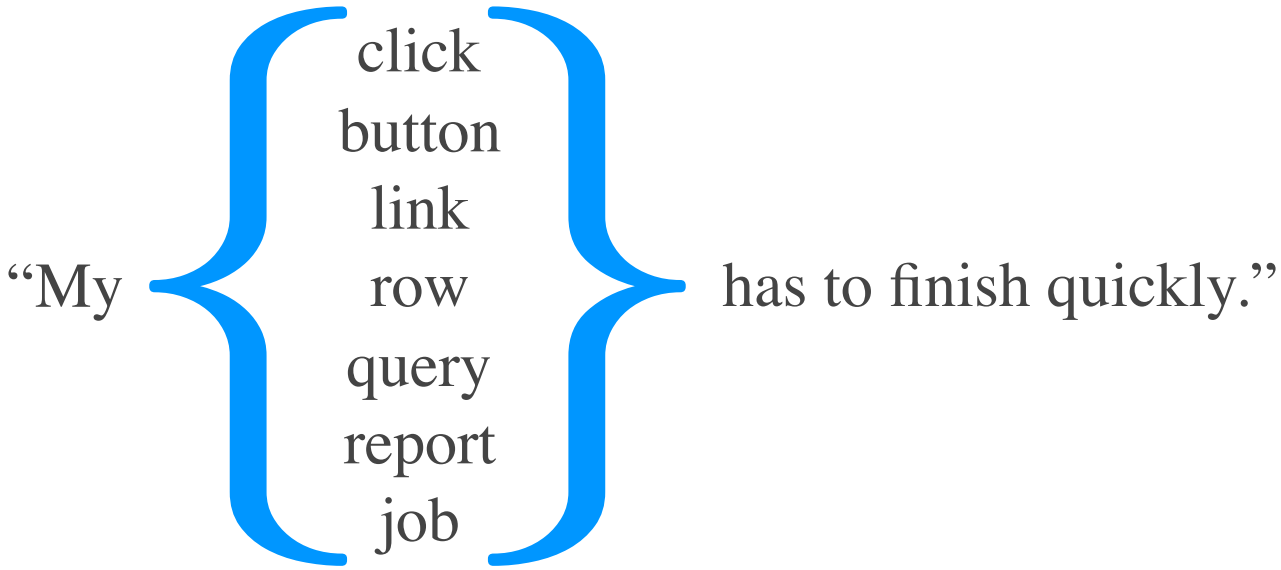
#define $PERF_{id}$ (($R_{id} \leq SLR_{id}$)?"FAST":"SLOW")

ID	USERNAME	OPERATION	R	SLR	PERF
1	FCHANG	OE BOOK	12.019	2.0	SLOW
2	RSMITH	OE SHIP	3.528	5.0	FAST
3	DJOHNSON	OE PICK	1.211	5.0	FAST
4	FFORBES	OE BOOK	0.716	2.0	FAST
5	FCHANG	OE BOOK	1.917	2.0	FAST
6	LBUMONT	PA MTCH	1.305	2.0	FAST

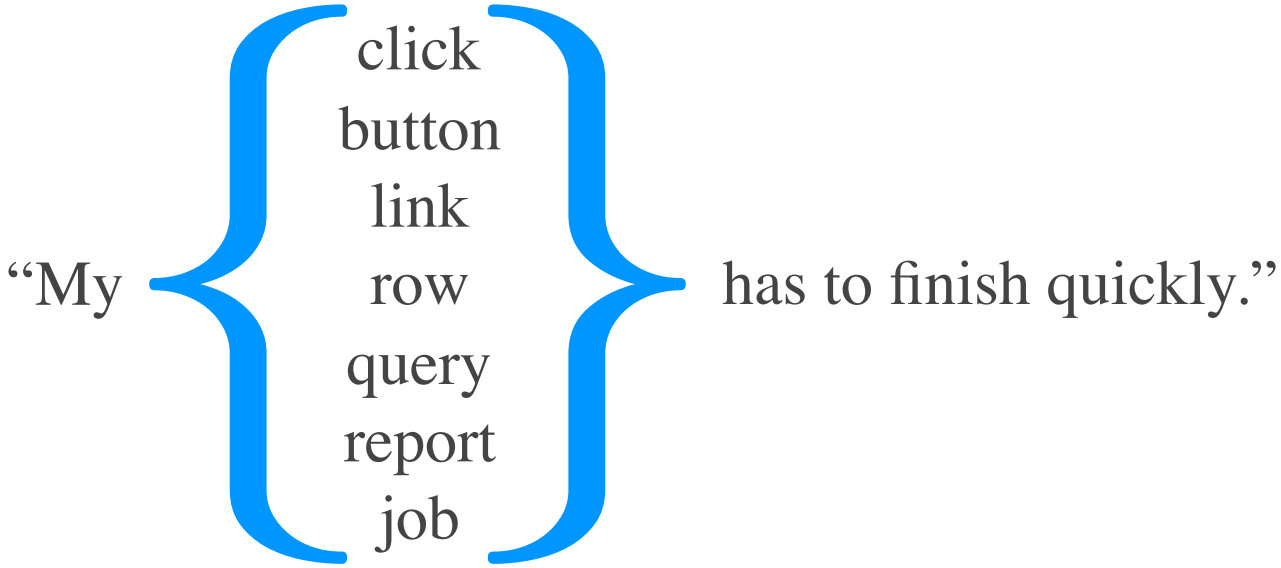
PERFORMANCE IS AN ATTRIBUTE OF EACH INDIVIDUAL EXPERIENCE WITH A SYSTEM.


EXPERIENCE
• ID
• TASK-ID
• USER-ID
• IP-ADDRESS
• START-TIME
• END-TIME
• ERROR-CODE
• WORK-DONE
N
1
TASK
• ID
• NAME
• CODE-PATH
• ...
1
N
SQL
• ID
• TASK-ID
• SQL-TEXT
• ...

```
<experience
   id       = "b3196c98-906d-4394-bc55-0339518a63b2"
   task-id  = "7"
   uid      = "238"
   ip       = "142.128.130.186"
   t0       = "2014-04-10T08:32:14.137886"
   t1       = "2014-04-10T08:32:17.891173"
   err      = ""
   work     = "3"
/>
```



This is what performance is.

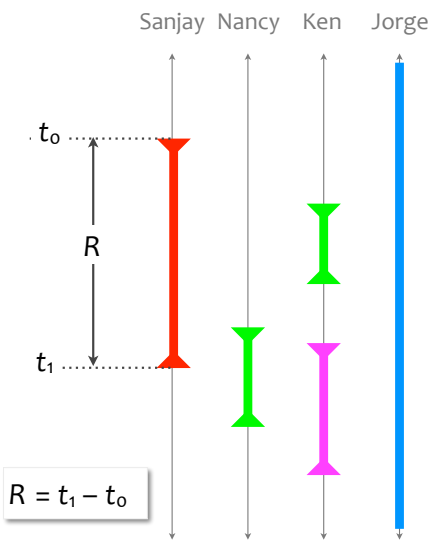


A performance **problem** is when it *doesn’t*.

"How long does it take?"

Response time (*R*)

Duration from service request to service fulfillment.



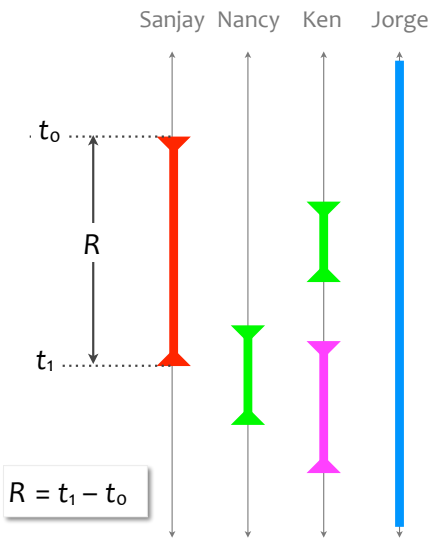
Two big questions...

1. How long?
2. Why?

"How long does it take?"

Response time (*R*)

Duration from service request to service fulfillment.



Two big questions...

1. How long?
2. Why?

Method R

Method R

1. Select the **experience** you need to improve.
2. Measure its **response time** (*R*) in detail.
3. Execute the best **net-payoff** remedy.
4. Repeat until economically **optimal**.

Method R

1. Select the **experience** you need to improve.
2. Measure its **response time** (*R*) in detail.
3. Execute the best **net-payoff** remedy.
4. Repeat until economically **optimal**.

Method R

1. **Focus** on the real, actual problem.
2. **Catch it in the act.**
3. Do the **smart** thing.
4. **Quit** when "helping" stops helping.

Method R

1. Select the **experience** you need to improve.
2. Measure its **response time** (*R*) in detail.
3. Execute the best **net-payoff** remedy.
4. Repeat until economically **optimal**.

Method R

2. Measure its **response time** (*R*) in detail.

How do you do this,
when the "it" is your code?

WHAT
YOU HAVE
LEARNED,
UNLEARN
YOU MUST.

Oracle extended SQL tracing
is a feature of *every* Oracle Database.

Oracle7 1992 Oracle8 1997 Oracle8*i* 2000 Oracle9*i* 2001 Oracle 10g 2004 Oracle 11g 2007 Oracle 12c 2013 Oracle 18c 2018

ORACLE DATABASE EXPRESS EDITION
ORACLE DATABASE STANDARD EDITION
ORACLE DATABASE ENTERPRISE EDITION
ORACLE EXADATA
ORACLE CLOUD

Measuring Oracle response times

1

Trace

2

Get the trace file

3

Understand its story

1

Trace

2

Get the trace file

3

Understand its story

This is the **hardest** part.

...But only the **first** time.
After that, it's lather, rinse, repeat.

Well, it's easy in Oracle APEX.

To decide at **run time** whether to trace your code...

Set **Enable Application Tracing** to **Yes** for your APEX instance.

`https://app.com/apex/f?p=150:1:5547991082303::NO:::&P_TRACE=YES`

Other technologies require a little more work.

First, the *basics*.

To decide at **compile time** to trace *all* your code...

```
   dbms_session.session_trace_enable(
       waits       => true,
       binds       => false,
       plan_stat   => 'FIRST_EXECUTION'
   );


-- Your OE BOOK code

dbms_session.session_trace_disable();
```

To decide at **run time** whether to trace your code...

```
if (should_trace('OE BOOK', dbms_random.value(0,1)) {
    dbms_session.session_trace_enable(
        waits       => true,
        binds       => false,
        plan_stat   => 'FIRST_EXECUTION'
    );
}

-- Your OE BOOK code

dbms_session.session_trace_disable();
```

...where `should_trace` looks like this.

```
sub should_trace(t, r) {
    select trace_proportion from trace_control where task_name = :t;
    return (r <= trace_proportion);
}
```

trace_control

task_name	trace_proportion
OE BOOK	0.05
OE PICK	0.02
OE SHIP	1
OE INVOICE	0.01

```
should_trace("OE BOOK", 0.00) → true
should_trace("OE BOOK", 0.01) → true
should_trace("OE BOOK", 0.02) → true
...
should_trace("OE BOOK", 0.05) → true
```

5%

```
should_trace("OE BOOK", 0.06) → false
should_trace("OE BOOK", 0.07) → false
should_trace("OE BOOK", 0.08) → false
...
should_trace("OE BOOK", 1.00) → false
```

95%

Oracle Database helps you implement

run time tracing decisions...

...without having to make your developers do the **if** block stuff.

The DBA does *this*, at **run time.**

```
dbms_monitor.serv_mod_act_trace_enable(
    service_name   => 'SYS$USERS',
    module_name    => 'OE BOOK',
    action_name    => dbms_monitor.all_actions,
    waits          => true,
    binds          => false,
    plan_stat      => 'FIRST_EXECUTION'
);
```

But this works only if your code
sets its module name to “OE BOOK”.

How you set your module name varies by technology.

SQL PL/SQL Java ADF .NET OBIEE

SQL PL/SQL

To set your code's module and action names...

```
-- PL/SQL example

dbms_application_info.set_module(
   module_name    => 'OE BOOK',
   action_name    => sys_guid()
);

-- Your OE BOOK code

dbms_application_info.set_module(
   module_name    => null,
   action_name    => null
);
```

Java ADF

To set your code's module and action names...

```
// JDBC 12c example

Properties p = new Properties();
p.put("OCSID.MODULE", "OE BOOK");
p.put("OCSID.ACTION", UUID.randomUUID().toString());
connection.setClientInfo(p);

// Your OE BOOK code

p.put("OCSID.MODULE", "");
p.put("OCSID.ACTION", "");
connection.setClientInfo(p);
```

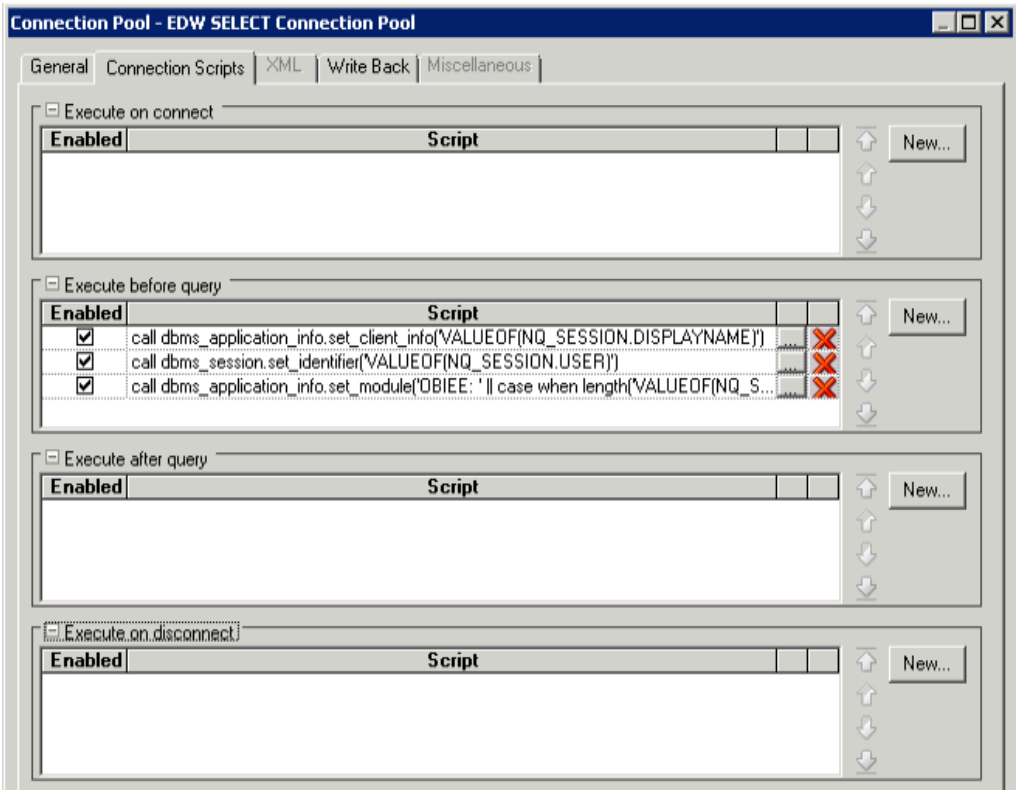
ODP.NET

To set your code's module and action names...

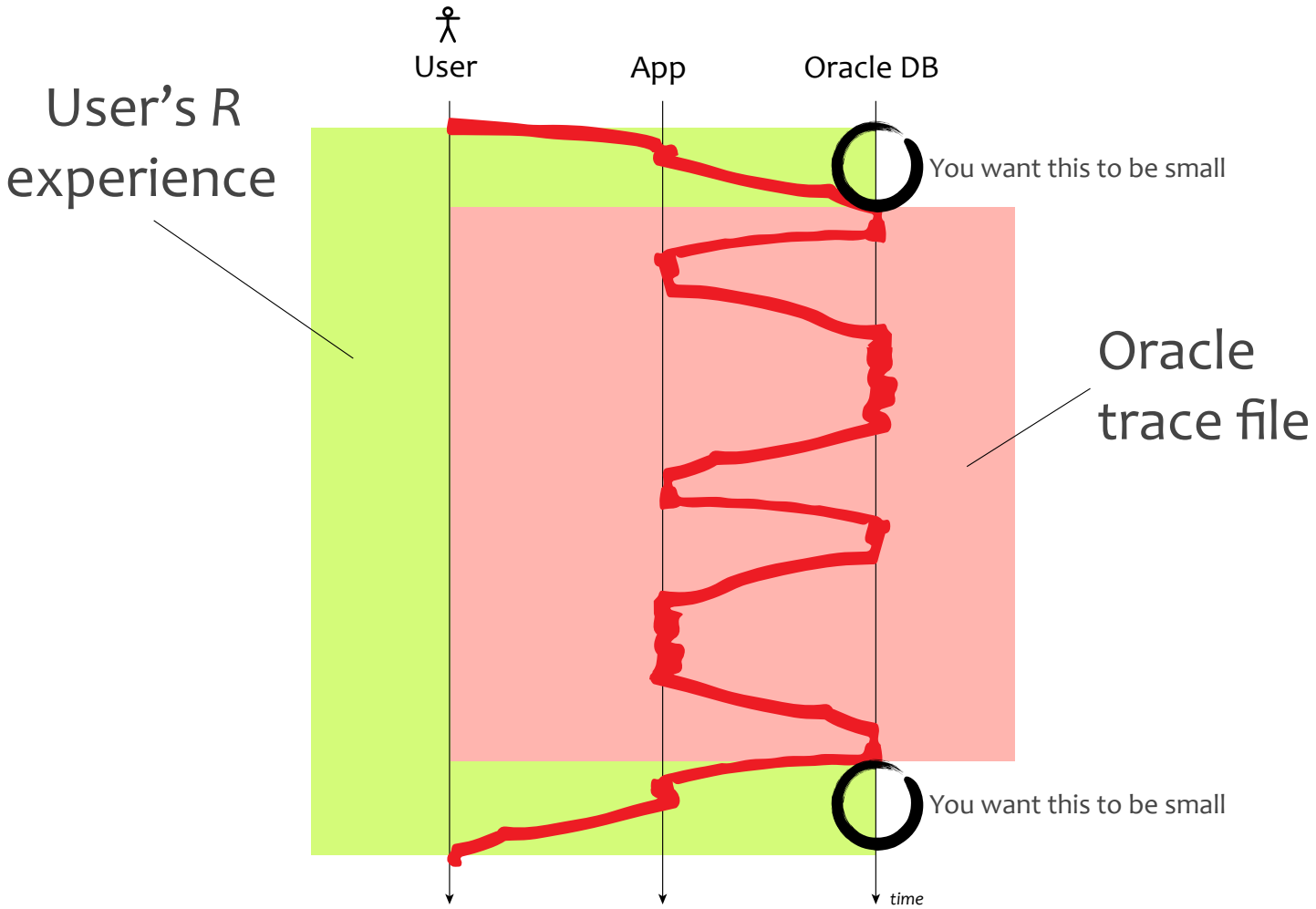
```
// C# example

conn.ModuleName = "OE BOOK";
conn.ActionName = Guid.NewGuid().toString();

// Your OE BOOK code

conn.ModuleName = "";
conn.ActionName = "";
```

OBIEE

To set your code's module and action names...

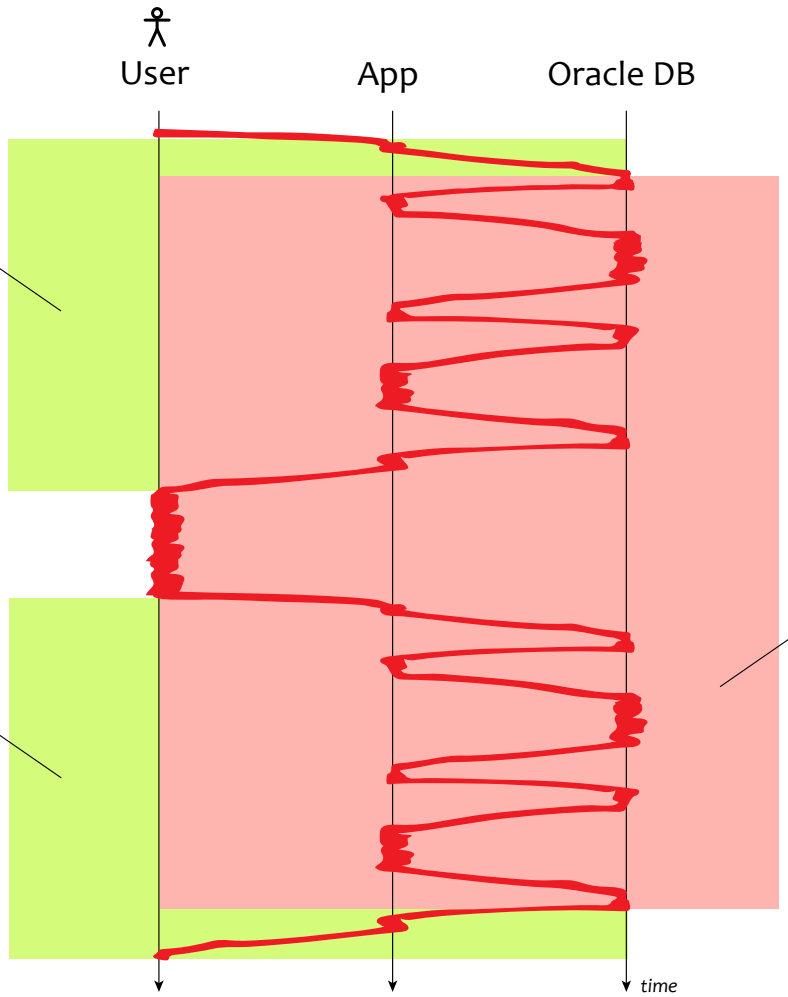


Here's the goal.


User
App
Oracle DB
User's R experience
You want this to be small
Oracle trace file
You want this to be small
time

An experience

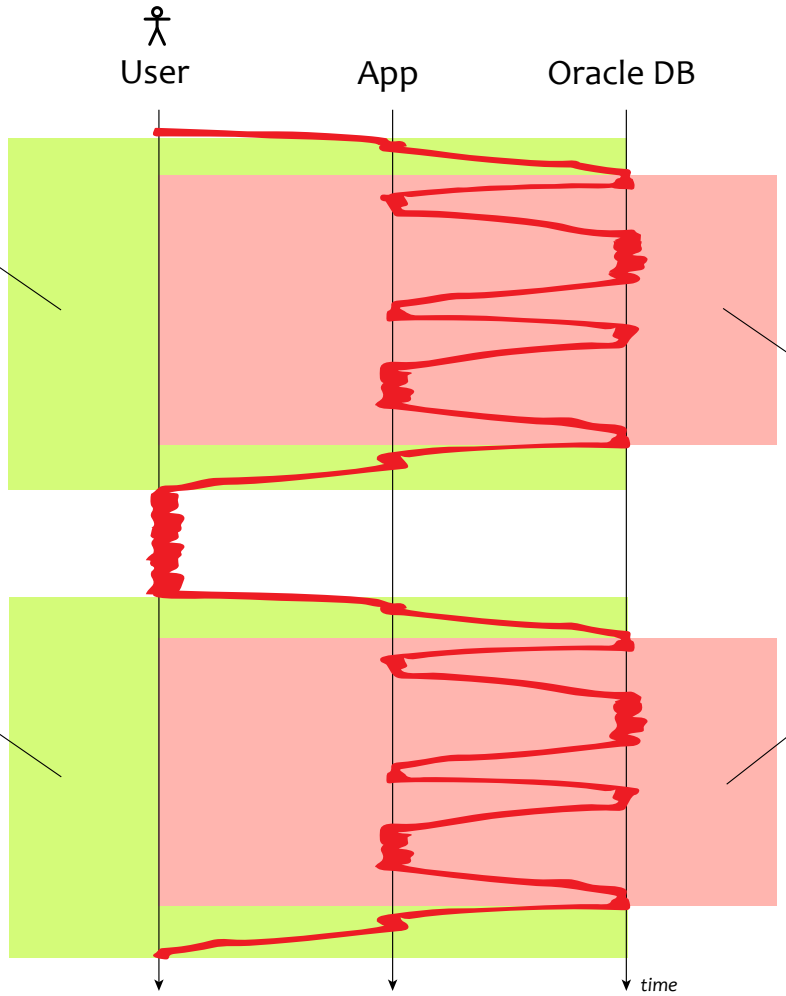
Another experience



Not the trace file you want

An experience

Another experience



You want one trace file per experience

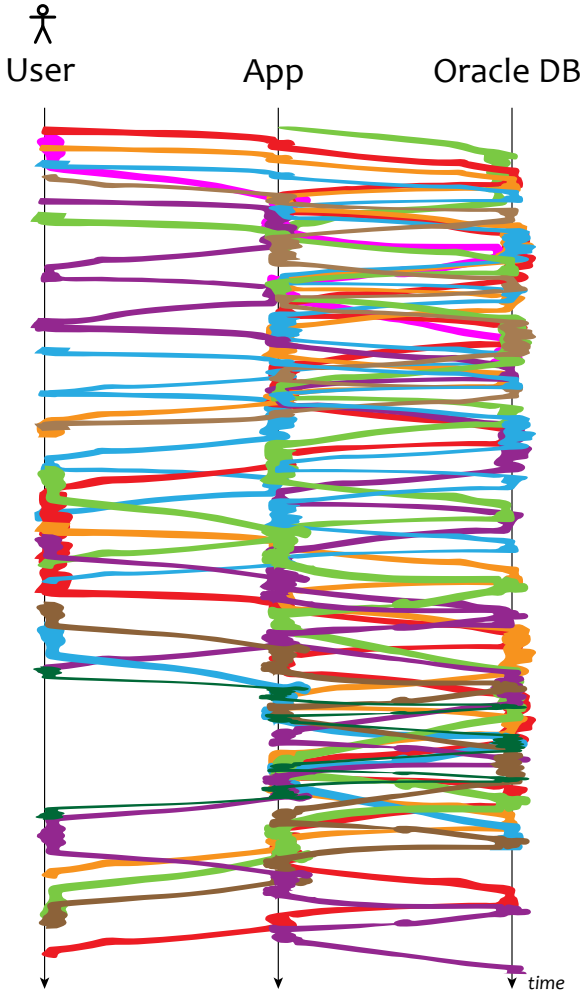
The goal:

Trace **exactly** each **user experience** you care about.

So that you can see how your code consumes time…

1. when it behaves properly, and
2. when it misbehaves.

WHAT
YOU HAVE
LEARNED,
UNLEARN
YOU MUST.



This is what you're looking at when you use systemwide aggregations.

1 Trace

2 Get the trace file

3 Understand its story

This is the **boring** part.

...But it's an inexpensive problem to solve.

Some things to know...

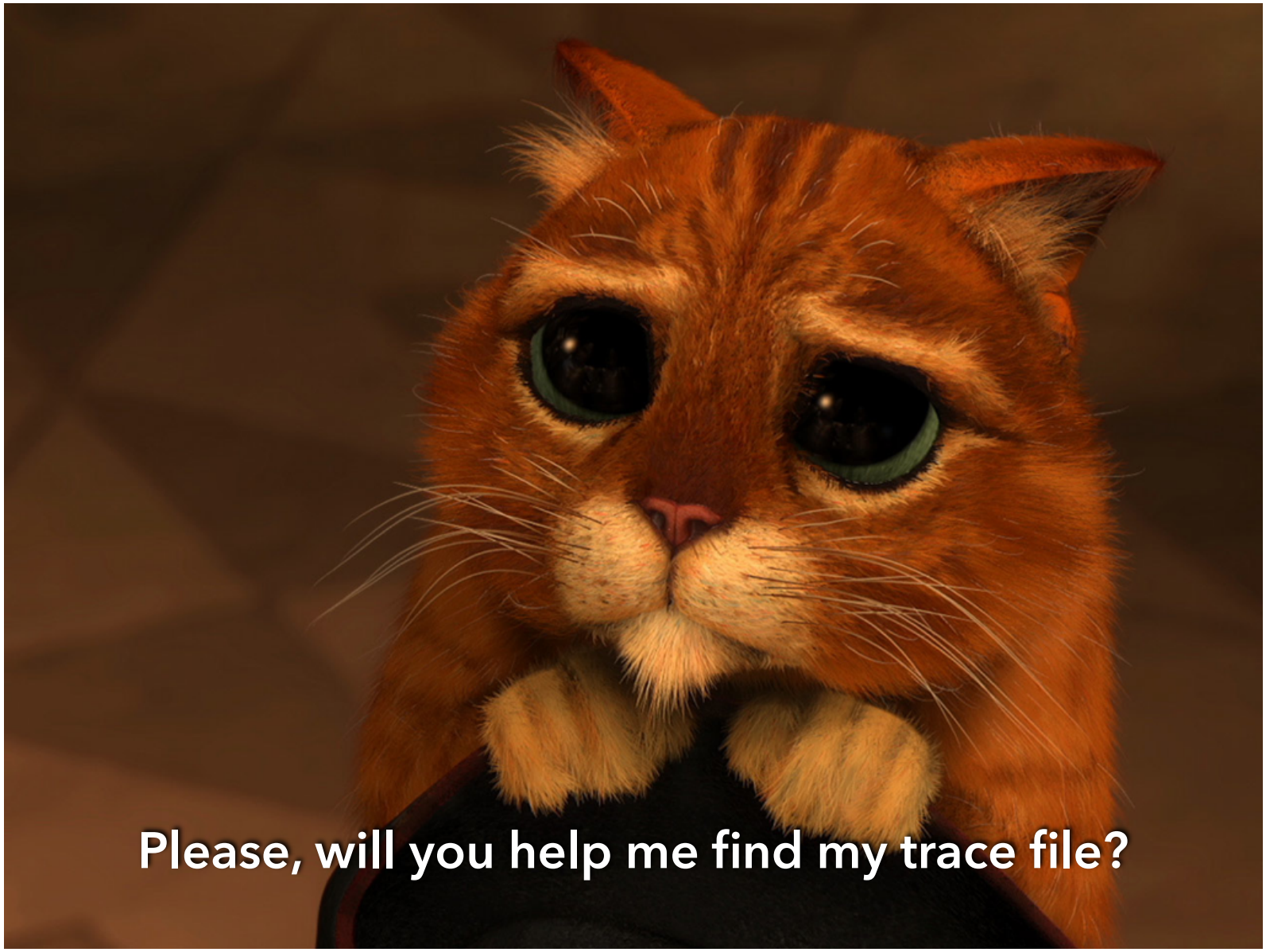
Your trace file is on the Oracle Database server(s), called:

```
select value tracefile from v$diag_info where name = 'Default Trace File'
```

Or, more generally:

```
select tracefile from v$process where addr=(
  select paddr from v$session where sid=(
    select sid from v$mystat where rownum=1
  )
)
```

Sessions with DOP = *k* can create more than 2*k* + 1 trace files.


Please, will you help me find my trace file?

There are lots of ways to fetch the trace data.

FTP
Samba
NFS mount
portable disk
USB thumb drive
Oracle Database directory objects
Method R Trace extension for Oracle SQL Developer

IT'S A SOLVED PROBLEM.

Fetching trace files can be easy.

You can build tools, or you can buy them.

1

Trace

2

Get the trace file

3

Understand its story

This is the FUN part.

What's **in** there?!

An Oracle trace file is a log that shows what your code did inside the Oracle Database.

Some things to know...

Oracle writes a trace line when a (db|os) call finishes.

There are two primary line formats: one for db calls, one for os calls.

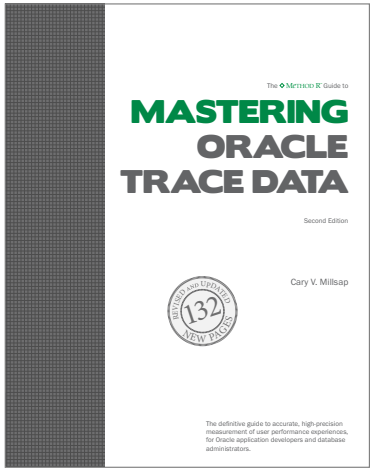
Each call is associated with a SQL or PL/SQL statement through a cursor id.

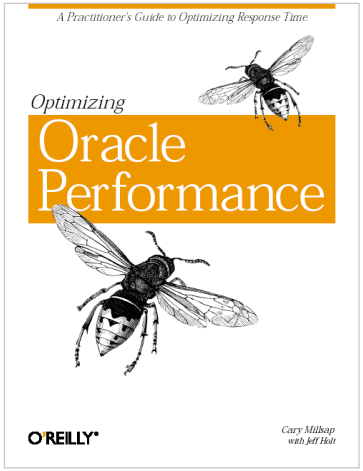
Each line contains a time stamp (tim) and a duration (e|ela).

$R \neq \sum$(e|ela) because parent call durations include child call durations.

For more details...

method-r.com





Let's look at some trace lines...

Oracle kernel code path

```
begin prepare
     CPU
     latch-related syscall
     CPU
end prepare
begin exec
     CPU
     write(SQLNET_OUT, result_to_client);
end exec
read(SQLNET_IN, next_request_from_client);
begin fetch
     CPU
     latch-related syscall
     CPU
     write(SQLNET_OUT, result_to_client);
end fetch
read(SQLNET_IN, next_request_from_client);
begin fetch
     CPU
     write(SQLNET_OUT, result_to_client);
     write(SQLNET_OUT, more_results);
     write(SQLNET_OUT, more_results);
end fetch
read(SQLNET_IN, next_request_from_client);
begin fetch
     CPU
     write(SQLNET_OUT, result_to_client);
     write(SQLNET_OUT, more_results);
     write(SQLNET_OUT, more_results);
end fetch
read(SQLNET_IN, next_request_from_client);
begin fetch
     CPU
     write(SQLNET_OUT, result_to_client);
     write(SQLNET_OUT, more_results);
     write(SQLNET_OUT, more_results);
end fetch
read(SQLNET_IN, next_request_from_client);
```

This is the kind of stuff your code causes the Oracle kernel to do.

Oracle kernel code path	Oracle extended SQL trace data
begin prepare	
CPU	
latch-related syscall	WAIT #42: nam='latch: library cache'…
CPU	
end prepare	PARSE #42:c=10000,…
begin exec	
CPU	
write(SQLNET_OUT, result_to_client);	WAIT #42: nam='SQL*Net message to client'…
end exec	EXEC #42:c=10000,…
read(SQLNET_IN, next_request_from_client);	WAIT #42: nam='SQL*Net message from client'…
begin fetch	
CPU	
latch-r	WAIT #42: nam='latch: cache buffers chains'…
CPU	
write(S nt);	WAIT #42: nam='SQL*Net message to client'…
end fetch	FETCH #42:c=20000,…
read(SQLNET_ ent);	WAIT #42: nam='SQL*Net message from client'…
begin fetch	
CPU	
write(S nt);	WAIT #42: nam='SQL*Net message to client'…
write(S	WAIT #42: nam='SQL*Net more data to client'…
write(S	WAIT #42: nam='SQL*Net more data to client'…
end fetch	FETCH #42:c=20000,…
read(SQLNET_ ent);	WAIT #42: nam='SQL*Net message from client'…
begin fetch	
CPU	
write(SQLNET_OUT, result_to_client);	WAIT #42: nam='SQL*Net message to client'…
write(SQLNET_OUT, more_results);	WAIT #42: nam='SQL*Net more data to client'…
write(SQLNET_OUT, more_results);	WAIT #42: nam='SQL*Net more data to client'…
end fetch	FETCH #42:c=20000,…
read(SQLNET_IN, next_request_from_client);	WAIT #42: nam='SQL*Net message from client'…
begin fetch	
CPU	
write(SQLNET_OUT, result_to_client);	WAIT #42: nam='SQL*Net message to client'…
write(SQLNET_OUT, more_results);	WAIT #42: nam='SQL*Net more data to client'…
write(SQLNET_OUT, more_results);	WAIT #42: nam='SQL*Net more data to client'…
end fetch	FETCH #42:c=20000,…
read(SQLNET_IN, next_request_from_client);	WAIT #42: nam='SQL*Net message from client'…

This is the kind of trace data the kernel produces.

Oracle extended SQL trace data

```
WAIT #42: nam='latch: library cache'…

PARSE #42:c=10000,…


WAIT #42: nam='SQL*Net message to client'…
EXEC #42:c=10000,…
WAIT #42: nam='SQL*Net message from client'…


WAIT #42: nam='latch: cache buffers chains'…

WAIT #42: nam='SQL*Net message to client'…
FETCH #42:c=20000,…
WAIT #42: nam='SQL*Net message from client'…


WAIT #42: nam='SQL*Net message to client'…
WAIT #42: nam='SQL*Net more data to client'…
WAIT #42: nam='SQL*Net more data to client'…
FETCH #42:c=20000,…
WAIT #42: nam='SQL*Net message from client'…


WAIT #42: nam='SQL*Net message to client'…
WAIT #42: nam='SQL*Net more data to client'…
WAIT #42: nam='SQL*Net more data to client'…
FETCH #42:c=20000,…
WAIT #42: nam='SQL*Net message from client'…


WAIT #42: nam='SQL*Net message to client'…
WAIT #42: nam='SQL*Net more data to client'…
WAIT #42: nam='SQL*Net more data to client'…
FETCH #42:c=20000,…
WAIT #42: nam='SQL*Net message from client'…
```

Of course, you don't directly get to see the kernel code path.

...Or that helpful grid that I drew for you.

Oracle extended SQL trace data

```
WAIT #42: nam='latch: library cache'…

PARSE #42:c=10000,…


WAIT #42: nam='SQL*Net message to client'…
EXEC #42:c=10000,…
WAIT #42: nam='SQL*Net message from client'…


WAIT #42: nam='latch: cache buffers chains'…

WAIT #42: nam='SQL*Net message to client'…
FETCH #42:c=20000,…
WAIT #42: nam='SQL*Net message from client'…


WAIT #42: nam='SQL*Net message to client'…
WAIT #42: nam='SQL*Net more data to client'…
WAIT #42: nam='SQL*Net more data to client'…
FETCH #42:c=20000,…
WAIT #42: nam='SQL*Net message from client'…


WAIT #42: nam='SQL*Net message to client'…
WAIT #42: nam='SQL*Net more data to client'…
WAIT #42: nam='SQL*Net more data to client'…
FETCH #42:c=20000,…
WAIT #42: nam='SQL*Net message from client'…


WAIT #42: nam='SQL*Net message to client'…
WAIT #42: nam='SQL*Net more data to client'…
WAIT #42: nam='SQL*Net more data to client'…
FETCH #42:c=20000,…
WAIT #42: nam='SQL*Net message from client'…
```

All you get to see is this.

Oracle extended SQL trace data

```
WAIT #42: nam='latch: library cache'…
PARSE #42:c=10000,…
WAIT #42: nam='SQL*Net message to client'…
EXEC #42:c=10000,…
WAIT #42: nam='SQL*Net message from client'…
WAIT #42: nam='latch: cache buffers chains'…
WAIT #42: nam='SQL*Net message to client'…
FETCH #42:c=20000,…
WAIT #42: nam='SQL*Net message from client'…
WAIT #42: nam='SQL*Net message to client'…
WAIT #42: nam='SQL*Net more data to client'…
WAIT #42: nam='SQL*Net more data to client'…
FETCH #42:c=20000,…
WAIT #42: nam='SQL*Net message from client'…
WAIT #42: nam='SQL*Net message to client'…
WAIT #42: nam='SQL*Net more data to client'…
WAIT #42: nam='SQL*Net more data to client'…
FETCH #42:c=20000,…
WAIT #42: nam='SQL*Net message from client'…
WAIT #42: nam='SQL*Net message to client'…
WAIT #42: nam='SQL*Net more data to client'…
WAIT #42: nam='SQL*Net more data to client'…
FETCH #42:c=20000,…
WAIT #42: nam='SQL*Net message from client'…
```

Oracle kernel code path

```
begin prepare
     CPU
     latch-related syscall
     CPU
end prepare
begin exec
     CPU
     write(SQLNET_OUT, result_to_client);
end exec
read(SQLNET_IN, next_request_from_client);
begin fetch
     CPU
     latch-related syscall
     CPU
     write(SQLNET_OUT, result_to_client);
end fetch
read(SQLNET_IN, next_request_from_client);
begin fetch
     CPU
     write(SQLNET_OUT, result_to_client);
     write(SQLNET_OUT, more_results);
     write(SQLNET_OUT, more_results);
end fetch
read(SQLNET_IN, next_request_from_client);
begin fetch
     CPU
     write(SQLNET_OUT, result_to_client);
     write(SQLNET_OUT, more_results);
     write(SQLNET_OUT, more_results);
end fetch
read(SQLNET_IN, next_request_from_client);
begin fetch
     CPU
     write(SQLNET_OUT, result_to_client);
     write(SQLNET_OUT, more_results);
     write(SQLNET_OUT, more_results);
end fetch
read(SQLNET_IN, next_request_from_client);
```

Oracle extended SQL trace data

```
WAIT #42: nam='latch: library cache'…
PARSE #42:c=10000,…
WAIT #42: nam='SQL*Net message to client'…
EXEC #42:c=10000,…
WAIT #42: nam='SQL*Net message from client'…
WAIT #42: nam='latch: cache buffers chains'…
WAIT #42: nam='SQL*Net message to client'…
FETCH #42
WAIT #42:                                    nt'…
WAIT #42:                                    '…
WAIT #42:                                    nt'…
WAIT #42:                                    nt'…
FETCH #42
WAIT #42:                                    nt'…
WAIT #42:                                    '…
WAIT #42:                                    nt'…
WAIT #42:                                    nt'…
FETCH #42
WAIT #42:                                    nt'…
WAIT #42:                                    '…
WAIT #42:                                    nt'…
WAIT #42:                                    nt'…
FETCH #42
WAIT #42:                                    nt'…
```

You can learn to envision the kernel's code path that motivated your trace file.

There are lots of ways to summarize a trace file.

tkprof
SQL Developer [Trace] Viewer
Trace Analyzer
tvdxstat
xtrace
OraSRP
Method R Workbench

IT'S A SOLVED PROBLEM.

Profiling trace files can be easy.

You can build tools, or you can buy them.

What you can do with trace files

Example 1

```
mrskew "r1-fixed.trc"
```

CALL-NAME	DURATION	%	CALLS	MEAN	MIN	MAX
SQL*Net message from client	1,403.927942	99.7%	2,161	0.649666	0.000000	0.927028
FETCH	3.013549	0.2%	2,161	0.001395	0.000000	0.005000
direct path read temp	1.259022	0.1%	83	0.015169	0.003287	0.046968
SQL*Net more data to client	0.141213	0.0%	2,460	0.000057	0.000005	0.001269
SQL*Net message to client	0.007964	0.0%	2,161	0.000004	0.000001	0.000376
TOTAL (5)	1,408.349690	100.0%	9,026	0.156033	0.000000	0.927028

99.7% of the time is 2,161 network round-trips.

What SQL statements cause the round-trips?

```
mrskew --group='($sqlid=~/^#/?"":"[".$sqlid."]")' --gl=SQLID --name='message from client' "r1-fixed.trc"

SQLID                DURATION       %  CALLS      MEAN       MIN       MAX
---------------  ------------  ------  -----  --------  --------  --------
[7d0bv6ds85q1f]  1,403.927942  100.0%  2,161  0.649666  0.000000  0.927028
---------------  ------------  ------  -----  --------  --------  --------
TOTAL (1)        1,403.927942  100.0%  2,161  0.649666  0.000000  0.927028
```

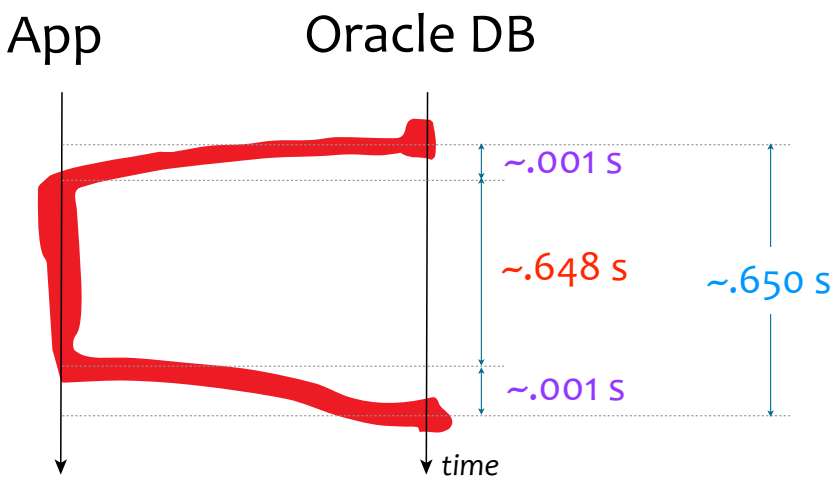
Just one. All 2,161 round-trips are executed on behalf of just one SQL statement.

```
mrskew --rc=p10.rc --name='SQL\*Net message from client' "r1-fixed.trc"
```

	RANGE {min ≤ e < max}		DURATION	%	CALLS	MEAN	MIN	MAX
1.	0.000000	0.000001	0.000000	0.0%	1	0.000000	0.000000	0.000000
2.	0.000001	0.000010						
3.	0.000010	0.000100						
4.	0.000100	0.001000						
5.	0.001000	0.010000						
6.	0.010000	0.100000						
7.	0.100000	1.000000	1,403.927942	100.0%	2,160	0.649967	0.547110	0.927028
8.	1.000000	10.000000						
9.	10.000000	100.000000						
10.	100.000000	1,000.000000						
11.	1,000.000000	+∞						
	TOTAL (11)		1,403.927942	100.0%	2,161	0.649666	0.000000	0.927028

Each round-trip consumes an average of .649967 ≈ .650 s.

Why?



Each *SQL*Net message from client* call (~.650 s) looks like this.

If round-trip network latency is ~.002 s, then this experience is spending ~.648 s in the Java code executed between database calls.

```
mrskew --name=':dbcall' --select='$row' --slabel=ROWS --precision=0 "r1-fixed.trc"

CALL-NAME     ROWS       %  CALLS  MEAN  MIN  MAX
---------  -------  ------  -----  ----  ---  ---
FETCH      216,017  100.0%  2,161   100   17  100
---------  -------  ------  -----  ----  ---  ---
TOTAL (1)  216,017  100.0%  2,161   100   17  100
```

One final check...

The trace file shows that the application, at least, is fetching an average of 100 rows per fetch call (per round-trip).

This helps explain the Java-side latency, but still, .648 s to process just 100 rows needs some explaining.

```
mrskew "r1-fixed.trc"

CALL-NAME                       DURATION       %  CALLS      MEAN       MIN       MAX
-----------------------------  ------------  ------  -----  --------  --------  --------
SQL*Net message from client  1,403.927942   99.7%  2,161  0.649666  0.000000  0.927028
FETCH                            3.013549    0.2%  2,161  0.001395  0.000000  0.005000
direct path read temp            1.259022    0.1%     83  0.015169  0.003287  0.046968
SQL*Net more data to client      0.141213    0.0%  2,460  0.000057  0.000005  0.001269
SQL*Net message to client        0.007964    0.0%  2,161  0.000004  0.000001  0.000376
-----------------------------  ------------  ------  -----  --------  --------  --------
TOTAL (5)                    1,408.349690  100.0%  9,026  0.156033  0.000000  0.927028
```

No matter how long you try to "fix the database" here, you're going to see at most only a .3% difference in response time.

The problem here is in the Java.

Example 2

```
mrskew "prd1_ora_9031.trc"
```

CALL-NAME	DURATION	%	CALLS	MEAN	MIN	MAX
PARSE	735.426197	78.9%	698	1.053619	0.000000	4.498316
SQL*Net message from client	104.762229	11.2%	1,378	0.076025	0.000391	3.554818
FETCH	91.800028	9.8%	680	0.135000	0.000000	0.506923
db file sequential read	0.104670	0.0%	14	0.007476	0.001067	0.016408
EXEC	0.083988	0.0%	349	0.000241	0.000000	0.002000
gc cr block 2-way	0.073233	0.0%	96	0.000763	0.000280	0.001968
gc current block 2-way	0.031298	0.0%	47	0.000666	0.000361	0.001640
gc current grant busy	0.028037	0.0%	47	0.000597	0.000156	0.001508
SQL*Net more data from client	0.025819	0.0%	837	0.000031	0.000000	0.002564
CLOSE	0.018999	0.0%	698	0.000027	0.000000	0.001000
12 others	0.061576	0.0%	1,633	0.000038	0.000000	0.001687
TOTAL (22)	932.416074	100.0%	6,477	0.143958	0.000000	4.498316

PARSE calls account for 78.9% of the experience duration.

That is *never* appropriate.

```
mrskew --rc=p10.rc --name=parse "prd1_ora_9031.trc"
```

	RANGE {min ≤ e < max}		DURATION	%	CALLS	MEAN	MIN	MAX
1.	0.000000	0.000001	0.000000	0.0%	307	0.000000	0.000000	0.000000
2.	0.000001	0.000010						
3.	0.000010	0.000100						
4.	0.000100	0.001000	0.007992	0.0%	8	0.000999	0.000999	0.000999
5.	0.001000	0.010000	0.033000	0.0%	33	0.001000	0.001000	0.001000
6.	0.010000	0.100000						
7.	0.100000	1.000000						
8.	1.000000	10.000000	735.385205	100.0%	350	2.101101	1.333797	4.498316
9.	10.000000	100.000000						
10.	100.000000	1,000.000000						
11.	1,000.000000	+∞						
	TOTAL (11)		735.426197	100.0%	698	1.053619	0.000000	4.498316

That's a lot of time spent parsing, and these PARSE calls are *really* expensive.

```
mrskew --name=parse --group='$sqlid' --gl=SQLID --sort=4nd "prd1_ora_9031.trc"
```

SQLID	DURATION	%	CALLS	MEAN	MIN	MAX
gkbss8w49204k	4.176363	0.6%	349	0.011967	0.000000	4.135371
66kf30526wrgy	3.153521	0.4%	1	3.153521	3.153521	3.153521
3r3dhkb0z824v	2.911558	0.4%	1	2.911558	2.911558	2.911558
3tzra8a2a7pny	1.605757	0.2%	1	1.605757	1.605757	1.605757
2hycpfzdzsu98	3.155520	0.4%	1	3.155520	3.155520	3.155520
6ppu3s1jszy3a	2.208665	0.3%	1	2.208665	2.208665	2.208665
66vkb784j9rcu	1.901711	0.3%	1	1.901711	1.901711	1.901711
5wamvs45j6nh4	1.492773	0.2%	1	1.492773	1.492773	1.492773
dj1buvhxg7h19	1.499772	0.2%	1	1.499772	1.499772	1.499772
41yrts4g94ghn	1.628753	0.2%	1	1.628753	1.628753	1.628753
340 others	711.691804	96.8%	340	2.093211	1.333797	4.498316
TOTAL (350)	735.426197	100.0%	698	1.053619	0.000000	4.498316

One statement was parsed 349 times; at least 348 of those are unnecessary.*

There are 350 distinct SQL statements executed by this report. ...Which is funny, because you know this report, and you don't remember there being that many.

*Actually all 349 are unnecessary, because the trace file shows that there's never an EXEC call associated with any of these PARSE calls, but that's a story for another day.

```
mrskew --rc=distinct-texts.rc "prd1_ora_9031.trc"
```

SSQLID	DISTINCT-TEXTS	%	CALLS	MEAN	MIN	MAX
4151812497	70	20.0%	70	1	1	1
3642320257	70	20.0%	70	1	1	1
2047770123	70	20.0%	70	1	1	1
1928547239	70	20.0%	70	1	1	1
1138917066	69	19.7%	69	1	1	1
3957414185	1	0.3%	349	0	0	1
TOTAL (6)	350	100.0%	698	1	0	1

For the first 5 "shared SQL ID" values shown here, there are ~70 distinct statements that could have been sharable.

You should be able to reduce the parse call count from 698 to 6, by writing sharable SQL statements, and pulling PARSE calls out of loops.

```
mrskew "prd1_ora_9031.trc"
```

CALL-NAME	DURATION	%	CALLS	MEAN	MIN	MAX
PARSE	735.426197	78.9%	698	1.053619	0.000000	4.498316
SQL*Net message from client	104.762229	11.2%	1,378	0.076025	0.000391	3.554818
FETCH	91.800028	9.8%	680	0.135000	0.000000	0.506923
db file sequential read	0.104670	0.0%	14	0.007476	0.001067	0.016408
EXEC	0.083988	0.0%	349	0.000241	0.000000	0.002000
gc cr block 2-way	0.073233	0.0%	96	0.000763	0.000280	0.001968
gc current block 2-way	0.031298	0.0%	47	0.000666	0.000361	0.001640
gc current grant busy	0.028037	0.0%	47	0.000597	0.000156	0.001508
SQL*Net more data from client	0.025819	0.0%	837	0.000031	0.000000	0.002564
CLOSE	0.018999	0.0%	698	0.000027	0.000000	0.001000
12 others	0.061576	0.0%	1,633	0.000038	0.000000	0.001687
TOTAL (22)	932.416074	100.0%	6,477	0.143958	0.000000	4.498316

Before your boss will let you "fix" this code, you have to predict the benefit.

Reducing the parse count from 698 to 6 should reduce parsing duration from ~735 to ~7, a savings of about 730 s. Response time should improve from ~932 s to ~200 s, just from eliminating the PARSE calls only.

You might have known that you should "use bind variables," but you couldn't have quantified the *R* impact on the experience without this trace file.

BASELINE: **BAD**

```
for each invoice number {
      cursor = parse("select ...where invoice_number = " + number);
      exec(cursor);
      loop over the result set to fetch all the rows;
}
```

This is horrific:

- Uses too much CPU for PARSE calls
- Serialization on *library cache* and *shared pool* latches
- Consumes too much memory in the library cache
- May execute too many network round-trips

BASELINE: **BAD**

```
for each invoice number {
      cursor = parse("select ...where invoice_number = " + number);
      exec(cursor);
      loop over the result set to fetch all the rows;
}
```

FIX 1 "Hey, let's use bind variables": **STILL BAD**

```
for each invoice number {
      cursor = parse("select ...where invoice_number = :a1");
      exec(cursor, number);
      loop over the result set to fetch all the rows;
}
```

A little better, but still really awful:

- Uses too much CPU for PARSE calls
- Serialization on *library cache* latches
- Maybe, too many network round-trips

FIX 1 “Hey, let’s use bind variables”: **STILL BAD**

```
for each invoice number {
      cursor = parse(“select ...where invoice_number = :a1”);
      exec(cursor, number);
      loop over the result set to fetch all the rows;
}
```

FIX 2: **BETTER**

```
cursor = parse(“select ...where invoice_number = :a1”);
for each invoice number {
      exec(cursor, number);
      loop over the result set to fetch all the rows;
}
```

Better (only 1 PARSE call now!), but still lots of network round-trips.

FIX 2: **BETTER**

```
cursor = parse("select ...where invoice_number = :a1");
for each invoice number {
      exec(cursor, number);
      loop over the result set to fetch all the rows;
}
```

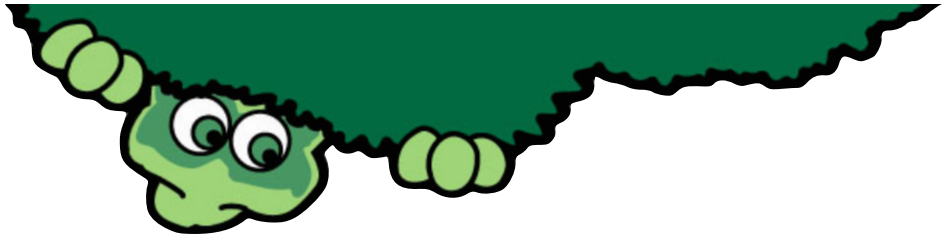
FIX 3: **BETTER YET**

```
cursor = parse("
      select ...where invoice_number
      in (select invoice number from wherever your for each was getting them)
");
exec(cursor);
loop over the result set to fetch all the rows;
```

Now, only 1 PARSE call, only 1 EXEC call, *and* the minimum possible number of network round-trips.*

*Unless there's a way to return fewer rows.

And so on...



A **trace file** shows you where your **time** has gone.

Performance **problems cannot hide** from that.

Bad SQL
Bad PL/SQL
Slow network
Missing indexes
Parsing in a loop
Hot block problems
Not enough memory
Disk latency problems
Row locking problems
Row-at-a-time processing
Bad data structure choice
Hardware misconfigurations
Too much load on the system
OS parameters set inadequately
Oracle parameters set inadequately
SQL returns more rows than it should
Database buffer cache hot/cold problems
Oracle query optimizer choosing bad plans
Reports run with poorly limiting parameter values
Inefficient code between database calls in the application

#ProTip

There are **only two** *possible* root causes for **any** response time problem:

1. Call count is too big.
2. Latency is too big.*

*Probably because someone else's call counts are too big.

```
mrskew --top=10 "prd1_ora_9031.trc"
```

CALL-NAME	DURATION	%	CALLS	MEAN	MIN	MAX
PARSE	735.426197	78.9%	698	1.053619	0.000000	4.498316
SQL*Net message from client	104.762229	11.2%	1,378	0.076025	0.000391	3.554818
FETCH	91.800028	9.8%	680	0.135000	0.000000	0.506923
db file sequential read	0.104670	0.0%	14	0.007476	0.001067	0.016408
EXEC	0.083988	0.0%	349	0.000241	0.000000	0.002000
gc cr block 2-way	0.073233	0.0%	96	0.000763	0.000280	0.001968
gc current block 2-way	0.031298	0.0%	47	0.000666	0.000361	0.001640
gc current grant busy	0.028037	0.0%	47	0.000597	0.000156	0.001508
SQL*Net more data from client	0.025819	0.0%	837	0.000031	0.000000	0.002564
CLOSE	0.018999	0.0%	698	0.000027	0.000000	0.001000
12 others	0.061576	0.0%	1,633	0.000038	0.000000	0.001687
TOTAL (22)	932.416074	100.0%	6,477	0.143958	0.000000	4.498316

See how there are only two ways to reduce a DURATION? You have the CALLS column, and the MEAN column.

Profiles like this make it easy to see how *anything* you do to make something go faster *must* translate to a manipulation of either CALLS or MEAN.

#ProTip

With a good trace file, you can **predict** the response time impact of a proposed change.*

*This is nearly impossible to do with systemwide aggregated statistics.

It just takes practice.

Conclusion

Your code does stuff.

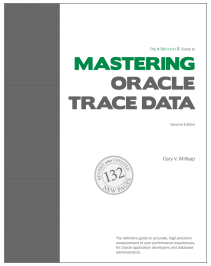
Including some stuff inside Oracle.

The time this stuff takes is your user's response time.

You can see exactly what it is.

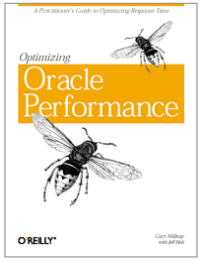
It's not that hard.

References



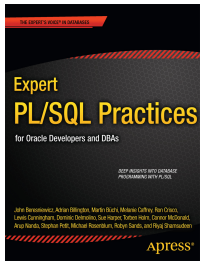
Cary Millsap. 2016.
Mastering Oracle Trace Data, 2ed.
Method R Corporation

Textbook for 1-day course that teaches you how to master Oracle trace data.



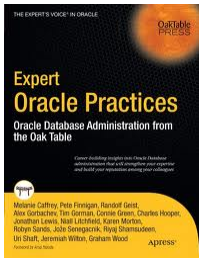
Cary Millsap, Jeff Holt. 2003.
Optimizing Oracle Performance.
O'Reilly

Detailed information about Oracle trace data and what to do with it.



Ron Crisco, et al. 2011.
Expert PL/SQL Practices.
Apress

Detailed information about instrumenting your Oracle application code.



Robyn Sands, et al. 2010.
Expert Oracle Practices.
Apress

Detailed information about instrumenting your Oracle application code.

Q&A

◇ Method R™

method-r.com

method-r.com/facebook

@MethodR

cary.millsap@method-r.com

www.cintra.com

@CintraSoftware

cmillsap@cintra.com