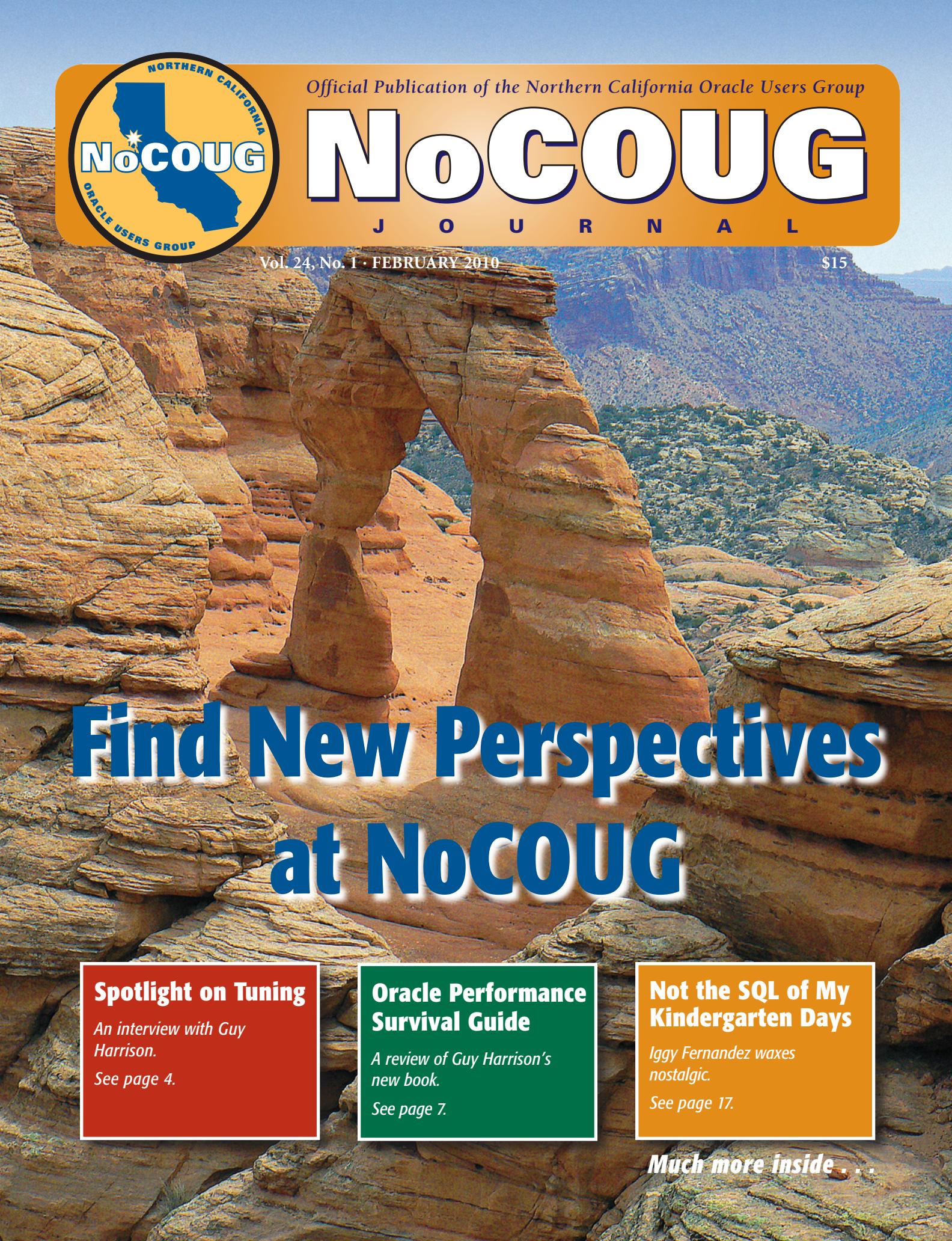
Official Publication of the Northern California Oracle Users Group

NoCOUG JOURNAL

Vol. 24, No. 1 · FEBRUARY 2010 $15



Find New Perspectives at NoCOUG

Spotlight on Tuning

An interview with Guy Harrison.

See page 4.

Oracle Performance Survival Guide

A review of Guy Harrison's new book.

See page 7.

Not the SQL of My Kindergarten Days

Iggy Fernandez waxes nostalgic.

See page 17.

Much more inside . . .

Standing On the Shoulders of Giants

Someday I'll be a better performance analyst than Craig Shallahamer, but in order to do that I will have to stand on his shoulders. Referring to his teacher Isaac Barrow and other prominent scientists of his day, Isaac Newton said: "*If I have seen further it is by standing on the shoulders of giants.*" Luckily I won't have to actually stand on Craig's shoulders or even travel to meet him; I can just read his new book *Oracle Performance Firefighting*, into which he has condensed twenty long years of Oracle performance firefighting experience.

You too can stand on the shoulders of a giant by reading this issue of the *NoCOUG Journal*. In this issue, we interview Guy Harrison, review his new book *Oracle Performance Survival Guide*, and even print an extract from the book. I hope you enjoy this issue of the *Journal*, and I hope to see you at our winter conference on Thursday, February 11, at the CarrAmerica conference center in Pleasanton. ▲

—Iggy Fernandez, *NoCOUG Journal* Editor

Table of Contents

ADVERTISERS

Publication Notices and Submission Format

The *NoCOUG Journal* is published four times a year by the Northern California Oracle Users Group (NoCOUG) approximately two weeks prior to the quarterly educational conferences.

Please send your questions, feedback, and submissions to the *NoCOUG Journal* editor at **journal@nocoug.org**.

The submission deadline for the upcoming May 2010 issue is February 28, 2010. Article submissions should be made in Microsoft Word format via email.

2010 NoCOUG BOARD

ADVERTISING RATES

The *NoCOUG Journal* is published quarterly.

Size	Per Issue	Per Year
Quarter Page	$125	$400
Half Page	$250	$800
Full Page	$500	$1,600
Inside Cover	$750	$2,400

Personnel recruitment ads are not accepted.

journal@nocoug.org

A New Decade, A NoCOUG You Can Count On

Hanan Hit

by Hanan Hit

Greetings Oracle professionals! A new year is here, and NoCOUG is here to help you develop your career and personal network, just as we've been doing for more than 22 years.

I'd like to start with a special thanks to the NoCOUG board members, who have done an outstanding job over the past year to keep NoCOUG as one of the most successful and prestigious user groups in the country—and worldwide. In addition I would like to thank the loyal user community for being such a great supporters of the NoCOUG organization.

In 2010 you can look forward to conferences with presentations by industry leaders such as Craig Shallahamer, Dan Tow, Neil Gunther, and Bradley Brown—to name a few.

We'll also be holding a special training event with Craig Shallahamer, and we'll be publishing the *NoCOUG Journal* each quarter and adding useful content to **www.nocoug.org** throughout the year.

In the last year we heard many stories about major drops in IT spending—for noncritical projects as well as operating budgets. I truly believe that we've reached the bottom of the decline and with the new decade upon us, we will see renewed IT investments in projects that will yield operational efficiencies and generate a better return to the business in the form of profits, competitive advantage, and more. Being a NoCOUG member can help Oracle professionals gain the required knowledge to benefit from these renewed IT investments.

If you're not a NoCOUG member, I encourage you to join today. Where else does $95 buy you four days of Oracle education throughout the year? You can join online at **www.nocoug.org**. In addition to education, you'll receive valuable information published in our quarterly journal and on the NoCOUG website. NoCOUG also offers great networking opportunities: at NoCOUG conferences you can meet scores of fellow Oracle professionals who live and work right here in Northern California. In addition, NoCOUG training days provide an opportunity to get further in-depth training in specialized areas at discounted prices. I also encourage everyone to spread the word: Tell your officemates, colleagues, and friends about NoCOUG. You can do a great favor for other Oracle professionals by introducing them to our users group. Increasing the NoCOUG membership, and thus enlarging our network, benefits all of us.

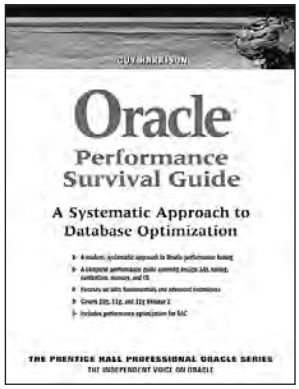
With the new year comes some changes in the NoCOUG leadership. Our users group is run by fourteen people. Thirteen are volunteers who take time out of their busy lives to share with the community and make NoCOUG what it is. It takes a lot of energy and time to manage an organization with over 450 members, four conferences per year, a quarterly publication, and a dynamic website.

I will be your loyal and faithful president this year, with Jen Hong at my side as vice president. Randy Samberg will continue to act as the director of conference programming, Iggy Fernandez as *Journal* editor, Naren Nagtode as secretary/treasurer, Joel Rosingana as membership director, Eric Hutchinson as webmaster, Chen Shapira as training day coordinator, Claudia Zeiler as track leader, and Noelle Stimely as director at large. I would like to welcome three additional volunteers: Jenny Lin as director of marketing, Scott Alexander as director at large, and Omar Anwar as track leader.

I look forward to seeing you at NoCOUG's Winter Conference, February 11 at the Carr America Conference Center in Pleasanton. Neil Gunther will kick off the event with a keynote entitled *Why Are There No Giants* which promises to be a very interesting topic.

After the keynote there will be 12 technical presentations, including two additional presentations by Neil Gunther. Bradley Brown will speak on SOA-based ODS and ESB, Dan Tow will present a case study of the importance of reaching recent rows first, and local Oracle user Chen Shapira's subject is "What Every DBA Should Know About TCP/IP." Robyn Sands will speak on "Measuring for Robust Performance." Andy Fenselau will cover "The Challenge of Virtualizing Databases," which has become a very hot topic. Ahbaid Gaffoor will speak on subjects such as script management for Oracle. Finally, we will have four Oracle team members reviewing the latest Oracle technologies.

I look forward to seeing you at the February 11 NoCOUG Winter Conference. Together we can get our careers off on the right foot in 2010—the start of a promising new decade. ▲



Spotlight on Tuning

With Guy Harrison

Guy Harrison

Guy Harrison is the author of the newly released Oracle Performance Survival Guide, MySql Stored Procedure Programming *(with Steven Feuerstein, 2006),* Oracle SQL High Performance Tuning *(2001), and* Oracle Desk Reference *(1999). He is currently a director of development at Quest Software, where he created the popular Spotlight family of products. He leads a software development team in Melbourne that is responsible for the Spotlight core technologies and for SQL Navigator, Spotlight on Oracle, Spotlight on Oracle RAC, Spotlight on MySQL, and Spotlight on SQL Server Enterprise. He can be found on the Internet* **at www.guyharrison.net**.

We spend a lot of time tuning databases. Why does it have to be that way? Is Oracle difficult to tune? Are we building the databases or applications wrong?

We dream of having a database that requires no tuning, but that probably won't happen—at least not until computers reach human levels of intelligence. We have to express our wishes to the database in terms of SQL and PL/SQL: poorly expressed or unnecessary requests are always going to put more demand on the DBMS, and so there'll always be the need to look at those requests and make sure that they are optimal. Also, there'll always be a variety of ways we can set up and configure the database to meet our anticipated demand. So it seems like we'll always have to tune our code and optimize our database configuration.

Although every release of the database automates some aspect of tuning, every release also introduces more functionality that you need to tweak if you want to get the best performance. For example, in 11*g*, Oracle can automatically allocate memory between the buffer cache, shared pool, and PGA—which is a Good Thing. However, 11*g* also introduced new caches: the database flash cache and the result set cache for instance. These new areas are not automatically managed, so if you want to use them, you need to manually determine the best configuration.

Oracle has the most advanced features of any RDBMS, and this creates more tuning challenges and opportunities. On the other hand, Oracle gives you more visibility into internal state than MySQL or SQL Server do. The wait interface and the time model in particular let you easily diagnose issues that would be totally obscure in other databases.

Your book talks about "tuning by layers." At first glance, it sounds like the *Compulsive Tuning Disorder* that we are often warned about. We are now told to focus on response times and wait times (Method R, Yapp, etc). Why should we tune by layers?

You are always better off tuning to response times and wait times, but your tuning efforts will often be more effective if you focus on the higher levels in the software stack first. Tuning by layers attempts to guide your tuning efforts toward *causes* rather than *symptoms*.

It's the higher levels of the stack—Application SQL for instance—that create the demand on the lower levels such as the I/O subsystem. It makes sense to minimize demand first: if you reduce the demand from the higher levels, the symptoms you are seeing in the lower levels may disappear.

For example it is better to reduce I/O *demand* by adding an index or otherwise tuning an SQL, rather than to increase I/O *capacity* by adding spindles to a disk array. Both approaches can reduce your I/O wait times, but the first focuses on the *cause*, while the second focuses on a *symptom*.

Because problems in one database layer can be caused or cured by configuration in the higher layer, the most efficient and effective way to optimize an Oracle database is to tune higher layers before tuning lower layers:

1. Reduce application demand to its logical minimum by tuning SQL, optimizing physical design (partitioning, indexing), and tuning PL/SQL.

2. Maximize concurrency by minimizing contention for locks, latches, buffers, and other resources in the Oracle code layer.

3. Having normalized logical I/O demand by the preceding steps, minimize the resulting physical I/O by optimizing Oracle memory.

4. Now that the physical I/O demand is realistic, configure the I/O subsystem to meet that demand by providing adequate bandwidth and evenly distributing the resulting load.

Cars need regular maintenance to keep them running smoothly. Don't databases need maintenance too? What maintenance would you suggest, if any?

When a car gets out of tune, it probably still gets you where you want to go, but it will cost you more in fuel and generate more pollution. That's essentially true for databases as well. As well as tuning databases to realize a response time or throughput goal, we should tune to reduce CPU and disk I/O, which in turn translates to lower power costs and environmental impact. I read recently that IT is contributing to between 2% and 6% of global power consumption and growing exponentially: I think we'll see increasing pressure to contain that growth, and performance optimization is an important part of the solution

The other reason to perform proactive tuning is to avoid the "brick wall" that you can hit when you run out of system resources. It's a lot easier to create an index or add some memory when system performance is adequate than to have to do so in crisis mode when a performance issue is bringing the application to its knees.

Routine, proactive tuning is a good candidate for the tuning-by-layers approach. Start by seeing if the SQLs with the highest cost can be tuned, reduce contention points that might be masking demand, optimize memory to avoid I/O, and then optimize the I/O subsystem to meet the resulting realistic demand.

STATSPACK seems to be out of fashion, but it's the only system-level diagnostic tool available to most DBAs. What good is STATSPACK really?

Well, I work on third-party tuning tools for Oracle (Spotlight on Oracle/RAC for instance), so I have a slight conflict of interest here! However, my view is that while advanced tools can massively improve the efficiency of the DBA and allow DBAs with limited experience to quickly determine bottlenecks and tuning opportunities, there's enough in the base Oracle product to solve most problems given enough time. If all I had were SQL*Plus and access to V$ views, I'd feel confident that I could eventually get to the bottom of most tuning issues.

I think STATSPACK is a fine set of scripts and framework for keeping history. I haven't used it all that much, though, since I've always had access to Quest tools such as Spotlight ☺.

Why bother to tune at all? Is big iron the cheaper answer to our tuning problems? My time is not cheap but memory and RAM disks get cheaper every day and CPUs keep getting faster. In his latest book, *Oracle on VMware*, Dr. Bert Scalzo makes a case for "solving" performance problems with hardware upgrades[1]. What is your opinion about this approach?

As I said before, datacenter power consumption costs and environmental impact are likely to become increasingly important. Bert is correct in that the economics of tuning are such that hardware upgrades are often more viable than lengthy tuning cycles. However, companies are going to be increasingly challenged to reduce power consumption from their data centers, and this may become at least as important as optimizing DBA time.

The other issue is scalability. Poorly tuned SQL often increases in demand faster than you can add hardware—especially if the data volumes and transaction rates are growing at the same time. If you have a transaction that is taking longer for each execution, and the execution rate is also increasing, then you may see exponential cost increases and you definitely can't add hardware at an exponential rate.

All that having been said, it's true that for certain workloads—data warehousing being the prime example—there's often no better solution to the performance challenge than to parallelize the workload across multiple RAC nodes and disk spindles and use a brute force solution.

One of the very unusual recommendations in your book is to consider adjusting _SPIN_COUNT in certain circumstances. Most Oracle performance tuning specialists are against the idea. Why do you recommend it?

Gulp . . . I didn't really mean to *recommend* altering an undocumented parameter. Oracle support will have me excommunicated!

What I do argue is that when latch contention can't be resolved by any other measure, then changing the spin count might be worth trying. After all, the _SPIN_COUNT has stayed at a constant value for decades, while CPUs are getting faster: the default value can't possibly be at the absolutely correct value for all circumstances and probably is below optimal for some databases.

I've experimented with adjusting spin count on a couple of occasions, and in both cases I found that for the right workload, you could reduce latch contention and improve throughput if you increased _SPIN_COUNT. You can see the results of my most recent tests at **tinyurl.com/spincount**.

Although I think adjusting _SPIN_COUNT is a valid strategy to try when all else fails, I definitely advise that other mechanisms for reducing latch contention be attempted first.

You're the architect of Spotlight on Oracle and similar products. Would you like to tell our members something about them while you have their attention?

Sure. Quest has been doing database tools for about 15 years now, and we have a huge user base in the Oracle community. Spotlight on Oracle and Spotlight on Oracle RAC are my "babies" at Quest—I designed them and I run the teams that work on them. My aim for Spotlight is to create the most sophisticated yet easy-to-use tuning tool for Oracle DBAs. The great thing about working at Quest is that I can build my own dream tool. Spotlight has the diagnostic capabilities that I most desire in a tuning tool.

The good news for Oracle DBAs—especially if you use

[1] Here's the full quote from Dr. Scalzo's book: *"Person hours cost so much more now than computer hardware even with inexpensive offshore outsourcing. It is now considered a sound business decision these days to throw cheap hardware at problems. It is at least, if not more, cost effective than having the staff [sic] tuned and optimized for the same net effect. Besides, a failed tuning and optimization effort leaves you exactly where you started. At least the hardware upgrade approach results in a faster/better server experiencing the same problem that may still have future value to the business once the fundamental problem is eventually corrected. And, if nothing else, the hardware can be depreciated, whereas the time spent tuning is always just a cost taken off the bottom line. So, with such cheap hardware, it might be a wiser business bet to throw hardware at some solutions sooner than was done in the past. One might go so far as to make an economic principle claim that the opportunity cost of tuning is foregoing cheap upgrades that might fix the issue and also possess intrinsic value. Stated this way, it is a safe bet that is where the business people would vote to spend."*

TOAD—is that the prices of these tools have really dropped over the past few years. You can get Spotlight on Oracle— and soon Spotlight on RAC—in the TOAD DBA suite now, at a fraction of the cost that it would have cost you a few years ago. TOAD users should definitely check out the DBA suite which offers—I think—an almost irresistible value.

> ***"MySQL might become an Oracle database any day now if the Sun acquisition completes, and if that happens knowing both MySQL and Oracle could be an advantage."***

It costs us about $14,000 per year to produce and distribute the *NoCOUG Journal* and considerably more than that to organize four conferences. We have about 500 members and about 250 attendees at each conference; we've stagnated at those levels for many years. Has Google made us obsolete? I begged my previous company to buy a corporate membership but, except for my manager, not a single person from that company ever showed up at our conferences. Should we close shop? What could we do better to attract more members and conference attendees?

Google is great for quick answers to sudden questions, but it doesn't provide a structured way to improve your overall expertise. It also doesn't provide the opportunities for networking that user groups provide.

We're clearly going through a transition in how we obtain and consume information. As an author I'm all too aware of how much the Web and web searches have changed the value proposition for periodicals, conferences, and books. I'm confident, though, that the value of the community in user groups will stand the test of time.

As far as the *Journal* goes, I think anyone who is publishing information needs to keep thinking about how best to get the information to the user. I'd be thinking how to exploit the new distribution mediums—Kindle, iPhone, podcasts, RSS aggregation, Twitter, etc. It's not going to be easy: just look at the challenges faced by the newspaper industry.

The other thing to remember is that the IT industry in general and the database market in particular are reaching a level of maturity that might feel like stagnation given the crazy growth we had during the dot.com bubble and Y2K, but it's probably a natural transition for an industry entering adulthood.

You and Steven Feuerstein are Oracle experts, yet you collaborated on a book on MySQL. Should we hedge our career bets by learning MySQL?

That's a good question. When we wrote the MySQL book, a lot of people thought MySQL might start to compete more directly with Oracle in the enterprise market. It didn't turn out that way: MySQL and Oracle still exist in pretty distinct segments of the overall database market. I don't think Oracle professionals should feel their career is threatened by MySQL.

On the other hand, MySQL might become an Oracle database any day now if the Sun acquisition completes, and if that happens knowing both MySQL and Oracle could be an advantage.

I think that all IT professionals should try hard to keep a diverse skill set. Learning new technologies helps to keep your mind active, puts your primary skills in context, and helps you to avoid being on the wrong side of a paradigm shift. If I were a PL/SQL programmer, I'd be making sure I knew at least one other language pretty well—maybe Java or Ruby. As a DBA, I'd be keen to get some experience with any other RDBMS—especially SQL Server or MySQL.

At the moment I'm very interested in some of the non-relational technologies that may enter the mainstream over the next few years. Hadoop and the "NoSQL" databases (such as Cassandra and Voldemort) are really interesting. It's too early to tell if they will become disruptive to the relational database, but they're definitely worth keeping an eye on.

Thanks for an excellent book and thanks for spending time with us today. KahPlah! I have your book now, but I'm always interested in buying good Oracle books. Do you have some personal favorites that you can recommend?

Thanks, Iggy!

There are so many good books are out there! I'm sure I'm going to miss some worthy contributions, but here goes:

Tom Kyte's *Expert Oracle Database Architecture* is the best overall book on Oracle architecture that I know of. It's highly readable, comprehensive, and accurate. Despite the word "Expert" in the title, I think it's suitable for people at almost any level of expertise. If I were only going to give someone a single book on Oracle, it would probably be this one.

In terms of introductory books, I thought your recent book (*Beginning Oracle Database 11g Administration*) was an excellent introduction to Oracle DBA. Julian Dyke's *Pro Oracle Database 10g RAC on Linux* is a good book to get up to speed if you're using RAC (with or without Linux).

For more advanced readers, the best book on Oracle SQL tuning is without doubt Jonathan Lewis's *Cost Based Oracle Fundamentals*. Other recent advanced tuning books include Craig Shallahamer's *Oracle Performance Firefighting* and Christian Antognini's *Troubleshooting Oracle Performance.*

There are lots of other books. I recently set up an online bibliography at **guyharrison.net/readingList**, where I list the books that I've found particularly useful.

Live long and prosper! ▲

Interview conducted by Iggy Fernandez

Oracle Performance Survival Guide

A Book Review by Dave Abercrombie

Details

Author: Guy Harrison

ISBN-13: 978-0-13-701195-7

ISBN-10: 0-13-70119-54

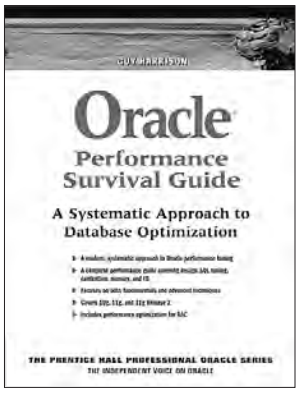
Pages: 730

Publication Date: October 2009

Edition: 1

Price: $59.99 (list)

Publisher: Pearson Education (Prentice Hall)



Summary

Overall review: An encyclopedic overview of all aspects of Oracle performance, addressing all layers from high-level application design through SQL tuning, PL/SQL, locks, and contention, all the way down to memory configuration and disk layout.

Target audience: Application developers and DBAs

Would you recommend this to others? Yes.

Who will get the most from this book? Application developers who want to do the right thing, and performance experts or DBAs who want to broaden their knowledge.

Is this book platform specific? Yes: 10*g*, 11*g*, 11*g*R2 (with context from 8*i* and 9*i*)

Why did I obtain this book? Guy Harrison has been a respected author in this field for more than a decade. Back in the days of version 8, his writing opened my eyes to the world of Oracle performance tuning. When I learned that he published this new book—updated and greatly expanded in scope—I was determined to read it.

Overall Review

This book is an encyclopedic overview of all aspects of Oracle performance. Guy Harrison takes a layered approach, starting at the top with application and data model design, where the focus is minimizing the demand for database resources. He then moves down into database code internals, where the focus is maximizing concurrency through reduction of lock, latch, and mutex contention. The next step is to optimize memory usage to minimize the need for physical I/O. Finally, he moves to the bottom layer, where the focus is on optimizing physical I/O at the disk layer.

Each of these layers is worthy of its own book, so to combine all of these topics in a single book is an ambitious goal. Indeed, experienced readers will often want a bit more detail or wonder why their favorite optimization is not mentioned. However, Harrison strikes a very good balance between depth and coverage. He also provides a very useful bibliography, including Oracle documentation, books, and Internet sites and blogs.

For each chapter, Harrison provides extraordinarily clear, concise, and helpful introductions and summaries. He also uses boxed borders to highlight particularly important points within the text. You can learn a great deal by simply reading these summaries and boxed items, and these can also be used to help the reader find relevant sections, which is especially valuable in a book of this length.

Part I: Methods, Concepts, and Tools

In recent years, Oracle performance experts have moved away from "ratio-based" tuning techniques toward use of the wait interface, and this has brought radical improvements in our ability to diagnose and tune. Harrison begins this book by warning us not to rely too much on wait event analysis. He warns that by concentrating only on the largest wait events, we may fall into the trap of treating symptoms rather than causes, we may be tempted to waste money on hardware-based solutions, and we may not reach permanent, scalable solutions.

Harrison's methodical tuning-by-layers approach. which forms the foundation of this book, helps us to avoid these traps. Harrison describes the four layers of an Oracle application: Application, Database, Memory, and Disk. Tuning by layers starts at the top-level **Application** layer, where the goal is to minimize the application workload by tuning application code (optimizing SQL, client-side caching, reducing request rate, etc.) or modifying the physical implementation (indexes, denormalization, partitioning, etc.). Once the application workload has been reduced, we move down to the next layer, **Database**, where the focus shifts to reducing contention and bottlenecks (transaction locks, latches, mutexes, etc.). The next

step is to optimize Oracle's **Memory** usage (block buffer cache, PGA, etc.) to reduce physical I/O. Only after problems in the preceding layers have been addressed does focus shift to the **Disk** layer, where we work to ensure adequate bandwidth and balance I/O demands. Interestingly, Harrison's tuning-by-layers approach does not explicitly cover CPU utilization issues such as the number or speed of CPUs.

Harrison devotes the next chapter to an overview of Oracle's basic architecture. Obviously, this huge topic cannot be thoroughly described in a single chapter, so some subtleties are glossed over and many aspects are greatly simplified. Experts may quibble with some passages, but a concise overview like this is both welcome and essential.

Harrison's next chapter covers the basic tools used for assessing and improving Oracle performance. Although several commercial tools are available for purchase, he focuses on those that come with Oracle. Here, and throughout the book, Harrison does an admirable job of pointing out tools like Active Session History (ASH) and Active Workload Repository (AWR) that are built in and begging to be used, but that require Oracle's extra Diagnostic Pack license for legal use.

Harrison first shows how to use `EXPLAIN PLAN` to generate a SQL execution plan, and he briefly illustrates how to use `DBMS_XPLAN` to view the plan. However, he does not go into detail about how the plan produced by `EXPLAIN PLAN` may vary from the "real" plan nor how to deal with this limitation. Harrison briefly hints at the use of setting `STATISTICS_LEVEL` to `ALL` but he does not illustrate this powerful technique. He misses a wonderful opportunity here to introduce the amazingly useful "tuning by cardinality feedback" approach popularized by Wolfgang Breitling. Harrison also neglects to emphasize the need for representative datasets when tuning: I'm sure we've all seen queries that worked fine in development but were problematic in production due to the lack of test data that adequately resembles production.

Harrison next illustrates basic SQL execution tracing and shows how to format trace files with Oracle's `tkprof`. He also provides tips on finding your trace files, including the use of the nifty `TRACEFILE_IDENTIFIER` session parameter trick. He explains when you might need to merge multiple trace files using Oracle's `trcsess` tool and shows how to use it. Harrison also introduces using `DBMS_MONITOR` to initiate traces by using the end-to-end metrics (aka `DBMS_APPLICATION_INFO`) tags `MODULE` and `ACTION`, although he does not illustrate the many other wonderful benefits of using these session-level tags. Of course, Harrison devotes several pages to interpretation of trace data. He concludes this concise and useful overview of tracing with pointers to `tkprof` alternatives.

Harrison walks us through the use of the `AUTOTRACE` tool with in `SQL*Plus`. This tool combines execution plan information from `DBMS_XPLAN` with a subset of session-level statistics from `V$SESSTAT`. This useful `AUTOTRACE` summary would have been improved, however, through mention of the benefits of querying `V$MYSTAT`, which provides performance details ignored by `AUTOTRACE`.

Harrison spends only a couple of pages summarizing Oracle's `V$` view interface, with an emphasis on the subset that exposes the wait interface. These topics are huge and complicated, with hundreds of `V$` views, wait events, and performance statistics, so he is only able to scratch the surface within the scope of this book. This book is not primarily a troubleshooting book, so it is no surprise that these topics are covered so briefly. However, it is perhaps surprising that no mention is made of Oracle's free STATSPACK or its enhancement `AWR` (which requires a Diagnostic Pack license). Systemwide tools like STATSPACK and friends can be essential in finding and prioritizing problems that can be solved with Harrison's tuning-by-layers approach.

Part II: Application and Database Design

Harrison emphasizes the need for sound database design, since the data model determines the ultimate performance limits of the application. He makes a clear distinction between logical and physical database design. Traditional logical design ensures that "all necessary data is correctly, completely, and unambiguously represented" while generally ignoring performance considerations. Design then proceeds to the physical stage, where performance and scalability goals may cause the logical design to shift. Harrison walks us through many common physical design optimizations, including denormalization, star schemas, and materialized views.

Since the scope of this book allows only one chapter for this overall topic, some details are necessarily skipped. For example, he does not define "primary key" or "foreign key," so the novice could become confused by his discussion of "artificial keys." However, Harrison does an excellent job of pointing out potential risks of denormalization and provides a nice discussion of "vertical" partitioning (moving infrequently used columns of a frequently scanned table into a separate table). Still, Harrison devotes only a few pages to the very powerful techniques of range, hash, and list partitioning. His brief coverage of star schemas does a good job of whetting our appetites, and he kindly provides a reference to Bert Scalzo's book on the topic. Likewise, Harrison's coverage of `PCTFREE`, `PCTUSED`, and `LOB` storage is very brief but clear and concise.

Harrison's overview of B*-Tree indexing is aided by some very helpful graphics. He correctly points out that "creating widely applicable and selective concatenated (i.e., multi-column) indexes should be a top priority of your indexing strategy." However, he devotes only three pages to this topic, and the extraordinarily powerful trick of including enough columns to eliminate the need to read table blocks (i.e., a "covering index") receives only one sentence. In my experience, ignorance of these techniques is one of the biggest contributors to performance troubles, and just a few more pages here would have been very beneficial. However, he provides very helpful advice about finding unused indexes. He also provides a very useful overview of other index issues such as bitmap indexes, DML overhead, index organized tables, clustering, and nested tables. Interestingly, he does not discuss the index clustering factor; this factor is one of the main reasons an index might not be used, and it often contributes to differences in execution plan between test datasets and production. I would have preferred to see more discussion about the importance of data distributions and clustering factors.

Harrison provides a nice overview of application use of bind variables. His advice to use the FORCE_MATCHING_SIGNATURE column of V$SQL to find statements that should be using bind variables was quite welcome. He neatly describes how setting CURSOR_SHARING to FORCE will bring the benefits of bind variables to applications that cannot be rewritten to include them, but he does not warn about the plan instability that commonly arises from bind variables peeking into histograms in such an environment.

Of course, the most efficient SQL statement is one that is not executed, and so Harrison discusses caching, including many potential pitfalls. His overview of the 11*g* feature "client-side result caching" is very helpful.

Harrison's discussion of array fetches is most welcome. He shows how to set the array size within Java application code and claims that it "can provide approximately an order of magnitude improvement for bulk queries."

Harrison provides an excellent overview of isolation levels: read committed (Oracle's default), read only, and serializable. This is combined with an excellent introduction to locking strategies, pessimistic and optimistic, and how they can be implemented. For example, he provides a clear example of using the pseudo-column ORA_ROWSCN and the table DDL keyword ROWDEPENDENCIES to implement optimistic locking strategies. His advice on how to choose a locking strategy is most helpful, since this issue can be quite confusing.

Harrison suggests using PL/SQL-stored procedures to improve performance by reducing network roundtrips. However, this discussion could have benefitted from a few words on other benefits (e.g., more reliable transaction management) and management issues (e.g., dependencies and recompilations).

Part III: SQL and PL/SQL Tuning

Harrison introduces the Oracle optimizer, pointing out that it "makes good decisions across a wide range of circumstances, **but it has not become self-aware yet, and human intervention is often still required**." He neatly summarizes the optimizer's decision-making process. He mentions the static DBA_* views that expose table and column statistics, but curiously does not mention the USER_* versions that are often more available to the typical developer. He very briefly summarizes the optimizer cost calculations: although Jonathan Lewis's work on this topic is recommended in the bibliography, a reminder to the reader here too would have been nice. Harrison's discussion of histograms, 10*g* plan instability from bind variable peeking, and the 11*g* fix for this called "Adaptive Cursor Sharing" is very helpful.

Harrison provides very useful guidance for using DBMS_STAT to gather table, column, and index statistics. His description of stale statistics is very helpful (see the ALL_TAB_MODIFICATIONS view), as is his advice on how to set systemwide defaults. Harrison does a good job of explaining how to gather histogram statistics, including warnings about some of their pitfalls, but he does not explicitly describe how to remove a histogram or why you might need to do so. Harrison neatly discusses "multicolumn extended statistics," but unless you read very carefully, you might not realize that this is an 11*g* feature only.

Harrison reminds us that optimizer hints are actually strict directives, but they can only be obeyed if they arise in the proper context. He advises us that we should use them "only after you have exhausted less direct methods." The scope of this book does not allow a thorough discussion of hints, but Harrison does a good job of summarizing the most important points.

He also does a remarkable job of contrasting outlines and profiles. Outlines are used to guarantee plan stability despite changes in statistics or database configuration, while profiles "are intended to increase optimizer flexibility." Profiles are SQL-specific statistics that "are created by a SQL tuning task and that can then be used by the SQL tuning advisor to determine an optimum plan." Harrison points out many advantages of the SQL tuning advisor: it can spend more time optimizing than the real-time optimizer, it can actually run the SQL to help it optimize, it can tell when indexes are missing, and it can create a profile. Harrison points out that use of profiles requires a Tuning Pack license. He then provides a very nice summary of the new 11*g* feature called "SQL Baselines" that "are intended to supplement SQL profiles and eventually replace stored outlines."

Harrison then goes on to describe how the optimizer decides when to use an index and when to do a full table scan. He goes into a fair amount of useful detail, partially replicating optimizer math, but again seems to ignore CLUSTERING_FACTOR. However, his section on avoiding accidental full table scans is excellent, with relevant advice regarding NOT EQUALS conditions, searching for NULL values, and avoiding accidentally disregarding an index by incorrectly using a function. A useful supplement to this section would have been Tom Kyte's often repeated advice on using a function-based index to quickly search for a needle in a haystack.

Harrison wonderfully points out that an "inexperienced SQL programmer often uses EXPLAIN PLAN to determine that a full table scan has been avoided. If there is no full table scan, the programmer might conclude that the plan is a good one. In fact, there are usually a wide variety of index-based retrievals possible, and merely ensuring that one of these access plans is used does not mean that the SQL is optimized." He then repeats the excellent advice to use a concatenated index, that such an index is optimized if its column order "supports the widest range of queries" and, if possible, that it completely avoids the need for any table access.

Harrison then provides many useful tips on—and possible problems arising from—searching for ranges using the LIKE operator and multivalve single-column lookups. Sometimes we need to do full table scans, so Harrison provides many useful tips for optimizing necessary full table scans. These include lowering the high-water mark, optimizing PCTFREE and PCTUSED, reducing row length, compressing the table, and optimizing use of the block buffer cache. Especially intriguing is the idea of using the SAMPLE clause for queries where approximate answers are acceptable. For example, if you need to calculate an average value of an attribute, a sample of only 5% of the rows may provide a result that is just as useful as the correct answer, but it will run about 20 times faster.

Harrison provides an excellent and widely accessible over-

view of Oracle's join methods: nested loops, sort-merge, and hash. He follows this with a very good description of how the optimizer chooses a join method. The reader is then well prepared to understand how these methods can be optimized, for example by tuning indexes or properly sizing memory. Finally, he provides useful guidance on methods to avoid joins: denormalization, index clusters, materialized views, and bitmap join indexes.

Harrison brings up some interesting facts about special joins. For example, he points out that left and right outer joins will force a join order, which can greatly impact performance. He provides an excellent overview of STAR joins. He briefly discusses hierarchical join methods, but more details would be most welcome here. One very interesting trick he describes is how to use the PARTITION BY feature of analytic functions to avoid expensive correlated subqueries. He also has very interesting advice for anti-joins (finding rows in one table that do not match rows in another table): "you should almost never use a NOT IN sub-query when any of the columns involved is nullable—for this type of query, you either want to define the columns as NOT NULL, or add IS NOT NULL clauses to the WHERE clause."

Harrison provides an excellent overview of Oracle sorting, including optimal, one-pass, and multi-pass sorts. He describes how to estimate memory needs by looking at the TempSpc output from DBMS_XPLAN. He explains how to do a 10032 trace event to obtain sort details. He describes the advantages and disadvantage of indexes used to avoid sorts. Harrison's discussion on sort- and set-related topics is thorough and practical: maximums, minimums, top-N (RANK and DENSE_RANK), and counting rows. He provides a great discussion of GROUP BY, with useful advice that "you should never use a HAVING in place of a WHERE." He has useful advice about UNION and UNION ALL, although his discussion of MINUS and INTERSECT is a little skimpy.

Harrison is clearly a fan of the focused use of PL/SQL to improve performance, and his chapter on it provides an excellent overview of its advantages. He briefly discusses the time model views and V$SQL as ways of assessing PL/SQL resource consumption, but he does not mention other useful instrumentation techniques such as those based on DBMS_APPLICATION_INFO or V$MYSTAT, nor does he address potential dependencies and recompilation issues relevant for maintaining a live production system. Harrison introduces the use of DBMS_PROFILER and its 11*g* enhancement, the hierarchical profiler DBMS_HPROF. He provides a decent overview of BULK COLLECT array processing, pointing out, for example, that the 10*g* complier will default to an array size of 100 for a simple SELECT loop even if you do not code it that way. However, this automatic array processing does not happen with DML, since he suggests to "use the FOREALL statement to perform bulk inserts, updates, deletes and merges." He has charts demonstrating the performance benefit of array processing, but a general discussion of all advantages and disadvantages is lacking. For example, Tom Kyte has written that inserts done with periodic commits generate more redo than single-commit versions, so it would be interesting to have a fuller discussion on this topic.

Harrison provides excellent advice on tuning PL/SQL—insisting, for example, that the SQL first be optimized. He offers plenty of useful tips regarding loop optimization, short circuiting expressions, expression order in IF and CASE statements, and the NOCOPY clause.

Harrison provides an excellent chapter on parallel SQL, starting with an explanation of Oracle's parallel architecture, including its slave pool and its use of direct I/O (bypassing the buffer cache). He gives thorough advice on when to use parallel processing, how to configure it, and how to monitor it (e.g., V$PQ_TQSTAT). His section on optimizing parallel performance wonderfully ties all these concepts together. He concludes this chapter with welcome details on a miscellany of other parallel topics, including RAC and CREATE TABLE AS SELECT.

Harrison includes a wide-ranging chapter on DML tuning. Basic factors include optimization of the WHERE clause and the amount of index maintenance overhead. However, most of his discussion uses logical reads as the main metric, ignoring the amount of REDO or UNDO generated. For example, he does not mention the use of global temporary tables to eliminate REDO for those parts of the application where it is appropriate. Nevertheless, his discussion of the following topics is quite excellent: direct path inserts, multi-table inserts, deletions, update joins (aka "correlated updates"), MERGE, COMMIT (as seldom as possible!) and NOWAIT (use very cautiously!).

Part IV: Minimizing Contention

Harrison starts his section on contention with a very useful summary of transaction locks, including types and modes (e.g. TX "transaction" type in "exclusive" mode X). He continues with a neat summary of lock controls such as FOR UPDATE, SKIP LOCKED, NOWAIT, and WAIT. He mentions use of V$SYSTEM_EVENT to assess the impact of locks, but he suggests instead to use one of his downloadable scripts that takes snapshots and calculates deltas. He, of course, mentions that Oracle's built-in ASH will do this for you automatically, but requires special licensing. Although he does not mention it, the V$EVENT_HISTOGRAM view can also be very helpful too, especially if you compute periodic deltas, as STATSPACK does. Harrison provides several real-time queries that you can use during a transaction pileup. Especially useful is a CONNECT BY query into V$SESSION that shows the lock pileup "tree." This information is so valuable that it has been implemented as V$WAIT_CHAINS in version 11*g*.

Harrison's discussion of latches and mutexes is fairly short. He describes the use of one of his downloadable scripts to take snapshots and compute deltas for latch/mutex waits. He does, however, provide useful explanation and advice for some of the most commonly encountered latch and mutex problems. His bibliography suggests sources for more details on these topics. Harrison is a proponent of altering the hidden parameter _SPIN_COUNT, and he discusses a commercial product that can be used to optimize it. However, he clearly acknowledges the controversial nature of this advice while recognizing its limits: he states that "if the average CPU queue length is approaching or greater than 1, increasing _SPIN_COUNT is unlikely to be effective."

Harrison concludes this section with a brief overview of the buffer cache architecture. He discusses issues surrounding free buffer waits, recovery writer waits, buffer busy waits, and redo log buffer waits. Again, his bibliography suggests sources for more details on these topics.

Part V: Optimizing Memory

Proceeding down into the next layer, Harrison shifts his attention to minimizing physical I/O through optimization of memory allocation. He begins with a short summary of buffer cache principles. This sets the stage for buffer cache configuration and tuning, including dealing with Automatic Shared Memory Management (ASMM), introduced in 10*g*. His coverage of these topics is thorough, helpful, and obviously grounded in practical experience.

Harrison moves next to optimizing PGA memory, which is mostly used for sorts and hashes. The basic goal is to do these operations in memory, avoiding physical I/O. He thoroughly discusses PGA memory management and measuring its usage and efficiency. He provides complete advice on sizing the PGA with `V$PGA_TARGET_ADVICE`. He also does an excellent job of describing how to use the 11*g* Automatic Memory Management (AMM) feature. His discussion of shared pool sizing with `V$SHARED_POOL_ADVICE` is very helpful.

Part VI: I/O Tuning and Clustering

Proceeding down into his fourth and final layer, Harrison shifts his attention to optimizing physical I/O. He begins with an overview of disk I/O concepts and Oracle's I/O architecture. He has extensive and useful advice on measuring and monitoring Oracle I/O, but again neglects mention of STATSPACK.

Regarding optimizing file I/O, he brings up the interesting point that to reduce latency, disks should be run at less than full—perhaps only 50% to 70% full. He also bemoans the fact that disk vendors publish *throughput* metrics at 100% utilization (where *latency* suffers), and publish *latency* metrics at 0% utilization (which is unrealistic).

Harrison provides a nice overview of the various RAID levels. He advises against RAID5 due to its write penalty. Vendors often tout using their write caches to minimize this problem, but Harrison points out that these help only for bursts of write activity, with sustained write activity likely leading to performance degradation.

Harrison provides several useful tips for optimizing physical I/O, especially with the goal of maintaining predictable response time under bursts of activity such as sorts. He strongly suggests dedicated non-RAID5 for redo files.

Harrison moves on to discuss advanced I/O techniques such as Automatic Storage Management (ASM; this is a huge topic—see his bibliography), solid state disks, and Oracle's Exadata Storage Server. He thoroughly discusses changing the database block size from its default 8 KB, but advises against it.

Harrison concludes his book by discussing RAC. He starts with a wonderful overview of its architecture and provides useful advice on measuring cluster overhead. One major goal is to reduce global cache latency. He provides clear, basic advice on how to achieve this goal by optimizing the interconnect, balancing the cluster, and—especially—minimizing global cache requests.

Conclusion

Guy Harrison's new book has a very ambitious scope: it addresses all aspects of Oracle performance, from data model and application design through traditional query tuning and database configuration—all the way down to disk subsystem design. As it takes the reader step by step through these layers, Harrison's book demonstrates the wisdom of tuning the higher layers first before proceeding down into the lower layers. Due to its comprehensive scope, it cannot delve deeply into details at every step. Each chapter could potentially be expanded into its own book. In fact, books have been written around some of these chapters and Harrison kindly refers to them in the bibliography. Therefore, this book is not the last word on SQL tuning, optimizer internals, Oracle troubleshooting, the SGA, or latch contention. However, its coherent approach, useful summaries and highlights, and efficient organization, make *Oracle Performance Survival Guide—A Systematic Approach to Database Optimization* a valuable and essential guide to anyone wishing to expand and deepen their Oracle performance skill set. ▲

"Its coherent approach, useful summaries and highlights, and efficient organization, make [Harrison's book] a valuable and essential guide to anyone wishing to expand and deepen their Oracle performance skill set."

Links

Author's website and blog: **www.guyharrison.net**

Main book website, including scripts and tools: **www.informit.com/store/product.aspx?isbn=0137011954**

Dave Abercrombie has worked at Convio (with a "v," not an "f") for about ten years, having helped to found GetActive Software before its merger with Convio. This company's business model is rather like a distributed denial of service attack against itself. Its customers are nonprofit membership organizations who want to use the Web to engage and activate their members. So each day, Convio sends tens of millions of emails to these members, and then tracks the ensuing member transactions and activities, such as donations, advocacy, and click-throughs. Dave has honed his troubleshooting and scalability skills by keeping these very busy databases happy. He has presented at Hotsos and is becoming a regular presenter at NoCOUG. He can be reached at **dabercrombie@convio.com**.

Oracle Performance Tuning: A Methodical Approach

An excerpt from *Oracle Performance Survival Guide* by Guy Harrison

This chapter is an excerpt from the book Oracle Performance Survival Guide *authored by Guy Harrison and published by Prentice Hall Professional, Oct. 2009, ISBN 0137011954;*

Oracle performance tuning has come a long way over the years, but it is too often still approached in a haphazard or inefficient manner. Consider the following cautionary tale:

A mission-critical application system is experiencing unsatisfactory performance. As an experienced Oracle performance specialist, you are called in to diagnose the problem. The first thing you do is examine the database wait times to see where the database is spending the majority of execution time. As we'll see later, this information can easily be found by looking in the V$SYSTEM_EVENT and V$SYS_TIME_MODEL views.

Looking at these views, two things stand out. First, the vast majority of database time is spent reading from disk devices. Second, the average time to read a single block from disk is much higher than you would expect given the capabilities of the disk hardware involved.

You suspect that the disk array might have insufficient IO bandwidth to support the application's demands. In other words, not enough physical disks are in the disk array to support the IO rate required. After a quick calculation, you recommend increasing the number of disk devices in the array by a factor of four. The dollar cost is substantial, as is the downtime required to redistribute data across the new disks within the array.[1] Nevertheless, something needs to be done, so management approves the expense and the downtime. Following the implementation, users report they are satisfied with performance, and you modestly take all the credit.

A successful outcome? You think so, until . . .

- Within a few months performance is again a problem and disk IO is again the culprit.
- Another Oracle performance expert is called in and reports that a single indexing change would have fixed the original problem with no dollar cost and no downtime.
- The new index is implemented, following which the IO rate is reduced to one-tenth of that observed during your original engagement. Management prepares to sell the now-surplus disk devices on eBay and marks your consulting record with a "do not reengage" stamp.
- Your significant other leaves you for an Oracle salesperson, and you end up shaving your head and becoming a monk.

After years of silent meditation, you realize that while your tuning efforts correctly focused on the activities consuming the most time within the database, they failed to differentiate between *causes* and *effects*. Consequently, you mistakenly dealt with an *effect*—the high disk IO rate—while neglecting the *cause* (a missing index).

In this chapter we consider a methodology that ensures that you focus on the root causes of Oracle performance problems. This approach avoids the repetitive trial-and-error process that is characteristic of a lot of performance-tuning efforts and ensures that you get the biggest performance gains for your tuning efforts.

A Brief History of Oracle Performance Tuning

In the early '90s, the discipline of tuning an Oracle server was nowhere near as well established as today. In fact, performance tuning was mostly limited to a couple of well-known "rules of thumb."

The most notorious of these guidelines was that you should tune the *Buffer Cache Hit Ratio*: the ratio that describes the proportion of blocks of data requested by a SQL that are found in memory. If ten blocks of data are requested, and nine of them are found in memory, the hit ratio is 90 percent. Increasing the buffer cache size until the ratio reached 90 percent to 95 percent was often suggested. Similar target values were suggested for other ratios, such as the latch hit ratio.

The problem with these "ratio-based" techniques was that although the ratios usually reflected some measure of internal Oracle efficiency, they were often only loosely associated with the performance experienced by an application using the database. For example, although it is obviously better for a block of data to be found in memory—resulting in a high hit rate—SQL statements that inefficiently read the same data over and over again would often result in a high hit rate. Indeed, a *very* high hit ratio is often a symptom of badly tuned SQL.

The emergence of wait information in Oracle version 7.1 provided an alternative method of approaching tuning. This wait information showed the amount of time Oracle sessions spent waiting for various events, such as a lock becoming available or a disk IO completing. By concentrating on the wait

[1] With some technologies, this downtime can be avoided; however, an extended period of degraded performance would still be required.

events that accounted for the greatest amount of total wait time, Oracle performance tuners could target their tuning efforts more effectively.

Pioneers of systematic Oracle performance tuning, such as Anjo Kolk, author of the famous "Yet Another Performance Profiling" (YAPP) methodology, promoted this technique vigorously.

Wait-based tuning took a surprisingly long time to reach the mainstream: 5–10 years passed between the original release of the wait information and widespread acceptance of the technique; however, today almost all Oracle professionals are familiar with wait-based tuning.

Moving Beyond a Symptomatic Approach

The shift from ratio-based to wait-based tuning has resulted in radical improvements in our ability to diagnose and tune Oracle-based applications. However, as noted previously, simplistically focusing on the largest component of response time can have several undesirable consequences:

- We might treat the symptoms rather than the causes of poor performance.
- We might be tempted to seek hardware-based solutions when configuration or application changes would be more cost-effective.
- We might deal with today's pain but fail to achieve a permanent or scalable solution.

To avoid the pitfalls of a narrow wait-based analysis, we need our tuning activities to follow a number of well-defined stages. These stages are dictated by the reality of how applications, databases, and operating systems interact. At a high level, database processing occurs in *layers*, as follows:

1. Applications send requests to the database in the form of SQL statements (including PL/SQL requests). The database responds to these requests with return codes and result sets.
2. To deal with an application request, the database must parse the SQL and perform various overhead operations (security, scheduling, and transaction management) before finally executing the SQL. These operations use operating system resources (CPU and memory) and might be subject to contention between concurrently executing database sessions.
3. Eventually, the database request needs to process (create, read, or change) some of the data in the database. The exact amount of data that needs to be processed can vary depending on the database design (indexing, for example) and the application (wording of the SQL, for example). Some of the required data will be in memory. The chance that a block will be in memory will be determined mainly by the frequency with which the data is requested and the amount of memory available to cache the data. When we access database data in memory, it's called a logical IO. Memory is also used to perform sorting and hashing operations.
4. If the block is not in memory, it must be accessed from disk, resulting in real physical IO. Physical IO is by far the most expensive of all operations, and consequently the database goes to a lot of effort to avoid performing unnecessary IO operations. However, some disk activity is inevitable. Disk IO also occurs when sorting and hashing operations are too large to complete in memory.

Activity in each of these layers influences the demand placed on the subsequent layer. For instance, if an SQL statement is submitted that somehow fails to exploit an index, it will require an excessive number of logical reads, which in turn will increase contention and eventually involve a lot of physical IO. It's tempting when you see a lot of IO or contention to deal with the symptom directly by tuning the disk layout. However, if you sequence your tuning efforts so as to work through the layers in order, you have a much better chance of fixing root causes and relieving performance at lower layers.

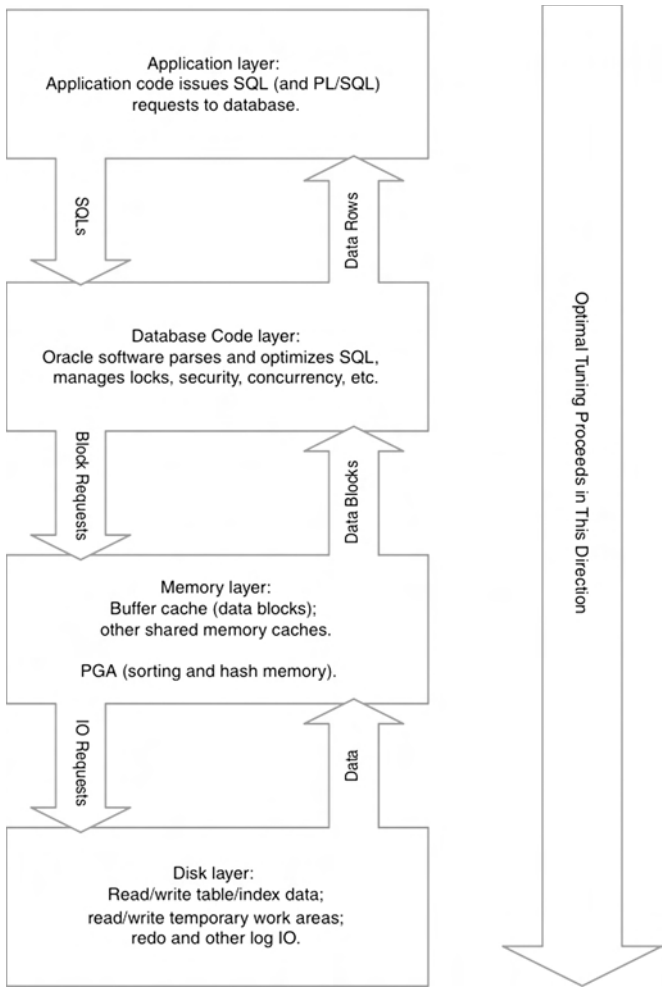


Figure 1–1 *The four major "layers" of the Oracle database.*

Here's the tuning by layers approach in a nutshell:

Problems in one database layer can be caused or cured by configuration in the higher layer. The logical steps in Oracle tuning are therefore:

1. **Reduce application demand to its logical minimum by tuning SQL and PL/SQL and optimizing physical design (partitioning, indexing, and so on).**

2. **Maximize concurrency by minimizing contention for locks, latches, buffers, and other resources in the Oracle code layer.**
3. **Having normalized logical IO demand by the preceding steps, minimize the resulting physical IO by optimizing Oracle memory.**
4. **Now that the physical IO demand is realistic, configure the IO subsystem to meet that demand by providing adequate IO bandwidth and evenly distributing the resulting load.**

The tuning procedures in this book are organized according to the tuning by layers approach.[2] In the remainder of this chapter, we will examine each of these steps, as shown in Figure 1–1, in turn.

State 1: Minimizing the Application Workload

Our first objective is to minimize the application's demands on the database. We want the database to satisfy the application's data requirements with the least possible processing. In other words, we want Oracle to work smarter, not harder.

Broadly speaking, we use two main techniques to reduce application workload:

- ***Tuning the application code***—This might involve changing application code—C#, Ruby or Java—so that it issues fewer requests to the database (by using a client-side cache, for instance). However, more often this will involve rewriting application SQL and/or PL/SQL.
- ***Modifying the physical implementation of the application's data***—This might involve indexing, denormalization, or partitioning.

Chapters 4 through 14 cover in detail the various techniques we can use to minimize application workload. Specifically:

- ***Structuring an application to avoid overloading the database***—Applications can avoid making needless requests of the database and can be architected to minimize lock and other contention.
- ***Using best practices when communicating with the Oracle server***—The programs that communicate with Oracle can be designed and implemented to minimize database round trips and unnecessary requests.
- ***Optimizing the physical database design***—This includes indexing, denormalization, partitioning, and other ways of physically structuring data to reduce the work required to execute SQL requests.
- ***Optimizing the Oracle query optimizer***—By correctly configuring the collection of optimizer statistics, overriding optimizer plans when necessary, and instituting ongoing monitoring of SQL performance.
- ***Tuning the performance of individual SQL statements***—This might involve changing the SQL execution plan using hints, stored outlines, profiles, and SQL rewrites.
- ***Using parallel SQL capabilities***—This allows you to apply multiple processes to the SQL execution.
- ***Tuning and using PL/SQL programs***—You can use PL/SQL used in certain circumstances to improve application performance, and PL/SQL programs present unique tuning issues and opportunities.

These techniques not only represent the logical place to start in our tuning efforts, but they also represent the techniques that provide the most dramatic performance improvements. It's not at all uncommon for SQL tuning to result in performance improvements of 100 or even 1,000 times: improvements that you rarely see when tuning contention, optimizing memory, or adjusting physical disk layout.

Stage 2: Reducing Contention and Bottlenecks

After we adjust the application workload demand to a sensible minimum, we are ready to tackle contention within the Oracle server. Contention occurs when two or more sessions want simultaneous access to a resource, such as a lock or memory buffer.

When the application demand hits the database, contention—the proverbial "bottleneck"—limits the amount of work that can be done. From the applications perspective, the database appears slow or stalled. At lower levels—the disk subsystem, for instance—the demand appears to be lower than it really is. The contention bottleneck prevents the demand from getting through the database code into the IO subsystem. Figure 1–2 illustrates the phenomenon.

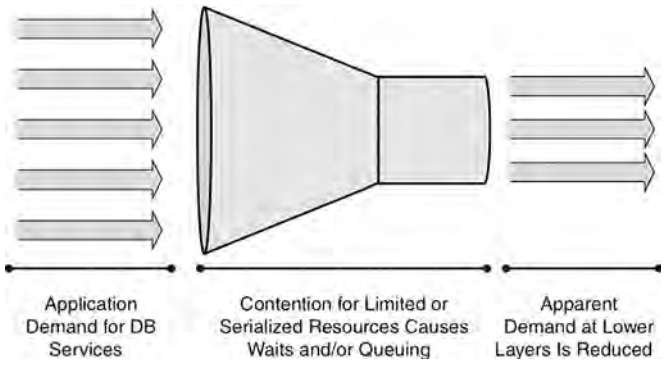


Figure 1–2 *Contention is the proverbial bottleneck.*

The two most prevalent forms of contention observed in Oracle-based applications are contention for rows within tables—generally showing up as waits for locks—and contention for areas of shared memory—showing up as waits for latches, memory buffers, and so on.

Lock contention is largely a factor of application design: Oracle's locking model allows for high concurrency because readers never wait for locks, writers never wait for readers, and locks are applied at the row level only. Typically, lock contention is caused by an application design that involves high simultaneous updates against a single row or in which locks are held for an excessive length of time, perhaps due to a pessimistic locking model. This sort of contention is almost impossible to eliminate without application logic

[2] The general concept of "tuning by layers" for Oracle was first proposed by Steve Adams (http://www.ixora.com.au/tips/layers.zip).

changes—changes that we address in the first stage of tuning. However, there are scenarios in which excessive lock contention is caused by database or schema configuration problems, or by Oracle internal mechanisms.

Contention for shared memory occurs when sessions want to read or write to shared memory in the SGA concurrently. All shared memory is protected by *latches* (or *mutexes*), which are similar to locks except that they prevent concurrent access to data in shared memory rather than data in tables. If a session needs to modify some data in memory, it acquires the relevant latch, or mutex, and if another session wants to read or modify the same data, a latch, or mutex, wait might occur. Contention for data blocks in the buffer cache can occur for other reasons as well: A variety of *buffer* waits can occur when a block of memory is unavailable due to conflicting processing demands between sessions.

Chapters 15–17 address the techniques for eliminating Oracle contention. Specifically:

- Detecting and dealing with lock contention, including Oracle internal locks
- Optimizing Oracle latching mechanisms that protect Oracle shared memory
- Identifying and correcting contention for shared memory itself

Stage 3: Reducing Physical IO

Now that the application demand has been minimized, and contention that might otherwise mask that demand eliminated, we turn our attention to reducing the time spent waiting for IO. In other words, before trying to reduce the time taken for *each* IO (IO latency), we try to reduce the amount of IO. As it turns out, reducing the *amount* of IO almost always reduces the IO latency, so attacking the volume of IO first is doubly effective. Having reduced application demand through SQL tuning and other means, we now try to further reduce IO by configuring memory to cache and buffer IO requests.

Most physical IO in an Oracle database occurs either because an application session requests data to satisfy a query or data modification request, because the session must sort or hash data, or must create a temporary segment to support a large join, ORDER BY, or similar operation.

Oracle's shared memory (the SGA) stores copies of data blocks in memory and eliminates the need to perform a disk IO if the requested data block is in that memory. Correctly allocating this memory goes a long way toward minimizing disk IO.

In the past, allocating SGA memory was a hit-and-miss affair. Luckily, in modern Oracle the server can automatically adjust memory allocations for you, or you can measure the effect of adjusting the size of the various memory pools by examining *advisories*, which accurately predict the effect of changing the sizes of those pools.

Oracle enables you to set up separate memory areas to cache blocks of different size and also enables you to nominate specific areas to cache data that might need to be kept in memory. Not all these memory areas will be automatically and dynamically resized by Oracle, and Oracle will not automatically allocate the areas in the first place, or assign specific data segments to these areas; those tasks are left to the DBA.

In addition to disk reads when accessing data not in the shared memory, Oracle might perform substantial IO when sorting or hashing data during ordering, grouping, or join operations. Where possible, Oracle performs a sort or hash operation in memory within the area of memory allocated for program use—the Program Global Area (PGA). However, if sufficient memory is not available, Oracle writes to—and reads from—temporary segments on disk to complete the sort or hash operation.

Oracle has improved its capability to automatically manage these memory areas in most recent releases. As of 10g, Oracle automatically resizes allocations *within* the PGA and the SGA, but will not shift memory *between* these areas. In 11g Oracle can move memory between the PGA and SGA as required—or at least, as Oracle calculates is required.

Despite the progress in automatic memory management, there's still a lot for the Oracle administrator to do to ensure optimal performance. These tasks include:

- Determining whether the correct amount of OS memory is available to Oracle
- Determining the correct division of memory between the PGA and SGA, or—in 11g—allowing Oracle's Automatic Memory Management to make that determination
- Fine-tuning the allocation of segments to specific memory areas
- Fine-tuning the parameters controlling sorting and joining
- Monitoring Oracle's memory allocations and over-riding if necessary
- Chapters 18–20 cover these memory optimization techniques.

Stage 4: Optimizing Disk IO

At this point, we've normalized the application workload—in particular the amount of logical IO demanded by the application. We've eliminated contention that might be blocking—and therefore masking—those logical IO requests. Finally, we've configured available memory to minimize the amount of logical IO that ends up causing physical IO. Now—and only now—it makes sense to make sure that our disk IO subsystem is up to the challenge.

To be sure, optimizing disk IO subsystems can be a complex and specialized task, but the basic principles are straightforward:

- Ensure the IO subsystem has enough bandwidth to cope with the physical IO demand. This is primarily determined by the number of distinct disk devices you have allocated. Disks vary in performance, but the average disk device might be able to perform approximately 100 random IOs per second before becoming saturated. Good response time usually requires that the disk be less than 100 percent utilized—say 50 percent to 75 percent. For most databases, meeting IO requirements means acquiring much more disk than simple storage require-

ments dictate. You need to acquire enough disks to sustain your IO rate with acceptable response time, not just enough disks to store all your data.

- Spread your load evenly across the disks you have allocated. The best way to do this is RAID 0 (Striping). The worst way—for most databases—is RAID 5, which incurs a heavy penalty on write IO.

The obvious symptom of an overly stressed IO subsystem is excessive delays responding to IO requests. The expected delay—called *service time*—varies from disk to disk, but even on the slowest disks should not exceed approximately 10ms. Disk arrays boasting large memory caches and Solid State Disk (SSD) devices might provide much lower latencies. Network Attached Storage (NAS) devices might also have a high network-related component to the service time.

Spreading the load across spindles is best done by hardware or software striping. Oracle's ASM technology provides a simple and universally available method of doing this for ordinary disk devices. Alternating datafiles across multiple disks is usually less effective, though still better than no striping at all. Most high-end databases employ the striping capabilities of a hardware disk array.

For most databases, optimizing the datafiles for read activity makes the most sense because Oracle sessions do not normally wait for datafile writes; the database writer process (DBWR) writes to disk asynchronously. However, if the DBWR cannot keep up with database activity, sessions need to wait for the DBWR to catch up. Likewise, we need to ensure that the flashback and redo log writer processes can keep up; otherwise, user sessions need to wait for these processes as well.

Chapters 21 and 22 covers the issues associated with optimizing disk IO:

- Understanding the Oracle IO mechanisms—buffered IO and direct IO, redo and archive log IO, flashback IO, and other mechanisms.
- Measuring IO performance and calculating optimal disk configurations.
- Using mechanisms for striping data, including RAID levels.
- Utilizing specific IO-related technologies such as ASM and SSD.

Summary

When faced with an obviously IO-bound database, it's tempting to deal with the most obvious symptom—the IO subsystem—immediately. Unfortunately, this usually results in treating the symptom rather than the cause, is often expensive, and is often ultimately futile. Because problems in one database layer can be caused or cured by configuration in the higher layer, the most efficient and effective way to optimize an Oracle database is to tune higher layers before tuning lower layers:

1. Reduce application demand to its logical minimum by tuning SQL, optimizing physical design (partitioning, indexing), and tuning PL/SQL.
2. Maximize concurrency by minimizing contention for locks, latches, buffers, and other resources in the Oracle code layer.
3. Having normalized logical IO demand by the preceding steps, minimize the resulting physical IO by optimizing Oracle memory.
4. Now that the physical IO demand is realistic, configure the IO subsystem to meet that demand by providing adequate bandwidth and evenly distributing the resulting load. ▲

It's Registration Time Again!

I know the year just zoomed by. I'm sure it's due to the great 2009 NoCOUG program. The board has another stellar year planned for you. Please register early to continue getting this award-winning *Journal* and streamline your conference registration. For our 2009 members who have not yet registered for 2010, this issue of the *NoCOUG Journal* is complimentary. We know you plan to register.

Check the address label on the back of the *Journal*. If there is an asterisk (*) affixed to your name, you are already registered for 2010. Keep in mind that the mailing list was pulled on 1/26/10. The (*) status indicator will only be used on this issue.

All 2009 members should have received an email or U.S. mail reminder by now. That will include your NoCOUG ID for online registration at **www.nocoug.org**. Contact me with any questions at **membership@nocoug.org**.

Hope to see you at the coming conferences!

—Joel Rosingana
NoCOUG Membership Director

Not the SQL of My Kindergarten Days

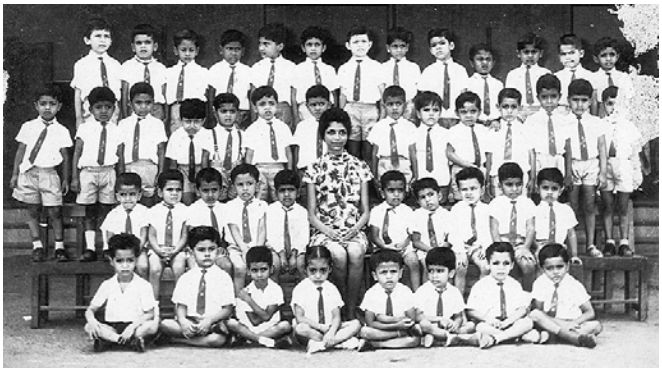
Iggy Fernandez

by Iggy Fernandez

I remember, I remember
The fir-trees dark and high;
I used to think their slender tops
Were close against the sky:
It was a childish ignorance,
But now 'tis little joy
To know I'm farther off from heaven
Than when I was a boy.

—Thomas Hood [1799–1845]

The relational model was invented by IBM researcher Edgar Codd when I was in kindergarten. Here's a picture of my kindergarten class; one of those cute little tykes is me.



Codd made proposals for "data base sublanguages" in his paper *Relational Completeness of Data Base Sublanguages* (1972). As IBM researcher Donald Chamberlin recalled later: *[Codd] gave a seminar and a lot of us went to listen to him. This was as I say a revelation for me because Codd had a bunch of queries that were fairly complicated queries and since I'd been studying CODASYL, I could imagine how those queries would have been represented in CODASYL by programs that were five pages long that would navigate through this labyrinth of pointers and stuff. Codd would sort of write them down as one-liners. These would be queries like, "Find the employees who earn more than their managers." He just whacked them out and you could sort of read them, and they weren't complicated at all, and I said, "Wow." This was kind of a conversion experience for me, that I understood what the relational thing was about after that.*

Donald Chamberlin and fellow IBM researcher Raymond Boyce went on to implement the first "data base sublanguage" based on Codd's proposals and described it in a short paper titled *SEQUEL: A Structured English Query Language* (1974). The acronym SEQUEL was later shortened to SQL because SEQUEL was a trademarked name; this means that the correct pronunciation of SQL is sequel not es-cue-el. Codd's paper and Chamberlin's paper can be downloaded from the Internet; they were written using manual typewriters, and Codd's paper even contains handwritten corrections.

The SQL of today is much more powerful than the SQL of my kindergarten days. The latest ANSI SQL standard comprises thousands of pages and the one-liners of Codd's day have given way to hundred-liners that solve extremely complex problems. In the example that follows, we demonstrate several powerful features of SQL, including common table expressions, scalar subqueries, pivoting, recursive common table expressions, outer joins, and analytic functions. Our assignment is to create a database load profile in time series format. A typical STATSPACK report only lists a point-in-time snapshot of such a database load profile, and it would therefore be useful to review the history of each component, such as logical reads or physical reads.

Load Profile	Per Second	Per Transaction
Redo size	901,736.28	20,043.04
Logical reads	247,983.21	5,511.96
Block changes	5,178.77	115.11
Physical reads	2,282.71	50.74
Physical writes	732.79	16.29
User calls	2,491.71	55.38
Parses	2,255.89	50.14
Hard parses	26.79	0.60
Sorts	1,759.71	39.11
Logons	3.32	0.07
Executes	12,469.57	277.16
Transactions	4.99	

The data for the problem is available in the stats$snapshot table (snap_id, snap_time, startup_time) and the stats$sysstat table (snap_id, name, value), both of which are part of the STATSPACK schema. The data values in stats$sysstat increase monotonically for the life of the database and start again from zero every time the database is restarted. Here is a sample of some raw data.

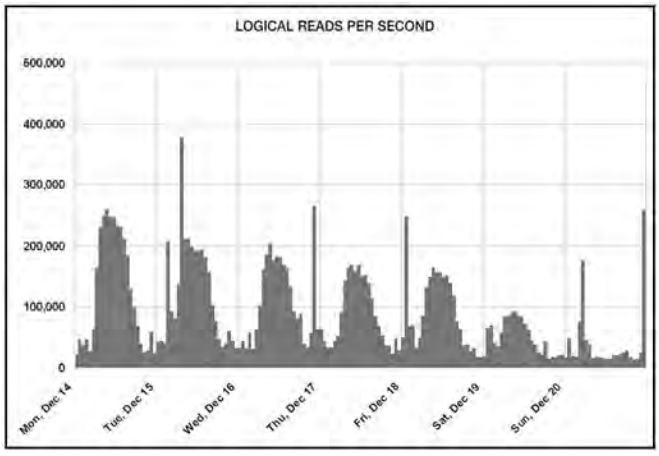
Snap Id	Snap Time	Startup Time
41566	12/14/2009 08:53	11/3/2009 22:48
41576	12/14/2009 09:23	11/3/2009 22:48
41586	12/14/2009 09:53	11/3/2009 22:48

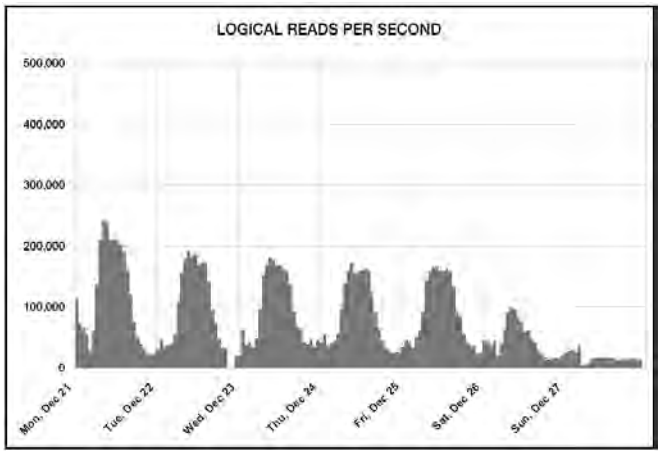
Snap Id	Name	Value
41566	session logical reads	4,272,750,011
41576	session logical reads	4,711,935,207
41586	session logical reads	5,158,057,003
41566	physical reads	57,151,567
41576	physical reads	61,038,074
41586	physical reads	65,144,663
41566	user commits	1,006,064
41576	user commits	1,082,359
41586	user commits	1,159,213
41566	user rollbacks	54,483
41576	user rollbacks	58,524
41586	user rollbacks	62,607

What we now need is the *time series representation* of the above data. Here is an example showing just a few of the columns. We used interpolation techniques to produce exactly one data point per hour. Also, we properly handled situations in which the database was restarted and the values in the database load profile were reset to zero.

Timestamp	Logical Reads Per Second	Physical Reads Per Second	Transactions Per Second
12/14/2009 09:00	258,450.28	2,047.59	46.68
12/14/2009 10:00	247,301.23	2,254.32	45.07
12/14/2009 11:00	246,408.77	2,293.54	45.35

From the time series data above, we can then generate nice graphs using our charting tool of choice such as Excel or RRDtool. Here are a couple of examples:





The first nifty SQL feature displayed in our solution is the "common table expression" (CTE). A CTE is similar to an "inline view" but offers several advantages. First, it divides the code into manageable pieces; long SQL statements that don't use common table expressions are very difficult to debug and maintain. There is also a performance consideration; the data of the CTE can be used multiple times within a SQL statement, but Oracle will not need to repeatedly recompute the CTE. Here is a list of the common table expressions in our solution; their names indicate their purpose and help the reader understand how the query progresses toward the final result.

- constants
- snaphots
- sysstat
- pivoted_sysstat
- numbers
- interpolation_formulas
- interpolated_sysstat

The other nifty feature seen below is "scalar subqueries"; a SQL query returning a single value can be placed wherever a "scalar"—a single value—is required.

```
WITH

constants AS

(

  SELECT

    (
      SELECT min(snap_id)
      FROM stats$snapshot
      WHERE snap_time >= trunc(to_date('&&begin_date')) - 1
    )
      AS begin_snap_id,

    (
      SELECT max(snap_id)
      FROM stats$snapshot
      WHERE snap_time <= trunc(to_date('&&begin_date')) + 8
    )
      AS end_snap_id

  FROM dual

),
```

Here are the results of the above code segment:

Begin Snap Id	End Snap Id
41323	44337

We next identify a subset of data from the stats$snapshot table. Scalar subqueries are once again on display. Notice how this CTE refers to the previous one.

```
snapshots AS

(

  SELECT
    snap_id,
    snap_time,
    startup_time
  FROM stats$snapshot

  WHERE dbid = &&dbid
  AND instance_number = &&instance_number
  AND snap_id >= (SELECT begin_snap_id FROM constants)
  AND snap_id <= (SELECT end_snap_id FROM constants)

),
```

Here are the results—with some rows omitted—of the above code segment:

Snap ID	Snap Time	Startup Time
41566	12/14/2009 08:53	11/3/2009 22:48
41576	12/14/2009 09:23	11/3/2009 22:48
41586	12/14/2009 09:53	11/3/2009 22:48

We next identify a subset of data from the stats$sysstat table. Scalar subqueries are once again on display.

```
sysstat AS

(

  SELECT
    snap_id,
    name,
    value AS value
  FROM stats$sysstat

  WHERE dbid = &&dbid
  AND instance_number = &&instance_number
  AND snap_id >= (SELECT begin_snap_id FROM constants)
  AND snap_id <= (SELECT end_snap_id FROM constants)
  AND name IN
  (
    'session logical reads',
    'physical reads',
    'user rollbacks',
    'user commits'
  )

),
```

Here are the results—with some rows omitted—of the above code segment:

Snap Id	Name	Value
41566	physical reads	57,151,567
41566	session logical reads	4,272,750,011
41566	user commits	1,006,064
41566	user rollbacks	54,483
41576	physical reads	61,038,074
41576	session logical reads	4,711,935,207
41576	user commits	1,082,359
41576	user rollbacks	58,524
41586	physical reads	65,144,663
41586	session logical reads	5,158,057,003
41586	user commits	1,159,213
41586	user rollbacks	62,607

The next SQL feature on display is the PIVOT operator, which was introduced in Oracle Database 11*g*R1. Pivoting is well known to Excel power users; it converts rows of data into a two-dimensional matrix. More information on the PIVOT operator and its sister operator, UNPIVOT, can be found in Arup Nanda's article *Oracle Database 11g: The Top New Features for DBAs and Developers*.

```
pivoted_sysstat AS

(

  SELECT
    snap_id,
    logical_reads AS logical_reads,
    physical_reads AS physical_reads,
    user_rollbacks + user_commits AS transactions
  FROM
    (
      SELECT *
      FROM sysstat
    )
    PIVOT
    (
      SUM(value)
      FOR NAME IN
      (
        'session logical reads' AS logical_reads,
        'physical reads' AS physical_reads,
        'user rollbacks' AS user_rollbacks,
        'user commits' AS user_commits
      )
    )

),
```

Here are the results—with some rows omitted—of the above code segment:

Snap ID	Logical Reads	Physical Reads	Transactions
41566	4,272,750,011	57,151,567	1,060,547
41576	4,711,935,207	61,038,074	1,140,883
41586	5,158,057,003	65,144,663	1,221,820

Here is the Oracle Database 10*g* version of the above code. It requires a very unintuitive use of the DECODE function.

```
pivoted_sysstat AS

(

  SELECT

    snap_id,

    sum(decode(name,'session logical reads',value,0))
      AS logical_reads,

    sum(decode(name,'physical reads',value,0))
      AS physical_reads,

    sum(decode(name,'user rollbacks',value,'user commits',value,0))
```

```
    AS transactions

  FROM sysstat
  GROUP BY snap_id

),
```

Next we have a simple example of a "recursive common table expression." Oracle has provided recursive functionality using CONNECT BY for a long time, but recursive common table expressions are new in Oracle Database 11*g*R2 and can handle problems that CONNECT BY could not. A recursive CTE takes the form of one or more "anchor members" plus a recursive member that invokes the CTE repeatedly until a terminating condition is encountered. A good explanation of recursive common table expressions can be found in an article by Jonathan Gennick titled *Understanding the WITH Clause*.

```
numbers(n) AS

(
  SELECT 1
  FROM dual

  UNION ALL

  SELECT n + 1
  FROM numbers
  WHERE n < 168
),
```

Here are the results—with some rows omitted—of the above code segment:

N
1
2
3
4
5

For contrast, here is the Oracle Database 10*g* version using the CONNECT BY clause.

```
numbers AS

(
  SELECT level AS n
  FROM dual
  CONNECT BY level <= 169
),
```

The next code section is fairly long and illustrates three interesting features. The CASE expression is a great advancement over the DECODE function and allows the evaluation of complex Boolean expressions. "Windowing functions"—a subclass of analytic functions—allow the evaluation of data contained in specified rows other than the current row. Finally, SQL now offers full support for outer joins of all flavors, including LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN; left outer joins are used below.

```
interpolation_formulas AS

(

  SELECT

    temp.timestamp,

    CASE
      WHEN s1.startup_time = s2.startup_time
      THEN s1.startup_time
      ELSE NULL
    END AS startup_time,

    CASE
      WHEN s1.startup_time = s2.startup_time
      THEN s2.snap_id
      ELSE NULL
    END AS next_snap_id,

    CASE
      WHEN s1.startup_time = s2.startup_time
      THEN (timestamp - s1.snap_time) / (s2.snap_time - s1.snap_time)
      ELSE NULL
    END AS fraction

  FROM

    (

      SELECT

        snap_id,
        snap_time AS timestamp,
        LAST_VALUE(snap_id IGNORE NULLS)
          OVER (ORDER BY snap_time ROWS BETWEEN UNBOUNDED
PRECEDING AND 1 PRECEDING)
          AS previous_snap_id,
        FIRST_VALUE(snap_id IGNORE NULLS)
          OVER (ORDER BY snap_time ROWS BETWEEN 1 FOLLOWING
AND UNBOUNDED FOLLOWING)
          AS next_snap_id

      FROM

        (
          SELECT snap_id, snap_time FROM snapshots
          UNION ALL
          SELECT NULL, trunc(to_date('&&begin_date')) + (n - 2) * 1/24 FROM
numbers
        )

    ) temp

    LEFT OUTER JOIN snapshots s1
    ON (temp.previous_snap_id = s1.snap_id)

    LEFT OUTER JOIN snapshots s2
    ON (temp.next_snap_id = S2.snap_id)

  WHERE temp.snap_id IS NULL

),
```

Here are the results—with some rows omitted—of the above code segment:

Timestamp	Startup Time	Previous Snap ID	Next Snap ID	Fraction
12/14/2009 09:00	11/3/2009 22:48	41566	41576	0.230940456
12/14/2009 10:00	11/3/2009 22:48	41586	41596	0.232648529
12/14/2009 11:00	11/3/2009 22:48	41606	41616	0.232686981

We're now ready to perform some interpolation magic. LEFT OUTER JOIN is once again on display in the following code section.

```
interpolated_sysstat AS
(
  SELECT
    timestamp,
    startup_time,
      ps1.logical_reads
    + if.fraction * (ps2.logical_reads - ps1.logical_reads)
      AS logical_reads,
      ps1.physical_reads
    + if.fraction * (ps2.physical_reads - ps1.physical_reads)
      AS physical_reads,
      ps1.transactions
    + if.fraction * (ps2.transactions - ps1.transactions)
      AS transactions
  FROM
    interpolation_formulas if
    LEFT OUTER JOIN pivoted_sysstat ps1
    ON (if.previous_snap_id = ps1.snap_id)
    LEFT OUTER JOIN pivoted_sysstat ps2
    ON (if.next_snap_id = ps2.snap_id)
),
```

Here are the results—with some rows omitted—of the above code segment:

Timestamp	Startup Time	Logical Reads	Physical Reads	Transactions
12/14/2009 09:00	11/3/2009 22:48	4,374,175,640.57	58,049,118.70	1,079,099.83
12/14/2009 10:00	11/3/2009 22:48	5,264,460,051.52	66,164,677.67	1,241,347.82
12/14/2009 11:00	11/3/2009 22:48	6,151,531,630.47	74,421,410.14	1,404,615.90

The next section uses the LAG analytic function to compute the difference between data values in adjacent rows.

```
delta_sysstat AS
(
  SELECT
    timestamp,
    logical_reads - lag(logical_reads)
      OVER (PARTITION BY startup_time ORDER BY timestamp)
      AS logical_reads,
    physical_reads - lag(physical_reads)
      OVER (PARTITION BY startup_time ORDER BY timestamp)
      AS physical_reads,
    transactions - lag(transactions)
      OVER (PARTITION BY startup_time ORDER BY timestamp)
      AS transactions
  FROM interpolated_sysstat
)
```

Here are the results—with some rows omitted—of the above code segment:

Timestamp	Logical Reads	Physical Reads	Transactions
12/14/2009 09:00	930,421,023.36	7,371,308.57	168,060.61
12/14/2009 10:00	890,284,410.95	8,115,558.97	162,247.99
12/14/2009 11:00	887,071,578.94	8,256,732.47	163,268.08

We've written a lot of code so far but it was always in unmanageable chunks. We're finally ready to display the results of the query; the final section is equally short and sweet.

```
SELECT
  timestamp,
  logical_reads / 3600
    AS logical_reads_per_second,
  physical_reads / 3600
    AS physical_reads_per_second,
  transactions / 3600
    AS transactions_per_second,
  logical_reads / transactions
    AS logical_reads_per_transaction,
  physical_reads / transactions
    AS physical_reads_per_transaction
FROM delta_sysstat
WHERE timestamp >= trunc(to_date('&&begin_date'))
ORDER BY timestamp;
```

Here are the final results with some rows and columns omitted:

Timestamp	Logical Reads Per Second	Physical Reads Per Second	Transactions Per Second
12/14/2009 09:00	258,450.28	2,047.59	46.68
12/14/2009 10:00	247,301.23	2,254.32	45.07
12/14/2009 11:00	246,408.77	2,293.54	45.35

I hope you enjoyed this little tour-by-example of the newer features of SQL. I didn't have enough space or time to do them any real justice, but you can easily find more information about them on the Internet. You can download all the code in this article from my blog. ▲

Iggy Fernandez is an Oracle DBA with Database Specialists and has more than ten years of experience in Oracle database administration. He is the editor of the quarterly journal of the Northern California Oracle Users Group (NoCOUG) and the author of Beginning Oracle Database 11g Administration (Apress, 2009). He blogs at **iggyfernandez.wordpress.com**.

Many Thanks to Our Sponsors

NoCOUG would like to acknowledge and thank our generous sponsors for their contributions. Without this sponsorship, it would not be possible to present regular events while offering low-cost memberships. If your company is able to offer sponsorship at any level, please contact NoCOUG's president, Hanan Hit, at **hithanan@gmail.com**. ▲

Long-term event sponsorship:

Chevron

Oracle Corp.

Thank you! Year 2010 Gold Vendors:

- Confio Software
- Database Specialists, Inc.
- Enteros
- Precise Software Solutions

For information about our Gold Vendor Program, contact the NoCOUG vendor coordinator via email at: **vendor_coordinator@nocoug.org**.

TREASURER'S REPORT

Naren Nagtode, *Treasurer*

Beginning Balance		
October 1, 2009		**$ 25,374.70**
Revenue		
Membership Dues	4,285.00	
Meeting Fees	750.00	
Vendor Receipts	2,500.00	
Advertising Fee	–	
Training Day	21,550.00	
Sponsorship	–	
Interest	4.86	
Paypal balance	–	
Total Revenue		**$ 29,089.86**
Expenses		
Regional Meeting	5,235.33	
Journal	4,272.47	
Membership	58.14	
Administration	1,350.05	
Website	–	
Board Meeting	578.20	
Marketing	100.00	
Insurance	523.00	
Vendors	59.20	
Tax	56.44	
Training Day	15,360.36	
Accounting	–	
P.O. Box	92.00	
Total Expenses		**$ 27,685.19**
Ending Balance		
December 31, 2009		**$ 26,779.37**

Architecting for E-Discovery

by John Weathington

John Weathington

Not responding to an e-discovery request is as good as an admission of guilt, and this downfall lies squarely on the shoulders of the information technology organization. In the well-known case of *Zubulake v. UBS Warburg LLC*, UBS could not produce potentially incriminating emails that the plaintiff was requesting, and the courts actually ruled that it was more likely these emails existed than not. This had a damaging effect on their case. Likewise in the *United States v. Phillip Morris USA, Inc.* lawsuit, Phillip Morris was fined $2.75 million for continuing to delete emails after their notice of litigation was issued.

Cases like this have triggered a tidal wave of fear in organizations, and just like the response to Sarbanes-Oxley, organizations have seemed to overreact, overcorrect, and overspend. And like Sarbanes-Oxley, I'm now hearing e-discovery used as a blanket excuse to justify information technology processes and spend that serve no business purpose. Continue down this road, and you won't need to worry about a lawsuit because there will be no company left to sue!

So how do you put these e-discovery concerns to rest for good? Well, you can't. E-discovery is like a reckless teenager: you do the best you can, and then you cross your fingers and hope nothing happens. Your chances of survival, however, are in direct proportion to how strong your foundation is. Your foundation is a good email retention system. Let's start by discussing how to put one of these together.

Email Retention Starts with Record Retention

In my opinion, email retention is really low-hanging fruit for IT, so it surprises me how often I see it done wrong. When done properly it can really prevent potential compliance problems; however when done improperly it can cost your company millions or more, and it all starts with record retention.

Your company absolutely must have a record retention policy. An email is just one of many ways a record can be created. This is important to understand: it's not about the email; it's about what's *in* the email that drives retention consideration.

If your company does not have a record retention policy, I recommend that you have an offsite with all the stakeholders and figure it out immediately. There's no reason why your company can't get through this before a week is through. I can't emphasize the importance enough. If you get caught with a legal problem and you have no email retention policy in place, you're as good as guilty.

Okay, with policy in hand, your next step is to create a policy database to manage and record things. In your policy database, you'll need to capture the following attributes:

- **Policy version and date**: Whenever the policy changes for any reason, a new record needs to go into your database so you can review what the policy was at any time.
- **Document type**: The document type will drive the retention and destruction properties. This is a general term, and it may end up being two or three fields, depending on your company's organization. Examples would be research, projects, and financial or medical records.
- **Retention period**: How long should documents of this type be retained? Once again, it's up to your company to decide how many phases of retention (i.e., onsite, offsite, etc.) your records need to go through.
- **The policy**: Obviously! You should have a scanned image of the policy available in your database in case there's any confusion.

Building the Email Retention System

How you handle your non-email records (i.e., instant messages, typed documents, etc.) is beyond the scope of this article; however be aware that it's just as important as your email retention system. Let's focus for now on building your system for email retention.

A good email retention system does four things:

- Captures every email and stores it in an immutable state
- Indexes the contents of every email so that it can be researched effectively
- Retains every email for exactly the period of time required (as dictated by its document type), then obliterates it and every trace that it existed

(continued on page 26)

- ➤ Has an "in case of emergency" switch that completely disables the obliteration functionality mentioned above

Sounds easy enough, right? Good—don't overcomplicate things. Start with a write-once, read-only database (similar to the old-style CDs). Centralize your email traffic and send everything to this database. The database needs to store every email in two forms. First rasterize the email into an image for permanency, and then hyper-index the contents—like any popular Internet search engine. This handles the first two bullets.

Metadata in the email should convey what document type we're dealing with, which will tell us what the retention period should be. With this information, stamp every single email with a "destroy on" date. On this date, blast this email to pieces unless the "in case of emergency" switch has been activated. Ensure that your email system is airtight and that there are no copies of this email floating around anywhere (e.g., in personal folders). Be very serious about destruction. Having incriminating email available can get you into more trouble than not having it available!

The "in case of emergency" switch is mandatory in case of a litigation hold. This is the trump card of email retention. If your legal department issues a litigation hold, all email traffic must be retained, no matter what, until the litigation hold is lifted.

"E-discovery can turn into an e-nightmare if not handled properly."

Preventive Control for E-Discovery—Get Your IT Organization Under Control

With a good email retention system in place, you now have the foundation set. To finish the job, there are three keys of effectively getting e-discovery under control, the first of which is getting your own act together. If you're not organized, get organized. If you're already organized, stay organized.

Know everything about your data. Know where all your servers are and their purpose. Know what's in every database, and maintain tight data governance control. Understand both your transactional systems and your data warehouses. Know every detail about every transformation that your reporting systems make to rearrange your data.

Know where your data is at all times, from the time it gets created until the time it is destroyed. Know how your data is backed up, where your data is stored, and how long it is stored there.

For the purposes of e-discovery, the key focuses should be email and instant messaging; however, e-discovery should not be the driver. Get everything documented and organized, because this information is vital to a properly running information technology organization.

Don't Do Anything Special for E-Discovery Purposes

Second, coordinate with finance, legal, and other departments to clearly understand what your document retention and destruction policies are, and make sure you do your part to comply. I once led the development of a compliance data warehouse that had a policy of keeping everything online for 11 years—and destroying anything older than that. The destruction is just as much a requirement as the retention.

Do everything you need to do to support your corporate policies (i.e. Sarbanes-Oxley, privacy, etc.) as they pertain to your business function, but don't make special accommodations in your normal business practice purely for e-discovery purposes (there's one exception: see the next section). You cannot anticipate what a potential lawsuit may require from an e-discovery standpoint, and the law does not require you to be clairvoyant.

Hope for the Best, but Plan for the Hold

Third and finally, prepare for a litigation hold. A litigation hold means that things are about to get interesting. When there's even the anticipation of a lawsuit, your legal department will mandate that you stay your information destruction process. Litigation holds override any and all other retention policies. This is a contingency that you absolutely need to plan for and execute flawlessly. You must have the capability of altering your systems so that information is retained longer than usual.

I suggest organizing fire drills with your legal department to accurately assess the capability and effectiveness of your contingency plan. Have legal create a mock lawsuit, and go through the motions as if it were real. Focus first on email and instant messaging; then branch out to other forms of electronically stored information (ESI) like Microsoft Word and Excel documents. The first time through you will invariably find weaknesses in your system; this is normal. Continue executing drills until you know, for sure, that you can react properly when it's the real thing.

E-discovery can turn into an e-nightmare if not handled properly. However, by building a good foundation with an email retention system, running an efficient and lean IT organization, and having a good litigation contingency plan, you can rest in confidence that you've done your diligence in the matter. Start discussions today with your legal department about assessing your capability to support them. ▲

John Weathington is a management consultant who helps companies dramatically improve efficiency and avoid penalties and fines. His San Francisco Bay Area-based company, Excellent Management Systems, Inc., has helped companies all over the world like Sun Microsystems, Cisco, and PayPal (an eBay company). He is a Project Management Professional (PMP) and a Six Sigma Black Belt, as well as an Oracle DBA and Business Intelligence Architect. He recently helped a large technology firm fortify a $100 million government contract with the implementation of a custom compliance data warehouse. If you would like more information, you can access his website at **www.excellentmanagementsystems.com**.

Database Specialists: DBA Pro Service

DBA PRO BENEFITS

- *Cost-effective and flexible extension of your IT team*
- *Proactive database maintenance and quick resolution of problems by Oracle experts*
- *Increased database uptime*
- *Improved database performance*
- *Constant database monitoring with Database Rx*
- *Onsite and offsite flexibility*
- *Reliable support from a stable team of DBAs familiar with your databases*

CUSTOMIZABLE SERVICE PLANS FOR ORACLE SYSTEMS

Keeping your Oracle database systems highly available takes knowledge, skill, and experience. It also takes knowing that each environment is different. From large companies that need additional DBA support and specialized expertise to small companies that don't require a full-time onsite DBA, flexibility is the key. That's why Database Specialists offers a flexible service called DBA Pro. With DBA Pro, we work with you to configure a program that best suits your needs and helps you deal with any Oracle issues that arise. You receive cost-effective basic services for development systems and more comprehensive plans for production and mission-critical Oracle systems.

DBA Pro's mix and match service components

Access to experienced senior Oracle expertise when you need it
We work as an extension of your team to set up and manage your Oracle databases to maintain reliability, scalability, and peak performance. When you become a DBA Pro client, you are assigned a primary and secondary Database Specialists DBA. They'll become intimately familiar with your systems. When you need us, just call our toll-free number or send email for assistance from an experienced DBA during regular business hours. If you need a fuller range of coverage with guaranteed response times, you may choose our 24 x 7 option.

24 x 7 availability with guaranteed response time
For managing mission-critical systems, no service is more valuable than being able to call on a team of experts to solve a database problem quickly and efficiently. You may call in an emergency request for help at any time, knowing your call will be answered by a Database Specialists DBA within a guaranteed response time.

Daily review and recommendations for database care
A Database Specialists DBA will perform a daily review of activity and alerts on your Oracle database. This aids in a proactive approach to managing your database systems. After each review, you receive personalized recommendations, comments, and action items via email. This information is stored in the Database Rx Performance Portal for future reference.

Monthly review and report
Looking at trends and focusing on performance, availability, and stability are critical over time. Each month, a Database Specialists DBA will review activity and alerts on your Oracle database and prepare a comprehensive report for you.

Proactive maintenance
When you want Database Specialists to handle ongoing proactive maintenance, we can automatically access your database remotely and address issues directly — if the maintenance procedure is one you have pre-authorized us to perform. You can rest assured knowing your Oracle systems are in good hands.

Onsite and offsite flexibility
You may choose to have Database Specialists consultants work onsite so they can work closely with your own DBA staff, or you may bring us onsite only for specific projects. Or you may choose to save money on travel time and infrastructure setup by having work done remotely. With DBA Pro we provide the most appropriate service program for you.

Database Specialists

CALL 1-888-648-0500 TO DISCUSS A SERVICE PLAN

NoCOUG
P.O. Box 3282
Danville, CA 94526

NoCOUG Winter Conference Schedule

Thursday, February 11, 2010, at CarrAmerica Conference Center, Pleasanton, CA

Please visit **www.nocoug.org** for updates and directions, and to submit your RSVP.
Cost: $50 admission fee for non-members. Members free. Includes lunch voucher.

8:00 a.m.–9:00	Registration and Continental Breakfast—Refreshments served
9:00–9:30	**Welcome:** Hanan Hit, NoCOUG president
9:30–10:30	**Keynote:** *Why Are There No Giants?*—Neil Gunther
10:30–11:00	**Break**
11:00–12:00	**Parallel Sessions #1**
	Auditorium: *Performance Analysis For Those Who Can't Wait*—Neil Gunther
	Tassajara: *make.sql: Script Management for Oracle*—Ahbaid Gaffoor, Amazon
	Diablo: *Cloud Computing Enabling Global Business Services Utilities*—Bradley Brown, TUSC
12:00–1:00 p.m.	**Lunch**
1:00–2:00	**Parallel Sessions #2**
	Auditorium: *Top Five Tips for Reducing Storage Cost While Improving Performance* —Jean-Pierre Dijcks, Oracle Corporation
	Tassajara: *Case Study on the Importance of Reaching Recent Rows First*—Dan Tow
	Diablo: *Building a User Definable and Flexible UI with Apex—A Case Study* **Editor's Pick** —Bradley Brown, TUSC
2:00–2:30	**Break and Refreshments**
2:30–3:30	**Parallel Sessions #3**
	Auditorium: *Best Practices for Data Loading with Oracle Database 11g Release 2* —Maria Colgan, Oracle Corporation
	Tassajara: *Measuring for Robust Performance*—Robyn Sands, Cisco Systems
	Diablo: *The Challenge of Virtualizing Databases*—Jedidiah Yueh, Delphix.com
3:30–4:00	**Raffle**
4:00–5:00	**Parallel Sessions #4**
	Auditorium: *Best Practices for Data Loading with Oracle 11g Release 2*—Oracle Corporation
	Tassajara: *Next-Generation SOA Platform*—Oracle Corporation
	Diablo: *What Every DBA Should Know About TCP/IP Networks*—Chen Shapira, Hewlett-Packard
5:00–	**NoCOUG Networking and No-Host Happy Hour at Faz Restaurant, 5121 Hopyard Road, Pleasanton.**

RSVP online at www.nocoug.org/rsvp.html

The *NoCOUG Journal* design and production: Giraffex, Inc., S.F.
Front cover photo–Delicate Arch, Arches National Park, Utah–Photo by Tom Wagner.