



Official Publication of the Northern California Oracle Users Group

# NoCOUG

J O U R N A L

Vol. 30, No. 2 • MAY 2016

\$15

## Not Only SQL?

### **Making SQL Great Again**

*A Panel Discussion.  
See page 4.*

### **Three Database Revolutions**

*A Book Excerpt.  
See page 13.*

### **Mission Possible**

*No Data Loss Without a  
Synchronous Network.  
See page 21.*

***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.



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 118<sup>th</sup> issue of the *NoCOUG Journal*. Enjoy! ▲

—NoCOUG Journal Editor

## Table of Contents

### ADVERTISERS

Interview.....	4	Dell Software .....	2
Book Notes .....	8	HGST.....	7
Book Excerpt.....	13	OraPub .....	11
Special Feature.....	21	Axxana .....	10
Treasurer's Report .....	25	Delphix .....	20
		Database Specialists .....	27
		SolarWinds .....	28

### Publication Notices and Submission Format

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

Please send your questions, feedback, and submissions to the *NoCOUG Journal* editor at [journal@nocoug.org](mailto:journal@nocoug.org).

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

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

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

## 2016 NoCOUG Board

### President

Iggy Fernandez

### President Emeritus

Hanan Hit

### President Emeritus

Naren Nagtode

### Vice President

Jeff Mahe

### Secretary/Treasurer

Sri Rajan

### Membership Director

Stephen Van Linge (Board Associate)

### Conference Director

Sai Devabhaktuni

### Vendor Coordinator

Omar Anwar

### Webmaster

Jimmy Brock

### Journal Editor

Iggy Fernandez

### IOUG Liaison

Kyle Hailey (Board Associate)

### Training Director

Tu Le

### Social Media Director

Vacant Position

### Marketing Director

Vacant Position

### Members at Large

Eric Hutchinson

Kamran Rassouli

Linda Yang

### Board Advisor

Tim Gorman

### Book Reviewer

Brian Hitchcock

## ADVERTISING RATES

The *NoCOUG Journal* is published quarterly.

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

Personnel recruitment ads are not accepted.

[journal@nocoug.org](mailto:journal@nocoug.org)

# Making SQL Great Again

*Editor's Note: Excerpts from a panel discussion titled "Making SQL Great Again (SQL is HUUUUUGE)" at YesSQL Summit 2016 organized by the Northern California Oracle Users Group (NoCOUG) at Oracle Corporation's headquarters in Redwood City, California. The complete video of the panel discussion has been published by Oracle Corporation on the Oracle Channel on YouTube. The panelists were Andrew (Andy) Mendelsohn (Executive Vice-President, Database Server Technologies, Oracle), Graham Wood (Architect, Oracle), Bryn Llewellyn (Distinguished Product Manager, Oracle), Hermann Baer (Senior Director, Product Manager, Oracle), Steven Feuerstein (Architect, Oracle). The moderator was Kyle Hailey, an Oracle ACE Director and member of the OakTable Network.*

**Why are we even having this discussion? Why is it necessary to defend SQL? Are NoSQL and Hadoop temporary phenomena that will eventually fade away just like object-oriented database management systems?**

**Andy Mendelsohn:** The first thing to understand is SQL does not go back to the beginning of time. Before the SQL databases came out in the '80s, there were "NoSQL" databases. OK, so people think that NoSQL is something new. NoSQL is actually prehistoric—it goes back to the beginning of time of information management. When the first computer was written, somebody came up with this idea: let's have an index—a B-tree, basically—and let's have an API to it, and those were the original key-value stores. There were things like—on the mainframe—there's ISAM and VSAM. Informix eventually had this product called C-ISAM. Berkeley DB came out. So the idea of a key-value store is ancient. And relational databases were basically created in the '80s because developers want to write reports and get information out of their databases. And key-value stores and other related non-SQL databases were terribly unproductive in writing reports. You'd take weeks to write a report to get a simple sorted list of information out of a NoSQL database, and that's where SQL came from. SQL was a huge leap forward in programmer productivity. Couple lines of SQL was equivalent to ten pages of writing code against a NoSQL-style API, key-value-store-style API. So NoSQL has been around before SQL and will be around. And SQL and NoSQL systems have co-existed for 30 years now. And so this whole question is sort of only relevant only for people who don't know their history—which is all the young developers out there, of course, who know nothing about what happened more than ten years ago, right? So, of course, NoSQL systems overall, they're very good at what they do—very simple applications that don't require the equivalent of writing reports

and joins and ORDER BYs and all that kind of stuff you do trivially in SQL. NoSQL is great at that. There is a use case for it and there always will be one. The more modern NoSQL systems like the MongoDBs, and Cassandras, and all those guys—and actually Amazon DynamoDB is probably the right one to actually look at. All Amazon did is, they were using Berkeley DB in their e-commerce system and they said, "You know, wouldn't it be great if we had N of these B-trees, not just one, and let's add a hash distribution layer in front of Berkeley DB," and that's where DynamoDB came from. And that's the direct ancestor of all the more recent NoSQL systems, so it's basically just a hashing layer on top of a B-tree. This is trivial technology, and that's why there are about 40 or 50 companies that have NoSQL databases—including Oracle—and, again, there's a great use case for these kinds of products. They do really simple applications really well; they scale nicely. And if you have an application that fits that mold and that's all you need, it's great. SQL systems obviously are much more flexible. They do everything NoSQL systems do, plus they make developers really productive doing really complex applications. If you look at our Fusion applications or SAP applications—real commercial business applications—if you want to write those in a NoSQL product, you'd need thousands of times the number of developers to write the same code as you would with SQL. And then the size of the market really shows you the difference between the use cases of the two products. Relational databases are over a thirty-billion-dollar market. NoSQL databases are one of these classic zero-billion dollar markets—maybe there's a few hundred million there, but basically they're free products; there's not a lot of value there. MongoDB: huge downloads. Everybody's playing with it, but almost nobody pays for it; there's not really a good business model.

The business model is actually another interesting aspect of the modern NoSQL databases. They have the freemium model where you can download it for free, you play with it, and then if you want support—a few people will pay for support—you pay by subscription model. Traditionally, relational databases on premise have the licensing model where you pay up front and then you pay for support, and now that everybody is moving to cloud, that model was now becoming available for all the relational databases as well. So the business model is also something that I think was interesting around NoSQL databases and is now becoming pretty prevalent as everybody makes their software available on clouds using the same kind of subscription model.

Hadoop is a whole different beast. I don't know if we should leave that for another question, but as far as I'm concerned, Hadoop is just becoming a relational database. The whole idea

***"[Amazon was] using Berkeley DB in their e-commerce system and they said, 'You know, wouldn't it be great if we had N of these B-trees, not just one, and let's add a hash distribution layer in front of Berkeley DB.'"***

that MapReduce was interesting was exciting a few years ago and everybody has pretty much decided, you know, “MapReduce is not interesting; we’re all going to offer SQL on top of HDFS.” And there’s 20 different SQL systems. What’s really happening, Hadoop is becoming a data warehouse/relational database—and beyond that it’s not clear that there’s much other use of Hadoop. But Hadoop is here to stay; HDFS is a nice distributed filesystem; it’s getting embedded in all the offerings from all the vendors, including Oracle—we have it on our Big Data appliance; we have it up on our cloud now; it’s a nice technology. Filesystems go back 40 years—Hadoop is just the latest generation: a cool filesystem technology that certainly will survive, and everybody is embracing Hadoop. MapReduce—that may survive, but it’s certainly not very popular. And again, at the end of the day you can always measure success by revenue and market share and all that, and you can measure the decline of the Hadoop fad by looking at HortonWorks’ stock price. A year ago they were a one-and-a-half billion dollar market cap company; now they’re 500 million. That pretty much reflects the hype cycle of what’s going on with Hadoop: it’s very popular, but it’s not going to be as successful as people thought a couple years ago. But I’ll let you go on to the next question.

**The NoSQL folks claim that NoSQL is “web scale.” Are relational database management systems “web scale”? How does PL/SQL fit into the performance picture? Is PL/SQL “web scale”?**

**Graham Wood:** What is “web scale”? Web scale is always “Oh wow, it’s really big data sets.” Right. So define “really big.” Six years ago now, when we were still doing TPC-Cs, we did 30 million tpmCs—and that’s 500,000 real ACID-compliant transactions per second, each one of those inserting five items, so that’s two-and-a-half million items being inserted every second. How much are you really doing through your website? Yeah, there are some web scale folks out there: there’s Amazon and there’s eBay—PayPal maybe—these guys do have millions of transactions per second, but there’s an awful lot more people who are web scale wannabes. “Oh, well of course we have to use this technology because we have to be able to scale.” To how many orders of magnitude more than where you are now? And because of that you’re going to be handcrafting things in systems that don’t have a lot of functionality—you have to write the functionality that you want to do—so things like joins for example. So, the whole web scale thing does seem to be very much a fashion. My favorite Larry quote is that he once said that the IT business is the only business that’s more fashion-conscious than ladies’ shoes, and I think that’s right. Web scale became a very fashionable thing to be; NoSQL is a very fashionable thing to be. Folks are caught up in the hype and—Andy was saying—these things go through a cycle; they’re much hyped and then they tail off. Relational databases have gone through 30 years of competitors coming and going.

**Bryn Llewellyn:** The other interesting thing about SQL that people rarely mention, and no one has mentioned it yet, is that you can’t write a program with it. Its scope of concept doesn’t extend beyond the single statement; there are not many things that you can achieve with a single SQL statement and, anyway, this piece of text that is so declarative and so wonderful: how do you make it go? Well there’s only one way to make it go: that’s to squirt it out of some or other ordinary IF-THEN-ELSE language. And now there’s just two ways of doing it: have that language

running outside the database or have it running inside. And the purpose of PL/SQL is to be inside the database and to be the vehicle for doing INSERT/UPDATE/DELETE/COMMIT in a coherent fashion so that the many operations on the many tables that you want to do to achieve a certain business purpose either succeed or fail as a unit controlled in that fashion . . . So that’s what PL/SQL is for. It’s just one of many things of getting the SQL done, but it has properties that make doing it that way hugely advantageous and then the SQL that the outside world does to the database is limited to the CALL statement.

**Kyle Hailey:** So, if I understand correctly, then using PL/SQL is the way to sort of supercharge your SQL to make it more efficient because you’re running it within the database?

***“This is trivial technology, and that’s why there are about 40 or 50 companies that have NoSQL databases—including Oracle.”***

**Bryn Llewellyn:** I wouldn’t put it that way. I would say that the purpose of any kind of persistence mechanism is to store the right stuff and to give you back only the right stuff. I didn’t invent that idea. I heard it from a conference platform, from Chris Date, but it’s certainly a crucial idea, isn’t it? So correctness is most important. Talking about performance is a luxury, at least in our world—the kind of application that a relational database is used for: money, people’s life or death, all that stuff. There’s no eventual consistency with my bank account. So, in that arena, correctness is most important, and security is also vital. As it happens, if you do the actual stuff that Graham considers a SQL—INSERT, UPDATE, DELETE, and SELECT—only within the database through a controlled API to achieve correctness and security, you happen also to get this bonus of optimal performance. That’s the way to look at it. And as far as web scale is concerned, well, that’s the operation on the data in the tables, and it’s agnostic at that stage as to how you get it done—whether it’s parallelized or all that stuff—that’s not the domain of discourse of PL/SQL.

**Why does Oracle Corporation sell a NoSQL DBMS?**

**Andy Mendelsohn:** I guess I already talked about this earlier but yes: there’s currently use cases for NoSQL databases and there have been for 40 years—and there will continue to be so. And so, for us at Oracle, we want to have a complete family of data management offerings that span all use cases that customers want to use, and so it made perfect sense for us to take Berkeley DB and, just like Amazon, we added a distribution layer on top of it and we have a very top-notch NoSQL product that we’re investing in quite significantly—and we’re actually going to move out to the cloud this year, where I think we’ll have a lot of customers using our NoSQL engine there in the cloud. And so, yes, there’s definitely a use case for NoSQL, and we want to have a complete family of data management products that solve all use cases.

**Kyle Hailey:** Any quick examples of in which use cases somebody should looking at NoSQL solutions?

**Andy Mendelsohn:** Well, it’s one of these things where relational databases can do everything. NoSQL does a small subset of what relational databases do, and so it’s up to the developers to



decide what they want to do. If they have a fairly complex application and part of that application is simple and can use a NoSQL database, well in that case you might as well just write the whole thing in Oracle SQL and be done with it. Why bother making your life more complex by trying to integrate together multiple data stores and deal with security across multiple data stores? But if you have a simple application that doesn't really need integration with lots of other data stores, sure. Like the classical use case that started NoSQL at Amazon was, you know, "I've got my product catalog and people are browsing all my products. All I want to do is let people look up a product by name and find the price and information about the product." Great use case for NoSQL. User profiles are good use cases. Here's the user ID; give me back all the information about the user that I can use to drive my front end. People are using it again like in fraud detection systems. You can do all the heavy lifting in a data warehouse to figure out people's credit scores and all kinds of other attributes, and then when they do the transaction, you can have all that all pre-computed, sitting in a NoSQL database, and do quick credit checking, fraud detection. Those are some examples; there's lots of use cases.

***If SQL is the best language for Big Data, what explains the rise of Hadoop?***

**Hermann Baer:** Andy alluded to this a little bit before. The short answer here would be "why Hadoop?"—it's because it's cheap and people think it's cool. Although, as Andy said, the coolness is starting to fade off a little bit, and whether it was ever as cool as Oracle red shoes is a different story. But now a little bit more serious. When we look at the question itself, the question itself isn't very clearly defined because it begs "what is Hadoop?" and "what is SQL?" and "what are these things that we compare with each other?" As Andy said before, Hadoop has multiple parts, some dealing with the persistence of data and some others dealing with the attempts of processing data—and the same is actually true with the notion of SQL in that question. People

***"NoSQL does a small subset of what relational databases do, and so it's up to the developers to decide what they want to [use]."***

probably, when this question comes up, automatically imply that when I talk about SQL, I am talking about fixed-structured data that persists somewhere, somehow, on a physical device, and this is actually not true. When we talk about SQL, or at least how we see the world here . . . we talk about the processing of data where it's not necessarily relevant where the data is stored. So talking about SQL as a development language or data manipulation/data access language, it doesn't matter where the data resides in fixed structures of data or whether the data is defined in a more loose context—whether this is a JSON document or anything else—or whether the data is even stored outside the database, and I think that is where Hadoop comes into the picture looking forward. As Andy said, HDFS is definitely a capable, cool, distributed filesystem, and it's cheap, undoubtedly. Someone has to take care of it, someone has to manage the data, and it definitely makes life easier to put data into any kind of persistence structure where

there's no structure to begin with—that is similar to NoSQL. You just put it in and it makes it easier just to potentially get data in, but, at some point in time, the rubber hits the road and people want to do something with the data, and that is where some kind of structure comes into place as well. And this is where we actually want to combine these two worlds, and we put a lot of effort into the development of SQL as the data processing language—make it easy and efficient to work on any kind of structure. We started actually back in the 9i days with the introduction of external tables, which was the very first step into making Oracle a data-processing engine. And this is what we are subsequently developing towards, and we have made big inroads here with the combination of data stored inside the database and data stored outside the database with Big Data SQL, where we're trying to optimize the processing for multiple different shades of data containers to work as efficiently and be as scalable as possible with the data. So I think that is where we have to clearly say that when we look at what we are doing here from the SQL perspective, we're dealing first and foremost with the data processing, and then we're optimizing whatever lies behind there in the data persistence layer to make this as unified, as global, and as effective as possible.

***What is the Oracle Developer Advocates team doing to defend RDBMS?***

**Steven Feuerstein:** So I'd say the Oracle Developer advocates team is focused in two major areas: one is generally content generation . . . so moving away from white papers and PowerPoints to videos primarily, scripts, and so on. I already mentioned Connor's video series on "Keep It Simple SQL." We'll be issuing our first episode of Schema Wars in the next few months, so keep an eye out for that. It's a little riff on Star Wars for database developers. Hopefully it will be very entertaining and go viral. So one area is just providing better content to help developers ingest and make sense of our technology. The second is that we're looking at modernizing and integrating kind of a constellation of websites that we're currently offering to our development community that are just not connected up and delivering enough value, though they're delivering lots of value. What do I mean? "Ask Tom" is still one of the most popular sites for Oracle developers, and it's going to get better . . . Live SQL is a website you can go to—[livesql.oracle.com](http://livesql.oracle.com)—where you can execute SQL and PL/SQL on a 12.1 database. No need to install anything. You can play around with SQL; you can download and run scripts. So it's a script repository, and we'll be upgrading that in terms of the number of scripts and the ability to comment on them and like them and share them. PL/SQL Challenge is a quiz platform that we'll be revamping and offering quizzes across SQL, PL/SQL, and—starting next month—in Application Express. So we're looking at connecting up a lot of these different resources to really give you the better tools to solve your problems and also for newcomers coming into our arena to be able to get their questions answered more quickly.

**Kyle Hailey:** It sounds like a lot of that is for the Oracle community or people coming in the Oracle community. Are we doing anything for these young developers who think Mongo is cool, not even knowing about Oracle?

**Steven Feuerstein:** Oracle is engaging in a number of initiatives around reaching the next generation of developers. Honestly, it's

tough going and we've got catching up to do, but we have an Oracle Academy program that has connections with hundreds of universities and thousands of professors, helping them build curriculum around Oracle, relational database generally, SQL generally. We are going to meetups—reaching out into these open-source communities and letting them know that our offerings are getting stronger and more viable for them. So, for example, there's now a Node.js driver. So you can build JavaScript applications against the Oracle Database through this driver. So we're getting a lot of the pieces in place so developers who are new to Oracle, new to SQL, will be able to engage in the technology with a much lower barrier to entry than has ever been possible before—and, of course, Cloud Database is going to be another very big part of that.

#### *What is Oracle doing to fend off NoSQL and Hadoop?*

**Andy Mendelsohn:** Over the years, whenever interesting new ideas have popped up in the database space, one of the interesting things is ... because Oracle and relational databases in general have this declarative language—this SQL language—and customers write applications at the SQL level, we are free to innovate in the infrastructure of the database with our query optimizer or access methods or parallel query, all that stuff; we are free to change widely, and as long as it makes your SQL applications run faster and with better performance, customers love it. And so a lot of ideas that have come out over the years like parallel query in the '80s and clustering—scalable clustering technologies and now sharding—are all transparent in the infrastructure of the database, and we are free to add them to Oracle and customers will get the advantage of interesting ideas. Like sharding is something that everybody is all excited about now; that's what makes you web scale supposedly. Well in Oracle 12c R2, we're introducing sharding to the product, and as always it's going to be completely transparent to your application, so all those applications you wrote over the years can now be sharded. Now sharding is actually one of these things where if you want the highest performance, you have to actually not be completely transparent to your application; you have to give us the shard key and tell us you know which shard you want this operation to act against, but we also give you a way of just saying, "Here's a SQL statement; Oracle, you just figure out which shards have the data and go do it for me," so we'll do that as well. But sharding is a good example of something we're assimilating into Oracle, and that supposed advantage of NoSQL will go away with 12c R2. We've also added things like JSON; that's in 12c R1, and so the relational model is very flexible. A lot of these supposed advantages of NoSQL systems are just going to be assimilated into Oracle databases, so customers want JSON, they want schemaless, they want sharding, whatever—all that's coming along with Oracle SQL, and Oracle relational databases are including pretty much everything that people think is interesting about NoSQL. So most developers, I think, will be very happy with what they get with Oracle versus NoSQL databases, because they get all the functionality of SQL—and they get a standard interface and all the other benefits of SQL—as opposed to going with NoSQL where, whichever NoSQL engine you go against, you lock yourselves in forever if there's no standards. And people used to really not like to be locked into individual products, especially in this space where the vendors are likely not to be around in five years. One of the nice things about relational databases is they

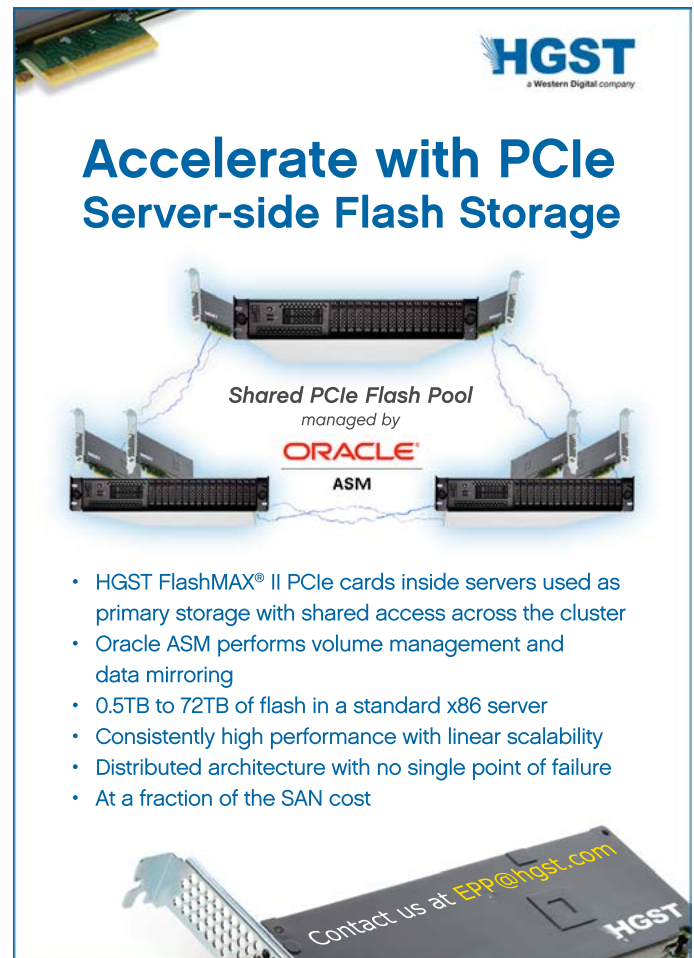
***"It's all going through cycles. SQL is now on the upswing again. NoSQL stuff is now going back on the downswing."***

do absorb any good idea that comes up in any competing data management are ... Graham quoted Larry—"we are in a fashion industry"—and three or four years ago, SQL went out of fashion. I think SQL is coming back into fashion. The Hadoop community is all behind SQL. It's definitely cool again. I don't think there's any question about that, and I think the NoSQL stuff is actually becoming less cool. It's all going through cycles. SQL is now on the upswing again. NoSQL stuff is now going back on the downswing. But again, NoSQL and SQL have co-existed for 40 years, and they'll co-exist for the next 40 years as well.

**Graham Wood:** So you can now have sharding in red.

**Kyle Hailey:** I think that's a great way to end it. SQL is now getting cool again—that's what I heard.

**Graham Wood:** Yes! ▲



**HGST**  
a Western Digital company

## Accelerate with PCIe Server-side Flash Storage

Shared PCIe Flash Pool  
managed by  
**ORACLE**  
ASM

- HGST FlashMAX® II PCIe cards inside servers used as primary storage with shared access across the cluster
- Oracle ASM performs volume management and data mirroring
- 0.5TB to 72TB of flash in a standard x86 server
- Consistently high performance with linear scalability
- Distributed architecture with no single point of failure
- At a fraction of the SAN cost

Contact us at [Epp@hgst.com](mailto:Epp@hgst.com)

# Keeping Up with Oracle Database 12c Multitenant—Book One

Book Notes by Brian Hitchcock

## Details

**Author:** Robert G. Freeman

**ISBN-13:** 978-1507628157

**Pages:** 157

**Date of Publication:** April 6, 2015

**Edition:** 1

**List Price:** \$11.99

**Publisher:** Amazon Digital Services LLC

## Summary

After reading the Oracle Press book, *Oracle Database 12c New Features*, by the same author, I was looking for other 12c books to read and review. This book is self-published; I was curious about the technical material itself, and I also wanted to see what a self-published technical book might be like. While I did learn some things about Oracle Database 12c, the poor quality of this self-published book really gets in the way, making it difficult to read. This is the first of four books in the series, but after over a year, no further volumes are available. At this rate, we will be reading about the next version of the Oracle database before we see the remaining books in this series.

## Introduction

The author tells us that this book is the first in the “Keeping Up with Oracle” series (note the capital “U” on “Up”). However, on the cover, just below the title, it is called the “Keeping up with Oracle” series (note the lowercase “u” on “up”). This isn’t a critical mistake, but it illustrates one of many silly errors contained in this book—errors that should have been caught well before publication. The author explains that traditional books are out of date by the time they are published and that we now have a new paradigm for technical books: a series of self-published books that can be produced more quickly to cover topics that are too specialized to warrant the cost of traditional publishing. It all sounds very promising. The author explains how easy it is to update a self-published book and how quickly new material can be produced. We are told that most Oracle DBAs will be managing Oracle Multitenant databases within three years.

## Chapter 1: Introduction to Oracle Multitenant

This chapter starts by telling us that Oracle Multitenant is a revolutionary architecture—a wave of the future—and that this chapter will discuss the concepts of the Container Database (CDB) and the Pluggable Database (PDB). The question of why we should care about Oracle Multitenant is addressed, especially

since you have to pay more to use it over the base 12c database. It turns out you can use Multitenant without any additional license fee but only if you have only one pluggable database. I think this is referred to as a difference without a distinction. Is there much point to having a multitenant database limited to one tenant?

A more useful answer to why we should care is that this is the future of the Oracle database. Oracle has officially stated that the “old Oracle architecture”—now called the “non-CDB architecture”—will be de-supported at some point. Since CDB is the future, we need to get with the program so we aren’t left behind. The author compares multitenant to RAC, offering that those who didn’t learn RAC were at a disadvantage. I agree with the logic to a certain extent, but then I chose not to focus on RAC and instead learned about WebLogic Server and Fusion Middleware. We can discuss which path is better relative to the job market over an adult beverage at the next NoCOUG post-conference happy hour! It seems that there has been complaining about the need to move to multitenant, but this won’t stop the beast that is Oracle as it trundles into the future.

The most discussed reason to move to Oracle Multitenant is that you can consolidate multiple databases into a single instance. It also simplifies the database upgrade process and has good cloning features. It makes better use of memory and CPU, and completely isolates each PDB within the CDB. Not surprisingly, all these reasons should lead to lower costs, additional license fees notwithstanding.

Multitenant architecture is described as being built on top of the traditional (i.e., “old”) database. Much of the 12c architecture will be familiar. The Container Database (CDB) is described as a container where multiple Oracle databases can share a single instance. Note that by default, when you create a database in 12c, it will be the old kind; you have to use the optional `enable_pluggable_database` clause to create a CDB using the `create database` command. The databases that share the instance are the Pluggable Databases (PDBs). Do not confuse any of this with the Huggable Database, which will be available for the holidays. I’m kidding of course, but the Oracle marketing machine may not be!

While the CDB can contain multiple PDBs, it can run on a RAC cluster. If you want to be confused, you have multiple databases in the CDB running on RAC and serving multiple database instances. The CDB does not contain most of the database metadata, which is found in each PDB. The main components of the CDB are described—the database instance, the CDB data dictionary, the ROOT container, the SEED container, and the PDB—with a diagram showing how they are related. The way the CDB is managed, using the familiar SPFILE, is described, along



with the SGA, which is allocated to the CDB and shared by all the PDBs. The way various tablespaces are implemented is covered—including UNDO, SYSTEM, and SYSAUX—and how the CDB data dictionary is linked to the PDBs. Database users are covered, with common users created in the CDB that can access all the PDBs, while each PDB has its own set of local database users.

The PDB is discussed next: it's a self-contained database and you can have up to 252 in a single CDB. Each PDB is isolated from all the other PDBs, although I think this isolation is not complete, since we have common users. The various ways to create a PDB are listed, with more discussion coming later. There is a lengthy and useful discussion of how the PDBs are kept logically and physically separate.

## Chapter 2: Creating the CDB

This chapter discusses creating the CDB, the container database that will contain the PDBs. The author chooses to focus on using the Database Creation Assistant (DBCA) utility instead of the create database command. We are told to refer to the Oracle documentation if we want to use the create database command. I've experienced enough rodeos to remember a time when it was widely believed that "real" DBAs did everything with SQL commands and never with any kind of GUI. Perhaps the modern rodeo has moved on. There are sections covering the steps needed before, during, and after CDB creation. The same Oracle software is used to create a CDB for the multitenant or the non-multitenant, "old school" Oracle database.

As part of the process of preparing to create the CDB, you need to think about all of the memory and tablespace that will be needed for the CDB and all the PDBs you plan to plug into the CDB. You also need to think about the database character set, since all the databases, CDB and PDB, will share the same single character set. The name of the CDB must be unique on any given server, and it is suggested that the name be unique across your entire environment. Several possibilities are offered for how to choose the unique database name, but in the end we are advised to use a random set of eight characters, so that the database name doesn't give away any information as to what might be inside the database. I can see the logic in this, but—as someone that supports many different databases day in and day out—I need to know whether I'm making a change to a production database or not. Random database names, in my opinion, take the security argument too far. I think there would be more operational mistakes ("I didn't realize this was production!"), and that needs to be traded off against security.

There is a detailed discussion about where to put the files for the CDB and how to size them. This covers file system performance, how critical storage is, and how many database performance issues are linked to the storage infrastructure. Starting with 12c, Oracle ships the Orion utility, which can be used to test your file system for IOPS, MBPS, and latency rates. I had not heard of this utility before. Star Trek's Scotty is quoted in this section, referring to the perils of overthinking. Datafile location and sizing are covered as well. Sizing redo logs is a common cause of performance issues we are told, and the DBCA will, by default, create very small redo log members. With multiple PDBs coming soon, the CDB needs larger redo logs. If you know which existing databases you will move into the new CDB, you can look at the redo logs for those databases and estimate what

will be needed for the CDB. An example is given of how to examine an existing database. Guidelines are given to help estimate the redo log needs for databases that will be plugged into the new CDB but that are not yet fully developed.

When discussing memory sizing for the CDB, the author asserts that DBAs somehow go wrong with memory when moving to new hardware, during upgrades, and when moving to Multitenant, and that most of the time, memory is overallocated. Depending on the cost of memory relative to all the other costs of a project, I'm not sure how bad it is to overallocate and move on. Specific steps to estimate the memory needed for the CDB are discussed, along with other options, such as Real-Application Testing (RAT). RAT, despite the name, is recommended to help with memory and redo log sizing.

The choice of database character set is discussed, along with the specific Oracle recommendation to use AL32UTF8 if at all possible, and the export/import process needed if you have to convert the character set of any existing databases.

Some other parameters to consider when creating the CDB include the PROCESS parameter. Each PDB will inherit the parameter settings of the associated CDB, and there are very few changes that can be made at the PDB level. Advice is given for several other parameters that will be shared by all the PDBs, such as memory-related parameters and DB\_BLOCK\_SIZE. There is a list of Oracle features that are not supported in a Multitenant 12c database. As of 12.1.0.2, among those features is Oracle Streams. I assume this list will be reduced as future versions of 12c are released.

At this point, the actual process of creating the CDB using both the DBCA and the create database statement is presented, with details of the SQL and screenshots for DBCA. Connecting to the newly created CDB is shown using the ORACLE\_SID environment variable and using the service name for the CDB.

## Chapter 3: Creating and Removing PDBs

Now that the CDB has been discussed and created, it is time to cover the PDB. The first section shows how to connect to a PDB, which requires using a service name. There is no environment variable you can set for connecting to a PDB, unlike the CDB where you can use SQL\*Plus after setting ORACLE\_SID. This also means that the database listener must be running and the PDB has been registered with the listener. Next we look at creating a PDB, which can be done manually in several ways. The first way is to use the Seed Container, which is a self-contained PDB that is part of the CDB and is used only for creating PDBs. The Seed Container makes it faster to create PDBs. The SQL for this is shown using the create pluggable database command. The file\_name\_convert option is used to place the files of the new PDB in a different location from those of the Seed Container. When the PDB is first created, it is mounted, not open. You have to open the new PDB using the alter pluggable database open command. The new PDB will have an admin user: a user that does not have SYSDBA privileges and can't be granted SYSDBA. This admin user can only administer the PDB. SQL is presented to look at the v\$pdb table to see the new PDB and its OPEN\_MODE.

Next is creating a PDB by cloning an existing local PDB, and this can be done with or without copying the data into the new PDB. This is done using the create pluggable database command with a target and source PDB name. Note that this creates the network services needed as well as the new PDB and, by default,

the data in the source PDB is copied to the new PDB. There is a no data option that can be used if you do not want the data copied. Note that this option is only available when cloning one PDB to create another PDB. If you are creating a PDB from a non-CDB database, you can't use the no data option. I find the nomenclature confusing at times. A database that is not a 12c Multitenant database is sometimes referred to as non-CDB, non-12c, or non-multitenant. It seems to me that a PDB is a non-CDB, but in this context, "non-CDB" refers to a non-container database, which means a traditional Oracle database. You can have 12c databases and they can be non-multitenant, but these are not container databases. The `create_pdb_clone` clause is mentioned but not discussed.

You can also create a new PDB by cloning an existing PDB over a network link. There are many steps to prepare and execute this, and the steps are shown along with all the SQL needed.

At this point, the fact that the other books in this series are not yet available becomes an issue. We are told that the process of plugging in and unplugging PDBs will be covered in Book 2 of this series. I would like to be able to move on to read about this, but I can't. It has been over a year since this first book was published, but Book 2 is not available.

You can create a PDB using DBCA. This section starts out by telling us that DBCA can also be used to plug in and unplug PDBs, and this will be discussed in Chapter 7. This confused me. I thought plugging and unplugging would be covered in Book 2, and this book—Book 1—only has five chapters. However, the Index is titled Chapter 6, so I guess Chapter 7 is the first chapter of Book 2? Oh well . . . moving on, the process of using DBCA is

explained, and screenshots are included as the Seed Database is used to create the new PDB. When you use DBCA, it shows you all the CDBs that exist on the system so you can choose where to place the new PDB. While using DBCA to create the new PDB, you can choose to implement other features such as Database Vault and Label Security, and you can customize the location of the files for the new PDB. We are told that creating a PDB from the SEED database does not take very long. I would like to hear more about this. How long it takes will affect how useful it is to clone PDBs within and among CDBs.

The final section of this chapter covers dropping and renaming a PDB. Dropping a PDB can be done manually using the drop pluggable database command, or with DBCA. If you drop a PDB without first unplugging it, you need to use the including datafiles clause. It isn't made clear, but I assume this means the datafiles will not be removed if you don't use this optional clause. Dropping a PDB using DBCA is shown with screenshots. It turns out that renaming a PDB can only be done using SQL; DBCA doesn't offer this functionality. When a PDB is renamed, the service name for the PDB is also updated.

The author includes a note at the end of this chapter telling us that there is another option to create PDBs: using the feature called CloneDB—which is fast because it uses copy-on-write technology—but this won't be discussed until Book 2. I will have to look elsewhere to learn about CloneDB if I want to know about it in the near future.

#### Chapter 4: Administration of CDB and PDB Databases

This chapter shows us how to handle many administrative tasks for both the CDB and the PDBs contained in it. A lot of this is very similar to what we have been doing for a long time with non-12c databases. Many of the data dictionary views have been changed to include information regarding containers. There are also changes to the `sys_context` package and the new `show` command for SQL\*Plus.

Each container—the CDB and all the PDBs—has a container ID. A new column called `CON_ID` has been added to many views such as `V$PDBS` and `V$CONTAINERS`. This new column is also used to join tables—for example, to get the names of all the tablespaces and the PDBs they are assigned to. An example of displaying the `CON_ID` is shown for both of these views. Because there are so many containers in the 12c multitenant database, you need to be careful where you are when you execute anything. The new `show` command helps with this. You can show `con_name`, `show con_id` and `show pdbs` when using SQL\*Plus to verify which container you are in. You can do the same thing in PL/SQL using `sys_context` which has additional attributes such as `CDB_NAME`, `CON_ID` and `CON_NAME`. The usage of `sys_context` is shown.

Another example of the new `CON_ID` column is shown using the `v$tablespace`. Now that there are multiple PDBs, and tablespace names are not unique across PDBs, you need to see the `CON_ID` for each tablespace to know exactly which PDB it is part of. Further, what you see when you query `DBA`, `USER`, and `ALL` views is limited to the data for the current container. Note that most views do not have data for all the PDBs but only for the PDB you are currently connected to. When you are in the `ROOT` container—the CDB—you only see information for the CDB. Examples are shown. There is a new type of data dictionary view, the `CDB_view`, which shows the information for all the PDBs—

## DATABASE RECOVERY HAS NEVER BEEN SO SUCCESSFUL!

Axxana's award winning **Phoenix System** offering unprecedented data protection and cross-application consistency for Oracle databases, including Exadata.



ORACLE  
PARTNER NETWORK

AXXANA  
BUILT TO LAST



info@axxana.com • www.axxana.com



but only when you are connected to the ROOT container. Selecting from the CDB\_TABLES view will show all the tables in all the PDBs. At the same time, the CDB\_ view only shows data for those PDBs that are open. As I mentioned before, I find this confusing. The CDB is the container database for all the PDBs, but at other times we refer to each PDB as a container as well. Perhaps I'll get used to it after a while.

A list of the most common V\$ views for Multitenant is shown. Most of the DBA\_ views have an associated CDB\_ view. One of the new views is named DBA\_PDB\_SAVED\_STATES which is explained a little later.

Next up is the topic of starting and stopping the CDB, which is done the same way as a non-multitenant database, namely using startup and shutdown, and the usual options such as shutdown abort. For PDBs, there are three ways to proceed. You can use the alter pluggable database command, you can use the SQL\*Plus startup/shutdown pluggable database, or you can change into the specific PDB from the CDB and use startup and shutdown. Examples of each of these options are shown.

With version 12.1.0.2, after the CDB is restarted the previous state (open or not) of each PDB is preserved. You have to execute the alter pluggable database save state command to enable this. A detailed example of doing this is provided.

You can control the amount of disk space each PDB consumes initially using the pdb\_storage\_clause with the create pluggable database command, or later on with the alter pluggable database command.

Since each PDB is very much like an old-fashioned (i.e., non-12c, non-multitenant) database, creating and dropping tablespac-

es works almost the same as before. The only change is that you are limited by the overall size established for the PDB.

The use of the alter system command is covered. Within the CDB, the usage is familiar. All the PDBs inherit the same system settings from the CDB, but these can be overridden using the scope= parameter to set a parameter for a specific PDB. The number of things you can alter within a PDB is limited.

Using alter session is no different in 12c. You use the alter session set container command to move to a PDB after connecting to the CDB.

The CDB has a temporary tablespace and each PDB has a separate temporary tablespace, and managing these tablespaces is no different in 12c.

Something that is completely new in 12c is the catcon.pl script, which can be used to execute a SQL script across multiple PDBs at the same time. We are told this script will be used for patching and upgrades and an example of using this script is shown. The last topic of the chapter describes how the entire CDB will crash if any of the PDBs have disk-related errors, but there is a new parameter that you can set to avoid this: the \_datafile\_write\_errors\_crash\_instance parameter. I'm assuming that this will be on the certification test!

## Chapter 5: Basic User and Administration of Multitenant Databases

The title of this chapter, as shown in the Table of Contents, doesn't make much sense. When we look at the actual chapter itself, the full title is Basic User Administration and Security of Multitenant Databases. Somehow, in the process of building the

# TRAINING WHEN YOU WANT IT

## TRAINING MEMBERSHIP INCLUDES

- Online training by Craig Shallahamer
- Mastering The Material email follow-up
- How-to webinars
- 24/7 unlimited access
- Priority response
- Discounts



# ORAPUB

Find out more at [orapub.com](http://orapub.com).  
Questions? Contact [support@orapub.com](mailto:support@orapub.com).

TOC, words were left out and switched around.

This chapter covers how to administer users and overall security in the 12c Multitenant database. Like other aspects of 12c, much is familiar, with a few new features to support the container and pluggable databases. Within each PDB, users are granted privileges and roles just as they are in a non-12c database. However, with 12c we now have common users—users that are created in the CDB and appear in all the PDBs as well. Users that are created in each PDB are just like the database users we have been creating in the Oracle database before 12c, but they are now called “local” users since they are local to the specific PDB. In the CDB, we can only create common users and these users have to have user names that start with C##. The common users are created in the root container of the CDB. If you are connected to a PDB, you can create normal users in the normal way, and you can’t create common users. The process of creating, dropping and assigning privileges and roles to common users is shown.

When you grant a common role to a common user while connected to the CDB, that role does not get granted to that same common user in any of the PDBs. For all grants, the scope is limited to the PDB where the grant was executed. We can cause a grant of a system privilege to be applied to all PDBs by adding the container=all clause. Several examples of all of this—including SQL—are shown, as well as some of the error messages you may get in specific situations. Granting object privileges to common users can only be done within a specific PDB. Best practices for all of this are offered as well.

A table is shown to clarify the scope of system and object grants made in the root of the CDB versus in a PDB and whether the container= option is available.

We want each PDB to be isolated from all the others, so we want to have separate local database administrators for each PDB. These users are local administrator accounts. Since local database users are limited to one PDB, you can have the same local username in multiple PDBs.

Having all the PDBs isolated from each other is good, but there will be data in one PDB that a user of another PDB needs to access. In the pre-12c world of the past we would create a database link for this user, and we do the same in 12c Multitenant. An example of the SQL for this is shown. You can also use the containers clause of the select statement, but this is limited to common users, and the object to be queried must be owned by that same common user.

This is the end of this book, and the author provides a summary of the planned content for books Two through Four of the series. Book Two will cover plugging in and unplugging PDBs. Book Three will discuss monitoring, upgrading, and Enterprise Manager. Book Four will end the series with coverage of resource manager, replication, Data Guard, and RAC. Since these books have not yet been published, I can’t read them, much as I would like to.

## Index

The index of a book is not often worth reviewing, but in this case, just glancing at the index, something caught my attention. For both CDB and PDB, the index lists 70 to 80 page numbers. This isn’t useful. It is no surprise that CDB and PDB come up often in the text of this book, but what is the point of listing almost every page number for any item in an index?

## Conclusion

The introduction to this book tells us that it was published in January of 2015. This book is the first of a four-book series and the other books are referred to in this book. As of March 2016, no other books in this series have been published. If the second book were to be published sometime in 2016, this means the full four-book series might not be available to readers until 2018. At this rate, won’t we be discussing the next version of the Oracle database?

When I went looking for information on the other volumes in this series I found the author’s website, where he explains just how simple it is to update a self-published book. In fact, we are told that it can be done in less than 24 hours. Why, then, in the more than a year since publication, hasn’t the author discovered and corrected all the errors I found?

I’m not a copyeditor, but I noticed close to 100 errors without really trying. These are not subtle issues of writing style but very simple problems like repeated words. Finding that many errors made me wonder if the author has even looked at a copy of the published book.

The upside of all this is that editors, copyeditors, and publishers are all shown to be very valuable. If you think you can publish a work of high quality without the help of these professionals, I urge you to read this book. Even the cover photo is strange, with the title of the book in a very small font. There is a reason that it generally takes training and talent to be a professional book designer, and you need to pay for the service if you want your book to look good.

From the comments in the introduction, it is clear that the author understands the promise of self-publishing: being able to bring a work to market quickly and to update that work with very little time and effort. It is ironic that these advantages were not realized. This book would be much better if it had been edited—and by “edited” I mean having someone read it and note the many errors that the casual reader will find. The ability to bring new material to market quickly was also not realized, as it has been over a year since this first book was published and none of the other volumes in the series have even been announced, let alone made available.

I emailed the author to ask about future volumes, but after several months, I have received no response. The brave new world of self-publishing holds much promise. However, along with that comes responsibility. When you tell your readers how easy it is for you to produce and update content, you have a responsibility to produce that content and provide corrections.

I learned things about 12c that will be useful, but the number of errors and the fact that the other books in the series have not appeared make me wonder if my time was well spent. ▲

---

*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/](http://www.brianhitchcock.net/oracle-dbafmw/). The statements and opinions expressed here are the author’s and do not necessarily represent those of Oracle Corporation.*

Copyright © 2016, Brian Hitchcock

May 2016



# Three Database Revolutions

by Guy Harrison



Guy Harrison

*Chapter 1 of Next Generation Databases published by Apress, Dec. 2015, ISBN 978-1484213308. Reprinted with permission. For a complete table of contents, please visit the publisher site: <http://www.apress.com/9781484213308>.*

This book is about a third revolution in database technology. The first revolution was driven by the emergence of the electronic computer, and the second revolution by the emergence of the relational database. The third revolution has resulted in an explosion of nonrelational database alternatives driven by the demands of modern applications that require global scope and continuous availability. In this chapter we'll provide an overview of these three waves of database technologies and discuss the market and technology forces leading to today's next generation databases.

Figure 1-1 shows a simple timeline of major database releases.

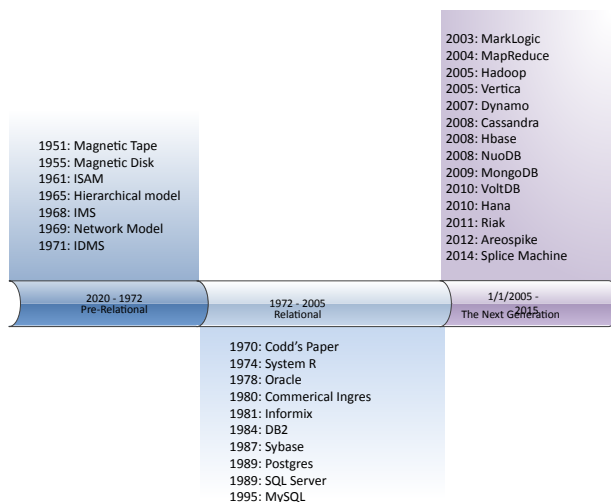


Figure 1-1. Timeline of major database releases and innovations

Figure 1-1 illustrates three major eras in database technology. In the 20 years following the widespread adoption of electronic computers, a range of increasingly sophisticated database systems emerged. Shortly after the definition of the relational model in 1970, almost every significant database system shared a common architecture. The three pillars of this architecture were the relational model, ACID transactions, and the SQL language.

However, starting around 2008, an explosion of new database systems occurred, and none of these adhered to the traditional

relational implementations. These new database systems are the subject of this book, and this chapter will show how the preceding waves of database technologies led to this next generation of database systems.

## Early Database Systems

Wikipedia defines a *database* as an “organized collection of data.” Although the term *database* entered our vocabulary only in the late 1960s, collecting and organizing data has been an integral factor in the development of human civilization and technology. Books—especially those with a strongly enforced structure such as dictionaries and encyclopedias—represent datasets in physical form. Libraries and other indexed archives of information represent preindustrial equivalents of modern database systems.

We can also see the genesis of the digital database in the adoption of punched cards and other physical media that could store information in a form that could be processed mechanically. In the 19th century, loom cards were used to “program” fabric looms to generate complex fabric patterns, while tabulating machines used punched cards to produce census statistics, and player pianos used perforated paper strips that represented melodies. Figure 1-2 shows a Hollerith tabulating machine being used to process the U.S. census in 1890.

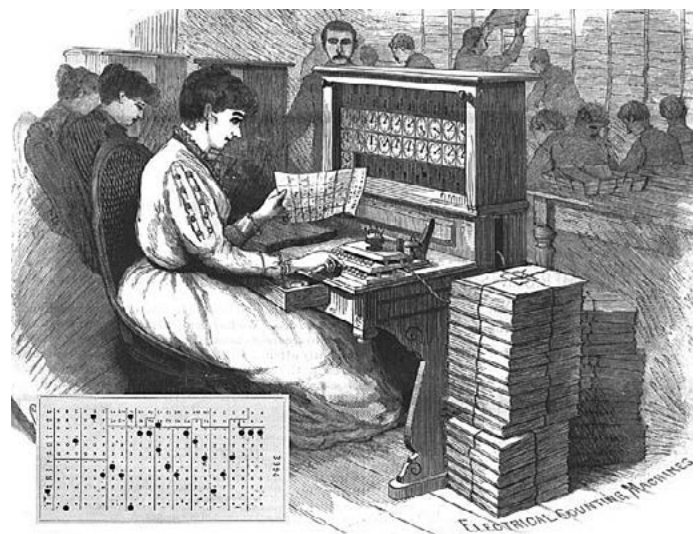


Figure 1-2. Tabulating machines and punched cards used to process 1890 U.S. census

The emergence of electronic computers following the Second World War represented the first revolution in databases. Some early digital computers were created to perform purely mathematical functions—calculating ballistic tables, for instance. But equally as often they were intended to operate on and manipulate data, such as processing encrypted Axis military communications.

Early “databases” used paper tape initially and eventually magnetic tape to store data sequentially. While it was possible to “fast forward” and “rewind” through these datasets, it was not until the emergence of the spinning magnetic disk in the mid-1950s that direct high-speed access to individual records became possible. Direct access allowed fast access to any item within a file of any size. The development of indexing methods such as ISAM (Index Sequential Access Method) made fast record-oriented access feasible and consequently allowed for the birth of the first OLTP (On-line Transaction Processing) computer systems.

ISAM and similar indexing structures powered the first electronic databases. However, these were completely under the control of the application—there were databases but no *Database Management Systems (DBMS)*.

### The First Database Revolution

Requiring every application to write its own data handling code was clearly a productivity issue: every application had to reinvent the database wheel. Furthermore, errors in application data handling code led inevitably to corrupted data. Allowing multiple users to concurrently access or change data without logically or physically corrupting the data requires sophisticated coding. Finally, optimization of data access through caching, pre-fetch, and other techniques required complicated and specialized algorithms that could not easily be duplicated in each application.

Therefore, it became desirable to externalize database handling logic from the application in a separate code base. This layer—the Database Management System, or DBMS—would minimize programmer overhead and ensure the performance and integrity of data access routines.

Early database systems enforced both a schema (a definition of the structure of the data within the database) and an access path (a fixed means of navigating from one record to another). For instance, the DBMS might have a definition of a CUSTOMER and an ORDER together with a defined access path that allowed you to retrieve the orders associated with a particular customer or the customer associated with a specific order.

These first-generation databases ran exclusively on the mainframe computer systems of the day—largely IBM mainframes. By the early 1970s, two major models of DBMS were competing for dominance. The *network model* was formalized by the CODASYL standard and implemented in databases such as IDMS, while the *hierarchical model* provided a somewhat simpler approach as was most notably found in IBM’s IMS (Information Management System). Figure 1-3 provides a comparison of these databases’ structural representation of data.

**Note:** These early systems are often described as “navigational” in nature because you must navigate from one object to another using pointers or links. For instance, to find an order in a hierarchical database, it may be necessary to first locate the customer, then follow the link to the customer’s orders.

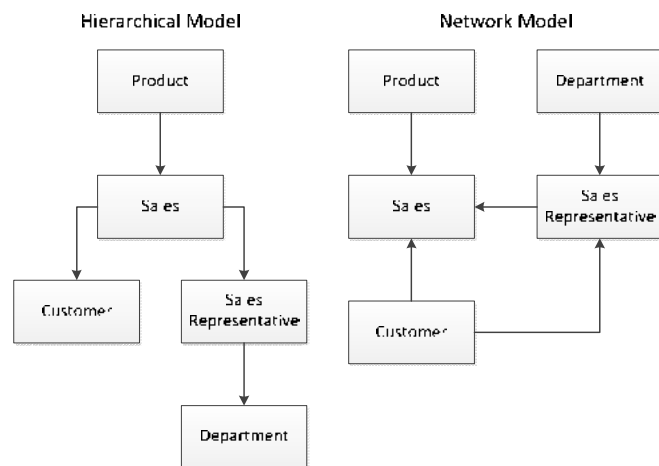


Figure 1-3. Hierarchical and network database models

Hierarchical and network database systems dominated during the era of mainframe computing and powered the vast majority of computer applications up until the late 1970s. However, these systems had several notable drawbacks.

First, the navigational databases were extremely inflexible in terms of data structure and query capabilities. Generally only queries that could be anticipated during the initial design phase were possible, and it was extremely difficult to add new data elements to an existing system.

Second, the database systems were centered on record at a time transaction processing—what we today refer to as CRUD (Create, Read, Update, Delete). Query operations, especially the sort of complex analytic queries that we today associate with business intelligence, required complex coding. The business demands for analytic-style reports grew rapidly as computer systems became increasingly integrated with business processes. As a result, most IT departments found themselves with huge backlogs of report requests and a whole generation of computer programmers writing repetitive COBOL report code.

### The Second Database Revolution

Arguably, no single person has had more influence over database technology than Edgar Codd. Codd received a mathematics degree from Oxford shortly after the Second World War and subsequently immigrated to the United States, where he worked for IBM on and off from 1949 onwards. Codd worked as a “programming mathematician” (ah, those were the days) and worked on some of IBM’s very first commercial electronic computers.

In the late 1960s, Codd was working at an IBM laboratory in San Jose, California. Codd was very familiar with the databases of the day, and he harbored significant reservations about their design. In particular, he felt that:

- **Existing databases were too hard to use.** Databases of the day could only be accessed by people with specialized programming skills.
- **Existing databases lacked a theoretical foundation.** Codd’s mathematical background encouraged him to think about data in terms of formal structures and logical operations; he regarded existing databases as using arbitrary representations that did not ensure logical consistency or provide the ability to deal with missing information.



- **Existing databases mixed logical and physical implementations.** The representation of data in existing databases matched the format of the physical storage in the database, rather than a logical representation of the data that could be comprehended by a nontechnical user.

Codd published an internal IBM paper outlining his ideas for a more formalized model for database systems, which then led to his 1970 paper “A Relational Model of Data for Large Shared Data Banks.”<sup>1</sup> This classic paper contained the core ideas that defined the *relational database model* that became the most significant—almost universal—model for database systems for a generation.

## Relational theory

The intricacies of relational database theory can be complex and are beyond the scope of this introduction. However, at its essence, the relational model describes how a given set of data should be presented to the user, rather than how it should be stored on disk or in memory. Key concepts of the relational model include:

- **Tuples**, an unordered set of attribute values. In an actual database system, a tuple corresponds to a row, and an attribute to a column value.
- **A relation**, which is a collection of distinct tuples and corresponds to a table in relational database implementations.
- **Constraints**, which enforce consistency of the database. Key constraints are used to identify tuples and relationships between tuples.
- **Operations** on relations such as joins, projections, unions, and so on. These operations always return relations. In practice, this means that a query on a table returns data in a tabular format.

A row in a table should be identifiable and efficiently accessed by a unique key value, and every column in that row must be dependent on that key value and no other identifier. Arrays and other structures that contain nested information are, therefore, not directly supported.

Levels of conformance to the relational model are described in the various “*normal forms*.” *Third normal form* is the most common level. Database practitioners typically remember the definition of third normal form by remembering that all non-key attributes must be dependent on “the key, the whole key, and nothing but the key—So Help Me Codd!”<sup>2</sup>

Figure 1-4 provides an example of normalization: the data on the left represents a fairly simple collection of data. However, it contains redundancy in student and test names, and the use of a repeating set of attributes for the test answers is dubious (while possibly within relational form, it implies that each test has the same number of questions and makes certain operations difficult). The five tables on the right represent a normalized representation of this data.

## Transaction Models

The relational model does not itself define the way in which the database handles concurrent data change requests. These changes—generally referred to as database *transactions*—raise

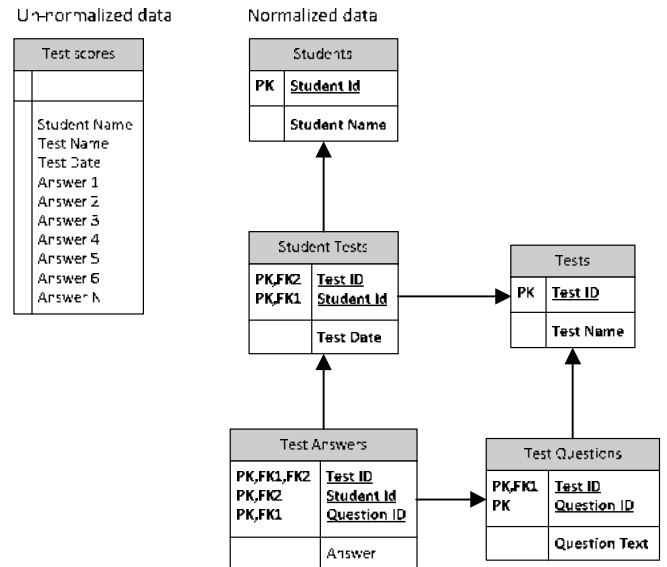


Figure 1-4. Normalized and un-normalized data

issues for all database systems because of the need to ensure consistency and integrity of data.

Jim Gray defined the most widely accepted transaction model in the late 1970s. As he put it, “A transaction is a transformation of state which has the properties of atomicity (all or nothing), durability (effects survive failures) and consistency (a correct transformation).”<sup>3</sup> This soon became popularized as *ACID transactions*: Atomic, Consistent, Independent, and Durable. An ACID transaction should be:

- **Atomic:** The transaction is indivisible—either all the statements in the transaction are applied to the database or none are.
- **Consistent:** The database remains in a consistent state before and after transaction execution.
- **Isolated:** While multiple transactions can be executed by one or more users simultaneously, one transaction should not see the effects of other in-progress transactions.
- **Durable:** Once a transaction is saved to the database (in SQL databases via the COMMIT command), its changes are expected to persist even if there is a failure of operating system or hardware.

ACID transactions became the standard for all serious database implementations, but also became most strongly associated with the relational databases that were emerging at about the time of Gray’s paper.

As we will see later, the restriction on scalability beyond a single data center implied by the ACID transaction model has been a key motivator for the development of new database architectures.

## The First Relational Databases

Initial reaction to the relational model was somewhat lukewarm. Existing vendors including IBM were disinclined to accept Codd’s underlying assumption: that the databases of the day were based on a flawed foundation. Furthermore, many had sincere reservations about the ability of a system to deliver adequate performance if the data representation was not fine-tuned

to the underlying access mechanisms. Would it be possible to create a high-performance database system that allowed data to be accessed in any way the user could possibly imagine?

IBM did, however, initiate a research program to develop a prototype relational database system in 1974, called System R. System R demonstrated that relational databases could deliver adequate performance, and it pioneered the *SQL language*. (Codd had specified that the relational system should include a query language, but had not mandated a specific syntax.) Also during this period, Mike Stonebraker at Berkeley started work on a database system that eventually was called *INGRES*. *INGRES* was also relational, but it used a non-SQL query language called *QUEL*.

At this point, Larry Ellison enters our story. Ellison was more entrepreneurial than academic by nature, though extremely technically sophisticated, having worked at Amdahl. Ellison was familiar both with Codd's work and with System R, and he believed that relational databases represented the future of database technology. In 1977, Ellison founded the company that would eventually become Oracle Corporation and which would release the first commercially successful relational database system.

### Database Wars!

It was during this period that minicomputers challenged and eventually ended the dominance of the mainframe computer. Compared with today's computer hardware, the minicomputers of the late '70s and early '80s were hardly "mini". But unlike mainframes, they required little or no specialized facilities, and they allowed mid-size companies for the first time to own their own computing infrastructure. These new hardware platforms ran new operating systems and created a demand for new databases that could run on these operating systems.

By 1981, IBM had released a commercial relational database called SQL/DS, but since it only ran on IBM mainframe operating systems, it had no influence in the rapidly growing minicomputer market. Ellison's *Oracle* database system was commercially released in 1979 and rapidly gained traction on the minicomputers provided by companies such as Digital and Data General. At the same time, the Berkeley *INGRES* project had given birth to the commercial relational database *Ingres*. Oracle and *Ingres* fought for dominance in the early minicomputer relational database market.

By the mid-'80s, the benefits of the relational database—if not the nuances of relational theory—had become widely understood. Database buyers appreciated in particular that the SQL language, now adopted by all vendors including *Ingres*, provided massive productivity gains for report writing and analytic queries. Furthermore, a next generation of database development tools—known at the time as 4GLs—were becoming increasingly popular and these new tools typically worked best with relational database servers. Finally, minicomputers offered clear price/performance advantages over mainframes especially in the midmarket, and here the relational database was pretty much the only game in town.

Indeed, relational databases became so dominant in terms of mindshare that the vendors of the older database systems became obliged to describe their offerings as also being relational. This prompted Codd to pen his famous 12 rules (actually 13 rules, starting at rule 0) as a sort of acid test to distinguish legitimate relational databases from pretenders.

During the succeeding decades many new database systems were introduced. These include *Sybase*, *Microsoft SQL Server*, *Informix*, *MySQL*, and *DB2*. While each of these systems attempts to differentiate by claiming superior performance, availability, functionality, or economy, they are virtually identical in their reliance on three key principles: Codd's relational model, the SQL language, and the ACID transaction model.

**Note** When we say RDBMS, we generally refer to a database that implements the relational data model, supports ACID transactions, and uses SQL for query and data manipulation.

### Client-server Computing

By the late 1980s, the relational model had clearly achieved decisive victory in the battle for database mindshare. This mind-share dominance translated into market dominance during the shift to *client-server computing*.

Minicomputers were in some respects "little mainframes": in a minicomputer application, all processing occurred on the minicomputer itself, and the user interacted with the application through dumb "green screen" terminals. However, even as the minicomputer was becoming a mainstay of business computing, a new revolution in application architecture was emerging.

The increasing prevalence of microcomputer platforms based on the IBM PC standard, and the emergence of graphical user interfaces such as Microsoft Windows, prompted a new application paradigm: client-server. In the client-server model, presentation logic was hosted on a PC terminal typically running Microsoft Windows. These PC-based client programs communicated with a database server typically running on a minicomputer. Application logic was often concentrated on the client side, but could also be located within the database server using the *stored procedures*—programs that ran inside the database.

Client-server allowed for a richness of experience that was unparalleled in the green-screen era, and by the early '90s, virtually all new applications aspired to the client-server architecture. Practically all client-development platforms assumed an RDBMS backend—indeed, usually assumed SQL as the vehicle for all requests between client and server.

### Object-oriented Programming and the OODBMS

Shortly after the client-server revolution, another significant paradigm shift impacted mainstream application-development languages. In traditional "procedural" programming languages, data and logic were essentially separate. Procedures would load and manipulate data within their logic, but the procedure itself did not contain the data in any meaningful way. *Object-oriented (OO) programming* merged attributes and behaviors into a single object. So, for instance, an employee object might represent the structure of employee records as well as operations that can be performed on those records—changing salary, promoting, retiring, and so on. For our purposes, the two most relevant principles of object-oriented programming are:

- **Encapsulation:** An object class encapsulates both data and actions (methods) that may be performed on that data. Indeed, an object may restrict direct access to the underlying data, requiring that modifications to the data be possible only via an object's methods. For instance, an employee class might include a method to retrieve salary and another method to modify salary. The salary-modifi-

cation method might include restrictions on minimum and maximum salaries, and the class might allow for no manipulation of salary outside of these methods.

- **Inheritance:** Object classes can inherit the characteristics of a parent class. The employee class might inherit all the properties of a people class (DOB, name, etc.) while adding properties and methods such as salary, employee date, and so on.

Object-oriented programming represented a huge gain in programmer productivity, application reliability, and performance. Throughout the late '80s and early '90s, most programming languages converted to an object-oriented model, and many significant new languages—such as Java—emerged that were natively object-oriented.

The object-oriented programming revolution set the stage for the first serious challenge to the relational database, which came along in the mid-1990s. Object-oriented developers were frustrated by what they saw as an impedance mismatch between the object-oriented representations of their data within their programs and the relational representation within the database. In an object-oriented program, all the details relevant to a logical unit of work would be stored within the one class or directly linked to that class. For instance, a customer object would contain all details about the customer, with links to objects that contained customer orders, which in turn had links to order line items. This representation was inherently nonrelational; indeed, the representation of data matched more closely to the network databases of the CODASYL era.

When an object was stored into or retrieved from a relational database, multiple SQL operations would be required to convert from the object-oriented representation to the relational representation. This was cumbersome for the programmer and could lead to performance or reliability issues. Figure 1-5 illustrates the problem.

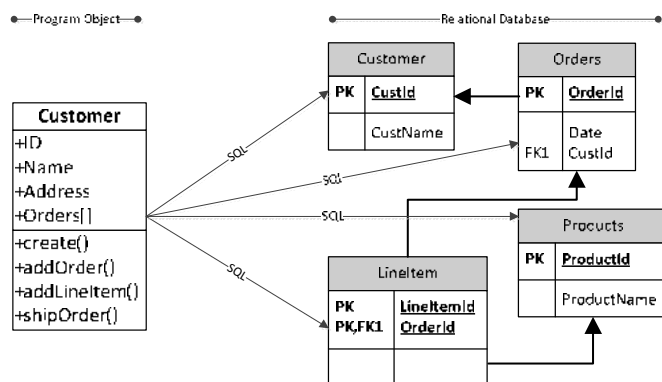


Figure 1-5. Storing an object in an RDBMS requires multiple SQL operations

Advocates of object-oriented programming began to see the relational database as a relic of the procedural past. This led to the rather infamous quote: “A relational database is like a garage that forces you to take your car apart and store the pieces in little drawers.”

The rapid success of object-oriented programming led almost inevitably to the proposition that an *Object Oriented Database Management System* (OODBMS) was better suited to meet the demands of modern applications. An OODBMS would store

program objects directly without normalization, and would allow applications to load and store their objects easily. The object-oriented database movement created a manifesto outlining the key arguments for and properties of OODBMS.<sup>4</sup> In implementation, OODBMS resembles the navigational model from the pre-relational era—pointers within one object (a customer, for instance) would allow navigation to related objects (such as orders).

Advocacy for the OODBMS model grew during the mid-'90s, and to many it did seem natural that the OODBMS would be the logical successor to the RDBMS. The incumbent relational database vendors—at this point, primarily Oracle, Informix, Sybase, and IBM—rapidly scrambled to implement OODBMS features within their RDBMS. Meanwhile, some pure OODBMS systems were developed and gained initial traction.

However, by the end of the decade, OODBMS systems had completely failed to gain market share. Mainstream database vendors such as Oracle and Informix had successfully implemented many OODBMS features, but even these features were rarely used. OO programmers became resigned to the use of RDBMS systems to persist objects, and the pain was somewhat alleviated by *Object-Relational Mapping* (ORM) frameworks that automated the most tedious aspects of the translation.

There are competing and not necessarily contradictory explanations for the failure of the OO database. For my part, I felt the advocates of the OODBMS model were concentrating on only the advantages an OODBMS offered to the application developer, and ignoring the disadvantages the new model had for those who wished to consume information for business purposes. Databases don't exist simply for the benefit of programmers; they represent significant assets that must be accessible to those who want to mine the information for decision making and business intelligence. By implementing a data model that could only be used by the programmer, and depriving the business user of a usable SQL interface, the OODBMS failed to gain support outside programming circles.

However, as we shall see in Chapter 4, the motivations for an OODBMS heavily influenced some of today's most popular non-relational databases.

## The Relational Plateau

Once the excitement over object-oriented databases had run its course, relational databases remained unchallenged until the latter half of the 2000s. In fact, for a period of roughly 10 years (1995–2005), no significant new databases were introduced: there were already enough RDBMS systems to saturate the market, and the stranglehold that the RDBMS model held on the market meant no nonrelational alternatives could emerge. Considering that this period essentially represents a time when the Internet grew from geeky curiosity to an all-pervasive global network, that no new database architectures emerged during this period is astonishing, and it is a testament to the power of the relational model.

## The Third Database Revolution

By the middle of the 2000s, the relational database seemed completely entrenched. Looking forward from 2005, it seemed that although we would see continuing and significant innovation within the relational database systems of the day, there were no signs of any radical changes to come. But in fact, the era of



complete relational database supremacy was just about to end.

In particular, the difference in application architectures between the client-server era and the era of massive web-scale applications created pressures on the relational database that could not be relieved through incremental innovation.

### Google and Hadoop

By 2005, Google was by far the biggest website in the world—and this had been true since a few years after Google first launched. When Google began, the relational database was already well established, but it was inadequate to deal with the volumes and velocity of the data confronting Google. The challenges that enterprises face with “big data” today are problems that Google first encountered almost 20 years ago. Very early on, Google had to invent new hardware and software architectures to store and process the exponentially growing quantity of websites it needed to index.

In 2003, Google revealed details of the distributed file system *GFS* that formed a foundation for its storage architecture,<sup>5</sup> and in 2004 it revealed details of the distributed parallel processing algorithm *MapReduce*, which was used to create World Wide Web indexes.<sup>6</sup> In 2006, Google revealed details about its *BigTable* distributed structured database.<sup>7</sup>

These concepts, together with other technologies, many of which also came from Google, formed the basis for the *Hadoop* project, which matured within Yahoo! and which experienced rapid uptake from 2007 on. The Hadoop ecosystem more than anything else became a technology enabler for the Big Data ecosystem we’ll discuss in more detail in Chapter 2.

### The Rest of the Web

While Google had an overall scale of operation and data volume way beyond that of any other web company, other websites had challenges of their own. Websites dedicated to online e-commerce—Amazon, for example—had a need for a transactional processing capability that could operate at massive scale. Early social networking sites such as MySpace and eventually Facebook faced similar challenges in scaling their infrastructure from thousands to millions of users.

Again, even the most expensive commercial RDBMS such as Oracle could not provide sufficient scalability to meet the demands of these sites. Oracle’s scaled-out RDBMS architecture (Oracle RAC) attempted to provide a roadmap for limitless scalability, but it was economically unattractive and never seemed to offer the scale required at the leading edge.

Many early websites attempted to scale open-source databases through a variety of do-it-yourself techniques. This involved utilizing distributed object cases such as Memcached to offload database load, database replication to spread database read activity, and eventually—when all else failed—“Sharding.”

*Sharding* involves partitioning the data across multiple databases based on a key attribute, such as the customer identifier. For instance, in Twitter and Facebook, customer data is split up across a very large number of MySQL databases. Most data for a specific user ends up on the one database, so that operations for a specific customer are quick. It’s up to the application to work out the correct shard and to route requests appropriately.

Sharding at sites like Facebook has allowed a MySQL-based system to scale up to massive levels, but the downsides of doing this are immense. Many relational operations and database-level

ACID transactions are lost. It becomes impossible to perform joins or maintain transactional integrity across shards. The operational costs of sharding, together with the loss of relational features, made many seek alternatives to the RDBMS.

Meanwhile, a similar dilemma within Amazon had resulted in development of an alternative model to strict ACID consistency within its homegrown data store. Amazon revealed details of this system, “Dynamo,” in 2008.<sup>8</sup>

Amazon’s Dynamo model, together with innovations from web developers seeking a “webscale” database, led to the emergence of what came to be known as *key-value databases*. We’ll discuss these in more detail in Chapter 3.

### Cloud Computing

The existence of applications and databases “in the cloud”—that is, accessed from the Internet—had been a persistent feature of the application landscape since the late 1990s. However, around 2008, cloud computing erupted somewhat abruptly as a major concern for large organizations and a huge opportunity for startups.

For the previous 5 to 10 years, mainstream adoption of computer applications had shifted from rich desktop applications based on the client-server model to web-based applications whose data stores and application servers resided somewhere accessible via the Internet—“the cloud.” This created a real challenge for emerging companies that needed somehow to establish sufficient hosting for early adopters, as well as the ability to scale up rapidly should they experience the much-desired exponential growth.

Between 2006 and 2008, Amazon rolled out *Elastic Compute Cloud (EC2)*. EC2 made available virtual machine images hosted on Amazon’s hardware infrastructure and accessible via the Internet. EC2 could be used to host web applications, and computing power could be relatively rapidly added on demand. Amazon added other services such as storage (S3, EBS), Virtual Private Cloud (VPC), a MapReduce service (EMR), and so on. The entire platform was known as *Amazon Web Services (AWS)* and was the first practical implementation of an *Infrastructure as a Service (IaaS)* cloud. AWS became the inspiration for cloud computing offerings from Google, Microsoft, and others.

For applications wishing to exploit the elastic scalability allowed by cloud computing platforms, existing relational databases were a poor fit. Oracle’s attempts to integrate grid computing into its architecture had met with only limited success and were economically and practically inadequate for these applications, which needed to be able to expand on demand. That demand for elastically scalable databases fueled the demand generated by web-based startups and accelerated the growth of key-value stores, often based on Amazon’s own Dynamo design. Indeed, Amazon offered nonrelational services in its cloud starting with *SimpleDB*, which eventually was replaced by *DynamoDB*.

### Document Databases

Programmers continued to be unhappy with the impedance mismatch between object-oriented and relational models. Object relational mapping systems only relieved a small amount of the inconvenience that occurred when a complex object needed to be stored on a relational database in normal form.

Starting about 2004, an increasing number of websites were able to offer a far richer interactive experience than had been the

case previously. This was enabled by the programming style known as *AJAX (Asynchronous JavaScript and XML)*, in which JavaScript within the browser communicates directly with a backend by transferring XML messages. XML was soon superseded by *JavaScript Object Notation (JSON)*, which is a self-describing format similar to XML but is more compact and tightly integrated into the JavaScript language.

JSON became the de facto format for storing—serializing—objects to disk. Some websites started storing JSON documents directly into columns within relational tables. It was only a matter of time before someone decided to eliminate the relational middleman and create a database in which JSON could be directly stored. These became known as *document databases*.

*CouchBase* and *MongoDB* are two popular JSON-oriented databases, though virtually all nonrelational databases—and most relational databases, as well—support JSON. Programmers like document databases for the same reasons they liked OODBMS: it relieves them of the laborious process of translating objects to relational format. We'll look at document databases in some detail in Chapter 4.

### The “NewSQL”

Neither the relational nor the ACID transaction model dictated the physical architecture for a relational database. However, partly because of a shared ancestry and partly because of the realities of the hardware of the day, most relational databases ended up being implemented in a very similar manner. The format of data on disk, the use of memory, the nature of locks, and so on varied only slightly among the major RDBMS implementations.

In 2007, Michael Stonebraker, pioneer of the Ingres and Postgres database systems, led a research team that published the seminal paper “The End of an Architectural Era (It's Time for a Complete Rewrite).”<sup>9</sup> This paper pointed out that the hardware assumptions that underlie the consensus relational architecture no longer applied, and that the variety of modern database workloads suggested a single architecture might not be optimal across all workloads.

Stonebraker and his team proposed a number of variants on the existing RDBMS design, each of which was optimized for a specific application workload. Two of these designs became particularly significant (although to be fair, neither design was necessarily completely unprecedented). *H-Store* described a pure in-memory distributed database while *C-Store* specified a design for a columnar database. Both these designs were extremely influential in the years to come and are the first examples of what came to be known as *NewSQL* database systems—databases that retain key characteristics of the RDBMS but that diverge from the common architecture exhibited by traditional systems such as Oracle and SQL Server. We'll examine these database types in Chapters 6 and 7.

### The Nonrelational Explosion

As we saw in Figure 1-1, a huge number of relational database systems emerged in the first half of the 2000s. In particular, a sort of “Cambrian explosion” occurred in the years 2008–2009: literally dozens of new database systems emerged in this short period. Many of these have fallen into disuse, but some—such as MongoDB, Cassandra, and HBase—have today captured significant market share.

At first, these new breeds of database systems lacked a com-

mon name. “Distributed Non-Relational Database Management System” (DNRDBMS) was proposed, but clearly wasn't going to capture anybody's imagination. However, in late 2009, the term *NoSQL* quickly caught on as shorthand for any database system that broke with the traditional SQL database.

In the opinion of many, NoSQL is an unfortunate term: it defines what a database is not rather than what it is, and it focuses attention on the presence or absence of the SQL language. Although it's true that most nonrelational systems do not support SQL, actually it is variance from the strict transactional and relational data model that motivated most NoSQL database designs.

By 2011, the term *NewSQL* became popularized as a means of describing this new breed of databases that, while not representing a complete break with the relational model, enhanced or significantly modified the fundamental principles—and this included columnar databases, discussed in Chapter 6, and in some of the in-memory databases discussed in Chapter 7.

Finally, the term *Big Data* burst onto mainstream consciousness in early 2012. Although the term refers mostly to the new ways in which data is being leveraged to create value, we generally understand “Big Data solutions” as convenient shorthand for technologies that support large and unstructured datasets such as Hadoop.

**Note** NoSQL, NewSQL, and Big Data are in many respects vaguely defined, overhyped, and overloaded terms. However, they represent the most widely understood phrases for referring to next-generation database technologies.

Loosely speaking, NoSQL databases reject the constraints of the relational model, including strict consistency and schemas. NewSQL databases retain many features of the relational model but amend the underlying technology in significant ways. Big Data systems are generally oriented around technologies within the Hadoop ecosystem, increasingly including Spark.

### Conclusion: One Size Doesn't Fit All

The first database revolution arose as an inevitable consequence of the emergence of electronic digital computers. In some respect, the databases of the first wave were electronic analogs of pre-computer technologies such as punched cards and tabulating machines. Early attempts to add a layer of structure and consistency to these databases may have improved programmer efficiency and data consistency, but they left the data locked in systems to which only programmers held the keys.

The second database revolution resulted from Edgar Codd's realization that database systems would be well served if they were based on a solid, formal, and mathematical foundation; that the representation of data should be independent of the physical storage implementation; and that databases should support flexible query mechanisms that do not require sophisticated programming skills.

The successful development of the modern relational database over such an extended time—more than 30 years of commercial dominance—represents a triumph of computer science and software engineering. Rarely has a software theoretical concept been so successfully and widely implemented as the relational database.

The third database revolution is not based on a single architectural foundation. If anything, it rests on the proposition that a single database architecture cannot meet the challenges posed in

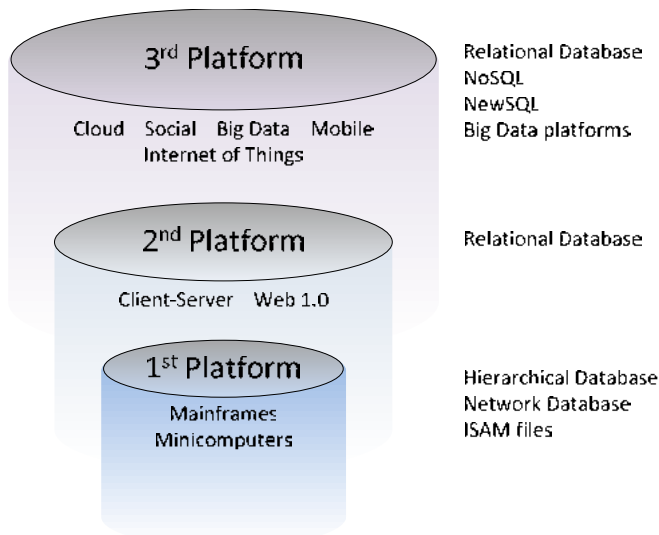


Figure 1-6. IDC's "three platforms" model corresponds to three waves of database technology

our modern digital world. The existence of massive social networking applications with hundreds of millions of users and the emergence of *Internet of Things* (IoT) applications with potentially billions of machine inputs, strain the relational database—and particularly the ACID transaction model—to the breaking point. At the other end of the scale we have applications that must run on mobile and wearable devices with limited memory and computing power. And we are awash with data, much of which is of unpredictable structure for which rendering to relational form is untenable.

**DELPHIX®**

Database Virtualization Software

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

www.delphix.com

The third wave of databases roughly corresponds to a third wave of computer applications. IDC and others often refer to this as "the third platform." The first platform was the mainframe, which was supported by pre-relational database systems. The second platform, client-server and early web applications, was supported by relational databases. The third platform is characterized by applications that involve cloud deployment, mobile presence, social networking, and the Internet of Things. The third platform demands a third wave of database technologies

***"Now that the hegemony of the RDBMS has been broken, we are free to design database systems whose only constraint is our imagination."***

that include but are not limited to relational systems. Figure 1-6 summarizes how the three platforms correspond to our three waves of database revolutions.

It's an exciting time to be working in the database industry. For a generation of software professionals (and most of my professional life), innovation in database technology occurred largely within the constraints of the ACID-compliant relational databases. Now that the hegemony of the RDBMS has been broken, we are free to design database systems whose only constraint is our imagination. It's well known that failure drives innovation. Some of these new database system concepts might not survive the test of time; however, there seems little chance that a single model will dominate the immediate future as completely as had the relational model. Database professionals will need to choose the most appropriate technology for their circumstances with care; in many cases, relational technology will continue be the best fit—but not always. ▲

#### Notes

1. <http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf>
2. William Kent, "A Simple Guide to Five Normal Forms in Relational Database Theory," 1983.
3. <http://research.microsoft.com/en-us/um/people/gray/papers/theTransactionConcept.pdf>
4. <https://www.cs.cmu.edu/~clamen/OODBMS/Manifesto/>
5. <http://research.google.com/archive/gfs.html>
6. <http://research.google.com/archive/mapreduce.html>
7. <http://research.google.com/archive/bigtable.html>
8. <http://queue.acm.org/detail.cfm?id=1466448>
9. <http://nms.csail.mit.edu/~stavros/pubs/hstore.pdf>

*Guy Harrison started working as a database developer in the mid-1980s. He has written numerous books on database development and performance optimization. He joined Quest Software (now part of Dell) in 2000, and currently leads the team that develops the Toad, Spotlight, and Shareplex product families. Guy lives in Melbourne Australia, with his wife, a variable number of adult children, a cat, three dogs, and a giant killer rabbit. Guy can be reached at [guy.a.harrison@gmail.com](mailto:guy.a.harrison@gmail.com) at <http://guyharrison.net> and is [@guyharrison](#) on Twitter.*



# No Data Loss Without a Synchronous Network

by Arup Nanda



Arup Nanda

I don't usually write about specific products, but once in a while I encounter something worth talking about, especially if it addresses a very common issue that anyone with datacenter management responsibilities will understand: avoiding the loss of last-minute data changes in a database after a disaster without having to use expensive synchronous replication. Phoenix Systems from Axxana solves that problem with an innovative out-of-the-box approach.

## Replication and Last Mile Transactions

Imagine this scenario: the datacenter is on fire. All components—servers, storage, network—are destroyed. You can't salvage anything of value. What happens to the data you need to continue your business? Fortunately you have a disaster recovery plan. You replicate the database to a remote datacenter using either Data Guard (from Oracle) or some storage level technology such as EMC's SRDF. So the database comes up on the remote site, but there is a problem. Since you used asynchronous replication (such as maximum performance mode in Data Guard), not all changes have made it to the remote site. The final few changes, usually known as "last mile" transactions, were yet to be shipped to the remote site when the disaster struck. These are lost forever in the fire. While you do have a database at the remote datacenter and that database is almost complete, the operative word is *almost*: some data inevitably is lost. What if you can't afford to be "almost" complete? For organizations such as financial institutions, hospitals, manufacturing concerns, and airlines where losing data is simply not an option, this is not an affordable luxury. Even in the case of other organizations where data loss may not be as unacceptable, the loss of data creates a sense of uncertainty, since you don't know exactly *what* was lost.

If you need to be 100% certain that all data is available at the remote site, what would you do?

It's easy using *synchronous* replication. All changes on the primary database are reflected on the remote database site in real time. If you used Data Guard, you would have run it in maximum protection mode. In that case, Oracle would have committed the changes in the remote database before confirming the commit at the local site. When the primary site is gone, the remote site is available with 100% of the data. Bye-bye, "almost."

Brilliant. So, why didn't you do it? Better yet, why doesn't everyone do it, since it is that simple? Data loss could be unacceptable in the worst case and unsettling at best. Why accept it?

## Synchronous Replication

Well, there is a pesky little detail: synchronous replication means the network connectivity has to be rock solid. Networks have three general characteristics: 1) throughput (how much data they can pump. Imagine a three-lane highway compared to a single-lane local road); 2) latency (how much time passes to process the data, not the actual speed. For instance, the car on the highway may travel at 70 miles per hour, but it will spend a few minutes on the ramp to the highway); and 3) reliability (does the data move from point A to point B with 100% accuracy, or does it have to be re-transmitted a few times?).

Synchronous replication requires the throughput to be very high (to support the large amounts of change at least during bursts of activity), the latency to be very low (otherwise the remote site will get the data late and respond to the primary even later, causing the primary to hold off the commit), and the network to be extremely reliable (otherwise the primary may think the remote site is not accessible and therefore shut itself down to protect the data). If you run Data Guard in maximum protection mode, you are essentially telling Oracle to make sure that the remote site has absolutely, positively, undoubtedly (add any other adverb you can think of) received the change and has committed.

If Oracle can't ensure that result for any reason—such as not getting the response in time due to a less reliable network—what choices does it have? If it allows further data changes, the changes are not yet there at the remote site. If the primary database fails at this point, the data is gone. Therefore Oracle has no choice but to shut down the primary database to stop any changes coming in. It's maximum *protection*, after all, and that's what you have instructed it to do.

So, if you decide to use maximum protection, you have to use a high throughput, a low latency, and an extremely reliable network infrastructure, which most public commercial network infrastructures are not. Either you have to contract a commercial carrier to provide this elevated level of service or build your own, such as using dark fiber. The costs of the network infrastructure become exponentially higher, especially when the remote site is farther away from the primary. In many cases, the cost of the network itself may be several times that of the database infrastructure it protects. Owing to the cost limitations, you may be forced to locate the remote site close by, e.g., in New York City and Hoboken, NJ. It will still be exceedingly expensive, and it may not offer the same degree of protection that you expect.

These two cities are close enough to be in the same exposure area of disasters such as floods, hurricanes, war, and so on. The farther away the remote site is, the more protected your data is, but your cost increases. Could you accomplish no data loss without this expensive proposition?

Until now, the decision was really black and white. If you want no data loss at all, you have no choice but to go for a super-expensive network solution. Many companies can't justify that high sticker price and therefore settle for potential data loss. Many, in fact, make detailed plans as a part of the business continuity efforts to handle this lost data.

It's assumed that zero data loss is synonymous with an expensive network. If you don't have a high throughput, a low latency, and a highly reliable network, you have to live with some data loss.

Here is the good news: no, you don't have to. Now, it's possible to have the cheaper commoditized public network infrastructure and still have complete data protection. Allow me to explain.

### Data Protection in Oracle

In Oracle, the datafiles are written asynchronously at different intervals unrelated to the data changes and commits. In other words, when you commit a change, the datafiles may not have that changed data. In fact the change occurs in the memory only (called a "buffer cache") and may not exist in the datafiles for hours afterwards. Similarly, when you make a change but don't commit, the data can still be persisted to the datafiles. Let me repeat that: the datafiles are updated with the changed data even if you didn't commit yet. This is the reason that if you have a storage or operating system level replication solution—even a synchronous one—replicating the data files, the remote site may or may not have the data, even hours after the change.

How does Oracle protect the data that was changed and committed but still in memory, if the datafiles do not have them? It captures the pre- and post-change data and packages them into something called "redo blocks." Remember, these have nothing to do with data blocks. These are merely changes created by activities performed on the database. This redo data—also known as "redo vector"—is written to a special area in memory called the "log buffer." When you commit, the relevant redo blocks from the log buffer are written to special files in the database called "redo log files," also known as "online redo log files." The commit waits until this writing—known as "redo flushing"—has ended. You can check the Oracle sessions waiting for this flushing to complete by looking at the event "log file sync." Since the changes—most importantly, the *committed* changes—are recorded in the redo log files, Oracle does not need to rely on the memory alone to know which changes are committed and which are not. In case of a failure, Oracle examines the redo logs to find these changes and updates the data files accordingly. Redo logs are very small compared to the datafiles.

By the way, redo flushing also occurs at other times—every three seconds, every filled 1 MB of log buffer, when a third of the log buffer is full, and some other events—but those additional flushes merely make sure the redo log file is up to date, even if there is no commit.

As you can see, this redo log file becomes the most important thing in the data recovery process. When a disaster occurs, you may have copies of the datafiles at the remote site (thanks to the

replication), but as you learned in the previous section, the copies are not useful yet since they may not have all the committed changes and may even have uncommitted changes. In other words, this copy is not considered "consistent" by Oracle. After a disaster, Oracle needs to check the redo log files and apply the changes to make the datafiles consistent. This is known as a "media recovery." You have to initiate the media recovery process. In case of a synchronous replication at the storage or operating system level, the redo logs are perfectly in sync with the primary site, and Oracle has no trouble getting to the last committed transaction just before the failure. There will be no data lost as a result of the recovery process. In case of Data Guard with maximum protection, this is not required since the changes are updated at the remote site anyway. But what about your cheaper commodity network with asynchronous replication? The redo logs at the remote site will not be up to date with the primary site's redo logs. When you perform media recovery, you can't get to the very last change before the failure, simply because you may not have it. Therefore you can perform a recovery only up to the last available information in the redo at the remote site. This is known as "incomplete" media recovery, distinguished from the earlier described "complete" media recovery. To complete the media recovery, you need the information in the redo log files at the primary site—but remember: the primary site is destroyed. This information is gone. You end up with a data loss. Perhaps even worse, you won't even know exactly how much you lost, since you don't have the access to the primary redo log files.

Now consider this situation carefully. All you need is the last redo log file from the primary database to complete the recovery. Unfortunately the file is not available because it's destroyed or otherwise inaccessible since the site itself is inaccessible. This tiny little file is the only thing that stays between you and complete recovery. What if you somehow magically had access to this file, even though the rest of the data center is gone? You would have been able to complete the recovery with no data loss and looked like a hero, all without a synchronous replication solution with a super-expensive network.

### Enter the Black Box

Oracle's redo data can be written to multiple files at the same time. This is called a "redo log group," and the files are called "members." Log flushing writes to all members before confirming the flush. As you can see, all members of a group have the same data. Multiple members are created only for redundancy. As long as one member of a group is available, Oracle can use it to perform recovery. This is where the new tool from Axxana comes in. It is a storage device named Phoenix, where you create the second member of the redo log groups. The first member of the group is in your normal storage as usual. When disaster strikes and nukes the primary site, you have a copy of the all-important redo log from the Phoenix system.

This is where the first benefit of Phoenix systems comes in. The storage is not just an ordinary one. It's encased in a bomb-proof, fireproof, and waterproof container that protects the internal storage from many calamities. The storage has normal connectivity, such as a network port to connect a network as a NAS and a fiber port to connect to a fiber switch as a SAN. Under ordinary circumstances you would use these ports to connect to your infrastructure system and use the storage. After

a disaster, due to the indestructible nature of the enclosure, this storage will most likely be intact. All you have to do is access the data from it and perform a complete recovery. No data needs to be lost.

But how do you get to the information on the Phoenix system? This is where the second benefit comes in. The Phoenix system transfers its data to another component of the Phoenix system at the remote site. The Phoenix system creates a replica of the required redo logs at the remote site. Since the only data on it are the small redo log files, the amount to transfer is very small and does not put a toll on your other network infrastructure.

Note a very important point: the Phoenix system does not perform any transfer of data during normal operation. It's only during a disaster that the system transports the required redo log files to complete a 100% recovery at the remote site.

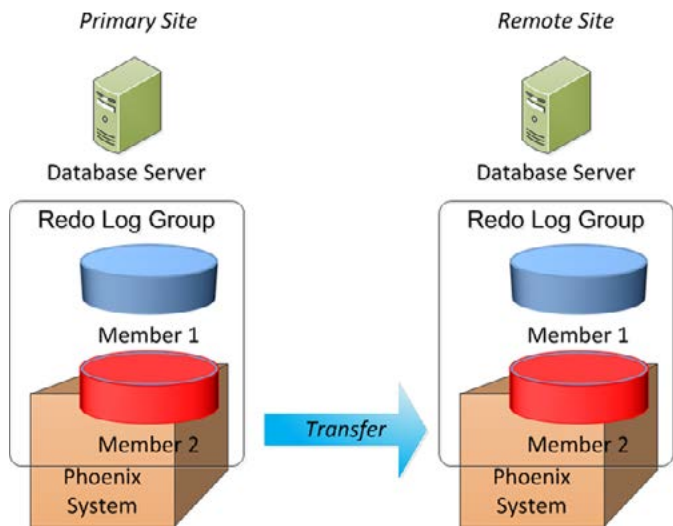


Figure 1

But it still depends on getting the data out of the Phoenix system that was at the primary site. What if the disaster site is physically inaccessible or it is not feasible to physically transport the Phoenix system to a location where it can be connected to your network? This is quite possible in the case of floods, hurricanes, or other natural disasters—or manmade ones like wars or strikes. The network cables are also likely out of commission after a disaster. Without that access, how can the Phoenix system extract the needed last mile transactions from the primary site?

No worries: there is a cellular modem built into the Phoenix system that allows you to connect to it from the remote site and transfer the data wirelessly over a cellular network. The system also has its own battery that allows it to stay operational even when external power is gone—a common occurrence in almost any calamity. What's more, the transfer of data after the disaster can also utilize this cellular connectivity, so you may not even need to physically connect this storage at the primary site (now defunct due to the disaster) to your network. The data you need to perform the complete no-data-loss recovery may already be transferred over to the system at the remote site and be waiting for you. In any case, you have access to the data—and all this comes without the need to invest in synchronous replication and expensive network infrastructure.

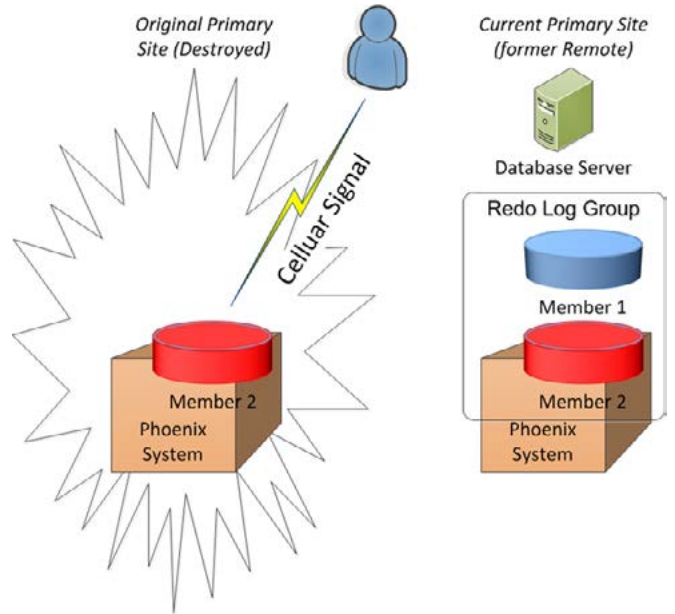


Figure 2

### Example

Let's see how it works with a hands-on approach inside a lab. I have two hosts:

	Host	Database
Primary	e02lora	adb
Standby	e02rora1	adbsb

The storage is ASM, not a filesystem. Replication in a filesystem is relatively simple, so I deliberately chose it to see if any ASM nuances are coming up. The Data Guard is configured with maximum performance, so there is no need for an expensive, fast low latency network infrastructure. The setup has an Axxana Phoenix system, which consists of three components:

- Black Box—the “special” protected storage system.
- Collector—a small server at the primary site that collects the changes to the files under the Axxana protection. In this case, they are controlfile, redo logs, and archived logs. I don't need to place any database files there because they are useless for recovery. Keeping this small set of files also makes it possible to put the protected files of many databases—not just one—inside a single Black Box.
- Recoverer—a small server at the standby site that receives the changes from the Black Box and keeps the files up to date when disaster occurs.

### Setting Up

First I ensured that the managed recovery process is running and the standby database is in MOUNTED mode.

Then I created a test schema in the production database.

```
e02lora$ sqlplus / as sysdba
SQL> grant create session, create table, create procedure, unlimited tablespace to
arup identified by arup;
```

Then I created some objects and data inside this schema.

```
SQL> connect arup/arup
SQL> create table T (c1 number, c2 timestamp);
SQL> create index IN_T ON T (C1);
SQL> create or replace procedure INS_T (p_c1 in number, p_commit in number) as
2 begin
3 for i in 1..p_c1 loop
4 insert into T values (i,current_timestamp);
```



```

5 if mod(i,p_commit)=0 then
6 commit;
7 end if;
8 end loop;
9 end;
10 /

SQL> create or replace procedure DEL_T (p_commit in number) as
2 begin
3 delete T where mod(C1, p_commit)=5;
4 commit;
5 end;
6 /
SQL> create or replace procedure UPD_T (p_commit number) as
2 begin
3 update T set C1 =-1 where mod(C1, p_commit)=3;
4 commit;
5 end;
6 /

SQL> exec INS_T (400000, 5000);

SQL> commit;

```

After creating the test data, I switched a few log files and waited a bit to make sure the changes are replicated to the standby.

## Simulating a Failure

Now to simulate a disconnected network. I didn't have any special tools with me, so I did the next best thing: I disabled the Ethernet interface.

```

e02rora$ su -
Password:
e02rora$ ifdown eth0

```

At this point the standby host will not receive the updates. All the changes made to the primary database will be located at the primary site only. I canceled the recovery process first.

```
SQL> ALTER DATABASE RECOVER MANAGED STANDBY DATABASE CANCEL;
```

I made a few changes to the primary data.

```

SQL> connect arup/arup
SQL> exec DEL_T (500);
SQL> exec UPD_T (500);

SQL> conn / as sysdba
SQL> alter system switch logfile;

SQL> conn arup/arup
SQL> select count(*) from T

COUNT(1)
-----
399200

```

The table has only 399,200 records. Since the Managed Recovery Process is stopped, these final changes will not be replicated to the standby. I can confirm that by opening the standby in read-only mode and checking the actual tables.

On the standby host:

```

SQL> conn / as sysdba
SQL> ALTER DATABASE OPEN READ ONLY;

Database altered.

SQL> SELECT COUNT (1) FROM ARUP.T;

COUNT(1)
-----
400000

```

The standby has all 400,000 rows, compared to 399,200 rows in the primary. The discrepancy is due to the unpropagated changes to the standby. At this point I simulate a failure in the primary by killing the pmon process.

```

e02lora$ ps -aef|grep pmon
UID PID PPID TTY STIME COMMAND
oracle 10184 10564 ptYO 16:26:17 pmon
e02lora$ kill -9 10184

```

The primary database is down. The data in the redo logs is lost as well, since they have not been propagated to the standby system yet.

## Performing Recovery

Now that we have simulated a disaster, let's see how to recover from it. Remember, since the Data Guard configuration is maximum performance, the database can only be recovered up to the most recent log entry. With Axxana software, however, there is additional data that can be pulled from the Black Box (the storage unit at the primary site that would not be destroyed). But how do *you*—the DBA—know which files are available at the standby site, which files are still left at the primary site and, most important, how to get those primary site files into the standby host? It gets even more complicated: since the Black Box is just a storage device, you have to mount the volumes and mount the ASM disks, etc. This may be fun when creating a brand new database, but it's definitely not so when you are under the gun to recover the database and bring your business online.

No worries. This is where the Axxana software comes to the rescue. I spun up the provided shell script at the standby site. This script contacts the Black Box at the primary, pulls the needed data, and completes the creation of the necessary files at the standby site. Once the files are at the standby site, all you have to do is to perform the typical managed standby database recovery to complete the recovery process. The best part of all? The script even gives you a step-by-step instruction sheet along with specific filenames that you can copy and paste when Oracle prompts for it. Here is how I call the script and the resultant output:

```

root@e02rora1 AxxRecovery# ./AxxRecovery.sh
Logging to '/home/oracle/AxxRecovery/logs/axxana.debug.log'
Calculating Optimization Parameters [done]
Attempting to connect to Axxana's Recoverer [done]

Perform Failover?

Warning!
This operation terminates the Axxana protection for all DBs.

1) OK
2) Cancel & Exit

```

At this point the script pauses and asks me for confirmation. I enter "1" and the script continues:

```

Requesting Axxana's Recoverer to 'FAILOVER' [done]

Recoverer communicates with the Black Box (ENTER_FAILOVER_START) [done]
Recoverer communicates with the Black Box (ENTER_FAILOVER_BBx) [done]
Recoverer communicates with the Black Box (ENTER_FAILOVER_CONNECT) [done]

Recoverer is in FAILOVER mode

Transferring required logs files of the database 'adb'.

The script shows me the names of the files along with their respective transfer status and the %age progress.

```

FILE NAME	Size in MBytes	Stage	Progress
ADB-group_3.258.902686913	50 (0)	Completed	100%
ADB-thread_1_seq_111.266.902852033	46 (46)	Completed	100%
ADB-group_2.257.902686911	50 (50)	Completed	100%
ADB-1454516139530A12435456XT_112.C	11 (11)	Completed	100%

(continued on page 26)

# Many Thanks to Our Sponsors

**N**oCOUG 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
- 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**

(continued from page 24)

```
ADB-group_1.256.902686901 | 50 (50) | Completed | 100%
=====
Total: | 208 (159) | 5 of 5 files recovered

All required log files exist at '/home/oracle/AxxRecovery/axx_recovery_files'

Preparing user recovery scripts ...

You can safely perform the Oracle RPO=0 recovery process

Please follow the RPO=0 recovery instructions that are described in the file '/home/oracle/AxxRecovery/axx_required_files/recovery_instructions.txt'
```

As I mentioned, the script creates a detailed step-by-step instruction sheet to be followed for the standby recovery. I am actually glad that it does not perform a recovery automatically. That is one process you want to watch so you can proceed with caution. You probably have only one chance at it, and rushing through it may force you to make missteps. But at the same time, you want to think as little as possible under such stressful conditions, so the detailed instruction sheet comes in handy. The last line shows the location of the instruction files. Here is what the file looks like in my case, after removing some banner items:

```
Step 1) Output setup
=====

Please log into the standby database (as sys user)
and then run the following commands at the SQL prompt:

SQL>
SET SERVEROUTPUT ON
SET LINES 999
SET PAGES 0

Step 2) Recovery using archive log files
=====

Please run the following commands at the SQL prompt:
* (You can safely ignore the following ORA Error 'Managed Standby Recovery not active')

SQL> ALTER DATABASE RECOVER MANAGED STANDBY DATABASE CANCEL;

SQL> RECOVER STANDBY DATABASE UNTIL CANCEL;

-- when requested, feed the following file(s):

+DATA/adbsb/archive/2016_02_03/thread_1_seq_110.382.902852027
/home/oracle/AxxRecovery/axx_recovery_files/E02LORA1/BBX/ADB/ARCHIVELOG/2016_02_03/
thread_1_seq_111.266.902852033

-- finally enter 'Cancel'

SQL> CANCEL

Step 3) Switching to primary control file
=====
Please run the following commands at the SQL prompt:

SQL> SHUTDOWN IMMEDIATE
SQL> STARTUP NOMOUNT
SQL> ALTER SYSTEM SET CONTROL_FILES='/home/oracle/AxxRecovery/axx_required_files/14545
16139530A12435456XT_112.C' SCOPE=SPFILE;
SQL> SHUTDOWN IMMEDIATE
SQL> STARTUP MOUNT

Step 4) Renaming data and redo log file names
=====
Please run the following SQL statement

SQL> @/home/oracle/AxxRecovery/axx_required_files/logAndDateFileRename.sql

Step 5) Recovery using primary control file
=====
Please run the following command at the SQL prompt:

SQL> RECOVER DATABASE UNTIL CANCEL USING BACKUP CONTROLFILE
-- when requested, feed the following file(s):

/home/oracle/AxxRecovery/axx_recovery_files/E02LORA1/BBX/ADB/ONLINELOG/
```

```
group_1.256.902686901
/home/oracle/AxxRecovery/axx_recovery_files/E02LORA1/BBX/ADB/ONLINELOG/
group_2.257.902686911
/home/oracle/AxxRecovery/axx_recovery_files/E02LORA1/BBX/ADB/ONLINELOG/
group_3.258.902686913

-- You should now see a message saying 'Media Recovery complete'

Step 6) Open the Standby Database
=====
Please run the following commands at the SQL prompt:

SQL> ALTER DATABASE OPEN RESETLOGS;
```

Well, it's dumbed down enough for those stressful moments associated with a standby database recovery, down to even the set page size commands in SQL\*Plus—quite easy to forget in stressful situations. Note that it shows the time it was generated at the very beginning, in a non-U.S. date format as dd/mm/yyyy. Anyway, I followed the instructions step by step. Step 4 requires some attention. It shows how to change the names of the redo logs and datafiles after the switchover.

For the sake of brevity I don't want to show the entire output. Here is an excerpt from the tail end of the activity:

```
Specify log: {suggested | filename | AUTO | CANCEL}
/home/oracle/AxxRecovery/axx_recovery_files/E02LORA1/BBX/ADB/ONLINELOG/
group_2.257.902686911
ORA-00279: change 5172890 generated at 02/03/2016 16:16:56 needed for thread 1
ORA-00289: suggestion : +DATA
ORA-00280: change 5172890 for thread 1 is in sequence #114
ORA-00278: log file '/home/oracle/AxxRecovery/axx_recovery_files/E02LORA1/BBX/ADB/ON-
LINELOG/group_2.257.902686911' no longer needed for this recovery

Specify log: {suggested | filename | AUTO | CANCEL}
/home/oracle/AxxRecovery/axx_recovery_files/E02LORA1/BBX/ADB/ONLINELOG/
group_3.258.902686913
Log applied.
Media recovery complete.

SQL> ALTER DATABASE OPEN RESETLOGS;

Database altered.

SQL> SELECT COUNT (1) FROM ARUP.T;

COUNT(1)
-----
399200
```

Voila! The output shows “Media recovery complete.” The count is 399,200—the same as the number in the production database. The recovery process got those last changes, and I accomplished my objective without synchronous replication.

## Conclusion

Recovery without data loss has always been dependent on a high throughput, a low latency, and an ultra-reliable network. Unfortunately, the cost of this infrastructure often precludes the use of no-data-loss setups in organizations. The Axxana solution is innovative in the sense that it addresses the issue with a completely out-of-the-box solution, slashing costs so dramatically that most corporations will be able to accomplish recovery with no data loss in Data Guard setups. ▲

*Arup Nanda is an award-winning technical leader, an architect, a planner, an Oracle technologist, the author of five books and 500+ articles, a speaker at 300+ sessions, the host of training seminars in 22 countries, a mentor, a friend, a father, and a husband—not necessarily in that order. He was named Oracle's DBA of the Year in 2003 and Architect of the Year in 2012. Arup blogs at [arup.blogspot.com](http://arup.blogspot.com) and tweets at [@ArupNanda](https://twitter.com/ArupNanda). He is an Oracle ACE Director and a member of the Oak Table Network.* ©Arup Nanda



# 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.



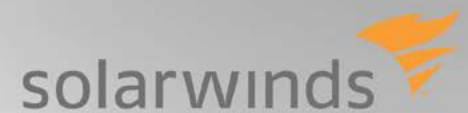
CALL 1 - 8 8 8 - 6 4 8 - 0 5 0 0 TO DISCUSS A SERVICE PLAN

**NoCOUG**

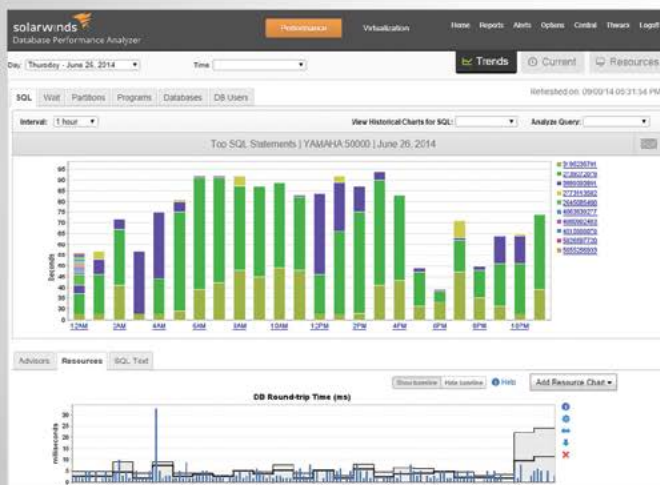
P.O. Box 3282

Danville, CA 94526

**RETURN SERVICE REQUESTED**



# See what's new in Database Performance Analyzer 9.0



- Storage I/O analysis for better understanding of storage performance
- Resource metric baselines to identify normal operating thresholds
- Resource Alerts for full-alert coverage
- SQL statement analysis with expert tuning advice

Download free trial at: [solarwinds.com/dpa-download](http://solarwinds.com/dpa-download)