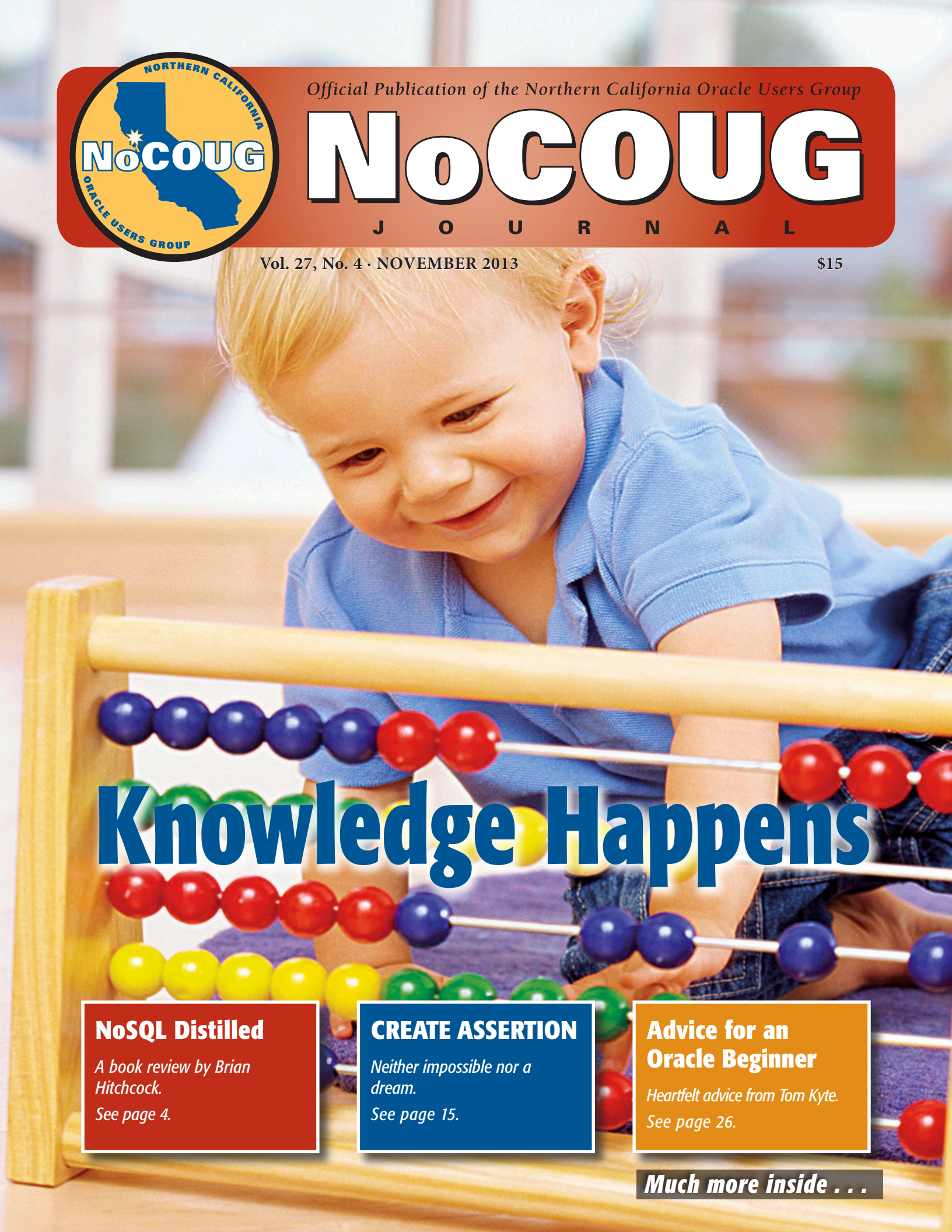
Official Publication of the Northern California Oracle Users Group

NoCOUG JOURNAL

Vol. 27, No. 4 · NOVEMBER 2013 $15



Knowledge Happens

NoSQL Distilled

A book review by Brian Hitchcock.

See page 4.

CREATE ASSERTION

Neither impossible nor a dream.

See page 15.

Advice for an Oracle Beginner

Heartfelt advice from Tom Kyte.

See page 26.

Much more inside . . .

DBMoto® Data Replication and Change Data Capture

- Need fast data updates?
- Have multiple databases or analytic systems?
- Don't have time for complicated applications?
- **FREE** Advice for Real-time Data

FAST, EASY, AFFORDABLE

FREE PAPER:

"Tips for Real-time Data and Reporting"
http://info.hitsw.com/nocoug8

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

ORACLE Gold Partner

HiT SOFTWARE®

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

This is the 108th issue of the *NoCOUG Journal*. Enjoy! ▲

—*NoCOUG Journal* Editor

Table of Contents

ADVERTISERS

Publication Notices and Submission Format

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

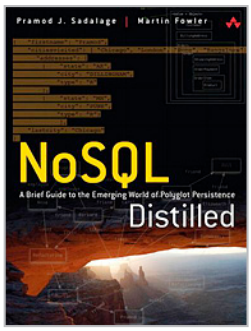
Please send your questions, feedback, and submissions to the *NoCOUG Journal* editor at **journal@nocoug.org**.

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

NoSQL Distilled

A Book Review by Brian Hitchcock

Details



Author: Pramod J. Sadalage and Martin Fowler

ISBN: 978-0-321-82662-6

Pages: 192

Year of Publication: 2012

Edition: 1

List Price: $39.99

Publisher: Addison-Wesley

Overall Review: Excellent; a fantastic introduction to a new world of databases.

Target Audience: Anyone involved with databases and the applications that depend on them.

Would you recommend this book to others: Yes, without question.

Who will get the most from this book? Those that are familiar with RDBMS products.

Is this book platform specific: No.

Why did I obtain this book? NoCOUG asked me to review this book, and they procured a copy of it for me.

Overall Review

I have worked with relational database management systems (RDBMSs) for a long time. This means I'm completely in the "YesSQL" world. Therefore, I could be among the most resistant to change. This book is wonderful in that it doesn't talk down to me about the "old ways." It simply explains in very clear terms exactly what NoSQL is and what it isn't. Most importantly, this book doesn't pretend that RDBMSs are going away or that they aren't still the best solution for many situations. Moreover, this book doesn't waste your time. It has information to communicate and it does so quickly, with clear examples. This is what technical writing should be: just long enough to make the points and get it done.

Preface

The authors start off explaining that the term "NoSQL" is ill-defined but usually refers to a number of nonrelational databases. The term also refers to schemaless data and systems where gains in performance are traded against other things like consistency. Thankfully, they are quick to point out that relational systems are still very good at many tasks. They then introduce the term "polyglot persistence," which is a world in which relational is not the only way to store and manage data.

A good explanation of why the reader should care about NoSQL databases is next. The authors quickly introduce the two main reasons: first, application development productivity and second, large-scale data. Along the way the fact that large datasets are usually run on clusters of servers is also brought up. These core concepts will be explained in detail as the book progresses.

After this we have a description of what is in the book, and I would recommend that you read these two pages whether or not you read the rest of the book, although I very much recommend reading all of it. We also see the usual description of who should read this book and a list of the specific NoSQL databases that are discussed in the book. I like the way the book is divided into two main sections. The first, "Understand," discusses what you need to understand about NoSQL and how it is different (and, in my opinion, how it is not different) from the RDBMS world. The second, "Implement," presents some details of how the various products operate, including some code snippets.

Chapter 1—Why NoSQL?

This chapter starts with a review of where we are and how RDBMSs came to run the world. In passing they mention object databases. Funny, I remember when object databases were going to take over, but it never happened. Strange how some "new things" come and go and others come and conquer. The section titled "The Value of Relational Databases" lays out the reasons that RDBMSs are so prevalent. This provides a straightforward context for the comparisons to come with NoSQL.

Next we learn about the "impedance mismatch," which is defined as the difference between the relational model and various in-memory data structures. I hadn't thought about it this way before; this illustrates the value of reading about new stuff. The best way to better understand your existing RDBMS is to learn about NoSQL systems. The mismatch is between how data is stored in the relational database as opposed to how it is used by the applications that build in-memory data structures.

We are told that there is a growing divide between application developers and database administrators. I thought it was a divide between DBAs and everyone else. No matter. My advice is simple and effective: Whatever side of this perceived divide you are on, offer to buy lunch for those on the other side(s). You will find the divide becomes vanishingly small.

There are several very good sections, including "Application and Integration Databases" and "Attack of the Clusters" ("Do not run—we are your Friends!"). The emergence of NoSQL is another good section. I think everyone should read this. At the end of each chapter we find "Key Points" that are well worth reviewing.

Chapter 2—Aggregate Data Models

This is where it starts to get real. This is where I had to start really thinking about what I was reading. I like this line: "A data

model is the model through which we perceive and manipulate our data." This leads to relational tables being the default data model. Each of the NoSQL solutions has a different data model. I'm getting ahead of myself here, but I immediately wonder how we will support all these different data models in one organization.

First we have a discussion of aggregates and an example comparing data stored in a relational system and a NoSQL system that uses the aggregate data model. The point is that data is stored in groups (the aggregate) instead of in normalized tables. Everything about one customer could be stored in one aggregate instead of spread out among many relational tables.

Next are the key-value and document data models. Now this NoSQL thing is starting to get complicated. I'm liking the aggregate data model and now two more arrive to confuse me. The key-value has a unique key that is associated with a (wait for it) "blob." I kid you not. The precise definition of this blob is as follows: "some big blob of mostly meaningless bits." Wow!

Then we have the document database, which I assumed (wrongly) was a database of documents. Well, sort of. Turns out a "document" can be any set of data, perhaps an XML file.

Then we have column-family stores, which look to me like groups of aggregates with a key value. The Key Points at the end of the chapter are great.

As we learn about more and different NoSQL data models, we find that some have indexes and some don't, and some have query languages and some don't. NoSQL is not a specific product or technique. It is a different way of looking at your data. Not all of your data has to be in a single data model. Again, I wonder about the support issues.

Chapter 3—More Details on Data Models

We learned about aggregates in the previous chapter. Now we look at how to deal with relationships between aggregates. If you have all the information for a customer in one aggregate record and all the orders for that customer in another aggregate, how would you find related pieces of data? Here we find that document databases offer some unique query features. We also see that the different data models have various issues with standard data manipulations such as updates. It really is getting more complicated as we move along. Which NoSQL data model is best for you and yours? As always, it depends. I like the summary of what happens when you need to have many relationships between multiple aggregates: "Things quickly get very hairy." Indeed, and back to that support issue someone keeps mumbling about.

For relationships, graph databases are very good. Anyone that has ever had a product recommended to them based on what they just saw on a website has been working with a graph database. The way relationships can be one way (I've always really liked her but she doesn't even know I exist) amused me. The potential for parody is huge.

Next we look at schemaless databases. With a relational database you must define a schema before you can store any data. For NoSQL database, and I'm quoting here, "storing data is much more casual." For me, the core of NoSQL is right here. Things that have been absolutely unthinkable before (do I really care if this write completes?) are now much more casual. Sometimes you want commitment; sometimes you don't. In the NoSQL world of tomorrow, you will have those that prefer the rigidity of relational (and you know who you are!) and those that are more "casual." It is explained however, that no matter how casual you want to be, there is an implicit schema in the data. Even if your database doesn't require a schema, your application still has to make some sense of the data.

This chapter also covers materialized views in the NoSQL world and discusses modeling data from a data access point of view.

Chapter 4—Distribution Models

Here we learn about distributing data across multiple nodes in a cluster. While this may seem like a strange idea at first, it is very similar to partitioning a relational table, except that each partition lives on a separate server. Strange, but not as different as it first appears. This is one of the main drivers of NoSQL, i.e., managing huge datasets on clusters of relatively small (cheap?) computers. Insert the usual cluster marketing presentation here. You can add nodes, nodes are cheap, etc. The dark side is presented right away, as we are told that running over a cluster adds complexity and is not to be done without a very good reason.

There are two data distribution models: replication and sharding. Replication copies the same data to multiple nodes, while sharding puts different data on different nodes. I like the description of replication and sharding as "orthogonal," which reminds me of vector calculus and simply means that you can use either or both.

This chapter then moves on to discuss the various forms of distribution starting with single-server followed by master-slave replication, sharding, and then peer-to-peer replication. I was pleased to read that the authors recommend that you start with the simplest distribution option, which is no distribution at all. NoSQL provides many new things, but don't use them unless you have a good reason. It is also noted that no distribution is easier for "operations people" to manage. I think I'm in that group. We like easy. We don't see it nearly as often as we would like.

Also, note that using NoSQL on a single server appears counterintuitive, since we hear so much about clusters of servers. The needs of your application and your business should always be more important than any new (or old) dogma. I've heard about sharding for a long time, and in this chapter I got the first clear explanation I've seen. The diagrams are simple, succinct, and even pretty. Successfully communicating complicated ideas in words and a few diagrams is an art form—an art form that is very successfully executed here. There is much more to learn from this chapter, culminating in a discussion of combining sharding and replication.

Chapter 5—Consistency

This is a new idea for me. I'm used to my world revolving around the production relational database, which is the system of record. I had to read carefully to keep up. While relational systems provide strong consistency, NoSQL systems bring up the terms "CAP theorem" and "eventual consistency." Even the definition of consistency requires discussions covering update and read consistency. When you run with your data spread across multiple nodes, you have issues because data that should be the same on multiple nodes may not be. Or, may not be for some amount of time.

I think this is one of the central ideas of NoSQL. Strip away all the product specifics and you have a new world where the result of your query may vary over time, and that is okay. The details come thick and fast as we see pessimistic and optimistic concurrency, conditional updates, and more. We are challenged to justify our conditioning, which tells us that we must avoid conflicts caused by anything less than total consistency.

The diagrams in this chapter are very good and really help explain what is going on. Even if you aren't swayed by the NoSQL story, the explanation of all the consistency issues is worth reading; it's very clear and concise.

We can all agree that consistency is a good thing, but rather than assuming that we must have it, we need to discuss how much we need and why. There are benefits to what is described as relaxing consistency. It is pointed out that even with transactions in a relational system, you have a choice of isolation levels, which are a form of relaxed consistency. More real-world goodness comes when we read that many systems need to give up some consistency because full-on transactions simply cost too much in terms of performance.

The CAP theorem is something I've never heard of before. While I'm not a theorem person, this is the only one in the whole book and it is well worth thinking about. It tells us that, out of the three properties consistency, availability, and partition tolerance (CAP), you can only get two. You need to read this chapter to appreciate what it all means. I'm struck by how complex the whole NoSQL "thing" is. It is way more involved than saying no to SQL.

Chapter 6—Version Stamps

This chapter opens with an interesting observation. NoSQL is criticized for not having transactions, but, if all the data you need is in a single aggregate, data manipulation on an aggregate is pretty close to a transaction. It is also observed that even with transactions, there are business processes that would take much longer than any real system can wait. Version stamps are a way to deal with this. If data has a version stamp, you can detect any consistency issues by comparing the version stamps. To me this is the same, or at least very similar to, timestamps. As long as your version stamps are increasing over time, you can tell in what order various data changes happened. This actually opens up some new possibilities. If you can, in theory, re-create any past state of your data, you can go back in time and look at different things.

When discussing the issues around business transactions that take too long for a single transaction, the authors refer to choosing a bottle of Talisker as a likely example. I had to Google this. Talisker is the only whisky distillery based on the Scottish island of Skye. Perhaps this fascination with whisky explains their "flexibility" with all things relational? Or, perhaps you have to travel to an island in Scotland to get far enough away from the relational empire to see that there are other possibilities?

We move on to discuss how to handle version stamps on multiple nodes. I've said it many times, but NoSQL is a lot more complicated than it appears. When you have a single server that controls the generation of the version stamps, things are pretty clear. Multiple nodes lead us to vector stamps, which are sets of counters, one for each node. The nodes then synchronize their vector stamps as needed. This allows determination of who did what, where, and when.

Chapter 7—Map-Reduce

So far we have seen that NoSQL databases store data differently from relational systems through sharding and replication. Along with this change in the way data is stored, the way processing is done changes as well. With a cluster of servers, you have many nodes across which to distribute the work. At the same time, you need to reduce the amount of data that needs to be shipped between nodes as the processing is done. The more you have to move data between nodes, the more you reduce the performance improvements due to parallel processing on multiple nodes. Ideally, you would process all the data on the node where the data is located. Map-reduce is a way to use multiple servers and keep the processing and the data needed by the processing together on the same machine. Again, the explanation and the diagrams presented are great. Although this is a small book, its graphics are much better than most of the other technical books I've read. The example given has aggregates for customers and orders. When you need to answer questions about total revenue for a given product, you need to look at a lot of aggregates. The first step is to map, which is a function that examines a single order aggregate and outputs a set of key-value pairs in which each pair is a line item from the order. Since the order aggregates are sharded across multiple servers, this map processing of each order can be done on each server on the orders stored there. This means that all the map processing is localized to a single server, which enables parallelization.

The reduce function takes multiple map outputs that have the same key and combines the values from those key-value pairs. The output of the reduce operation gathers all the map outputs to generate the query results. Further refinements to the basic map-reduce processing include partitioning and combining, which further increases parallelism and reduces how much data must be moved between nodes.

Chapter 8—Key-Value Databases

The previous chapters dealt with understanding the concepts behind NoSQL. Now we learn about some of the issues involved in actually building NoSQL systems. Since I am not about to implement NoSQL, I didn't get as much from these chapters. This is not a criticism in any way. These chapters move fast, which is good, and a lot of code snippets are presented. I am not a developer, so I can't comment much on code in any form. I do think many readers will get a lot from the code that is discussed.

This chapter discusses details of a key-value store. Specific examples are Riak, Redis, and Berkeley DB. I have heard about Berkeley DB for many years but I never realized that it was NoSQL. I personally look forward to the time when I will be supporting HamsterDB. If nothing else, the names of all the NoSQL products are so much more interesting than in the relational world. The features of key-value stores that are covered include consistency, transactions, query features, data structures, and scaling. In a key-value store you can only query on the key. For anything else you have to handle the query in the application code. This aspect seems pretty limiting to me. It seems like a point solution that is very specific to solving one sort of problem. This is not useful for general-purpose database processing. The design of the key is discussed. I had never heard of "curl," which is a command-line tool for transferring data with URL syntax. There is a lot to learn about the NoSQL world.

The chapter ends with a list of suitable user cases and a section titled, "When not to use." For someone like me who is new to the subject, these sections are great.

Chapter 9—Document Databases

Here we learn that the documents stored in document databases can be XML, JSON, BSON, and other self-describing structures. Each document stored is the value part of a key-value pair. Unlike key-value stores we have seen previously, the documents (the "values") can be examined. A table compares the terminology used by Oracle and a document database called MongoDB. A document in a document database is compared to a row in a relational table. You can add new attributes to a document without the need to make changes to any of the other documents. This is very different from the relational world, in which all the rows of a table must have all the same attributes, even if many of them are NULL. In passing, the authors tell us that Lotus Notes is "reviled." So much I have to learn! Coverage of the features of document databases includes consistency, transactions, availability, query features, scaling, and—again—suitable use cases and when not to use a document database. In these sections I learned that MongoDB has a query language that supports "where" and "order by" clauses as well as "explain," which shows the execution plan. NoSQL is *not* no SQL! I was glad to see sharding compared to partitioning in the relational world. I didn't realize that NoSQL databases dynamically move data between nodes as part of sharding to maintain balance. One of the suitable use cases is event logging. This is the first time that an example of NoSQL has really made sense to me, because I look at a lot of log files for the multiple components of Fusion Middleware. They are all different in layout and size. Having one place where all these log files could be stored would be great.

Chapter 10—Column-Family Stores

This chapter covers the specifics of Cassandra, Hbase, and Amazon SimpleDB. A comparison is made between Oracle and the terms used to describe the components of the Cassandra database. Diagrams are presented to illustrate the data model used. In the section on consistency we see that stale data is handled by "read repair," and that it is okay if some writes are lost. This just sounds wrong. I know it's correct in the context of NoSQL, but the training in the "old ways" is hard to overcome. Further, we are told that the system designers will need to "tune the consistency" as the application requirements change. This sounds like a big support cost that comes with NoSQL. Perhaps I just don't see the equivalent issue in the relational world. In the transactions section I learn that a commit log is used to apply changes if a node is lost, just like the redo log in Oracle. Funny how many RDBMS features are popping up in the NoSQL world! You can use external transaction libraries, such as ZooKeeper. Where do we keep our transactions? In the zoo of course! On the weekend we can take the kids to the zoo to see the transactions in their "habitats." The query language section tells us that Cassandra has a query language that supports SQL-like commands. Yet again, NoSQL doesn't mean not using SQL. One of the suitable use cases presented is for expiring usage, where you make use of expiring columns that are automatically deleted after a given time. This is useful for data that is being used for a customer demo or for banner ads on websites.

The chapter ends with an example of when not to use a column-family store. Specifically, Cassandra is not good for prototypes in which the column family design is changing rapidly. This slows down developer productivity.

Chapter 11—Graph Databases

Graph databases are, in my opinion, the one kind of NoSQL database that is the furthest from my relational experience and therefore the most interesting. The description that opens this chapter is great. Graph databases store entities and the relationships between them. The entities are also called "nodes" and have properties. Another way to look at this is that each node is an object in your application. The relationships between these objects are the "edges" of the graph and these edges have properties. One of these properties is "directional significance." Once you have all these relationships in the database, you can search for patterns. This is obviously useful for social networks and recommending products based on previous sales, etc. Querying the graph database is called "traversing the graph." It is explained that a relational database can store a relationship such as the canonical employee–manager relationship. However, trying to add a second relationship would require significant schema changes.

The section covering scaling provided a fascinating insight. It turns out that for graph databases, sharding is difficult. Hang on a minute, I thought sharding was one of the central ideas of NoSQL databases. As we have seen before, NoSQL is a diverse set of new ideas and products. It is difficult to say exactly what NoSQL is and what it isn't. There are special challenges to scaling graph databases that I found interesting.

Graph databases are the obvious choice for social networks, location-based services, and recommendations. At the same time, it is very difficult to update all the nodes, so they aren't a good choice for applications in which the properties of a large number of the application objects are changing often.

Chapter 12—Schema Migrations

Part of the hype around NoSQL databases is that they are "schemaless." Setting aside, for the moment, what exactly that means, it is true that NoSQL databases are much more flexible when it comes to making changes to the structure of the data. At the same time, the process of making changes can be expensive. This chapter reviews how a relational database requires a schema to be defined before any data can be stored. Changes to the schema require many changes to tables and other database objects. In general, a relational database must be changed before an application can be changed. NoSQL databases try to avoid this by offering more flexibility with respect to schema changes. As we examine this topic, it becomes more complicated. The important point is made that even if you accept that NoSQL databases really are schemaless, someone has to make sense of the application data, and this requires some form of a schema. This may all be handled by the application code, but it is still handled somewhere. I'm glad the authors refer to the use of the term "schemaless" as misleading. The application has to read the data from the database and make sense of it. That process is a schema, even if we say that the database itself does not impose a schema on the data stored inside.

Further, while it is true that a NoSQL database does not impose schema requirements the way a relational system does, it is also true that any change to data structure does require a change

to the application code. The application has to be able to make sense of the data that has and has not changed. NoSQL databases don't eliminate the impact of schema changes, they move them out of the database.

The impacts of schema changes to graph databases and aggregate structure are discussed.

Chapter 13—Polyglot Persistence

This chapter presents a very good idea that needs to be heard much more widely. As we have seen, the various NoSQL databases are designed to solve very different, very specific problems. Any real-world business system is going to need different NoSQL databases to support different aspects of the business. For example, the same business may need one NoSQL database for its shopping cart and another for its social networking. Trying to get both from any single database, NoSQL or relational, is not going to be optimal. A great point is made that this isn't unique to the NoSQL world. In the relational world, OLTP and OLAP processing are often forced to coexist in a single database.

The idea here is that what we need isn't relational or NoSQL, but a combination of whatever persistence is best for the specific problem we need to solve. The term "polyglot persistence" is cumbersome but very powerful. We need to stop looking for a single database that will do all things for everyone.

This concept is then extended to web services. We look at the world as if each application has to have its own dedicated database. As we see that different databases are best for different kinds of data storage and processing, we should move to a world where we have multiple data stores that applications use to get the storage and processing they need.

This chapter really helped me understand why web services are a big deal. My working experience has always been one application talking to one database. As more data is available through web services, we need to change our outlook. Further, this shift will have a big impact on our organizations. Who is responsible to the users of our applications when they depend on multiple data stores? How will we support this polyglot persistence? How will a business find the resources needed to support all the different databases that we will need?

Chapter 14—Beyond NoSQL

Now that we have seen a lot of information about NoSQL databases, it is interesting to drop back and look around at how much NoSQL stuff we already have in our organizations. I had never thought of a file system as a database, but it is. The comparison is fascinating. File systems don't impose any structure on the data that is stored in any given file. There is a key-value relationship to each file. There is little control over concurrency beyond file locking. This is very similar to NoSQL, with locking only at the aggregate level. File systems are cheap; everyone has one and they hold huge amounts of data on multiple nodes.

Next up is "event sourcing." I had never heard of this. The idea is that you keep all the changes that were ever made to the data. I found this very interesting. If you can create any state of any data at any time in the past, you don't have to worry so much about storing the current state of the application.

Other ideas discussed include memory image, where the application state is only maintained in memory. This has obvious performance benefits. It also simplifies application programming, because there is no need to translate data structures from memory to a persistent store (disk). This is followed by version control, XML databases, and object databases.

This chapter is very interesting and closes with what I think may be the single best thing I got from this book: As we all get more comfortable with the idea of polyglot persistence, we need to look around for anything that helps us solve problems, whether it is being marketed as NoSQL or not.

Chapter 15—Choosing Your Database

A lot of material has been presented about NoSQL databases. Now is the time to put it all into practice. How exactly do you decide which database to use? The two main benefits of using NoSQL databases are presented as programmer productivity and performance. Each is discussed in detail. Highlights include the statement that all NoSQL systems are better for nonuniform data, and that there is no way to really measure programmer productivity. If we can't quantify one of the two main benefits of using NoSQL databases, how do we sell them to our organizations? Similarly, there are difficulties when trying to quantify the specific performance benefits of one NoSQL database over another.

This leads to a very important paragraph in which we are reminded that, in the majority of cases, relational is still the best option. I wasn't expecting this to be stated so clearly in a book about NoSQL, but I'm glad it was. I'm not opposed to NoSQL, but I am opposed to organizations moving to the "next big thing" just because they want to look cool. I agree with the authors that you need to be able to show a real advantage to using NoSQL before you move away from relational.

In closing, we are told that most people won't be moving to NoSQL anytime soon. I think this is very true. NoSQL has some definite advantages, but they come with a set of tradeoffs. Time will tell which aspects of NoSQL demonstrate real value in the real world.

Conclusion

This book is well worth your time. Many new ideas are clearly explained. This book is what all technical books should be: clear, precise, and short. I enjoyed reading this book, and I think everyone who works with relational databases needs to be aware of the ideas presented. If you are really pressed for time, here is the minimalist way to get the most out of this book in the least amount of time: First, read the Preface. Second, read the Key Points at the end of each chapter. You will have seen enough insights to keep you awake through that interminable weekly staff meeting. ▲

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

The False Premise of NoSQL

by Iggy Fernandez

Iggy Fernandez

In the August 2013 issue of the *NoCOUG Journal*, relational theoreticians C.J. Date and Hugh Darwen were asked for their opinions on the NoSQL phenomenon. In my opinion, their comments were right on the mark, even though Date openly admits that he knows almost nothing about NoSQL products.

In discussing Amazon Dynamo—the forerunner of the NoSQL movement—and the products that followed it, Date and Darwen made these astute observations:

"Developers tend to be more concerned with convenience in database definition and updating than with the ease of deriving useful and reliable information from the database."—Darwen

"If there's a suggestion that Amazon's various disaster scenarios, regarding tornados and the rest, are somehow more of a problem for relational systems than they are for nonrelational ones, then of course I reject that suggestion 100 percent."—Date

"Those who disparage relational are almost invariably very far from being properly informed and almost invariably equate 'relational' with [current implementations]."—Darwen

Dynamo Assumptions and Requirements

Date and Darwen's remarks are spot on. Here is the Dynamo use case from the 2007 ACM paper by Amazon: *"Customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers. . . . There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications. . . . Experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability. . . . Dynamo targets applications that operate with weaker consistency (the "C" in ACID) if this results in high availability."*

To paraphrase, Amazon's goals were extreme performance, extreme scalability, and extreme availability, and it concluded that the only way to achieve its ends was to discard the relational model.

Functional Segmentation

Amazon started off on the right track. Its first innovation was to break up the traditional monolithic, enterprise-wide database service into simpler component services such as the best-seller list service, the shopping cart service, the customer preferences service, the sales rank service, and the product catalog service. This avoids a single point of failure.

Amazon's first DBA, Jeremiah Wilton, explained Amazon's approach in his answer to the question "Is 24x7 a myth?" in an interview published in the *NoCOUG Journal* in November 2007. He said "*The best availability in the industry comes from application software that is predicated upon a surprising assumption: The databases upon which the software relies will inevitably fail. The better the software's ability to continue operating in such a situation, the higher the overall service's availability will be. But isn't Oracle unbreakable? At the database level, regardless of the measures taken to improve availability, outages will occur from time to time. An outage may be from a required upgrade or a bug. Knowing this, if you engineer application software to handle this eventuality, then a database outage will have less or no impact on end users. In summary, there are many ways to improve a single database's availability. But the highest availability comes from thoughtful engineering of the entire application architecture.*"

As an example, the shopping cart service should not be affected if the checkout service is having hiccups.

Functional segmentation can result in temporary inconsistencies if, for example, the shopping cart data is not in the same database as the product catalog and occasional inconsistencies result. Occasionally, an item that is present in a shopping cart may go out of stock. Occasionally, an item that is present in a shopping cart may be repriced. The problems can be resolved when the customer decides to check out, if not earlier. As an Amazon customer, I occasionally leave items in my shopping cart but don't complete a purchase. When I resume shopping, I sometimes get a notification that an item in my shopping chart is no longer in stock or has been repriced. This technique is called "eventual consistency." We will return to this subject later in this article and argue that "eventual consistency" is not in conflict with the relational model.

Sharding

Amazon's solution for extreme scalability was "sharding" or horizontal partitioning of all the tables in a hierarchical schema among shared-nothing database servers. The simple hierarchical

schemas that resulted from compartmentation were very shardable.

Here is a simple hierarchical schema from Dr. Codd's 1970 paper that introduced relational theory to the world[1]:

```
employee' (employee#, name, birthdate)
jobhistory' (employee#, jobdate, title)
salaryhistory' (employee#, jobdate, salarydate, salary)
children' (employee#, childname, birthyear)
```

Note that the jobhistory, salaryhistory, and children tables have composite keys. In each case, the leading column of the composite key is the employee#. Therefore, all four tables can be partitioned using the employee#.

There's no conflict with the relational model here either.

> ***"Developers tend to be more concerned with convenience in database definition and updating than with the ease of deriving useful and reliable information from the database. . . . Those who disparage relational are almost invariably very far from being properly informed and almost invariably equate 'relational' with [current implementations]."***

Replication

Amazon then saw that one of the keys to extreme availability was data replication. Multiple copies of the shopping cart are allowed to exist and, if one of the replicas becomes unresponsive, the data can be served by one of the other replicas. The technique used by Dynamo has a close parallel in the well-known technique of "multimaster replication." However, because of network latencies, the copies may occasionally get out of sync and the customer may occasionally encounter a stale version of the shopping cart. Once again, this can be handled appropriately by the application tier; the node that falls behind can catch up eventually or inconsistencies can be detected and resolved at an opportune time, such as at checkout. This is eventual consistency in action.

The inventor of relational theory, Dr. Codd, was acutely aware of the potential overhead of consistency checking. In his 1970 paper, he said:

"There are, of course, several possible ways in which a system can detect inconsistencies and respond to them. In one approach the system checks for possible inconsistency whenever an insertion, deletion, or key update occurs. ***Naturally, such checking will slow these operations down.*** *[emphasis added] If an inconsistency has been generated, details are logged internally, and if it is not remedied within some reasonable time interval, either the user or someone responsible for the security and integrity of the data is notified. Another approach is to conduct consistency checking as a batch operation once a day or less frequently."*

In other words, the inventor of relational theory would not have found a conflict between his relational model and the "eventual consistency" that is one of the hallmarks of the NoSQL products of today. However, Amazon imagined a conflict because it quite understandably conflated the relational model with the ACID guarantees of database management systems. However, ACID has nothing to do with the relational model per se (although relational theory does come in very handy in defining consistency constraints); pre-relational database management systems such as IMS provided ACID guarantees and so did post-relational object-oriented database management systems.

The tradeoff between consistency and performance is as important in the wired world of today as it was in Dr. Codd's world. We cannot cast stones at Dynamo for the infraction of not guaranteeing the synchronization of replicated data or allowing temporary inconsistencies between functional segments, because violations of the consistency requirement are equally commonplace in the relational camp. The replication technique used by Dynamo is known in the relational camp as "multimaster replication." Application developers in the relational camp are warned about the negative impact of integrity constraints.[2,3,4,5] And, most importantly, no DBMS that aspires to the relational moniker currently implements the SQL-92 "CREATE ASSERTION" feature that is necessary to provide the consistency guarantee. For a detailed analysis of this anomaly, refer to Toon Koppelaars's article "CREATE ASSERTION: The Impossible Dream?" in the August 2013 issue of the *NoCOUG Journal*.

The False Premise of NoSQL

The final hurdle was extreme performance, and that's where Amazon went astray. Amazon believed that the relational model makes data retrieval and updates inefficient and therefore chose to store data as BLOBs. Amazon's objection to the relational model is colorfully summarized by the following statement attributed to Esther Dyson, the editor of the Release 1.0 newsletter, *"Using tables to store objects is like driving your car home and then disassembling it to put it in the garage. It can be assembled again in the morning, but one eventually asks whether this is the*

---

[1] "A Relational Model of Data for Large Shared Data Banks." Reprinted with permission in the 100th issue of the *NoCOUG Journal*. (**http://www.nocoug.org/Journal/NoCOUG_Journal_201111.pdf**)

[2] "Using primary and foreign keys can impact performance. Avoid using them when possible." (**http://docs.oracle.com/cd/E17904_01/core.1111/e10108/adapters.htm#BABCCCIH**)

[3] "For performance reasons, the Oracle BPEL Process Manager, Oracle Mediator, human workflow, Oracle B2B, SOA Infrastructure, and Oracle BPM Suite schemas have no foreign key constraints to enforce integrity." (**http://docs.oracle.com/cd/E23943_01/admin.1111/e10226/soaadmin_partition.htm#CJHCJIJI**)

[4] "For database independence, applications typically do not store the primary key-foreign key relationships in the database itself; rather, the relationships are enforced in the application." (**http://docs.oracle.com/cd/E25178_01/fusionapps.1111/e14496/securing.htm#CHDDGFHH**)

[5] "The ETL process commonly verifies that certain constraints are true. For example, it can validate all of the foreign keys in the data coming into the fact table. This means that you can trust it to provide clean data, instead of implementing constraints in the data warehouse." (**http://docs.oracle.com/cd/E24693_01/server.11203/e16579/constra.htm#i1006300**)

"In one approach the system checks for possible inconsistency whenever an insertion, deletion, or key update occurs. Naturally, such checking will slow these operations down. If an inconsistency has been generated, details are logged internally, and if it is not remedied within some reasonable time interval, either the user or someone responsible for the security and integrity of the data is notified. Another approach is to conduct consistency checking as a batch operation once a day or less frequently."

most efficient way to park a car." The statement dates back to 1988 and was much quoted when object-oriented databases were in vogue.[6]

Since the shopping cart is an object, doesn't disassembling it for storage make data retrieval and updates inefficient? The belief stems from an unfounded assumption that has found its way into every relational implementation to date—that every table should map to physical storage. In reality, the relational model is a logical model and, therefore, it does not concern itself with storage details at all. It would be perfectly legitimate to store the shopping cart in a physical form that resembled a shopping cart while still offering a relational model of the data complete with SQL. In other words, the physical representation could be optimized for the most important use case—retrieving the entire shopping-cart object using its key—without affecting the relational model of the data. It would also be perfectly legitimate to provide a nonrelational API for the important use cases. Dr. Codd himself gave conditional blessing to such nonrelational APIs in his 1985 *Computerworld* article, "Is Your DBMS Really Relational?," in which he says, *"If a relational system has a low-level (single-record-at-a-time) language, that low level [should not] be used to subvert or bypass the integrity rules and constraints expressed in the higher level relational language (multiple-records-at-a-time)."*

Have I mentioned that Dr. Codd invented relational theory?

Zeroth Normal Form

In fact, the key-blob or "key-value" approach used by Dynamo and the products that followed it is exactly equivalent to "zeroth" normal form in relational terminology.[7] In his 1970 paper, Dr. Codd says: *"Nonatomic values can be discussed within the relational framework. Thus, some domains may have relations as elements. These relations may, in turn, be defined on nonsimple domains, and so on. For example, one of the domains on which the relation employee is defined might be salary history. An element of the salary history domain is a binary relation defined on the domain date and the domain salary. The salary history domain is the set of all such binary relations. At any instant of time there are as many instances of the salary history relation in the data bank as there are employees. In contrast, there is only one instance of the employee relation."* In common parlance, a relation with nonsimple domains is said to be in "zeroth" normal form or unnormalized. Dr. Codd suggested that unnormalized relations should be normalized for ease of use. Here is the unnormalized employee relation from Dr. Codd's paper:

```
employee (
  employee#,
  name,
  birthdate,
  jobhistory (jobdate, title, salaryhistory (salarydate, salary)),
  children (childname, birthyear)
)
```

The above unnormalized relation can be decomposed into four normalized relations as follows.

```
employee' (employee#, name, birthdate)
jobhistory' (employee#, jobdate, title)
salaryhistory' (employee#, jobdate, salarydate, salary)
children' (employee#, childname, birthyear)
```

However, this is not to suggest that these normalized relations must necessarily be mapped to individual buckets of physical storage. Dr. Codd differentiated between the stored set, the named set, and the expressible set. In the above example, we have five relations but, if we preferred it, the unnormalized employee relation could be the only member of the stored set. Alternatively, if we preferred it, all five relations could be part of the stored set; that is, we could legitimately store redundant representations of the data. However, the common belief blessed by current practice is that the normalized relations should be the only members of the stored set.

Even if the stored set contains only normalized relations, they need not map to different buckets of physical storage. Oracle is unique among database management systems that aspire to the relational moniker in providing a convenient construct called the "table cluster" that is suitable for hierarchical schemas. In Dr. Codd's example, employee# would be the cluster key, and rows corresponding to the same cluster key from all four tables could be stored in the same physical block on disk. If the cluster was a "hash cluster," no indexes would be required to retrieve records corresponding to the same cluster key from all four tables.

Table Clusters in Oracle Database

Here's a demonstration of using Oracle table clusters to store records from four tables in the same block and retrieving all the

[6] I've been unable to find the statement in the Release 1.0 archives at **http://www.sbw.org/release1.0/** so I don't really know the true source or author of the statement. However, the statement is popularly attributed to Esther Dyson and claimed to have been published in the Release 1.0 newsletter. I found a claim that the statement is found in the September 1988 issue, but that didn't pan out.

[7] Chris Date is a strong proponent of "relation-valued attributes" (RVAs) and argues that relations with RVAs are as "normal" as those without. See "What First Normal Form Really Means" in Date on Database: Writings 2000–2006 (Apress, 2006).

"Using primary and foreign keys can impact performance. Avoid using them when possible. . . . For performance reasons, the Oracle BPEL Process Manager, Oracle Mediator, human workflow, Oracle B2B, SOA Infrastructure, and Oracle BPM Suite schemas have no foreign key constraints to enforce integrity."

components of the "employee cart" without using indexes. First we create four normalized tables and prove that all the records of a single employee including job history, salary history, and children are stored in a single database block so that there is never any join penalty when assembling employee data. Then we create an object-relational view that assembles employee information into a single unnormalized structure and show how to insert into this view using an "INSTEAD OF" trigger.

The following demonstration was performed using Oracle Database 11.2.0.2 running on a prebuilt developer VM for Oracle VM VirtualBox. First, we create a table cluster and add four tables to the cluster.

```
CREATE CLUSTER employees (employee# INTEGER) hashkeys 1000;

CREATE TABLE employees
(
  employee# INTEGER NOT NULL,
  name VARCHAR2(16),
  birth_date DATE,
  CONSTRAINT employees_pk
    PRIMARY KEY (employee#)
)
CLUSTER employees (employee#);

CREATE TABLE job_history
(
  employee# INTEGER NOT NULL,
  job_date DATE NOT NULL,
  title VARCHAR2(16),
  CONSTRAINT job_history_pk
    PRIMARY KEY (employee#, job_date),
  CONSTRAINT job_history_fk1
    FOREIGN KEY (employee#)
    REFERENCES employees
)
CLUSTER employees (employee#);

CREATE TABLE salary_history
(
  employee# INTEGER NOT NULL,
  job_date DATE NOT NULL,
  salary_date DATE NOT NULL,
  salary NUMBER,
  CONSTRAINT salary_history_pk
    PRIMARY KEY (employee#, job_date, salary_date),
  CONSTRAINT salary_history_fk1
    FOREIGN KEY (employee#)
    REFERENCES employees,
  CONSTRAINT salary_history_fk2
    FOREIGN KEY (employee#, job_date)
    REFERENCES job_history
)
CLUSTER employees (employee#);

CREATE TABLE children
(
  employee# INTEGER NOT NULL,
  child_name VARCHAR2(16) NOT NULL,
  birth_date DATE,
  CONSTRAINT children_pk
    PRIMARY KEY (employee#, child_name),
  CONSTRAINT children_fk1
    FOREIGN KEY (employee#) REFERENCES employees
)
CLUSTER employees (employee#);
```

Then we insert data into all four tables. We can use DBMS_ROWID.BLOCKNUMBER to confirm that all the new records have been stored in a single database block, even though they belong to different tables. Therefore the join penalty has been eliminated.

```
INSERT INTO employees
  VALUES (1, 'IGNATIUS', '01-JAN-1970');

INSERT INTO children
  VALUES (1, 'INIGA', '01-JAN-2001');
INSERT INTO children
  VALUES (1, 'INIGO', '01-JAN-2002');

INSERT INTO job_history
  VALUES (1, '01-JAN-1991', 'PROGRAMMER');
INSERT INTO job_history
  VALUES (1, '01-JAN-1992', 'DATABASE ADMIN');

INSERT INTO salary_history
  VALUES (1, '01-JAN-1991', '1-FEB-1991', 1000);
INSERT INTO salary_history
  VALUES (1, '01-JAN-1991', '1-MAR-1991', 1000);
INSERT INTO salary_history
  VALUES (1, '01-JAN-1992', '1-FEB-1992', 2000);
INSERT INTO salary_history
  VALUES (1, '01-JAN-1992', '1-MAR-1992', 2000);

SELECT DISTINCT DBMS_ROWID.ROWID_BLOCK_NUMBER(rowid)
  FROM employees WHERE employee# = 1;
SELECT DISTINCT DBMS_ROWID.ROWID_BLOCK_NUMBER(rowid)
  FROM children WHERE employee# = 1;
SELECT DISTINCT DBMS_ROWID.ROWID_BLOCK_NUMBER(rowid)
  FROM job_history WHERE employee# = 1;
SELECT DISTINCT DBMS_ROWID.ROWID_BLOCK_NUMBER(rowid)
  FROM salary_history WHERE employee# = 1;
```

Next we create an object-relational view that presents each employee as an object.

```
CREATE OR REPLACE TYPE children_rec AS
OBJECT
(
  child_name VARCHAR2(16),
  birth_date DATE
)
/

CREATE OR REPLACE TYPE children_tab AS
TABLE OF children_rec
/

CREATE OR REPLACE TYPE salary_history_rec AS
OBJECT
(
  salary_date DATE,
  salary NUMBER
)
```

```
/

CREATE OR REPLACE TYPE salary_history_tab AS
TABLE OF salary_history_rec
/

CREATE OR REPLACE TYPE job_history_rec AS
OBJECT
(
  job_date DATE,
  title VARCHAR2(16),
  salary_history SALARY_HISTORY_TAB
)
/

CREATE OR REPLACE TYPE job_history_tab AS
TABLE of job_history_rec
/

CREATE OR REPLACE TYPE employee_rec AS
OBJECT
(
  employee# INTEGER,
  name VARCHAR2(16),
  birth_date DATE,
  children CHILDREN_TAB,
  job_history JOB_HISTORY_TAB
)
/

create or replace view employees_view as
SELECT
  employee#,
  name,
  birth_date,
  CAST
  (
    MULTISET
    (
      SELECT
        child_name,
        birth_date
      FROM children
      WHERE employee#=e.employee#
    )
    AS children_tab
  ) children,
  CAST
  (
    MULTISET
    (
      SELECT
        job_date,
        title,
        CAST
        (
          MULTISET
          (
            SELECT salary_date, salary
            FROM salary_history
            WHERE employee#=e.employee#
            AND job_date=jh.job_date
          )
          AS salary_history_tab
        ) salary_history
      FROM job_history jh
      WHERE employee#=e.employee#
    )
    AS job_history_tab
  ) job_history
FROM employees e;
```

Let's retrieve one employee object and look at the query execution plan. No indexes are used in retrieving records from each of the four tables. The cost of the plan is just 1. This is the minimum achievable cost, indicating that there is no join penalty. The results are shown in Figure 1, below.

```
SELECT * FROM employees_view WHERE employee# = 1;
```

Next, let's create an "INSTEAD OF" trigger so that we insert into the view directly; that is, we use a single insert statement instead of multiple insert statements. The trigger will do all the heavy lifting for us.

```
CREATE OR REPLACE TRIGGER employees_view_insert
INSTEAD OF INSERT ON employees_view
REFERENCING NEW AS n
FOR EACH ROW
DECLARE
  i NUMBER;
BEGIN
  INSERT INTO employees
  VALUES
  (
    :n.employee#,
    :n.name,
    :n.birth_date
  );

  FOR i IN :n.children.FIRST .. :n.children.LAST
  LOOP
    INSERT INTO children
```

```
select * from employees_view where employee#=1;

 EMPLOYEE# NAME       BIRTH_DATE CHILDREN                                       JOB HISTORY
---------- ---------- ---------- ---------------------------------------------- ----------------------------------------------
         1 IGNATIUS   01-JAN-70  CHILDREN_TAB(CHILDREN_REC('INIGA', '01-JAN-01' JOB_HISTORY_TAB(JOB_HISTORY_REC('01-JAN-91', '
                                 ), CHILDREN_REC('INIGO', '01-JAN-02'))        PROGRAMMER', SALARY_HISTORY_TAB(SALARY_HISTORY
                                                                               _REC('01-FEB-91', 1000), SALARY_HISTORY_REC('0
                                                                               1-MAR-91', 1000))), JOB_HISTORY_REC('01-JAN-92
                                                                               ', 'DATABASE ADMIN', SALARY_HISTORY_TAB(SALARY
                                                                               _HISTORY_REC('01-FEB-92', 2000), SALARY_HISTOR
                                                                               Y_REC('01-MAR-92', 2000))))

-------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation         | Name           | Starts | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                |      1 |        |       |     1 (100)|          |      1 |00:00:00.01 |       1 |
|*  1 |  TABLE ACCESS HASH| CHILDREN       |      1 |      2 |    34 |     1   (0)| 00:00:01 |      2 |00:00:00.01 |       1 |
|*  2 |  TABLE ACCESS HASH| SALARY_HISTORY |      2 |      2 |    44 |     1   (0)| 00:00:01 |      4 |00:00:00.01 |       3 |
|*  3 |  TABLE ACCESS HASH| JOB_HISTORY    |      1 |      2 |    48 |     1   (0)| 00:00:01 |      2 |00:00:00.01 |       1 |
|*  4 |  TABLE ACCESS HASH| EMPLOYEES      |      1 |      1 |    20 |     1   (0)| 00:00:01 |      1 |00:00:00.01 |       1 |
-------------------------------------------------------------------------------------------------------------------------------
```

Figure 1

```
    VALUES
    (
      :n.employee#,
      :n.children(i).child_name,
      :n.children(i).birth_date
    );
  END LOOP;

  FOR i IN :n.job_history.FIRST .. :n.job_history.LAST
  LOOP
    INSERT INTO job_history VALUES
    (
      :n.employee#,
      :n.job_history(i).job_date,
      :n.job_history(i).title
    );
    FOR j IN :n.job_history(i).salary_history.FIRST .. :n.job_history(i).salary_
history.LAST
    LOOP
      INSERT INTO salary_history
      VALUES
      (
        :n.employee#,
        :n.job_history(i).job_date,
        :n.job_history(i).salary_history(j).salary_date,
        :n.job_history(i).salary_history(j).salary
       );
    END LOOP;
  END LOOP;
END;
/
```

Finally, let's insert an employee object directly into the view.

```
INSERT INTO employees_view
VALUES
(
  2,
  'YGNACIO',
  '01-JAN-70',
  CHILDREN_TAB
  (
    CHILDREN_REC('INIGA', '01-JAN-01'),
    CHILDREN_REC('INIGO', '01-JAN-02')
  ),
  JOB_HISTORY_TAB
  (
    JOB_HISTORY_REC
    (
      '01-JAN-91',
      'PROGRAMMER',
      SALARY_HISTORY_TAB
      (
        SALARY_HISTORY_REC('01-FEB-91', 1000),
        SALARY_HISTORY_REC('01-MAR-91', 1000)
      )
    ),
    JOB_HISTORY_REC
    (
      '01-JAN-92',
      'DATABASE ADMIN',
      SALARY_HISTORY_TAB
      (
        SALARY_HISTORY_REC('01-FEB-92', 2000),
        SALARY_HISTORY_REC('01-MAR-92', 2000)
      )
    )
  )
);
```

Amazon vs. eBay

The eBay e-commerce platform is as large as that of Amazon, and eBay had the same goals as Amazon: extreme performance, extreme scalability, and extreme availability. To achieve these goals, eBay also used functional segmentation, sharding, and replication. However, eBay did not see the need to abandon the relational model in the local instances and continued to use Oracle and SQL in these instances. This proves that it is possible to build a modern e-commerce platform without abandoning the relational model.

Conclusion

NoSQL is based on the false premise that the relational model creates a join penalty. Three of the four pillars of the NoSQL approach—functional segmentation, sharding, and replication—are compatible with the relational model. Amazon had an opportunity to take the relational model to the next level but did not rise to the occasion. Amazon could have eaten its cake (extreme performance, extreme scalability, and extreme availability for important use cases such as shopping carts) and had it too (the relational model with all its wonderful declarative power). As Hugh Darwen said in his *NoCOUG Journal* interview: *"Developers tend to be more concerned with convenience in database definition and updating than with the ease of deriving useful and reliable information from the database. . . . Those who disparage relational are almost invariably very far from being properly informed and almost invariably equate 'relational' with [current implementations]."* I'll leave you with that thought. ▲

The statements and opinions expressed here are the author's and do not necessarily represent those of Oracle Corporation.

CREATE ASSERTION: Neither Impossible nor a Dream

Erwin Smout

by Erwin Smout

It must be that enforcement of business rules is hot again. Toon Koppelaars had an article on CREATE ASSERTION in the August 2013 edition of the *NoCOUG Journal*, asking if it is an "impossible dream." Iggy Fernandez mentioned the subject in the May 2013 edition. And I see the topic pop up every now and then in numerous other places and discussion forums.

The general narrative is usually along the lines of "no vendor offers support for CREATE ASSERTION [or even cross-row CHECK constraints, for that matter], so the best we can do is write our own triggers." Now sit tight. Bold claim ahead: This is simply not true. We can do better, because the "problem" of supporting assertions has been solved . . . and implemented. You can try it out within the hour if you want. But first, perhaps, you'll finish reading this article.

Before substantiating my claim, I'll comment a bit more on constraint-related matters, especially about the problems of constraint enforcement that programmers still face, even when they're using triggers and, possibly, generator tools.

First, as far as I know, computing which tables to add triggers to is still left to the user. Ceri/Widom already demonstrated this information to be automatically computable in their award-winning 1990 paper.[1] And it's not a trivial computation to make. Journal articles typically limit themselves to simple examples, and for these, the computation is indeed easy enough for humans to do it "on sight." But I invite anyone familiar with Toon Koppelaars' famous *Applied Mathematics for Database Professionals* (AM4DP) book[2] to do the exercise for the pièce de résistance case: You are allowed to teach a certain course only if: (1) you have been employed for at least one year, or (2) you have attended that course first and the trainer of that course offering attends your first teach as participant. Try to figure out all the triggers you'd need to define in order to enforce this rule correctly and completely. Hint: If your result doesn't include DELETE from Course Session Subscriptions plus DELETE from Course Sessions, then you still have it wrong (and there are yet a few more that aren't so intuitively clear).

Second, specifying which queries to fire for each individual trigger identified in the foregoing step is even more of a nightmare. Getting it completely right will be outright impossible for most normal humans. Once again, try getting it right from the get-go for the given example of training sessions, for the case of DELETE from course subscriptions. Now you might argue, isn't the analysis work involved here exactly the same as when the business rule is to be enforced through application logic? Well, you're right: indeed it is. But what does that imply exactly? That I'm wrong in claiming that it's impossible to do right, because in practice we are already doing it perfectly, or that I'm still right in pointing out the difficulties, and that you only need to observe practice to see that this is, indeed, one area in which we are failing, and rather abysmally at that? Have you never yet been investigating how "inconsistent" data managed to make its way to the database, and when you found the reason, slapped yourself, saying something like, "Oops, how did we come to overlook that possibility in our analysis"? I'm rather inclined to rest my case and stand by my claim that the task is so complex, it is impossible to get it perfectly right.

Third, there are additional dimensions to the problem that (a) are mentioned only cursorily or, worse, not at all, even in highly respectable treatments such as Toon Koppelaars' AM4DP book, and (b) further complicate the foregoing problems by orders of magnitude. One such dimension is deferred constraint checking. You probably know when that is needed/inevitable: whenever a rule is such that satisfying it inevitably requires simultaneous updates to different tables or requires simultaneous updates of different types (insert/delete) to the same table, for the same reason. With deferred constraint checking, the constraints are checked at commit time. Achieving the same effect with triggers would require something like before-commit triggers, and even if those were supported, what you would then be doing is essentially (a) update tables; (b) check everything; (c) if anything is not okay, undo updates to tables. However, the preferred approach for updating is (a) check everything *before* doing the updates, and proceed with the updates *only* if everything is okay.

The first point makes the job of achieving complete business rules checking using triggers real tough; the second point makes it nigh impossible; and the third point makes it unclear whether that ultimate goal is even achievable at all (using triggers, and with the desideratum of "as efficiently as possible" also implying "no shoddy trial-and-error approaches").

[1] Stefano Ceri and Jennifer Widom, 1990. "Deriving Production Rules for Constraint Maintenance," VLDB proc., pp. 566–577.

[2] Lex de Haan and Toon Koppelaars, *Applied Mathematics for Database Professionals*, Apress, ISBN 978-1-59059-745-3.

Showtime

But as I claimed: the good news is that the problem has been solved. The material I'll be using to substantiate this claim is based on my DBMS project, SIRA_PRISE[3] (but please also see the final footnote). You can find more information on the website; for the purpose of this article I'll briefly list the most important characteristics here:

- It's a relational DBMS, based on the ideas from *The Third Manifesto*.[4]
- As such, it is a "No to SQL" system, and it doesn't speak or understand SQL.
- It supports an extensive set of operators of the relational algebra, including transitive closure, summarizeby, divideby, and group/ungroup. In other words, in terms of relational data manipulation, it's aimed toward the ability to answer any query that an SQL system can answer.
- It supports multiple assignment, i.e., one DML statement can update multiple distinct tables/relvars (or do multiple distinct types of update operations on the same table/relvar).
- Database constraints that constitute a key on a virtual relvar (*Third Manifesto*-speak for "view") can be defined as such (i.e., it supports the equivalent of a hypothetical "CREATE VIEW . . . KEY . . ." in SQL).
- It offers support for database integrity using declarative database constraints. This essentially means that it supports CREATE ASSERTION.

The main page on the SIRA_PRISE website has a comparison between what the enforcement of a given constraint looks like in SQL and what it looks like in SIRA_PRISE, and this can be considered equivalent to what it would look like in an SQL system that supports ASSERTIONs. The constraint I'll use to illustrate my point is inspired by the example in the AM4DP book and implements the business rule, "Employees cannot earn more than their manager." This is a single-table constraint on the EMP table from the AM4DP example. The assumed table structure is as in the book, modulo things such as attribute naming that I've altered to match one preferred naming style of mine. One last disclaimer: These code snippets have never been intended for anything more than illustration of concepts and ideas, so this code has deliberately never been proofed on a real operational system.

Let's start with dissecting the "SQL triggers" approach. First, there is some facility, housekeeping stuff that must be put in order. In particular, we need this thing called a "transition effect table":

```
create global temporary table EMP_TE
(
  DML char(1) not null check (DML in ('I' ,'U', 'D')),
  ROW_ID rowid,
  nr_emp ...,
  nm_job ...,
  dt_hired ...,
  cd_sgrade ...,
  am_sal ...,
  nr_dept ...,
  nr_emp_mgr ...,
  check(DML<>'I' or row_id is not null),
  check(DML<>'U' or row_id is not null),
  check(DML<>'D' or row_id is null)
)
on commit delete rows;
```

Points arising:

- The structure of this table must be kept in sync with the structure of the EMP table itself; that is, any structural changes applied later to the EMP table must also be applied to this TE table.
- If an existing row is affected (Delete or Update), it appears in this table in the form of its column values. If a new row is involved (Insert or Update), it appears in this table in the form of the rowid pointing to the row itself in the "base" table (EMP).
- All columns in this table except DML must be nullable (see the trigger below for why).
- The check constraints are not strictly necessary (as long as the only updaters to this table are the triggers below), but they could also be extended to enforce that nullability of the nr_emp, . . . columns is exactly as in the EMP table itself, in the cases where DML IN ('U','D').

Next, there's some housekeeping to do in order to get the proper content into this TE table:

```
create trigger EMP_BIUD_TE
before insert or update or delete on EMP
begin
  -- reset before every DML
  delete from EMP_TE;
end;

create trigger EMP_AIUDR_TE
after insert or delete or update on EMP for each row
begin
  -- register statement tuple
  if INSERTING then
    -- only store 'pointer' to inserted row
    insert into EMP_TE (DML,ROW_ID) values ('I', :new.rowid);
  elseif UPDATING then
    -- snapshot of old row plus pointer to replacing row
    insert into EMP_TE
    (
      DML,
      row_id,
      nr_emp,
      nm_job,
      dt_hired,
      cd_sgrade,
      am_sal,
      nr_dept,
      nr_emp_mgr
    )
    values
    (
      'U',
      :new.rowid,
      :old.nr_emp,
      ...,
      :old.nr_dept,
      :old.nr_emp_mgr
    );
  elseif DELETING
```

[3] SIRA_PRISE, **http://shark.armchair.mb.ca/~erwin**.

[4] Chris Date and Hugh Darwen, *Databases, Types and the Relational Model*, Addison-Wesley, ISBN 0-321-39942-0, **http://www.thethirdmanifesto.com**.

```
  --snapshot of old row
  insert into EMP_TE
  (
    DML,
    row_id,
    nr_emp,
    nm_job,
    dt_hired,
    cd_sgrade,
    am_sal,
    nr_dept,
    nr_emp_mgr
  )
  values
  (
    'D',
    null,
    :old.nr_emp,
    ...,
    :old.nr_dept,
    :old.nr_emp_mgr
  );
 end if;
 --
end;
```

The first trigger empties our TE table before any update; the second copies over values from the EMP table to the TE table.

Points arising:

- The updates are already tentatively being applied to the EMP table, *after* which we are going to set up our TE table to verify whether the update is acceptable or not (and if it's not, to undo the already applied update).

Next, for ease of reference, we define some views that allow us to distinguish the current inserts, deletes, and updates from one another.

```
create view V_EMP_ITE as
select e.*
from EMP_TE te, EMP e
where DML='I' and te.row_id = e.rowid;

create view V_EMP_UTE as
select
  e.nr_emp as n_nr_emp,
  e.nm_job as n_nm_job,
  ...,
  e.nr_dept as n_nr_dept,
  e.nr_emp_mgr as n_nr_emp_mgr,
  te.nr_emp as o_nr_emp,
  te.nm_job as o_nm_job,
  ...,
  te.nr_dept as o_nr_dept,
  te.nr_emp_mgr as o_nr_emp_mgr
from emp_te te, emp e
where DML = 'U' and te.ROW_ID = e.rowid;

create view V_EMP_DTE as
select
  nr_emp,
  nm_job,
  ...,
  nr_dept,
  nr_emp_mgr
from emp_te
where DML='D';
```

Points arising:

- These three views are the constructs that are actually interesting and that we will be using in the piece of code to enforce our business rule.
- Note that these three views make for three additional places where a table is defined whose heading is supposed to be kept perfectly in sync with that of our EMP table. We definitely don't want to be forced into doing this by hand, and I personally don't even want to be forced into putting myself in front of a generator tool to push that "sync" button manually. This sort of stuff must be internalized into the DBMS and made available automatically.

At this point I'd like to skip over to SIRA_PRISE for a moment. "Internalizing that sort of stuff into the DBMS" is exactly what has been done in that project. Each SIRA_PRISE relvar ("table") is automatically associated with two constructs that can be invoked as INSERTS(<relvarname>) and DELETES(<relvarname>). These constructs can legitimately be invoked in any expression in which the context of execution is an update DML statement, and they yield exactly the relations ("tables") from that ITE and DTE view. Points arising:

- You are now, of course, bound to ask, "What about the updates view UTE?" I'll come to that later.
- As far as the enforcement of *database* constraints is concerned, these constructs must actually not even be known by the user for the system to work properly. The reasons for exposing these two constructs to the SIRA_PRISE user has nothing to do with database constraints.

This latter point is evidenced by the fact that there is no trace of INSERTS() and/or DELETES() invocations in the equivalent SIRA_PRISE code that I will show you shortly.

Back to SQL and triggers. We have now arrived at the point where we can finally write down the code that will enforce our business rule:

```
create trigger EMP_AIS_R47 after insert on EMP
declare pl_dummy varchar(40);
begin
  -- Inserting an employee. Check salary < manager's salary
  for r in (select distinct nr_emp, am_sal, nr_emp_mgr from V_EMP_ITE i)
  loop
    begin
      -- acquire serialization lock
      p_request_lock('R47'||to_char(r.nr_emp));
      --
      select 'Constraint R47 is satisfied' into pl_dummy
      from DUAL
      where exists (
        select 'manager with higher salary'
        from EMP e
        where e.nr_emp = r.nr_emp_mgr and e.am_sal <= r.am_sal
      );
      --
      exception when no_data_found then
      raise_application_error (
        -20999,
        'Constraint R47 is violated for employee'||to_char(r.nr_emp)
      );
      --
    end;
  end loop;
end;
```

Points arising:

- The issue with the serialization lock is explained in depth in AM4DP. It is for the most part Oracle-specific, and if it could be eliminated, then we could perhaps also replace that explicit FOR loop by a simple natural join. This is

something that Codd would definitely have preferred over this particular version.

- Anyway, for each row in our ITE view (rows that have been inserted into EMP), we execute a query to check a certain condition.
- But exactly which query is to be executed here is for the database designer to craft by hand. I've claimed before that this can be even more of a nightmare. Another observation I made, that typical journal articles limit themselves to very simple examples—and this typically makes things seem relatively simple—also applies. What needs to be done here is to bring two rows together (an employee and his manager) and do a comparison on them. How difficult can that be? Well, you can try it out. I've deliberately put a mistake in the foregoing code. Can you spot it? And even if the answer to that is "yes," I'm confident that you'll still appreciate the difficulty of crafting this sort of query by hand if the business rule involves 7 or 12 or whatever number of tables, all tied together with any number of JOINs, EXCEPTs, UNIONs, and the like.
- Also far from obvious: there is also presumably a foreign key from nr_emp_mgr to nr_emp. What will happen in this trigger if that FK is violated? Does the order of execution of the triggers affect the nature of the error message we get in case of violations? This is yet another thorny issue, and from practical experience, I can tell that it's relatively easy to write these constraint-enforcing queries (or the constraints themselves) in such a way that they sometimes cause extremely unintuitive failure messages.
- We are only doing this check for the inserts into EMP. But clearly, updates could violate our business rule as well. We need to define a second similar trigger for the update case:

```
create trigger EMP_AUS_R47
after update on EMP
declare pl_dummy varchar(80)
begin
  -- changing a manager or salary requires checking against new manager's
salary
  for r in (
    select
      n_nr_emp as nr_emp,
      n_am_sal as am_sal,
      n_nr_emp_mgr as nr_emp_mgr
    from V_EMP_UTE e
    where e.o_nr_emp=e.n_nr_emp
    and (e.n_am_sal > e.o_am_sal || e.n_nr_emp_mgr <> e.o_nr_emp_mgr)
  )
  loop
    begin
      -- acquire serialization lock
      p_request_lock('R47' || to_char(r.nr_emp));
      select 'Constraint R47 is satisfied' into pl_dummy
      from DUAL
      where exists (
        select 'manager with higher salary'
        from EMP e
        where e.nr_emp = r.nr_emp_mgr and e.am_sal >= r.am_sal );
      --
      exception when no_data_found then
      --
      raise_application_error(-20999, 'Constraint R47 is violated for employee'
|| to_char(r.nr_emp));
      --
    end;
  end loop;
--
end;
```

Points arising:

- With this one, there is considerable danger of making certain logical mistakes as a consequence of (let's not kid ourselves) the ever-pressing desire to "optimize as much as possible." For example, we could perfectly add that predicate "WHERE N_AM_SAL > O_AM_SAL" in that SELECT that drives the for loop, can't we? We don't need to actually do this query if someone's salary has decreased, right? If someone's salary has decreased, then it certainly cannot have gone above that of his manager, right? Wrong! It still can if at the same time the poor employee-with-decreased-salary also gets another manager!
- So we extend the predicate (that's what the example given here shows) to also do the check for employees who changed managers. Is the trigger now correct? Will it now correctly detect all possible violations of the business rule? Hint: What if a manager's salary is decreased below that of one of his employees? Where is that detected?
- Conclusion: Our update trigger as given here still needs further extending to cater to the case of a manager's salary being decreased. So once again I think this shows how even an extremely simple business rule leads to a significant number of intricate possibilities that are all too easily overlooked (ultimately leading to the business rule not actually being enforced in all circumstances). We don't want this stuff left in the hands of humans. It's just too complicated for us, and all the more so if we are operating under time pressure. We need this stuff computed by the machine.

And that's where SIRA_PRISE steps in. Enforcing the very same business rule "Employees cannot earn more than their manager" is just a matter of declaring the following constraint:

```
assert databaseconstraint, databaseconstraint (
  tuple (
    constraintlabel(R47 - Employees cannot earn more than the employee
they report to.)
    errorcode(115147)
    sp_expression (
      restrict (
        join (
          emp,
          project (
            rename (
              emp,
              (nr_emp_mgr, mgrmgr, nr_emp, nr_emp_mgr, am_sal_mthly,
am_sal_mthly_mgr)
            ),
            (nr_emp_mgr, am_sal_mthly_mgr)
          )
        ),
        gt(am_sal_mthly, am_sal_mthly_mgr)
      )
    )
  )
)
```

This is probably more like gibberish than like DDL to the seasoned SQL/Oracle guy, so let's dissect this stuff a little bit.

- The assert DATABASECONSTRAINT portion means that we are declaring a database constraint. "DATABASECONSTRAINT" is just the name of a catalog relvar, and "assert" means that we are about to INSERT into it, roughly speaking.

- The databaseconstraint(...) construct denotes essentially a relation ("table") value that matches the type of the concerned database relvar ("table")—something roughly similar to VALUES(...) in SQL.
- The tuple(...) construct corresponds roughly to an SQL ROW(...) construct.
- The foregoing is thus roughly equivalent to INSERT INTO DATABASECONSTRAINT (constraintlabel, errorcode, sp_expression) VALUES(...).
- The constraintlabel (a "column" in the databaseconstraint "table") is recorded in the catalog for purposes of documentation.
- The errorcode is the identifying key for any constraint in the system.
- The sp_expression is a column of type String (the equivalent of SQL VARCHAR) holding the defining expression for the constraint. In reality, character escaping using backslashes has to be applied to this command, but that escaping has been left out here for reasons of readability.

It's this latter expression that defines the nature of the constraint, so let's take a closer look at that expression itself, from the outside in.

- The outermost portion 'RESTRICT(... , GT(...))' is a RESTRICT on some join, with the restrict condition: am_sal_mthly > am_sal_mthly_mgr.
- That restrict is applied to the natural join denoted by JOIN (EMP , ...), which serves to join each employee to his manager, so to speak.
- The second argument is a projection of a rename (equivalent to SELECT NR_EMP AS NR_EMP_MGR, AM_SAL_MTHLY AS AM_SAL_MTHLY_MGR) of EMP.

Hence this entire expression is the equivalent of

```
SELECT * FROM (
  SELECT * FROM EMP
  NATURAL JOIN
  SELECT
    NR_EMP AS NR_EMP_MGR,
    AM_SAL_MTHLY AS AM_SAL_MTHLY_MGR
  FROM EMP
)
WHERE AM_SAL_MTHLY > AM_SAL_MTHLY_MGR;
```

This is the query that would identify all the employees who earn more than their manager or, in other words, all the employees who are in violation of the constraint! SIRA_PRISE calls this the "faults expression" for that reason. It's also the query that you would have to provide if you had "CREATE ASSERTION employee_paid_too_much CHECK NOT EXISTS (...)". And the query you would be using to enforce the constraint through the materialized view trick is, of course, also very closely related to this one.

In addition to being checked for general validity as a database query, the following three extra steps are also undertaken by SIRA_PRISE as a consequence of this expression being declared as defining a database constraint:

- All involved relvars ("tables") are identified (just EMP in this case).
- For all involved relvars, it is computed which update operation types could possibly affect the outcome of the database constraint (just INSERT in this case; UPDATE is always regarded as being both an INSERT and a DELETE, so finding "sensitivity" to INSERT covers UPDATE as well).
- For the constraint overall, an expression is computed that determines, for any given database update, whether it will make the faults expression nonempty. If it does, then that particular database update is unacceptable because of the implied constraint violation. These expressions tend to get horrendously lengthy when spelled out, but the good thing is that no one ever has to actually take a closer look at them! This is, of course, the very heart of the constraint enforcement engine, and it has the following desirable properties:
- For each database update, there is *exactly one check per potentially affected constraint* to see if there is a violation of that constraint or not. (Contrast this with the triggers-based approach in which there are potentially lots of queries, one per individual row.)
- For each database update, compliance to all the declared constraints can effectively be determined prior to actually writing out the updates to the database (contrast this with applying the updates and then using the triggers to see if there is now a violation).

If you are looking for a downside, it has to do with that so-called "optimization" in the update triggers. I know currently of no way to decide *reliably* that optimizations such as "if the salary has decreased then there's no need to check" can safely be applied. So they aren't. But as I've shown, it's easy enough to apply such optimizations prematurely, and wrongly. They're dangerous. And the upside to counter it is that the machinery will not miss possibilities such as "What if the manager's salary has decreased" because there's a simple mechanical computation underneath.

As a second example, let's actually set up (but please also see the final footnote) and explore a simple case of "complex constraint" enforcement (this time without looking at the SQL version). You all know bill-of-material structures. Parents have children, children can themselves be parents of their own children, etc., etc. And no person in the structure can ever be among his own ancestry, at whatever level/generation. Or parts can, themselves, be an assembly of other parts, and no part can ever be contained within itself. Two tables are required for this type of problem, one for defining the "existence" of individuals (parts, persons ...) and another one for defining the "containment relationship" between the individuals. The second one typically contains just the two identifiers (parentid and childid) of the "related" individuals, and it's the only one we need as far as the constraint that we're interested in is concerned.

```
# define the business elements

add attribute, attribute (
  tuple (
```

```
    attributename(parentid)
    typename(long)
  )
  tuple (
    attributename(childid)
    typename(long)
  )
)

# define the logical structure

add relvar, relvar (
  tuple (
    relvarname(nocoug)
    relvarpredicate(§parentid§ is the parent of §childid§)
  )
)

add relvarattribute, relvarattribute (
  tuple (
    relvarname(nocoug)
    attributename(parentid)
  )
  tuple (
    relvarname(nocoug)
    attributename(childid)
  )
)

add key, key (
  tuple (
    relvarname(nocoug)
    errorcode(777777)
  )
)

add keyattribute, keyattribute (
  tuple (
    attributename(childid)
    errorcode(777777)
  )
)

# define the storage resources
# the cmd() wrappers have the effect of chaining together all the individual
assignments into a single multi-update.

cmd (
  add dbmsfile, dbmsfile (
    tuple (
      filename(NOCOUG.SPDB)
      pagesize(32768)
    )
  )
)
cmd (
  add storagespace, storagespace (
    tuple (
      filename(NOCOUG.SPDB)
      storagespaceid(1)
      pagecount(32768)
      extentscount(7)
    )
  )
)
cmd (
  add hashedrecordspace, hashedrecordspace (
    tuple (
      filename(NOCOUG.SPDB)
      storagespaceid(1)
      gapcompressionthreshold(4)
      maximumgaps(9)
    )
  )
)

# define the physical design

add recordtype, recordtype (
  tuple (
    relvarname(nocoug)
    recordtypename(R1_nocoug)
    maximumlength(80)
    filename(nocoug.spdb)
    storagespaceid(1)
  )
  tuple (
    relvarname(nocoug)
    recordtypename(R2_nocoug)
    maximumlength(80)
    filename(nocoug.spdb)
    storagespaceid(1)
  )
)

add recordattribute, recordattribute (
  tuple (
    relvarname(nocoug)
    recordtypename(R1_nocoug)
    ordinal(5)
    attributename(childid)
  )
  tuple (
    relvarname(nocoug)
    recordtypename(R1_nocoug)
    ordinal(15)
    attributename(parentid)
  )
  tuple (
    relvarname(nocoug)
    recordtypename(R2_nocoug)
    ordinal(5)
    attributename(childid)
  )
  tuple (
    relvarname(nocoug)
    recordtypename(R2_nocoug)
    ordinal(15)
    attributename(parentid)
  )
)

add indexattribute, indexattribute (
  tuple (
    relvarname(nocoug)
    recordtypename(R1_nocoug)
    ordinal(7)
    attributename(childid)
  )
  tuple (
    relvarname(nocoug)
    recordtypename(R2_nocoug)
    ordinal(7)
    attributename(parentid)
  )
)
```

The faults expression to use in the needed database constraint on the relvar is

```
RESTRICT(TCLOSE(nocoug, (parentid, childid)), EQ(parentid, childid))
```

Once again we'll take a brief look at the anatomy of the formulation of this faults expression:

- The TCLOSE expression produces the (parentid, childid) pairs of individuals such that parentid denotes an individual that is an ancestor of the individual denoted by childid, at any level. (In SQL, achieving this would involve a WITH RECURSIVE ... expression.)
- The RESTRICT on that, then, obviously yields only those pairs that denote an ancestry relationship between an individual and itself, and such a scenario is clearly at fault.

(As already indicated, this is a declaration, an algebraic expression, of the rule. This does NOT imply that this query as such is executed upon each database update. If it did, I'd be really quite stupid to even be submitting this article. The queries that are executed for actually verifying the constraint in the face of a given update are derived from this one, and are aimed at maximal efficiency.)

Our relvar is still empty; now let's try to get some unacceptable data in. The command tries to insert three tuples corresponding to the pairs (1,3), (3,7), and (7,1), which clearly form a cycle, which constitutes a violation of the constraint.

```
add nocoug, nocoug (
  tuple(parentid(1) childid(3))
  tuple(parentid(3) childid(7))
  tuple(parentid(7) childid(1))
)

Constraint violation(777778)
Constraint 777778 violated.  Violating tuple attribute values are :
CHILDID=1, PARENTID=1
```

Okay, so that fails appropriately.

Now let's see if we can manage to get the same invalid data in, in multiple update steps. First we add two of the three foregoing tuples, which should be acceptable:

```
add nocoug, nocoug (
  tuple(parentid(1) childid(3))
  tuple(parentid(3) childid(7))
)
```

That works, as it should.

Now we try to add the third tuple, which would bring the database into a state of constraint violation because of the two tuples that are already in the database:

```
add nocoug, nocoug (
  tuple(parentid(7) childid(1))
)

Constraint violation(777778)
Constraint 777778 violated.  Violating tuple attribute values are :
CHILDID=1, PARENTID=1
```

That fails, as it should.

Let's also illustrate a case of simultaneous delete/insert. The following update simultaneously removes one of the two existing tuples while adding the same one that was formerly rejected. The overall effect of this update will not bring the database into a state of constraint violation, because after all the specified updates are done, the relvar will still hold no cycles, and thus the update should be accepted. Note that, in the language of SIRA_PRISE, the cmd() wrappers have the effect of chaining together all the individual assignments into a single multi-update:

```
cmd (
  add nocoug, nocoug (
    tuple(parentid(7) childid(1))
  )
)
cmd (
  delete nocoug, nocoug (
    tuple(parentid(3) childid(7))
  )
)
```

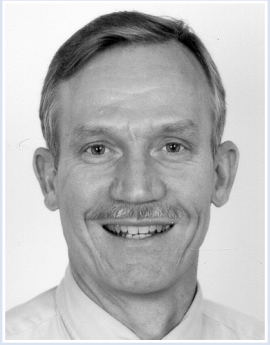
That works, as it should.

There are many more cases and examples to show; for example, constraints on aggregated data or addressing that question you undoubtedly have itching by now ("And how does all this behave in the face of bigger volumes?"), but time and space don't allow me to address all that right here right now. However, the bottom line should be clear: all of this works (efficiently insofar as logically possible) without a single programmer having to write a single byte of procedural code for the constraint/business rule to be enforced. I've presented some examples here for TCLOSE; rest assured that the analysis has been done for the other operators of the relational algebra too, and it works equally well for any of them. CREATE ASSERTION is neither impossible nor a dream.[5] ▲

Erwin Smout is an expert in data management from Antwerp, Belgium and the creator of SIRA_PRISE. More information about SIRA_PRISE can be found on the SIRA_PRISE website **http://shark.armchair.mb.ca/~erwin/**.

[5] For practical reasons, some parts of the syntax of the examples presented here are for version 1.5. This version is still under development, however, and hence these examples, as given, will not all work on the version that is currently publicly available. For trying these examples out for real, some retro-engineering will have to be done to get them running on the current 1.4 version, especially in the area of physical design (storage spaces and record types).

Flashback: A Misleading Name



Chris Lawson

by Chris Lawson

Despite its appealing name, a "flashback" query can run very slowly. On a large production system, a flashback query going back a few hours can easily take ten hours. What—how can that be?

This happens because Oracle must reconstruct an object as it existed at a certain time. This is the same idea as read-consistency. This reconstruction happens one block at a time, going backwards in time, undoing each transaction.[1]

Starting the Undo

There are other issues with a flashback query that make the process run even more slowly. Of course, Oracle does indeed save the undo information—we can certainly find it, and a flashback query really does work. Here's the problem: The structure of undo segments is heavily biased toward quickly *saving transaction information*—not quickly *reversing transactions*.

Before Oracle can reconstruct an object, it has to *identify* what needs to be undone. One would think this is a trivial step, but that's not so. This can be very time consuming—especially when the database has undergone lots of recent transactions.

Transaction Table

In each undo segment header there lies a critical structure known as the *transaction table*. It's not a "table" as we normally think of one. Maybe a "list" would have been a better name. The transaction table identifies the undo information held in that undo segment. For example, any given entry points to where to find the actual undo block.

That sounds excellent, but the entire transaction table only has information for 34 transactions. (Yes, that sounds small to me, too.) Each entry is called a transaction *slot*. As more transactions are housed in a given undo segment, transaction slots, being so few, are very often *overwritten*. The information is not lost, of course, but to find it, several extra steps are required. On a very busy system, it could take thousands of extra reads just to find where to start. (That's why I observed that Oracle seems very biased toward going forward with the undo and not actually applying it.)

Remember—all this effort is before Oracle even starts the "real" work of rebuilding the object of interest to the time desired. Of course, that final step will add even more time. The point is, the delay of determining where to start can be vastly more than the work required to actually do the reconstructing of the object.

[1] Thanks especially to Jonathan Lewis, who has tested this interesting feature in *Oracle Core: Essential Internals for DBAs and Developers*, Apress, 2011.

Troubleshooting

Troubleshooting a flashback query delay is not so easy. On a busy system, I have seen flashback queries require *millions* of extra reads to flashback a small table with only 20,000 transactions that needed to be undone. If you query the active session history for the session of interest, it will show that it is performing sequential reads from an undo tablespace. One could easily be fooled into thinking (as I did) that there must have been a huge number of transactions on the table of interest. We know better now—the reads were actually Oracle synthesizing the undo information in the transaction table and not actually applying it to the object of interest.

Recycling Undo?

When a transaction table slot is reused, what happens to the valuable information that used to be kept in that slot? Well, there's one logical place for it to go—somewhere in undo-land. In fact, Oracle stores the old slot information right at the beginning of the new undo block that used that slot. In this way, the information is linked together. Therefore, when we perform a flashback query, we can discover what the transaction table looked like at some prior state.

Undoing the Undo?

Hey, wait a minute—all this almost sounds like "undoing the undo!" You're right, and Oracle calls it "Transaction Table Rollback." You can also get a summary in the AWR report, in the Instance Activity section:

Statistic	Total	per Second	per Trans
transaction tables consistent read rollbacks	1,869	0.10	0.00
transaction tables consistent reads - undo records applied	9,577,664	531.95	3.91

Measuring Undo of the Undo?

You can also quantify this event in real time, to get a feel for how often this is happening. On a busy system, it is likely to be happening all the time. Let's see how we do this on a busy RAC system. Here is one way to see this occurring in the current connected sessions. This would be helpful to know if someone is doing a flashback query that seems to be running far longer than expected.

In this script, I look for large values of transaction table undo and list the sessions. I also ignore the background processes (that's why I exclude programs like 'oracle'):

```
Col Module Format A22
Col Sid Format 99999
Col Program Format A20
Col Inst Format 9999
Col Trundo Format 9999999

Select
  One.Inst_Id INST,
  One.Sid,
  Substr(Program,1,20) PROG,
  Substr(Module,1,20) Mod,
  Value TRUNDO
From
  Gv$Sesstat One,
  V$Statname Two,
  Gv$Session Three
Where One.Statistic# = Two.Statistic#
And One.Inst_Id = Three.Inst_Id
And One.Sid = Three.Sid
And Name = 'transaction tables consistent reads - undo records applied'
And Program Not Like 'Oracle@%'
And Value > 90000
Order By Value;

INST   SID    PROG                 MOD              TRUNDO
7      1978   xtsora@cisxx01 (TNS  xtsora@risint01  (TNS 157315
4      408    xtsora@cisxx01 (TNS  xtsora@risint01  (TNS 178481
```

We can see above that there were two active sessions that appear to be impacted.

What Can I Do?

The essence of the problem is having to repeatedly reconstruct the contents of the "slots" in the transaction table. If there were fewer reuses of the slots, then there would be less work required. Oracle support has suggested keeping more undo segments online—and therefore more slots available.

This is accomplished by setting the underscore parameter "_rollback_segment_count." The idea is to override the automatic undo process and force more undo segments to stay online. It seems like the number of reused "slots" should go down commensurately with the extra undo segments that are kept online. So, if we keep 4x as many undo segments online, I would expect to see approximately a 4x reduction in transaction table rollbacks. That's the theory, anyway, but I haven't confirmed it yet.

How Does the Story End?

As of this date, we have not yet tried the secret rollback segment parameter. We are wondering about adverse effects and intend to test the parameter with all of our batch jobs.

We can't help wondering why the database thinks that it's a good idea to take undo segments offline in the first place (and what will happen when we block that). Perhaps the caching effect is better when there are fewer undo segments involved?

We haven't been able to get a clear answer to that question. I am eager to see what happens. ▲

Chris Lawson is an Oracle Ace and performance specialist in the San Francisco Bay Area. He is the author of The Art & Science of Oracle Performance Tuning *as well as* Snappy Interviews: 100 Questions to Ask Oracle DBAs. *When he's not solving performance problems, Chris is an avid hiker and geocacher, where he is known as Bassocantor. Chris can be reached at:* **Chris@OracleMagician.com**.

The NoCOUG mascot showing off the raffle prizes at the summer conference. All his little brothers and sisters found good homes with NoCOUG members!

It Takes *Very Little* Effort to Help NoCOUG!

Naren Nagtode

by Naren Nagtode

NoCOUG would not have thrived for more than 25 years without the efforts of volunteers. Many of you have told me that you would like to help NoCOUG but don't have a lot of time. But there are a couple of ways in which you can help NoCOUG that take *very* little effort on your part:

- NoCOUG's *biggest* challenge is reaching more Oracle professionals in the Bay Area, but the solution is literally at the tips of your fingers! The simplest and *most* effective way to help NoCOUG literally requires only a few keystrokes on your part! All you have to do is to forward our email messages to your friends and colleagues. How easy is that? They can attend their first conference for free! Attendance at our conferences would surely *double* if every NoCOUG member forwarded our email messages to at least *one* other person.
- Another simple and effective way for you to help NoCOUG is to update your profile stored in our member management system. Our board members can better choose topics and venues that meet your needs if they knew your job role and the city in which you live and work. It will only take you a few seconds to update your profile. You can do so at **http://nocoug.wildapricot.org**.

Our next conference is on Thursday, November 21, at Network Meeting Center in the TechMart on Great America Parkway in Santa Clara in the South Bay. You're welcome to attend for half a day if you cannot attend for the whole day. Our keynote speaker is none other than Tom Kyte, and he will talk about the best new features in the recently released Oracle Database 12*c*. Another highlight of this conference is the "RAC Attack" hands-on lab sessions. RAC experts from Database Specialists will help you install and configure RAC on your own laptop. Conference sponsor Fusion-io will be hosting a wine-and-cheese reception for the conference speakers the day before the conference at 4:30 p.m. at the same venue. It's an opportunity to network with Oracle experts in a social setting while enjoying the hospitality of Fusion-io. You can RSVP for the wine-and-cheese reception when you RSVP for the conference.

I'll see you at the wine-and-cheese reception on November 20 and the conference on November 21. ▲

The NoCOUG mascot showing off the NoCOUG Journal at the summer conference.

Advice for an Oracle Beginner?

Tom Kyte

by Tom Kyte

Originally published in the February 2011 issue of the NoCOUG Journal.

When I first started out in IT, I had no experience, no training, no background whatsoever. I was a math major fresh out of college in the year 1987. I hadn't taken any computer courses beyond the initial "introduction to" type of classes. It wasn't until I got a job as a computer programmer—advertised as "no experience required"—that I started even really using computers.

So, given that I had no experience, no real formal training—how did I get started, how did I get to where I am today? I think it comes down to two simple words: mentorship and participation.

> ***"Participation in the Oracle community is what took me from being just another programmer to being 'AskTom'. Without the act of participating, I do not think I would be where I am today."***

When I first started out as an entry-level programmer, I had an excellent mentor. This was probably the key difference between success and failure for me. My mentor—who back then was about the age I am now (that is, he was old)—took the time to teach me the ropes. He taught me the right way to do things—not the fast way, not the "shortcuts," not the checklist of things to do—but the right way. In many cases, the right way isn't the easy way, isn't the quickest way . . . but it is ultimately the best way. He taught me many things I myself teach these days. Simple things such as "make your subroutine fit on a screen, you have to see it all," "instrument your code to death," "write as little code as you can but as much as you have to," "code defensively; don't trust anyone else to just know what to do with your code," and "test, test, test, benchmark and test again." My mentor made me the programmer I am.

The second item—participation—is what propelled me beyond being just a programmer. In the early 1990s, I started participating in online forums hosted on Usenet. For those who never heard of it, "Usenet" was Twitter, blogging, Facebook—any social network goes here—before any of them were invented. Usenet consisted of a relatively small (by today's standards) group of individuals that would discuss topics of interest to them. My interest was, of course, all things Oracle, and discuss we did. I "met" in a virtual sense on those discussion forums many people I still correspond with and interact with face to face. I learned a lot from them—and they (hopefully!) learned a thing or two from me. It was on these forums that I found the answers to many of my questions—and formulated answers to questions from others. This give-and-take allowed my knowledge of Oracle, programming, and databases in general to expand and grow immeasurably. Participation in the Oracle community is what took me from being just another programmer to being "AskTom." It gave me the confidence to write my first book in the year 2000: *Expert One on One Oracle*. It also gave me the audience for such a book. Without the act of participating, I do not think I would be where I am today.

So, in short, find a mentor. This is crucial. Find someone that you trust, that you respect, that others trust and respect. Learn from your mentor. Then, start participating. Participate in your local user group. Get up in front of an audience and present on some technical topic. Attend conferences. Get active in a discussion forum, such as those on **http://otn.oracle.com**. Don't be afraid to make mistakes (you will; I did), but make sure to learn from them. That would be my advice. ▲

---

*Tom Kyte is a Senior Technical Architect in Oracle's Server Technology Division. Before starting at Oracle, Tom worked as a systems integrator building large-scale heterogeneous databases and applications, mostly for military and government customers. Tom spends a great deal of time working with the Oracle database and, more specifically, working with people who are working with the Oracle database. In addition, he is the Tom behind the "AskTom" column in Oracle Magazine, answering people's questions about the Oracle database and its tools (***http://asktom.oracle.com***). Tom is also the author of* Expert Oracle Database Architecture: Oracle Database 9*i*, 10*g*, and 11*g* Programming Techniques and Solutions *(Apress, 2010), co-author of* Beginning Oracle Programing *(Wrox Press, 2002), and author of* Effective Oracle by Design *(Oracle Press, 2003). These are books about the general use of the database and how to develop successful Oracle applications.*

Database Specialists: DBA Pro Service

DBA PRO BENEFITS

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

CUSTOMIZABLE SERVICE PLANS FOR ORACLE SYSTEMS

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

DBA Pro's mix and match service components

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

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

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

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

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

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

DatabaseSpecialists

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

NoCOUG
P.O. Box 3282
Danville, CA 94526

RETURN SERVICE REQUESTED

NoCOUG Fall Conference Schedule

Thursday, November 21, 2013—Network Meeting Center, Santa Clara, CA

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

8:00–9:00 a.m.	Registration and Continental Breakfast—Refreshments served
9:00–9:30	**Welcome:** Naren Nagtode, NoCOUG president
9:30–10:30	**Keynote:** *Introducing Oracle Database 12c*—Tom Kyte, Oracle Corporation
10:30–11:00	**Break**
11:00–12:00	**Parallel Sessions #1**
	Santa Clara: *Advanced Analytic Functions*—Tom Kyte, Oracle Corporation
	Cupertino: *Ten MORE Surprising Performance Tactics*—Chris Lawson, PG&E
	Los Altos: *Oracle NoSQL Database Application Development*—Robert Greene, Oracle Corporation
12:00–1:00 p.m.	**Lunch**
1:00–2:00	**Parallel Sessions #2**
	Santa Clara: *Empowering Your Data-Sharing Architecture for Continuous Availability* —Susan Wong, Dell Software Group
	Cupertino: *Live Oracle Active DataGuard Failover Under Extreme Workload*—Ganesh Balabharathi, Fusion-io
	Los Altos: *Improving MySQL Performance with Hadoop*—Sastry Vedantam, Oracle Corporation
2:00–2:30	**Break and Refreshments**
2:30–3:30	**Parallel Sessions #3**
	Santa Clara: *Using Oracle Execution Plans for Performance Gains*—Janis Griffin, Confio
	Cupertino: *Architecting Multi-Tenancy Analytics in the Cloud Using OBIEE and Oracle DB* —Mayank Srivastava and Hanan Hit, Xtime
	Los Altos: *RAC Attack Part I*—Ian Jones and Terry Sutton, Database Specialists
3:30–4:00	**Raffle**
4:00–5:00	**Parallel Sessions #4**
	Santa Clara: *Oracle Database Scalability—Some Perspectives*—Sai Devabhaktuni and John Kanagaraj, PayPal
	Cupertino: *Scaling ETL with Hadoop*—Gwen Shapira, Cloudera **Journal Editor's Pick**
	Los Altos: *RAC Attack Part II*—Ian Jones and Terry Sutton, Database Specialists
5:00–	**NoCOUG Networking and No-Host Happy Hour at Evolution Cafe & Bar, Hyatt Regency Santa Clara**

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

The *NoCOUG Journal* design and production: Giraffex, Inc., S.F.
Front cover photo: Close-up of a baby boy playing with an abacus by George Doyle, Photos.com.