Official Publication of the Northern California Oracle Users Group

NoCOUG JOURNAL

Vol. 32, No. 1 · FEBRUARY 2018

Hard Words

Mogens Norgaard brings the past for judgment into the thousand-eyed present.

See page 4.

Oracle Licensing in the Cloud—Part IV

Comply or else.

See page 16.

12 Things to Love about 12*c*R2

The SQL advocate advocates.

See page 20.

Much more inside . . .

The New Phoenix by Axxana

For Oracle Databases and Applications

Zero Data Loss
at Unlimited Distances

Synchronous Protection
at Maximum Performance

- Low Cost Replication Lines
- Shortest Recovery Time
- Full Consistency Across Multiple Databases

AXXANA
BUILT TO LAST

www.axxana.com

ORACLE Gold Partner

ORACLE BIG DATA APPLIANCE READY
ORACLE DATABASE READY
ORACLE EXADATA READY
ORACLE LINUX READY

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 copy-editor 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, David Gonzalez at Layton Printing Services deftly brings the *Journal* to life on an offset printer.

This is the 125th issue of the *NoCOUG Journal*. Enjoy! ▲

—*NoCOUG Journal* Editor

Table of Contents

Interview 4

Brian's Notes 11

Special Feature 15

Special Feature 18

ADVERTISERS

Axxana 2

Quest 13

OraPub 28

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.*

2018 NoCOUG Board

Dan Grant
Exhibitor Coordinator

Eric Hutchinson
Webmaster

Iggy Fernandez
President, Journal Editor

Jeff Mahe
Vice President

Kamran Rassouli
Social Director

Larry Stimely
Member at Large

Liqun Sun
Brand Ambassador

Michael Cunningham
Director of Special Events

Naren Nagtode
Secretary, Treasurer, President Emeritus

Noelle Stimely
Membership Director

Roy Prowell
Director of Publicity and Marketing

Saibabu Devabhaktuni
Conference Chair

Tu Le
Speaker Coordinator

Volunteers

Tim Gorman
Board Advisor

Brian Hitchcock
Book Reviewer

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

Hard Words

by Mogens Norgaard

Mogens Norgaard

"I read the other day some verses written by an eminent painter which were original and not conventional. The soul always hears an admonition in such lines, let the subject be what it may. The sentiment they instill is of more value than any thought they may contain. To believe our own thought, to believe that what is true for you in your private heart is true for all men,—that is genius. . . . the highest merit we ascribe to Moses, Plato, and Milton is, that they set at naught books and traditions, and spoke not what men but what they thought. A man should learn to detect and watch that gleam of light which flashes across his mind from within, more than the lustre of the firmament of bards and sages. Yet he dismisses without notice his thought, because it is his. . . . to-morrow a stranger will say with masterly good sense precisely what we have thought and felt all the time, and we shall be forced to take with shame our own opinion from another."

—Ralph Waldo Emerson, mid-19th century American philosopher and poet.

Editor's note:** In the early days of Oracle RAC, the "enfant terrible" of the Oracle community, Mogens Norgaard, wrote a provocative paper titled "You Probably Don't Need RAC." The technology has matured and improved since the date of the paper and, therefore, a number of the technical details in the paper are no longer valid. However, the underlying message of the paper is that you need to make an informed decision, justify the increased complexity and cost, and consider the alternatives. In the August 2011 issue, we asked Mogens whether we "probably needed" such things as RAC, Exadata, Oracle 12c, MySQL, certifications, ITIL, NoCOUG, the* NoCOUG Journal*, and printed books. In this issue, we ask whether we probably need "data protection regulation" (the European Union's latest push for consumer privacy), the cloud, cryptocurrencies, the internet of things, and NoSQL. We also offered Mogens the opportunity to revise his previous opinions because, as so eloquently said by the American prophet of self-confidence and nonconformity Ralph Waldo Emerson, "The other terror that scares us from self-trust is our consistency; a reverence for our past act or word, because the eyes of others have no other data for computing our orbit than our past acts, and we are loath to disappoint them. . . . Why drag about this corpse of your memory, lest you contradict somewhat you have stated in this or that public place? Suppose you should contradict yourself; what then? It seems to be a rule of wisdom never to rely on your memory alone, scarcely even in acts of pure memory, but to bring the past for judgment into the thousand-eyed present, and live ever in a new day. . . .A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines. With consistency a great soul has simply nothing to do. He may as well concern himself with his shadow on the wall.* ***Speak what you think now in hard words, and to-morrow speak what to-morrow thinks in hard words again, though it contradict every thing you said to-day.*—'Ah, so you shall be sure to be misunderstood.'—Is it so bad, then, to be misunderstood? Pythagoras was misunderstood, and Socrates, and Jesus, and Luther, and Copernicus, and Galileo, and Newton, and every pure and wise spirit that ever took flesh. To be great is to be misunderstood."*

To blindly believe anything that Mogens says here would not win you any points with him—or with Emerson. As always, the challenge is to make an informed decision tailored to your unique circumstances while considering the alternatives. Do that and you will win his respect in spades. Even more if you prove him wrong.

"To blindly believe anything that Mogens says here would not win you any points with him—or with Emerson. As always, the challenge is to make an informed decision tailored to your unique circumstances while considering the alternatives. Do that and you will win his respect in spades. Even more if you prove him wrong."

"All those IoT and IIoT thingies create so much data that we can't even begin to imagine it. That data needs storage and computing to happen on it, and it needs it fast. That won't happen using the cloud. The LATEncy will kill it. So we need local compute and storage for this. Say hello to the periphery, the edge, or whatever."

Do we probably need a General Data Protection Regulation (GDPR)? What are Europeans trying to hide anyway?

GDPR is **awesome**. Law firms are hiring like crazy all over Europe because there will be *so* many lawsuits after May 25. Class-action stuff will be allowed, which is new. Yeehaw.

GDPR is **cool** because the citizens of the EU area become the owners of their own data on May 25, not Walmart or your doctor or Facebook. You can make deals with them, you might be able to sell your data to others, you get The Right To Be Forgotten, and so on. Although it's my understanding that a request from you to your tax authorities about being forgotten probably will not go through.

GDPR is **wonderful** because it creates a new market, including the initial Fear, Uncertainty, and Doubt, thus making everyone richer,

GDPR is **incredible**, because I have been on national television twice (at length), national radio four times (at length), and in most newspapers and hipster magazines in Denmark, just because I sent out a press release stating, "I want to buy your personal data." Also, some of my guys are currently busy creating a platform that allows citizens and companies to handle all sorts of stuff regarding this. We call it Datatrade.dk.

In my Datatrade.dk project, we see three groups of interested citizens: the ones that are flat out curious about what sort of data the companies have on them, the ones that just want to be forgotten (aka off-grid), and the ones that might be tempted to trade their personal data on the platform. Other needs will be met by the platform when it comes to companies: storing up-to-date information for them instead of them doing it, and an automatic pipeline to the law firms when a request is not met on schedule, etc.

Fun fact: Everything on paper should also be GDPR-compliant. So all those old CVs and papers and binders in your offices should be GDPR'ed before May 25. The general consensus among my friends is to throw most of it out (about time, as they say), but imagine public bureaucracies and their less-than-digital tendency to keep records of everything :-).

GDPR will be **yuuuuge**. It will require many "stable geniuses" in the coming years.

Embrace, embrace, embrace!

Do we probably need to move our databases to the cloud this week, this month, or this year?

Yes. This week might be too late. Better do it tonight while having a good beer or 12. Until some months ago I used to say that the business case for having your own server room is stone dead.

But as some Danish philosopher wisely said many years ago: "You're always wrong."

Turns out, just as the cloud is looking set to really, truly dominate our current planet, something else pops up, which will be *so* much bigger and more complex than the cloud: the fog, aka edge computing.

All those IoT and IIoT (yes, we tend to add an extra I every two months or so) thingies create so much data that we can't even begin to imagine it. That data needs storage and computing to happen on it, and it needs it fast. That won't happen using the cloud. The LATEncy will kill it. So we need local compute and storage for this. Say hello to the periphery, the edge, or whatever.

Listen to what *The Economist* wrote this week (January 19, 2018):

"General Data Protection Regulation (GDPR) is cool because the citizens of the EU area become the owners of their own data on May 25, not Walmart or your doctor or Facebook. You can make deals with them, you might be able to sell your data to others, you get The Right To Be Forgotten, and so on."

About Vapor IO, Nokia, and Whole Foods: *"But smaller and more local data centres are springing up everywhere. Firms such as EdgeConneX and vXchnge have built networks of urban data centres. Vapor IO, a startup, has developed a data centre in a box that looks like a round fridge and can be quickly put in any basement. Makers of telecoms equipment, including Ericsson and Nokia, as well as network operators, talk a lot about 'mobile edge computing', which amounts to putting computers next to wireless base stations or in central switching offices. Some also speculate that one reason why Amazon last year bought Whole Foods, a chain of grocery shops, for nearly $14bn, was to accumulate property for local data centres."*

About Microsoft and the Weather Company: *"Big cloud-computing providers are also trying to colonise the periphery. In May Microsoft changed its slogan from 'mobile first, cloud first' to 'intelligent cloud and intelligent edge'. It sells services that dispatch software containers with AI algorithms to any device. AWS's portfolio now includes a service called Greengrass, which turns clusters of IoT devices into mini-clouds. In buying the Weather Company* *for $2bn in 2015, IBM wanted weather data, but also thousands of 'points of presence' for edge computing."*

About Air Computing: *"Whoever prevails, computing will become an increasingly movable feast, bits of which can be found in even the smallest devices. Processing will occur wherever it is best placed for any given application. Data experts have already started using another term: 'fog computing'. But the metaphor is a bit, well, foggy. Better, and more poetic, would be 'air computing': it is everywhere and gives things life."*

"AWS's portfolio now includes a service called Greengrass, which turns clusters of IoT devices into mini-clouds. . . . Some also speculate that one reason why Amazon last year bought Whole Foods, a chain of grocery shops, for nearly $14bn, was to accumulate property for local data centres."

So I think we'll see a lot of NDCs (Nano Data Centers) spring up everywhere, and we'll have so much more complexity, which is fantastic. Problems drive our incredibly luxurious lives, and hence increased complexity is very welcome.

As for databases as you and I have known them, just get them out of there and into the cloud unless you have some weird requirements. Or get rid of them altogether. We Do Not Use Databases.

Do we probably need bitcoin? Alt-coins?

Cryptos are so cool. They make all sorts of freedoms possible, which is why the horror regimes are the first to outlaw them. As if that can stop it. We have a saying in Danish, "The cat is out of the sack," meaning there's no stopping it, now that people have seen it work. It's like outlawing ride sharing from Uber: alternatives spring up really fast. Cryptos are, just like any other currency, based on trust. That's all.

But what I really, really, really love is the blockchain. It represents the first time mankind has a tool that allows two strangers to completely trust each other. It will change more in our lives than the internet, and it will make things so much more efficient and cheap. For instance, a group of Wall Street banks estimate that it will give them $21 billion in pure cost savings. The blockchain attacks the center, not the periphery, so Uber, Airbnb, and all those other disrupters can and will be disrupted by this technology, but the drivers, the renters, and all the other citizens of our current planet will benefit immensely.

There will be many variations of the blockchain, some relying on smarter and faster algorithms or ideas, some giving us a little less trust in order to provide much greater speed, and so on. But again: The cat is out of the sack. We have now seen that it's possible to do the impossible. The generals' dilemma has been solved.

I've personally made a small fortune on BTC, by the way . . . and I'm about to launch a really cool service that will make me a bank director, which I always wanted to be.

My dad would be proud of me. He was in a small Danish bank in his youth before he joined the Danish Army. He'll be sitting in the cloud watching his weird, left-leaning son become a bank director!

Do we probably need the internet of things? Will Alexa rule them all?

As stated in the first question, I really appreciate stuff that complicates everything. It makes all of us so much richer.

I'm also launching a separate company that will address both Industrial IoT (IIoT) customers and ordinary citizens that just want their laptop backed up, their Wi-Fi to work in the basement, and hopefully also Hue lightbulbs, Alexa, smartwatches, Eve room monitors, the Nokia scale, and so much more.

I'll let bright, young, beautiful, and cheap people run around and set up, fix, and adjust all this in private homes and small businesses. I shall conquer the IoT home front!

Alexa might dominate, since Amazon consistently does all the right things. They've even made it easy to return stuff you bought now. A company like that will succeed.

But it will probably grow too big and will be forced to break up just like Standard Oil and AT&T. But I'm always wrong.

Do we probably need NoSQL? Please say no.

Yes. Ha!

It was probably a bad idea to put the word "No" in the name, just as Oracle really hated the "Not" in my RAC paper. But it's too late to change it to YeSQL.

Some of my guys are old, bitter, twisted idiots like you and me (old idiots are afraid of anything new and anything foreign, as you know), and yet they find interesting things to use NoSQL databases for.

Although all the redundancy drives them mad for reasons of principle.

And the young people use it, so that's the future by definition.

And it creates a bit more chaos and complexity because they are so different.

"We have now seen that it's possible to do the impossible. . . . The blockchain represents the first time mankind has a tool that allows two strangers to completely trust each other. It will change more in our lives than the internet."

"Some of my guys are old, bitter, twisted idiots like you and me (old idiots are afraid of anything new and anything foreign, as you know), and yet they find interesting things to use NoSQL databases for. Although all the redundancy drives them mad for reasons of principle. And the young people use it, so that's the future by definition."

There you have it. We need it. We Do Not Use Relational, unless forced.

[August 2011] Years ago you said that we probably don't need RAC. Have you recanted yet? Do we probably need RAC?

I still think very, very few shops actually need RAC. Fantastic technology—just like, say, head-up display (HUD) for cars—but few really need it. RAC still has all the hallmarks of something people will want to buy: It increases complexity immensely, it's expensive, it requires specialists that are increasingly hard to find, there are always excellent alternatives—and it's pretty much perpetually unstable. For all those good reasons, more and more customers are using it. Either because manly types like to increase chaos, or because I've been telling people not to use it since around the year 2000. Whenever I recommend or don't recommend something, most customers go out and do exactly the opposite, so in that sense I have a great deal of influence in the market.

[February 2018 update] MTAF/YAAW: Man Tager Altid Fejl/You Are Always Wrong.

Said by a Danish philosopher many moons ago. It's a hard truth. For instance, I recently learned, to my *big* surprise, that Denmark had nukes (for use on Nike/Hawk and Honest John missile systems, and on the F-104 Starfighters) during the Cold War. My dad was an officer. I spent many years in the National Guard as a sergeant. I never thought we were a nuclear nation, because we always told everyone that we weren't. But we were. I also recently learned that Churchill was totally against the invasion on D-day. So: you should expect to be proven wrong, to learn new details, etc. And you should rejoice when that happens, even if you've told 42,000 friends what you used to believe.

With that in mind, I have been forced—brutally, but not violently—by the *NoCOUG Journal* to revisit stuff I claimed with certainty seven years ago. Ouch.

I'm not even sure I have 20/20 hindsight. I don't think many have. But let's go through it:

On RAC: I cannot imagine anyone running RAC today, but I know they are. The Hadoop revolution took care of that. We now have those sub-zero response times; infinite, galactic scaling; and much better hair and personal lives, thanks to Mr. Cutting & Friends. However, to make up for all those upsides, it's also pretty much free.

[August 2011] Do we probably need Exadata? Is Big Iron the ultimate answer to the great question of life, the universe, and everything?

In some ways, Exadata is the new RAC. It's a lot about hardware, uptime, performance, amazing technology—and price. It's also approaching the "Peak of Inflated Expectations" as seen in Gartner's hype cycle, and it will soon set its course downwards toward the "Trough of Disillusionment." Just like with RAC, I simply *love* the technology—a lot of good guys that I like and respect are on it, but few really need it. One of the things I love about it is that there isn't any SAN involved, since I believe SANs are responsible for a lot of the instability we see in IT systems today. I tend to think about Exadata as a big mainframe that could potentially do away with hundreds of smaller servers and disk systems, which appeals hugely to me. On the other hand, the pricing and complexity makes it something akin to RAC—that's my current thinking.

"Alexa might dominate [the Internet of Things], since Amazon consistently does all the right things. They've even made it easy to return stuff you bought now. A company like that will succeed. But it will probably grow too big and will be forced to break up just like Standard Oil and AT&T."

[February 2018 update] On Exadata: See my update on RAC.

[August 2011] Do we probably need Oracle Database 12c (or whatever the next version of Oracle Database will be named)?

Since Oracle 7.3, that fantastic database has had pretty much everything normal customers need. It has become more and more fantastic; it has amazing features that are light years ahead of competitors—and fewer and fewer are using the database as it should be used (they're using it as a data dump, as Tom Kyte said many years ago), so the irony is that as the database approaches a state of nirvana (stability, scalability, predictability, diagnosability, and so forth—fewer and fewer are using it as it should be used (in my view), and more and more are just dumping data into it and fetching it.

[February 2018 update] On Oracle 18 through Oracle 42: Free, relational databases like MySQL and PostgreSQL, plus the whole NoSQL revolution, means that new and fantastic Oracle features will only be used by existing customers.

[August 2011] Do we probably need MySQL? Or do we get what we pay for?

As customers (and especially new, freshly faced programmers) want to use new things instead of things that work and perform, it becomes more and more logical to use MySQL or other databases instead of the best one of them all: Oracle. Since MySQL succeeded in becoming popular among students and their professors, it is immensely popular among them when they leave school (the professors stay, of course, since they don't know enough about databases to actually get a real job working with them outside academia). So MySQL will be used a lot. And it's an OK database, especially if we're talking the InnoDB engine.

"Since Oracle 7.3, that fantastic database has had pretty much everything normal customers need. It has become more and more fantastic; it has amazing features that are light years ahead of competitors—and fewer and fewer are using the database as it should be used (they're using it as a data dump, as Tom Kyte said many years ago)."

[February 2018 update] True story: When Ken Jacobs got a call many years ago from a Finn who had written InnoDB who said that he would sell it to MySQL unless Oracle was interested, Ken called Larry and got the go-ahead in 20 minutes. Why do I know that? Because Tuomas Pystynen and I were supposed to have had dinner with the Finn in Helsinki that evening (including one vodka at a time), but he cancelled. And then Ken wrote to me, Bjoern Engsig, and one more asking us this simple question: "We just bought MySQL—do you guys have any suggestions as to what we can use it for?" And we didn't. None of us. So for that, I'm not impressed with myself.

[August 2011] Do we probably need certification? Or do we learn best by making terrible mistakes on expensive production systems?

I hate certifications. They prove nothing, and they become a very bad replacement for real education, training, and knowledge. Among Windows and Cisco folks, it's immensely popular, but you can now feed all the farm animals in Denmark (and we've got quite a few, especially a lot of pigs) with certified Microsoft and Cisco people. It's taken by students (what?!? instead of real education, they train them in something that concrete? I find it really stupid), among unemployed (we have a lot of programs for those folks here), and what have you. They're worthless, and a lot of people think it will help them finding a job, thereby providing false hopes and security.

YPDNC.

[February 2018 update] On certifications: I was wrong. They serve a purpose. I should have seen it. I'm sorry. Fortunately, everyone else saw their usefulness, so as usual my predictions did absolutely no harm (or good).

[August 2011] Do we probably need ITIL? Should we resist those who try to control and hinder us?

When you begin doing "best practices" stuff like ITIL, you've lost. You're pouring cement down the org chart in your shop, and God bless you for that—it helps the rest of us compete. "Best practices" means copying and imitating others that have shops that are unlike yours. Standardizing and automating activity in brain-based shops always seemed strange to me. The results—surprise!—are totally predictable: jobs become immensely boring, response times become horrible, queues are everywhere, and nothing new can happen unless a boss high up *really* demands it. It's Eastern Europe—now with computers. Oh, and it's hype; it's modern right now but will be replaced by the next silly thing (like LEAN—what a fantastically stupid idea, too). Maybe we'll have LEAN ITIL one day? Or Balanced Score Card–adjusted ITIL? Or Total Quality Management of LEAN ITIL?

The funny thing is that Taylor's ideas (called "scientific management") were *never* proved, and he was actually fired from Bethlehem Steel after his idiotic idea of having a Very Big Hungarian lift 16 tons in one day (hence all the songs about 16 tons), because he cheated with the results and didn't get anything done that worked. Not one piece of his work has ever been proved to actually work. His "opponent" was Mayo (around the 1920s), with his experiments into altering the work environment (hence the constant org changes and office redos that everybody thinks must be good for something)—and his work has never been proved either. And he cheated too, by the way, which he later had to admit. So all this management stuff is bollocks, and ITIL is one of its latest fads. I say: Out with it. Let's have our lives and dignities back, please.

[February 2018 update] Well, I think I was also wrong here. The world needed stability and quiet and so on. For that, ITIL served a purpose (and several other similar systems). Then came DevOps, Baby, and Lean Agile Scrum, and many other wonderful and wild methods. So perhaps it's safe to say: old, frozen world: ITIL. For the rest of us: anything else, but probably DevOps, Baby.

[August 2011] NoCOUG membership and attendance has been declining for years. Do we probably need NoCOUG anymore? We'll celebrate our 25th anniversary in November. Should we

"Free, relational databases like MySQL and PostgreSQL, plus the whole NoSQL revolution, means that new and fantastic Oracle features will only be used by existing customers."

"Magazines should not be available anymore in print. Nor should they (in my view) be available on a silly website that people have to go to using a PC, a browser, and all sorts of other old-days technology. The smartphone is the computer now. Move the NoCOUG Journal there aggressively."

have a big party and close up the shop? Or should we keep marching for another 25 years?

No. Oracle User Groups are dead as such. Just like user groups for mainframe operators or typesetters. You can make the downward sloping angle less steep by doing all sorts of things, but it's the same with all Oracle user groups around the world. I think I have a "technical fix" or at least something crazy and funny that can prolong NoCOUG's life artificially: move onto the Net aggressively and do it with video everywhere. Let it be possible to leave video responses to technical questions (why doesn't Facebook have that?); let it be possible to upload video or audio or text replies to debates and other things via a smartphone app. Let there be places where the members can drink different beers at the same time and chat about it (and show the beer on the screen), etc., etc. In other words: Abandon the real world before all the other user groups do it—and perhaps that way you can swallow the other user groups around you and gradually have World Dominance.

[February 2018 update] [No update provided.]

[August 2011] It costs a fortune to produce and print the NoCOUG Journal. Do we probably need the **NoCOUG Journal** ***anymore?***

I have subscribed to the world's arguably best magazine, *The Economist*, since 1983. Recently they came out with an app, and now I don't open the printed edition any more (I still receive it for some reason). It's so much cooler to have the magazine with me everywhere I go, and I can sit in the bathroom and get half of the articles in there read. It's the way. Magazines should not be available anymore in print. Nor should they (in my view) be available on a silly website that people have to go to using a PC, a browser, and all sorts of other old-days technology. The smartphone is the computer now. Move the magazine there aggressively, and in the process, why not create a template that other user groups could take advantage of? Or the Mother of All User-group Apps (MOAUA) that will allow one user group after another to plug in, so people can read all the good stuff all over the world?

[February 2018 update] [No update provided.]

[August 2011] I'm writing a book on physical database design techniques like indexing, clustering, partitioning, and materialization. Do we probably need YABB (Yet Another Big Book)?

No, certainly not. Drop the project immediately, unless you can use it as an excuse to get away from the family now and then. Or, if you *must* get all this knowledge you have out of your system, make an app that people can have on their phone and actually USE in real-life situations. Abandon books immediately, especially the physical ones.

[February 2018 update] [No update provided.] ▲

Mogens Nørgaard is a sought-after speaker for his different approach to, well, everything. Sparekassen SDS, 1987–1990: Database administrator, supporter, and trainer. Established a large DW, managed hundreds of users, taught DW classes, and founded the Danish Oracle User Group (OUG/DK). Oracle Denmark, 1990–2000: support analyst, support team lead; founded Premium Services (35+ people); trainer; and consultant. Authored papers, spoke at conferences, wrote corporate teaching materials, held advanced classes, and managed many escalations at Danish C20 companies. Miracle A/S, 2000–2013: founder & CEO. Worked as consultant, escalation manager, trainer, coach, and speaker. Founded The OakTable Network. Authored internationally published papers; lead author of Oracle Insights: Tales of the OakTable, *co-author of several other Oracle-related books. Fair & Square, 2013–present: Founder & CEO. Works as consultant, escalation manager, trouble shooter, trainer, and speaker. CIMA, 2016–present: Founder & CEO. CIMA does real, expert-level DevOps, AWS, and Big Data/advanced analytics/machine learning—and is on the way to becoming a leading data broker.*

"Oracle User Groups are dead as such. Just like user groups for mainframe operators or typesetters. You can make the downward sloping angle less steep by doing all sorts of things, but it's the same with all Oracle user groups around the world. . . . Move onto the Net aggressively and do it with video everywhere. . . . Let it be possible to upload video or audio or text replies to debates and other things via a smartphone app."

JavaScript: The Good Parts

Book Notes by Brian Hitchcock

Brian Hitchcock

Details

Author: Douglas Crockford

ISBN-13: 978-0-596-51774-8

Date of Publication: May 2008

Publisher: O'Reilly Media

Summary

I'm neither a programmer nor a developer, so why did I decide to read this book? I've seen JavaScript in various places over the years, and I was curious about how it works. Many years ago I was required to take some basic Java programming training, and I wondered what, if anything, was similar between Java and JavaScript. I was looking for a single book to help me learn about JavaScript, and a friend who works full time with JavaScript strongly recommended this book. I read this book on Safari, and I recommend that you do the same.

Preface

The author tells us that this book is intended for programmers who are looking at JavaScript for the first time and those who have been using it but want to learn more about how it works. I don't fall into either of those categories, but we will see what I learn. Right away, I'm interested to discover that JavaScript is "unconventional," and a "small" language. I'm not sure what that means when used to describe a programming language, but I will play along. The goal of this book is to get the reader to think in JavaScript versus simply learning the language. This book covers only the most important parts of JavaScript and not everything you might need to know.

And then we are told, "This book is not for beginners … This book is not for dummies." I'm sure I fall into the former and I'm not sure I can be a reliable judge as to the latter. We are also told that this book is dense; i.e., it covers a lot of ground quickly and may require multiple readings before most of the material will make sense.

Chapter 1: Good Parts

We get some background as to why the author wrote this book. In a previous life, he would try to learn everything there was to know about the languages he was working with and try to use all the available features in his work. I appreciate the author's honesty that this was also driven by a desire to show off and to be the expert in the group. This explains some of the things I have seen in my years at work: things that were needlessly complicated and virtually impossible to support. I'm intrigued to read that languages have features that are simply design errors otherwise known as mistakes. I've always assumed programming languages were mostly magic, and when it was too complicated for me to understand, it was because I was in that dummies group. Further, it is pointed out that while a language may well be endorsed by a standards group, that does not mean the mistakes have been removed. Once a programming language has been used in the real world, removing any part, no matter how bad, would cause existing bad programs to break. Yes, the author uses the phrase "bad programs." I like this book! Then we are told that standards usually add more features that may makes things worse, creating more bad parts. And so, it goes.

From here, we are told that the best way forward is to accept the mistakes, focus on the parts of a language that are good, and—as much as possible—simply don't use the bad parts. Apparently, JavaScript has more bad parts than most languages, since it went from nothing to global adoption so quickly that it didn't have time to be refined. It seems that when Java applets failed, JavaScript became the standard for web programming. The author feels very strongly about this point, as we learn that JavaScript is buried under a steaming pile of good intentions and was for a long time considered an unsightly, incompetent toy. Wow! But how does the author really feel?

"Languages have features that are simply design errors otherwise known as mistakes. . . . While a language may well be endorsed by a standards group, that does not mean the mistakes have been removed. Once a programming language has been used in the real world, removing any part, no matter how bad, would cause existing bad programs to break."

While the book focuses on the parts of JavaScript that the author wants us to use, it is clear that these parts are more than enough to build worthwhile programs for both small and large projects. Avoiding the bad parts of JavaScript does not mean that you can't get stuff done.

Given this background, the next section addresses why we want to use JavaScript at all. The answer is simple: it's the language of the web browser, and therefore it's one of the most popular programming languages. It is also one of the most despised. Such drama! It seems that JavaScript gets unfairly blamed for the API of the browser, the awful Document Object Model (DOM). I don't know much about DOMs so I can't comment on this, but it sounds bad.

I'm amused to learn that with JavaScript, you can be productive while not knowing much about the language itself or programming in general. The author says this indicates that JavaScript has "expressive power." We are advised, however, that JavaScript is even better when you do know more about it.

A discussion of the good and bad parts of JavaScript follows. The good parts are functions, loose typing, dynamic objects, and an expressive object literal notation. The bad parts include global variables. I learned that JavaScript is the first lambda language to be widely used. I had no clue what a lambda language is, so I looked it up—and I still have no idea. This is a good example of what the author meant when he said that this book is not for beginners. Finally, after so much discussion of what's wrong with JavaScript, comes the truly interesting question: why should I use it? The answer is that you don't have a choice. On the other hand, JavaScript is really good and can be a lot of fun. Overall, the good parts more than outweigh the bad. You just have to know which is which and how to avoid the bad.

Given how much I don't know about programming, I much appreciate that with JavaScript, you can start writing simple programs right away with no other software. You just need a browser. For a beginner like me this is a big deal. I don't have to download some bloated development environment or learn commands to compile code. Nice. The unavoidable Hello World program is shown, and I was able to make it work. Very nice!

Chapter 2: Grammar

This chapter covers what you would expect, given its title. We see a number of railroad diagrams. I think they could have explained the use of white space better. I'm not sure what a fraction means in JavaScript, but my guess is that it's what I'd call a decimal like 3.123. There is a discussion of an operation that can't produce a normal result; the result is NAN—which is not equal to any value, including itself. This seems a lot like NULL in SQL.

Terms that are mentioned but not explained include linker, namespace, function, and scope. It is assumed that we know what these are. I will have to look elsewhere to understand a prototype chain. Again, I am not complaining: I wanted to read this book and I knew it wasn't for beginners.

We are told that the grammar of just the good parts of JavaScript is much easier than the grammar of the entire language. Does that mean that if someone doesn't know all of JavaScript, it makes them a better JavaScript programmer? One thing I did understand is that comments should only be done using // at the end of the line. The comment form /* */ is not good, because the same characters are used in regular expressions. JavaScript has only one number type. There is no integer type, so 1.0 and 1 are the same thing; this eliminates a lot of numeric type errors.

There are sections covering strings, statements, expressions, literals, and functions.

Chapter 3: Objects

In JavaScript, if something isn't a number, a string, true or false, null or undefined, it is an object. I now know that objects are mutable keyed collections, and an object is a container of properties where a property has a name and a value. Objects are class-free—which is a departure for me from my Java training, where classes were a big deal. Objects can inherit properties from other objects using the prototype linkage feature. There are sections discussing object literals, retrieval, update, reference, prototype, reflection, enumeration, delete, and global abatement—and no: this has nothing to do with mosquitos. Here it means to minimize the potential problems that come from using global variables. You create a single global variable for your application.

"The best way forward is to accept the mistakes, focus on the parts of a language that are good, and—as much as possible—simply don't use the bad parts. Apparently, JavaScript has more bad parts than most languages, since it went from nothing to global adoption so quickly that it didn't have time to be refined."

There are code snippets illustrating all of this. I was able to understand most of them. The first example starts with "var," which is not explained. I assume this means "variable," but I'm not sure; my background tells me that a variable is a number or a string. I'm also not equipped to really understand the object. prototype. It seems that all objects are linked to this in some fashion.

Chapter 4: Functions

It turns out that functions are the best part of the best parts of JavaScript. A function is an enclosed set of statements, and functions are the basic modular piece of JavaScript. We are told that programming in general really boils down to taking a set of requirements and creating a set of functions and data structures. That sounds easy. Functions are objects; so, like objects, functions are a bunch of names with values and link to the object. prototype. I fear I will have to do battle with this object.prototype at the end of the game. I find it confusing that functions can be stored in variables. The special thing about functions is that while they are objects, they can be invoked. There are many sections describing invocation, arguments, exceptions, recursion, and other topics. My favorites were curry and memoization. Try to get memoization past your spell checker. "Curry" means that you create a new function from a function and an argument. "Memoization" means that functions use objects to store the results of previous operations. Computing the Fibonacci numbers is shown as an example of how to use memoization. Is memoization a legal Scrabble word?

Chapter 5: Inheritance

This chapter tells us that inheritance in Java—which is referred to as a classical language—is a form of code reuse that can reduce the cost of developing software. Inheritance is also good because it specifies a system of types. This means that the programmer doesn't have do any casting operations, which can cause confusion and errors. JavaScript doesn't cast but it supports many possible inheritance patterns. In classical languages, objects are instances of classes and one class can inherit from another class. JavaScript is a prototypal language where objects inherit directly from other objects.

"With JavaScript, you can start writing simple programs right away with no other software. You just need a browser. For a beginner like me this is a big deal. I don't have to download some bloated development environment or learn commands to compile code. "

There are sections on various aspects of inheritance, including pseudo classical, object specifiers, prototypal, and parts. Not having classes seems like a big departure to me, but it does make things simpler.

Chapter 6: Arrays

The chapter begins with a definition of arrays that makes them sound pretty cool: a linear allocation of memory where elements are located using integers to compute offsets. Then we are told that JavaScript has nothing like this. So sad! JavaScript has an object that acts like an array. Array subscripts are converted into strings that become properties. Overall, the JavaScript implementation of arrays is much slower than a "real" array but is more convenient, and the provided set of methods is very good.

The following sections cover array topics, including literals, length, delete, enumeration, confusion, methods, and dimensions. JavaScript arrays are not limited to storing one type. There is no out-of-bounds error; if you add an element beyond the existing array, the array is automatically extended. If you reduce the length of the array, the elements beyond the new length are deleted. "Confusion" refers to the common challenge of when to use an object versus an array; this is often done incorrectly.

Chapter 7: Regular Expressions

JavaScript is made up of many other languages: syntax from Java, functions from Scheme, inheritance from Self, and regular expressions from Perl. I had not heard of Scheme or Self, but they are dialects of Lisp and Smalltalk, respectively. "Regular expressions" are defined as specifications of the syntax of a simple language. I don't claim to understand that definition, but I usually see regular expressions used with Perl, for example, to extract text. In JavaScript, regular expressions provide a large performance improvement over the equivalent string operation. I have always found regular expressions hard to understand. We are told that they can be complex because the same character can be a literal in one position and an operator in another. In general, regular expressions are hard to write and risky to modify, which makes them hard to support. The author describes them as terse bordering on cryptic and almost indecipherable. Given all of this, it is frustrating to learn that they are widely used.

The following sections provide an example of a regular expression and discuss the construction and elements of a regular expression. Reading these sections demonstrates just how confusing regular expressions can be. In JavaScript they are also poor choices when it comes to handling internationalization.

Chapter 8: Methods

This chapter is more of a reference section. It lists all the built-in methods that JavaScript provides. There are sections covering the methods for arrays, functions, numbers, objects, RegExp (regular expression), and strings. In the array section we read about the default comparison function that doesn't test the type of the array elements to be compared, so the result can be shockingly incorrect. I am beginning to understand why there are so many software bugs running around in the wild.

Chapter 9: Style

While the information in this interesting chapter isn't required in order to master the JavaScript language, it is important. It opens with a discussion of how complex software is and tells us that maintaining software means converting one correct program to a different, but still correct, program. This is challenging. With this in mind, good programs should be clear, and what they are doing should be easily understood. Specific features of JavaScript make this problematic. The loose typing and high level of error tolerance mean that the compilation process can't detect many errors. The way to deal with this is to be stricter as we write JavaScript programs. We are given specific ways to code in JavaScript that will eliminate many common errors. We are reminded that JavaScript was not designed or implemented with quality in mind.

Chapter 10: Beautiful Features

This chapter is more philosophy than training. The author tells us more about why this book focuses on the good parts of JavaScript. I liked the statement that parsing is a big deal in computing; this means that if you can write a compiler for a language, this in itself is a way to demonstrate the completeness of that language. Is this what goes on in computer science classes? The good parts of JavaScript are listed, including functions as first-class objects, dynamic objects with prototypal inheritance, and object and array literals. The last paragraphs are worth reading and thinking about. They state that all products have good parts. An example is a microwave oven that has many more features than anyone needs, so most people reduce the complexity to only those functions that they use. These are the good parts of the microwave oven. Finally, the last statement is that it would be nice if products, including software, only had good parts. Of

"Programming languages, which appear to be magic to the uninitiated, are products of humans and history. JavaScript dealt with a set of problems that needed to be solved. It isn't pretty, but it was, as most things in history are, in the right place at the right time. "

course, it is not the user who determines which parts are good—which makes designing products, including software, so difficult.

Appendix A: Awful Parts

Here we have a list of the bad parts of JavaScript that you probably can't avoid using, so you have to deal with them. You won't be surprised to see global variables at the top of this list. There are many other sections, including Unicode, NaN, and Falsy Values. The Unicode section interested me because it turns out that JavaScript was designed back when Unicode was limited to 64K characters. Unicode has grown to over a million characters since then. Since JavaScript only supports 16-bit characters, it must use pairs of characters to represent Unicode characters. I saw this on the job, years ago, when all the Kanji disappeared from an application. It took me three days, but I found them—all encoded as pairs of characters hiding in the database after a well-intentioned Unicode conversion had been executed.

Appendix B: Bad Parts

Here we have a list of features that you can easily avoid. These include the equality operators == and !=. These should be avoided in favor of === and !==. We should also avoid the continue statement and many others.

Appendix C: JSLint

Back when C was new and exciting, the compilers didn't catch some common errors, so an additional program called lint was built to check a source file for these known issues. The C language matured and no longer needs lint, but JavaScript is still immature, so we have JSLint. If we use JSLint, we can bring out the elegant code that hides inside the sloppy language that is JavaScript. There are many sections here, each describing how JSLint helps find common errors. As always, global variables are first to be examined.

Appendix D: Syntax Diagrams

The railroad diagrams are shown for all parts of JavaScript.

Appendix E: JSON

A very brief description of JSON is given, followed by a link to the JSON organization website.

Conclusion

It is good for us to look outside of our familiar area of experience. I was curious about JavaScript, and while I didn't understand parts of this book, I'm glad I read it. Next, I plan to read a beginner's book on JavaScript. I realize that reading this book first will seem illogical to most, but it worked for me. I'll have some background on the bigger design issues for JavaScript as I go back and learn more of the basics. I will be looking for some specific things while I move through the next book. It was good for me to learn that programming languages, which appear to be magic to the uninitiated, are products of humans and history. JavaScript dealt with a set of problems that needed to be solved. It isn't pretty, but it was, as most things in history are, in the right place at the right time. If you are interested in something outside your usual technical experience, don't resist: just start learning about it. Don't worry if you are starting in the right place or not; you will learn something that will guide you as you learn more. ▲

Brian Hitchcock works for Oracle Corporation where he has been supporting Fusion Middleware since 2013. Before that, he supported Fusion Applications and the Federal OnDemand group. He was with Sun Microsystems for 15 years (before it was acquired by Oracle Corporation) where he supported Oracle databases and Oracle Applications. His contact information and all his book reviews and presentations are available at **www.brianhitchcock.net/oracle-dbafmw/***. The statements and opinions expressed here are the author's and do not necessarily represent those of Oracle Corporation.*

Copyright © 2018, Brian Hitchcock

DATABASE MANAGEMENT SOLUTIONS

Develop | Manage | Optimize | Monitor | Replicate

Maximize your Database Investments.

Quest™

Oracle Licensing in the Cloud—Part IV

by Mohammad Inamullah

Mohammad Inamullah

***Editor's Note:** This article contains information on Oracle licensing that is provided as-is and without guarantee of applicability or accuracy. Given the complex nature of Oracle licensing and the ease with which license compliance risk factors can change significantly due to individual circumstances, readers are advised to obtain legal and/or expert licensing advice independently before performing any actions based on the information provided.*

Over my last three articles in the *NoCOUG Journal*, I have covered Oracle licensing in third-party clouds. My discussion has focused on deploying traditional, on-premises Oracle license entitlements in public cloud environments like Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). I touched on Oracle Cloud only briefly.

In this article, I will focus on Oracle Cloud. To narrow the scope of the discussion, I will look at deploying existing license inventories in Oracle's different cloud offerings. I will also go over the key contractual documents involved.

The article will focus primarily on Oracle Cloud's IaaS offerings, and briefly touch on PaaS (specifically, Database service). We will skip SaaS offerings for the time being. That's not to imply there isn't much to talk about with SaaS: understanding SaaS contracts and optimizing your investment in Oracle's SaaS environments are important topics. Much can be said about the myriad SaaS offerings and their service descriptions and metrics.

A comparison of Oracle Cloud with other cloud providers, from an offerings and pricing perspective, is beyond the scope of this article.

The Basics—Processors, NUPs, and OCPUs

As discussed in my previous *NoCOUG Journal* articles, Oracle's traditional, on-premises licensing includes several metrics for different products. However, the Processor and Named User Plus (NUP) metrics are predominant for database, middleware, and several applications products.

To recap, the number of Processor licenses required for a product depends on the number of cores running the Oracle software; the core count is then multiplied by a contractually defined Processor Core Factor. This is based on the chip type and is 0.5 for most x86 CISC processors. For example, suppose we are licensing Database Enterprise Edition on a server with one Xeon processor with eight cores. We would need: 1 processor * 8 cores * 0.5 (core factor) = 4 Processor licenses

The Processor license metric has not changed much since the early 2000s. The other long-running metric is the NUP metric. As its name suggests, it's based on actual, named users. However, it still has processor-related minimum requirements. For databases, it's a minimum of 25 NUPs per processor; for middleware products, it's generally ten NUPs per processor. Customers must license the higher of the actual number of users and minimums based on processor counts.

The OCPU

With the emergence of Oracle Cloud, Oracle introduced a virtual, cloud-based Processor metric called OCPU (Oracle documentation defines OCPU as Oracle Compute Unit. Shouldn't that be "OCU"?). Several months ago, the Oracle Cloud website provided a clear and unambiguous definition of what an OCPU is (which I covered in my first *NoCOUG Journal* article on the topic). The website stated: "one physical core of an Intel Xeon processor with hyper-threading enabled. Each OCPU corresponds to two hardware execution threads . . ." In short, an OCPU core was equivalent to a physical processing core. Strangely, the current website does not clearly document this. For that information, you must dig into the IaaS and PaaS Cloud service descriptions document at **www.oracle.com/us/corporate/contracts/paas-iaas-public-cloud-2140609.pdf**.

An obvious question arises: what is the contractually correct way to translate existing Processor license entitlements to their OCPU equivalents? Fortunately, this is well documented in the Important Notes section of Oracle's Processor Core Factor Table located at **www.oracle.com/us/corporate/contracts/processor-core-factor-table-070634.pdf**. It's crucial to note that this document is contractually binding because—unlike Oracle's policy on

"Oracle introduced a virtual, cloud-based Processor metric called OCPU … One Processor license will cover two OCPUs. This is consistent with what customers would be doing on premises. One Processor license will cover two Xeon cores, irrespective of whether hyper-threading is enabled."

licensing in third-party clouds—the Processor Core Factor Table is not just a policy; it is contractually referenced by your OLSA/OMA Oracle agreements.

1. The first important takeaway from the Processor Core Factor Table is that one Processor license will cover two OCPUs. This is consistent with what customers would be doing on premises. One Processor license will cover two Xeon cores, irrespective of whether hyper-threading is enabled. As such, the translation to OCPU is quite reasonable and appears to be consistent with the value customers would be getting on premises.
2. Second, the NUP minimums are consistent with customer requirements for on-premises NUP licensing.
3. Third, the Ravello service uses the vCPU metric, so it's no surprise that these are correlated as hyper-threads, with two threads equivalent to a physical core. As such, one Processor license covers four Ravello vCPUs.
4. Finally, for Standard Edition products, customers can be at a disadvantage with this update. As documented, one Processor license will only cover four OCPUs. However, in the on-premises world, Standard Edition products are licensed by the number of occupied sockets and not core counts (SE2 did place a limit, though). For example, a customer could deploy Database Standard Edition on one occupied socket with 16 or 20 cores and still only need one Processor license, since only one socket was occupied. However, the update limits that in Oracle Cloud. One Processor license will only cover up to four OCPUs, functionally equivalent to four physical cores.

With a discussion of OCPU and Processor metrics and calculating equivalents out of the way, let's look at some of the Oracle Cloud offerings and discuss the value and compliance considerations for each.

Oracle PaaS

Upon browsing the Oracle PaaS website, one sees over 40 PaaS offerings. The database service features options for shared tenancy and dedicated, bare-metal server hosting. Among the shared tenancy options, there are license-included and bring your own license (BYOL) options. For license-included options, there are services with Database Standard Edition 2 and Enterprise Edition. Depending on the Enterprise Edition package selected, various options and management packs are included in the license. The top-end service includes RAC, In-Memory, and Active Data Guard.

The BYOL service options require the customer to bring existing licenses for Database SE, SE1, SE2, or EE, depending on the service selected. Interestingly, while customers are mostly expected to bring licenses for any options and packs they will use, some options and packs are included for free for the Enterprise Edition services. Specifically, Data Masking and Subsetting Pack, Diagnostics and Tuning Packs, and Real Application Testing are included for free. By comparison, it's worth noting that AWS RDS with Oracle Database Enterprise Edition does not include any options or management packs.

Likewise, bare-metal hosting options include license-included and BYOL options. Similar to the shared-tenancy options, the hosting options variously include some to most of the important options and management packs.

For all BYOL situations, it's the customer's responsibility to have sufficient licenses to cover the underlying OCPUs.

For anyone considering Oracle Cloud Database service, I would recommend reviewing their Database service documentation portal located at **https://docs.us-phoenix-1.oraclecloud.com/Content/Database/Concepts/databaseoverview.htm**.

Oracle IaaS

Oracle IaaS offerings cover several infrastructure areas, including compute, networking, storage, and others.

The Compute service, comparable to AWS's EC2 service, provides several instance-sized options for bare-metal and shared-tenancy virtual machines. Oddly, Oracle's website also lists

"The Ravello service uses the vCPU metric, so it's no surprise that these are correlated as hyperthreads, with two threads equivalent to a physical core. As such, one Processor license covers four Ravello vCPUs."

"Database" under the IaaS section. I find this odd, since the same Database services, with identical prices, are also a subset under the Database PaaS services webpage. Stranger still is that it costs the same as the Database PaaS service.

While IaaS makes sense for general computing purposes, does it make sense to use Oracle IaaS options over PaaS where PaaS services are available? For example, does it make sense for customers to launch a compute instance and install their own Oracle Database? Especially when a similar PaaS service is also offered? For the most part, for customers looking to benefit from optimization and reduce management costs, PaaS options tend to make more sense than comparable IaaS options. However, there is one subtle benefit IaaS provides: it presumably gives Oracle less insight into what's going on inside the instance. This can be important in the event of a license audit. For Oracle's auditors, it would be infeasible to automatically extract usage details for a manually installed Oracle Database or WebLogic installation in an IaaS instance. For PaaS, there is no reason to believe the auditors would have any difficulty in extracting usage details on options and packs without much involvement from the customer. The IaaS alternative would add a few layers of complexity for the auditors and likely place more control of the audit back in the hands of the customer.

The CSA, the Cloud Computing Policy, and Other Important Documents

The Oracle Cloud Services Agreement (CSA) is the key agreement underlying a customer's Oracle Cloud relationship. The U.S. CSA is located at **www.oracle.com/us/corporate/contracts/saas-online-csa-us-1894130.pdf**.

The document is 11 pages long and, like every other Oracle licensing document, should be read carefully and thoroughly. Too many of my clients have signed CSAs without reading them and understanding their key points. I'll go over some of the important points below:

1. Definitions: While it's important to review and understand all the definitions in this section, a few are worth pointing out.
   a. "Data Center Region" is the geographic location where your service will reside. It will be specified on the order document. It's important to ensure that the region is noted correctly on the ordering document and is as expected.
   b. "Service Specifications" provides technical details on the services. These can be accessed at **www.oracle.com/us/corporate/contracts/cloud-services/index.html**. The documents and service descriptions provide important technical details on the services provided. Readers can further drill town to specific service details. For example, key terms and details for the IaaS and PaaS services can be found via links under the "ORACLE PLATFORM AS A SERVICE (PAAS) AND INFRASTRUCTURE AS A SERVICE (IAAS)" section.

"Does it make sense to use Oracle IaaS options over PaaS where PaaS services are available? For example, does it make sense for customers to launch a compute instance and install their own Oracle Database? Especially when a similar PaaS service is also offered? "

2. Audit clause: Section 23.8 establishes the audit clause. Oracle's OMA and OLSA agreements have defined audit clauses for decades; these stipulate that Oracle may initiate an audit "upon 45 days' notice." However, the CSA audit clause is different in that it gives the customer no notice period. In short, Oracle can initiate an audit with no warning and expect immediate cooperation. The customer will have little or no time to do any stock-taking. Furthermore, given the nature of data that Oracle may already have due to Oracle Cloud usage, Oracle auditors may be able to drive the audit more rapidly than expected.
3. On an interesting note, Section 23.4 states that "Oracle Programs and Services are not designed for or specifically intended for use in nuclear facilities or other hazardous applications . . ." I found this interesting.
4. Service analyses: Not unexpectedly, Oracle uses your information to compile usage and performance statistics for a variety of reasons.

On another note, the Oracle policy document titled "Licensing Oracle Software in the Cloud Computing Environment," and the restrictive policies mentioned therein, does not apply to Oracle Cloud.

Additionally, I strongly recommend readers to review the service descriptions and metrics documents located at the Oracle Cloud Services website mentioned above (**www.oracle.com/us/corporate/contracts/cloud-services/index.html**). These documents contain specific details for the different cloud offerings across SaaS, PaaS, and IaaS. Customers should navigate to specific Oracle Cloud documentation and ensure that they understand the technical and metric details for the product of interest. Suppose a customer is interested in the Oracle InForm Clinical Trials SaaS service. The customer would find important metric and service details via this portal in the form of the InForm Trial Capacity Cloud Service document located at **www.oracle.com/us/corporate/contracts/ohs-trial-capacity-cloud-service-sd-2952144.pdf**.

A Note on Oracle Unlimited License Agreements (ULAs) and Oracle Cloud

As I discussed in my previous articles on Oracle licensing in third-party clouds, Oracle's policy on cloud licensing in third-party clouds introduces several non-contractual limitations and restrictions for Oracle ULA customers. This includes the restriction that ULA customers may not claim cloud-deployed Oracle program quantities in their ULA certification. In the case of Oracle Cloud, I am not aware of any similar limitations through any of the Oracle Cloud documents. In other words, there is no reason for customers to have any issues with claiming their deployments in Oracle Cloud as part of their ULA certification quantities.

Value and Compliance—Key Considerations for Oracle Cloud

When assessing the value and compliance factors for Oracle Cloud offerings, customers should keep in mind several important factors:

1. The list price for license-included versus BYOL climbs very rapidly. A thorough analysis of Oracle Cloud vs. on-premises costs is beyond the scope of this article, but customers should perform a thorough, multi-year analysis of the expected costs.
2. A complicating factor for license audits would be order of precedence of the audit clauses in the CSA and OMA/OLSA. I have not come across this problem yet, but if Oracle audits a cloud customer, issues of audit scope will complicate things right away.
3. It's easy to see apparent cost savings in moving on-premises workloads to the Oracle Cloud. For example, there may be some cost savings if some additional options and

"Oracle's OMA and OLSA agreements have defined audit clauses for decades; these stipulate that Oracle may initiate an audit "upon 45 days' notice." However, the CSA audit clause is different in that it gives the customer no notice period. "

"Oracle's policy on cloud licensing in third-party clouds introduces the restriction that ULA customers may not claim cloud-deployed Oracle program quantities in their ULA certification. In the case of Oracle Cloud, there is no reason for customers to have any issues with claiming their deployments in Oracle Cloud as part of their ULA certification quantities. "

packs are needed. Customers should model the pricing for several years and factor in at least the following:

a. Quantify licenses they own and those they need, and the most economical way to bridge the gap—that is, buying additional on-premises licenses or availing themselves of products freely bundled into the Oracle Cloud offerings.

b. The ongoing cost of physical, on-premises options.

c. Comparison of the cost of their on-premises options, along with their license investment and annual support spend, to the different Oracle Cloud options (PaaS, license-included and BYOL, and IaaS + licensing costs, etc.).

d. Customers can expect to get some Oracle Cloud credits for migrations of existing license entitlement. However, it's important to remember that this is a one-way decision; there is no realistic way to back out. Once you trade in on-premises Oracle licenses for cloud credits, you will not be able to get them back if you decide to leave Oracle Cloud.

e. Renewal pricing: what happens when the Oracle Cloud subscription term expires? Oracle would be glad to sell long, multi-year Cloud contracts, but customers should seriously consider what will happen to pricing for subsequent renewals.

4. Compliance should still be a top priority in Oracle Cloud. While it is unlikely that Oracle will target its customers soon for license audits in the Oracle Cloud environment, there is no reason to become lax about this. With BYOL options, customers still have the responsibility to ensure that they are compliant in their processor-to-OCPU calculations and that their license entitlements sufficiently cover their usage. Additionally, customers still have to ensure that any user-based NUP licenses are sufficiently budgeted.

5. Some licensing rules and pricing may be different from traditional on-premises options. This is important for value and compliance calculations. In some cases metered vs. non-metered choices can have a significant value impact as well. For example, if a customer is considering Oracle GoldenGate in the cloud, the metered, pay-as-you-go option will cost around $1,000 per OCPU/month. On the other hand, the non-metered, flat-fee option is $3,000 per OCPU/month. When customers hear pitches about the economics of non-metered options, they ought to review and understand these details independently in order to make an informed decision.

In summary, Oracle Cloud represents an obvious cloud option for existing Oracle customers. However, getting the best value out of the move—in the short and long terms—and guarding against compliance issues requires customers to perform thorough due diligence. ▲

Addendum: *This addendum is relevant to my prior articles in the NoCOUG Journal; it concerns recent changes in Azure's compute offerings and corresponding changes to Oracle's licensing policy in third-party cloud environments.*

In April 2017, Microsoft started introducing vCPU-based compute options for its VMs. According to Microsoft, one vCPU corresponds to one hyper-thread on its latest class of Intel Xeon processors. This approach mirrors AWS's vCPU approach to its EC2 and RDS services, in which instance horsepower is defined in terms of hyper-threads. Previously, Microsoft's offerings were solely calibrated using CPU cores and ignored Intel's hyper-threading capability. Over the course of 2017, Azure introduced several classes of vCPU-based compute options. While core-based compute classes still exist, it's clear that Microsoft is moving to a vCPU approach. Accordingly, in January 2018, Oracle's policy document titled "Licensing Oracle Software in the Cloud Computing Environment" was updated, presumably to accommodate this change in Azure. The updated policy now provides mirror language for AWS (EC2 and RDS) and Azure. Previously, one CPU core in Azure corresponded to one Oracle Processor license. Now, however, according to the updated policy, if hyper-threading is enabled, two Azure vCPUs correspond to one Oracle Processor license. If hyper-threading is not enabled, one Azure vCPU corresponds to one Oracle Processor license. The guidance for Database Standard Edition (SE) licensing is also updated. Database SE may only be licensed in Azure instances with up to 16 vCPUs; the prior limit was eight CPU cores. Similarly, Database SE1 and SE2 may only be licensed in Azure instances with up to eight vCPUs; the prior limit was four CPU cores.

Also new in this policy update is that Oracle has taken a position on the use of named-user licenses in the AWS and Azure clouds. According to the policy update, if Database SE2 is being licensed by Named User Plus (NUP) licenses, customers must license at least ten NUPs per eight AWS vCPUs or eight Azure vCPUs. Strangely though, the policy provides no minimum-NUP guidance for the older products—Database SE and Database SE1.

While readers should be aware of Oracle's policy changes, my central position remains the same: Oracle's policy on licensing Oracle software in third-party clouds is of no contractual value, and customers should not allow Oracle to bring this policy into their purchasing and licensing discussions.

Mohammad Inamullah is the Principal at Redwood Compliance in Palo Alto, California. He can be reached at **mohammad@redwoodcompliance.com**.

© 2018 Mohammad Inamullah

12 Things Developers Love About Oracle Database 12*c* Release 2

by Chris Saxon

Chris Saxon

Oracle Database 12*c* Release 2 (12.2) is available on Oracle Cloud and on-premises. With it comes a host of new features to help you write better, faster applications. Here's my rundown of the top 12 new features to help you when developing applications for Oracle Database.

Top Feature I—JSON from SQL

12.1.0.2 brought JSON support to Oracle Database. This helped you work with JSON documents stored in clobs or varchar2s. These are fantastic, but storing raw JSON should be the exception, not the norm. Most of the time you should shred your JSON documents into relational tables. This leaves you with a problem though: getting the data back out in JSON format. Trying to write your own JSON generator is difficult, so in 12.2 we offer a whole host of options to help you get the job done. 12.2 provides four key functions to help you write SQL that returns data in JSON format:

- JSON_object
- JSON_objectagg
- JSON_array
- JSON_arrayagg

You use the JSON_object* functions to create a series of key-value pair documents—i.e., the output has curly braces {}. The JSON_array* functions take a list of values and return it as an array—i.e., in square brackets []. For each row in the input, the non-agg versions of these functions output a row. The agg versions combine multiple rows into a single document or array. How do these work? Let's look at an example.

Say you're using the classic employees and departments tables. For each department you want a JSON document that contains:

- The department name
- An array of its employees
- Each element of this array should be its own document, listing the employee's name and their job title.

For example:

```
{
  "department": "Accounting",
  "employees": [
    {
      "name": "Shelley,Higgins",
      "job": "Accounting Manager"
    },
    {
      "name": "William,Gietz",
      "job": "Public Accountant"
    }
  ]
}
```

How do you create this using the new functions? Let's work from the inside out:

First you need a document for each employee. This has two attributes: name and job. Pass these into a JSON_object call.

Then you need to turn these into an array. Wrap the JSON_object in a JSON_arrayagg, and group by department to split the employees for each one into a separate array.

Finally you have a single document per department, so you need another JSON_object with department and employees attributes. The values for these are the department name and the results of the JSON_arrayagg call in the previous step.

Put it all together and you get:

```
select json_object (
           'department' value d.department_name,
           'employees' value json_arrayagg (
              json_object (
                'name' value first_name || ',' || last_name,
                'job' value job_title )))
from hr.departments d, hr.employees e, hr.jobs j
where d.department_id = e.department_id
and e.job_id = j.job_id
group by d.department_name;
```

And voila! You have your JSON.

Top Feature II—JSON in PL/SQL

Now you have your JSON document, but what if you want to edit it? Say you want to change the names to uppercase and add a title element. The previous document becomes:

```
{
  "department": "Accounting",
  "employees": [
    {
      "name": "SHELLEY,HIGGINS",
      "job": "Accounting Manager",
      "title": ""
    },
    {
      "name": "WILLIAM,GIETZ",
      "job": "Public Accountant",
      "title": ""
    }
  ]
}
```

If you're generating the document, it's easiest to add these in the SQL (this presumes you want to change a JSON document from an external source). To help with this, there are new PL/SQL objects that enable you to access, modify, and add elements to a JSON document with get/put calls.

The key object types are as follows:

- json_element_t—a supertype for documents and arrays
- json_object_t—for working with JSON documents
- json_array_t—for working with JSON arrays

The first thing you need to do is create the JSON object. Do this by parsing the document:

```
doc := json_object_t.parse('
  {
    "department": "Accounting",
    "employees": [
      {
        "name": "Shelley,Higgins",
        "job": "Accounting Manager"
      },
      {
        "name": "William,Gietz",
        "job": "Public Accountant"
      }
    ]
  }
');
You can then access the employees array using get:
emps := treat(doc.get('employees') as json_array_t) ;
```

The treat function casts the element to the appropriate type (JSON_array_t here). Once you have the array, you can loop through the employees. Put adds a new key if it's not already present; otherwise it overwrites the existing value.

```
for i in 0 .. emps.get_size - 1 loop
  emp := treat(emps.get(i) as json_object_t);
  emp.put('title', '');
  emp.put('name', upper(emp.get_String('name')));
end loop;
```

The get functions return a reference to the original object. If you get some JSON and modify it, the original document also changes. If you don't want this, clone the element when you get it. For example:

```
emps := treat(doc.get('employees') as json_array_t).clone
```

The complete PL/SQL block to transform the JSON is

```
declare
  doc json_object_t;
  emps json_array_t;
  emp json_object_t;
begin
  doc := json_object_t.parse('{
  "department": "Accounting",
  "employees": [
    {
      "name": "Shelley,Higgins",
      "job": "Accounting Manager"
    },
    {
      "name": "William,Gietz",
      "job": "Public Accountant"
    }
  ]
}');

  emps := treat(doc.get('employees') as json_array_t) ;

  for i in 0 .. emps.get_size - 1 loop
    emp := treat(emps.get(i) as json_object_t);
    emp.put('title', '');
    emp.put('name', upper(emp.get_String('name')));
  end loop;

  dbms_output.put_line(doc.to_String);
end;
/

{
  "department": "Accounting",
  "employees": [
    {
      "name": "SHELLEY,HIGGINS",
      "job": "Accounting Manager",
      "title": ""
    },
    {
      "name": "WILLIAM,GIETZ",
      "job": "Public Accountant",
      "title": ""
    }
  ]
}
```

Now you can generate JSON from SQL and change it in PL/SQL; you have powerful options to work with it, and there are a raft of other improvements to JSON functionality in 12.2. Other enhancements include:

- JSON_exists function
- Support for In-Memory, Partitioning and Materialized Views
- Search indexes
- GeoJSON
- JSON Data Guide

If you're desperate to work with JSON, I recommend checking these out.

Top Feature III—Looooooooooooooooong Names

"There are only two hard things in Computer Science: cache invalidation and naming things."—Phil Karlton

Oracle Database handles cache invalidation for you. As a developer you don't have to worry about this, but when it comes to naming things, we've made it harder than it ought to be. Why? Take the following example:

```
alter table customer_addresses add constraint
  customer_addresses_customer_id_fk
  foreign key ( customer_id )
  references customers ( customer_id );
```

Looks like a standard foreign key creation, right? But there's a problem. Run it and you'll get:

```
SQL Error: ORA-00972: identifier is too long
```

Aaarrghh! The constraint name is just a tiny bit too long. Staying within the 30-byte limit can be tricky, particularly if you have naming standards you have to follow. As a result, many people have asked for us to allow longer names. Starting in 12.2 we've increased this limit. The maximum is now 128 bytes. Now you can create objects like:

```
create table with_a_really_really_really_really_really_long_name (
  and_lots_and_lots_and_lots_and_lots_and_lots_of int,
  really_really_really_really_really_long_columns int
);
```

Remember: the limit is 128 bytes, not characters. If you're using a multi-byte character set, you'll find you can't create:

```
create table tabléééééééééééééééééééééééééééééééééééééééééééééééééééééééééééééé (
  is_67_chars_but_130_bytes int
);
```

This is because é uses two bytes in character sets such as UTF8. So even though the string above is only 67 characters, it needs 130 bytes. I know some of you are desperate to startCreatingTablesWithRidiculouslyLongNames. But before you rush out to do so, check your code. If your dev team uses sloppy coding practices, there may be some traps waiting for you . . .

Top Feature IV—Robust Code Using Constants for Data Type Lengths

Most applications have at least one piece of PL/SQL that selects from the data dictionary. For example:

```
begin
  select table_name
  into   tab
  from   user_tables
  where  ...
```

Because the maximum length of a table name has been 30 bytes forever, some developers took to declaring the variable as follows:

```
declare
  tab varchar2(30);
```

Who needs more than 30 characters, right? But, as we just saw, upgrade to 12.2 and the limit is now 128 bytes. It's only a matter of time before people create tables with longer names. Eventually this code will fail with:

```
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
```

What to do?

It would be nice if you could change the maximum length of a varchar2 dynamically. Then, instead of combing through your PL/SQL and changing varchar2 (30) to varchar2 (128), you could increase the size in a single place. Fortunately, in 12.2 you can. The new release enables you to declare a variable length using a constant. That means you could create a constants package:

```
create or replace package constants as
  tab_length constant pls_integer := 128;
end constants;
/
```

And then use it when declaring your variables:

```
declare
  tab varchar2( constants.tab_length );
```

If we ever increase the length of names again, you only need to make one change: the constant's value. Note that these aren't fully dynamic. The PL/SQL compiler has to know the value for the variable size at compile time. This means you can't base it on the results of a query. User-defined functions are also out. In order to enable the variable to hold longer strings, you need to increase the value of constants.tab_length and recompile your code.

You may be thinking that for something as common as object names, surely Oracle provides something stating their max length? The good news is that we do. In DBMS_STANDARD you'll find new constants, including ora_max_name_len. As the name suggests, this states the maximum length for object names. Therefore, you can change your table name variable declarations to:

```
declare
  tab varchar2( ora_max_name_len );
begin
```

The best part is that you can make your code futureproof now. By using conditional compilation you can change your data dictionary–based variable declarations to:

```
declare
  $if DBMS_DB_VERSION.VER_LE_12_1 $then
    tab varchar2( 30 );
  $else
    tab varchar2( ora_max_name_len );
  $end
```

Then when you upgrade, the tab variable will automatically have the larger limit. You may be thinking that it sounds like a lot of work . . . and you're right. You can also make your variables 12.2 compatible now with type anchoring:

```
declare
  tab user_tables.table_name%type;
```

Whichever method you use, start preparing your code now. It may be a long time until you upgrade, but the more robust your code is, the easier it'll be for you to use the new features.

Variable declarations are one of the more obvious problems you'll meet with longer names. Let's look at a more subtle issue.

Top Feature V—Listagg Improved on Overflow

The following query returns a comma-separated list of indexes for each table in your schema:

```
select table_name,
       listagg(index_name, ',' 
       ) within group (order by index_name) inds
from   user_indexes
group  by table_name;
```

This is all very well and good, but there's a potential problem. Listagg() returns a varchar2. This is limited to 4,000 bytes (32,767 if you're using extended data types). In 12.1 and 11.2, you needed 130 or more indexes on a table before you started running into issues. If you have that many indexes on one table, you've got bigger problems than hitting this limit. However, this changes in 12.2. With longer names, you could hit this limit at just over 30 indexes on a table. While still a large number, this is plausible, particularly in reporting databases and data warehouses. Also, you can be sure that someone, somewhere will start creating "self-documenting" indexes, such as:

```
create index
  reducing_the_monthly_invoice_run_
  from_four_hours_to_three_minutes_
  PROJ12345_make_everything_faster_
  csaxon_thanks_everyone_yeah_baby on ...
```

Create too many of these and your listagg query will throw frustrating ORA-01489 errors. To get around this is tricky, so in 12.2 we've added an overflow clause. To use it, place "on overflow truncate" after the separator:

```
select table_name,
       listagg(index_name, ','
          on overflow truncate
       ) within group (order by index_name) inds
from   user_indexes
group  by table_name;
```

With this in place, instead of an exception your output will now look something like the following:

```
...lots_and_lots_and_lots,of_indexes,...(42)
```

The "…" at the end indicates that the output is larger than Oracle can return. The number in brackets reflects how many characters Oracle trimmed from the results. So not only can you see that there is more data, you can also get an indication of how much there is. The full syntax of this is:

```
  listagg (
    things, ','
    [ on overflow (truncate|error) ]
    [ text ] [ (with|without) count ]
  ) within group (order by cols)
```

Now you can explicitly say whether you want error or truncation semantics. There's a good chance that you've already written code to handle the ORA-1489 errors. To keep the behavior of your code the same, the default remains error. The text and count clauses control what appears at the end of the string. If you want to replace "..." with "more," "extra," or a "click for more" hyperlink, just provide your new string.

```
  select table_name,
         listagg(index_name, ','
            on overflow truncate
            '<a href="http://www.showfulldetails.com">click here</a>'
         ) within group (order by index_name) inds
  from   user_indexes
  group  by table_name;
```

You can also remove the number of trimmed characters by specifying "without count."

Top Feature VI—Lightning-Fast SQL with Real-Time Materialized Views

Materialized views (MVs) can give an amazing performance boost. Once you create one based on your query, Oracle can get the results directly from the MV instead of executing the statement itself. This can make SQL significantly faster, especially when the query processes millions of rows, but there are only a handful in the output.

There's just one problem: The data in the MV has to be fresh; otherwise, Oracle won't do the rewrite. You could, of course, query the MV directly, but the data will still be old. So, you need to keep the materialized view up to date. The easiest way is to declare it as "fast refresh on commit," but this is easier said than done. Doing this raises a couple of issues:

- Only some queries support on commit refreshes.
- Oracle serializes MV refreshes.

If you have complex SQL, you may not be able to use query rewrite. And even if you can, on high-transaction systems the refresh overhead may cripple your system. So, instead of "fast refresh on commit," you make the MV "fast refresh on demand" and create a job to update it that runs every second. But no matter how frequently you run the job, there will always be times when the MV is stale; query performance could switch between lightning fast and dog slow—a guaranteed way to upset your users.

How do you overcome this? With real-time materialized views. These give the best of both worlds. You can refresh your MV on demand but still have it return up-to-date information. To do this, create the MV with the clause "on query computation." For example:

```
create table t (
  x not null primary key, y not null) as
  select rownum x, mod(rownum, 10) y from dual
  connect by level <= 1000;

create materialized view log on t
with rowid (x, y) including new values;

create materialized view mv
refresh fast on demand
enable on query computation
enable query rewrite
as
  select y , count(*) c1
  from t
  group by y;
```

With this, you can add more data to your table:

```
insert into t
  select 1000+rownum, 1 from dual
  connect by level <= 100;

commit;
```

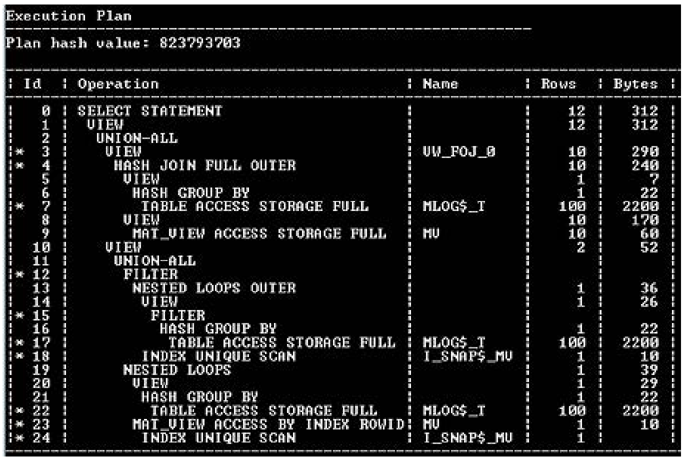
Oracle can still use the MV to rewrite—*even though the MV is stale.*

```
select /*+ rewrite */y , count(*) from t
group by y;
```

It does this by:

- Querying the stale MV
- Applying the inserts, updates, and deletes in the MV log to it

This can lead to some scary-looking execution plans.



```
Execution Plan
----------------------------------------------------------
Plan hash value: 823793703

---------------------------------------------------------------------------
| Id  | Operation                            | Name       | Rows  | Bytes |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |            |    12 |   312 |
|   1 |  VIEW                                |            |    12 |   312 |
|   2 |   UNION-ALL                          |            |       |       |
|*  3 |    VIEW                              | VW_FOJ_0   |    10 |   290 |
|*  4 |     HASH JOIN FULL OUTER             |            |    10 |   240 |
|   5 |      VIEW                            |            |     1 |     7 |
|   6 |       HASH GROUP BY                  |            |     1 |    22 |
|*  7 |        TABLE ACCESS STORAGE FULL     | MLOG$_T    |   100 |  2200 |
|   8 |      VIEW                            |            |    10 |   170 |
|   9 |       MAT_VIEW ACCESS STORAGE FULL   | MV         |    10 |    60 |
|  10 |    VIEW                              |            |     2 |    52 |
|  11 |     UNION-ALL                        |            |       |       |
|* 12 |      FILTER                          |            |       |       |
|  13 |       NESTED LOOPS OUTER             |            |     1 |    36 |
|  14 |        VIEW                          |            |     1 |    26 |
|* 15 |         FILTER                       |            |       |       |
|  16 |          HASH GROUP BY               |            |     1 |    22 |
|* 17 |           TABLE ACCESS STORAGE FULL  | MLOG$_T    |   100 |  2200 |
|* 18 |        INDEX UNIQUE SCAN             | I_SNAP$_MV |     1 |    10 |
|  19 |      NESTED LOOPS                    |            |     1 |    39 |
|  20 |       VIEW                           |            |     1 |    29 |
|  21 |        HASH GROUP BY                 |            |     1 |    22 |
|* 22 |         TABLE ACCESS STORAGE FULL    | MLOG$_T    |   100 |  2200 |
|* 23 |       MAT_VIEW ACCESS BY INDEX ROWID | MV         |     1 |    10 |
|* 24 |        INDEX UNIQUE SCAN             | I_SNAP$_MV |     1 |       |
---------------------------------------------------------------------------
```

The point to remember is that Oracle is reading the materialized view log and then applying the changes to the MV. The longer you leave it between refreshes, the more data there will be. You'll need to test to find the sweet spot between balancing the refresh process and applying MV change logs on query rewrite.

You can even get the up-to-date information when you query the MV directly. To do so, add the fresh_mv hint:

```
select /*+ fresh_mv */* from mv;
```

The really cool part? You can convert your existing MVs to real time with the following command:

```
alter materialized view mv
enable on query computation;
```

This makes MVs much easier to work with, opening up your querying tuning options.

Top Feature VII—Approximate Query Enhancements

If you do data analysis, you often need to answer questions such as:

- ➤ How many customers visited our website yesterday?
- ➤ How many different products did we sell last month?
- ➤ How many unique SQL statements did the database execute last week?

(OK, maybe that last one is just me.) In any case, often these questions are simply the starting point for further analysis. You just want a quick estimate. Answering these questions normally needs a count distinct along the lines of:

```
select count ( distinct customer_id )
from website_hits;
```

However, these queries can take a long time to run, and waiting for the answer is frustrating. It's worse if you're getting the figures for someone else—like your boss—and the figures are needed for a meeting that starts in a minute. Especially when your query takes at least ten minutes. In cases like this you just need a quick estimate. After all, your boss will round your figure to one or two significant digits anyway.

In 12.1.0.2 we introduced approx_count_distinct, which returns an estimate of how many different values there are in the target column. This is typically over 99% accurate and could be significantly faster than exact results. This is cool, but to take advantage of it, you need to change your code, which could be a time-consuming task. This is especially true because most of the time you'll want to be able to switch between exact and approximate results. A simple find+replace is out; instead you'll have to pass in a flag to toggle between modes.

If you're a big user of distinct counts this could be a lot of work, so in 12.2 we introduced a new parameter: approx_for_count_distinct. Set this to true as follows:

```
alter session set approx_for_count_distinct = true;
```

Oracle then implicitly converts all count distincts to the approximate version. While playing with this you may notice a couple of other new parameters:

- ➤ approx_for_aggregation
- ➤ approx_for_percentile

What are these all about? Well, in 12.2 we've created a new function: approx_percentile. This is the approximate version of the percentile_disc and percentile_cont functions. It's the same concept as approx_count_distinct, just applied to these functions. The syntax for it is

```
approx_percentile (
  <expression> [ deterministic ],
  [ ('ERROR_RATE' | 'CONFIDENCE') ]
) within group ( order by <expression>)
```

As you can see, this has a couple of extra clauses over approx_count_distinct.

Deterministic Results

"Deterministic" defines whether you get the same results each time you run it on the same data set. "Non-deterministic" is the default, meaning that you could get different answers each time. You may be wondering why you would ever want non-deterministic results. There are a couple of reasons:

- ➤ Non-deterministic results are faster.
- ➤ You can only get deterministic results on numeric values.

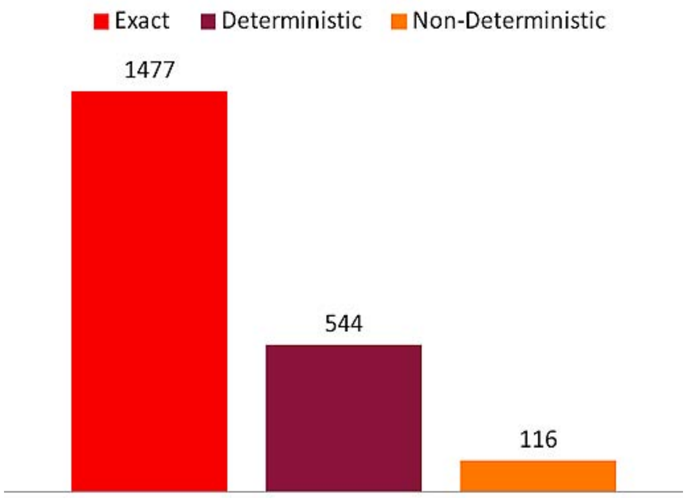
For instance, if you want the 10th percentile in a range of dates, you have to go non-deterministic. But is the time saving for non-deterministic results worth it? To find out, I created a 16 million row table Exadata Express Cloud Service and then compared the run time of the following exact, deterministic, and non-deterministic percentiles:

```
select percentile_disc(0.1)
within group (order by y)
from   super_massive;

select approx_percentile(0.1 deterministic)
within group (order by y)
from   super_massive;

select approx_percentile(0.1)
within group (order by y)
from   super_massive;
```

Averaging the time for three runs of each gave the following results:



The figure shows the average run time in hundredths of a second. Non-deterministic results are around 5x faster than deterministic and nearly 15x faster than exact results. If an estimate is all you need, you can save yourself a lot of time using approx_percentile.

ERROR_RATE and CONFIDENCE

If you're getting estimated figures, it does raise the question of just how accurate the results are. If they're 99.9999% accurate, that's almost certainly "good enough." But what if they're only 98% accurate? Or 95%? At some point the error is too large for you to rely on the estimate, and you'll want to switch back to exact calculations. But to do this, you need to know what the error is. To find this, pass ERROR_RATE or CONFIDENCE as the second parameter. Then you'll get the accuracy figures instead of the function result. "Confidence" is how certain we are that the answer is correct. The error rate gives the level of inaccuracy, which is perfect for finding out how good the approximation is. And there's more . . .

The stats geeks among you will know that median is a special case of percentile, which means that there's also an approx_median function available. This works in the same way as approx_percentile. But how do these functions relate to the parameter approx_for_percentile? Well, there are two percentile functions in Oracle: percentile_disc and percentile_cont. You have the option to convert either of these or both of them, as well as to do so in a deterministic manner or not. The values this takes are

- all deterministic
- percentile_disc deterministic
- percentile_cont deterministic
- all
- percentile_disc
- percentile_cont
- none

Top Feature VIII—Verify Data Type Conversions

Validating that a date is indeed a date is one of those all-too-common problems. A prime cause of this is the terrible practice of storing dates as strings. One of the biggest issues it is enables people to store things that clearly aren't dates in "date" columns:

```
create table dodgy_dates (
  id            int,
  is_this_a_date varchar2(20)
);

insert into dodgy_dates
values (1, 'abc');
```

Along with a whole bunch of values that might be dates:

```
insert into dodgy_dates
values (2, '20150101');

insert into dodgy_dates
values (3, '01-jan-2016');

insert into dodgy_dates
values (4, '01/01/2016');
```

Returning only the valid dates is tricky. If you try to convert everything using to_date(), you'll get exceptions:

```
select t.*
from   dodgy_dates t
where  to_date(is_this_a_date) < sysdate;

ORA-01858: a non-numeric character was found where a numeric was expected
```

Or maybe:

```
ORA-01861: literal does not match format string
```

Or:

```
ORA-01843: not a valid month
```

You could get around this by writing your own is_date() function. Or, if you're really brave, use a regular expression. Either way, it's a lot of unnecessary work. To make your life easier, we've created a new function: validate_conversion. You pass this a value and a data type; then Oracle will tell you whether it can do the conversion. If it can, it returns one. Otherwise you get zero. To return the rows in the table that can be real dates, place this in your where clause:

```
select t.*
from   dodgy_dates t
where  validate_conversion(is_this_a_date as date) = 1;

ID         IS_THIS_A_DATE
---------- --------------------
3          01-jan-2016
```

There's no error. But where did rows 2 and 4 go? They're possible dates too. Validate_conversion only tests one date format at a time. By default this is your NLS_date_format. Each client can set their own format. If you rely on this, you may get unexpected results. To avoid this, I strongly recommend that you pass the format as a parameter. For example:

```
select t.*
from   dodgy_dates t
where  validate_conversion(is_this_a_date as date, 'yyyymmdd') = 1;

ID         IS_THIS_A_DATE
---------- --------------------
2          20150101
```

In order to return all of the possible dates, you'll need to call this multiple times:

```
select t.*
from   dodgy_dates t
where  validate_conversion(is_this_a_date as date, 'yyyymmdd') = 1 or
       validate_conversion(is_this_a_date as date, 'dd/mm/yyyy') = 1 or
       validate_conversion(is_this_a_date as date, 'dd-mon-yyyy') = 1;

ID         IS_THIS_A_DATE
---------- --------------------
2          20150101
3          01-jan-2016
4          01/01/2016
```

This isn't just for dates. You can use validate_conversion with any of the following data types:

- binary_double
- binary_float
- date
- interval day to second
- interval year to month
- number
- timestamp
- timestamp with time zone

If you want to convert strings to dates, you'll need similar logic in the select. This will test the expression against various format masks. If it matches, call to_date with the relevant mask:

```
case
  when validate_conversion(is_this_a_date as date, 'yyyymmdd') = 1
  then to_date(is_this_a_date, 'yyyymmdd')
  when validate_conversion(is_this_a_date as date, 'dd/mm/yyyy') = 1
  then to_date(is_this_a_date, 'dd/mm/yyyy')
  when validate_conversion(is_this_a_date as date, 'dd-mon-yyyy') = 1
  then to_date(is_this_a_date, 'dd-mon-yyyy')
end
```

This is clunky, but fortunately 12.2 has more functionality to support data type conversions.

Handle Casting Conversion Errors

From time to time you'll want to cast a value to a different data type. This can bring problems if your values are incompatible with the desired type. You could overcome this with the validate_conversion function we discussed above, but there is another way. Cast now has a "default on conversion error" clause. This specifies which value Oracle returns if it can't convert the expression to the type you wanted. For example, say you're attempting to cast a varchar2 column to a date, but it happens to include the value "not a date." You'd get a nasty error:

```
select cast ( 'not a date' as date )
from   dual;

ORA-01858: a non-numeric character was found where a numeric was expected
```

With the new clause you can tell Oracle to return a "magic date" instead of throwing an exception. For example:

```
select cast (
         'not a date' as date
         default date'0001-01-01' on conversion error
       ) dt
from   dual;

DT
--------------------
01-JAN-0001 00:00:00
```

You can then add checks to your code for this magic value. Note that the default value has to match the data type you're converting to. If you're casting to a date, you can't return a string:

```
select cast (
         '01012010' as date
         default 'not a date' on conversion error
       ) dt
from   dual;

ORA-01858: a non-numeric character was found where a numeric was expected
```

And, as with validate_conversion, cast uses your NLS settings for the default format. If you want to override these, pass the format as a parameter:

```
select cast (
         '01012010' as date
         default '01010001' on conversion error,
         'ddmmyyyy'
       ) dt
from   dual;

DT
--------------------
01-JAN-2010 00:00:00
```

This is neat, but at first glance it seems, well, limited. After all, how often do you use cast? If you're like me, the answer is "rarely." But there's more to it than that: the conversion error clause also applies to other casting functions, such as:

- to_date()
- to_number()
- to_yminterval()

That's *really* useful. These are functions you use all the time. Now you can write data type conversions like this:

```
select to_date(
         'not a date' default '01010001' on conversion error,
         'ddmmyyyy'
       ) dt
from   dual;

DT
--------------------
01-JAN-0001 00:00:00
```

Combining this with validate_conversion makes changing expressions to a new data type much easier.

Top Feature IX—Single Statement Table Partitioning

Here's a question that frequently comes up on Ask Tom: How do I convert a non-partitioned table to a partitioned one? Before 12.2 this was a convoluted process. You had to create a partitioned copy of the table and transfer the data over. You could use DBMS_redefinition to do this online, but it was a headache and easy to get wrong. In Oracle Database 12*c* Release 2 it's easy. All you need is a single alter table command:

```
create table t ( x int, y int, z int );

alter table t modify
partition by range (x) interval (100) (
  partition p1 values less than (100)
) online;
```

And you're done! "But what about all the indexes?" I hear you cry. Well, you can convert them too! Just add an update indexes clause and state whether you want them to be local or global after the conversion:

```
create index iy on t (y);
create index iz on t (z);

alter table t modify
partition by range (x) interval (100) (
  partition p1 values less than (100)
) update indexes (
  iy local,
  iz global
);
```

If you really want to, you can give your global indexes different partitioning schemes. While you can change from a non-partitioned table to partitioned, you can't go back again. You also can't change the partitioning scheme—e.g., go from list to range. Try to do so and you'll get:

```
ORA-14427: table does not support modification to a partitioned state DDL
```

But if you want to get really fancy, you can go directly from a normal table to one with subpartitions.

```
alter table t modify
partition by range (x) interval (100)
  subpartition by hash (y) subpartitions 4 (
    partition p1 values less than (100)
) online;
```

And there are even more improvements to partitioning, as we'll see.

Top Feature X—Automatic List Partitioning

List partitions are great when you have a column with a specific set of values that you want to carve into separate partitions. Values like states, countries, and currencies are all good examples. Reference data like these rarely change, but they do change. For example, South Sudan came into being in 2011.

If you list partitioned your data by country, you need to keep your partitions up to date—particularly if you let customers provide their own values—or you could end up with embarrassing errors such as:

```
SQL Error: ORA-14400: inserted partition key does not map to any partition
```

This, of course, will happen at 2 a.m.—and it's a great way to incur the wrath of the on-call DBA. To avoid this you could create a default partition, and any new values would then go into it. This will prevent inserts from throwing exceptions, but all new values go into the default partition. Over time this partition would fill up with all the new values.

You need a regular maintenance task to split values out as needed. 12.2 resolves this problem with Automatic List Partitioning. Every time you insert new values, Oracle will create the new partition on the fly. To use it, simply place the automatic keyword after the partition column:

```
create table orders (
  customer_id      integer not null,
  order_datetime   date not null,
  country_iso_code varchar2(2) not null
) partition by list (country_iso_code) automatic (
  partition pUS values ('US'),
  partition pGB values ('GB'),
  partition pDE values ('DE'),
  partition pFR values ('FR'),
  partition pIT values ('IT')
);

insert into orders values (1, sysdate, 'ZA');

select partition_name
from   user_tab_partitions
where  table_name = 'ORDERS';

PARTITION_NAME
--------------
PDE
PFR
PGB
PIT
PUS
SYS_P1386
```

Each new partition will have a system-generated name, but you may want to change them to meaningful names. You can do this with:

```
alter table orders rename partition SYS_P1386 to pZA;
```

Be aware, however, that the default partition and automatic list partitioning are mutually exclusive options:

```
create table orders (
  customer_id      integer not null,
  order_datetime   date not null,
  country_iso_code varchar2(2) not null
) partition by list (country_iso_code) automatic (
  partition pUS values ('US'),
  partition pGB values ('GB'),
  partition pDE values ('DE'),
  partition pFR values ('FR'),
  partition pIT values ('IT'),
  partition pDEF values (default)
);

SQL Error: ORA-14851: DEFAULT [sub]partition cannot be specified for AUTOLIST [sub]
partitioned objects.
```

This makes sense when you think about it, but if you want to migrate list partitions with a default to automatic, you'll need to go through a process. First split everything out of the default partition, and then drop it:

```
create table orders (
  customer_id integer not null,
  order_datetime date not null,
  country_iso_code varchar2(2) not null
) partition by list (country_iso_code) (
  partition pUS values ('US'),
  partition pGB values ('GB'),
  partition pDE values ('DE'),
  partition pFR values ('FR'),
  partition pIT values ('IT'),
  partition pDEF values (default)
);

insert into orders values (1, sysdate, 'ZA');
insert into orders values (2, sysdate, 'JP');

alter table orders split partition pDEF into (
  partition pZA values ('ZA'),
  partition pJP values ('JP'),
  partition pDEF
);

alter table orders drop partition pDEF;

alter table orders set partitioning automatic;
```

Note that this leaves a brief time when there's no default partition, and automatic partitioning isn't ready. You may want to take a short outage to do this.

Top Feature XI—Mark Old Code as "Not For Use"

Times change. New code quickly becomes legacy code, and legacy code is often superseded by better, faster code. So, you deprecate the old code, but this creates a problem: How do you stop people from using the legacy modules?

People tend to stick with what they know. Even after you've repeatedly told everyone to move to the new module, there's always (at least) one developer who insists on using the deprecated procedure instead of the newer, shinier option—and in complex applications it's tough to keep track of what's obsolete.

This problem is tough to solve. In order to help you with the deprecation process, we've introduced a new pragma for this. To use it, place

```
pragma deprecate ( deprecated_thing, 'Message to other developers' );
```

below the retired section. Great—but how does it help? We've added a bunch of new PL/SQL warnings: PLW-6019 to PLW-6022. Enable these and Oracle will tell you if you're using deprecated code:

```
alter session set plsql_warnings = 'enable:(6019,6020,6021,6022)';
create or replace procedure your_old_code is
  pragma deprecate (
    your_old_code, 'This is deprecated. Use new_code instead!'
  );
begin
  null;
end your_old_code;
/
show err

Warning(2,3): PLW-06019: entity YOUR_OLD_CODE is deprecated
```

This is great, but we've all been ignoring the "AUTHID DEFINER" warning forever. If code is truly obsolete, it would be good if you could keep people from using it altogether. Fortunately, you can. Here's the great thing about warnings: you can upgrade them to be errors. PLW-6020 is thrown when you write code calling a deprecated item. Set this to error and the offending code won't compile:

```
alter session set plsql_warnings = 'error:6020';
create or replace procedure calling_old_code is
begin
  your_old_code();
end calling_old_code;
/
sho err

3/3 PLS-06020: reference to a deprecated entity: This is deprecated. Use new_code
instead!
```

Of course, if you turn PLW-6020 into an error systemwide, a lot of stuff might break. Luckily, you can selectively upgrade it on given objects:

```
alter procedure calling_old_code compile plsql_warnings = 'error:6020' reuse settings;
```

Now you have the power to force others to stop using prehistoric code.

Top Feature XII—PL/SQL Code Coverage

We've covered a lot of new functionality. Some of it you'll use straight away; other bits will wait a while. In any case, when you upgrade to 12.2, you'll want to test all your code to ensure that it works as expected, which raises the question, "How much of my code did the tests actually run?"

Coverage metrics will help immensely with this. Simple line-level analysis of the tests isn't good enough. To see why, consider the code below. We have a basic function that returns its argument and calls dbms_output. The procedure calls the function twice in a single if statement:

```
create or replace function f (p int)
  return int as
begin
  dbms_output.put_line('Executed: ' || p);
  return p;
end;
/

create or replace procedure p is
begin
  if f(1) = 1 or f(2) = 2 then
    dbms_output.put_line('this');
  else
    dbms_output.put_line('that');
  end if;
end p;
/
```

Due to short-circuit evaluation, f(2) is never executed. You can see this from the output:

```
SQL> exec p;

Executed: 1

this
```

Anything working at the line level will incorrectly report this as fully covered. To overcome this, you need details of basic block executions. A "basic block" is a piece of code that either runs completely or not at all. Code always belongs to exactly one basic block. The following example has four basic blocks, one for each call to f and two for the calls to dbms_output.put_line:

```
  if f(1) = 1 or f(2) = 2 then
    dbms_output.put_line('this');
  else
    dbms_output.put_line('that');
  end if;
```

The new code coverage functionality measures and reports on these basic blocks. Using it is easy. First you need to create coverage tables to store the metrics:

```
exec dbms_plsql_code_coverage.create_coverage_tables;
```

Then call start_coverage before your test and stop_coverage after:

```
declare
  run_id pls_integer;
begin
  run_id := dbms_plsql_code_coverage.start_coverage('TEST');
  p;
  dbms_plsql_code_coverage.stop_coverage;
end;
/
```

You can then get metrics by querying the dbmspcc* tables that hold these details:

```
select owner, name, type,
       round( ( sum(covered)/count(*) * 100), 2) pct_covered
from   dbmspcc_runs r
join   dbmspcc_units u
on     r.run_id = u.run_id
join   dbmspcc_blocks b
on     r.run_id = b.run_id
and    u.object_id = b.object_id
where  r.run_comment = 'TEST'
group  by owner, name, type;

OWNER  NAME  TYPE       PCT_COVERED
-----  ----- ---------- -----------
CHRIS  P     PROCEDURE  50
CHRIS  F     FUNCTION   100
```

This is all well and good, but there's always some code that your tests don't cover. Maybe it's deprecated, so you don't need test it, or it's just-in-case code to cover theoretically possible but practically impossible cases— such as the infamous "when others" exception handler. These sections should be excluded from your reports. Fortunately you can do this with the coverage pragma. By marking lines as NOT_FEASIBLE, you can filter them out of your reports:

```
create or replace procedure p is
begin
  if f(1) = 1 or f(2) = 2 then
    dbms_output.put_line('this');
  else
    pragma coverage ('NOT_FEASIBLE');
    dbms_output.put_line('that');
  end if;
end p;
/
```

Rerun the tests and you can hide the untestable parts in your report.

```
select owner, name, type,
       round( ( sum(covered)/count(*) * 100), 2) pct_covered
from   dbmspcc_runs r
join   dbmspcc_units u
on     r.run_id = u.run_id
join   dbmspcc_blocks b
on     r.run_id = b.run_id
and    u.object_id = b.object_id
where  r.run_comment = 'TEST'
and    b.not_feasible = 0
group  by owner, name, type;

OWNER  NAME  TYPE       PCT_COVERAGE
-----  ----  ---------  ------------
CHRIS  P     PROCEDURE  66.67
CHRIS  F     FUNCTION   100
If you really want to, you can exclude whole sections of code by wrapping it in two
coverage pragmas: NOT_FEASIBLE_START and NOT_FEASIBLE_END:
begin
  pragma coverage ('NOT_FEASIBLE_START');
  a_section();
  of_untestable_code();
  pragma coverage ('NOT_FEASIBLE_END');
end;
/
```

Wrap Up

Do you want to get your hands on Oracle Database 12*c* Release 2? Head over to the database options on Oracle Cloud. There you can get access with Oracle Database Cloud Service or Exadata Cloud Service, or you can let us look after your database with the Exadata Express Cloud Service. Alternatively you can download it from OTN.

Chris Saxon is an Oracle Developer Advocate for SQL. He blogs at All Things SQL and creates YouTube videos combining SQL and magic at The Magic of SQL. Reach out to Chris via Twitter or on Ask Tom.

© 2018 Chris Saxon

Where in the World is Kerry Osborne? Season 31 Episode 4

Kerry Osborne fans were delighted when Kerry Osborne suddenly showed up to attend Tanel Põder's workshop at the fall conference in downtown Oakland.

Thanks to Axxana for hosting the post-conference happy hour at The Trappist. Can you spot Kerry in his trademark baseball cap?

NoCOUG
P.O. Box 3282
Danville, CA 94526

RETURN SERVICE REQUESTED

LIVE VIRTUAL CLASSES

TAUGHT BY
CRAIG SHALLAHAMER

- Oracle Tuning Fastpath
- Oracle Buffer Cache Performance Analysis And Tuning
- Tuning Oracle Using An AWR Report
- Tuning Oracle Using Advanced ASH Strategies

Use coupon code **NOCOUG10** for 10% off!
Questions? Contact support@orapub.com.

ORAPUB

REGISTER TODAY

Find out more at orapub.com.