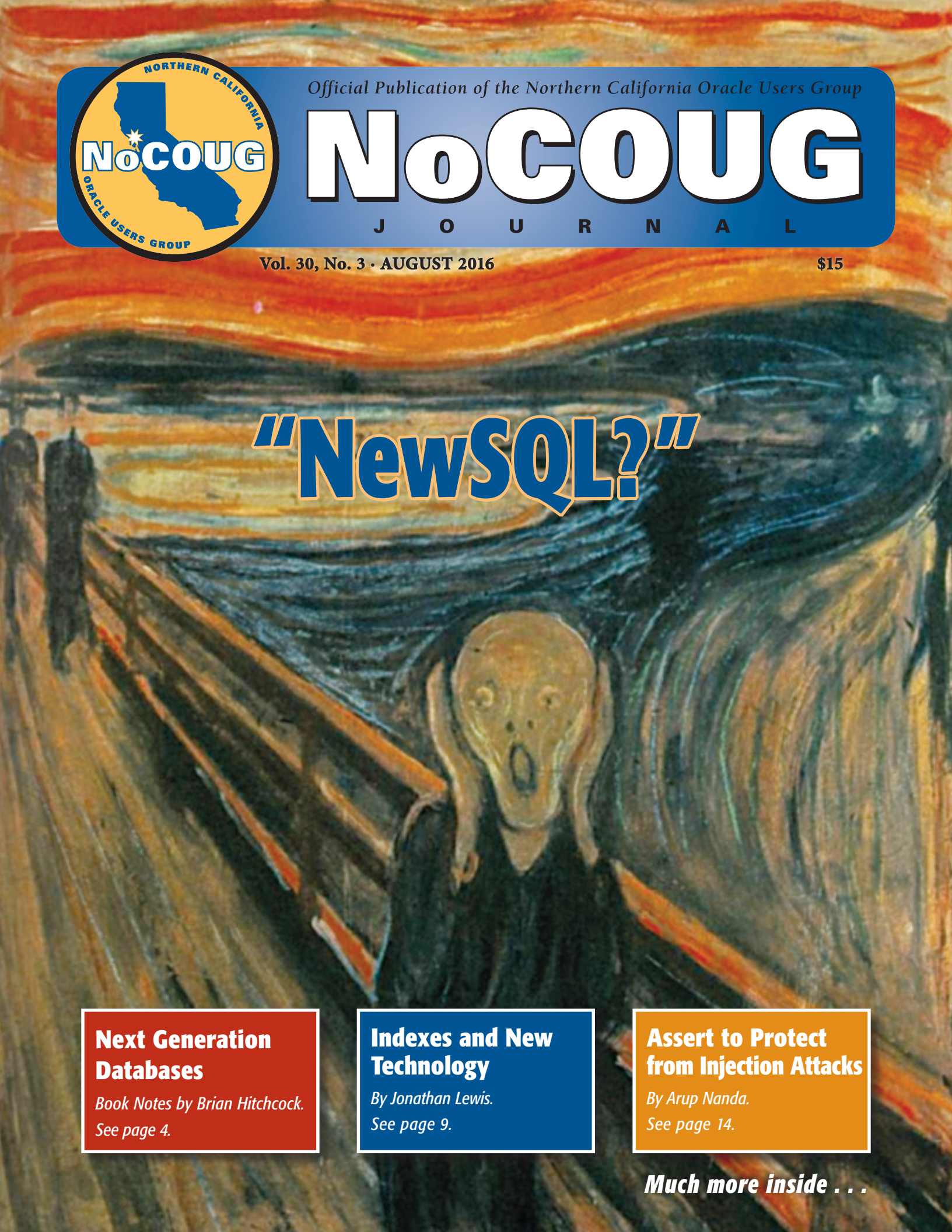
Official Publication of the Northern California Oracle Users Group

NoCOUG JOURNAL

Vol. 30, No. 3 · AUGUST 2016 $15



"NewSQL?"

Much more inside . . .

It's time for

ZeroIMPACT

Oracle database replication at half the cost

SharePlex

your golden alternative

Visit the Dell Software booth and experience how you can:

- Dramatically **reduce** downtime by up to 99%.
- Eliminate fire drills and **migrate** at your speed.
- Minimize risk and **prevent** data loss.
- **Validate** migration success.

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

This is the 119th 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.

New Generation Databases

Book Notes by Brian Hitchcock

Details

Author: Guy Harrison

ISBN-13: 978-1-4842-1330-8

Pages: 235

Date of Publication: December 14, 2015

Edition: 1

List Price: $39.99

Publisher: Apress

Summary

You should read this book. If you are starting out in a database-related career—which I think includes most, if not all, programming jobs—or if you are changing fields within the broader database field, the information in this book will be very helpful in the interview and on the job. I learned a great deal about things that I had already heard about along with many others that I have never heard of. Of all the books I have reviewed over many years, this is one of the very few that I think everyone could benefit from. I read this book online on Safari, and I recommend that you do the same.

Introduction

This book is split into two sets of chapters. The first set is called Part I, Next Generation Databases, and is made up of chapters 1–7. The second is Part II, The Gory Details, and is made up of chapters 8–12.

The first section covers the major developments in database systems, while the second part looks in detail at some of the issues that affect database systems in general.

Chapter 1: Three Database Revolutions

We start with a description of the three revolutions. The first was the development of computers, followed by the initial relational database and the range of nonrelational databases that we have seen more recently. A timeline is shown with a list of the major developments within each period. This timeline is broken into Pre-Relational 1950–1972, Relational 1972–2005, and The Next Generation 2005–2015. Before each of these revolutions is discussed in detail, we have a brief overview of the earliest history of the database, starting with structured datasets that at the time were known simply as "books." This was followed by loom cards that stored programs for fabric looms, paper tapes for player pianos, and punch cards used to process data for the U.S. census in 1890. The earliest digital computers that appeared after WW II used paper tape and, later on, magnetic tape to store their datasets. Rotating magnetic disks allowed random access to data, and indexing methods such as ISAM gave rise to the first systems that we would now describe as OLTP. While these developments are described as databases, we are told that they did not represent a DBMS because these datasets were completely controlled by the application.

The first revolution began when the logic that handled the database was separated from the data itself to form the first DBMS, usually running on IBM mainframes. I remember the first computer I used, an IBM 360 in the basement of my high school back in the 1970s. These early databases used one of two models to access data: the hierarchical or the network model. A diagram shows the structure of a sample schema in both of these models. Both of these models are described as extremely inflexible with regard to both data structure and querying the data. I didn't realize that these systems were all about record-at-a-time transaction processing which required lots of coding, usually in COBOL, to get anything done.

The second revolution was basically the rise of the RDBMS in the 1970s. This section starts with a review of the problems with the existing databases of the time. They were hard to use, weren't based on any theory, and didn't separate the logical and physical implementations of the data. This all changed when Edgar Codd published his famous paper defining the relational database model that would become the basis for most database systems that followed. There are sections covering transaction models, System R, the first relational database, and the database war that developed between Oracle and Ingres. This second phase of database development also saw the advent of client-server computing, object-oriented programming, and the object-oriented DBMS.

The third revolution happened in the mid-2000s, when the entrenched RDBMS products ran up against the endless performance needs of massive web applications. As the number of users went from thousands to millions, the existing RDBMS system could not keep up. At this time, Google was the biggest website on the planet, and it had to develop its own database systems. The result was MapReduce for parallel processing and the BigTable distributed structured database. These developments led to the Hadoop project at Yahoo; Amazon and Facebook had similar issues.

Oracle RAC is mentioned as an attempt to scale out the RDBMS architecture—an attempt, we are told, that failed. Cloud computing, specifically Amazon Web Services, is discussed as well as Amazon's SimpleDB and DynamoDB. This is followed by a discussion of document databases, JavaScript, XML, and JSON. The first of the NewSQL databases, in-memory and columnar databases, are described, followed by the nonrelational explosion.

This first chapter ends with a discussion of how any single type of database system doesn't work well for all the modern workloads. The three database revolutions are compared to the three platforms that developed at similar times: the first platform was the mainframe, the second was the client server, and the third was the cloud, including social media and mobile devices.

It is interesting to see so many topics covered so quickly; it really helps to see the trends. The author tells us that for all the changes we have seen, relational systems are still the best choice for many situations.

Chapter 2: Google, Big Data, and Hadoop

This chapter focuses on why Hadoop came to be. A quote from HBO's Silicon Valley is great, referring to "all the selfies and useless files people refuse to delete on the cloud." I wonder how much of what we call Big Data is actually wasted space. Hadoop was developed by Google to deal with massive unstructured data storage. How to define Big Data is discussed as well as the impact of the cloud, mobile devices, and social media.

The explanation of Google PageRank is great. Google's development of its own hardware and software stack is covered, and an example, with diagrams, of how MapReduce works is presented. Hadoop is the open-source version of the Google stack, consisting of Google File System, MapReduce, and BigTable. The author tells us that nothing else has had as big an impact on Big Data as Hadoop.

Hadoop's origins, its power, and its architecture are covered in detail.

Also discussed is HBase, which uses the Hadoop file system and was one of the first NoSQL database systems. A diagram compares the HBase data model to the relational equivalent. For all of Hadoop's power, accessing the data was not easy. Facebook and Yahoo developed Hive and Pig respectively as query tools for Hadoop. Hive is described as "SQL for Hadoop," since it made Hadoop accessible to anyone who was familiar with SQL. Yahoo's PIG is covered, and a diagram compares PIG's Pig Latin script to the equivalent SQL.

This chapter ends with coverage of the Hadoop ecosystem, which is made up of an interesting cast of components, including Flume, Oozie, and my personal favorite, Zookeeper. There is also a messaging component called Kafka. I find this amusing. Perhaps dealing with massive sets of unstructured data is worse than waking up to realize you have turned into a beetle? Hadoop is described as the most significant change in database architecture since the relational model, but it lacks support for transactions. This led to NoSQL, which is covered next.

Chapter 3: Sharding, Amazon, and the Birth of NoSQL

The author includes one or more quotes at the beginning of each chapter, and the quotes included in this chapter are great. I will quote the first one: "Step 1 – Shard database. Step 2 – shoot yourself." I will leave the second one for you to look up. I'm sure it reflects the feelings of many customers who contact product support—and I say that as someone who works in support.

This chapter describes the move to NoSQL systems. It begins with the observation that after the initial release of MySQL in 1995, it was ten years before the NoSQL web-scale transactional database appeared in 2005. It sounds somewhat silly, but the World Wide Web started out as a collection of static documents that are linked. We are told that this is called "Web 1.0" and the "vast majority" of websites are still this way. We have made great progress, but the more things change . . .

We now have Web 2.0, which has dynamic content and transaction support. I'm amused that the author explains that Web 2.0 did not come about by a controlled process of development; rather, it came about as developers scrambled to cope with the endless need to scale.

There are sections that describe how the transition from Web 1.0 to 2.0 came about. While it was relatively easy to scale up web servers, it was not as easy to do for the database servers.

As the author tells us, the most powerful database server could not keep up with the need. During this time, open-source software in the form of Linux and MySQL become more popular than UNIX and Oracle as the OS and database for website development. To overcome MySQL scalability issues, the Memcached utility and MySQL replication were used to provide a distributed object cache. This helped but only for the read capability.

Sharding provides partitioning so that the largest tables in the database are spread out across many database servers. We are told that Facebook uses MySQL replication, sharding, and Memcached to support some impressive read and update rates. I won't quote them all to you, but it is worth reading about. Sharding has a dark side, however, namely that it is very complex to maintain.

The next section, titled "Death by a Thousand Shards," describes the complexity issues. Oracle RAC is mentioned, in that it failed to provide an alternative to the MySQL sharded model. This is followed by a discussion of the Cap theorem, which states that you can have at most two of the following: consistency, availability, and partition tolerance. This leads to a discussion of different ways to deal with less-than-perfect consistency. Dynamo was Amazon's solution to their needs for a massive online website. Dynamo does not have a data model. Instead, it has a key value associated with an unstructured binary object: the key-value store. The discussion of all the tradeoffs made is worthwhile. Note that for all the technical wizardry, Dynamo is described as impenetrable to all but programmers.

Chapter 4: Document Databases

Document databases are new to me. They are defined as non-relational databases where data is stored as structured documents and where the structure is XML or JSON. Don't get locked into the "document" part; these databases can be used for almost anything and not just for storing documents. Document databases provided a solution to the issues between object-oriented and relational databases, a middle ground between no schema (key-value store) and the rigid schema of relational. Because the documents had a structure, ad hoc queries were possible.

There are sections discussing XML and XML databases, as well as how their success led to relational databases including XML support.

XML was useful, but it wasted space and was expensive to process. This led to JSON, the JavaScript Object Notation. While XML databases are good for content-management applications, JSON is better for storing and updating dynamic content and transactional data, which is needed for web applications. An example is given comparing a simple data model in both relational and JSON databases. Note that there is no equivalent to the third normal form of the relational model in the document database.

JSON databases are discussed, including CouchDB, which also used MemBase. This is basically Memcached that can be modified and stored on disk. It was used by Zynga for their Farmville online games.

When Google bought DoubleClick, they needed a new data storage engine that would be highly scalable to support serving ads online. They couldn't find anything that met the need, so they developed MongoDB. MongoDB is a JSON document database that uses BSON, which is binary encoded JSON and pro-

vides a reasonably useful query capability. A diagram shows a MongoDB JavaScript query and the equivalent SQL statement. Since I have not worked with document databases, I found the diagrams with comparisons to SQL very useful. MongoDB, we are told, is the most widely used nonrelational database because of its popularity with developers.

Relational database vendors have added support for JSON just as they did for XML. Some have added features to their implementation of SQL to access and manipulate JSON documents in the database.

Chapter 5: Tables Are Not Your Friends: Graph Databases

So far, we have been learning about relational, key-value stores, and document databases. Now we are told that these are all databases that store information about "things." When you need to work with relationships between things, you need something different. Enter the graph database, which is how Facebook tracks the relationships between people. It is interesting that the relational model is completely capable of supporting a graph database, but SQL isn't the best for graph data, and performance issues become a problem for large graphs. Also interesting is that the NoSQL databases described to this point are even worse for graph databases, because they don't support relationships between objects.

A graph is defined as consisting of vertices representing objects and edges or relationships connecting those objects. Both vertices and edges have properties. It turns out that graph databases have a formal theoretical basis. The basic operation is called a "graph traversal," which means moving through the graph to find, for example, friends of your friends. Yes, Kevin Bacon is mentioned. An example demonstrates how a simple graph structure could be supported in the relational model, but there isn't any SQL command to traverse the graph—and performance is an issue as you move farther along the graph.

Further discussion covers RDF, the Resource Description Framework for modeling web resources, and its query language SPARQL. Note that Oracle Spatial is an implementation of a native RDF database. A richer graph model is provided by the Property Graph; this was the basis of Neo4J, which is the most popular graph database. Neo4J is Java-based; it supports transactions and billions of nodes. The supplied query language is Cypher, and an example is shown to create a simple graph structure. While I hadn't heard of most of this, it was interesting to see examples of the different query languages. Gremlin is an alternative query language for Neo4J, and Oracle is supporting an open-source version of Cypher.

The last section of this chapter goes into the details of how graph processing is done. This requires a way to traverse the graph without needing to perform index lookups. This section explains why graph databases can't run on distributed servers, at least so far. Graph compute engines are presented, which can be used by other database systems that are not specifically graph oriented but need to support graph processing.

Chapter 6: Column Databases

The author's quote to open this chapter may be the best one of all. Before discussing column databases, the author explains how much we are conditioned to think of data as a set of rows, based on the books and spreadsheets we use all the time. It is interesting how much of our world is based on some very specific and very arbitrary events. Writing, printed on pages, top to bottom, left to right, could all have been done differently. High-tech systems like databases turn out to be based on these arguably arbitrary things.

As with some of the previous topics, things in the relational world were just fine as long as long-running analytical programs were running in the background, during off-hours. As analytic and decision support applications started to demand nearly interactive response times, things needed to change. Having separate systems for OLTP and data warehouse applications wasn't enough. They had to use different schemas as well. The well-known star schema is reviewed with its fact and dimension tables. However, even with a dedicated schema, relational data warehouse applications were limited by CPU and I/O performance.

The alternative approach is to store data in a columnar format where the values for a specific column are located in the same data blocks. For many analytic queries, you want to aggregate many values in the same column so performance improves. The columnar format also supports efficient compression, since many of the values in a column will be the same. However, the downside is updating or inserting rows, since this will affect many data blocks. The commercial implementations Sybase IQ, C-Store, and Vertica are discussed. There are solutions to the insert and update issues. These architectures include write-optimized delta store, which is memory resident and usually not compressed. Oracle's Enhanced Hybrid Columnar Compression (EHCC) is described as an interesting attempt to have the best of both row and column storage formats. The author doesn't comment on how successful EHCC is in achieving these dual goals.

Chapter 7: The End of Disk? SSD and In-Memory Databases

I really liked this chapter. While I have always been aware of how much we all want to avoid physical I/O, I had not realized how much memory and disk performance have diverged. CPU and memory performance followed Moore's law to ever-greater levels of speed, but disks, not so much. What would our world be like if spinning disk access times had followed Moore's law? This is answered early on. The edge of the disk would have to be moving at 10 times the speed of light! This means that memory and CPU performance has been constantly moving away from that of physical I/O. The performance cost of physical I/O has been getting worse and worse over time. I didn't know that while Solid State Disk (SSD) is much faster than magnetic disk, writing data to SSD is much slower than reading. The details of why this is so are presented as well as a discussion of the economics of disk in general. I also learned that relational databases that usually depend on fast writes for some form of redo log perform badly on SSD. Several solutions to these problems are discussed.

The details of in-memory systems are covered next. There is a very good chart showing how much the cost of memory has fallen while the capacity has gone up (make note of the exponential trend). It is important to realize that while memory is more capable than ever, the size of most databases has also grown very quickly. It is practical now for many midsize databases to operate entirely in memory. In-memory systems have to address the fact that data could be lost if there is a power failure and, if the database is contained entirely in memory, there is no need to cache any data. This requires changes to the database architecture that can include replicating, snapshots to disk, and some form of transaction log.

At this point, various commercial implementations of in-memory databases are discussed, including TimesTen, Redis, SAP Hana, and VoltDB. Oracle 12.1 has a new feature, the "in-memory" feature, which is a column store to work alongside the usual disk-based row store. The chapter ends with coverage of the Berkeley Analytics Data Stack and the Spark processing engine, which is an in-memory implementation of Hadoop.

This chapter was my favorite. I was aware of the relative performance and cost issues of disk, SSD, and memory, but I had never appreciated why disk wasn't keeping pace and just how quickly memory had improved in terms of cost and capacity. Just to be clear, disks aren't going away; they still offer the lowest cost for storing the most data.

Chapter 8: Distributed Database Patterns

The discussion now turns to distributed databases, both relational and nonrelational. As databases grew to support web applications that could suddenly need huge increases in capacity, it was difficult to continue the existing trend of ever-larger single servers. Scaling out became more important than scaling up. Distributed databases provided a solution and could be built on commodity servers that were much less expensive than a single massive server from one vendor. Before covering distributed database, there is a review of the progression from the mainframe with dumb terminals to client-server to web applications, all of which used a single database server. Replication was used as an initial step toward distributed databases.

The merits of shared-everything, shared-disk, and shared-nothing distributed database servers are covered. Transactions, rebalancing, and cached data are all issues affecting the usefulness of each of the shared architectures. Having looked at distributed relational databases, similar issues are considered for the nonrelational databases. Things like balancing availability and consistency come up. Since I have always worked with relational, it is interesting to read about systems where consistency is something you can dial up and down, and trade off against other things. It's sort of like turning gravity up, down, and maybe even off. For the nonrelational databases, MongoDB, HBase, and Cassandra are discussed, covering sharding, replication, and hashing. Sharding and various sharding mechanisms are covered in detail with multiple diagrams. I liked the sections on write concern and read preference—so many ideas I hadn't heard about before.

Chapter 9: Consistency Models

This chapter formally discusses consistency. The ACID model of consistency, wherein transactions are atomic, consistent, isolated, and durable, is used by relational databases and usually requires locks to implement. The nonrelational designs wanted to move away from the restrictions imposed by ACID consistency. The author explains that while many people believe nonrelational systems don't provide much in the way of consistency, this is not accurate. They can supply strict consistency, but only at the single-object level. Various types of consistency are explained: consistency with other users, within a single session, within a single request, and my favorite—consistency with reality. That last one is not very popular at staff meetings!

A second consistency model that has been used in relational databases is MVCC, multi-version concurrency control. MVCC reduces the locking issues and still provides ACID consistency. Transaction timestamps are used to control which version of data is shown to which queries. Oracle calls this the "system change number (SCN)," while Microsoft SQL Server calls this the "transaction sequence number." For transactions that need to go across multiple databases, we have two-phase commit (2PC). I thought 2PC had pretty much gone away because of the performance penalty as each transaction goes through the two phases, the commit-request phase followed by the commit phase. You can imagine how much time this could take if you had many nodes in your distributed database. It is also explained that for 2PC, if a transaction fails, it can become an in-doubt transaction. And we all know what we do with an in-doubt transaction: that's right, we contact the database administrator!

While ACID transactions are the highest level of consistency, there are other models. Most nonrelational systems limit consistency to a single object or operation. Many different levels of consistency are described for single-object operations, from strict to eventual and even weak. This is all summarized with the observation that nonrelational systems use strict or eventual consistency and RDBMS systems offer ACID.

These issues are reviewed for MongoDB, covering locking, replica sets and eventual consistency, and HBase. Cassandra and its tunable consistency is covered in detail with sections on write and read consistency, timestamps and granularity and lightweight transactions. I was especially fascinated by the description of a vector clock, which is a way of implementing timestamps used in Cassandra.

Chapter 10: Data Models and Storage

This chapter reviews and compares the various data models that are used by the database systems discussed in previous chapters. The data models described include relational; key-value; Google's BigTable; JSON in document databases; and the node, relationship, properties model of graph databases. The relational model is reviewed (again) to cover both normalization and the star schema used in data warehouses. Key-value stores are next, but there is no formal definition for data representation in the model, since the key and the value stored are both binary data. A specific key-value store, Amazon's Riak, is discussed in detail. This includes further discussion of vector clocks that Riak uses to handle update conflicts. Google's BigTable forms the basis of the data model used in HBase and Cassandra, among others. One of the specific features of this data model is the column family structure where related columns are grouped to optimize disk access.

Cassandra's query language (CQL) is explained with examples as are Cassandra collections that support repeating groups within column values. JSON is the data model for document databases, and these documents are made up of arrays, objects, and values. BSON is a more efficient way to set up JSON data.

One of the reasons that the relational model has been so successful is the inherent separation between the logical data representation and the way the data was actually stored. This separation allowed the development of the columnar database. The way data is stored in a typical relational system is shown, from the database files to the buffer pool to the transaction commits being written to some form of redo log. It is pointed out that almost all transactional database systems have some form of transaction log, whether they are formally a relational system or not. The B-tree index is a staple of the relational world, and its structure is detailed as well as the expense of maintaining the index when data changes cause index splits. This has led to the development of the log-

structured merge (LSM) tree that is designed to handle applications that require lots of write activity. The LSM tree has an in-memory tree and many on-disk trees. This is used in both HBase and Cassandra. As data changes, the in-memory trees are periodically merged with the on-disk trees by the compaction process.

This discussion also includes SSTables, bloom filters, and tombstones, which are the way deletes are implemented. The details of the compaction process reveal that it is possible for a row that has been deleted to be resurrected. I had to read that part several times, and I'm still not sure I believe it. The last section covers secondary indexing. Relational systems support multiple indexes on the same data. Indexes can be built on a primary key and one or more foreign keys. This doesn't work for nonrelational systems. In a key-value store system, the only way to index is on the key-value. The details of how secondary indexes are implemented in NoSQL databases are provided for Cassandra, MongoDB, Riak, and Hbase.

Chapter 11: Languages and Programming Interfaces

Here we look at SQL and its rivals. The two quotes supplied by the author to start this chapter are worth serious study and contemplation. It won't come as a surprise that SQL is far from perfect, but it is also a survivor. Love it or loathe it, SQL continues to be useful to enough users that it hasn't really been superseded. A basic primer for SQL is provided, including a listing of the major ANSI and ISO SQL standards. I had never seen this before: the select statement is referred to as DQL (data query language). I guess we always need more acronyms. The author's summary is this: "SQL remains the most significant database language."

Having said this, SQL isn't an option for many of the NoSQL databases, but these systems have to provide some way to get data into and out of the database. We are told that these systems were, for the most part, developed by and for programmers, which means a low-level Java API is all that is provided. For many of the available NoSQL databases, there is a section describing in detail the API that is supplied. I am not a programmer, so each of these sections seemed very detailed, and it made me realize how hard it would be for me to access data through these APIs.

As already mentioned, SQL is a survivor, and the next section discusses how SQL has been added to many NoSQL products. After this we have sections covering the SQL implementation provided by various NoSQL systems. It is worthwhile to read these sections even if you won't be trying to use any of these SQL-like APIs. It really shows why SQL is so popular. These sections also make it clear how hard it is to integrate various NoSQL products when each has its own version of a SQL-like API. There are sections for Hive, Impala, Spark SQL, Couchbase N1QL, and Apache Drill. Finally, there is a listing of what the author describes as SQL-on-NoSQL systems. For each of these we have a brief description and which NoSQL databases it works with.

Chapter 12: Databases of the Future

Up to this point, we have been reading about the history of databases and all the various systems that exist today. Now the author turns to the future and tells us that the current situation with many different database architectures will move to a hybrid system that can handle most if not all of the required workloads. This begins with a section that revisits the revolution(s) that we have been examining through all the previous chapters. Next is a review of the nonrelational systems that were created to address the web application workloads that the existing relational systems could not handle. There is a debate as to whether we have simply come back to where we started, with today's nonrelational systems resembling the pre-relational systems that we started with.

The author reviews and explains both sides of this. Next is the question of how to decide which of the available database systems is best. Interestingly, after all that we have seen, we are told that relational is still the best choice for most applications. Things get interesting as the author offers that we can "have it all" in a single database system. Such a system would support tunable consistency from strict ACID transactions to eventual consistency, as well as the features of a relational model and a document store with JSON data types within relational tables. I'd like to hear how all this would work. Specifically, if I enter a query, how will the database decide which pieces of the hybrid system will parse, process, and retrieve the needed data? Will the optimizer be capable of processing many more possible execution plans that could use various permutations of all the pieces of a hybrid database system? I assume we will see how this all plays out during the next stage in the ongoing evolution of database systems.

The sections that follow show how various parts of the currently available database systems could be pulled together into a hybrid system. These sections cover consistency models, schemas, database languages, and storage. An idealized consolidated database system is described. We are told that Oracle has come as close as any other vendor to this future database. This is followed by a detailed discussion of how Oracle is supporting many of the features of this future hybrid database. There are sections on Oracle support for JSON, graph, and sharding. While Oracle gets the most attention, other systems that are trying to combine relational and nonrelational models are NuoDB, Splice Machine, Cassandra, and Apache Kudu. A final section covers some new technologies that could disrupt the author's vision of the future, including storage, blockchain, and quantum computing.

Appendix A: Database Survey

This section provides a short description for each of the database systems discussed in the book. For each we have information about the company that supplies the database, the vendor's description, the author's opinion of the product, the data and transactional model used, and comments on support for clustering and APIs for querying.

Conclusion

I don't think I've read a database book that presented so much material that was new to me in such a short time. This book is an excellent review of where we are, where it all came from, and where we are going. It is also an excellent value because it doesn't take long to read and you will learn a great deal about the entire database field. ▲

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

Indexes and New Technology

by Jonathan Lewis

Jonathan Lewis

Over the last eight years there have been many dramatic changes to the technology underpinning the Oracle RDBMS. We have Exadata systems, in-memory columnar storage, container database, and the move to cloud computing. The changes in software apply not only to the storage mechanisms but also to the optimizer features available.

Despite all the changes, though, there's still the common hope that we can make the most cost-effective use of the technologies we've licensed—and for critical systems, that still means understanding how to create the best indexing strategies.

Intent

Before we look too closely at new technology, it's worth doing a quick review of the environments we have to work with and why we worry about indexing.

Broadly speaking, we can split the possible environments into two: OLTP (online transaction processing) and DSS/DW (Decision Support System/Data Warehouse), bearing in mind, of course, that even the purest OLTP system is likely to have end-of-day, end-of-week, and end-of-period processing and reporting requirements that can look very much like DSS processing, so there's rarely a completely clear-cut distinction between the two extremes.

So why do we worry about indexing? First, because indexing gives us the option of getting the results we want in the least amount of time *or* using the fewest resources. The two are not always mutually compatible; in fact, when you consider the "fewest resources" requirement, you should extend that to "fewest resources when it really matters." And remember: you can always decide to use more resources when they're available to minimize resource usage when resources are scarce. Looked at in this fashion, "indexing" can expand itself to include such things as using materialized views, using clusters, cloning data, and using index-organized tables —all currently available strategies that allow you to work harder at one point in time to make it possible to get a faster response at another point in time. In fact, of course, basic indexes (B-tree or bitmap) are just ways of cloning data and investing resources as data goes in to allow faster access when you want to pull the data out. For DSS/DW systems (particularly) you can also consider the fact that partitioning may give you the benefit of "indexing" without the costs of index maintenance: if you can partition a table into 1,024 pieces in just the right way (and that's not always possible), then a segment scan of the one interesting partition might be faster (and possibly more efficient) than using an index to pick one row in a thousand from a non-partitioned table.

Indexing comes down to knowing the available technology options so well that we will be able to work out how a particular option can help us save time or reduce (or relocate) resource usage but, at the same time, allow us to recognize what threats that option will introduce at other points in our system.

Old "new technology"

I am frequently surprised how even some of the oldest features of indexes fail to get into production systems—especially systems that involve in-house coding. Two features, in particular, are index compression and "function-based" indexes.

Consider an index like ***(client_code, order_date)***—without going into lots of technical preamble, I hope I can say that this looks like the type of index you might have on an ***orders*** table where each client places many orders over time. In an index like this where the first column (or columns) of an index are likely to be repetitive, it's almost certain that the index should be created with a level of compression (which, in this context, means de-duplication). So *"create index ord_i1 on orders(client_code, order_date) compress 1"*. You are likely to see a tiny increase in CPU usage during maintenance and retrieval but have an index taking up less space at steady state. It's surprising how little repetition is needed before the space saving from de-duplication starts to outweigh the extra metadata in the leaf block—even with a very short column, an average of four repetitions is enough to start saving space.

Note 1: Index compression cannot be used with bitmap indexes.

Note 2: For further details on the mechanisms and costs of index compression, see: http://allthingsoracle.com/compression-in-oracle-part-4-basic-index-compression/ and http://allthingsoracle.com/compression-in-oracle-part-5-costs-of-index-compression/.

"Function-based" indexes (I use quotation marks on the name because it's a shorthand for "indexes that include function-based columns") are a particularly wonderful way of creating extremely precise indexes with minimal costs of maintenance while introducing virtually no risk of being accidentally hijacked by other parts of the application, as often happens when you try to add indexes to an ailing system. Function-based indexes can be B-tree or bitmap.

Consider a table where data arrives and each row passes

through several states, ending up at a "completed" state—let's say we have a column called ***state*** that can hold the values "New," "Picked," "Delivered," Invoiced," and "Completed." Over a period of years we have accumulated several million rows that are in the completed state, but at any one moment we have a couple of thousand rows in the other states, and we have code that wants to identify the small number of rows that are not yet completed. It makes sense to create an index based on the ***state*** column—but the typical approach to doing this is to create a simple, though huge, B-tree index on the column and then discover that it's necessary to hint the code or create a histogram on the column to make sure that the optimizer knows that the data is extremely skewed and queries for any value other than "Completed" should use the index. Cunning use of the technology, however, allows us to do the following:

```
create index ord_fbi1 on orders(
          case when  state != 'Completed' then state end
) compress
;

-- gather simple stats on the hidden column

select    *
from      orders
where
          case when state != 'Completed' then state end = 'New'
;
```

The index is the smallest possible index identifying just the rows we might be interested in, requiring the smallest amount of real-time maintenance. It needs a carefully crafted SQL statement to trigger its use and doesn't need a histogram to be gathered because there's (relatively speaking) no skew across the small number of distinct values it holds. In newer versions of Oracle we're more likely to create a "proper" virtual column for this expression and index the virtual column, and if we're using 12*c*, we make that column invisible.

Slightly Newer Technology

As the technology changes we need to repeat the questions we originally asked when we designed our indexing strategies: How much work does it take to maintain this index?, How much work does it save?, and How much contention does it introduce? Consider the effects of serial direct path reads and Bloom filters (as a couple of examples of relatively recent changes to the Oracle code base).

Let's start with the ***serial direct path read***—an enhancement that appeared in 11*g* to allow tablescans (or index fast full scans) to bypass the buffer cache and be read directly into the private memory of a process.

Part of the resource consumption we normally expect from a tablescan comes from the need to find free buffers, acquiring cache buffers chains latches and cache buffers LRU latches, possibly moving buffers to the write list, possibly forcing buffers to be written by DBWR, and possibly forcing DBWR to tell LGWR to write the log buffer to disc. That's quite a lot of work, and we could end up with a large segment scan taking over nearly 25% of the buffer cache for the duration. All in all, that's a lot of work, a lot of CPU (latch) competition for the rest of the application, and a lot of data being flushed from the buffer cache.

Reading the table into private memory eliminates a lot of CPU and a lot of latch contention—so much so that we might seriously consider dropping an index that we've created especially for a particular large-scale query because it uses so much less resource to scan into private memory than it does to read through an index. Of course when considering this decision, we do have to know that a direct path scan is preceded by a segment-level checkpoint that writes all dirty buffers *for that segment* to disc before the scan starts. And we do need to consider the nature and frequency of that query: would buffering (at least some of) the segment be helpful because of the frequency with which this query runs?

I picked ***Bloom filters*** as my second example because it's nice to mention joins in the context of indexing, and Bloom filters (which first appeared in 10*g*) are particularly relevant to optimizing joins. Again, we can consider the case of a query that accesses data scattered across the length of a very large table. We may have decided that with a suitable, precisely targeted index, the best (or, perhaps, least worst) access path for a particular class of query is a nested loop join from a driving table into this large table. Serial direct path reads might start to move you in the direction of getting rid of that index and using a tablescan with hash join instead—but part of (and sometimes the major component of) the work done in a hash join is the CPU of probing the in-memory hash table (i.e., the build table) and following the linked lists where hash collisions occur to discover whether or not you've got a false positive.

Bloom filters allow Oracle to eliminate most (usually) of the tablescan rows that are not going to find a match in the build table before attempting the hash join, thus saving CPU. The effect is much more noticeable on parallel execution, of course, where Bloom filters can eliminate huge amounts of data that might otherwise have to travel between parallel servers before being discarded—and they're even more noticeable on Exadata systems where the Bloom filters can be passed to the storage servers so that the only data coming back to the database server is data that is (probably) going to be needed in the join.

Are there any downsides to Bloom filters? No, except that sometimes you suddenly notice that they're not being used when you thought they would be; at present there are circumstances where a Bloom filter will not be used in an "insert/select" statement when it is used in the corresponding select or create as select.

If you've got any very large tables with indexes that you've had to create for a specific one or two tasks, it's worth reconsidering your options and reviewing the possible costs and benefits of using any new feature that's aimed at reducing workload or the data volume that you have to process. New features will probably have hidden costs, but they may allow you to get rid of a few very expensive indexes that have been constructed for very limited purposes.

New Technology

I suppose you can't really call Exadata and all its adjuncts, such as storage indexes and hybrid columnar compression, "new" anymore, but we're getting a lot closer to the leading edge, and the in-memory columnar store brings us right up to date. Before I get there, though, I'd like to mention a new feature of 12*c*: partial indexes on partitioned tables.

In a similar vein to function-based indexes, which allow you to index only the most interesting data, partial indexing allows you some freedom to limit the partitions that appear in indexes on partitioned tables. One of the simplest examples of this fea-

ture would be to think of a table using time-based range partitioning; you could imagine having several indexes on this table to support popular queries about the most recent data but not wanting to waste resources with an index covering the entire table.

You can, with a little care and a lot of discipline, set this up manually for local indexes in earlier versions of Oracle, but 12*c* allows you to do this (a) in a robust fashion, and (b) for global (or globally partitioned) indexes as well. In 12*c* you use the *"indexing off"* clause to identify each *table* partition that should be ignored when an index is created and then identify which indexes are allowed to ignore these table partitions by using the *"indexing partial"* clause as you define the index. For example:

```
create table pt_range (
    date_loaded date,
    client_code number
 )
partition by range(date_loaded) (
    partition p2014 values less than (to_date('20150101','yyyymmdd')) indexing off,
    partition p2015 values less than (to_date('20160101','yyyymmdd')) indexing on,
    partition p2016 values less than (to_date('20170101','yyyymmdd')) indexing on
)
;

create index ptr_i1 on pt_range(client_code) indexing partial;;
```

This index will be a global index, but it won't hold any index entries for partition p2014. (The *"indexing on"* option is the default for the table in this case, so it isn't actually needed here.) If you now run a query like *"select where client_code = 123 and date_loaded = to_date('01-Jan-2016')"*, the optimizer can choose different execution paths for different partitions based on the different index availability.

Not only does partial indexing save space, but (for any non-unique global indexes that don't include the partitioning key) the number of leaf blocks Oracle has to range scan through is reduced because of the absence of keys relating to the unindexed partitions (which is a side effect of the way the ***extended rowid***, which is stored as if it were the last column of the key, starts with the ***data_object_id*** of the table partition).

The example I've given is about global indexes, but the same mechanism can be used with local indexes. As far as local indexes are concerned, I'd actually like to see this idea pushed a little further—I'd like, for example to be able to have a robust mechanism to define an index (or pair of indexes) on ***(colX)*** with the proviso that it should be a ***bitmap*** index for old, static data and a ***B-tree*** index for new, still-changing data. It's something I can fake in 11*g*, but I don't like doing it. At present the requirement that there is one pattern of "on/off" that applies to all partial indexes is just a little too restrictive; nevertheless, the implementation as it stands offers great scope for improving the cost/benefit balance when creating indexes on partitioned tables.

Partitioning, of course, tends to push us into the arena of large data volumes, where the Exadata solution first appeared as the massively powerful machine that was engineered to handle enormous tablescans very quickly—partly through parallelism, partly through partitioning, and partly through some very clever software (and partly through extreme hardware, of course).

New Technology (Exadata)

Exadata's key features were

- Hybrid Columnar Compression (EHCC) that, on a good day, could pack a terabyte of data into a mere 10 GB of storage so the job of physically reading the data into memory could be much quicker
- Separation (for direct path reads) of database server work and storage server work, allowing the "heavy lifting" bit of the work to happen away from the "complex analysis" bit of the work with predicates—including Bloom filters—being offloaded (pushed down) to the storage servers, and column projections (not database blocks or "compression units") being returned
- EHCC, again, because it could store the values of a given column from more than 32,700 consecutive rows as a single chunk, making comparisons like "column = {constant}" much more efficient because it didn't require any code to calculate its way along every single row to find the right column before doing the comparison
- Storage indexes, holding metadata about each megabyte of compressed data, so that you could skip reading a megabyte at a time because a tiny metadata check could tell you that that megabyte held no data of interest

The savings in I/O time and (server) CPU time were so dramatic that that the initial marketing legend suggested that you'd never need indexes again—for DSS and DW systems, at least. It didn't take long to discover how overly optimistic this view was on two counts.

On the "brute force" side, it is possible to do a tablescan incredibly quickly, but if you had to do several very large tablescans with hash joins, the resources needed in the join might be far

greater than the resources needed for the tablescans—both in terms of CPU and, if you weren't able to engineer the join correctly, in the consequences of Oracle being unable to use the Bloom filters early enough in the plan. In effect, you might find that you saved 99% on half of the job because the tablescans were fast, but you saved nothing on the other half of the job. As far as big joins are concerned, the heavy lifting isn't necessarily about getting the starting data from the base tables. Moreover, if the column projection being returned on a call to the storage server exceeds the limit on the message size (1 MB), then the compression unit would be returned and the whole decompression and extraction process would have to be repeated at the database server.

On the indexing side of the equation it doesn't take long to realize two things: first, just because you're doing work at the storage servers, that does not mean you're *not* using up a finite set of resources (apologies for the double negative); you're just hiding the usage at the far end of a wire (Infiniband). Getting the job done as quickly as possible isn't necessarily the best way of doing the job. Second, even if you can scan and filter 100 million rows per second but need to acquire 2,000 randomly scattered rows from a billion-row table, it's probably still quicker, and less resource threatening, to fetch those rows one at a time through an index. The thinking behind the choice between tablescans and indexing hasn't changed; what has changed is the number of mechanisms you need to consider when deciding where the break point between the options should be.

Some of the points that need to be considered are fairly easy to identify: e.g., what level of compression should be used (space vs. CPU)?, which objects should be assigned to the flash memory?, etc., but some of the traditional points have become more complex. Historically, a large index range scan on a large table could leave you rereading table blocks as you walked through the index; similarly, with EHCC even a relatively small (2,000 rows, say) index range scan could leave you repeatedly decompressing the same compression unit and using up surprising amounts of CPU at the database server. The standard questions of "how much data?" and "where is it?" still apply when deciding whether or not an index would help, but now you need to think about which rows are in which ***compression unit*** and consider the effects on your CPU of the threat that possibly one row equals one compression unit.

Here are a few figures from 12.1.0.2 using a ZFS Pillar VM for a query to select 600 rows from 12,000,000 using an index on a data set that showed an unlucky pattern of distribution; tables ***t1_query_low*** and ***t1_query_high*** were created as a simple, serial, CTAS from table ***t1*** to create copies with (as their names suggest) "query low" and "query high" EHCC.

The table sizes were as follows:

```
TABLE_NAME             BLOCKS
-------------------- ----------
T1_QUERY_HIGH            3726 (931 compression units)
T1_QUERY_LOW            41154 (10,505 compression units)
T1                     193518
```

First the query and plan (while using t1_query_high):

```
select id from t1_query_high where n_120 between 1 and 5;
-------------------------------------------------------------------------------
| Id  | Operation                          | Name          | Rows  | Cost (%CPU)|
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |               |       |   724 (100)|
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| T1_QUERY_HIGH |   718 |   724   (1)|
|   2 |   SORT CLUSTER BY ROWID BATCHED    |               |   718 |     4   (0)|
|*  3 |    INDEX RANGE SCAN                | T1_QH_I1      |   718 |     4   (0)|
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("N_120">=1 AND "N_120"<=5)
```

You'll notice the operation "sort cluster by rowid batched"—this is Oracle's attempt to minimize the number of compression units it has to visit by sorting the rowids it has extracted from the index so that if two rows belong in the same compression unit, that unit will only be decompressed once. This can help, but unfortunately Oracle seems to do this by fetching arrays of 101 rowids at a time. My ***n_120*** column happens to be defined in such a way that every compression unit will hold five suitable rows, but any one batch of consecutive rowids fetched by the index won't necessarily see (and sort) all those rowids. (The operation doesn't appear for the noncompressed table, of course, but irrespective of the level of compression, the cost of the query was reported as 724.)

A few performance figures after flushing the buffer cache:

```
Non-compressed table
DB time: 0.05 seconds
CPU Used: < 0.01 seconds
Table physical reads: 190

Compressed for query low:
DB time: 0.99 seconds
CPU Used: 0.20 seconds
Table physical reads: 2,148
EHCC Query Low CUs Decompressed: 600
sorts (memory): 6
sorts (rows): 600

Compressed for query high
DB time: 0.83 seconds
CPU used: 0.18 seconds
Table physical reads: 1,044
EHCC Query High CUs Decompressed 600
sorts (memory): 6
sorts (rows): 600
```

You'll notice that for both the compression examples Oracle had to decompress 600 CUs, even though the required data was actually limited to 120 compression units with five relevant rows per unit. You'll also notice that the DB time was startlingly low considering the number of (db file sequential) reads—which is the side effect of caching at the O/S level.

I was a little surprised by the last two results. I had been expecting both of them to do far more work than the query against the noncompressed table, but I had thought that the query high compression would result in higher CPU utilization than the query low compression. I suspect that in this case the reduction in blocks read into the buffer cache probably offset any other increase in CPU due to the compression method. Nevertheless the point is clear—the way in which you estimate the relative cost of indexed access is made more complicated by two factors: the need to access compression units rather than blocks and the CPU cost of decompressing to apply predicates.

New Technology (In-Memory Column Store)

The Exadata dababase machine hides some of the CPU usage (and its *"storage index"* technology) at the far end of a wire, so my little demo using nothing but the ZFS storage technology doesn't give a complete picture of every new detail you have to think

through when trying to balance the costs and benefits of tablescans and index-based access for Exadata. You could make a similar point about the In-Memory Column Store (IMCS): the key benefit you aim for with IMCS on an Exadata database machine may not match the benefit you have in mind if you're starting from just ZFS storage or even basic disk storage.

For many systems IMCS will address the question, "What shall I do with all this (spare) memory?"; for Exadata systems it can also address a gap in the Exadata design—specifically, you may find that the Vector Transformation that the optimizer is allowed to produce when the ***inmemory_size*** parameter is set is of far greater value than the savings you can make by trying to change your indexing strategies. Whatever your starting point, though, IMCS does change the balance of power between tablescans and indexed access paths.

Having created a 12 M row table for my previous example, I modified it with the following command:

```
alter table t1 inmemory priority critical memcompress for query high;
```

It was lucky that I had assigned 500 MB to the in-memory column store, because my table took up 370 MB in the 1 MB pool (with a further 1.6 M in the 64 KB pool). You might compare this (not very favorably) with the effect of the ***query high*** compression I had previously achieved—my data with ***memcompress high*** actually needed more memory than the disk space needed for ***EHCC query low***. In part this is because the in-memory compression methods used are engineered to (a) allow Oracle to do predicate comparison without doing decompression, which should lower the CPU usage, and (b) allow modern CPUs to use vector operations that allow comparison against multiple rows to be performed in a single step, potentially allowing filter operations to run some 4 to 8 times faster.

With my table in-memory, my base query against t1 (hinted with a /*+ ***full(t1)*** */ ran in 5 hundredths of a second, using 4 hundredths of a second CPU, saving significantly on the CPU consumption used in the EHCC tests for indexed access, and with a similar DB time to the indexed access path to non-compressed data. So the question I'd have to ask myself is this: "Is it worth committing 370 MB of memory to the table and using 4/100 sec CPU rather than creating and maintaining that index and (possibly) seeing a couple of hundred physical accesses for each query execution?" There is no easy answer to this question, but a few experiments with real data and real queries can give you a good idea of where the best trade-offs can be found, especially if you also make use of the "segment statistics" ***(v$segstat)*** information to get an idea of which indexes are subject to the most change and most reads; the fewer indexes you have, the more likely it is that blocks you need to read (and need to modify) stay cached longer and get written to disc fewer times per modification.

Another detail that comes into consideration with IMCS when reviewing indexes, though, is that the IMCS equivalent of Exadata's storage indexes (i.e., the metadata stored in the 64 KB pool) cover *every* column in the table and contain a little more information about each column than the equivalent storage index—allowing Oracle to make use of the metadata to bypass compression units in many more cases than Exadata can. So the extra price you pay in memory effectively gives you an index on every column—it's almost like being able to create bitmap indexes on every column in an OLTP system without the disastrous locking effects that bitmap indexes produce on concurrent DML.

For comparative purposes, my time for a serial tablescan of the *"query low"* version of the table was 145 seconds, of which 20 seconds was CPU time and 125 seconds I/O time, while the time for the tablescan of the *"query high"* version of the table was 15 seconds, of which 5.5 seconds was CPU time and 9.5 seconds I/O time. In both cases the I/O wait time was made up of a relatively small number of direct path reads of 32 blocks each and a much larger number of db file sequential reads (though ***v$sesstat*** and ***v$segstat*** were far from consistent in the number of *"physical reads"* and *"physical reads direct"* they recorded). There seemed to be some very buggy behavior going on, so I'm not inclined to trust these figures as the basis for making any decisions about typical production systems: you have to test on the hardware you've got for the data (volume and pattern) you've got.

Conclusion

Indexing strategies aim at reducing response time or (possibly, and) reducing resource usage at the moment it really matters. The mechanisms in the database that make this possible have been changing constantly over time, and as the technology changes we ought to review the indexes we have created. We may find that we can take advantage of new features to eliminate some indexes and reduce the cost—or improve the effectiveness—of others. Whatever technology we use, though, the basic questions don't change:

- How much data am I trying to access through this index?
- How scattered is that data?
- How much benefit do I get from this index?
- How much, and when, do I pay for that benefit?
- What are the costs of not having it?

The questions stay the same, but the technology gives us new details that we ought to consider as we answer those questions. ▲

Jonathan Lewis is a well-known figure in the Oracle Database world with more than 25 years of experience using the software. He has published three books about Oracle Database—the most recent being "Oracle Core" published by Apress in 2011—and contributed to three others. He runs a couple of websites and contributes to newsgroups, forums, and user group publications and events around the world. Jonathan has been self-employed for most of his time in the I.T. industry. He specializes in short-term assignments, typically of a design, review, or troubleshooting nature—often spending no more than two or three days at a client's site to address problems. He conducts Oracle Database seminars all over the world and has visited more than 50 different countries and more than a dozen US states to talk about, or troubleshoot, Oracle Database.

Assert to Protect from Injection Attacks

by Arup Nanda


Arup Nanda

John Smith, the principal database architect at Acme Bank, has been no stranger to visitors with elevated emotions, but today the bar seems to have been raised even higher. The atmosphere is a cross between somber and frantic. The cause: the bank's applications have been hacked by SQL injection attacks. Like many organizations, the developers at Acme have taken all well-known preventive steps against this risk, verified and certified by numerous database security specialists. However, their effort came under scrutiny when an attack actually happened that caused massive financial loss as well as irreparable damage to the bank's image. It's no wonder this incident attracted senior management attention and a demand to take some "concrete and actionable steps," which brought this coterie of visitors led by Clara, the chief information officer of Acme, to John's office today. Obviously, declares Clara, the normal SQL injection attack protections are not sufficient; something more is needed, and it's up to John to devise a possible solution.

The Problem

Debby, the lead developer, starts the discussions with a very small problem that quickly represents the crux of the issue without getting into too much detail. The bank website needs a username and password for customers to log in, which is stored in a single database table. To demonstrate, Debby creates that table and inserts a sample data for logging in as a user named DEBBY.

```
create table user_access
(
   username        varchar2(10) not null primary key,
   password        varchar2(10) not null
);
insert into user_access values ('DEBBY','DEBBYPASS');
commit;
```

It would then seem the code to select the password from the table uses a SQL query like the following:

```
select password
from user_access
where username = username_from_the_field_on_website
```

Being a savvy developer, Debby is well aware of the security issues that arise out of this bad coding. This would have required the user running the application to have select privilege on the USER_ACCESS table. But that privilege would have allowed the user to select from this table outside of the application and to see anyone's password—clearly not ideal. Therefore, she devises something different. She creates a function that accepts username and password as parameters and merely returns a decision about whether the password is correct or not. That way, the calling user from the website will not need to have select privilege on the USER_ACCESS table, yet the password is verified. Here is how the function looks:

```
create or replace function is_password_correct
(
   p_username in varchar2,
   p_password in varchar2
)
return varchar2
is
   l_stmt    varchar2(4000);
   l_check   varchar2(10) := 'wrong';
begin
   l_stmt :=
     'select ''correct'' from user_access '||
     'where username = '''||
     p_username ||''' and password = '''
     ||p_password||'''';
   dbms_output.put_line('l_stmt='||l_stmt);
   execute immediate l_stmt into l_check;
   return l_check;
end;
/
```

The website connects to the database using the user RUNUSER, which does not have privileges on the USER_ACCESS table. It does have execute privilege on the above function. So, from the website, Debby merely calls this function using the parameters the users entered on the form:

```
select is_password_correct('DEBBY','DEBBYPASS') from dual;
```

This returns "correct" if the user/password combination is correct; otherwise it returns null. If it's correct, the website proceeds further. Here is an example of what happens when the correct password is used:

```
SQL> select
  2     is_password_correct('DEBBY','DEBBYPASS')
  3        "Password Check"
  4  from dual
  5  /

Password Check
--------------
correct
```

```
SQL>
The function returned "correct," as expected. Now, here is an example of an incorrect
password:
SQL> select
  2     is_password_correct('DEBBY','WrongPass')
  3       "Password Check"
  4  from dual
  5  /

Password Check
--------------

SQL>
```

This returned NULL, as expected. So far so good, as far as checking the password is concerned. Now a hacker who does not know the password uses the following SQL. Debby points to the unusual string passed as a value to the password parameter of the function:

```
SQL> select
  2     is_password_correct('DEBBY','WrongPass'' or ''1''=''1')
  3       "Password Check"
  4  from dual
  5  /

Password Check
--------------
correct

SQL>
```

The function reported the password as correct, even though it was clearly entered incorrectly. The additional strings at the end did the trick. The dynamic SQL was transformed to something like this:

```
select 'correct' from user_access where username = 'DEBBY' and password = 'WrongPass'
or '1'='1'
```

The last predicate '1'= '1' did the trick, which was the OR condition. So, as long as that was satisfied, the query worked. This is a classic example of SQL injection attack, Debby explained. To counter the possibility of this attack, she made extensive checks in her code to find out if the user entered this pattern, e.g. '1'= '1'. However, the problem was, she realized—unfortunately much later—that the hacker didn't need to enter '1'= '1'; any value would have been sufficient, as long as they were the same. Here are some examples:

```
is_password_correct('DEBBY','WrongPass'' or ''2''=''2')
is_password_correct('DEBBY','WrongPass'' or ''apples''=''apples')
```

And, that's what happened; some hacker used a string that was not checked by the code, and the function allowed it to get in. But the problem is compounded, Debby noted, because this type of attack is not just for bypassing the user/password checks; it can be used anywhere there is a WHERE clause in the query selecting from a table. To remediate this she could write code to check for mismatched single quotes, but that would make the code voluminous and difficult to maintain. Besides, many junior developers probably won't be able to use those extensive checks, making the risk all the more apparent.

The question on everyone's mind is whether there is an approach to prevent hackers from entering this type of string to bypass checks. It has to be simple enough for everyone to use and implement immediately but effective enough to prevent injection attacks.

John assures the group that such an approach exists, and the mood in the room changes from despair to hope.

Bind Variables

The simplest solution, John explains, is to use a bind variable instead of appending values to the dynamically generated string to be executed. He rewrites the function as shown below. The changes are highlighted in bold.

```
create or replace function is_password_correct
(
   p_username in varchar2,
   p_password in varchar2
)
return varchar2
is
   l_stmt           varchar2(4000);
   l_check           varchar2(10) := 'wrong';
begin
   l_stmt :=
     'select ''correct'' from user_access '||
     'where username = '''||
     p_username ||''' and password = :l_password';
   dbms_output.put_line('l_stmt='||l_stmt);
   execute immediate l_stmt into l_check
         using p_password;
   return l_check;
end;
/
```

Now when a hacker enters the string, he will not get past the check:

```
SQL> select
  2     is_password_correct('DEBBY','WrongPass'' or ''1''=''1')
  3  from dual
  4  /

IS_PASSWORD_CORRECT('DEBBY','WRONGPASS''OR''1''=''1')
-----------------------------------------------------
l_stmt=select 'correct' from user_access where username = 'DEBBY' and password =
:l_password

SQL>
```

The function accurately deflected the SQL injection attack. John made it clear that the first defense against a SQL injection attack is to change all the dynamically constructed strings replacing placeholders with bind variables. Another bonus is the lack of repeated parsing of the SQL statements and the resultant performance gains.

While others beam at this suggestion, Debby is less than thrilled. First, she explains, this requires a lot of changes everywhere, making it expensive, prone to bugs, and time consuming. Second, she continues, it solves this simple case, but what about the cases where the string simply has to be dynamically constructed by appending? For instance, in some cases the columns, the table name, and other cases would not be known in advance, requiring them to be constructed at runtime only. Bind variables won't help in those cases. There are possibilities of using ref cursors, but they need change—sometimes drastic—in the code, a possibility Debby wasn't happy to consider. They all want to know if there are any other approaches.

Assertion

John assures them that there are other solutions. The risk of SQL injection attacks is so rampant, he explains, that Oracle actually built a tool into the database in the form of a package called DBMS_ASSERT. This tool has been available since Oracle

10g, but it is so little known that almost no one uses it, and Debby and her team are no exception. The tool can make the life of developers much easier by making available not only a simple check for these injected strings but by cleaning them as well. John starts off by showing what a specific function enquote_literal() in this package can do. This function expects a string and returns the same string by stripping the quotes and returning the string enclosed within quotes to be used as an input to another code. Here is an example from John:

```
SQL> select dbms_assert.enquote_literal('WrongPass') from dual;

DBMS_ASSERT.ENQUOTE_LITERAL('WRONGPASS')
----------------------------------------
'WrongPass'

SQL>
```

The function merely returns the same string. Now he passes the same injection attack string the hacker used:

```
SQL> select dbms_assert.enquote_literal('WrongPass'' or ''1''=''1') from dual;
```

The function produces an error:

```
select dbms_assert.enquote_literal('WrongPass'' or ''1''=''1') from dual
       *
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at "SYS.DBMS_ASSERT", line 409
ORA-06512: at "SYS.DBMS_ASSERT", line 493
```

The query failed with a PL/SQL value error. Why? The hacker gave a string as an input; so, as a string the input was perfectly fine. What caused the error? The problem was that the quotes were not matched up, which indicated that the literal was not correct. The tool pointed it out without any kind of complex check.

One more important property of the Assertion tool, John points out, is that the return value is already a quoted string.

Using that tool, John rewrites the original function as follows. The changes are highlighted in bold.

```
create or replace function is_password_correct
(
   p_username in varchar2,
   p_password in varchar2
)
return varchar2
is
   l_stmt           varchar2(4000);
   l_check          varchar2(10) := 'wrong';
begin
   l_stmt :=
     'select ''correct'' from user_access '||
     'where username = '''||
     p_username ||''' and password = '||
     sys.dbms_assert.enquote_literal(p_password)
     ||'';
   dbms_output.put_line('l_stmt='||l_stmt);
   execute immediate l_stmt into l_check;
   return nvl(l_check,'wrong');
end;
/
```

Drawing everyone's attention to this code, John points to how he used the function enquote_literal() in the DBMS_ASSERT package on the value p_password to get a clean value. After this function is created, when the user passes

```
select
   is_password_correct('DEBBY','WrongPass'' or ''1''=''1')
from dual;

The output comes back as
ERROR at line 2:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at "SYS.DBMS_ASSERT", line 409
ORA-06512: at "SYS.DBMS_ASSERT", line 493
ORA-06512: at "ACME.IS_PASSWORD_CORRECT", line 11
```

The error shows very clearly that the value passed triggered a PL/SQL value error. But when a good value is passed, the error doesn't show up, as shown below:

```
select is_password_correct('DEBBY','DEBBYPASS') from dual;
select is_password_correct('DEBBY','WrongPass') from dual;
```

In the first case, the password is the correct one; so the result of the query is "correct"; in the second case, the password is wrong; so the result is null. But in both cases above, the query didn't fail with the numeric error, since the input was not injected.

Debby's face is clearly shining now, but there is still a sliver of dark shadow. She points out that the error message isn't very intuitive. PL/SQL numeric or value error is not necessarily the indication of SQL injection attacks. She wants to know if there is any way to warn the user clearly of the possible attack.

There is, answers John. Since the error occurs in the call to the dbms_assert package, John can easily check right afterwards. So, he creates a separate block inside the main code to check for the input string and uses a named exception:

```
create or replace function is_password_correct
(
   p_username in varchar2,
   p_password in varchar2
)
return varchar2
is
   l_stmt              varchar2(4000);
   l_sanitized_password varchar2(4000);
   l_check             varchar2(10) := 'wrong';
   l_possible_injection_exception  exception;

begin
   declare
      l_possible_injection_exception  exception;
      pragma exception_init (l_possible_injection_exception,-6502);
   begin
      l_sanitized_password := sys.dbms_assert.enquote_literal(p_password);
   exception
      when l_possible_injection_exception then
         raise_application_error (-20001,'Possible SQL Injection Attack');
      when OTHERS then
         raise;
   end;

   l_stmt :=
     'select ''correct'' from user_access '||
     'where username = '''||
     p_username ||''' and password = '||
     l_sanitized_password
     ||'';
   dbms_output.put_line('l_stmt='||l_stmt);
   execute immediate l_stmt into l_check;
   return nvl(l_check,'wrong');
end;
/
```

With this new code, this is the result when Debby calls the function with the same injection string:

```
SQL> select
  2     is_password_correct('DEBBY','WrongPass'' or ''1''=''1')
  3  from dual
  4  /
   is_password_correct('DEBBY','WrongPass'' or ''1''=''1')
   *
ERROR at line 2:
ORA-20001: Possible SQL Injection Attack
ORA-06512: at "ACME.IS_PASSWORD_CORRECT", line 21
```

This time the error is very clear—"Possible SQL Injection Attack"—which is what Debby wanted. All she had to do, John explains, is to add the dbms_assert.enquote_literal() function to check and get the sanitized input value wherever the input is used in a predicate (the WHERE condition) in a query. Since the function merely returns the same input value if it is not polluted by any injection attacks, Debby can use it for any input values, whether there is risk of injection attacks or not, which eliminates the possibility of the attack. She beams with happiness. The relieved group leaves John's office with a solution and an action plan.

More Uses

John demonstrated only one use case of the DBMS_ASSERT package, albeit the most useful one. There are many more functions in that package. For instance, the function simple_sql_name() checks if the input is a valid SQL identifier; otherwise, it fails with an error ORA-44003.

```
SQL> select dbms_assert.simple_sql_name('ACME1,a') from dual;
select dbms_assert.simple_sql_name('ACME1,a') from dual
       *
ERROR at line 1:
ORA-44003: invalid SQL name
ORA-06512: at "SYS.DBMS_ASSERT", line 206
```

Due to lack of time, John couldn't explain all the features of the package, so he asked the group to educate themselves on the remainder.

Conclusion

SQL injection is one of the most common database security issues, probably right after insider attacks. In this article we described a simple but highly effective way to prevent SQL injection attacks in codes using a tool called DBMS_ASSERT that has been available in Oracle for quite a long time. The function enquote_literal() in this package can take any literal string and check for possible injection attacks in the string, leaving behind a clean, sanitized string that is safe to be used in queries without risk of SQL injection. For more information on using this package, refer to Oracle documents on this topic at http://docs.oracle.com/database/121/ARPLS/d_assert.htm#ARPLS231. ▲

Arup Nanda has been an Oracle DBA for over 20 years, with experience spanning all aspects from modeling to performance tuning and Exadata. He speaks frequently, has authored about 300 articles, co-authored five books, blogs at arup.blogspot.com, and mentors new and seasoned DBAs. He was named Oracle's DBA of the Year in 2003 and Enterprise Architect of the Year in 2012, and is an ACE Director and a member of the Oak Table Network.

New Partitioned Index Features in Oracle Database 12c

Darl Kuhn

by Darl Kuhn

Oracle Database 12*c* includes several new features related to indexes and partitioning. This article explores the topics of partial indexes and asynchronous global index maintenance. These new features have a positive impact on both performance and availability. We'll first explore the subject of partial indexes.

Partial Indexes

Starting with Oracle Database 12*c*, you can create either local or global indexes on a subset of partitions in a table. You may want to do this if you've pre-created partitions and don't yet have data for range partitions that map to future dates—the idea being that you'll build the index after the partitions have been loaded (at some future date).

You set up the use of a partial index by first specifying INDEXING ON|OFF for each partition in the table. In this next example, PART_1 has indexing turned on and PART_2 has indexing turned off:

```
CREATE TABLE p_table (a int)
   PARTITION BY RANGE (a)
   (PARTITION part_1 VALUES LESS THAN(1000) INDEXING ON,
   PARTITION part_2 VALUES LESS THAN(2000) INDEXING OFF);

Table created.
```

Next, a partial local index is created:

```
create index pi1 on p_table(a) local indexing partial;

Index created.
```

In this scenario, the INDEXING PARTIAL clause instructs Oracle to build and make usable only local index partitions that point to partitions in the table that were defined with INDEXING ON. In this case, one usable index partition will be created with index entries pointing to data in the PART_1 table partition:

```
col index_name form a25
col partition_name form a25
col status form a15

select a.index_name, a.partition_name, a.status
   from user_ind_partitions a, user_indexes b
   where b.table_name = 'P_TABLE'
   and a.index_name = b.index_name;

INDEX_NAME                PARTITION_NAME            STATUS
------------------------- ------------------------- ---------------
PI1                       PART_1                    USABLE
PI1                       PART_2                    UNUSABLE
```

Next we'll insert some test data, generate statistics, set autotrace on, and run a query that should locate data in the PART_1 partition:

```
insert into p_table select rownum from dual connect by level < 2000;

1999 rows created.

exec dbms_stats.gather_table_stats(user,'P_TABLE');

PL/SQL procedure successfully completed.

explain plan for select * from p_table where a = 20;

Explained.

select * from table(dbms_xplan.display(null,null,'BASIC +PARTITION'));
-----------------------------------------------------
| Id  | Operation              | Name | Pstart| Pstop |
-----------------------------------------------------
|   0 | SELECT STATEMENT       |      |       |       |
|   1 |  PARTITION RANGE SINGLE|      |     1 |     1 |
|   2 |   INDEX RANGE SCAN     | PI1  |     1 |     1 |
-----------------------------------------------------
```

As expected, the optimizer was able to generate an execution plan utilizing the index. Next, a query is issued that selects data from the partition defined with INDEXING OFF:

```
explain plan for select * from p_table where a = 1500;

Explained.

select * from table(dbms_xplan.display(null,null,'BASIC +PARTITION'));

-----------------------------------------------------
| Id  | Operation              | Name    | Pstart| Pstop |
-----------------------------------------------------
|   0 | SELECT STATEMENT       |         |       |       |
|   1 |  PARTITION RANGE SINGLE|         |     2 |     2 |
|   2 |   TABLE ACCESS FULL    | P_TABLE |     2 |     2 |
-----------------------------------------------------
```

The output shows that a full table scan of PART_2 was required, as there is no usable index with entries pointing at data in PART_2. We can instruct Oracle to create index entries pointing to data in PART_2 by rebuilding the index partition associated with the PART_2 partition:

```
alter index pi1 rebuild partition part_2;

Index altered.
```

Rerunning the previous select query shows that the optimizer is now utilizing the local partitioned index pointing to the PART_2 table partition:

```
-----------------------------------------------------
| Id  | Operation             | Name | Pstart| Pstop |
-----------------------------------------------------
|   0 | SELECT STATEMENT      |      |       |       |
|   1 |  PARTITION RANGE SINGLE|     |     2 |     2 |
|   2 |   INDEX RANGE SCAN    | PI1  |     2 |     2 |
-----------------------------------------------------
```

In this way, partial indexes allow you to disable the index while the table partition is being loaded (increasing the loading speed), and then later you can rebuild the partial index to make it available.

Asynchronous Global Index Maintenance

Recall that starting with Oracle 9*i* and higher you can maintain global indexes while dropping or truncating partitions via the UPDATE GLOBAL INDEXES clause. However, such operations come at a cost in terms of time and resource consumption.

Starting with Oracle Database 12*c*, when dropping or truncating table partitions, Oracle postpones the removal of the global index entries associated with the dropped or truncated partitions. This is known as *asynchronous global index maintenance*. Oracle postpones the maintenance of the global index to a future time while keeping the global index usable. The idea is that this improves the performance of dropping/truncating partitions while keeping any global indexes in a usable state. The actual cleanup of the index entries is done later (asynchronously), either by the DBA or by an automatically scheduled Oracle job. The point is not that less work is being done; rather, it's that the cleanup of index entries is decoupled from the DROP/TRUNCATE statement.

A small example will demonstrate asynchronous global index maintenance. To set this up, we create a table in an 11*g* database, populate it with test data, and create a global index:

```
CREATE TABLE partitioned
     ( timestamp date,
       id        int
     )
     PARTITION BY RANGE (timestamp)
     (PARTITION fy_2014 VALUES LESS THAN
     (to_date('01-jan-2015','dd-mon-yyyy')),
     PARTITION fy_2015 VALUES LESS THAN
     ( to_date('01-jan-2016','dd-mon-yyyy')));

insert into partitioned partition(fy_2014)
     select to_date('31-dec-2014','dd-mon-yyyy')-mod(rownum,364), rownum
     from dual connect by level < 100000;

99999 rows created.

insert into partitioned partition(fy_2015)
    select to_date('31-dec-2015','dd-mon-yyyy')-mod(rownum,364), rownum
    from dual connect by level < 100000;

99999 rows created.

create index partitioned_idx_global
    on partitioned(timestamp)
    GLOBAL;

Index created.
```

Next we'll run a query to retrieve the current values of redo size and db block gets statistics for the current session:

```
col r1 new_value r2
col b1 new_value b2

select * from
   (select b.value r1
    from v$statname a, v$mystat b
    where a.statistic# = b.statistic#
    and a.name = 'redo size'),
   (select b.value b1
    from v$statname a, v$mystat b
    where a.statistic# = b.statistic#
    and a.name = 'db block gets');

        R1         B1
---------- ----------
   4816712       4512
```

Then a partition is dropped with the UPDATE GLOBAL INDEXES clause specified:

```
alter table partitioned drop partition fy_2014 update global indexes;

Table altered.
```

Now we'll calculate the amount of redo generated and the number of current blocks accessed:

```
select * from
   (select b.value - &r2 redo_gen
    from v$statname a, v$mystat b
    where a.statistic# = b.statistic#
    and a.name = 'redo size'),
   (select b.value - &b2 db_block_gets
    from v$statname a, v$mystat b
    where a.statistic# = b.statistic#
    and a.name = 'db block gets');

old   2: (select b.value - &r2 redo_gen
new   2: (select b.value -    4816712 redo_gen
old   6: (select b.value - &b2 db_block_gets
new   6: (select b.value -       4512 db_block_gets

  REDO_GEN DB_BLOCK_GETS
---------- -------------
   2459820          1495
```

If we rerun the same code in an Oracle Database 12*c* database, we get the following when dropping the partition with UPDATE GLOBAL INDEXES specified:

```
REDO_GEN DB_BLOCK_GETS
---------- -------------
     9872           43
```

Compared to the 11*g* example, a fraction of the redo is generated and fewer blocks are accessed when running this in an Oracle Database 12*c* database. The reason behind this is that Oracle doesn't immediately perform the index maintenance of removing the index entries from the dropped partition. Rather, these entries are marked as *orphaned* and will later be cleaned up by Oracle. The existence of orphaned entries can be verified via the following:

```
select index_name, orphaned_entries, status from user_indexes
   where table_name='PARTITIONED';

INDEX_NAME               ORP STATUS
------------------------ --- --------
PARTITIONED_IDX_GLOBAL   YES VALID
```

How do the orphaned entries get cleaned up? Oracle Database 12*c* has an automatically scheduled PMO_DEFERRED_GIDX_MAINT_JOB, which runs in a nightly maintenance window. If you don't want to wait for that job, you can manually clean up the entries yourself:

```
exec dbms_part.cleanup_gidx;
PL/SQL procedure successfully completed.
```

We can now verify that the orphaned entries have been cleaned up:

```
select index_name, orphaned_entries, status from user_indexes
   where table_name='PARTITIONED';

INDEX_NAME                ORP STATUS
------------------------- --- --------------
PARTITIONED_IDX_GLOBAL    NO  VALID
```

In this way you can perform operations such as dropping and truncating partitions and still leave your global indexes in a usable state without the immediate overhead of cleaning up the index entries as part of the drop/truncate operation.

See MOS note 1482264.1 for further details on asynchronous global index maintenance.

Summary

Oracle Database 12*c* contains a large number of new features. This article explored two features that impact indexes in a partitioned environment. Partial indexes allow you to build a subset of index partitions. You may want to do this if your table doesn't initially contain data for all partitions in the table. Also, we investigated asynchronous global index maintenance. This feature allows you to decouple the maintenance of a global index from operations such as dropping or truncating table partitions. This allows you to perform the global index maintenance when it's more convenient.

Acknowledgements

This article was taken from material contained in *Expert Oracle Database Architecture*, 3rd edition, Apress, authored by Tom Kyte and Darl Kuhn. ▲

Darl Kuhn is currently a DBA working for Oracle Corporation. He has written books on a variety of IT topics, including SQL, Performance Tuning, Oracle Internals, Linux, Backup and Recovery, RMAN, and Database Administration. Darl's most recent book is Linux and Solaris Recipes for Oracle DBAs, published by Apress.

PL/SQL VARCHAR2 Memory Allocation

by Michael Rosenblum

Michael Rosenblum

The biggest challenge of explaining efficient ways of using the VARCHAR2 datatype is the fact that its PL/SQL implementation is often confused with its table storage implementation. As a quick reminder, VARCHAR2 is named for the VARiable length of CHARacters, which implies dynamic allocation of resources, i.e. storage/memory. This mechanism is usually efficient from the perspective of persistent tables because it saves a significant amount of space. But from a memory allocation perspective, it is a mixed blessing because reallocation of memory comes with a price.

For years, Oracle did something unexpected (and rarely noticed!): in some cases, the PL/SQL engine decided to allocate the maximum length of memory to the variable, regardless of the amount of data being stored.

In Oracle Database 10g, the mechanism is simple. Up to and including VARCHAR2(1999), the memory is fully allocated, while above this length, it becomes dynamic. There is even explicit mention of this in the documentation: **http://tinyurl.com/Varchar2Limit10g**. However, from Oracle Database 11*g* onward, the documentation does not mention this topic, so some testing was required. To find out the precise details, a special measuring mechanism is needed (thanks to Tim Hall for this concept):

```
CREATE OR REPLACE FUNCTION f_getPGA_nr RETURN NUMBER AS
v_nr NUMBER;
BEGIN
SELECT b.value
INTO v_nr
FROM v$statname a, v$mystat b
WHERE a.statistic# = b.statistic#
AND a.name = 'session pga memory';
RETURN v_nr;
END;
/
CREATE OR REPLACE PROCEDURE p_testVarchar2 (pi_length_nr NUMBER) IS
BEGIN
EXECUTE IMMEDIATE
'DECLARE'||chr(10)||
'    v_before_nr NUMBER;'||CHR(10)||
'    v_after_nr NUMBER;'||CHR(10)||
'    v_level_nr NUMBER:=0;'||CHR(10)||
'    PROCEDURE p_DrillDown(pi_tx VARCHAR2) IS'||CHR(10)||
'        v_tx VARCHAR2('||pi_length_nr||'):=pi_tx;'||CHR(10)||
'    BEGIN'||chr(10)||
'        v_level_nr:=v_level_nr+1;'||CHR(10)||
'        IF v_level_nr <=1000 THEN'||CHR(10)||
'            p_DrillDown(pi_tx);'||CHR(10)||
'        END IF;'||CHR(10)||
'    END;'||CHR(10)||
'BEGIN'||CHR(10)||
'    v_before_nr:=f_getPGA_nr;'||CHR(10)||
'    p_DrillDown(''A'');'||CHR(10)||
'    v_after_nr:=f_getPGA_nr;'||CHR(10)||
'    dbms_output.put_line(rpad('''||pi_length_nr||':'',6,'' '')
      ||(v_after_nr-v_before_nr));'||CHR(10)||
     'END;';
END;
/
```

The procedure P_TESTVARCHAR2 generates an anonymous PL/SQL block that recursively calls the procedure P_DRILLDOWN 1000 times. This means that inside of the procedure P_DRILLDOWN, a local variable V_TX is initialized each time. The length of this variable is defined by the input parameter of P_TESTVARCHAR2, but the length of data to be stored is 1 (single letter 'A'). To measure the memory allocation, the anonymous block captures PGA statistics from V$MYSTAT. The following is the result of tests run in Oracle Database 11*g* R2 (11.2.0.4 64-bit on MS Windows 2008R2 + CPUApr2016) and in Oracle Database 12c R1 (12.1.0.2 64-bit on MS Windows 2008R2 + CPUApr2016):

Here are the results for Oracle Database 11*g*.

```
SQL> connect misha/MISHA@localDB
SQL> set serveroutput on
SQL> exec p_TestVarchar2(1);
1    :458752
...
100  :589824
...
1000 :1900544
...
2000 :3145728
...
3000 :4653056
...
4000 :9371648
...
4001 :524288
...
32767:524288
```

And here are the results for Oracle Database 12*c*.

```
 SQL> connect misha/MISHA@PDB
SQL> set serveroutput on
SQL> exec p_TestVarchar2(1);
1    :458752
...
100  :589824
...
```

```
1000 :1835008
...
2000 :3080192
...
3000 :4653056
...
4000 :9306112
...
4001 :589824
...
32767:589824
```

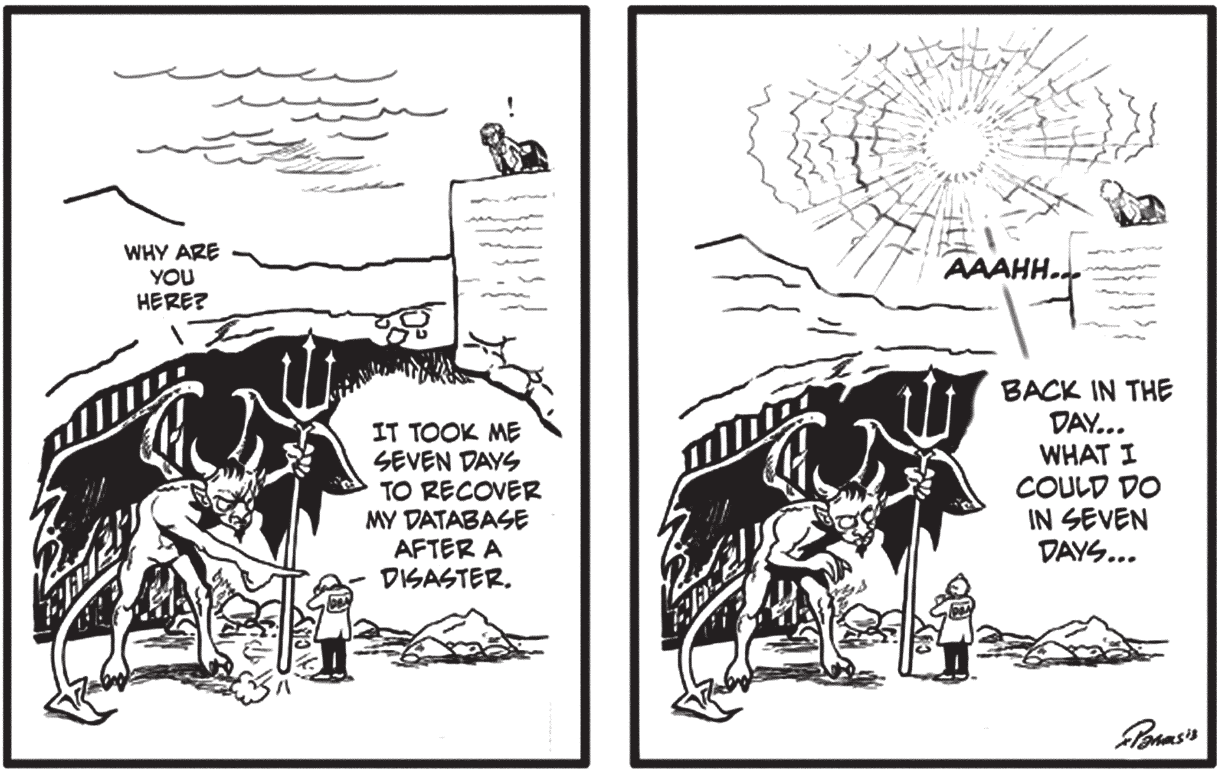
Obviously, the same implementation difference detected in Oracle Database 10*g* exists in higher versions as well, in both Oracle Database 11*g* and 12*c*. However, the threshold has been moved from 1999 to 4000 characters. If you are using a lot of VARCHAR2 variables that are being initialized at the same time, the impact on overall PGA usage can be significant. In the previous test, for 1000 variables, if you change VARCHAR2(4000) to VARCHAR2(4001), the savings will be twentyfold.

From a practical standpoint, my suggestions can be summarized into the following set of rules:

- Declaring all of your string variables VARCHAR2(4000) is a bad idea.
- If you know (more or less) how much text you need to manipulate, you should declare the variable using that expected length plus a safety margin (if needed).
- If you don't have a good idea of the necessary text length and the variable in question will be used only in the context of PL/SQL, the easiest option is to declare it as VARCHAR2(32767). Don't be afraid of the big number! This way, you can store as much information as you like without worrying about either memory or length restrictions.
- If you cannot correctly guesstimate the text length (at least in the range of plus/minus a few hundred), but it has to be used in the context of SQL, the solution is a bit more complicated. Since SQL does not support VARCHAR2 above 4000 (unless you explicitly enable it in Oracle Database 12*c*), having variables that can potentially hold longer text may cause problems, although you can get better length precision by using the TABLE.COLUMN%TYPE-declaration mechanism. ▲

Michael Rosenblum is a Software Architect/Senior DBA at Dulcian, Inc. where he is responsible for system tuning and application architecture. Michael supports Dulcian developers by writing complex PL/SQL routines and researching new features. He is the co-author of PL/SQL for Dummies (Wiley Press, 2006), Oracle PL/SQL Performance Tuning Tips and Techniques (Rosenblum & Dorsey, Oracle Press, 2014), contributing author of Expert PL/SQL Practices (APress, 2011), and many database-related articles and conference papers. Michael is an Oracle ACE, and frequent presenter at conferences (Oracle OpenWorld, ODTUG, IOUG Collaborate, RMOUG, NYOUG) and winner of the ODTUG Kaleidoscope 2009 Best Speaker Award.

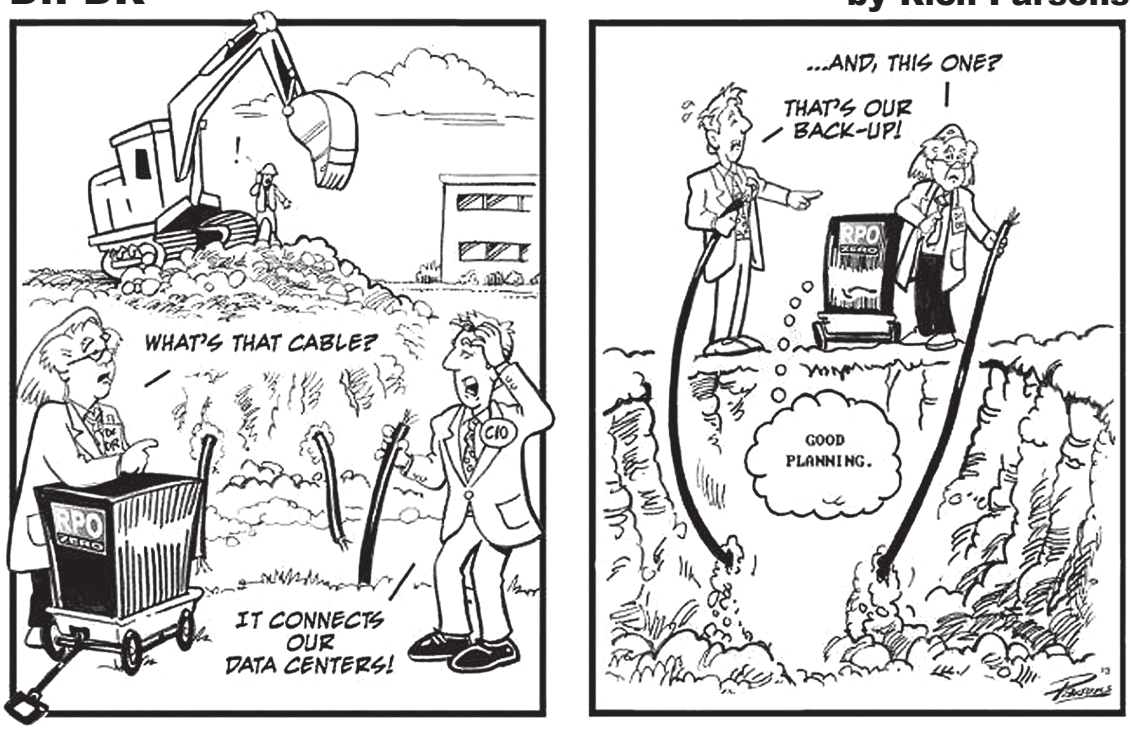
Dr. DR **by Rich Parsons**



Dr. DR is brought to you by Axxana.

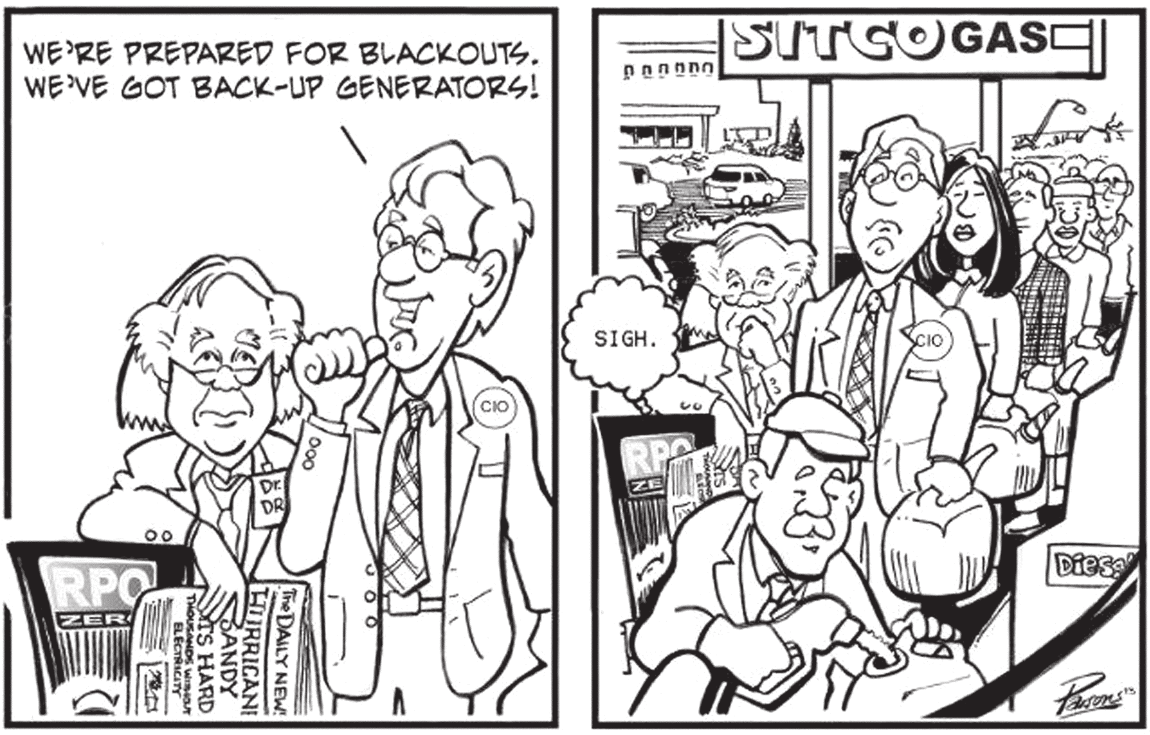
Dr. DR

by Rich Parsons



Dr. DR

By Rich Parsons



Dr. DR is brought to you by Axxana.

NoCOUG Quarterly Picnic

Mission Peak Regional Preserve

Many Thanks to Our Sponsors

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

Long-term event sponsorship:

Chevron

Oracle Corp.

Thank you! Gold Vendors:

- Axxana
- Cohesity
- Database Specialists
- Dell Software
- Delphix
- EMC
- HGST

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

TREASURER'S REPORT

Sri Rajan, Treasurer

Beginning Balance		
January 1, 2016		**$ 50,271.60**
Revenue		
Corporate Membership	7,700.00	
Individual Membership	6,790.00	
Conference Walk-in Fees	1,287.00	
Training Day Fees	3,530.00	
Gold Vendor Fees	4,000.00	
Silver Vendor Fees	500.00	
Conference Sponsorships	2,000.00	
Journal Advertising	1,000.00	
Charitable Contributions	-	
Interest	1.41	
Cash Back	115.62	
Total Revenue		**$ 26,924.03**
Expenses		
Conference Expenses	12,872.14	
Journal Expenses	4,300.37	
Training Day Expenses	1,636.85	
Board Expenses	568.94	
PayPal Expenses	773.60	
Software Dues	137.95	
Insurance	-	
Office Expenses	70.00	
Meetup Expenses	-	
Taxes and Filings	-	
Marketing Expenses	-	
Total Expenses		**$ 20,359.85**
Ending Balance		
March 31, 2016		**$ 56,835.78**

Why You Should Write

by Jonathan Gennick

Jonathan Gennick

One of the best things you can do for your career is to write about whatever it is that you're doing or learning each day. Write to share knowledge and help others come up after you. Doing so helps put your own knowledge and skills on a better foundation. As well, you begin to develop a reputation and build a network of people who know what you are good at and can be called on to do.

Bill Zinser's 1993 classic, Writing to Learn, makes my first point in the very title words of his book. My first-ever authoring experience was to write five chapters in a book on PL/SQL. My five assigned topics were on the simpler-end of the spectrum, yet I learned a great deal as I was forced by writing on those topics to explore fully all the corner cases. As well, the effort to present clearly for others to read and understand helped to reorganize the content in my own mind so that in the end I came away with a much better grasp of the material. I wrote, and I learned.

Creating a resume was the thing to do when I graduated college in the mid-1980s. I'd still create one today, but I'd go one better and create a blog and look for opportunities to contribute to journals and become active in user groups of like-minded professionals. Look at almost any well-known name in our field —Cary Millsap, Arup Nanda, Steve Feuerstein, Jonathan Lewis —these people all have one thing in common: They write. They write on their blog, and in journals, and all sorts of opportunities flow their direction as a result.

Is it possible that the big names are big names because of their writing, and not the other way 'round? I wouldn't quite argue it that way, but I'm sure none of the people whom I've named would be where they are today had they not been putting themselves out there through their writing. Talk to them, and I bet you'll find that speaking opportunities and better and more interesting work assignments came about in part because of their efforts in putting themselves out there in the written word.

What to write about? My advice is to write about what you're doing every day. Did you solve an interesting problem at work? Make a short case study. Write up a one-page description. Because any problem you've solved, someone else somewhere is likely also looking to solve it.

Because anything can be made interesting if looked at in the right light. Just this week I went live with a post on adjusting bicycle brake reach. You can read it at **https://www.loosescrews.com/brake-reach/**. Every bike mechanic knows about what I wrote about, but I guarantee there are people with bicycles for whom the knowledge is brand new.

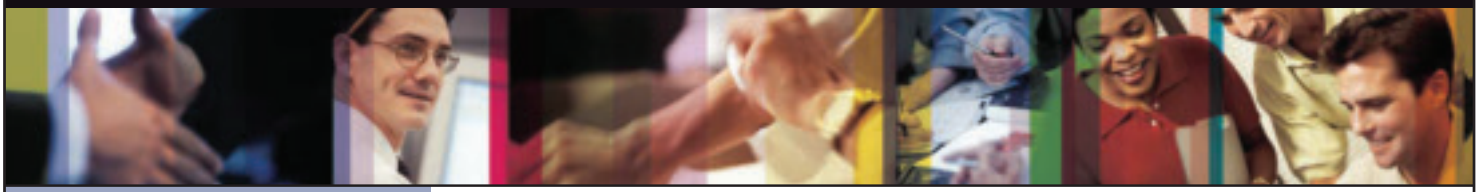
Expertise is like a pyramid. The number of entry-level people in any field far outnumbers those at the apex. Tim Ford recognizes this in his Entry-Level Content Challenge blog post from earlier this year. Find it at **http://thesqlagentman.com/2016/01/entry-level-content/**. Also read Steve Hood's response at **https://simplesqlserver.com/2016/01/26/tim-fords-entry-level-content-challenge/**. Steve's advice is good: "Talk about subjects you know well, even if you're not at the level you feel comfortable teaching advanced topics."

Writing can open doors and lead to incredible opportunities. Begin with a blog. That's a safe and easy beginning. But don't stop there. Aim higher. User group journals such as the one you're reading now offer tremendous opportunity to become known and make a difference in your field. Take advantage. You'll be helping your journal editors and user group peers, and I promise some benefits will flow your direction too.

Jonathan Gennick is a book editor with Apress, and a former Oracle Database practitioner. Inspired by Tim Ford's Entry-Level Content Challenge, Jonathan is aiming to write one entry-level post per month on his blog. Find his entries to-date at http://gennick.com/database/?tag=%23EntryLevel.

One of the best things you can do for your career is to write about whatever it is that you're doing or learning each day. Write to share knowledge and help others come up after you. Doing so helps put your own knowledge and skills on a better foundation. As well, you begin to develop a reputation and build a network of people who know what you are good at and can be called on to do.

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