Method R Software White Paper № 1

A First Look at Using Method R Workbench Software

By Cary Millsap, Method R Corporation

Audience: Developers of Oracle-based application software, Oracle database administrators, Oracle performance analysts, and project leaders who care about Oracle system performance.

Many people underestimate the value and overestimate the difficulty of Oracle tracing. Tracing is unique in how well it directly connects objective performance data to end-user performance experiences. This makes it the ideal bridge among a business's users, its software developers, and its operational runtime support staff. With the right understanding and tools, tracing is no more difficult to obtain than other forms of Oracle diagnostic data. This paper demonstrates how to use the Method R Workbench software package to convert trace files into profiles, repair an improperly collected trace file, use profiles to diagnose an Oracle-based software performance problem, predict the effect of proposed remedies upon the end-user performance experience, fix the problem, assess the fix for potential side-effects, and then measure the end-user experience after the fix.

Contents

Revised 2012-12-20

The Trace Data Perspective

Oracle professionals use many kinds of data to diagnose software performance problems. My favorite is Oracle's extended SQL trace feature. I use it almost exclusively. Here's why:

- *It connects me directly to the user experience.* When users describe performance problems, they usually talk about experiences: "It takes 30 minutes to run the TPS report. I want it to run in 1 minute like it used to." I want performance data that connects me directly to that user experience. I want to see what my user's code did, without being deceived by aggregations or distracted by activity outside the scope of my analysis. Tracing gives me the richest available description of how an Oracle-based application consumes time, and I can focus the measurement upon any user's experience that I want to measure. It's the best data we have for connecting developers and operators to the performance issues of the business.
- *It allows me to predict what users will experience as I fix things.* Before I start fixing things, I want to know how much time each of the remedies I'm considering will save for my user—in hours, minutes, and seconds. I need to be able to estimate the benefit I'll create before I start spending my time (or my customer's money) trying things out. Tracing helps me do that.
- *It gives me the detail that I need.* When I diagnose whether a system is working efficiently, I need to see how the application's code path works, in detail. Tracing gives me a step-by-step account of the interaction between applications and their databases. This lets me have very detailed conversations with application developers, database administrators, operating system administrators, network administrators, storage administrators, and so on, whenever I need to.
- *It's always there when I need it.* I want performance data that I can access on every version and every edition of Oracle, and I want to use the same diagnostic methods and tools for all the Oracle systems I work on. The extended SQL trace feature is available with every edition of the Oracle Database from Express Edition (XE) to Exadata, in every release since 7.0, and I can use it regardless of which Oracle add-ons you've bought.
- *Tracing is completely programmable.* An application developer can design programs to make tracing really easy. You can even make programs trace themselves automatically when they are most at risk of behaving poorly.

Trace Data Difficulties

Tracing connects me to the experiences my users are feeling, and it gives me advantages of focus, predictability, detail, reliability, and control. However, using trace data is more difficult than viewing data in a graphical dashboard. Certainly, without tools to aid you, tracing is quite a bit more work. Here are some of the difficulties people have with tracing:

- *Activating and deactivating the tracing feature requires care.* A subplot in this paper is an example of this. Collecting trace data is quite easy; collecting the *right* trace data is more challenging. However, learning how to collect trace data correctly is not a significant obstacle for anyone willing to try. With tools like the Method R Trace extension for the Oracle SQL Developer interactive development environment, the Method R Instrumentation Library for Oracle (ILO), Oracle triggers, and applications with tracing features built right in, tracing is no problem at all.

- *To trace it, you'll probably have to rerun it.* When a user has a performance problem, odds are that to get a trace file for the experience, you'll have to run her program again. Advocates of always-on diagnostics point this out as a significant limitation of tracing. However, I don't see it that way. To debug a problem (a performance error *or* a functional error), you need a reproducible test case; if you don't have one, then how can you ever demonstrate that you've fixed the problem? A reproducible test case thus is a requirement for a problem-solving project, and if you have one, then having to run some program a second time isn't a limitation.
- *Finding and fetching the right trace file is tedious.* If every time you need a trace file, you have to talk to somebody (because your DBA won't give you permissions to find and fetch your trace file yourself), then getting trace files is not just tedious, it's *agonizing*. Even if you have all the privileges you need to go get your own trace files, the mental context switch overhead of switching applications to find and transfer your trace file from the database server to your desktop is just another unwanted friction in your day. However, the burden of finding and fetching trace files can be eliminated through software automation. Method R Trace makes the problem go away for developers and analysts who use Oracle SQL Developer. Your application programmers can make it go away, too; finding and fetching files is all programmable.
- *Interpreting trace file data can be difficult.* There aren't many good references that explain Oracle trace data, and anyway, you would never get much done if you had to analyze 1-GB trace files with a spreadsheet and a text editor. Oracle's **tkprof** is meant to do some of the work for you, but there's a lot it doesn't do. The point of this paper is to show how my company's Method R Profiler and Method R Tools software can help you learn everything your trace files can teach you.

None of these difficulties overwhelms the value of tracing, which allows me to connect my technical measurements directly to the end-user's performance experience. Unfortunately, the inconveniences stop some people from trying to use trace data when it would be the perfect data source for them.

My team and I—who use trace data almost every day—don't like those inconveniences any more than you would, so we've spent lots of time building tools to overcome all the little bothers that go with tracing. This paper covers some of those tools and how we use them. My aim is to show you how my colleagues and I trace Oracle-based software to solve problems and make better software.

The Case

The case I'll walk you through is a miniature replica of a problem that many of our customers have had. The mistake I'll describe is an easy one to make. Most tools are poor at detecting it, and many DBAs don't understand how to fix it.

Because I want to show you so many new things at once, I am going to use a very simple **sqlplus** program to demonstrate the problem I want to describe. Please don't be too concerned about risks of oversimplification. More complicated programs suffer from the same kind of problem that I'll illustrate here, and those programs will relent to the same kind of analysis that I'll show you. ...Maybe, like many of our customers, you'll find out that your performance problems are simpler than you thought.

This case starts like most of the ones I've worked on: a user has complained that a program he runs takes too long: about half a minute. He wants it to run faster. I asked him to trace his experience, and he gave me a 25.7-MB trace file called **slow.trc**. Here we go.

The Response Time Profile

The first thing I do with a trace file is run it through the Method R Profiler. The resulting Profile Report is a 97-KB HTML document. It's a few pages long, but hyperlinks make the report easy to navigate. Section 1.1 of the report is a table called the Profile by Subroutine. It explains the response time represented in the trace file, grouped by subroutine calls that have been executed by the Oracle Database kernel.

Profile by subroutine for the trace file that I had hoped would describe the user's 32-second experience. A thumbnail of the entire Profile Report is shown at right.

	Subroutine	Duration seconds	Duration % R	Cumulative duration seconds	Cumulative duration % R	Call count	Duration per call (seconds) mean	min	skew	max	Drill-down
1.	**SQL*Net message from client [think time]**	**54.415**	**63.1%**	54.415	63.1%	**2**	27.207605	26.294757		28.120454	SQL
2.	**SQL*Net message from client**	**22.181**	**25.7%**	76.596	88.9%	**75,000**	0.000296	0.000196		0.095116	SQL
3.	unaccounted-for between dbcalls	6.292	7.3%	82.888	96.2%	150,038	0.000042	-0.000037		0.326858	SQL
4.	unaccounted-for within dbcalls	2.584	3.0%	85.472	99.2%	75,101	0.000034	-0.011754		0.019578	
5.	CPU service, FETCH calls	0.588	0.7%	86.060	99.9%	75,014	0.000008	0.000000		0.212014	
6.	CPU service, unreported call(s)	0.230	0.3%	86.289	100.1%	1	0.229538	0.229538		0.229538	
7.	SQL*Net message to client	0.090	0.1%	86.379	100.2%	75,002	0.000001	0.000000		0.002551	
8.	CPU service, EXEC calls	0.012	0.0%	86.391	100.2%	31	0.000387	0.000000		0.012000	
9.	db file scattered read [blocks ≤ 16]	0.006	0.0%	86.397	100.3%	72	0.000081	0.000024		0.000166	
10.	db file scattered read [16 < blocks ≤ 32]	0.002	0.0%	86.399	100.3%	9	0.000233	0.000158		0.000348	
11.	db file sequential read	0.001	0.0%	86.400	100.3%	13	0.000040	0.000020		0.000099	
12.	log file sync	0.000	0.0%	86.400	100.3%	1	0.000165	0.000165		0.000165	
13.	Disk file operations I/O	0.000	0.0%	86.400	100.3%	1	0.000128	0.000128		0.000128	
14.	CPU service, PARSE calls	0.000	0.0%	86.400	100.3%	25	0.000000	0.000000		0.000000	
15.	CPU service, CLOSE calls	-0.224	-0.3%	86.176	100.0%	32	-0.007000	-0.224014		0.000000	
16.	Total	86.176	100.0%								

It's bad news at the bottom of this profile, because I wanted an explanation for why the user's experience lasted about 32 seconds, but this table explains an 86.176-second experience. *This is not a measure of my user's experience*. Now I have to figure out which data to pay attention to, and which to ignore.

The red subroutines named in the top of the report (the ones with the longest durations) are where I always begin my investigation:

- **SQL*Net message from client [think time]** calls, quantity 2, consumed 54.415 seconds, or 63.1% of the measured 86.176 seconds. Since there are only two, I can tell by the min-max data that one is 26.294757 seconds long, and the other is 28.120454 seconds long.
- **SQL*Net message from client** calls, quantity 75,000, consumed 22.181 seconds, or 25.7% of the measured 86.176 seconds.

You may have been taught that you should ignore all **SQL*Net message from client** events because they represent "database idle time." In this case, it's tempting to do that, because obviously there's more time accounted for here than I want; however, when you work with profiles that describe user response time experiences, discarding all your **SQL*Net message from client** calls is not what you'll want to do.

Here, if I discard *all* the **SQL*Net message from client** duration, I'll be left with an explanation for only about 9 seconds of response time (86 – (54 + 22) = 9). That's not enough to explain my user's 32-second experience. However, if I discard only the two calls labeled "[think time]," what I'd have left would be just right:

86.176	Profile duration
–54.415	**SQL*Net message from client [think time]** duration
31.761	User's experience duration

Tempting. ...But what would it *mean* to discard those two calls?

SQL*Net message from client and "Think Time"

The events that most Oracle documentation calls "wait events" are operating system calls (syscalls) to which Oracle kernel developers have assigned special Oracle names. For example, Oracle calls a **pread** on my Linux system a **db file sequential read** event; I have verified this by using the Linux **strace** utility.

The durations that Oracle reports are syscall response times measured by the Oracle kernel like this:

t_0 = timestamp
Oracle kernel makes a syscall (for example, a **pread**)
t_1 = timestamp
$t_1 - t_0$ is the measured duration of the call

When you're tracing an Oracle process that makes syscalls, that process writes lines beginning with the word "WAIT" into a trace file. For example, when an Oracle kernel process completes a **pread** on my Linux system to read a block from a database file, it writes a trace file line that looks like this:

```
WAIT #5362568: nam='db file sequential read' ela= 66 file#=4 block#=17964 blocks=1 obj#=115132 tim=1352932110079701
```

This line says that my **db file sequential read** call consumed 66 microseconds (µs) of elapsed duration and concluded at 16:28:30.079701 CST on 2012-11-14. I know this because the **mrtim** tool in the Method R Tools suite reports a human-readable date and time for a given **tim** value.

Likewise, the following line means something similar:

```
WAIT #5363184: nam='SQL*Net message from client' ela= 246 driver id=1413697536 #bytes=1 p3=0 obj#=115132 tim=1352932110223333
```

This means that the Oracle kernel executed a **read** call upon the file handle to which my application client is connected. This call consumed 246 µs of elapsed time and concluded at 16:28:30.223333 CST on 2012-11-14. **SQL*Net message from client** duration is no different from any other duration in the following regard: if a call contributes to a user's response time experience, it is important; otherwise, it is not.

If a call contributes to a user's response time experience, it is important; otherwise, it is not.

So, the right question to ask is not whether an event is "idle"; the right question is whether an event contributed to a user's response time experience. Some idle events contribute to user response times, and some do not.

The name **SQL*Net message from client** with the string "**[think time]**" appended to it is a name created by the Method R Profiler. By default, calls named this are defined as **SQL*Net message from client** calls that have individual durations of 1.0 seconds or longer. This definition is a Profiler configuration option that you can change if you want, but most people never need to. When the Oracle Database blocks on a **SQL*Net message from client** call for a second or more, it often means that time is being consumed either by client-tier code path, or by a user's brain. It's time you may not want in your profile.

What's Wrong with the Trace?

My user's experience lasted about 32 seconds, but my trace file explains an 86.176-second experience. If I ignore the two **SQL*Net message from client [think time]** calls, it

looks like my profile will come out just right. Is it okay to ignore those two calls? Where did they come from?

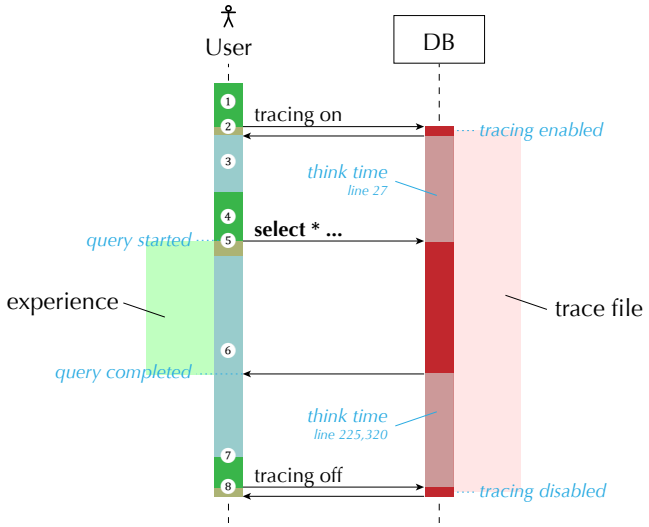
The two calls are the result of a mistake in how the trace file was collected. Here's what the trace file looks like:

```
     1. Trace file /opt/oracle/diag/rdbms/v11202/V11202/trace/V11202_ora_24517.trc

        … # Events that happened before the user's experience began are recorded here on lines 1 through 27.

    26. *** 2012-11-14 16:28:30.069
    27. WAIT #5184972: nam='SQL*Net message from client' ela= 26294757 … tim=1352932110069389

        … # Events that happened during the user's experience happened here, on lines 28 through 225,318.

225319. *** 2012-11-14 16:29:29.720
225320. WAIT #5363184: nam='SQL*Net message from client' ela= 28120454 … tim=1352932169720041

        … # Events that happened after the user's experience ended are recorded here on lines 225,319 through 225,676.
```

This trace file contains information about more time than just the user's experience because he collected it like this:

1. The user typed an **alter session** command from **sqlplus** to enable the trace.
2. He executed the statement by pressing Enter.
3. He attended to an email for a few seconds.
4. The user typed the slow SQL statement into **sqlplus**.
5. He executed the statement by pressing Enter.
6. He returned again to his email.
7. He noticed that the statement had completed.
8. He exited **sqlplus** to disable the trace.

Sequence diagram illustrating how my user collected the 86.176-second trace file.



Here's the problem: the experience I need to fix begins with step 5 and ends somewhere during step 6, but the trace file explains everything from step 2 through step 8. The two "think time" events dominate the illustration. The **SQL*Net message from client [think time]** call on line 27 describes the duration of steps 3 through 5. The

SQL*Net message from client [think time] call on line 225,320 describes the duration from when the statement completed until my user completed step 8.

User response time experiences usually contain short-latency **SQL*Net message from client** calls; they're the network roundtrips that represent normal communications between an application and its database. However, multi-second **SQL*Net message from client** calls usually indicate either a trace file that was collected carelessly or an application tier that executes a lot of code path between database calls. In this case, it was just a little bit of uninformed carelessness in how the trace file was created.

The Method R Profiler feature of grouping these *think time* calls into a separate row from the other **SQL*Net message from client** calls helps me envision what the trace file would look like if these calls weren't in the profile. I can see exactly how much time these two calls consumed, distinct from the time consumed by sub-second **SQL*Net message from client** calls. However, I would really prefer to eliminate these calls entirely; otherwise, I'll have to subtract unwanted time and rework the percentages every time I create a profile. They just complicate my analysis and make it more difficult for me to explain to my colleagues what I'm doing.

So do I need to ask my user to rerun the trace? Or is there something else I can do?

Repairing the Trace

Just deleting calls from a trace file sounds easy, right? You could just fire up your favorite text editor, find the two long-duration **SQL*Net message from client** call lines, delete them, and then rerun the Profiler? Well, if you do that, you'll get a nasty surprise:

Hand-editing the trace file does *not* lead to a profile that explains the user's 32-second experience.

	Subroutine	Duration seconds	Duration % R	Cumulative duration seconds	Cumulative duration % R	Call count	Duration per call (seconds) mean	min	skew	max	Drill-down
1.	**unaccounted-for between dbcalls**	**34.411**	**39.9%**	34.411	39.9%	**150,036**	0.000229	-0.000037		28.120685	SQL
2.	**CPU service, unreported call(s)**	**26.525**	**30.8%**	60.936	70.7%	**1**	26.524953	26.524953		26.524953	SQL
3.	**SQL*Net message from client**	**22.181**	**25.7%**	83.117	96.5%	**75,000**	0.000296	0.000196		0.095116	SQL
4.	unaccounted-for within dbcalls	2.584	3.0%	85.701	99.4%	75,101	0.000034	-0.011754		0.019578	
5.	CPU service, FETCH calls	0.588	0.7%	86.289	100.1%	75,014	0.000008	0.000000		0.212014	
6.	SQL*Net message to client	0.090	0.1%	86.379	100.2%	75,002	0.000001	0.000000		0.002551	
7.	CPU service, EXEC calls	0.012	0.0%	86.391	100.2%	31	0.000387	0.000000		0.012000	
8.	db file scattered read [blocks ≤ 16]	0.006	0.0%	86.397	100.3%	72	0.000081	0.000024		0.000166	
9.	db file scattered read [16 < blocks ≤ 32]	0.002	0.0%	86.399	100.3%	9	0.000233	0.000156		0.000348	
10.	db file sequential read	0.001	0.0%	86.400	100.3%	13	0.000040	0.000020		0.000099	
11.	log file sync	0.000	0.0%	86.400	100.3%	1	0.000165	0.000165		0.000165	
12.	Disk file operations I/O	0.000	0.0%	86.400	100.3%	1	0.000128	0.000128		0.000128	
13.	CPU service, PARSE calls	0.000	0.0%	86.400	100.3%	25	0.000000	0.000000		0.000000	
14.	CPU service, CLOSE calls	-0.224	-0.3%	86.176	100.0%	32	-0.007000	-0.224014		0.000000	
15.	Total	86.176	100.0%								

When you edit a trace file this way, the duration won't change. In this case, the Profiler still accounts for 86.176 seconds. The Profiler can tell that the clock is moving (because each line in the trace file representing a dbcall or a syscall has a timestamp on it), but now that I've deleted two lines, the trace file doesn't give any way to account for the durations of the two calls that elapsed.

It takes considerably more work to fix the trace file properly. Without a tool to do it for me, this is the point in the project where I'd go back to the user for a better trace file. ...But I never liked doing that. Users have better things to do than submit to what sounds like perfectionist compulsions to capture trace data again and again until we get

it just exactly right. Surely, the original trace file has everything in there that I need; I just need to do a better job of taking the nut out of the shell.

This problem is why we built the Method R Tools utility called **mrcallrm**. It takes a file name and list of line numbers containing calls I wish weren't in the file, and **mrcallrm** sets their durations to zero. It also adjusts the appropriate dates and timestamps in the trace file, to make it look as though those calls had run with zero-second latencies to begin with.

So then, how do I know which line numbers to list? I've already shown you the numbers of the lines I want to eliminate (27 and 225,320), but how did I figure it out? The **mrskew** utility (part of Method R Tools) gives me exactly the report I need; it's the "Response time by line number for a given call name pattern" report. Running that report on my **slow.trc** file, using the default regular expression pattern « **SQL*Net message from client** » gives me the line numbers I want:

```
Response time by line number for given call name pattern
mrskew --group=$line --glabel=LINE --name=SQL\*Net message from client "…/slow.trc"
2012-11-28T15:53:23.000926-0600
Elapsed: 8.032000 s

          LINE   DURATION       %   CALLS       MEAN        MIN        MAX
--------------  ---------  ------  ------  ---------  ---------  ---------
        225320  28.120454   36.7%       1  28.120454  28.120454  28.120454
            27  26.294757   34.3%       1  26.294757  26.294757  26.294757
        128369   0.095116    0.1%       1   0.095116   0.095116   0.095116
        126077   0.085262    0.1%       1   0.085262   0.085262   0.085262
        136396   0.085224    0.1%       1   0.085224   0.085224   0.085224
        127223   0.085157    0.1%       1   0.085157   0.085157   0.085157
        134137   0.075118    0.1%       1   0.075118   0.075118   0.075118
        151048   0.075102    0.1%       1   0.075102   0.075102   0.075102
        124391   0.065070    0.1%       1   0.065070   0.065070   0.065070
        137472   0.064926    0.1%       1   0.064926   0.064926   0.064926
 74,992 others  21.550131   28.1%  74,992   0.000287   0.000196   0.060859
--------------  ---------  ------  ------  ---------  ---------  ---------
TOTAL (75,002)  76.596317  100.0%  75,002   0.001021   0.000196  28.120454
```

The first two calls listed here contribute nearly 30 seconds apiece, whereas each of the remaining calls contributes less than 100 milliseconds apiece. The two high-latency calls on lines 225,320 and 27 are my think time events that I want to eliminate, and the other calls were part of my user's response time. The **mrcallrm** command I need is thus:

```
    mrcallrm --lines=225320,27 slow.trc >slow-mrcallrm.trc
```

This command creates a new file called **slow-mrcallrm.trc** that still contains the two high-latency **SQL*Net message from client** calls, but now those calls' durations have been set to zero, and the clock values throughout the file have been adjusted. Profiling this file gives the result I want.

Correcting the trace file with *mrcallrm* does create a profile by subroutine that explains the 32-second experience.

	Subroutine	Duration seconds	Duration % R	Cumulative duration seconds	Cumulative duration % R	Call count	Duration per call (seconds) mean	min	skew	max	Drill-down
1.	**SQL*Net message from client**	**22.181**	**69.8%**	22.181	69.8%	**75,002**	0.000296	0.000000		0.095116	SQL
2.	unaccounted-for between dbcalls	6.292	19.8%	28.473	89.6%	150,038	0.000042	-0.000037		0.326858	SQL
3.	unaccounted-for within dbcalls	2.584	8.1%	31.057	97.8%	75,101	0.000034	-0.011754		0.019578	SQL
4.	CPU service, FETCH calls	0.588	1.9%	31.645	99.6%	75,014	0.000008	0.000000		0.212014	
5.	CPU service, unreported call(s)	0.230	0.7%	31.874	100.4%	1	0.229538	0.229538		0.229538	
6.	SQL*Net message to client	0.090	0.3%	31.964	100.6%	75,002	0.000001	0.000000		0.002551	
7.	CPU service, EXEC calls	0.012	0.0%	31.976	100.7%	31	0.000387	0.000000		0.012000	
8.	db file scattered read [blocks ≤ 16]	0.006	0.0%	31.982	100.7%	72	0.000081	0.000024		0.000166	
9.	db file scattered read [16 < blocks ≤ 32]	0.002	0.0%	31.984	100.7%	9	0.000233	0.000156		0.000348	
10.	db file sequential read	0.001	0.0%	31.984	100.7%	13	0.000040	0.000020		0.000099	
11.	log file sync	0.000	0.0%	31.985	100.7%	1	0.000165	0.000165		0.000165	
12.	Disk file operations I/O	0.000	0.0%	31.985	100.7%	1	0.000128	0.000128		0.000128	
13.	CPU service, PARSE calls	0.000	0.0%	31.985	100.7%	25	0.000000	0.000000		0.000000	
14.	CPU service, CLOSE calls	-0.224	-0.7%	31.761	100.0%	32	-0.007000	-0.224014		0.000000	
15.	Total	31.761	100.0%								

Finally, I have an explanation for exactly the 31.761 seconds I want to explain. I can describe exactly where my user's time has gone. ...But ugh: the profile shows that 69.8% of his time was consumed by **SQL*Net message from client** calls. I made one problem go away by ignoring **SQL*Net message from client** calls; do I need to ignore some more of those calls?

Idle Events Revisited

The new profile leads to an interesting question. Which parts of a profile are safe to ignore? Can we discard all the so-called idle events?

A thought experiment yields the answer. Imagine sitting down with my user. You're appointed to help this guy. Look him square in the eye, and try to convince him that 22.181 seconds of his intolerably long 31.761-second experience is "idle time," time that's not your Oracle Database's fault, and so therefore it's not your fault either.

...It's not going to work, is it? Your job doesn't end with absolving your database or yourself; it ends with finding the root cause of a real business problem and then either fixing it or explaining why it makes more economic sense not to.

So, which parts of a profile are safe to ignore? If your profile matches the end user's experience, then the answer is simple: you can ignore *only* the events whose contributions to the experience duration were inconsequential. In other words, you can't ignore *any* events that contribute significant time to the profile, no matter what they're called.

If a duration dominates your user's performance experience, then you have to pay attention to it, no matter what it's called.

The Method R Profiler helps you prioritize your attention in three ways:

- Sorting — The Profiler sorts response time contributions in descending order of duration. You need to pay the most attention to the top lines of a profile.
- Color — The Profiler uses color to attract your attention to the data you should pay attention to first.
- Elision — The Profiler aggregates inconsequential response time contributors to save your mind the bother of consciously ignoring long lists of information you shouldn't be looking at.

Analyzing the Profile Report

My adjusted trace file does describe the user's experience, so it's finally time for the fun part: figuring out how I can help make my user's program go faster. To do that, I'll use both the Method R Profiler and the Method R Tools suite. The Profiler, as you've already seen, produces a fixed-format HTML report that uses color to attract attention and hyperlinks to help me navigate across different levels of detail. The MR Tools suite gives me flexibility beyond what the Profiler offers, and it produces copy-and-paste friendly plain text output, which is easy to use in reports like the one you're reading right now.

I'll begin, as I did before, with the Profiler. The first section, which I've already shown you, is the profile by subroutine:

Profile by subroutine for the user's 32-second experience.

	Subroutine	Duration seconds	Duration % R	Cumulative duration seconds	Cumulative duration % R	Call count	Duration per call (seconds) mean	min	skew	max	Drill-down
1.	**SQL*Net message from client**	**22.181**	**69.8%**	22.181	69.8%	**75,002**	0.000296	0.000000		0.095116	SQL
2.	unaccounted-for between dbcalls	6.292	19.8%	28.473	89.6%	150,038	0.000042	-0.000037		0.326858	SQL
3.	unaccounted-for within dbcalls	2.584	8.1%	31.057	97.8%	75,101	0.000034	-0.011754		0.019578	SQL
4.	CPU service, FETCH calls	0.588	1.9%	31.645	99.6%	75,014	0.000008	0.000000		0.212014	
5.	CPU service, unreported call(s)	0.230	0.7%	31.874	100.4%	1	0.229538	0.229538		0.229538	
6.	SQL*Net message to client	0.090	0.3%	31.964	100.6%	75,002	0.000001	0.000000		0.002551	
7.	CPU service, EXEC calls	0.012	0.0%	31.976	100.7%	31	0.000387	0.000000		0.012000	
8.	db file scattered read [blocks ≤ 16]	0.006	0.0%	31.982	100.7%	72	0.000081	0.000024		0.000166	
9.	db file scattered read [16 < blocks ≤ 32]	0.002	0.0%	31.984	100.7%	9	0.000233	0.000156		0.000348	
10.	db file sequential read	0.001	0.0%	31.984	100.7%	13	0.000040	0.000020		0.000099	
11.	log file sync	0.000	0.0%	31.985	100.7%	1	0.000165	0.000165		0.000165	
12.	Disk file operations I/O	0.000	0.0%	31.985	100.7%	1	0.000128	0.000128		0.000128	
13.	CPU service, PARSE calls	0.000	0.0%	31.985	100.7%	25	0.000000	0.000000		0.000000	
14.	CPU service, CLOSE calls	-0.224	-0.7%	31.761	100.0%	32	-0.007000	-0.224014		0.000000	
15.	Total	31.761	100.0%								

This table shows that 69.8% of the user's 31.761-second experience was consumed by **SQL*Net message from client** time. You know now that this is time that the Oracle kernel process spends blocked on **read** syscalls, awaiting input from its client program. I can't ignore these calls, because they were an important part of my end user's experience.

Now it's time to figure out what exactly this client program is trying to do. I can learn more by looking at the profile by SQL statement section.

Profile by SQL statement for the user's 32-second experience.

	Statement	Duration of self seconds	Duration of self % R	Cumulative duration seconds	Cumulative duration % R	Distinct cursors	Distinct texts	Statement size Lines	Statement size Bytes	Drill-down
1.	**[0f44xtf7g083u] select * from video_1**	**31.505**	**99.2%**	31.505	99.2%	1	1	2	22	stats
2.	[(no id for statement)] <not accounted for within other statements>	0.230	0.7%	31.735	99.9%	1				
3.	[57w71dgk5qbtx] SELECT DISTINCT SUBSTR(KGLNAOBJ,...	0.211	0.7%	31.946	100.6%	1	1	8	185	
4.	[dj8vc373zqyhp] SELECT /* OPT_DYN_SAMP */ /*+ AL...	0.024	0.1%	31.969	100.7%	1	1	4	450	stats
5.	18 other statements	-0.209	-0.7%	31.761	100.0%	20				
6.	Total	31.761	100.0%			24	18			

This table shows that just one SQL statement has consumed 99.2% of the total response time. Clearly this is the statement to target for the next phase of my analysis. The next section in the Profiler output is a profile showing the relationships among Oracle cursors. It shows me the contexts in which this statement appears.

Revised 2012-12-20

Profile by Cursor for the user's 32-second experience.

	Cursor statement	Duration of self seconds	Duration of self % R	Duration incl descendants seconds	Duration incl descendants % R	Drill-down
1.	**[0f44xtf7g083u] select * from video_1**	**31.505**	**99.2%**	31.530	99.3%	stats
2.	[dj8vc373zqyhp] SELECT /* OPT_DYN_SAMP */ /*+ AL...	0.024	0.1%	0.024	0.1%	stats
3.	[96g93hntrzjtr] select /*+ rule */ bucket_cnt, r...	0.001	0.0%	0.001	0.0%	stats
4.	[96g93hntrzjtr] select /*+ rule */ bucket_cnt, r...	0.001	0.0%	0.001	0.0%	stats
5.	[(no id for statement)] <not accounted for within other statements>	0.230	0.7%	0.230	0.7%	
6.	[(no id for statement)] <(no text for statement)>	0.001	0.0%	0.001	0.0%	stats
7.	[f711myt0q6cma] insert into sys.aud$(sessionid,...	0.000	0.0%	0.000	0.0%	stats
8.	[(no id for statement)] <(no text for statement)>	-0.228	-0.7%	0.000	0.0%	stats
9.	[21m25rd465y7b] BEGIN IF USER NOT IN ('SYS', 'SY...	0.001	0.0%	0.226	0.7%	stats
10.	[fb1zhb51087uh] begin ilo_timer.flush_ilo_runs; ...	0.009	0.0%	0.224	0.7%	stats
11.	[57w71dgk5qbtx] SELECT DISTINCT SUBSTR(KGLNAOBJ,...	0.211	0.7%	0.211	0.7%	
12.	[5mkm5m920sxrz] DELETE FROM DBMS_ALERT_INFO WHER...	0.001	0.0%	0.001	0.0%	stats
13.	[8ggw94h7mvxd7] COMMIT	0.001	0.0%	0.001	0.0%	
14.	[gwbmd8dtgz8r1] DELETE DBMS_ALERT_INFO WHERE SID...	0.000	0.0%	0.000	0.0%	
15.	7 other cursors	0.001	0.0%	0.001	0.0%	
16.	[1v717nvrhgbn9] SELECT USER FROM SYS.DUAL	0.001	0.0%	0.001	0.0%	
17.	[f711myt0q6cma] insert into sys.aud$(sessionid,...	0.002	0.0%	0.002	0.0%	stats
18.	[(no id for statement)] <synthetic commit (preceding statement has id="21m25rd465y7b")>	0.000	0.0%	0.000	0.0%	
19.	Total	31.761	100.0%			

In this case, the statement appears in only one context: as a parent with child and grandchild cursors listed on lines 2 through 4. Sometimes the same statement will show up as a child of several different cursors. Statements like « **commit** » or « **select sysdate from dual** » commonly work this way. My profile by cursor table proves that the dominant statement's children have durations that are inconsequential, amounting to only 0.1% of my user's total response time. The sole focus of my attention thus needs to be this « **select * from video_1** » statement.

I can drill into its details by clicking on the "stats" link in the rightmost column of row 1. The click takes me to the section of the Profiler output where I can see the profile for the statement's 31.505-second contribution to my user's experience.

Profile by Subroutine for the « select * from video_1 » statement that accounted for 99.2% of the response time.

	Subroutine	Duration seconds	Duration %	Duration % R	Cumulative duration seconds	Cumulative duration %	Cumulative duration % R	Call count	Duration per call (seconds) mean	min	skew	max
1.	**SQL*Net message from client**	**22.181**	**70.4%**	**69.8%**	22.181	70.4%	69.8%	**75,001**	0.000296	0.000000		0.095116
2.	unaccounted-for between dbcalls	6.291	20.0%	19.8%	28.472	90.4%	89.6%	150,035	0.000042	-0.000037		0.326858
3.	unaccounted-for within dbcalls	2.583	8.2%	8.1%	31.054	98.6%	97.8%	75,004	0.000034	-0.011754		0.019578
4.	CPU service, FETCH calls	0.356	1.1%	1.1%	31.410	99.7%	98.9%	75,001	0.000005	0.000000		0.020002
5.	SQL*Net message to client	0.090	0.3%	0.3%	31.500	100.0%	99.2%	75,001	0.000001	0.000000		0.002551
6.	db file scattered read [blocks ≤ 16]	0.003	0.0%	0.0%	31.503	100.0%	99.2%	41	0.000066	0.000024		0.000166
7.	db file scattered read [16 < blocks ≤ 32]	0.002	0.0%	0.0%	31.505	100.0%	99.2%	9	0.000233	0.000156		0.000348
8.	db file sequential read	0.000	0.0%	0.0%	31.505	100.0%	99.2%	4	0.000027	0.000020		0.000037
9.	CPU service, EXEC calls	0.000	0.0%	0.0%	31.505	100.0%	99.2%	1	0.000000	0.000000		0.000000
10.	CPU service, PARSE calls	0.000	0.0%	0.0%	31.505	100.0%	99.2%	1	0.000000	0.000000		0.000000
11.	CPU service, CLOSE calls	0.000	0.0%	0.0%	31.505	100.0%	99.2%	1	0.000000	0.000000		0.000000
12.	Total	31.505	100.0%	99.2%								

This profile looks a lot like the profile for the entire experience, because the statement accounts for almost all (99.2%) of the total experience. So it comes as no surprise that the statement's execution duration is dominated by **SQL*Net message from client** call durations as well. There are 75,001 **SQL*Net message from client** calls attributable to this statement. Why so many? The statement's profile by database call shows why.

Profile by Database Call for the « select * from video_1 » statement that accounted for 99.2% of the response time.

	Database call	Duration seconds	Duration %	Duration % R	Contribution (seconds) CPU	Contribution (seconds) Other	Call count	Rows processed	Database buffer cache accesses (LIO blocks) Total	CR mode	CU mode	Oracle blocks read from OS (PIO blocks)	Library cache misses
1.	**between calls**	**28.562**	**90.7%**	**89.9%**	0.000	28.562	0	0	0	0	0	0	0
2.	FETCH	2.937	9.3%	9.2%	0.356	2.581	**75,001**	150,000	75,391	75,391	0	483	0
3.	PARSE	0.007	0.0%	0.0%	0.000	0.007	1	0	1	1	0	0	1
4.	CLOSE	0.000	0.0%	0.0%	0.000	0.000	1	0	0	0	0	0	0
5.	EXEC	0.000	0.0%	0.0%	0.000	0.000	1	0	0	0	0	0	0
6.	Total	31.505	100.0%	99.2%	0.356	31.149	75,004	150,000	75,392	75,392	0	483	1
7.	Total per execution	31.505	100.0%	99.2%	0.356	31.149	75,004	150,000	75,392	75,392	0	483	1
8.	Total per row	0.000	100.0%	0.0%	0.000	0.000	1	1	**1**	1	0	0	0

The profile by database call for the statement explains exactly the same 31.505 seconds that the statement's profile by subroutine call for the statement did, but from a different perspective—from the database call dimension. The statement's duration was dominated (90.7%) by time consumed *between* database calls. The red 75,001 **fetch** call count (row 2) attracts my attention. This program executed 75,001 database **fetch** calls to retrieve 150,000 rows.

It looks like the program is fetching just 150,000 / 75,001 ≈ 2 rows at a time, but from the Profiler output alone, I can't prove it. The application *could* have fetched, for example, 42 rows per call for 3,571 calls, 18 rows on one call, and 0 rows on 71,429 calls. On my MR Tools console, I can run a quick "Rows returned by dbcall" report that shows exactly what I want to know without having to look through the raw trace data myself:

```
Rows returned by dbcall
mrskew --name=dbcall --select=$row --slabel=ROWS --precision=0 "…/slow-mrcallrm.trc"
2012-11-28T17:31:25.000431-0600
Elapsed: 6.440000 s

CALL-NAME     ROWS       %   CALLS  MEAN  MIN  MAX
---------  -------  ------  ------  ----  ---  ---
FETCH      150,000  100.0%  75,001     2    1    2
PARSE            0    0.0%       1     0    0    0
XCTEND           0    0.0%       9     0    0    0
CLOSE            0    0.0%       3     0    0    0
EXEC             0    0.0%       1     0    0    0
---------  -------  ------  ------  ----  ---  ---
TOTAL (5)  150,000  100.0%  75,015     2    0    2
```

This table proves that the program fetched 150,000 rows, two at a time:

- The mean number of rows returned by each **fetch** call was 2.
- The minimum number of rows returned by a **fetch** call was 1.
- The maximum number of rows returned by a **fetch** call was 2.

So, this program fetched 150,000 rows, two at a time. No wonder it's spending so much time between database calls: the application executes a network roundtrip every time it makes a database call, and with 75,001 **fetch** calls, that's a lot of roundtrips. Even though the average call latency is small (0.000296 seconds per **SQL*Net message from client** call), there are so many calls that their total duration adds up to over 28.5 seconds.

How much time could I save if I could reduce the number of roundtrips?

Predicting the Improvement

If I can retrieve more than two rows per fetch call, I should be able to eliminate some network roundtrips. Eliminating some roundtrips should reduce the duration of my user's response time experience, but how much time savings should I expect? Your boss (your consulting client, etc.) will want to know that before he decides to let you have a go at the solution.

To create such an estimate, the first report I run is the MR Tools "Response time histogram for a given call name pattern" report. This will show me any skew in my call latencies that I might need to know about. Here is the report that I ran to see skew in **SQL*Net message from client** latencies. I modified the report to show only those calls associated with the SQL id of the « **select * from video_1** » query that I learned from the Profiler accounted for 99.2% of the user's response time experience:

```
Response time histogram for given call name pattern (modified)
mrskew --rc=p10 --name=SQL\*Net message from client --where=$sqlid eq "0f44xtf7g083u" "…/slow-mrcallrm.trc"
2012-11-28T17:52:11.000289-0600
Elapsed: 6.440000 s

        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  20.218516   91.2%  74,756  0.000270  0.000196  0.000994
 5.     0.001000     0.010000   0.519500    2.3%     207  0.002510  0.001002  0.009754
 6.     0.010000     0.100000   1.443090    6.5%      37  0.039002  0.010216  0.095116
 7.     0.100000     1.000000
 8.     1.000000    10.000000
 9.    10.000000   100.000000
10.   100.000000 1,000.000000
11. 1,000.000000           +∞
----------------------------  ---------  ------  ------  --------  --------  --------
                   TOTAL (11)  22.181106  100.0%  75,001  0.000296  0.000000  0.095116
```

I already knew there were 75,001 calls averaging 296 microseconds (µs) apiece, but here I can see more detail. If I could eliminate 37 especially slow calls, I'd eliminate 1.443 seconds from my user's experience, but the interesting bucket is the 20 seconds spent on 74,756 calls lasting between 100 µs and 1 ms each. The overwhelming majority of my user's **SQL*Net message from client** time was spent on calls lasting less than 1 ms apiece. This looks like a demand (call count) problem not a supply (latency) problem.

So, how can I reduce **SQL*Net message from client** call demand? By changing the application. If I can cause each **fetch** call to retrieve, say, 100 rows per call, the **fetch** call count for this statement should drop from 150,000 / 2 ≈ 75,001 to 150,000 / 100 ≈ 1,501. This, in turn, should eliminate 75,001 − 1,501 = 73,500 **SQL*Net message from client** calls from my user's experience. If everything else including the roundtrip latencies remains constant, then I would expect response time to improve as shown in the following table.

Predicting the experience duration change that will accompany the call count change.

	Baseline		Predicted	
Subroutine	Call count	Duration (seconds)	Call count	Duration (seconds)
SQL*Net message from client (0.000296 seconds each)	75,002	22.181	1,502	0.445
Other		9.58		9.58
Total		31.761		10.025

That is, if I can find a way to increase the array fetch size from 2 to 100, then I think I can make my user's 32-second experience run in just 10 seconds. I believe it's worth a try.

The Fix

My user ran his program—just one statement—in **sqlplus**, so the repair was easy. Here's what it looked like before:

```
set pagesize 2  timing on  termout off
select * from video_1;
```

And here's what it looked like after:

```
set pagesize 2  timing on  termout off  arraysize 100
select * from video_1;
```

What trips people up sometimes is that the only way to fix this problem is to change the source code for the program itself. There's no database setting that fixes this problem.

What you need to look up in your programming documentation to set your Oracle array fetch size.

Language/tool	Keywords pertaining to Oracle array fetch size manipulation
sqlplus	**arraysize**
Java	**setFetchSize**
PHP	**oci_set_prefetch**
Perl	**RowCacheSize**
Python	**arraysize**
C	**OCIStmtFetch nrows**
C++	**setPrefetchRowCount**
Visual Basic	**FetchSize**
C#	**Fetchsize**
Ruby	**OCI_ATTR_PREFETCH_ROWS**
Oracle SQL Developer	"Sql Array Fetch Size"

My next step will be to run the improved program and measure its duration, of course, by tracing it. This time, I'll properly scope the trace file, so I won't have to spend time with another **mrcallrm** step. The key is to activate tracing *immediately* before running the statement I'm measuring, and then deactivate tracing *immediately* when the statement completes. Here's the whole thing in **sqlplus**:[1]

```
set pagesize 2  timing on  termout off  arraysize 100
exec dbms_monitor.session_trace_enable(null, null, true, true)
select * from video_1;
exit
```

I could also have created a perfectly scoped trace file by executing the script within Oracle SQL Developer, with the Method R Trace extension installed and activated. Either way, I will end up with a perfectly scoped trace file, but by using MR Trace, I'd get the added benefit of the tool automatically copying the file to my laptop. Even when I trace programs from outside SQL Developer, I use MR Trace to fetch my Oracle trace files.

[1] You can simulate this case yourself by defining a view called **video_1** defined as « **select * from dba_source where rownum <= 150000** ».

Results of the Fix

The results of the fix are spectacular:

***Before*: profile by subroutine for the user's 32-second experience with array fetch size set to 2.**

	Subroutine	Duration seconds	Duration % R	Cumulative duration seconds	Cumulative duration % R	Call count	Duration per call (seconds) mean	min	skew	max	Drill-down
1.	**SQL*Net message from client**	**22.181**	**69.8%**	22.181	69.8%	**75,002**	0.000296	0.000000		0.095116	SQL
2.	unaccounted-for between dbcalls	6.292	19.8%	28.473	89.6%	150,038	0.000042	-0.000037		0.326858	SQL
3.	unaccounted-for within dbcalls	2.584	8.1%	31.057	97.8%	75,101	0.000034	-0.011754		0.019578	SQL
4.	CPU service, FETCH calls	0.588	1.9%	31.645	99.6%	75,014	0.000008	0.000000		0.212014	
5.	CPU service, unreported call(s)	0.230	0.7%	31.874	100.4%	1	0.229538	0.229538		0.229538	
6.	SQL*Net message to client	0.090	0.3%	31.964	100.6%	75,002	0.000001	0.000000		0.002551	
7.	CPU service, EXEC calls	0.012	0.0%	31.976	100.7%	31	0.000387	0.000000		0.012000	
8.	db file scattered read [blocks ≤ 16]	0.006	0.0%	31.982	100.7%	72	0.000081	0.000024		0.000166	
9.	db file scattered read [16 < blocks ≤ 32]	0.002	0.0%	31.984	100.7%	9	0.000233	0.000156		0.000348	
10.	db file sequential read	0.001	0.0%	31.984	100.7%	13	0.000040	0.000020		0.000099	
11.	log file sync	0.000	0.0%	31.985	100.7%	1	0.000165	0.000165		0.000165	
12.	Disk file operations I/O	0.000	0.0%	31.985	100.7%	1	0.000128	0.000128		0.000128	
13.	CPU service, PARSE calls	0.000	0.0%	31.985	100.7%	25	0.000000	0.000000		0.000000	
14.	CPU service, CLOSE calls	-0.224	-0.7%	31.761	100.0%	32	-0.007000	-0.224014		0.000000	
15.	Total	31.761	100.0%								

***After*: profile by subroutine for the user's program after changing the array fetch size to 100.**

	Subroutine	Duration seconds	Duration % R	Cumulative duration seconds	Cumulative duration % R	Call count	Duration per call (seconds) mean	min	skew	max	Drill-down
1.	**SQL*Net message from client**	**1.268**	**67.3%**	1.268	67.3%	**1,502**	0.000844	0.000646		0.016222	SQL
2.	CPU service, FETCH calls	0.276	14.7%	1.544	82.0%	1,508	0.000183	0.000000		0.228014	SQL
3.	CPU service, unreported call(s)	0.263	14.0%	1.807	95.9%	1	0.263422	0.263422		0.263422	SQL
4.	unaccounted-for between dbcalls	0.166	8.8%	1.973	104.7%	3,010	0.000055	-0.000013		0.001170	SQL
5.	unaccounted-for within dbcalls	0.136	7.2%	2.109	112.0%	1,581	0.000086	-0.008312		0.010180	SQL
6.	CPU service, EXEC calls	0.028	1.5%	2.137	113.4%	25	0.001120	0.000000		0.024001	
7.	SQL*Net message to client	0.002	0.1%	2.139	113.6%	1,502	0.000002	0.000000		0.000038	
8.	log file sync	0.000	0.0%	2.139	113.6%	1	0.000389	0.000389		0.000389	
9.	Disk file operations I/O	0.000	0.0%	2.140	113.6%	1	0.000121	0.000121		0.000121	
10.	SQL*Net more data to client	0.000	0.0%	2.140	113.6%	1	0.000014	0.000014		0.000014	
11.	CPU service, PARSE calls	0.000	0.0%	2.140	113.6%	23	0.000000	0.000000		0.000000	
12.	CPU service, CLOSE calls	-0.256	-13.6%	1.884	100.0%	26	-0.009847	-0.256016		0.000000	SQL
13.	Total	1.884	100.0%								

I had hoped that the duration would fall from 31 seconds to 10, but this fix performed even better than that: duration fell to just 1.884 seconds. That's an elimination of 94% of the original time my user spent waiting on his result.

These things can happen on a good day.

Why was the improvement even better than I had predicted? Let's look at the profiles line-by-line:

- The first line in the original profile, **SQL*Net message from client** duration, dropped from 22.181 seconds to 1.268 seconds, a savings of about 21 seconds. I had figured that the total response time of **SQL*Net message from client** calls would drop from 22.181 seconds to 0.445 seconds, so I missed by about 0.8 seconds. The per-call latency did not remain constant; it rose from 296 µs to 844 µs, but my estimate was materially accurate because the call count behaved exactly as I had predicted, from 75,002 calls to 1,502 calls.
- The second row in the original profile, **unaccounted-for between dbcalls**, dropped from 6.292 seconds to just 0.166 seconds, a savings of over 6 seconds. The average latency stayed in the tens-of-microseconds range, but the call count fell from 150,038

to 3,010 because of the reduction in the **fetch** and **SQL*Net message from client** call counts.

- The third row in the original profile, **unaccounted-for within dbcalls**, dropped from 2.584 seconds to 0.136 seconds, a savings of another 2.4 seconds. Again, the average latency stayed in the tens-of-microseconds range, and the call count fell from 75,101 to 1,581 because of the **fetch** call count reduction.

These three profile improvements describe the difference between a program that makes its user wait so long that he loses focus, and a program that takes only a couple of seconds to finish.

The Profiler shows what the fix did at the statement level.

Profile by database call for original « select * from video_1 » statement, using an array fetch size of 100.

	Database call	Duration seconds	Duration %	Duration % R	Contribution (seconds) CPU	Contribution (seconds) Other	Call count	Rows processed	Database buffer cache accesses (LIO blocks) Total	CR mode	CU mode	Oracle blocks read from OS (PIO blocks)	Library cache misses
1.	**between calls**	**1.434**	**88.6%**	**76.1%**	0.000	1.434	0	0	0	0	0	0	0
2.	FETCH	0.184	11.3%	9.7%	0.048	0.136	1,501	150,000	2,239	2,239	0	0	0
3.	PARSE	0.000	0.0%	0.0%	0.000	0.000	1	0	0	0	0	0	0
4.	EXEC	0.000	0.0%	0.0%	0.000	0.000	1	0	0	0	0	0	0
5.	CLOSE	0.000	0.0%	0.0%	0.000	0.000	1	0	0	0	0	0	0
6.	Total	1.618	100.0%	85.9%	0.048	1.570	1,504	150,000	2,239	2,239	0	0	0
7.	Total per execution	1.618	100.0%	85.9%	0.048	1.570	1,504	150,000	2,239	2,239	0	0	0
8.	Total per row	0.000	100.0%	0.0%	0.000	0.000	0	1	**0**	0	0	0	0

This is what I had expected: only 1,501 **fetch** calls to retrieve all 150,000 rows. Finally, MR Tools confirms that the fix did indeed change the array fetch size as I had intended:

```
Rows returned by dbcall
mrskew --name=dbcall --select=$row --slabel=ROWS --precision=0 "…/fast.trc"
2012-11-29T16:36:42.000982-0600
Elapsed: 1.503000 s

CALL-NAME     ROWS       %  CALLS  MEAN  MIN  MAX
---------  -------  ------  -----  ----  ---  ---
FETCH      150,000  100.0%  1,501   100    1  100
PARSE            0    0.0%      1     0    0    0
XCTEND           0    0.0%      9     0    0    0
CLOSE            0    0.0%      3     0    0    0
EXEC             0    0.0%      1     0    0    0
---------  -------  ------  -----  ----  ---  ---
TOTAL (5)  150,000  100.0%  1,515    99    0  100
```

...And so the story ends.

Final Thoughts

Now that I've focused your attention for so many pages on the plight of a poor array fetch size, you might feel compelled to go check your application for array fetch size problems. I hope you won't…. Why not? Because I could have chosen any of thousands of problems that you might have—for instance, involving bad user interface design, or an impossibly unrealistic data design, or poorly written SQL, or badly configured hardware. Should you have to check for all those problems right now, too? Is checking and verifying *everything* that could possibly be wrong with a system the only way to confirm its efficiency?

That's no way to live.

I've been a performance analyst for Oracle-based systems since 1989. The most important thing I've learned about this job is the importance of connecting my analysis

with the end-user experience. One of the most frustrating scenarios for a business is when a user feels performance pain, but the performance monitoring dashboards show that everything is a-okay. I need to see an objective measurement of exactly where my user's time is going. Oracle's extended SQL trace files give me the measurements I need, but I also need excellent tools to extract the information that those files have to tell me.

My company, Method R Corporation, creates tools that help you connect directly with your end-users' performance experiences. Method R Trace is our extension for Oracle SQL Developer that collects and retrieves perfectly scoped trace files with no extra clicks. For trace files that come to you without perfect scoping, two programs in the Method R Tools suite—**mrskew** and **mrcallrm**—make it easy to identify and fix scoping errors. When your trace file maps to the experience you're analyzing, The Method R Profiler shows you exactly where your user's time has gone. It uses color to attract your attention to the right data, hyperlinks to guide your navigation, and elision to keep you from wasting your time on things that don't matter. When you require data mining beyond what the Profiler provides, **mrskew** gives you remarkably broad capabilities.

Profiling began in computer science as a procedure that developers executed upon their code to reveal where their code path was spending its time. Profiling helps you prevent bad code from reaching your production system. It helps you fix problems in production systems. Profiling as a habit throughout the software development life cycle serves as an excellent early detection system for performance problems. The magic of profiling is that it shows you how your code spends your users' time, no matter where that is. You don't have to know in advance what problems to go looking for; the profile takes you right to where the time is being consumed.

Profiling offers another tremendous advantage over traditional dashboard applications. Because the profile so closely connects you to the user's experience, it's much easier to predict—directly in hours, minutes, and seconds—how much time you'll be able to save by working on a given program. This allows you to know in advance how much benefit you can expect for a proposed investment into a remedy activity. If you're a consultant, it's how you can estimate the value of a benefit that you'll be proposing for your client to pay for. It's also how you'll know when you've reached the limit on tuning a program. When a program is efficient already, you need to know that so you won't waste time trying to "tune" it. The profile can show you that.

Further Reading

Millsap, Cary. 2012. "The Method R Profiling Ecosystem" at http://method-r.com/blog/191-the-method-r-profiling-ecosystem

A blog post describing how the software tools in the Method R Workbench software package fit together to help programmers write faster, more efficient programs.

Millsap, Cary. 2011. "Mastering Oracle Trace Data" at http://method-r.com/courses/trace-data-masterclass

A 1-day course taught by Cary Millsap covering Oracle extended SQL trace files, and Method R Trace and Method R Tools software in detail.

Millsap, Cary. 2011. "Mastering performance with Oracle extended SQL trace" at http://method-r.com/downloads/doc_view/72-mastering-performance-with-extended-sql-trace?tmpl=component&format=raw

This paper explains details about collecting and interpreting Oracle extended SQL trace files.

Millsap, Cary. 2010. "Thinking clearly about performance" at http://method-r.com/downloads/doc_view/44-thinking-clearly-about-performance?tmpl=component&format=raw

This paper explains fundamental principles that make performance problem solving and prevention simpler and more reliable.

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

This book describes a reliable, repeatable, and deterministic method for isolating Oracle system performance problems. It focuses on the one statistic that truly matters: response time as seen by the users of a system.

About Method R Corporation

Method R Corporation was founded in 2008 by Oracle performance specialist Cary Millsap. We help companies of all sizes get the best possible value out of their software application systems. We sell industrial strength software such as the products featured in this paper, education courses for developers and database administrators, and a wide range of consulting services. We are the creators of the method called "Method R," as originally documented in the book *Optimizing Oracle Performance*, for which our own Cary Millsap and Jeff Holt were named *Oracle Magazine* Authors of the Year.

Method R Corporation
Southlake, Texas
United States of America
http://method-r.com
info@method-r.com
@MethodR