

Performance Problems from the Field

OAUG Spring 2002 Toronto 22 May 2002

Cary Millsap

Manager and Co-Founder

Hotsos Enterprises, Ltd.

Agenda

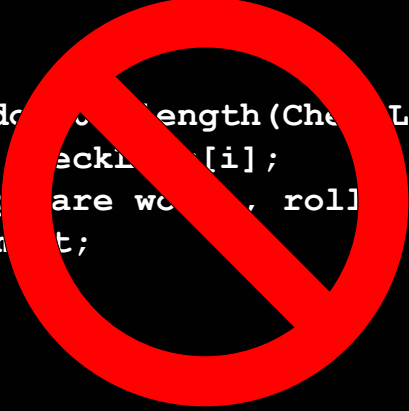
- How we collect field data
- How we analyze field data
- Three typical performance problems

Good method?

```
A: i = random(0, length(CheckList)-1);  
   execute CheckList[i];  
   if things are worse, roll back;  
   else commit;  
   goto A;
```

Good method?

```
A: i = random.randrange(ChessList)-1;  
execute ChessList[i];  
if things are wrong, roll back;  
else commit;  
goto A;
```



The Hotsos method

- Choose $1 \leq n \leq 5$ programs
 - ...that will optimize the **business**
- Gather level-8 trace data for each program
 - ...during a **problem interval**
- Analyze the data
 - ...to determine the **optimal remedy**
- Implement the remedy

Choosing programs

•Choose the 1-5 programs whose performance improvement will most benefit the **business**. DO NOT let system-wide statistics set your optimization priorities. *Example:* Querying v\$system_event shows that 'latch free' is your system's bottleneck, but your business says that the PYUGEN program is your problem. Then focus on PYUGEN.

Gathering trace data

- Collect data **only** for the problem program, and **only** for the problem time interval. Collecting more will allow irrelevant data to bury your relevant data. Collecting less prohibits you from seeing everything you need to see. See <http://www.hotsos.com/downloads/1/10046a> for complete details about **how** to activate 10046 tracing.
- The free Hotsos data collector helps with data collection (<http://www.hotsos.com/products/sparky>).

Analyzing the data

- By analyzing the data for all 1-5 programs, you can determine which problem/solution pair is the most attractive to the business. Without this preliminary analysis, you could not optimize your optimization investment, because you couldn't have known the **costs** and potential **benefits** of the options.
- We use the **Hotsos Profiler**, a product that we sell, to analyze 10046 trace data. Hotsos Profiler minimizes the time we require to analyze 10046 trace data. However, many people have written simpler 10046 parsers to assist in the analysis. Our **Hotsos Clinic** training event describes how (<http://www.hotsos.com/training/clinic>).

Implementing the remedy

- Various barriers slow down this step, not the least of which is change control. Another common barrier is the failure to convince others to take the action you recommend. If you justify your recommendation by saying it will improve a hit ratio, who could blame them. But if you can justify your recommendation with a quantified, tested response time improvement, it increases your chances of success.

Focus of the method

- Response time
 - NOT hit ratios!
- Program statistics
 - NOT system-wide statistics
- Problem time interval
 - NOT fixed-duration polling intervals

Response time

- We focus on response time, because good performance is, by definition, optimally small response time. Response time is a direct, unambiguous measure of user satisfaction.
- The popular alternative is to optimize **hit ratios**. The problem with hit ratios is that they do **not** correlate to performance. In the old days, hit ratios were all we had. But since 1992, with the release of Oracle 7.0.12, we've been able to measure Oracle response times and no longer need to rely on ambiguous and often misleading hit ratios. Most unfortunately, hit ratio tuning is still taught today.

Program

- Optimizing a session optimizes the business because of the way we have chosen the sessions under analysis. If we have done our job in step 1 of the Hotsos Method, then we know that reducing response time for the session we're analyzing is the efficient thing to do for the business. Even if the program is slow because of some other program's inefficiency, we'll reliably see the root cause of the problem in the 10046 data.
- System-wide data don't help; they introduce ambiguity into the optimization process.

Problem time interval

- A common mistake is to collect data for a duration that is not *exactly* the problem interval.
- Data for too large of a time interval is a problem; relevant data get buried beneath irrelevant data.
- Data for too small of a time interval is a problem; you can't see what you don't look at.

Why we use 10046 trace data

- We use it almost exclusively
 - High-impact, time efficient, measurable, predictive, reliable, deterministic, finite, practical
- Reliably identifies problems caused by...
 - Traced session itself
 - Interaction with other sessions
 - Slow hardware
 - Excessive context switching, paging, swapping, ...

Oracle 10046 trace data provides the means for our Hotsos Method to meet each of these optimization method objectives:

- High impact—no more “working hard on the wrong thing, expecting improvement from it, then ending up with nothing” [Pelz 2000]
- Time efficiency—method must make normal DBAs as fast as the best analysts in my old Oracle System Performance Group
- Measurability—must measure exactly what the user sees, not some derivative measure like hit ratios
- Predictive capacity—must be able to predict the exact outcome of a suggestion without having to try it first
- Reliability—must identify performance problem causes no matter what the problem is
- Determinism—two competent analyst looking at the same data will draw the same conclusions
- Finiteness—must be easy to determine when cost of improvement exceeds cost of status quo
- Practicality—any tools must be inexpensive and usable across Unix, Linux, VMS, Windows, etc.

The ability to detect all the problem types listed on this slide with 10046 data is the result of new, original Hotsos research. Authors still teach, for example, that you cannot detect CPU saturation with 10046 data. These authors are incorrect. The technical details of our research are explained in detail in our Hotsos Clinic (<http://www.hotsos.com/training/clinic>).

Case 1

Oracle Payroll still slow after CPU upgrade

Subjective examination

- Oracle Payroll processing has been slow for several months
- Quest's Spotlight on Oracle™ shows plainly that system bottleneck is `latch free` for cache buffers chains latches
- Client's app developers have determined that Payroll program SQL is efficient
- Recent upgrade of 12 CPUs from 700MHz to 1GHz has made problem even worse

Objective examination

Oracle Kernel Event	Duration	Calls	Avg	
SQL*Net message from client	984.01s	49.6%	95,161	0.010340s
SQL*Net more data from client	418.82s	21.1%	3,345	0.125208s
db file sequential read	279.54s	14.1%	45,085	0.006200s
CPU service	248.69s	12.5%	222,760	0.001116s
unaccounted-for	27.73s	1.4%		
latch free	23.69s	1.2%	34,695	0.000683s
log file sync	1.09s	0.1%	506	0.002154s
SQL*Net more data to client	0.83s	0.0%	15,982	0.000052s
log file switch completion	0.28s	0.0%	3	0.093333s
enqueue	0.25s	0.0%	106	0.002358s
SQL*Net message to client	0.24s	0.0%	95,161	0.000003s
buffer busy waits	0.22s	0.0%	67	0.003284s
db file scattered read	0.01s	0.0%	2	0.005000s
SQL*Net break/reset to client	0.00s	0.0%	2	0.000000s
Total	1,985.40s	100.0%		

Notice that 'latch free' is a microscopically small component of this program's response time. It doesn't matter that the whole system's "biggest bottleneck" is 'latch free'; **this** program's bottleneck has nothing to do with latches.

In fact, 70.7% of this program's response time is consumed by SQL*Net operations.

- The 'SQL*Net message from client' is a tally of the time that the Oracle kernel spends blocked on a socket read from a client program to which the kernel has sent a response.
- The 'SQL*Net more data from client' is a tally of the time that Oracle waits for a request of (as the name implies) more data from a client. For example, PYUGEN has lots of SQL statements in the client program that are so big that they won't fit into one SQL*Net packet. Therefore, a single parse() call from the client to the server requires multiple SQL*Net transmissions. After the kernel receives the first piece of the SQL text, it will tally time to 'SQL*Net more data from client' as it awaits subsequent pieces.

As always, there are two ways to attack a performance problem: reduce the **number of calls**, or reduce the average **latency per call**.

- In this case, reducing the number of calls will require some application redesign. It would be a good idea (e.g., Oracle should use packed procedure calls instead of inflicting 3,345 'more data' calls upon the application), but it would be difficult to motivate the application vendor (Oracle in this case) to make the changes.
- The latencies (10ms+) look like LAN latencies. This is a batch job. If we could run this batch job on the database server host, we could use much-faster **IPC** instead of the LAN, which should reduce SQL*Net latencies by a factor of maybe 100 or more...

Objective examination

```
host name = dalunix150.xxxxxxxxx.com
instance name = prod
Oracle version = 8.1.6.3.0
session id = 611
program = PYUGEN@dalunix150.xxxxxxxxx.com (TNS V1-V2)
trace file = /prod/u001/app/./udump/ora_922341.trc
t0 = Wed Sep 12 2001 14:10:27 (388941433)
t1 = Wed Sep 12 2001 14:43:32 (389139973)
interval duration = 1,985.40s
total lines = 1,760,351
ignored lines = 16,093
```

Alas, this client program is already running on the database host! Note that the client program called **PYUGEN** is running on the same host as the db server. This means that the client and server can communicate via IPC instead of TCP/IP. Quick tests at the client site proved that IPC was about two orders of magnitude faster than TCP/IP on this system.

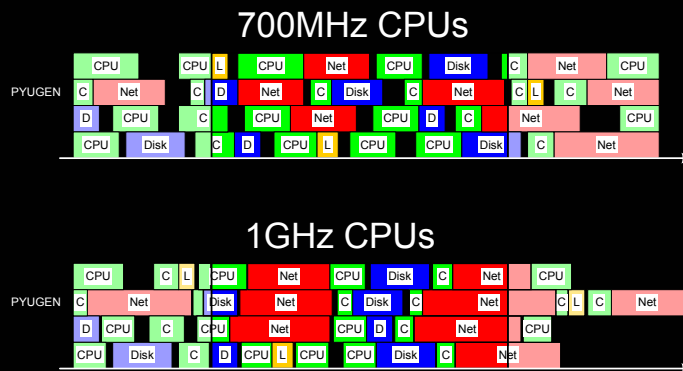
The reason that this client used TCP/IP as the default protocol is that the administrative staff wanted to share a single tnsnames.ora file across the entire system. Our recommendation would require the administration of **two** tnsnames.ora files instead of just one: a local one, and a remote one. But the payoff would be much smaller SQL*Net latencies for the system's batch jobs.

Assessment

- Client and server processes on the same host were communicating via the network card
- Setting `PROTOCOL=BEQ` in `tnsnames.ora` eliminated over 50% of process response time
- CPU upgrade caused performance degradation because now more processes could compete for the NIC sooner than before

Ironically, any tool that views `v$system_event` or `v$session_event` during fixed-duration polling intervals **must** ignore SQL*Net operations entirely. Any such tool would have failed to detect this type of performance problem.

Why the upgrade didn't work



Queueing theory predicts that, on a busy system, an upgrade to a component that is not **your** bottleneck can cause performance degradation for you. The picture here shows how. By upgrading to faster CPUs, all the CPU service durations get smaller. However, because before the upgrade PYUGEN spends only about 18% of its time consuming the CPU resource, the CPU upgrade does not provide much elapsed time benefit to PYUGEN.

However, there are lots of other programs on the system running concurrently with PYUGEN, and lots of these programs definitely benefit from the CPU upgrade. But by shortening the duration for each CPU service request, the other processes make their network I/O requests sooner, crunching more network I/O calls into the time interval during which PYUGEN is running. The effect is longer service requests for network I/O calls, which further intensifies the network bottleneck for PYUGEN.

Other processes on the system have benefited from the CPU upgrade, but PYUGEN actually consumes **more** elapsed time after the CPU upgrade. Since PYUGEN's performance was deemed important to the business (trying to improve its performance is what motivated the CPU upgrade in the first place!), the expensive upgrade actually produced not only **positive cost** but **negative benefit** for the business, a net double-whammy.

Lessons learned

- Don't let your ∇ \$ tables tell you what your "system problem is"
- Let your **business** define your problem programs
- Find out what's consuming **their response time**
- **Don't upgrade** until you've decomposed your program's response time

Case 2

Inefficient SQL kills performance

Subjective examination

- Oracle Purchasing batch job requires over 40 minutes of runtime
- Several jobs like this had motivated the “requirement” to use Oracle Parallel Server

Objective examination

Oracle Kernel Event	Duration	Calls	Avg	
CPU service	1,527.51s	60.8%	158,257	0.009652s
db file sequential read	432.03s	17.2%	62,495	0.006913s
unaccounted-for	209.56s	8.3%		
global cache lock s to x	99.87s	4.0%	3,434	0.029083s
global cache lock open s	85.93s	3.4%	3,507	0.024502s
global cache lock open x	57.88s	2.3%	1,930	0.029990s
latch free	26.77s	1.1%	1,010	0.026505s
SQL*Net message from client	19.11s	0.8%	6,714	0.002846s
write complete waits	11.13s	0.4%	155	0.071806s
row cache lock	11.10s	0.4%	485	0.022887s
enqueue	11.09s	0.4%	330	0.033606s
log file switch completion	7.31s	0.3%	15	0.487333s
log file sync	3.31s	0.1%	39	0.084872s
wait for DLM latch	2.95s	0.1%	91	0.032418s
...				
Total	2,510.50s	100.0%		

This resource profile is typical of programs that contain inefficient SQL. The most prominent component of response time is 'CPU service' (a Hotsos Profiler pseudoevent obtained from the "c=" fields of the raw trace data), which is consumed processing LIO calls, latch spins, sorts, hashes, language processing, and the like.

Seeing 'db file sequential read' or even 'db file scattered read' is also common for Oracle databases. At about 20% of the total execution time for this program, it's a distant second place in relation to the concern we should have about reducing the total 'CPU service' duration.

The 'unaccounted-for' event is another Hotsos Profiler pseudoevent that accounts principally for operating system context switch time and measurement error (<http://www.hotsos.com/dnloads/1/kevents/?name=unaccounted-for>).

Some events that are unique to Oracle Parallel Server (OPS) are also conspicuous here. OPS appears to inflict a penalty of about 240 seconds (10%) upon the program's total execution time ('global cache lock%' events). The minutes of overhead of course add up quickly when they're part of every single program that runs on the system.

Our next step is to determine what SQL statement or statements are consuming most of the program's response time...

Objective examination

Interval Time by Statement (118 distinct statements)

```
----- Exclusive -----  
Elapsed      Pct          CPU      SQL Statement Tree  
-----  
1,450.96s   57.8%    1,065.95s   704365403  
...        ...        ...        ...  
785.13s    31.3%    371.92s    3277176312  
...        ...        ...        ...
```

Hotsos Profiler makes it easy to determine which SQL statements account for which components of a program's total response time. In this case, it is very easy to see that one SQL statement (704365403) accounts for well over half of the slow program's response time. This is the statement that we should examine first.

Objective examination

```
statement id = 704365403
```

```
update po_requisitions_interface set requisition_header_id=:b0
where (req_number_segment1=:b1
and request_id=:b2)
```

Action	Count	Rows	----- Response Time -----			LIO Blks	PIO Blks
			Elapsed	CPU	Waits		
Parse	0	0	0.00	0.00	0.00	0	0
Execute	1,166	0	1,454.98	1,066.43	388.55	8,216,887	3,547
Fetch	0	0	0.00	0.00	0.00	0	0
Total	1,166	0	1,454.98	1,066.43	388.55	8,216,887	3,547
Per Exe	1	0	1.25	0.91	0.33	7,047	3
Per Row	1,166	1	1,454.98	1,066.43	388.55	8,216,887	3,547

Note that this program consumed over 8 million LIOs distributed across 1,166 executions of the statement, to determine that there are no rows that should be updated.

Without even consulting an execution plan, we can determine with reasonable certainty that creating an index on **po.requisitions_interface** will dramatically reduce the LIO processing required to fulfill this update statement. A reasonable guess is that creating a composite index on the **req_number_segment1** and **request_id** columns would reduce the LIO count per execution from 7,047 to less than 10.

How much of a response time reduction should this create?

- Presently, the program is executing $8,216,887 \text{ LIO} / 1066.43 \text{ sec} = 7,705 \text{ LIO/sec}$. If we reduce the LIO count from 8,216,887 to only 11,660 (ten per execution), then we should expect this statement to consume only about 2 sec of CPU time.

- Furthermore, eliminating LIO calls will eliminate some (probably most) of the PIO calls that the unnecessary LIO calls motivated. So most of the wait time (388.55 seconds) will disappear as well (<http://www.hotsos.com/dnloads/1.Millsap2001.11.14-LIO.pdf>).

The total impact of repairing this one SQL statement should be on the order of 20 minutes of response time reduction (about 50% of the program's total response time). The next step in the analysis of this 40-minute (now 20-minute) job is to identify the SQL statement(s) that account for the preponderance of the remaining elapsed time.

Note, by the way, that by this statement's database buffer cache hit ratio is excellent: it's $(8,216,887 - 3,547) / 8,216,887 = 99.9568\%$. By the hit ratio standard, this statement was already exquisitely well-tuned. **Hit ratios simply don't work** (<http://www.hotsos.com/dnloads/1.Millsap2001.02.26-CacheRatio.pdf>).

Assessment

- Creating a new index reduced response time by about 20 minutes (50%)
- The same index actually fixed other inefficient SQL statements as well, resulting in response time reduction > 90%
- Repeated application of these techniques reduced IT expenses dramatically

Several exciting things happened for this customer after repeated application of the techniques illustrated here:

- An almost 10x elapsed time reduction in month-end financials close processing.
- Reduction from more than 6 hours (before) to less than 20 minutes (after) in more than *twenty* different standard and customized Oracle Applications batch programs.
- Reduction from 71.78 seconds per execution (before) to 0.88 seconds per execution (after) in a custom applications query that is executed 3,000 times per day.
- Reduction in several standard and customized applications on-line functions from several minutes to just a few seconds.
- Ultimately, the customer was able to decommission its Oracle Parallel Server installation. The optimized workload was small enough to fit on a single server. This further improved performance (eliminating all the 'global cache lock%' events), and reduced hardware, software, and labor costs by a factor of at least 2x for the system.

Lessons learned

- Knowing *which SQL to optimize* is key
- *LIO call elimination* frees up CPU capacity
- Workload reduction *saves money*
- Sometimes, the SQL optimization is *surprisingly simple*
- Hit ratios *don't work*

Case 3

Inefficient application kills performance

Subjective examination

- PowerBuilder application is too slow
- Many neighboring companies are having the same performance problems

Objective examination

Oracle Kernel Event	Duration	Calls	Avg
SQL*Net message from client	166.60s	91.7%	6,094 0.027338s
CPU service	9.65s	5.3%	18,750 0.000515s
unaccounted-for	2.22s	1.2%	
db file sequential read	1.59s	0.9%	1,740 0.000914s
log file sync	1.12s	0.6%	681 0.001645s
SQL*Net more data from client	0.25s	0.1%	71 0.003521s
SQL*Net more data to client	0.11s	0.1%	108 0.001019s
free buffer waits	0.09s	0.0%	4 0.022500s
SQL*Net message to client	0.04s	0.0%	6,094 0.000007s
db file scattered read	0.04s	0.0%	34 0.001176s
log file switch completion	0.03s	0.0%	1 0.030000s
log buffer space	0.01s	0.0%	2 0.005000s
latch free	0.01s	0.0%	1 0.010000s
direct path read	0.00s	0.0%	5 0.000000s
direct path write	0.00s	0.0%	2 0.000000s
Total	181.76s	100.0%	

When we first received the trace data behind this profile in one of our classes, I was apologetic. I re-emphasized to the student how important it is that we collect trace data for **exactly** the problem interval. On first glance, this profile seemed to be stereotypical of trace data that contains lots of extra “server idle” time because of sloppy data collection. However, there are two undeniable pieces of evidence to the contrary:

- The student assured me that she was very meticulous in how she created the trace data (“every second you see here was part of the user’s response time”).
- And of course, look at the number of calls to ‘SQL*Net message from client’. Over six thousand calls to the in-between-calls event to perform less than a dozen seconds of real SQL processing? Unconscionable!

Suddenly this became a very interesting profile.

Objective examination

Interval Time by Statement (808 distinct statements; 705 similar)

```
----- Exclusive -----  
Elapsed      Pct      CPU      Similar  SQL Statement Tree  
-----  
2.78s      1.5%     1.40s     302     2038166283  
...      ...      ...      ...     ...
```

The first indication of root cause is shown here...

- There are 808 distinct SQL statements in this program, and 705 of them are distinct but shareable. This indicates that the application does not use bind variables. This is an immediate indication that this application can never scale to large user counts (http://asktom.oracle.com/pls/ask/f?p=4950:8::::F4950_P8_DISPLAYID:2444907911913).
- The worst time consumer in the whole program is a SQL statement that consumes only 1.5% of the total program response time. This is usually another indication that most of the response time is spent **between** SQL statements, not inside them.

Objective examination

```
statement id = 2038166283
```

Oracle Kernel Event	Duration		Calls	Avg
-----	-----	-----	-----	-----
SQL*Net message from client	38.43s	90.8%	1,632	0.023548s
CPU service	2.18s	5.2%	909	0.002398s
db file sequential read	1.34s	3.2%	323	0.004149s
log file sync	0.32s	0.8%	230	0.001391s
free buffer waits	0.04s	0.1%	2	0.020000s
SQL*Net message to client	0.00s	0.0%	1,566	0.000000s
-----	-----	-----	-----	-----
Total	42.31s	100.0%		

The resource profile for virtually every SQL statement looked like this. Lots of time consumed by the cursor on 'SQL*Net message from client' Oracle kernel event processing. Again, remember, time logged to this event is time consumed by the database after fulfilling a database call, but before the next database call arrives.

A lot of Oracle analysts refer to this as "server idle time," which is indeed true. However, it is a mistake to ignore this so-called "idle time" because when we have chosen our observation interval carefully (as the Hotsos Method dictates), this time is **part of the user's response time**.

This phenomenon almost always indicates wasteful application design. We can confirm or refute this hypothesis by looking at the statement's database call summary...

Objective examination

```
statement id = 2038166283
```

```
INSERT INTO STAGING_AREA (TMSP_LAST_UPDT, OBJECT_RESULT,  
USER_LAST_UPDT, DOC_OBJ_ID, TRADE_NAME_ID, LANGUAGE_CODE)  
values(TO_DATE('11/05/2001 16:39:04', 'MM/DD/YYYY HH24:MI:SS'), 'if (  
exists ( stdphrase ( "HND_CW" ) ), stdphrase ( "HND_CW" ) , "" )',  
'sa', 61, 54213, 'NO_LANG')
```

```
----- Response Time -----  
Action   LCM Count Rows Elapsed   CPU   Waits   LIO Blks  PIO Blks  
-----  
Parse    303  606   0      0.58   0.68  -0.10     9        3  
Execute  0    303  303    2.88   1.50   1.38     4,759    331  
Fetch    0    0    0      0.00   0.00   0.00     0        0  
-----  
Total    303  909  303    3.46   2.18   1.28     4,768    334  
Per Exe  1    1    1      0.01   0.01   0.00     16       1  
Per Row  1    1    1      0.01   0.01   0.00     16       1
```

The problems are prominent in the database call statistics.

Bind variables

- First, the application is generating SQL that cannot possibly be shared. This mistake begs for both library cache and shared pool latch contention. It causes unnecessary memory consumption in the shared pool and unnecessary CPU workload to manage the extra parsing. At least the date and any high cardinality values should be referenced in the SQL text bind variables instead of literals.
- Notice that the absence of bind variables has caused 303 library cache misses (LCMs). The CPU workload associated with the parsing and library cache management activities is completely unnecessary here.

Excessive parsing

- Scalability 101: **Never** parse every time you execute. For each parse, you should be able to execute many times, each perhaps with a different set of values bound to the statement's bind variables. Well, this application goes a step even worse: it parses **twice** for every execute. It parses once, forgets the result, and then immediately parses again for every execution. Amazing.

No array processing

- The application inserts only one row per execution. However, Oracle is capable of operating much more efficiently by inserting many rows per execute call. Another opportunity to reduce workload, and—in this case, more importantly—the number of 'SQL*Net message from client' waits between db calls.

Impact

- Virtually every statement in this program's execution was parsed 606 times, executed 303, and then many were fetched 303 times as well. Using bind variables and reducing the number of parses to 1 per statement reduces the total parse count from thousands to approximately 100 (808 distinct statements, 705 similar). Using array insert and array fetch operations reduces the total execute and fetch count by a similar magnitude. In the end, we should be able to run this program with just a few hundred db calls instead of over 6,000. The response time savings should be on the order of 90%.

Scalability 101

```
HORRIBLE:    loop
                cursor = parse(text)           /* BAD! */
                execute(cursor, val1, val2)
                close(cursor)                  /* also bad! */
            end

EVEN WORSE: loop
                cursor = parse(text)           /* BAD! */
                cursor = parse(text)          /* just kill me */
                execute(cursor, val1, val2)
                close(cursor)                  /* also bad! */
            end

MUCH BETTER: cursor = parse(text)           /* outside the loop*/
                loop
                    execute(cursor, val1, val2)
                end
                close(cursor)                  /* outside the loop */
```

Many application developers write code that won't scale. Here is an example of one common mistake ("HORRIBLE"): opening and closing a cursor inside a loop. The solution is simple in principle. Simply parse once, execute many times with different bind values, and then close the cursor once. For some reason, the PowerBuilder application in this example did something even worse than horrible; it parsed **twice** per execution.

Modern multi-tier applications scale even better than one parse per database connection; they perform **one parse per application SQL statement**. To achieve this scalability improvement, an application middle tier at startup time will open all the cursors that the application will ever need. Applications then share these cursors through a process of **connection pooling**. Connection pooling can make trace data collection a little more difficult (<http://www.hotsos.com/dnloads/1/10046a>), but the scalability advantage is well worth the effort for applications that are required to serve tens of thousands of concurrent users (<http://www.hotsos.com/dnloads/1.Holt,Millsap2000.03.01-Scaling.pdf>).

Assessment

- Use *bind variables* to share SQL
- Eliminate unnecessary *parse calls*
- Use *array processing*
- Response time reduction > **90%**

Lessons learned

- You **can't just ignore** SQL*Net message from client
- **Too many database calls** can ruin performance
- ...Even when your SQL is just fine

Q U E S T I O N S
A N S W E R S