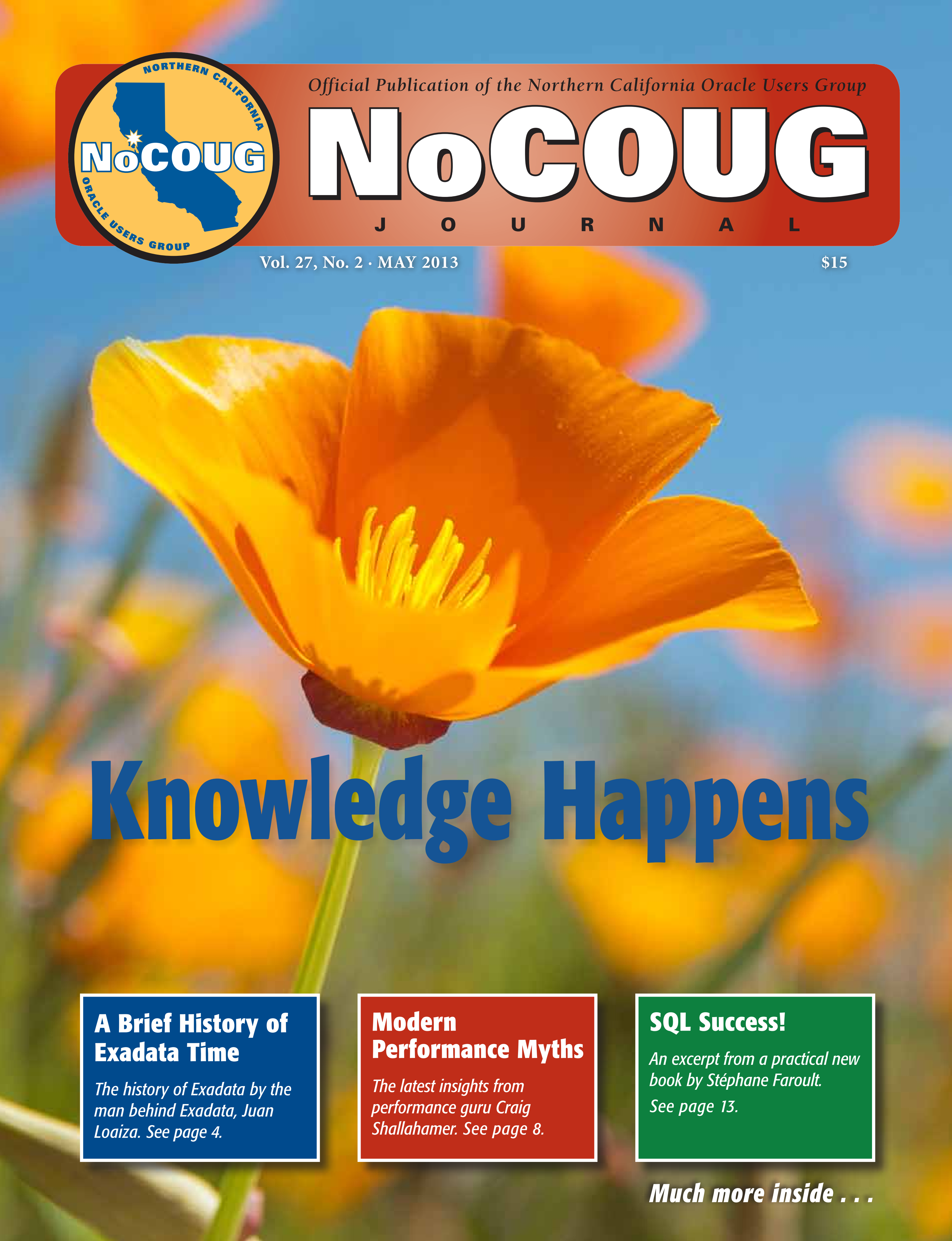
Official Publication of the Northern California Oracle Users Group

NoCOUG JOURNAL

Vol. 27, No. 2 · MAY 2013 $15



Knowledge Happens

A Brief History of Exadata Time

The history of Exadata by the man behind Exadata, Juan Loaiza. See page 4.

Modern Performance Myths

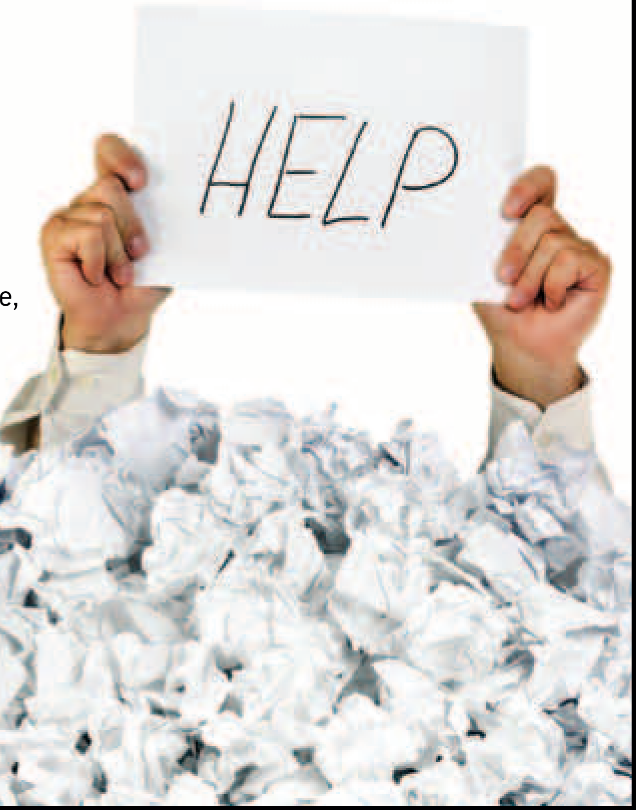
The latest insights from performance guru Craig Shallahamer. See page 8.

SQL Success!

An excerpt from a practical new book by Stéphane Faroult.
See page 13.

Much more inside . . .

Why You Need Change Data Capture for Big Data



DBMoto®

Change Data Capture for Big Data

- Real-time updates for high-speed analytic systems such as IBM Netezza, HP Vertica, Actian Vectorwise, SAP HANA
- Support for heterogeneous databases including Oracle, IBM, Informix, Sybase, Microsoft SQL Server
- Automatic data updates with Change Data Capture – no manual intervention
- Support for Cloud and Remote data
- Immediate, fresh data for Reporting, Analytics, Data warehousing
- Easy to set up and maintain
- Free evaluation at www.HITSW.com

FREE PAPER *"Change Data Capture for Big Data"*
http://info.hitsw.com/NoCOUG6

HiT® SOFTWARE
A BackOffice Associates, LLC Company

DBMoto®

Real-time Data Replication and Change Data Capture
for the toughest data replication jobs.

T +1.408.345.4001 www.hitsw.com info@hitsw.com

Professionals at Work

First there are the IT professionals who write for the *Journal*. A very special mention goes to Brian Hitchcock, who has written dozens of book reviews over a 12-year period. The professional pictures on the front cover are supplied by Photos.com.

Next, the *Journal* is professionally copyedited and proofread by veteran copyeditor Karen Mead of Creative Solutions. Karen polishes phrasing and calls out misused words (such as "reminiscences" instead of "reminisces"). She dots every i, crosses every t, checks every quote, and verifies every URL.

Then, the *Journal* is expertly designed by graphics duo Kenneth Lockerbie and Richard Repas of San Francisco–based Giraffex.

And, finally, Jo Dziubek at Andover Printing Services deftly brings the *Journal* to life on an HP Indigo digital printer.

This is the 106th issue of the *NoCOUG Journal*. Enjoy! ▲

—*NoCOUG Journal* Editor

Table of Contents

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 each issue is eight weeks prior to the quarterly conference. Article submissions should be made in Microsoft Word format via email.

Copyright © by the Northern California Oracle Users Group except where otherwise indicated.

NoCOUG does not warrant the NoCOUG Journal *to be error-free.*

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 Brief History of Exadata Time

with Juan Loaiza

Juan Loaiza

In this adaptation of a recorded interview conducted by the IOUG director of communities, Alex Gorbachev, NoCOUG readers are treated to a brief history of Exadata time, narrated by none other than the man behind Exadata, Juan Loaiza, senior vice president of Systems Technology at Oracle Corporation. Juan joined the Oracle Database development organization in 1988 and has contributed to every Oracle Database release since Oracle Version 6. He is in charge of developing the mission-critical capabilities of Oracle Database, including data and transaction management, high availability, performance, backup and recovery, enterprise replication, and Oracle Exadata.

I often hear people refer to you as the man behind Exadata. Can I ask you to give us a brief introduction to your role in Exadata?

In the early 2000s, one of the big issues we started seeing at a lot of our customers was bottlenecks, particularly in warehouse systems but also in OLTP systems, where the hardware wasn't really keeping up. There were bottlenecks in the hardware that were preventing the database software from really performing the way that we wanted it to. And so my group started a project that evolved into what is now Exadata. The idea was to design the best possible hardware platform for the Oracle Database. Until then, Oracle used whatever hardware was out on the market, but we started seeing bottlenecks so we decided to design something from scratch. And in doing so, we leveraged our 20–30 years of database experience to determine what would be the ideal platform for running the Oracle database. That's the thinking that produced the Exadata platform as we know it today.

Tell us a little bit about the evolution of the platform since the first appearance of the V1 release, and how Oracle made its decisions on what to improve and what features to add—you know, did Oracle kind of lead it or did the customers lead the decisions? How did that process work as you evolved the Exadata platform to X3?

We introduced our V1 product in the fall of 2008, and the goal was to establish the architecture for the Exadata platform. With V1 we had scale-out servers, scale-out storage, and InfiniBand, and we put intelligence in the storage with Exadata Smart Scan. Oracle implemented the storage layer using what is normally thought of as compute servers for storage—so we have a lot of compute power in the storage—and put database software on the storage servers to run data-intensive processing. Those are the basic building blocks of the Exadata architecture that were introduced with Exadata V1.

Another aspect of Exadata V1 was that it was the first engineered system. We made sure that components in the system worked really well together, and that the whole system was highly optimized—from the firmware through the OS through the database through the interconnect. We also had to make a lot of changes to the Oracle Database so that it could talk directly to the storage servers and push database processing into storage.

About a year later, we released the 11.2 version of the Oracle Database, and along with that came Exadata V2. We refreshed the hardware, using the latest chips, the latest disks, and the latest memory, but we also added two key new features. The first was the PCI Flash cache—adding high performance Flash to Exadata—and the second was Hybrid Columnar Compression, a very effective compression technology that we use primarily for warehousing and archiving. Exadata V2 was a very big release for us.

In 2010 we introduced several more improvements: we introduced Exadata X2-8, which has eight socket servers with lots of memory, support for Solaris in the database servers, and the Super Cluster variant of Exadata, which uses SPARC nodes. And that brings us to today and our introduction of Exadata X3, which has a lot of hardware enhancements as well as some new software technologies.

Oracle switched the platform to Sun after the acquisition—was that with V2?

Exadata V1 used HP hardware. Exadata V2 used Sun hardware. Oracle has always worked very closely with Sun, but of course, with Sun becoming a part of Oracle, our relationship became much closer. And, we had a very clear directive from Larry Ellison that engineered systems were critical to Oracle's overall strategy, and that was understood by both the database and hardware teams. We quickly got all the cooperation, all the features, all the fixes, and all the improvements that we wanted from Sun. When you are a single company and the direction is set very clearly by the leadership, then the hardware and the software integration can advance much faster. If you are two different companies, there are always different priorities in the different companies and this slows down progress.

Can you share with us what may be your own favorite new feature that is coming with the Oracle Exadata X3 generation?

We just launched Exadata X3 in late 2012, and the really big change is a shift toward in-memory processing. When Exadata V1 first launched, it was a purely disk-based system. We added Flash in V2, but still the focus was primarily on disks. We had a disk focus, with Flash for acceleration. The big change with Exadata X3 is that we've increased the Flash memory capacity very significantly: we've quadrupled the amount of Flash memory. Now, in an Exadata X3 full rack, we have 22 terabytes of Flash memory—which is a huge amount of Flash memory—and we have added software capabilities so that we can use that Flash memory very effectively. Now all reads and writes go directly to Flash. When you combine that with our compression technologies—both the Advanced Compression for OLTP and Hybrid Columnar Compression for warehousing and archiving—you can fit a very, very large amount of database data in Flash. Between the 22 terabytes of Flash per rack, the compression technologies, and the new technologies that put all the I/Os in Flash, the vast majority of customers can keep all of their active data in Flash. That's really a turning point, not just for us, but for the industry. The focus used to be purely on disk; then it became disk with Flash for acceleration, and now, really, the central focus of the architecture is memory: Flash memory and DRAM memory. That's where we are going to see the bulk of the future activity. Disk is still very useful because they can store vast amounts of data for a very low cost, but the focus of active data has really shifted toward Flash and DRAM memory. That's really what I think the big story is in Exadata X3: databases moving from being primarily spinning disk based, to databases primarily being based on semiconductor memory technologies—like Flash and DRAM.

Given the larger flash capacity and the fact that those cards also have a 40% higher throughput, giving Exadata 40% higher scan throughput, isn't InfiniBand starting to become a bottleneck in the data transfer? And if InfiniBand does become a bottleneck, how do you plan to address that?

InfiniBand still has lots of bandwidth in the Exadata X3 architecture. There are two common use cases: there is random I/O, and scan I/O. For random I/O, we can execute up to one and a half million random I/Os per second. If you do the math, one and a half million I/Os per second times 8K per I/O equals 12 gigabytes per second. Now that's a very, very large transfer rate, but it is well within the bounds of what we can achieve on Exadata. Where you could potentially run into bottlenecks is scans. Exadata can scan up to 100 gigabytes per second from disk and Flash, and if we actually had to ship 100 gigabytes per second from storage, we would run into a bottleneck in InfiniBand. But the way that Exadata works, we are able to push the scans into storage and filter the data before it's sent back to the database servers. So although it's possible to max out InfiniBand if you write just the right query, what I've seen is that it almost never happens in practice because you either get random I/Os that fit comfortably within the throughput of InfiniBand, or you get data scans, in which case the filtering happens at the storage level, or you get CPU-intensive operations, in which case the CPUs become the limiting resource. In the future, as database CPUs become faster and Flash becomes faster, we'll also be introducing faster InfiniBand. There are going to be various steps along the way, but ultimately, a few years down the line, there is 100-gigabit-per-second InfiniBand that is well on the way to being developed.

So with all those improvements in Exadata X3, and we focused on a few of those a little more, what type of applications do you think will benefit the most from the Exadata X3 enhancements and improvements?

Exadata X3 has been designed to work really well for all types of applications, including OLTP, warehousing, mixed workloads, and cloud, and I think they all will benefit from the enhancements in Exadata X3. The move to Flash and DRAM—in-memory technologies—is going to help across the board, across all the different types of applications.

For example, if you look at OLTP applications, the fact that the Flash capacity has increased by four times in X3, the number of CPUs has gone up, and the DRAM capacity has gone up, means we will be able to run OLTP much faster. With 22 terabytes of Flash, most OLTP databases will fit entirely or almost entirely in Flash, and so those should speed up a lot. Write-intensive OLTP workloads will particularly benefit from the new write-back Flash cache capability.

The very large amount of Flash will also benefit warehousing applications. Either the whole warehouse or at least the last year or two's data will fit entirely in Flash. Warehouses will run much faster Since Exadata X3 can run 100 gigabytes per second scans of data in Flash. X3 will also speed ETL jobs, particularly if there are indexes, because indexed ETL jobs tend to be very random I/O limited.

On the consolidation and cloud side, I think X3 will be very beneficial also. One of the things you have to worry about when you are consolidating a lot of databases or putting a lot of databases into a cloud is whether you have enough I/O capacity for all these databases. If you are putting 100 databases together or 1,000 databases together, you have to be very careful to ensure that one of them doesn't swamp the I/O subsystem and make all the other databases perform poorly. Now, with the large capacity of Flash and the large numbers of I/Os that we get from Flash, that's much less of a concern. We basically just increased the I/O headroom of the system by around 20 times, and so you don't have to worry nearly as much about one database in a consolidated environment swamping out all the rest of the databases.

Juan, you mentioned consolidation deployment scenarios with Exadata and how X3 helps here. One of the things I know customers are very concerned about is power, and I noticed that with X3 we have lower power than previous versions. How does it work? We have faster CPUs and we have more—and faster—Flash and more memory, but at the same time we actually consume less power. How did you make that work?

A number of improvements were made to power consumption also. The main improvement was that the chips that we use inside Exadata X3 just use less power. But there are also system design improvements. We use the fastest processors available, and those tend to run hot, so a lot of the system power goes into the fans that keep the processors cool. In Exadata X3, the airflow was redesigned and a new generation

"We leveraged our 20–30 years of database experience to determine what would be the ideal platform for running the Oracle database. That's the thinking that produced the Exadata platform as we know it today. . . . Exadata X3 has been designed to work really well for all types of applications, including OLTP, warehousing, mixed workloads, and cloud."

of fans was introduced that reduce the power consumption quite a bit. Also, the high-capacity disks have very low-power utilization. So it really is the sum of a lot of changes that resulted in lowering power and cooling by up to 30%. It's really good to be able to say that in X3 not only do we have faster processors, more memory, and more Flash, we also use less power, and it's all at the same price as the previous systems. We were able to achieve huge performance improvements while reducing power and keeping the price the same.

I know that Exadata adoption for larger deployments for larger customers is pretty good, but a lot of the smaller customers find it a steep entry point to get on Exadata and start adopting it. So is that why a one-eighth rack was created? How do you think it's going to change the Exadata adoption?

When we launched Exadata X3, we also introduced a new eighth-rack configuration, and the goal was to get the price point of Exadata down so that we get more users running on Exadata. That includes smaller companies, smaller departments, smaller applications, and smaller countries. It also gets the price point down for things like testing environments, DR environments—that kind of thing. The interesting thing about the eighth rack is that from a hardware perspective, it's exactly the same as a quarter rack. What we've done is disabled half the hardware. So for example, in the eighth rack we have half the CPU cores disabled, half the disks disabled, half the Flash disabled. That lowers the entry cost both at the hardware level and at the software level, so we can get more customers and more applications running on Exadata. And then it's very simple to upgrade, because we can just run a script to re-enable all the disabled hardware—it's sort of capacity-on-demand-ish. We can re-enable all that hardware for customers that need more hardware. It's very quick and very simple to do.

We've been focusing a bit on the hardware part of Exadata X3. Talking about the software side of Exadata, what are some of the improvements that customers should expect soon from Oracle in its sort-of-secret-sauce area in the software area of Exadata?

The team that developed Exadata is hard at work on the next generations already. We have Oracle Database 12c coming, and we've put a lot of changes into that release and into Exadata software. Among the improvements, for example, is a much more effective way of consolidating databases. We talked about this a little bit at OpenWorld. We are introducing the concept of a pluggable database and the ability to consolidate databases much more effectively, much more simply, with less hardware and much less management than is possible today. That's what we talked about as our multi-tenant, pluggable database. There will be special support for that in Exadata. For example, when we consolidate lots of databases, we can control the I/O usage by database or by job within Exadata. So that's one big area we are putting a lot of focus into—consolidation and clouds.

Another area that we talked about earlier is compression. Today we have Hybrid Columnar Compression, and today's technology is primarily focused on warehousing and archiving. We're doing a lot of work to bring Hybrid Columnar Compression to OLTP and mixed workloads also.

Another big area of focus is our RAC cache fusion technology. Today we take very good advantage of the InfiniBand network for I/O. It provides very good bandwidth at very low CPU usage because we use RDMA technology.

Now we are working on improving the throughput of cache fusion on our InfiniBand hardware. We're going to be introducing technology where we talk directly from the database to the InfiniBand cards, bypassing the complete OS software stack including interrupts to get the latency and CPU usage down for small inter-node messages. This is going to provide a big improvement for OLTP and for RAC cache fusion. That's another point of integration between hardware and software we are adding.

We're doing a lot of work on in-memory database technologies also, and you'll see huge performance improvements coming as we integrate DRAM technologies with special CPU chip technologies. We talked a little bit at OpenWorld about some of the changes that we will be making to the CPUs such as putting database functions directly into the CPUs.

So there's a lot of advanced engineering that's going on, and what is available today with Exadata is really just the beginning. We're no longer focused on making the basic platform work, we're now primarily focused on value add—things like improving consolidation, improving performance, improving compression—increasing the value of all the Exadata technologies even farther.

We're also adding technologies for our Oracle applications to leverage Exadata better. You'll have more and more reasons to choose Exadata as your hardware platform in the future as we integrate our software and our hardware even more closely together. ▲

Based on a recorded interview conducted by Alex Gorbachev, CTO of Pythian. Pythian is a global industry leader in remote database administration services and consulting for Oracle Database, Oracle Applications, and MySQL.

Nothing To Fear But Fear Itself

by Naren Nagtode

Naren Nagtode

Technology is changing every day, and you hear about new technologies and terminology that you never heard of before. It is easy to be in denial and tell yourself that these buzzwords are just hype and will fade away. A few years back we never heard of NoSQL, Big Data, and the Cloud, but they have started becoming more and more prominent.

It is easy to feel unsettled with new trends. However, there is no need to feel that way. Here are a few things you can do to keep your fears away:

Acknowledge that technology goes through cycles of change. The database and data areas are no exception. You have survived and thrived through change cycles before, and you will do it again!

Identify skills, tools, and technologies that are relevant to your career. One way to gauge the skills that you might want to develop is to periodically scan job postings and articles, and carefully look at what skills are being included in the job descriptions in your specialization—and then compare them with your skills. This will give you a guide to enhance the skills you already have and to acquire those you don't have.

Learn about new technologies and terminology. NoCOUG conferences and events are an excellent resource to learn about them. There is a mix of topics from the introductory to the expert level that will enhance your skills and knowledge so that you are ready for the new challenges.

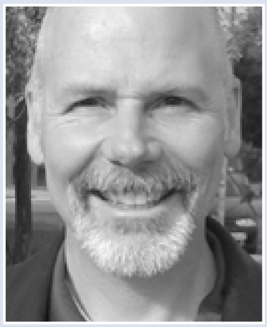
Network with peers and learn from them. The best way to learn and understand what is important and what is just hype is to discuss it with friends and peers in the field. NoCOUG provides a great way of meeting and networking with peers. Also, checking out what the vendors are talking about will give you an indication of the new tools and techniques that you should become familiar with.

At the spring conference on Wednesday, May 22, at the California Center Pleasanton, NoCOUG members will have the opportunity to spend a whole day learning application development with SQL Developer Data Modeler and Oracle Application Express (APEX). I'll see you there! ▲

Impressionist photo of the Fusion-io conference room where the Pi Day meetup was held on 3/14.

Modern Performance Myths

by Craig Shallahamer


Craig Shallahamer

Oracle performance analysis has come a long way in the last 20 years. First there was the "just add more resources" approach and tuning the blatantly poor SQL. Then there was ratio analysis, followed by wait event analysis, time based analysis, and unit of work time based analysis. In addition to the performance diagnosis and analysis evolution, Oracle as a product has changed, and architectures are more diverse. Yet with all this change, the myths I'm presenting are in many ways timeless. They relate to complexity, basic statistics, efficiency, and doctrinal purity.

Myth #1

The first myth is *Better tools are the solution to increasing architecture complexity.* I was attending a product demonstration a few years ago, and the presenter said something like, "Architectures are increasing in complexity. The solution is better tools." I looked around and everyone was agreeing, like in the Apple "1984" commercial, "Yes . . . Complexity is progress . . . We need more and better tools . . . Who will bring us these new and better tools?"

I was thinking to myself, how about we focus on reducing the complexity?! Am I alone in thinking that with each increase in complexity, there is an increase in potential problems, which means an increase in risk?

A few years ago I did some consulting for a very large and well-known ecommerce company. I was amazed at the lengths they went to keep complexity and risk low, uptime high, and performance consistently good. In addition, they architected their systems so workload could be easily and quickly partitioned. By keeping the complexity low, they were able to manage performance more simply and adjust more quickly. Their transaction throughput levels and on-line brand presence led them to the path of minimizing architectural complexity, resulting in an amazing uptime.

Myth #2

The second myth is *Users experience the average SQL elapsed time.* Concurrency, multiple execution plans, different bind variables, and the current cache situation make the average elapsed time less relevant—and perhaps even misleading. Let me explain.

If I tell someone the average elapsed time for their key SQL statement is 10 seconds, 99.9% of the time they will picture in their minds a bell curve. They will think to themselves, "OK. This means that usually the statement runs for about 10 seconds, sometimes less and sometimes more." Unless verified, my research clearly shows that there is likely to be a significant difference between the average and the typical elapsed time(s). This means we are missetting expectations and flat-out misleading users. Not the place we want to be!

It is very simple to calculate the average statement elapsed time. Even a Statspack report will show you the total elapsed time and the total number of executions over the report interval. The average elapsed time is simply the total elapsed time divided by the total number of executions. So it's very enticing to make a quick statement about the elapsed time using the average.

Now suppose this SQL statement has two execution plans: one typically completes in 2 seconds and the other completes in 18 seconds. Now also assume that they both get executed the same number of times (don't count on this in your environment!). The average elapsed time would then be 10 seconds. Now I picture myself telling my client that I had discovered the key SQL statement and its average elapsed time is 10 seconds. The "2 seconds" group would think I'm making the situation look worse than it is so I can claim an amazing performance improvement. The "12 seconds" group would think I'm simply an idiot. Either way, I lose. So I needed a way to find out the truth.

"How did James Tiberius Kirk beat the Kobayashi Maru? He changed the rules. To beat the performance game, Oracle sometimes allows performance analysts to change the rules as well."

What I learned through my research is that if the statement is important to the users and I care about not misleading them, I need to collect some data, calculate the statistics, and create a histogram. My research clearly showed three reliable ways to collect elapsed times: Oracle tracing by SQL_ID, instrumenting application code, and sampling running SQL. I have created a low overhead tool called "SQL Sampler" to sample elapsed time. You can download it for free from my website. The moral of this story is *Do not expect users to experience average elapsed times, and if appropriate, provide details about the typical elapsed times.*

Myth #3

The third myth is *Infrastructure-as-a-Service vendors want efficient systems.* Fact: An Infrastructure-as-a-Service vendor generates revenue when your system consumes CPU cycles, performs I/O operations, and sends/receives network packets. Therefore, Infrastructure-as-a-Service vendors want you to consume more CPU cycles, perform more I/O operations, and send/receive more network packets. If that doesn't convince you, think about the marketing messages promising that all the computing resources you will ever need will be easily available.

The solution to "buy more hardware" is now very, very easy. There are no additional contracts; no one to call to install more hardware; no approvals, purchase orders, or approval chain of command; and on and on. This is a vendor's dream!

The message here is "Eat more and be happy, for tomorrow we die," not "Eat healthy and prosper" (or something like that).

But there is good news! Now performance specialists can quantify their work monetarily. Instead of saying the business process dropped from 2 hours to 2 minutes, we can also say we have saved the company $2 million each year!

How can we make such an outlandish statement? It's because the quantitative performance analyst knows how much less CPU time will be consumed and how many fewer I/O operations and network transfers are required. And the analyst knows how much CPU, I/O, and network resources cost! (If you don't know how much they cost, then ask.)

This is yet another reason to get good—very good—at quantitative Oracle performance analysis! We need to understand that it's now very easy to acquire more computing capacity, so the pressure and priority to optimize can be allowed to decrease . . . but tomorrow we die. Focus on the ability to quantify our performance work monetarily, and the priorities will more likely align properly.

Myth #4

The fourth myth is *It is heresy to consider the impact of deviating from core relational principles.* How did James Tiberius Kirk beat the Kobayashi Maru? He changed the rules. To beat the performance game, Oracle sometimes allows performance analysts to change the rules as well. While this is relational heresy to some, the freedom is intoxicating!

Here is an example of what I'm talking about: Would you seriously consider allowing a commit statement to immediately return as successful when the associated redo has not been written to disk?

If you have been working with Oracle databases for many years, there are probably many concepts that we will naturally agree or disagree on . . . even without really thinking about it. For sure, I am a big fan of simplicity and choosing consistent performance over the hope of amazing yet inconsistent performance. But sometimes there are ways to have your cake and eat it too.

Back to my example about commits: Oracle's commit write facility provides the ability for a commit to quickly return even though the redo has not been written to an on-line redo log. The have-your-cake aspect is that this can be applied at the instance and session—and even at the statement—level.

A few years ago, while introducing this "feature" in my Oracle Performance Firefighting class, a student proudly announced that her company uses the commit write facility. The other students were shocked and honestly appalled. I was smiling like a Cheshire cat! She went on to explain that their application was about social networking. She posed the question, "If you are adding people to your social network and during the save-your-work process there is a failure, what would you do?" Everyone replied, "Just re-enter the information." She replied, "Exactly! So why would a company want to spend literally millions more on hardware in the unlikely event of a failure when the business solution impact is to simply re-enter a few email addresses?" There was silence and then a big smile on everyone's face!

"It may be OK to break some fundamental rules as long as the business and its users are not negatively affected. But it takes a brave person to face the immediate "you're a heretic" response."

My point is that it may be OK to break some fundamental rules as long as the business and its users are not negatively affected. But it takes a brave person to face the immediate "you're a heretic" response. So the next time you're facing a serious performance issue, think about a few possible ways to break the rules. ▲

Craig Shallahamer is a researcher and author who has taught thousands of Oracle professionals. For over 20 years, his focus has been Oracle performance analysis, including firefighting and capacity planning. He is the founder and president of OraPub and an Oracle ACE Director, and he regularly posts his newest research on his performance research blog, "A Wider View." He can be reached at **craig@orapub.com**.

Oracle Performance Research Seminar 2013

Wednesday, August 14 at TechMart, Santa Clara

▪ ▪ ▪

This 1-day research seminar is a unique Oracle performance analysis course answering the "why" questions. Scientific experimentation has led to a number of amazing discoveries about how Oracle really works and how to use this information to our advantage.

This seminar was developed by OraPub founder and Oracle ACE Director Craig Shallahamer, author of the books *Oracle Performance Firefighting* and *Forecasting Oracle Performance*, and the insightful blog, *A Wider View*. Craig is a gifted teacher with 20+ years of experience. He has a knack for making complex topics come alive and become seemingly uncomplicated.

Each year Craig takes his raw research and weaves it into a cohesive seminar that is extremely practical day to day. We hope you can join us for this fast-paced and exciting day!

You will

- Gain deep performance insights.
- Learn about general performance analysis topics (e.g., transaction arrival patterns and serialization) that have a profound impact on performance—all based on experimental research.
- Learn about specific performance topics, such as Oracle internal structures and algorithms—all based on experimental research.
- Learn how to use the research in your daily performance analysis work.
- Learn how to develop, run, and analyze your own performance experiments.

Craig will cover

- Research tricks and tips for Oracle performance analysis
- Cursors, child cursors, and more child cursors
- Parsing: hard, soft, and softer—how bad is bad?
- Altering DML batch sizes
- Resolving log file sync issues using commit write
- Dealing with skewed data: data gathering, graphing, diagnosis, and analysis
- Merging Oracle time-based analysis with queuing theory (making it practical)
- Creating realistic sleep times when loading a system
- Are you sure it's the index root block?
- Calculating CPU utilization from v$ views on any Oracle system—and more!

Register now at **www.nocoug.org**

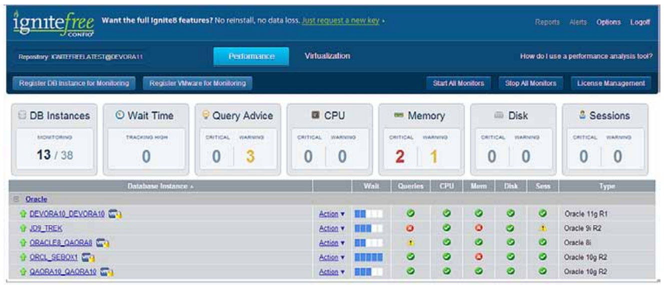
IgniteFree!

by Jerry Esbaugh

The only thing that has proven constant in the world of technology is change. Unfortunately, change frequently brings an increase in complexity. This growth in complexity can be seen in the majority of performance monitoring tools that still focus on yesterday's challenge of DBMS health monitoring. A recent Forrester report states that *"today's common complaint is no longer as much about a total failure of a component (binary events) as it is about sluggishness (analog performance)."* ("*Turn Big Data Inward With IT Analytics*," by Glenn O'Donnell and Jean-Pierre Garbani, Forrester, December 5, 2012.) Dealing with this transformational change requires detailed performance analytics that automatically condense a plethora of data into relevant performance graphs that are quickly understood by a wide range of IT professionals.

Confio IgniteFree

IgniteFree is an Oracle performance tool that gives you free access to the power of response-time monitoring and analysis. As a freemium product, there is a substantial amount of Ignite's power and functionality contained within. IgniteFree monitors continuously and provides real-time Oracle performance information and session detail, while IgniteFreeVM correlates server resource utilization back to SQL response time on physical and virtual (VMware) servers. Both IgniteFree and IgniteFreeVM give DBAs and developers a great introduction to the power of response time analysis on physical and VMware servers respectively. To view historical information greater than 1 hour, monitor more than 20 instances, or conduct in-depth performance analysis, just request an Ignite trial key.


ignitefree CONFIO
Want the full Ignite8 features? No reinstall, no data loss. Just request a new key ›
Reports
Alerts
Options
Logoff
Repository: IGNITEFREELATEST@DEVORA11
Performance
Virtualization
How do I use a performance analysis tool?
Register DB Instance for Monitoring
Register VMware for Monitoring
Start All Monitors
Stop All Monitors
License Management

DB Instances	Wait Time	Query Advice		CPU		Memory		Disk		Sessions	
MONITORING	TRACKING HIGH	CRITICAL	WARNING	CRITICAL	WARNING	CRITICAL	WARNING	CRITICAL	WARNING	CRITICAL	WARNING
13 / 38	0	0	3	0	0	2	1	0	0	0	0

Database Instance ▲		Wait	Queries	CPU	Mem	Disk	Sess	Type
Oracle								
DEVORA10_DEVORA10	Action ▾	▮▮▯▯▯	✓	✓	✓	✓	✓	Oracle 11g R1
JD9_TREK	Action ▾	▮▮▮▯▯	✗	✓	✗	✓	⚠	Oracle 9i R2
ORACLE8_QAORA8	Action ▾	▮▮▯▯▯	⚠	✓	✓	✓	✓	Oracle 8i
ORCL_SEBOX1	Action ▾	▮▮▮▮▮	✓	✓	✗	✓	✓	Oracle 10g R2
QAORA10_QAORA10	Action ▾	▮▮▮▯▯	✓	✓	✓	✓	✓	Oracle 10g R2

Figure 1: IgniteFree's web browser dashboard provides a high-level view of your datacenter.

Complements and extends OEM

The fact that the number of Oracle teams deploying Confio Ignite is growing at a rate of 47% per year is a testament to the value that Ignite delivers. The combination of OEM and Ignite helps a DBA quickly understand and communicate database performance issues, resulting in a savings of time, money, and frustration.

Ignite serves an important role with its ability to analyze millions of data points automatically, and pinpoint the most significant contributors to poor Oracle performance. As senior DBA Kathy Gibbs pointed out in her white paper "Confio Ignite or OEM? The DBA Says Both!," "*There are some very distinct differences between OEM and Ignite. OEM (sometimes referred to as Grid Control) was created to help manage the Oracle database environment. In contrast, Ignite is a tool that focuses on database performance monitoring using wait time analysis. Confio Ignite is a comprehensive database performance analysis and monitoring solution for DBAs, IT managers, and application developers. Ignite identifies performance bottlenecks, improves application service, and reduces overall cost of Oracle database operations.*"

Focuses on response time

Ignite is unique in combining the most important analysis technique, response time analysis, with a full set of customizable server physical resource monitors, VMware monitors, advisors, alerts, and reports into a single product. In addition to Oracle, Ignite also monitors SQL Server, Sybase, and DB2 from a unified dashboard. While many tools on the market (including OEM) focus on server health management, Ignite is focused on pinpointing performance issues and proactively monitoring 24/7 to ensure best-in-class database performance.

The time between a SQL query request and response is the time that applications and people are kept waiting. Ignite monitors 24/7 with less than a 1% load on monitored database instances. All data is stored in a separate repository, so there is zero impact of any performance analytics on the monitored instances. Easily understood charts make performance problems quickly understood and communicated.

Within just three clicks, you can pinpoint the root cause of most problems; this golden triangle includes worst-performing SQL queries, total response time, and the specific wait events.

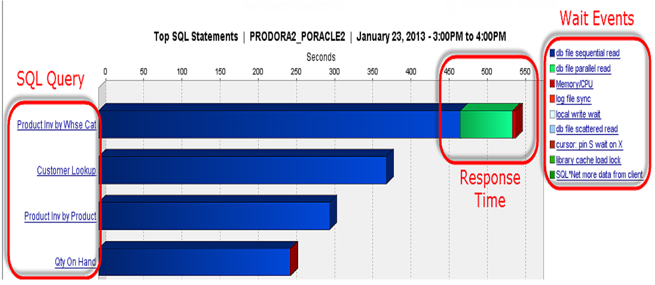


Figure 2: Ignite gives you a clear picture, including the SQL queries, response time, and wait events.

Server resource correlation

When it comes to performance, there are two distinct perspectives: resource utilization and query response. Resource utilization is important, but it all too frequently leads to the conclusion that more hardware is necessary, when the actual problem resides with the SQL query, blocking/locking, indexes, or explain plans.

By focusing on query response first, Ignite identifies the time periods that are most problematic and correlates this back to resource utilization. This approach creates a clear picture of exactly what the problem is and how best to resolve it.

IgniteFreeVM adds visibility to the physical host that VMware's virtual machines run on, allowing you to identify resource and provisioning issues for databases running on VMware. The grey arrows in Figure 3 identify changes to VMware configuration, including vMotion events where the database server is moved between hosts. In addition, IgniteFreeVM monitors and displays extensive physical and virtual server metrics, including CPU, memory, and network, and extensive storage layer visibility.

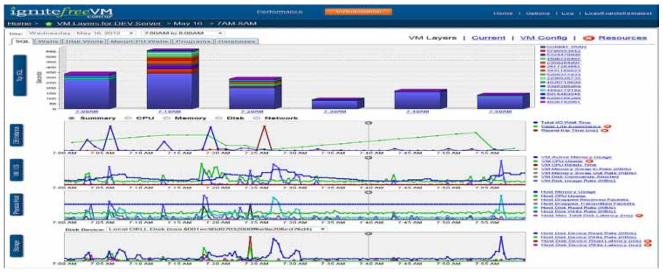


Figure 3. Identify the top SQL bottlenecks and correlate them to physical and virtual machine utilization.

Collaboration

Achieving best-in-class Oracle performance is a collaborative effort. It requires management buy-in, well-thought-out architecture, developer interaction, and a skilled DBA's analysis and tuning skills.

Ignite allows the entire team (developers, architects, managers, QA engineers, and operations DBAs) to collaborate on the performance of applications. By improving visibility and removing communication barriers, you can focus on maximizing database performance. Key features supporting collaboration include:

- ➤ Web browser interface
- ➤ Easy-to-understand graphs and alarms
- ➤ Automated reporting
- ➤ Proactive alerting
- ➤ Multidimensional views

The result is being able to solve performance issues quickly and without finger pointing. In addition, proactive use allows the entire team to identify anomalies and trends before they affect your end users.

Maximizing your Exadata investment

As Tanel Põder points out in the book, Expert Oracle Exadata, *"[T]o avoid wasting effort on fixing the wrong problem, we really need to measure what matters! For database performance, nothing matters more than response time."* Ignite has added the capability that identifies the wait events having the greatest impact on response, helping you get the most out of your Exadata investment.

There are three distinct issues faced at different times when adopting Exadata:

1. The wealth of information provided by Exadata can be overwhelming to learn and master. Many people fall into the trap of monitoring Exadata metrics, misleading them into focusing on, and fixing, the wrong problem. Ignite makes your entry into Exadata much easier by providing clear visibility on the metrics that have a direct correlation to performance.
2. Regardless of hardware, the old adage is true: bad code and bad architecture, even on Exadata, are still bad. Ignite gives you the ability not only to identify performance bottlenecks but also to communicate the actual root cause of the performance issues to others. This not only accelerates the problem identification, it also provides baseline metrics to validate the effectiveness of any changes.
3. The benefits of Exadata for OLTP, even with its new smart Flash cache, are not fully realized if the smart Flash cache is not properly configured. Beyond the obvious Flash cache hit ratio, the block read latencies provided by Ignite are one of the most effective measures of cache performance. Other key factors include:

- ➤ I/O saved by cell offload
- ➤ Smart scan efficiency
- ➤ Cell single block physical read latency
- ➤ Cell multiblock physical read latency
- ➤ Cell smart table scan latency

Exadata is a big investment. When this hardware is offline or underperforming, it is not delivering value to your organization. Effective monitoring is key to maximizing your investment.

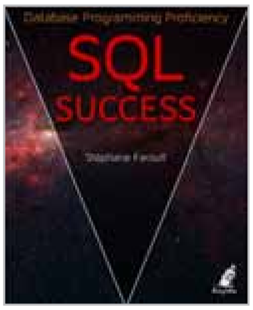
Conclusion

Automated performance analysis is a key to simplifying complexity and obtaining best-in-class Oracle performance. Ignite is the choice of 36% of the Fortune 50 companies because of its ability to support the largest, most-demanding database environments. As the fastest-growing company in the database performance management market, Confio recently received the prestigious Deloitte Technology Fast 500 and Inc. 500/5000 awards.

See for yourself. Download the completely free version, IgniteFree, which includes the basic features for use on up to 20 instances, from **www.confio.com**. ▲

SQL Success!

by Stéphane Faroult



This is an excerpt from SQL Success: Database Programming Proficiency, *a practical new book by Stéphane Faroult which will be released later this year. Stéphane is the author of* The Art of SQL *[O'Reilly, 2006] and* Refactoring SQL Applications *[O'Reilly, 2008.] His new book is light—though rigorous—on theory and heavy on practice. It gives equal coverage to Oracle, SQL Server, MySQL, PostgreSQL, DB2, and SQLite. The full outline and more excerpts can be found at* **http://edu.konagora.com.**

There is one thing that you must be conscious of: A modern DBMS scans tables very fast—so fast, that failure to properly index tables may pass unnoticed until it hurts. For instance, I have artificially grown the *people* table from the sample movie database up to about 400,000 rows, and I have run, on a fairly modest server, two searches, one using an index and one scanning the table. I've run the queries a large number of times to get an average time per execution, and I got 0.1 millisecond when finding rows using the index and 0.4 seconds when scanning all the rows—inspecting 400,000 rows in 0.4 seconds isn't bad in itself. Table scan performance can be much, much better on high-range machines with state-of-the-art disks—or, even better, most of the data already in memory.

On a single query, with a comfortably sub-second response time in both cases, a human being will hardly notice much of a difference, even if we extract the result from the database 4,000 times faster in one case than in the other case. The problem becomes noticeable—very noticeable as soon as queries are repeated a large number of times—for two reasons:

- First, the simple cumulative effect acts as a magnifying glass. If you repeat the operation 100 times in a program, for a human being there is a large difference between 0.01 second and 40 seconds.
- Second, the fact that if to simplify we say that out of the 0.4 seconds required when scanning the table, 0.3 are pure computer processing (the remainder being various delays such as waiting for the completion of disk access operations) you can only perform three such operations per second and per processor. If queries are issued at a higher rate—for instance, by many web users accessing the same site simultaneously—the various queries will be entered into a waiting line by the DBMS until they can be executed by one of the computer's processors, and the response time seen by users will be the sum of the actual execution time (one-third of a second) and of the time spent waiting, which can be very long. Perceived performance will crash all of a sudden as soon as the load exceeds a limit that depends on the power of the machine, even if unitary testing gave acceptable response times—in practice, performance will crash as soon as queries arrive faster than they can be executed (it's exactly the same phenomenon as traffic jams at toll gates: when cars arrive faster than the time required to pay, lines grow).

There is another point to consider: so far I have compared fetching one row using an index to fetching one row by scanning a table, and when I talked about repeating the operation in a loop, I was talking about scanning repetitively a table. If we have many different rows to fetch, we can probably fetch them in a single pass over the whole of the table. If fetching one row using the index is, as in my example, about 4,000 times faster than inspecting all rows, when a query returns 4,000 rows it makes no difference whether we are using the index to fetch rows one by one or scanning the table and returning rows as we find they match the search criteria—the time will be the same. And if we want to return 40,000 rows out of 400,000, then it will take 4 seconds if we use the index to locate each of them individually, while the full scan will still take 0.4 seconds—the time required to scan the table is the same whether we ultimately return zero, one, or several thousand rows. With indexes, the more rows we return, the longer it takes. It really is a question of using the right algorithm for the right volume of data.

This is why determining which columns really require indexing is of the highest practical importance.

Choosing Which Columns to Index

Determining which columns would benefit from indexing in addition to those automatically indexed by constraints requires some thought—in fact, quite a lot of thought. The columns that are the best candidates for indexing must satisfy a number of conditions:

- First, they must be used often as search criteria. A typical example would be a table that contains currency exchange rates. Your criterion for searching such a table will never be "Which are the rows corresponding to an exchange rate of 2.5678?" Although the rate itself is what you really want from the table, it's the currency code and, if the table stores historical values, the date that will drive the searches. And when you think "index" for this table, you should think "currency code" and "date" (which are likely to be the primary keys). A column that is only rarely used as the main search criterion isn't necessarily a good candidate either, unless 1) it's critical that the queries that rely on this column return very fast, and 2) the index really makes a difference. Performance gains for occasional queries must be put in balance against the overhead induced by index maintenance when inserting, deleting, and perhaps updating, as well as against the extra storage required. Good overall performance is often a question of compromise.

- Second, the columns must contain values that are selective enough to help the DBMS engine to focus fast enough on the final result set. Once again, a comparison with libraries may help you understand: in the library of the computer science department of a university, it makes little sense to index books that mention the word "computer"—all the books will probably contain it. Words like "memory" or "disk" are unlikely to be much more helpful, and even "SQL" will probably get a mention in many books. The value of a search criterion is inversely proportional to its frequency. You have experienced it with web search engines: The rarer the value you are looking for, the faster it allows the DBMS to identify the result set you want to see and the more sense it makes to index it.

I want to briefly point out a potential snag: if I take the example of table *movies* in the movie database, should column *country* be indexed for searches? The problem is that some countries produce thousands of movies every year, and for them the country alone isn't a very selective criterion. It's likely that the query will involve other search criteria and that some of those will be more effective for helping the DBMS shape the final result set. Even if the country is the only search criterion, it is quite possible that plainly scanning all the rows in the table will be more efficient than finding references in the index and then fetching rows one by one (a full table scan will certainly be faster if the country is "*us*").

On the other hand, the very same "country" criterion that isn't very significant for countries with a buoyant movie industry may be very selective for some countries where film-making is still an art or a craft and not an industry, and where very few films are produced.

The "big" database management systems have sophisticated optimizers and usually collect statistics about the distribution of values, and they know which are "popular" values (a "popular" value is a value that you find very often and that isn't very selective). When the SQL engine evaluates how best to run a query, it checks whether indexes can be used to speed up the search, and if this is the case, a second check can be performed to verify that the value that is searched is rare enough to make the index search faster than a plain table scan. A DBMS may decide not to use an existing index because the value that is searched is too common.

Column *country* in table *movies* is also a foreign key (referencing column *country_code* in table *countries*). Many people recommend always indexing foreign keys (some DBMS products require foreign keys to be indexed); I have mixed views on this topic.

The main rationale for indexing foreign keys is linked to an aspect of referential integrity that people often forget: if foreign keys prevent you from inserting into the referencing table a row with a value that cannot be found in the referenced table, they also prevent you from deleting from the referenced table a row that matches rows in the referencing table, because the disappearance of this row would make data inconsistent. For instance, you cannot delete a row from table *countries* if *movies* contains films from this country. If I try to delete one row from *countries*, the SQL engine must first look into *movies* (and possibly other tables) to see if there is a row referencing it. In the absence of any index on column *country* of *movies* (the foreign key column) this will require scanning table *movies* until either the DBMS finds a film for this country—which would prevent deletion—or it has checked all films, which may take a long time if the table is really big. Moreover, while doing so, the SQL engine will need to prevent the insertion by other users of rows into table *movies* because someone might want to insert the first film for a small country at the same time as the DBMS checks whether another session can safely remove this country from table *countries*. An index on column *country* of table *movies* allows checking consistency very fast.

But this particular concern with foreign keys only happens if we want to delete some countries, and for many reference tables—and *countries* is a good example—we'll never want to delete rows (even if the uncertainties of geopolitics redraw the map, if some countries disappear and other countries are created, we may want to keep references to the state of the world when a movie was released). The problem with indexing all foreign keys is that some tables have many foreign keys: I could, for instance, add other attributes to table *movies* such as genre, main language, company owning the rights to the film, and so on, all of which would probably be foreign keys. As I have just told you, indexes are maintained as tables are changed. If your database is a read-only decision support system, you can happily index all columns in your tables. If your tables are heavily changed, whether you insert, delete, or update many rows, the additional work required to maintain indexes consumes processor time as well as, very often, disks accesses, and significantly slows down change operations to the database. I'll want to index some of the columns—those that are important for searches—but not all of them.

This is a test you can easily replicate. I have inserted 100,000 rows into a table of a dozen columns.

In the same time I could insert 100 rows without any index,

- After creating a primary key index, I inserted 65;
- After adding a second index, I was down to 22;
- After adding another index, I could only insert 15; and
- A last index brought my throughput down to 5.

The actual numbers you get may differ with the DBMS; variations can be more or less dramatic, but since indexes are maintained in real time, each index inflicts a performance penalty, and it can be verified with any database management system. It's not a concern in a read-only decision support system, but it can be one in an operational system with a lot of insertions.

I certainly want some indexes besides those that are required to ensure data integrity, but only if they are worth the cost.

Even worse, when a large number of processes are concurrently inserting into a table, it is common to see concurrent accesses to indexes being the biggest bottleneck.

It means the benefits that are expected of each index must be carefully assessed against the predictable overhead. Contrary to what I have seen advised more than once in forums by well-meaning but unenlightened advisors, you shouldn't index all columns that appear in where clauses: once again, you should only index columns that are important to searches and would benefit from an index.

Checking That Indexes Are Used

Before I discuss further a number of important index-related issues, it's time to introduce an SQL command that is a part of the standard but the output of which is highly dependent on the underlying DBMS, and that tells you (broadly) how a DBMS processes a statement. This command is *explain* and it is used as such with DB2 and Oracle:

```
explain plan for <SQL statement>
```

Note: Prior to running an explain statement, with DB2 you must install some system tables that are created by a script named explain.ddl, located in the DB2 directories. You can run the script with the following command: *db2 -tf explain.ddl.*

With PostgreSQL, MySQL, and SQLite, it's simply:

```
explain <SQL statement>
```

Note: Actually, with SQLite explain <query> returns a very detailed trace for the developers of the product, but SQLite also supports explain query plan <query>, which shows summarily, in an intelligible way for people like you and me, how the query was run.

Explain doesn't exist in SQL Server, but you can obtain the same result by running

```
set showplan_all on
```

prior to running the query (better still, you have the *Display Estimated Execution Plan* icon located a few icons away from *Execute* in SQL Server Management Studio).

For instance, you can check how a DBMS runs a simple query by typing, depending on the product you are using,

```
explain plan for select * from movies where country = 'gb'
```

or

```
explain select * from movies where country = 'gb'
```

or

```
explain query plan select * from movies where country = 'gb'
```

When an SQL statement is preceded by *explain*, it isn't executed. Instead, it's analyzed by the SQL engine that determines how best to run the statement and produces what is called the *execution plan*. The execution plan, among other things, gives the names of the tables and indexes that are accessed to run a query. With some products, *explain* immediately displays some output; with some other products, it just stores the plan somewhere, and the plan has to be queried from some temporary table as a second step.

Products that store the plan in tables use column names that aren't really self-explanatory nor an ideal of user-friendliness. This is why I would recommend using IBM Data Studio for DB2 for this purpose (the "Open Visual Explain" icon is located next to the "Run SQL" icon) or SQL Developer (the "Explain Plan" icon is two icons away from the "Run Statement" icon) with Oracle. With these tools there is no need to type the *explain* command—just type the SQL statement and click on the button that generates the execution plan.

If you really, really want to display an Oracle execution plan under SQL*Plus, you should run the following command after the *explain* statement

```
select plan_table_output from table(dbms_xplan.display());
```

You can try *explain* immediately with two queries (adapt the syntax to your DBMS):

First, try

```
explain select * from movies
```

Then, try

```
explain select * from movies where movieid = 1
```

The second execution plan shows that the SQL engine would use the index associated with the primary key.

Just a word about execution plans: don't try to assign to them any "good" or "bad" qualifier. Some people are obsessed by execution plans and, particularly, tables that are accessed without passing by an index. There are times when using an index is a pretty dumb thing to do, and what matters is not *how* but *how fast* statements are run. Besides, execution plans are particularly difficult to read (except for very simple queries) and based on information—such as how data is physically stored in the files—that has nothing to do with the logic of a business or an application. Even senior database administrators often have a hard time with them. Study your queries more than execution plans.

As the tables in the sample database are small, using an index or not using an index makes no perceptible difference in terms of response time; it would be different with large tables. *Explain* (or the equivalent) will help you see when a DBMS uses indexes—indexes on which you are relying for performance cannot always be used, as we are going to see now.

Keeping Indexes Usable

When you are introduced to indexing, you usually think that if a column that is referenced in a query is indexed, then everything is fine: the SQL engine will use the index and the query will run fast. Reality is more complicated. First of all, as

I have told you, sometimes you need indexes and sometimes you don't, and you need to determine which columns require indexing. A DBMS will not create indexes for you.

Note: A DBMS will not create indexes but sometimes it can suggest them. Corporate-grade DBMS products usually include (or let you buy as an expensive additional option) automated "performance advisors" that can tell you where additional indexes might improve performance by analyzing queries that are run. Be aware that the analysis is based on existing queries, and often rewriting queries is more efficient than adding more indexes, as you are going to see in this section.

If the existence of an index is a prerequisite to indexed access, it's far from the only condition. I have already mentioned that an optimizer may choose not to use an index because the value that is searched is too common, and scanning the table is faster overall than searching the index and then retrieving rows one by one. There are also cases when an index exists, but the query is written in such a way that the index cannot be used. This is what we are going to see now. If you want to shine when optimizing queries, you need to understand what can get in the way of the tree search that is applied to indexes when the DBMS tries to locate the addresses of rows that contain a particular value.

Using Composite Indexes Correctly

To be correctly used, composite indexes (indexes for which the index key is, in fact, the combination of several columns) demand that several conditions are satisfied. What you must understand is that when an index is built on several columns (most products allow at least 16 columns in an index, but in practice, composite indexes usually involve two or three columns), it's like having an index on a single column that would be the concatenation of all the columns in the index. A real-life example of a composite index would be a phonebook, in which the key to finding a phone number is composed of the surname, the first name, and the address, in this order. The interesting point is that to be able to find a number, you don't need to know all the components in the index key, but you absolutely need the first one. If you have met people at a party whom you would like to meet again and who told you their first name and the street where they live but not their surname, a phonebook will be useless to find their number (unless you read the phone book from A to Z, which you might call a full phone book scan); your best chance to get their phone number will be to ask a common acquaintance to whom the first name will mean something (in a way, the common acquaintance will act as an index on the first name).

The same is true with indexes in relational databases: If you don't provide, either directly or indirectly (that is, by providing the value through a join), the value that the first column in the index must match, the index will be unusable. But if you provide only the value for the first column, then the index will help you locate rows that are candidates for the final result set—in the same way that if you only have a surname and an address, assuming the surname isn't too common, you can scan all the entries for that surname in the phone book until you find the good address, ignoring first names in the process. That means that whenever you create a composite index, you must be extra careful about the order of the columns, because it will determine when the DBMS will or will not be able to use the index. Columns that will always be referred to in the *where* clause must come first. Columns that may not appear in the *where* clause must come last. Of course, it's always possible (and commonly done) to separately index columns that are already part of a composite index but not in the lead position, when they can appear as the only search criterion. Remember, however, that the additional cost of maintaining an index must be justified.

You can test this behavior with *explain* in the sample database by querying all columns from table *people*, which has a composite index on the surname and the first name, in this order. A condition that will include the surname will use the index, with or without a condition on the first name. But a condition on the first name without the surname will not use it.

You can recognize the same problem pattern on searches such as

```
where surname like '%man%'
```

that don't provide the beginning of the string to match or conditions that use regular expressions; if the column that is searched is indexed, you have exactly the same issue as with composite indexes. This is why in Chapter 6 I have shown how you could transform an approximate search for a title into a precise search for keywords, because any search for chunks of text will ultimately translate into some kind of scanning unless another criterion is provided that allows an efficient index search.

Not Applying Functions to Indexed Columns

The other important point for performance is related to functions, and the problem comes from the tree structure that is used to find row locators quickly. I told you that the tree structure is what allows the DBMS to mimic the type of search you would perform with a dictionary, and I insisted on searches being based on comparisons; let's say that instead of asking you to find the definition of a word in the dictionary, I ask you to give me the definition of all words that contain A-T-A as the second, third, and fourth letter respectively. There is no other possibility when searching the dictionary than trying, one by one, all possibilities for the first letter, because words that match the search criterion can be found anywhere in the dictionary, c*ata*clysm, f*ata*l, m*ata*dor, or, s*ata*y, among others—not forgetting d*ata* and d*ata*base. Having a condition applied to the second, third, and fourth letters instead of the full word prevents alphabetical comparisons.

If I reword the problem definition in SQL syntax with a mock table,

```
select word_definition
from word_dictionary
where word = 'database'
```

is fast and easy if column word is indexed, but

```
select word_definition
from word_dictionary
where substr(word, 2, 3) = 'ata'
```

is slow and painful, even with the same index.

Isolating three letters—in other words, using function *substr()*—completely breaks the order on which we rely to find words. Whenever we apply a function and transform values, the key ordering that is assumed when searching the index becomes moot, even in the rare cases when the function or expression doesn't affect order; this is what would happen, for instance, if you were extracting the first three letters of a word. Many DBMS products play it safe, assume that order *may* be changed, and won't search the index. In some cases the DBMS might find it more efficient to scan the index than to scan the table (an index is usually smaller than the table it's built upon), but in any case it will not be an efficient index search.

I told you in Chapter 3 that functions (and expressions) shouldn't be applied to columns in the *where* clause, and the reason I told you so is precisely because functions "break" indexes. If you apply a function to a column that isn't indexed, it doesn't really matter, but I routinely see functions or expressions that are applied to indexed columns.

One common reason for applying functions to columns in *where* clauses is performing searches that are not sensitive to case. If the DBMS product is case sensitive for text and if you don't force case when inserting data, then a case-insensitive search will usually become a condition such as

```
where upper(surname) = upper('some input')
```

because internally uppercase letters come before lowercase letters (and accented letters come after them). When case isn't consistent, a DBMS that is case sensitive would, for instance, order surnames like this, with uppercase characters first (because this is how internal codes that represent letters have been assigned):

```
MILES        < Uppercase letters first, sorted
O'brien
Stewart
Marvin       < Lowercase letters next, sorted
wayne
```

Values would be ordered like this in an index too.

Note: If you are using a case-sensitive operating system such as Linux, you also get the same type of ordering when you list files in a directory.

Imagine now that you have a dictionary in which words and names are ordered as in the preceding example, with a random, unpredictable case, and that you are looking for the name that, once in uppercase, is MARVIN (in other words, *where upper(surname) = 'MARVIN'*). If you open the dictionary at random and land on the "Stewart" page, because of the uppercase transformation you are unable to say whether the value you are looking for will be before this page (which would be the case if the name had been entered as 'Marvin') or after, as it is here. The same thing happens, for the same reason, with an index. Apply upper() to the searched column, and you won't be able to perform a search in the index and use it to locate rows.

How functions such as *upper()* or *substr()* (*substring()* with SQL Server) prevent you from using indexes is again something that you can try with *explain*.

Another common reason for applying functions to columns is type conversion, particularly with dates; to find all rows related to June this year, many people would, for instance, use a function such as *extract()* twice to say that the month must match 6 for June and that the year must match the current year:

```
where extract(month from date_column) = 6
  and extract(year from date_column) = extract(year from
current_date)
```

extract() is a conversion function, as it converts the date into two different integer values. If *date_column* is indexed, the index becomes unusable because an index on a date column relies on a chronological order to store the indexed values. In such a case, the proper way to write the query is to replace the conditions by a range condition and say that the values of *date_column* must comprise between June 1 and June 30 (inclusive) of the current year. In that case you are comparing a date column to date values, and you can use the index to locate the first row at or after June 1 and then collect the locations of rows corresponding to the following dates, up to June 30.

Avoiding Implicit Conversions

You also sometimes have implicit conversions between, for instance, character columns that only contain digits and number constants.

Suppose that you store in a table student or employee numbers, stored as character strings. It makes sense to store them as characters; they don't represent amounts or quantities, just strings of digits. I told you at the beginning of this chapter that the value in the index that is associated to a locator is just the collection of bytes found in the column at the row indicated by the locator—and that the actual meaning of these bytes depends on the data type of the column. For instance, here is how 12345 is stored in Oracle, depending on the data type:

```
SQL> select 'Number' as datatype, dump(12345) as storage from dual
  2  union all
  3  select 'Varchar2', dump('12345') from dual;

DATATYPE    STORAGE
--------    ----------------------------
Number      Typ=2 Len=4: 195,2,24,46
Varchar2    Typ=96 Len=5: 49,50,51,52,53
```

If you store the *number* 12345 in Oracle, it will be internally represented by four bytes that will take, respectively, the values 195, 2, 24, and 46. If you store the *character string* '12345', internally it will be five bytes, each one corresponding to the ASCII code of a digit (in that case, 49 to 53). The internal representation of the number will be different in another DBMS, but in all cases the internal representation of a number and of a string of characters will be wildly different.

Now, imagine that you write in a query a condition such as

```
where studentid = 12345
```

If the DBMS looks in the index for the four bytes 195, 2, 24, and 46, it won't find them—or if it ever finds the bytes corresponding to a number, those bytes will not correspond to the representation of the number as a string. The DBMS cannot compare apples and oranges; it knows that the *studentid* column stores character strings and that it is compared to a number. It might have been designed to say, "If people compare values of different types, return an error." However, SQL was initially designed as a user-friendly language for non-developers—even if that was wishful thinking. The user-friendly way to cope with impossible comparisons is to convert one of the two values so that we can compare values with identical data types. Which one should we convert? We could say, "Let's convert 12345 to a string," and run under the hood

```
where studentid = '12345'
```

Then, we have a major problem: What if the student identifier was entered as 0000012345 in the table? It's not uncommon to pad with zeros to the left string identifiers composed of digits; it allows them to be sorted properly, and not to have student 2345 appear *after* student 12345. If you brutally take a string equivalent of a number, values that are numerically equal will no longer be so: 100 is the same as 100.00, but '100' isn't the same as '100.00.'

If the student identifier was entered as '0000012345' and we search for '12345,' the query will tell you "not found," which is true at the byte level but not from a real-life standpoint. Not user friendly. So, the reasonable thing to do is to convert the string value found in the column to number. It's not riskless: we may encounter values that aren't entirely composed of digits and get a conversion error, but then we will have tried, at least, and the user will be warned. Better to say, "We couldn't convert" than to give a wrong answer, as with the opposite conversion. Unfortunately, with the conversion the index will become moot as the byte order of values will change, and it will become impossible to descend the index tree.

So, the comparison will result in a full table scan, because the philosophy of a DBMS is "better to be slow and right than fast and wrong"—which makes sense. In such a case, a developer should enter the value as a correctly formatted string (as stored in the table). Of course, what happens here between column and constant can also happen when comparing two columns of different data types, in a join for instance.

Avoid data type conversion when possible, and if you need it, make it explicit with a function such as *cast()*; don't let it be implicit. You'll understand better why an index isn't used when you see the function.

In all cases, when an index cannot be used, it boils down to the fact that the tree that was built over the list of (value, row locator) pairs can no longer be descended efficiently.

Indexing Expressions

The incompatibility between applying functions to columns in search conditions and using indexes is often a major hindrance. You can store some data in uppercase to avoid calls to *upper()* in the queries because it's possible to massage data on retrieval to make it look better and because (most often) converting to uppercase doesn't lose any information. Unfortunately, if you want to use function *soundex()*, which we have seen in Chapter 6, to run an approximate search on the names in table *people*, then you have a big problem because you cannot reconstruct a name from the soundex value; actually, *soundex()* was precisely devised so that many names may have the same soundex value. If, for instance, you want to find people who have a name sounding like *Stuart*, you will find yourself writing

```
where soundex(surname) = soundex('Stuart')
  and ...
```

which cannot use the index on the surname. One option is to add another column to the table, say *surname_soundex*, and either insert the soundex value of the surname each time you insert a row or, if you cannot modify all the programs that insert actors and directors, populate the column with a trigger each time a new person is added to the *people* table (solutions that involve triggers are always complicated and shouldn't be your first choice for solving an SQL problem). You can then index this column and write

```
where surname_soundex = soundex('Stuart')
  and ...
```

An expression referring to *surname_soundex* (no function here) could use the index on the new column and allow searching the table very quickly.

This solution isn't completely satisfying; with the soundex value, we are managing data that end users will never see (it will only appear in *where* clauses of queries). If we ever need to correct a misspelled surname, we may have to change the soundex value as well, and clearly the soundex value is redundant information that adds nothing to what we already know: the soundex value of the surname is fully determined by the surname. We are violating here, only for performance reasons, Bill Kent's rule that non-key attributes (such as the soundex value) must depend on the full key that identifies a row in table *people*, even if in queries we may use *peopleid*, surname plus first name compose the real-life key. The surname is only a part of the key.

When no other possibility is available, bending the rules of good design can be a solution. There may be, however, a much better way that conciliates clean database design and performance. Some products, but not all, allow creating an index on the result of an expression or a function, for instance,

```
create index people_surname_soundex_index on
people(soundex(surname));
```

When such an index exists, if you write

```
where soundex(surname) = soundex('Stuart')
  and ...
```

the SQL engine will recognize the same expression as was used to create the index and use this index, thus locating the rows with the correct *soundex()* value extremely fast.

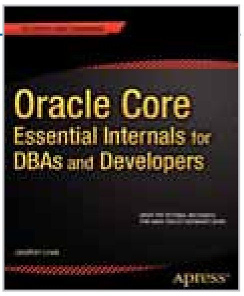
Creating an index on the result of a function isn't always possible; the expression or function has to be *deterministic*. What does "deterministic" mean? Simply that when you call the function several times with the same parameters, it should return the same result. Always. This is the case with a function such as *soundex()*, but a surprisingly high number of functions are not deterministic, especially date conversion functions, because they depend on machine or database settings. Typically, a function such as *datename()*, a SQL Server function that may return, among other things, the name of the month, is not deterministic, because if you change the language settings,

(continued on page 24)

Oracle Core: Essential Internals for DBAs and Developers

A Book Review by Brian Hitchcock

Details



Authors: Jonathan Lewis

ISBN: 978-1-4302-3954-3

Pages: 280

Year of Publication: 2011

Edition: 1

List Price: $39.99

Publisher: Apress

Overall Review: Excellent, concise coverage of Oracle database fundamentals.

Target Audience: Anyone interested in Oracle RDBMS internals.

Would you recommend this book to others: Yes.

Who will get the most from this book? DBAs.

Is this book platform specific: No.

Why did I obtain this book? See overall interview below.

Overall Review

I attended Jonathan's presentation at Oaktable World (also known as Oracle Closed World), and copies of this book were given to all attendees. While I hadn't planned to read a book about Oracle database internals, the topic has always interested me. I was also intrigued because this book is not huge. The typical Oracle book these days looks more like a phone book or a doorstop. Many books claim to cover everything you need to know and are just too long. I was eager to learn the most important concepts about Oracle internals but, at the same time, I didn't have any interest in trying to read a long book.

It has been a long time since I have needed to look at performance issues in the database. With applications getting more and more complex, many performance issues move to the middle tier, well removed from the database. Therefore, I had no immediate need to read this book. But, as so often happens, one door closes and another door opens. The only good approach to this aspect of life in the real world is to move quickly enough to avoid getting squished in the closing door—and to get through the open door before it, too, closes and you are left in the hallway all alone. Before I finished this review, I was reassigned to a new group, and within a week I was asked to look at a performance issue that I found was being caused by ITL waits. This is the universe's way of reminding us to keep an open mind. You don't know what will be useful today, tomorrow, next week—or in your new group!

Many times while I was reading this book, I simply wrote in my notes "I never thought of that!" There is a lot of useful information in this refreshingly short book.

Introduction

I liked the author's explanation for why he wrote this book. He explains that while you can find information about parts and pieces of Oracle, *"What you won't find is a cohesive narrative that puts all the right bits together in the right order to give you a picture of how the whole thing works and why it has to work the way it does."*

I also like this summary of what you will learn as you read this book. To quote the author, *"[I]t boils down to undo, redo, data caching, and shared SQL."* It is refreshing to find an author who doesn't use the complexity of Oracle to confuse the audience. Yes, Oracle is very complicated, but there are basic concepts that you need to know and that you can use to understand the specific parts of Oracle you have to deal with every day.

The author provides links to more detailed sources, so you can learn more if you want to, as well as updates to each chapter online.

Chapter 1—Getting Started

The author explains how he will begin and what will be discussed in what order. First up is a discussion of Oracle processes. There is good detail here. For example, there is a location in SGA that serves as a clock that coordinates the activity in the instance. This is the System Change Number (SCN). I liked calling the SCN a "clock." This made me think about SCN from a different perspective.

Next up is a great summary of what happens when a user submits a query to the database. This is what you really need to keep in mind when working with Oracle. I want to quote this paragraph in its entirety: *"An end user sends requests in the form of SQL (or PL/SQL) statements to a server process; each statement has to be interpreted and executed; the process has to acquire the correct data in a timely fashion; the process may have to change data in a correct and timely fashion; and the instance has to protect the database from corruption."* This concise overview is a good way to tie together everything that will be covered in the following chapters.

Chapter 2—Redo and Undo

We start out with an interesting question: what is the most important bit of Oracle technology? The answer is not something new from marketing. It is the change vector! This is the way Oracle describes the changes to data blocks, and it's what enables both redo and undo.

This is also what allows Oracle to minimize the conflict between readers and writers. This has been hugely important to Oracle marketing over the years: writers don't block readers … or is it the other way around, or both?

The way Oracle approaches changing data is not what you think. It is reasonable to expect the process to find the data and then change it. The four-step process is described that will create the following:

First, a description of how to make the change (redo change vector). Then, a description of how to reverse the change (undo record). Next, a description of how to create the description of how to reverse the change and, finally, change the data (redo change vector).

If this sounds confusing, I encourage you to read the full text of this discussion. It very much changed my understanding and appreciation of undo and redo.

A detailed example is given that includes symbolic dumps of the data block as the process of changing data progresses through all four steps. I'm not much good with symbolic dumps, but it's all here for those that like that sort of thing.

Another example of things in this book that I had not thought about before is that Oracle keeps two copies of everything: one copy in the data files and another copy in the redo log files. The figure provided in this section really helped me see this.

ACID properties (Atomicity, Consistency, Isolation, Durability) are discussed in detail. Why the undo mechanism enables concurrency (readers and writers not blocking each other) is explained. The generation of redo is a very simple mechanism in that, basically, you "write and forget." A feature of redo that I'd never heard about before is (starting in 10*g*) private redo and in-memory undo. Until a transaction completes, all change vectors are stored in private redo, and only at commit does the transaction need the redo log allocation latch to write redo.

The details of the complexity of the rollback process are presented. This has bothered me for years. Why does it take so long to roll back a transaction? It turns out that more buffer visits on undo blocks are needed when rolling back than when the transaction was initially executed. Now I have an answer for my users when rolling back takes a long time.

Chapter 3—Transactions and Consistency

The changes made by different users must be kept separate until their transactions commit, and the database must be able to quickly change which changes are visible to which users.

The reason rollback can cause long db startup time (after shutdown abort for example) is further explained: *"Since rolling back real changes is (or ought to be) a rare event compared to committing them, Oracle is engineered to make the commit as fast as possible and allows the rollback mechanism to be much slower."* Think about it … this tells us that rolling back is not what Oracle is built to do fast. If you roll back a long-running transaction, don't be upset that it takes a long time. This explains why rollback is so slow.

Another detail about rollback that I had never thought about is presented as follows: *"It isn't commonly realized, by the way, that when Oracle has applied all the relevant undo records, the last thing it does is update the transaction table slot to show that the transaction is complete—in other words, it commits."* Even rolling back requires a commit. It makes sense once it is explained. These are very interesting details of rollback, stuff I don't think you'll find anywhere else. I also learned how a single undo block can contain undo records for multiple transactions. This happens when a transaction commits and the free space left in the last UNDO block can be assigned to another transaction.

The Interested Transaction List (ITL) is explained. The ITL exists to identify transactions that recently changed a data block. An interesting bit of Oracle trivia is that a value of c1 02 is Oracle's internal representation of decimal 1. LOL!

Another new concept: Consistent doesn't mean historic. While discussing consistency, a detailed example shows how we sometimes construct a version of the block that has never actually existed! This is necessary to support read consistency. Also described is how to compute the work done to maintain read consistency. It turns out that Flashback query does all the work of read consistency and a lot more of it.

In the discussion of Commit SCN, there is this interesting detail: Sometimes Oracle shares the work among multiple sessions. For example, if a large number of blocks need to be cleaned up, this would take a long time for a single session to complete. Some of this work will be done by each of the next bunch of sessions, even though these sessions may not have caused these blocks to need cleanup.

In the section covering Commit Cleanout, it comes out that checkpoint makes Oracle copy any dirty blocks to disk, but it doesn't make Oracle remove them from the buffer cache.

In the section on ORA-1555, we have this comment: *"If you don't know that Oracle error 1555 translates into "snapshot too old" you can't be a real DBA."* This caught my attention because when I first started working on Oracle Applications, I was advised to stay away from that because "it isn't real DBA work"—and now I've worked with Fusion Applications, so I'm really far removed from being a "real" DBA!

Finally, in a section on LOBs, something that has always confused me is explained. There are special methods for handling undo and redo on LOBs. The discussion is very good, and I recommend that you read this part. Also, the summary is great: *"Essentially, Oracle doesn't update LOBs."* I have always wondered about this.

Chapter 4—Locks and Latches

This chapter provides a very clear and concise explanation of locks and latches. Locks are polite and tend to be held for long time. Latches are pushy and should be held very briefly. See? That was clear and concise! Pins and mutexes are also covered. Pointers, linked lists, and hash tables are explained to support the discussion of locks and latches. There is a good explanation of what could happen to linked lists if we didn't have locks and latches. Despite the often-quoted Oracle marketing line that "readers don't block writers" and the other way around, the author tells us, *"[W]hen you get down to the raw memory level, there are moments when readers must block writers, and where a single write must block all other operations."* Wow! The things you can think!

I found the explanation of what a latch is to be fascinating. *"Essentially a latch is the combination of a memory location in the SGA and an atomic CPU operation that can be used to check and change the value of that location."* An atomic CPU operation is explained as a single CPU instruction that can both check and change a location in memory. Further, multiple CPUs mean that any set of operations can be interrupted, hence the need for a single CPU operation that is "atomic." I've never had the latch and the CPU operations explained like this.

The detailed explanation of how this would all break down if we didn't have an atomic CPU operation is great. The drawbacks of exclusive latches and how Oracle deals with this from 9*i* is another "wow" moment, as are the examples of what Oracle writes in memory location for a latch.

It's interesting how things change over time . . . or don't. A specific example of this is the choice of units for wait event times. Measuring wait event time in 1/100th of a second started with Oracle 6, when CPUs that ran a few MHz were "fast."

The description of how the wakeup mechanism works (instead of just "sleeping") is great. Specific advice on how to prevent long wait time for latches is provided. The explanation of the difference between a lock and an enqueue provided insight into something that has always confused me.

Based on the author's experience, the truth about deadlocks—versus the official story that one process is chosen at random to be killed—is that the longest waiting session gets killed.

Most useful to me is the message that despite all the talk about deadlocks, there is no perfect solution for handling deadlocks. If I had a nickel for every time I had to explain to a user that there wasn't anything I could do about the deadlocks that simply happen in their code, I'd be retired somewhere far, far away from Redwood Shores. I never thought before about one more aspect of deadlocks—that they are not limited to two sessions.

Chapter 5—Caches and Copies

I greatly appreciate the specific advice the author provides, including this example: *"Personally I prefer to see 8KB as the size for the default as this is the size that sees most testing in the field and is generally the option least likely to result in odd problems."*

Good advice: keep it simple! Another example regarding db_writers_processes: *"[C]ontrary to frequent comments on the Internet, this is a parameter that rarely needs to be adjusted on a production system."* So much time is spent (wasted?) by "experts" endlessly debating the merits of configuration changes that, in reality, have little or no effect at all.

Chapter 6—Writing and Recovery

I was confident that I understood the recovery process, but I learned some new things in this chapter. The database writer will not write a changed block before the log writer has written the redo that describes the changes. This write-ahead logging is critical to the recovery mechanism. The redo logs (online and archived) are the definitive version of the database. And the part I hadn't thought about before is that the datafiles are just a recent, approximate snapshot of the database. I had not seen such a detailed discussion of what triggers the log writer to start writing.

Now we have one of the very few times I was not completely happy with this book. I read the following statement: *"PL/SQL doesn't always wait for the write to complete."* Really? I want to (need to?) believe that the redo is written before anything else happens. I want to hear more about this. Isn't this a big concern? This is followed by a description of an ACID anomaly. Again, I would think the issue the author describes would be a big deal, but the author doesn't seem upset, although I remain concerned.

I had never seen anything about how Oracle "wastes" redo space in certain circumstances. The author explains that when writing the redo log to disk, Oracle "never looks back" to keep the code as simple as possible, and this explains why, sometimes, Oracle doesn't make full use of all the available redo log space. The discussion of the database writer versus the log writer includes what happens when a block has changed so recently that the redo log for this latest change hasn't been written. One new (to me) aspect of checkpoints is that a query can cause a checkpoint. I didn't know that.

Related to redo logs, the author expresses concern over subtle changes that may have been made to support standby databases. I found this to be very interesting. While discussing Flashback Database, the author points out, *"The opportunities offered by inventive use of the redo logs are wonderful."* I agree.

Chapter 7—Parsing and Optimizing

This chapter starts with interesting trivia. The dictionary cache is also called the "row cache" because, in the past, it used to cache individual rows of data from the data dictionary instead of data blocks.

I had never heard of bootstrap objects. These are how Oracle knows how to find the first things it needs to know from the database to find out about everything else in the database. Various aspects of parsing are discussed, including an explanation of what a parse call is and the sequence of activity for parsing. Also presented are how cursor caching is done and issues around closing cursors.

Here is an amusing item from the discussion of the details of the extents of the shared pool: *"[E]ach freeable chunk will be linked to a recreatable chunk and the owner of the recreatable chunk is responsible for freeing the linked freeable chunks when it frees the recreatable chunk."* Another wow!

Chapter 8—RAC and Ruin

First, I am simply quoting the chapter title. Any association between RAC and the word "ruin" is strictly the author's choice. The author explains: *"I felt the need for a little alliteration in the chapter title, and it is very easy to ruin things if you don't understand a little bit about how RAC works; but RAC*

(continued on page 24)

What's So Sacred About Relational Anyway?

by Iggy Fernandez

Iggy Fernandez

"Bring the past for judgment into the thousand-eyed present, and live ever in a new day."

—Ralph Waldo Emerson

First, a short but fun quiz. The answers are at the end of the article.

1. Before relational databases, there were network databases. Network databases were codified by the Conference on Data Systems Languages (CODASYL) in 1969. According to the website of a prominent software company: *"CODASYL DBMS is a multiuser, CODASYL-compliant database management system for OpenVMS operating systems. CODASYL DBMS is designed for databases of all levels of complexity, ranging from simple hierarchies to sophisticated networks with multilevel relationships. CODASYL DBMS provides a reliable operating platform for application environments where stability, high availability, and throughput are essential."*

   CODASYL DBMS was created by Digital Equipment Corporation. It continues to be supported by:

   a) Oracle Corporation

   b) IBM

   c) Computer Associates

   d) Software AG

2. According to the IBM website:

   ➤ DBMS "X" manages a large percentage of the world's corporate data;

   ➤ Over 95% of Fortune 1000 companies use DBMS "X;"

   ➤ DBMS "X" manages over 15 petabytes of production data;

   ➤ $2.5 trillion is transferred through DBMS "X" by one customer every day; and

   ➤ DBMS "X" can process 21,000 transactions per second.

   DBMS "X" is:

   a) A pre-relational DBMS that helped put the first man on the moon

   b) A relational DBMS

   c) An object-oriented DBMS

   d) A NoSQL DBMS

3. Indian Railways is the world's second-largest railway, with 6,853 stations; 63,028 kilometers of track; 37,840 passenger coaches; and 222,147 freight cars. Annually it carries some 4.83 billion passengers and 492 million tons of freight. Of the 11 million passengers who climb aboard one of 8,520 trains each day, about 550,000 have reserved accommodations. Their journeys can start in any part of India and end in any other part, with travel times as long as 48 hours and distances up to several thousand kilometers. The challenge was to provide a reservation system that can support such a huge scale of operations—regardless of whether it's measured by kilometers, passenger numbers, routing complexity, or simply the sheer scale of India. (Source: HP website.)

   What sort of DBMS was used by Indian Railways in 2000 to build a state-of-the-art passenger reservation system on HP OpenVMS AlphaServer systems?

   a) No DBMS was used

   b) A pre-relational DBMS

   c) A relational DBMS

   d) An object-oriented DBMS

   e) A NoSQL DBMS

The surprising answers to the above questions should make us stop to think. As relational practitioners, we need to understand why the relational model is sacred. If we cannot explain why the relational model is sacred, we cannot hope to convert the unbelievers, can we?

So stick with me as I attempt to explain.

Simplicity and Naturalness

Dr. Codd personally believed that the chief advantage of the relational model was its simplicity and consequent appeal to users (especially casual users) who have little or no training in programming. He singles out this advantage in the opening sentence of his very first paper on relational theory, *A Rela-*

tional Model of Data for Large Shared Data Banks, faithfully reproduced in the 100th issue of the *NoCOUG Journal* (down to the misspelling of the city name Phoenix in the References section): *"Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation)."* He makes the point very forcefully in a subsequent paper, *Normalized Data Base Structure: A Brief Tutorial*: *"In the choice of logical data structures that a system is to support, there is one consideration of absolutely paramount importance—and that is the convenience of the majority of users. . . . To make formatted data bases readily accessible to users (especially casual users) who have little or no training in programming we must provide the simplest possible data structures and almost natural language. . . . What could be a simpler, more universally needed, and more universally understood data structure than a table? Why not permit such users to view all the data in a data base in a tabular way?"*

But does the appeal to users (especially casual users) who have little or no training in programming make relational sacred to computer professionals? Should computer professionals like you and me be protected from having to know how the data is organized in the machine? Will we develop high-performance applications if we are ignorant about those little details? If your answers are in the negative, then read on.

Computational Elegance

Dividing 3704 by 14 is a lot easier than dividing MMMDCCIV by XIV (Roman notation), wouldn't you agree? The computational elegance of the relational model is unquestionable. The co-inventor of the SQL Language, Donald Chamberlin, reminisces: *"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."* (The 1995 SQL Reunion: People, Projects, and Politics)

But is computational elegance the holy grail of computer professionals? Is it the be-all and end-all of application software development? If your answers are in the negative, then read on.

Derivability, Redundancy, and Consistency of Relations

The true importance of relational theory is highlighted by the title of the original (and considerably shorter) version of Codd's first paper. That version predated the published version by a year, and the title was *"Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks."* The title of this unpublished version emphasizes that the real importance of relational theory is that it provides a rigorous method of asserting consistency constraints that must be satisfied by the data within the database. For example, the following assertion written in Structured Query Language (SQL) states that the company must have at least 50 employees:

```
CREATE ASSERTION employees_a1 AS CHECK (
  (SELECT COUNT(*) FROM employees) >= 50
)
```

In practice, the above consistency check would have to be implemented using a database trigger, since relational vendors do not support the CREATE ASSERTION feature.

As another example, the following "referential integrity constraint" links the Employees table to the Departments table:

```
CREATE ASSERTION employees_departments_fk AS CHECK (
  NOT EXISTS (
    SELECT * FROM employees e
    WHERE NOT EXISTS (
      SELECT * FROM departments d
      WHERE d.department_id = e.department_id
    )
  )
)
```

In practice, referential integrity constraints are implemented using simplified syntax that obfuscates the theoretical underpinnings. The database administrator or application developer need simply say:

```
ALTER TABLE employees ADD CONSTRAINT employee_departments_fk
FOREIGN KEY department_id REFERENCES departments
```

Strange as it may sound, Oracle Database did not enforce referential integrity constraints until Version 7 was released in the 1990s (by which time Oracle Corporation was already the world's largest database company). From the January 1989 issue of *Software Magazine: "About six or seven years ago when I worked for a vendor that made a Codasyl DBMS called Seed, I spoke at a conference. Also speaking was Larry Rowe, one of the founders of Relational Technology, Inc. and one of the developers of the relational DBMS Ingres. We were about to be clobbered by these new relational systems. He suggested to me that the best way to compete against the relational systems was to point out that they did not support referential integrity. Well, back then, virtually no one understood the problem enough to make it an issue. Today, Codasyl DBMSs are an endangered species, and referential integrity is a hot topic used by the relational DBMSs to compete among themselves."*

So there you have it. Relational is sacred because it gives application software developers the ability to assert and enforce consistency of data in databases.

Now go tell everyone! But before you leave, here's a little tidbit that may really surprise you—something to tell your NoSQL friends.

Eventual Consistency!

Dr. Codd was acutely aware of the potential overhead of consistency checking. In the concluding section of his famous paper, he says: *"There are, of course, several possible ways in which a system can detect inconsistencies and respond to them. In one approach the system checks for possible inconsistency whenever an insertion, deletion, or key update occurs.* ***Naturally,***

such checking will slow these operations down. *[emphasis added] If an inconsistency has been generated, details are logged internally, and if it is not remedied within some reasonable time interval, either the user or someone responsible for the security and integrity of the data is notified. Another approach is to conduct consistency checking as a batch operation once a day or less frequently. Inputs causing the inconsistencies which remain in the data bank state at checking time can be tracked down if the system maintains a journal of all state-changing transactions."*

This approach can be dubbed "eventual consistency" and is used by NoSQL systems; that is, distributed replicas may not be in sync. It is ironic that a principle elucidated by Dr. Codd in the first paper on relational theory should have found its first application in systems that *reject* relational theory. ▲

Answers to quiz questions: 1 (a) Oracle Corporation; 2 (a) IMS; 3 (a) No DBMS was used.

BOOK EXCERPT *(continued from page 18)*

then the name that will be returned will change. The problem is that indexed values are stored in the index. If some settings are modified and, all of a sudden, applying a function to a column no longer returns the same value as the pre-calculated value that was stored in the index, it becomes impossible to retrieve the data. If you could index the function that returns the name of a month (SQL Server will prevent you from doing so) you would store in the index key values that a different language setting would make irrelevant. Clearly, this isn't acceptable: the purpose of indexes is to provide an answer faster, not to change the answer by saying *no data found* (or perhaps, more to the point, something such as *¡No se ha encontrado ningun dato!*) when you search for Enero in an index that was built when the current language was English and refers to a lot of rows for *January*.

Note: *Enero* is Spanish for *January*.

You can be on shifting ground even with date functions that return numerical values, which you might believe to be language neutral. For instance, the number of the day in a week is counted differently in different countries. If you ship to several countries a program that uses the number of the day of the week, you cannot guarantee that the behavior will be the same everywhere. ▲

Stéphane Faroult first discovered relational databases and the SQL language back in 1983. He joined Oracle France in their early days (after a brief spell with IBM and a bout of teaching at the University of Ottawa) and soon developed an interest in performance and tuning topics. After leaving Oracle in 1988, he briefly tried to reform and did a bit of operational research, but after one year, he succumbed again to relational databases. He has been continuously performing database consultancy since then, and founded RoughSea Ltd in 1998. He is the author of The Art of SQL *[O'Reilly, 2006] and* Refactoring SQL Applications *[O'Reilly, 2008].*

BOOK REVIEW *(continued from page 21)*

doesn't necessarily lead to ruin." In this chapter, the author focuses on what we need to understand about RAC, which he tells us is global enqueues and cache coherency.

I was pleased to read that the difficult part of RAC is getting it installed and running in the first place. There is a great diagram and list of key points describing RAC. The explanation of virtual IP addresses was very good.

After this introduction to RAC, the big question about RAC is put forward: *"But it's complicated and why would you want to deal with something complicated?"* We are told to remember that complexity also means more people, more downtime, etc. With this background, the author explains the benefits of RAC and provides good insights, one of which is the following: *"[T]he rate at which an instance can handle redo generation is the ultimate bottleneck in an Oracle system."* One side effect of a big (RAC) system is that it has so much overcapacity that it hides performance issues until it's too late.

I've read a lot of things about the overhead of RAC so I was pleased with the clarity I found here. For RAC, once you get to three instances, the level of overhead doesn't get any worse.

The Global Resource Directory (GRD) and cache fusion are explained, and a diagram explaining how more nodes mean more interconnect traffic is presented. A good discussion covers details of how RAC handles sequences and caches them. This includes issues around sequences, auditing, and scalability.

This chapter ends with a very important summary of RAC and the warning that moving to RAC can cause the performance of badly designed applications to become worse.

Appendix—Dumping and Debugging

Here the author explains how he used Oradebug to investigate and demonstrate the inner workings of Oracle.

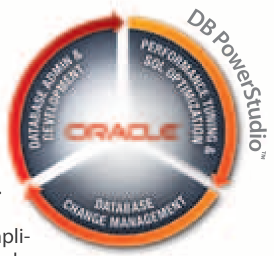
Conclusion

I appreciate the author's brevity. He quickly gets to the point. There is always more detail, but the crucial concepts can be illustrated quickly and efficiently, as they are in this book. While reading, there were many times when I wrote in the margins "I never thought of that," "I've always wondered about that," and "Wow!" I highly recommend that you read this book, no matter what your level of Oracle expertise may be. ▲

Brian Hitchcock worked for Sun Microsystems for 15 years supporting Oracle databases and Oracle Applications. Since Oracle acquired Sun, he has been with Oracle supporting the On Demand refresh group and, most recently, the Federal On Demand DBA group. All of his book reviews and presentations—and his contact information—are available at **www.brianhitchcock.net**. ***The statements and opinions expressed here are the author's and do not necessarily represent those of Oracle Corporation.***

Oracle Database Administration, Development, and Performance Tuning...
Only Faster.

Taking care of your company's data is an important job.
Making it easier and faster is our's.


DB PowerStudio™
PERFORMANCE TUNING & SQL OPTIMIZATION
DATABASE ADMIN & DEVELOPMENT
ORACLE
DATABASE CHANGE MANAGEMENT

Introducing DB PowerStudio for Oracle. It provides proven, highly-visual tools that save time and reduce errors by simplifying and automating many of the complex things you need to do to take care of your data, and your customers.

Whether you already use OEM or some other third-party tool, you'll find you can do many things faster with DB PowerStudio for Oracle.

> **Easier administration** with DBArtisan®
> **Faster development** with Rapid SQL™
> **Faster performance** with DB Optimizer™
> **Simplified change management** with DB Change Manager™

Go Faster Now. >>> Get Free Trials and More at **www.embarcadero.com**

Don't forget Data Modeling! Embarcadero **ER/Studio**®, the industry's best tool for collaborative data modeling.

embarcadero®

Database Monitoring and Performance Analysis

Only Ignite delivers a complete picture of database performance from an end user perspective. It analyzes response time, queries, sessions, and server resources to show both historical and real-time performance.

Go to www.Confio.com and try Ignite 8 today!

CONFIO
Confio Software, Boulder Colorado.

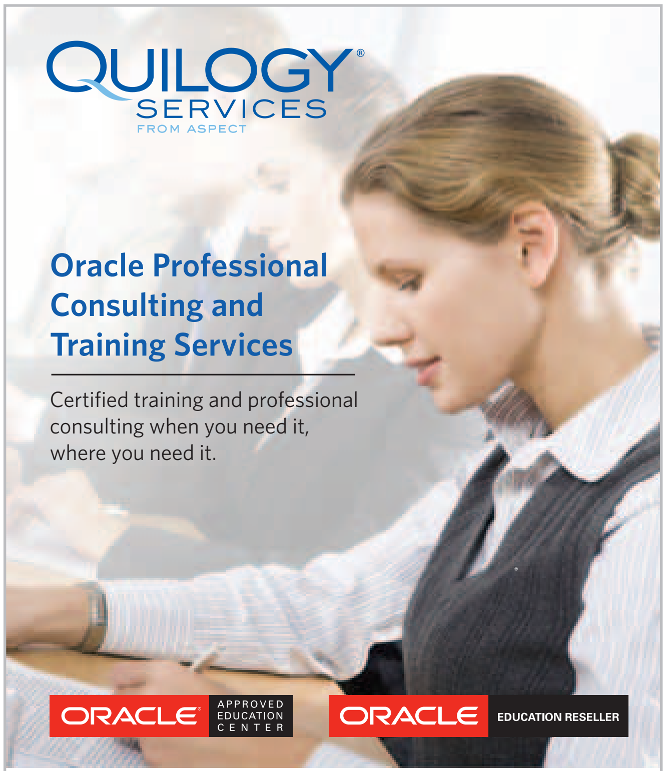
DELPHIX®

Database Virtualization Software

- Consolidate Infrastructure.
- Instantly Provision and Refresh.
- Maximize Performance.

www.delphix.com

QUILOGY® SERVICES
FROM ASPECT



Oracle Professional Consulting and Training Services

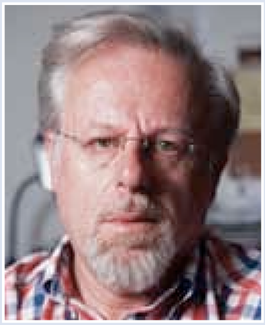
Certified training and professional consulting when you need it, where you need it.

ORACLE APPROVED EDUCATION CENTER
ORACLE EDUCATION RESELLER

www.quilogyservices.com
education@aspect.com
866.784.5649

Aspect®

"Unstructured Data"—A Contradiction in Terms


Fabian Pascal

by Fabian Pascal

Suppose I asked you what the following means:

41onMastrsecrahM7719
norerF2i1dkuJcysl1795
enr9e1b682ebtlrs8tAomepS

You are probably at a loss. How about this?

Masterson	14 March 1977
Fredrickson	21 July 1975
Albertson	28 September 1968

I assure you that in both cases, the number of characters and the characters themselves are the same. What's the difference?

In the first case we have randomized strings of letters and digits. This is what is referred to as *noise*. In the second case, the letters and digits are *organized* in some way; there is a *structure*. Note, though, that the structure allows us only to suspect there is some *meaning*—we still don't know what that is. We can tell there are names and dates—this is what the elements mean *individually*—but we still don't know the meaning of the whole.

Consider now the following structure:

NAME	DATE
Masterson	14 March 1977
Fredrickson	21 July 1975
Albertson	28 September 1968

What does the table mean? We still can't interpret it beyond the individual meanings. Different readers could and probably would give it different interpretations, depending on the *context* in which they used it. For example, in the context of hiring employees, the table could mean "Employee named NAME was hired on date DATE"; in the context of hospital births, it could mean "Baby named NAME was born on date DATE." The capitalized terms are placeholders for specific data values appearing in the rows. What can we conclude from this?

Noise is unstructured and, therefore, meaningless. Data, on the other hand, has meaning and, therefore, is *by definition* structured, or organized in some way. But as we have just seen, while there is no meaning without structure, the meaning is *not in the structure itself*: neither structure—text or table—*by itself* allows us to interpret what it means.

So what is, then, "unstructured data"? Well, essentially, a contradiction in terms. Either there is *some* structure and, therefore, data, or there is noise. The issue is never *whether* data is structured or to what degree (e.g., "semi-structured"), but *what structure is most useful for a given informational purpose*. As *mis*used in the industry, "unstructured data" means data not structured *in R-tables*, data of an *unknown* structure, or data with an existent structure that we don't want to alter.

The core function of database management is information retrieval: answering queries while ensuring data **consistency** and *provably logically correct results*. This involves the **manipulation** of data, and different structures are amenable to different kinds of manipulation and yield different types of results—with varying degrees of soundness and cost-effectiveness and, therefore, usefulness for a given informational purpose.

No manipulation can retrieve information from noise—it does not carry any, and there is no soundness to speak of. But soundness is critical for data, and it is their structure that determines what manipulation is possible and, therefore, their usefulness for a given purpose. Text, video, graphics, tables—they are not noise but different data structures with different manipulative properties.

Structure and manipulation are what a **data model** consists of—another misused and abused term that needs debunking. Stay tuned. ▲

Fabian Pascal is an independent writer, publisher, lecturer and analyst, specializing in data fundamentals and the relational model for database management. He has authored three books and hundreds of articles for the trade press, and has lectured and taught educational seminars at the business and academic levels. He is the founder, in 2000, of DATABASE DEBUNKINGS, and its editor and publisher. He is also the author and publisher of the PRACTICAL DATABASE FOUNDATIONS series of papers, dedicated to making data fundamentals accessible to data practitioners without compromising theoretical rigor.

"Either there is some structure and, therefore, data, or there is noise. The issue is never **whether** *data is structured or to what degree (e.g., "semi-structured"), but* **what structure is most useful for a given informational purpose."**

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.

DatabaseSpecialists

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

NoCOUG
P.O. Box 3282
Danville, CA 94526

RETURN SERVICE REQUESTED

NoCOUG Spring Conference Schedule

Wednesday, May 22, 2013—California Center Pleasanton

Please visit **http://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:** Naren Nagtode, NoCOUG president
9:30–10:30	**Keynote:** *Schema, NoSQL, and Database Management*—Fabian Pascal
10:30–11:00	**Break**
11:00–12:00	**Parallel Sessions #1**
	Auditorium: *Oracle Database Appliance I/O and Performance Architecture*—Tammy Bednar, Oracle Corp.
	Tassajara: *Re-Engineering Your Database Using Oracle SQL Developer Data Modeler 3.1*—David Peake, Oracle Technology Network
	Diablo: *Hadoop for Oracle DBAs: Part 1*—Ahbaid Gaffoor, A9.com
12:00–1:00 p.m.	**Lunch**
1:00–2:00	**Parallel Sessions #2**
	Auditorium: *Re-Platforming with Zero Downtime Using Oracle RAC, CRS, and ASM*—Amit Das, PayPal
	Tassajara: *Testing and Debugging Procedures Using SQL Developer 3.1*—David Peake, Oracle Technology Network
	Diablo: *Hadoop for Oracle DBAs: Part 2—Pig, Hive, and HBase*—Ahbaid Gaffoor, A9.com
2:00–2:30	**Raffle**
2:30–3:30	**Parallel Sessions #3**
	Auditorium: *Policy-Managed RAC Databases—Why and How*—Mark Scardina, Oracle Corp.
	Tassajara: *Building an Application Using Oracle Application Express: Part 1*—David Peake, Oracle Technology Network
	Diablo: *Impala: A Modern SQL Engine for Hadoop*—Justin Erickson, Cloudera
3:30–4:00	**Break and Refreshments**
4:00–5:00	**Parallel Sessions #4**
	Auditorium: *Diving into the Latest Oracle Zero-Day Bugs*—Slavik Markovich, McAfee
	Tassajara: *Enhancing Your Oracle Application Express Application: Part 2*—David Peake, Oracle Technology Network
	Diablo: *TBD*
5:00–	**NoCOUG Networking and No-Host Happy Hour**

RSVP *required* at http://www.nocoug.org

The *NoCOUG Journal* design and production: Giraffex, Inc., S.F.
Front cover photo: California Golden Poppy, Kimberly Beck, Photos.com.