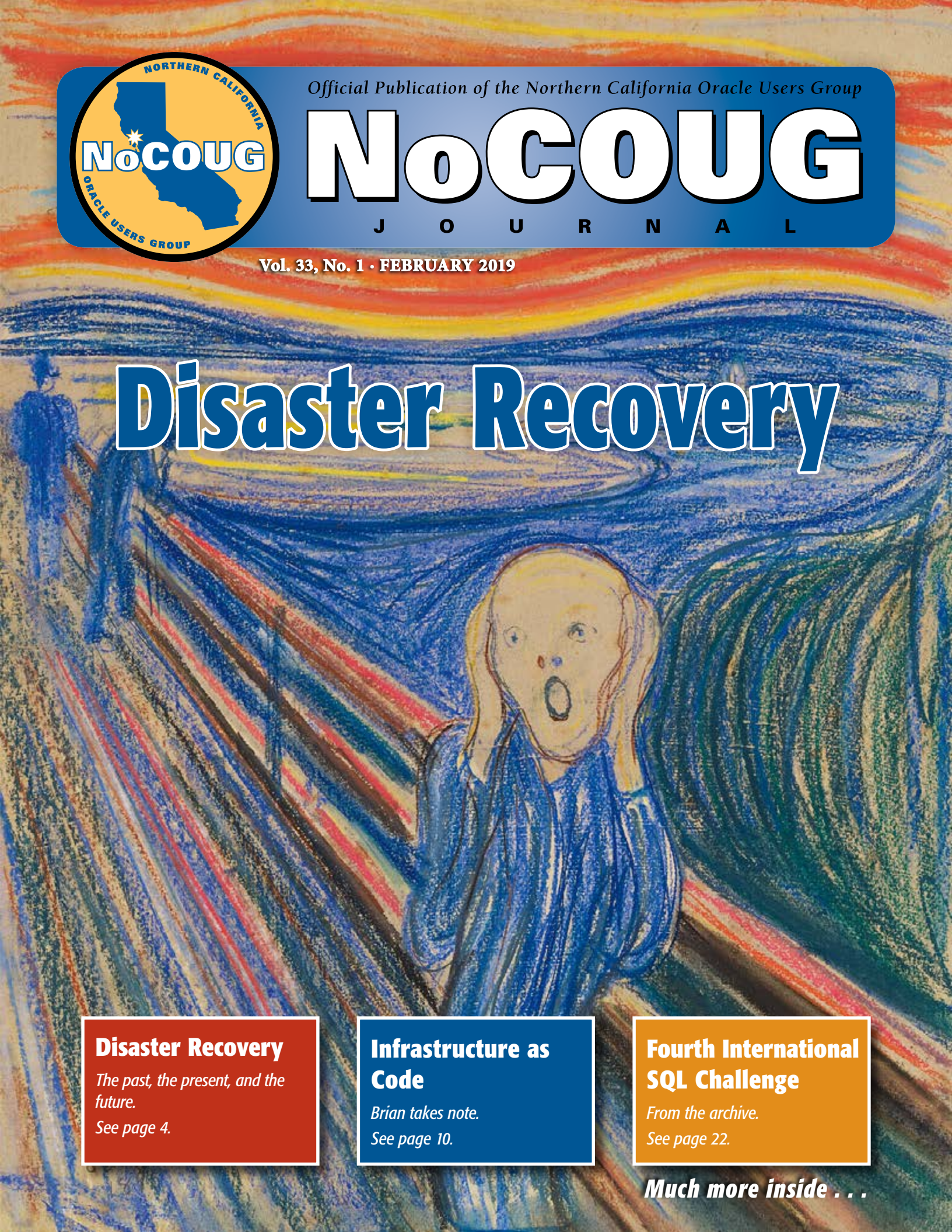
Official Publication of the Northern California Oracle Users Group

NoCOUG JOURNAL

Vol. 33, No. 1 · FEBRUARY 2019

Disaster Recovery



Much more inside . . .

The New Phoenix by Axxana

For Oracle Databases and Applications

Zero Data Loss at Unlimited Distances

Synchronous Protection at Maximum Performance

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

AXXANA

BUILT TO LAST

www.axxana.com

ORACLE Gold Partner

ORACLE BIG DATA APPLIANCE READY

ORACLE DATABASE READY

ORACLE EXADATA READY

ORACLE LINUX READY

Professionals at Work

First there are the IT professionals who write for the *Journal*. A very special mention goes to Brian Hitchcock, who has written dozens of book reviews over a 12-year period.

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, the *Journal* is printed and shipped to us. ▲

Table of Contents

Publication Notices and Submission Format

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

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

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

2019 NoCOUG Board

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

Disaster Recovery: Past, Present, and Future

by Alex Winokur

Alex Winokur

Introduction

Disaster recovery is now on the list of top concerns of every CIO. In this article we review the evolution of the disaster recovery landscape, from its inception until today. We look at the current understanding of disaster behavior and, as a result, the disaster recovery processes. We also try to cautiously anticipate the future, outlining the main challenges associated with disaster recovery.

The Past

The computer industry is relatively young. The first commercial computers appeared somewhere in the 1950s—not even 70 years ago. The history of disaster recovery (DR) is even younger. Table 1 outlines the appearance of the various technologies necessary to construct a modern DR solution.

From Magnetic Tapes to Data Networks

The first magnetic tapes for computers were used as input/output devices. That is, input was punched onto punch cards that were then stored offline to magnetic tapes. Later, UNIVAC I, one of the first commercial computers, was able to read these tapes and process their data. Later still, output was similarly directed to magnetic tapes that were connected offline to printers for printing purposes. Tapes began to be used as a backup medium only after 1954, with the introduction of the mass storage device (RAMAC).

Although modern wide-area communication networks date back to 1974, data has been transmitted over long-distance communication lines since 1837 via telegraphy systems. These telegraphy communications have since evolved to data transmission over telephone lines using modems.

Modems were widely introduced in 1958 to connect United States air defense systems; however, their throughput was very low compared to what we have today. The FAA clustered system deployed communication that was designed for computers to communicate with their peripherals (e.g., tapes). Local-area networks (LANs) as we now know them had not been invented yet.

Technology	Introduced	Comments
Long-distance data transmission over electrical lines	1837	First commercial telegraphy systems
Magnetic tapes	1928	Magnetic tapes for voice recording; first used for computers with UNIVAC I in 1952
Uninterruptible power supply (UPS)	1934	Grant date of rotary UPS patent
First commercial computer in the United States	1952	UNIVersal Automatic Computer (UNIVAC I)
First mass storage system	1956	IBM 305 RAMAC (i.e., Random Access Method of Accounting and Control) with capacity of 5 MB
First mass production computer	1964	IBM 360
First commercially clustered system	1970	IBM 360 clustered system for the Federal Aviation Administration (FAA)
First commercial data network	1974	X.25 wide-area network (WAN) protocol developed for telephone companies

Table 1. Early history of DR technology development

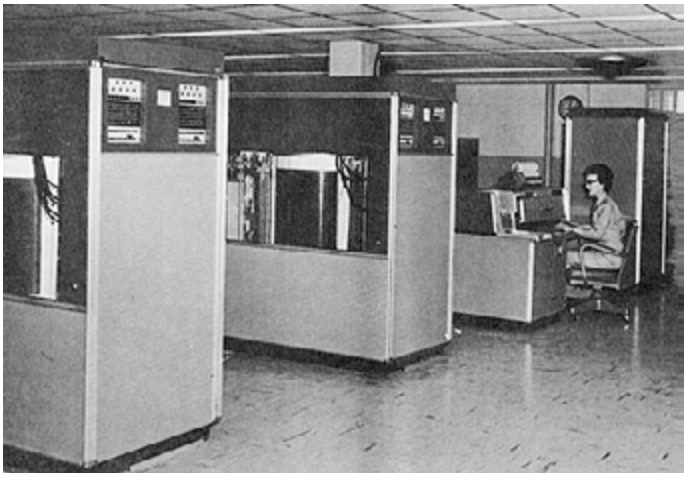


Figure 1. IBM 305 RAMAC at U.S. Army Red River Arsenal. Foreground: two 350 disk drives. Background: 380 console and 305 processing unit

Early Attempts at Disaster Recovery

It wasn't until the 1970s that concerns about disaster recovery started to emerge. In that decade, the deployment of IBM 360 computers reached a critical mass, and they became a vital part of almost every organization. Until the mid-1970s, the perception was that if a computer failed, it would be possible to fail back to paper-based operation as was done in the 1960s. However, the widespread rise of digital technologies in the 1970s led to a corresponding increase in technological failures on one hand, while on the other hand, theoretical calculations, backed by real-world evidence, showed that switching back to paper-based work was not practical.

The emergence of terrorist groups in Europe like the Red Brigades in Italy and the Baader-Meinhof Group in Germany further escalated concerns about the disruption of computer operations. These organizations specifically targeted financial institutions. The fear was that one of them would try to blow up a bank's data centers.

At that time, communication networks were in their infancy, and replication between data centers was not practical.

Parallel workloads. IBM came up with the idea to use the FAA clustering technology to build two adjoining computer rooms that were separated by a steel wall and had one node cluster in each room. The idea was to run the same workload twice and to be able to immediately fail over from one system to the other in case one system was attacked. A closer analysis revealed that in a case of a terror attack, the only surviving object would be the steel wall, so the plan was abandoned.

Hot, warm, and cold sites. The inability of computer vendors (IBM was the main vendor at the time) to provide an adequate DR solution made way for dedicated DR firms like Sungard AS to provide hot, warm, or cold alternate sites. Hot sites, for example, were duplicates of the primary site; they independently ran the same workloads as the primary site, as communication between the two sites was not available at the time. Cold sites served as repositories for backup tapes. Following a disaster at the primary site, operations would resume at the cold site by allocating equipment, executing a restore from backup operations, and restarting the applications. Warms sites were a compromise between a hot site and a cold site. These sites had hardware and connectivity already established; however, recovery was still done by restoring the data from backups before the applications could be restarted.

Backups and high availability. The major advances in the 1980s were around backups and high availability. On the backup side, regulations requiring banks to have a testable backup plan were enacted. These were probably the first DR regulations to be imposed on banks; many more followed through the years. On the high availability side, Digital Equipment Corporation (DEC) made the most significant advances in LAN communications (DECnet) and clustering (VAXcluster).

The Turning Point

On February 26, 1993, the first bombing of the World Trade Center (WTC) took place. This was probably the most significant event shaping the disaster recovery solution architectures of today. People realized that the existing disaster recovery solutions, which were mainly based on tape backups, were not sufficient. They understood that too much data would be lost in a real disaster event. By this time, communication networks had matured, and EMC became the first to introduce a storage-to-storage replication software called Symmetrix Remote Data Facility (SRDF).

The first few years of the 21st century will always be remembered for the events of September 11, 2001—the date of the complete annihilation of the World Trade Center. Government, industry, and technology leaders realized then that some disasters can affect the whole nation, and therefore DR had to be taken much more seriously. In particular, the attack demonstrated that existing DR plans were not adequate to cope with disasters of such magnitude. The notion of local, regional, and nationwide disasters crystalized, and it was realized that recovery methods that work for local disasters don't necessarily work for regional ones.

SEC Directives

In response, the Securities Exchange Commission (SEC) issued a set of very specific directives in the form of the "Interagency Paper on Sound Practices to Strengthen the Resilience of the U.S." These regulations, still intact today, bind all financial institutions. The DR practices that were codified in the SEC regulations quickly propagated to other sectors, and disaster recovery became a major area of activity for all organizations relying on IT infrastructure.

The essence of these regulations is as follows:

1. The economic stance of the United States cannot be compromised under any circumstance.

Behind the Scenes at IBM

At the beginning of the 1990s, I was with IBM's research division. At the time, we were busy developing an innovative solution to shorten the backup window, as backups were the foundation for all DR, and the existing backup windows (dead hours during the night) started to be insufficient to complete the daily backup. The solution, called "concurrent copy," was the ancestor of all snapshotting technologies, and it was the first intelligent function running within the storage subsystem. The WTC event in 1993 left IBM fighting the "yesterday battles" of developing a backup solution while giving EMC the opportunity to introduce storage-based replication and become the leader in the storage industry.

2. Relevant financial institutions are obliged to correctly, without any data loss, resume operations by the next business day following a disaster.
3. Alternate disaster recovery sites must use different physical infrastructure (electricity, communication, water, transportation, and so on) than the primary site.

Note that Requirements 2 and 3 above are somewhat contradictory. Requirement 2 necessitates synchronous replication to facilitate zero data loss, while Requirement 3 basically dictates long distances between sites—thereby making the use of synchronous replication impossible. This contradiction is not addressed within the regulations and is left to each implementer to deal with at its own discretion.

The secret to resolving this contradiction lies in the ability to reconstruct missing data if or when data loss occurs. The nature of most critical data is such that there is always at least one other instance of this data somewhere in the universe. The trick is to locate it, determine how much of it is missing in the database, and augment the surviving instance of the database with this data. This process is called "data reconciliation," and it has become a critical component of modern disaster recovery. [See The Data Reconciliation Process sidebar.]

The Present

The second decade of the 21st century has been characterized by new types of disaster threats, including sophisticated cyberattacks and extreme weather hazards caused by global warming. It is also characterized by new DR paradigms, like DR automation, disaster recovery as a service (DRaaS), and active-active configurations.

These new technologies are for the most part still in their infancy. DR automation tools attempt to orchestrate a complete site recovery through invocation of one "site failover" command, but they are still very limited in scope. A typical tool in this category is the VMware Site Recovery Manager (SRM). DRaaS attempts to reduce the cost of DR-compliant installation by locating the secondary site in the cloud. The new active-active configurations try to reduce equipment costs and recovery time by utilizing techniques that are used in the context of high availability—that is, to recover from a component failure rather than a complete site failure.

Disasters vs. Catastrophes

The following definitions of disasters and disaster recovery have been refined over the years to make a clear distinction between the two main aspects of business continuity: high availability protection and disaster recovery. This distinction is important because it crystalizes the difference between disaster recovery and a single component failure recovery covered by highly available configurations, and in doing so also accounts for the limitations of using active-active solutions for DR.

A "disaster" in the context of IT is either a significant adverse event that causes an inability to continue operation of the data center or a data loss event where recovery cannot be based on equipment at the data center. In essence, "disaster recovery" is a set of procedures aimed at resuming operations following a disaster by failing over to a secondary site.

From a DR procedures perspective, it is customary to classify disasters into 1) regional disasters like weather hazards, earthquakes, floods, and electricity blackouts, and 2) local disasters like local fires, onsite electrical failures, and cooling system failures.

Over the years, I have also noticed a third, independent classification of disasters. Disasters can also be classified as catastrophes. In principal, a "catastrophe" is a disastrous event where in the course of a disaster, something unexpected happens that causes the disaster recovery plans to dramatically miss their service level agreement (SLA); that is, they typically exceed their RTO.

The Data Reconciliation Process

If data is lost as a result of a disaster, the database becomes misaligned with the real world. The longer this misalignment exists, the greater the risk of application inconsistencies and operational disruptions. Therefore, following a disaster, it is very important to align back the databases with the real world as soon as possible. This process of alignment is called "data reconciliation."

The reconciliation process has two important characteristics:

1. It is based on the fact that the data lost in a disaster exists somewhere in the real word, and thus it can be reconstructed in the database.
2. The duration and complexity of the reconciliation is proportional to the recovery point objective (RPO); that is, it's proportional to the amount of data lost.

One of the most common misconceptions in disaster recovery is that RPO (for example, RPO = 5) refers to how many minutes of data the organization is willing to lose. What RPO really means is that the organization must be able to reconstruct and reconsolidate (i.e., reconcile) that last five minutes of missing data. Note that the higher the RPO (and therefore, the greater the data loss), the longer the recovery time objective (RTO) and the costlier the reconciliation process. Catastrophes typically occur when RPO is compromised and the reconciliation process takes much longer.

In most cases, the reconciliation process is quite complicated, consisting of time-consuming processes to identify the data gaps and then resubmitting the missing transactions to realign the databases with real-world status. This is a costly, mainly manual, error-prone process that greatly prolongs the recovery time of the systems and magnifies risks associated with downtime. ▲

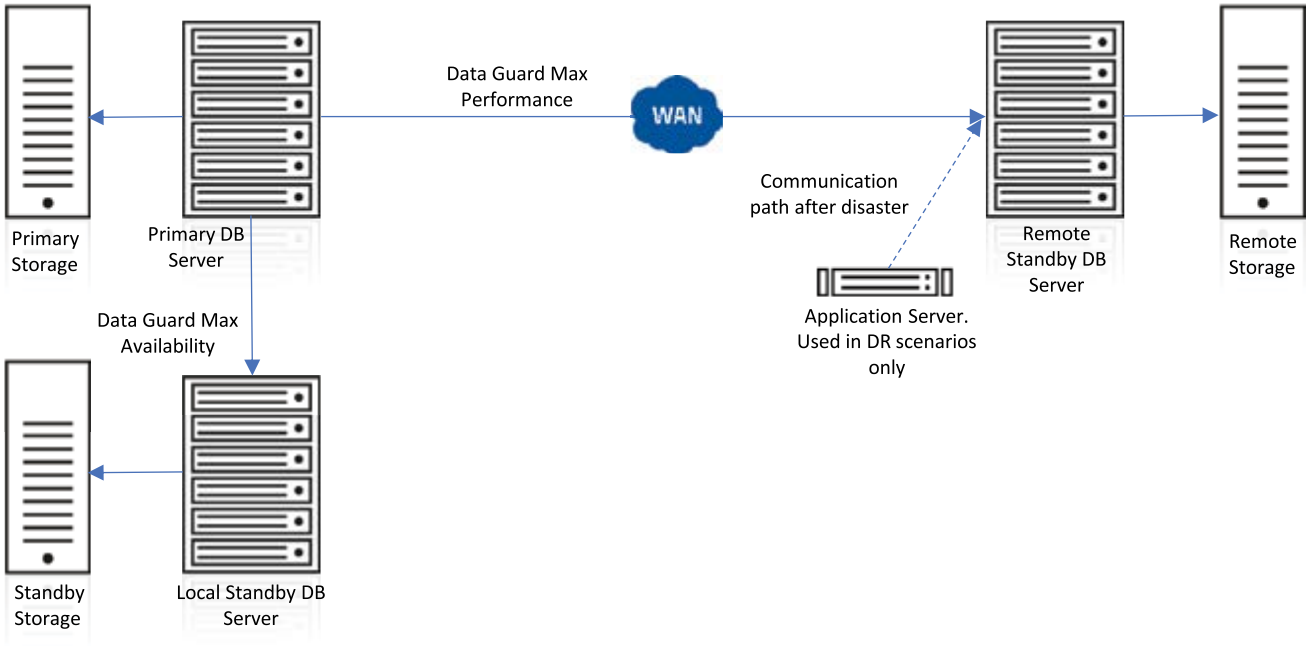


Figure 2. Typical DR configuration

When DR procedures go as planned for regional and local disasters, organizations fail over to a secondary site and resume operations within pre-determined parameters for recovery time (i.e., RTO) and data loss (i.e., RPO). The organization's SLAs, business continuity plans, and risk management goals align with these objectives, and the organization is prepared to accept the consequent outcomes. A catastrophe occurs when these SLAs are compromised.

Catastrophes can also result from simply failing to execute the DR procedures as specified, typically due to human errors. However, for the sake of this article, let's be optimistic and assume that DR plans are always executed flawlessly. We shall concentrate only on unexpected events that are beyond human control.

Most of the disaster events that have been reported in the news recently (for example, the Amazon Prime Day outage in July 2018 and the British Airways bank holiday outage in 2017) have been catastrophes related to local disasters. If DR could have been properly applied to the disruptions at hand, nobody would have noticed that there had been a problem, as the DR procedures were designed to provide almost zero recovery time and hence zero downtime.

The following two examples provide a closer look at how catastrophes occur.

9/11. Following the September 11 attack, several banks experienced major outages. Most of them had a fully equipped alternate site in Jersey City—no more than five miles away from their primary site. However, the failover failed miserably because the banks' DR plans called for critical personnel to travel from their primary site to their alternate site, but nobody could get out of Manhattan.

A data center power failure during a major snowstorm in New England. Under normal DR operations at this organization, the data was synchronously replicated to an alternate site. However, 90 seconds prior to a power failure at the primary site, the central communication switch in the area lost power too, which cut all WAN communications. As a result, the primary site continued to produce data for 90 seconds without replication to the secondary site; that is, until it experienced the power failure. When it finally failed over to the alternate site, 90 seconds of transactions were missing; and because the DR procedures were not designed to address recovery where data loss has occurred, the organization experienced catastrophic downtime.

The common theme of these two examples is that in addition to the disaster at the data center there was some additional—unrelated—malfunction that turned a "normal" disaster into a catastrophe. In the first case, it was a transportation failure; in the second case, it was a central switch failure. Interestingly, both failures occurred to infrastructure elements that were completely outside the control of the organizations that experienced the catastrophe. Failure of the surrounding infrastructure is indeed one of the major causes for catastrophes. This is also the reason why the SEC regulations put so much emphasis on infrastructure separation between the primary and secondary data center.

Current DR Configurations

In this section, I've included examples of two traditional DR configurations that separate the primary and secondary center, as stipulated by the SEC. These configurations have predominated in the past decade or so, but they cannot ensure zero data loss in rolling disasters and other disaster scenarios, and they are being challenged by new paradigms such as that introduced by Axxana's Phoenix. While a detailed discussion would be outside the scope of this article, suffice it to say that Axxana's Phoenix makes it possible to avoid catastrophes such as those just described—something that is not possible with traditional synchronous replication models.

Typical DR configuration. Figure 2 presents a typical disaster recovery configuration. It consists of a primary site, a remote site, and another set of equipment at the primary site, which serves as a local standby.

The main goal of the local standby installation is to provide redundancy to the production equipment at the primary site. The standby equipment is designed to provide nearly seamless failover capabilities in case of an equipment failure—not in a di-

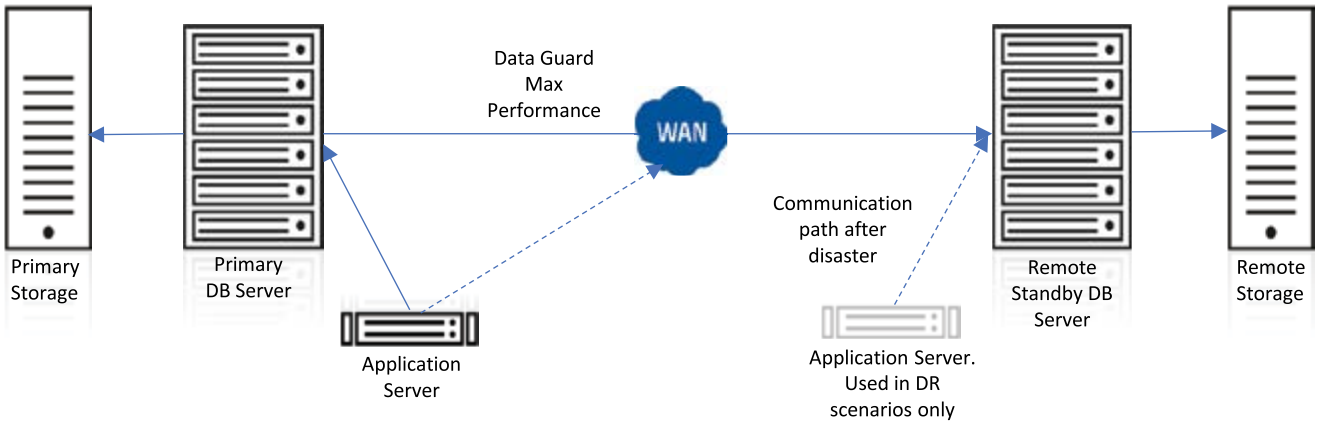


Figure 3. DR cost-saving configuration

saster scenario. The remote site is typically located at a distance that guarantees infrastructure independence (communication, power, water, transportation, etc.), to minimize the chances of a catastrophe. It should be noted that the typical DR configuration is very wasteful. Essentially, an organization has to triple the cost of equipment and software licenses—not to mention the increased personnel costs and the cost of high-bandwidth communications—to support the configuration of Figure 2.

Traditional ideal DR configuration. Figure 3 illustrates the traditional ideal DR configuration. Here, the remote site serves both for DR purposes and high availability purposes. Such configurations are sometimes realized in the form of extended clusters like Oracle RAC One Node on Extended Distance. Although traditionally considered the ideal, they are a trade-off between survivability, performance, and cost. The organization saves on the cost of one set of equipment and licenses, but it compromises survivability and performance. That's because the two sites have to be in close proximity to share the same infrastructure, so they are more likely to both be affected by the same regional disasters; at the same time, performance is compromised due to the increased latency caused by separating the two cluster nodes from each other.

True zero-data-loss configuration. Figure 4 represents a cost-saving solution with Axxana's Phoenix. In case of a disaster, Axxana's Phoenix provides a zero-data-loss recovery to any distance. So, with the help of Oracle's high availability support (Fast-Start Failover and Transparent Application Failover), Phoenix provides functionality very similar to extended cluster functionality. With Phoenix, however, it can be implemented over much longer distances and with much smaller latency, providing true cost savings over the typical configuration shown in Figure 3.

The Future

In my view, the future is going to be a constant race between new threats and new disaster recovery technologies.

New Threats and Challenges

In terms of threats, global warming creates new weather hazards that are fiercer, more frequent, and far more damaging than in the past—and in areas that have not previously experienced such events. Terror attacks are on the rise, thereby increasing threats to national infrastructures (potential regional disasters). Cyberattacks—in particular ransomware, which destroys data—are a new type of disaster. They are becoming more prolific, more sophisticated and targeted, and more damaging.

At the same time, data center operations are becoming more and more complex. Data is growing exponentially. Instead of getting simpler and more robust, infrastructures are getting more

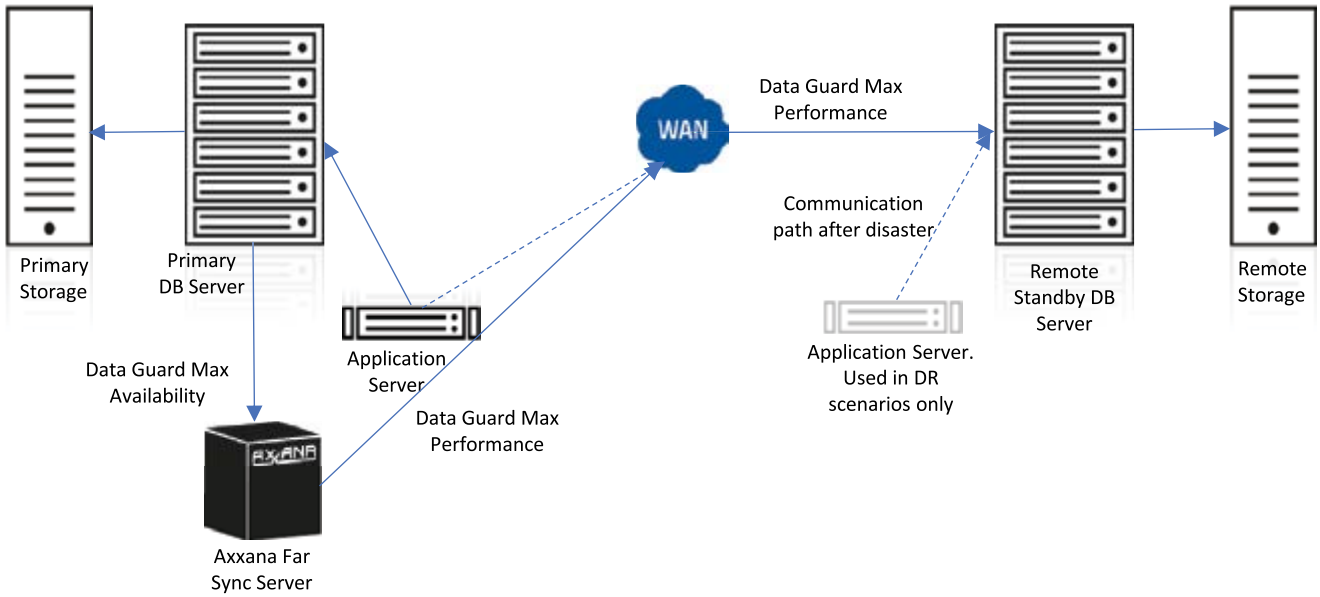


Figure 4. Consolidation of DR and high availability configurations with Axxana's Phoenix

"On one hand, a traditional zero-data-loss DR configuration requires organizations to implement and manage not only a primary site but also a local standby and remote standby; doing so essentially triples the costs of critical infrastructure, even though only one-third of it (the primary site) is utilized in normal operation."

diversified and fragmented. In addition to legacy architectures that aren't likely to be replaced for many years, new paradigms like public, hybrid, and private clouds; hyperconverged systems; and software-defined storage are being introduced. Adding to that are an increasing scarcity of qualified IT workers and economic pressures that limit IT spending. Combined, these factors contribute to data center vulnerabilities and to more frequent events requiring disaster recovery.

So, this is on the threat side. What is there for us on the technology side?

New Technologies

Axxana's Phoenix is at the forefront of new technologies that guarantee zero data loss in any DR configuration (and therefore ensure rapid recovery), but I will leave the details of our solution to a different discussion.

AI and machine learning. Apart from Axxana's Phoenix, the most promising technologies on the horizon revolve around artificial intelligence (AI) and machine learning. These technologies enable DR processes to become more "intelligent," efficient, and predictive by using data from DR tests, real-world DR operations, and past disaster scenarios; in doing so, disaster recovery processes can be designed to better anticipate and respond to unexpected catastrophic events. These technologies, if correctly applied, can shorten RTO and significantly increase the success rate of disaster recovery operations. The following examples suggest only a few of their potential applications in various phases of disaster recovery:

- They can be applied to improve the DR planning stage, resulting in more robust DR procedures.
- When a disaster occurs, they can assist in the assessment phase to provide faster and better decision-making regarding failover operations.
- They can significantly improve the failover process itself, monitoring its progress and automatically invoking corrective actions if something goes wrong.

When these technologies mature, the entire DR cycle from planning to execution can be fully automated. They carry the promise of much better outcomes than processes done by humans because they can process and better "comprehend" far more data in very complex environments with hundreds of components and thousands of different failure sequences and disaster scenarios.

New models of protection against cyberattacks. The second front where technology can greatly help with disaster recovery is on the cyberattack front. Right now, organizations are spending millions of dollars on various intrusion prevention, intrusion detection, and asset protection tools. The evolution should be from protecting individual organizations to protecting the global network. Instead of fragmented, per-organization defense measures, the global communication network should be "cleaned" of threats that can create data center disasters. So, for example, phishing attacks that would compromise a data center's access control mechanisms should be filtered out in the network—or in the cloud—instead of reaching and being filtered at the end points.

Conclusion

Disaster recovery has come a long way—from naive tape backup operations to complex site recovery operations and data reconciliation techniques. The expenses associated with disaster protection don't seem to go down over the years; on the contrary, they are only increasing.

The major challenge of DR readiness is in its return on investment (ROI) model. On one hand, a traditional zero-data-loss DR configuration requires organizations to implement and manage not only a primary site but also a local standby and remote standby; doing so essentially triples the costs of critical infrastructure, even though only one-third of it (the primary site) is utilized in normal operation.

On the other hand, if a disaster occurs and the proper measures are not in place, the financial losses, reputation damage, regulatory backlash, and other risks can be devastating. As organizations move into the future, they will need to address the increasing volumes and criticality of data. The right disaster recovery solution will no longer be an option; it will be essential for mitigating risk, and ultimately, for staying in business. ▲

"On the other hand, if a disaster occurs and the proper measures are not in place, the financial losses, reputation damage, regulatory backlash, and other risks can be devastating. As organizations move into the future, they will need to address the increasing volumes and criticality of data. The right disaster recovery solution will no longer be an option; it will be essential for mitigating risk, and ultimately, for staying in business."

Infrastructure as Code—Managing Servers in the Cloud

Brian Hitchcock

Book notes by Brian Hitchcock

Details

Author: Kief Morris

ISBN-13: 978-1-491-92435-8

Publication Date: June 2016

Publisher: O'Reilly Media

Summary

This book introduced me to a lot of new ideas. I had heard of infrastructure as code, but I had no idea what the term really meant. There is a lot of good detail here that helps explain the impacts you will experience as you move to this new way of building and supporting IT systems.

Preface

Teams are using automated tools to build software and the infrastructure it runs on. The tools are described as "infrastructure as code" and operate on files that define servers and networking resources as well as other elements. These tools are used to process the files to build and maintain systems. The DevOps movement—a collaboration between software developers and software operations—manages infrastructure using a software development paradigm.

Many teams struggle to make the transition from classic infrastructure management, which I call "normal," to this new paradigm. The author explains his journey to the cloud in a section titled, "How I Learned to Stop Worrying and to Love the Cloud," a reference, I assume, to *Dr. Strangelove*. Starting with hosted systems, virtualization brought infrastructure as code into focus. The journey continued from rack servers supporting many services to VMware. With virtual servers, you could cleanly split each service onto its own VM. Old problems went away only to be replaced by new ones, which required a new way of thinking. Servers were so much easier to create; many more servers were created than anyone thought possible. The ability to build a new server in less than a day led to over 100 VMs running and an endless need for more storage to support them. The author compares this to Disney's "The Sorcerer's Apprentice" from *Fantasia*. In the animated classic, Mickey Mouse thinks he knows how to work the magic but quickly finds himself unable to control his environment. The author states that the number of VMs overwhelmed them. As things broke and were fixed, no record of the changes was made. It was tough to install software updates across all systems, because there are so many differences among them, with numerous combinations of versions and components. Tools that were new at the time, Puppet and Chef, didn't help clean up the existing mess. The existing systems were too different, causing any automated configuration attempt to fail. Using these new tools worked only for new servers.

As so often happens, it wasn't the cloud that fixed the problem—it was the move to the cloud, requiring the rebuilding of all existing systems, that allowed automated configuration to succeed. Starting fresh with controlled configurations for all servers also meant that any server could be rebuilt from scratch automatically. Learning the new ways was painful. Highly automated infrastructure presented many challenges.

The author then addresses why he wrote this book. Many teams are using cloud, virtualization, and automation tools, but they don't have the time to make it all work. The endless daily crises take all of their time. This book provides a practical vision for managing IT infrastructure. This book is for the usual IT suspects and software developers that need to manage their own infrastructure. The reader is assumed to have some experience with virtualization or Infrastructure as a Service (IaaS), and automation software such as Puppet, Chevy, and Ansible.

Chapter 1—Challenges and Principles

New technologies promise to transform IT infrastructure management, but many teams aren't seeing the benefits—for too many, they just see more of the same mess. So why infrastructure as code? All this cool new stuff should save time and effort, reducing the time spent on routine drudgery. On a personal note, my job is 100% routine drudgery, so I'm eager to learn about this new world. IT operations can't keep up with the daily demands and can't fix existing problems to keep everything from collapsing. Cloud and automation tools should make it much easier to make changes, but you have to set up the needed process to get the advertised benefits. Trying to apply the pre-cloud, pre-automation principles—where you planned a change and it took days to weeks to implement—won't work when an automated approach can make changes in minutes or less. The author points out that existing (i.e., legacy) change management is often ignored to get things done quickly. This legacy approach doesn't cope well with the speed of change possible with cloud and new tools. Infrastructure as code enters the picture to help.

We are told that in the "Iron Age" of IT, systems were tied directly to physical hardware. Lots of manual work was required to provision and maintain these systems. I'm quite fond of the

Iron Age, with all the breast plates and goofy helmets—and I especially like the horses. Making changes to such systems takes lots of planning and nothing gets done quickly.

So, what exactly is infrastructure as code? We are told that it is an approach to infrastructure automation—based on practices from software development—that supports a consistent process for provisioning and changing systems. Changes are made to system definitions, and the process of applying changes is automated. The central idea is that this approach treats infrastructure as if it were software and data. Now you can see how a version control system (VCS) could impose order on all configurations and changes. We are told that the most demanding environments, such as Amazon, Netflix, and Google, have proven that this approach works. This book's goal is to explain how to make all this happen and to warn us about the things that can go wrong.

Note that infrastructure as code is not just for the cloud. While it works really well in the cloud, it can be used to great benefit in worlds that are not in a cloud.

There are challenges, of course, with dynamic infrastructure that is created and controlled through software: server sprawl as too many servers get created, configuration drift as things get changed, and snowflake servers that are so fragile no one will touch them. A good war story is offered, explaining how different Perl versions among servers was the root cause of a major outage.

We then hear about fragile infrastructure—or, as the author describes it: don't touch that server, don't point at it, don't even look at it. Such servers are critical, but no one supporting them knows how they were built or what the impact of any change will be.

One of the principles of infrastructure as code is "cattle not pets," which means that servers, and other resources, are not sensitive beings to be nurtured but resources to be created, changed, and replaced all the time. The term "antifragility" is introduced, meaning "beyond robust." I think "robust" works well enough by itself.

I have some observations. Even if all this automation will work, how many IT persons live for the drama? Similarly, if your organization doesn't move away from routine, repetitive tasks, do they really value your time? One of the stated goals is to have users set up what they need through self-service interfaces. Users, in my opinion, can't be left to manage resources. Each one of them would take all available resources to process their personal email. Since you can, in theory, rebuild any server at any time, we are told that recovery from an outage will be quick. The assumption that you can recover quickly can be as bad as assuming failure can be prevented. The Titanic was not unsinkable. Continuous improvement also means continuous new issues. I agree that the transition to infrastructure as code will be difficult. I have no idea how to rebuild any of the systems I support.

I also wonder if everything about a complex system can be put in a text file. The discussion assumes this but does not address it. If one system is the data source for another system controlled by another group, how is this configuration handled? If the other group makes changes, do we record the new configuration or do they? Would all groups be forced into one single-version control system? We are told that it is difficult to write automated tests for existing legacy systems, but every existing system is a legacy system. Only if you are starting from scratch can you hope to build systems in this new way. And any time your organization acquires another company or different parts of your company are reorganized, you will have many new and different systems that will need to be integrated into your automated infrastructure process.

Chapter 2—Dynamic Infrastructure Platforms

This chapter describes different types of platforms that support provisioning and managing infrastructure resources. A dynamic infrastructure platform is a system that provides servers, storage, and networking resources in a way that can be allocated and managed programmatically. The usual list of platform types and who provides them is shown. A dynamic infrastructure platform is required to be programmable, on-demand, and self-service. Each of these requirements is discussed in detail. The sidebar showing how the NIST defines a cloud is worth reading. The key infrastructure resources provided by the platform are compute, storage, and networking. The types of dynamic infrastructure platforms available are growing and changing as more startup ventures and established vendors join the market. Each type is discussed. The "antipattern" discussion was especially useful. You can use virtualization tools but not let users access them dynamically and not allow a self-service model—but that isn't infrastructure as code. Using different cloud types is reviewed. When deciding on such a platform you will have to choose between public or private. Security is the number one concern when moving to a public cloud. Other customers with whom you will be sharing resources may be your competitors, and some may be criminals. Left unsaid is whether or not your competitors are criminals, but we must move on.

Cloud portability needs to be considered to avoid lock-in. The only way to prevent this is to build your infrastructure on multiple vendors; this sounds good, but there are always tradeoffs: it will reduce the cost benefits of moving to the cloud.

I never thought I'd see Formula One racing come up in a book like this, but here it is in the "Mechanical Sympathy with the Cloud and Virtualization" section! For an IT person, the more you know how the system works, the more you can get the best from it, just like a racing driver. There are several great points here. Among them, you don't have to know what is happening underneath all the abstraction . . . until you do! A quote worth remembering is "hardware still lurks beneath the abstraction." The story of what Netflix had to do to get AWS instances that met their needs in the real world is just wonderful. You cannot make this stuff up! We are told that in many cases, there may be a limited number of services that will never be suitable for moving to a public cloud. I'd like to hear more about this. Specific examples would have been useful.

Chapter 3—Infrastructure Definition Tools

These are the tools that you will use to manage the compute, storage, and networking resources. They use the platforms described earlier to implement the specified resources. These tools are layered on top of the platform to gain the full benefit of infrastructure as code. How to select and use these tools is covered as well as examples of how to define the resources. When choosing tools for infrastructure as code, we are warned that many tools on the market don't really work well for infrastructure as code, even though their marketing claims otherwise. The first requirement that these tools must meet is to provide a scriptable interface. Specifically, the tool must not rely on a GUI interface. This surprised me, since so much of what I see advertised promotes

drag-and-drop operations. The authors tell us that to support true self-service, tools must make it "possible for technical people to get under the hood," and this requires command-line tools, programmable APIs, and open-source code. Other requirements that are discussed include unattended mode for command-line tools, support for unattended execution, and externalized configuration.

I recommend the section describing how ad hoc scripts lead to the automation fear spiral. There are sections that cover using a standard VCS tool, configuring definition files, and working with infrastructure definition tools and configuration registries. I liked the sections on the basics of using a VCS, the definition of "provisioning," and the pitfalls of tight coupling with a configuration registry. Several examples of scripts and configuration files are discussed. I'd like to hear more about specific examples: we are told we can create servers, but what does that really mean? I can see creating web servers, but how about a complicated system with a database server, web servers, middleware servers, DR servers, and more? Can you orchestrate creating and modifying all of the pieces of such a system? I'd like to see some discussions of what infrastructure as code *can't* do—what level of complexity can't be automated.

Chapter 4—Server Configuration Tools

Virtualization and moving to the cloud have made configuration tools like Puppet and Chef more popular. They support creating and updating large numbers of new servers. Docker is a containerization tool that is more recent and is used to package, distribute, and run applications. The container has the application and pieces of the operating system.

One of the goals of automated server management is that new servers can be provisioned on demand in a few minutes, without human involvement. When a change is made it is applied to all servers, again without human intervention. The process is repeatable, consistent, and self-documented. Changes are safe and easy to make, and automated tests are run each time a change is made. Controlled testing and staged release strategies are supported.

The discussion of tools for different server management functions covers tools for creating and configuring servers, tools for packaging server templates and running commands on servers, and using configuration from a central registry. The section covering security tradeoffs that come along with all this automation is very good. There are lots of opportunities for evildoers. We also learn how general scripting languages can be used effectively with automation. There are several server change management models: ad hoc change management, configuration synchronization, immutable infrastructure, and containerized services. Containerization products such as Docker and Windows Containers are an alternative way to install and run applications on servers.

When choosing tools that connect to managed servers, what about security? What prevents me from making a destructive change to a configuration file knowing it will be propagated to all managed servers?

The next section addresses this issue. It sounds like we are creating a whole new set of security issues, as if we didn't have enough already.

Chapter 5—General Infrastructure Services

Previous chapters discussed tools to provision and configure the core compute, networking, and storage resources needed to build infrastructure as code. You will need a range of other services and tools as well. Examples include DNS and monitoring, message queues, and databases. Really? All this techno-wizardry and we still need databases? I assumed databases had been virtualized and abstracted out of existence!

There are considerations for infrastructure services and tools. All services and tools must support these features: services can be easily rebuilt, the managed elements of the infrastructure are disposable, those same elements are always changing, routine requests are self-service or happen automatically, and changes are safe and easy. To support all of this, we need externalized definition files, self-documenting systems and processes, a version for everything, continuous testing, small changes made instead of batches of changes, and continuously available servers. This is a long list of requirements, and many products that claim to support infrastructure as code have been around too long to have been built to meet all of them.

Referring to software products that have been around a while as legacy, we have a list of the ways they can fail to support infrastructure as code: they don't automatically handle adding and removing infrastructure elements, they assume installation on static servers, they require manual configuration usually via a UI, they can't replicate configuration easily, and it is difficult to test configuration changes. When you are considering services and tools, you need to check for tools that work with externalized configuration, that assume infrastructure is dynamic, and that offer cloud-compatible licensing and loose coupling. Other sections cover "Sharing a Service Between Teams"; "Monitoring: Alerting, Metrics, and Logging"; "Service Discovery"; "Distributed Process Management"; and "Software Deployment."

Chapter 6—Patterns for Provisioning Servers

So far, we have been learning about the tooling needed for creating, configuring, and changing infrastructure elements. Most of the time is spent on servers, and Part II looks at provisioning and making changes to servers.

Server provisioning typically involves procuring hardware, creating the instance, setting up disks, installing the OS and other software, and configuring networking. Existing servers may be re-provisioned when major changes are needed. The lifecycle of a server is discussed. A server template image can be created that is made from a snapshot of an existing server that has already been configured. The specific issues for creating a new server, updating an existing server, and replacing a server are examined. The immutable server pattern doesn't make configuration updates to the server; instead, any changes are made to the template and the server is rebuilt entirely. If you are a fan of this pattern, you don't allow any change to be made to an existing production server. If the infrastructure is "code," you don't change production software on the server, you develop a new release of the software.

There are patterns for creating servers. You can do this by cloning an existing server, using a snapshot saved from a running server, building from a server template, or booting from an OS installation image. In this section we also see the antipattern discussed—in this case, the handcrafted server. This is the most expedient way to get a new server set up, but it doesn't scale. Other antipatterns are the hot-cloned server and the snowflake factory, where automated tools are run manually each time a server is created. The result is that each server is unique and no

one knows how each one was set up. After creating a new server, changes may be needed before it is ready for use. It may need patches or seed data. In theory, you could put all of this in the server template, but then the templates get more complex and require better management. Several strategies for dealing with this are explained.

Chapter 7—Patterns for Managing Server Templates

At this point, we have templates used by our automation tools to build servers. How do we manage all these templates? Templates need to be kept up to date, and the process to do so needs to be repeatable and able to scale. You can outsource this task: there are vendors that will provide prepackaged templates, oftentimes supplied by the hosting provider. This can be very good in the early days of a new infrastructure, but you will want more control as time goes by.

Next, we see how to provision servers using templates. This section addresses the question of which elements of the server configuration should be in the template and which will be added after the server is created. This is a spectrum, and the discussion ranges from provisioning at creation time to provisioning in the template. Immutable servers come up again.

The process for building a server template has steps: selecting the origin image, applying customizations, and packaging the template image. Each of these steps is covered and the term "baking" a server is explained, wherein the newly created server is saved in a format that can be used later to build new servers.

Origin images covers the first step in the process of creating the template. An image of a running server, the origin image, can be generated several ways. We are told not to do this by hot cloning a running server; rather, the origin image must be created from a clean server that has not been used for anything else.

Next is updating server templates. Since we can bake a template, when we update a template, one approach is called "reheating the template." I'm starting to think I've wandered into the "Great British Baking Show." I wonder what Mr. Hollywood would do for a new server? We also learn that over time, templates build up "cruft," which is a technical term I have not seen before, and is not a good thing.

Different servers have different roles in your infrastructure, and you will need to build templates for each role: for example, servers that need different operating systems or software distributions. You may also need different templates for servers tuned for databases versus web servers. It is possible to automate server template management. Tools are available to do all this for you, spin up the server, apply changes, and bake the template to make it ready for use.

Chapter 8—Patterns for Updating and Changing Servers

Keeping all this dynamic infrastructure up to date is more difficult. If not done correctly, you get a "sprawling estate of inconsistent servers." You won't be surprised that I work in just such an estate. It gets worse in that the more sprawl you have, the harder it is to automate management. Over time, this spirals out of consistency, and you lose any infrastructure as code you might have had to begin with. In theory, all you have to do is not allow any changes to a server outside the automated process. That way you always have the latest template file that was used to implement the change. There are models for server change management. The traditional approach is ad hoc change management. Someone decides a change is needed, and someone is tasked with editing a file or running a one-off script to make the change. Who can guess what my world is like? Of course, the author tells us that this approach is the very opposite of infrastructure as code. Other models are discussed, including continuous configuration synchronization, where the configuration tool is executing automatically, perhaps every hour. Immutable servers come up again, no surprise, but now we have a section about situations where such servers really aren't immutable. Go figure!

Next, we see general patterns and practices. Many practices are described. We start with keeping the templates as small as possible, followed by replacing servers when the server template changes. This section fascinates me as we are told to replace running servers whenever the template is updated. I can see this working for a bunch of web servers, updated in sequence, but I'm not clear how this applies to database servers, for example. Another pattern is Phoenix servers, where you rebuild servers automatically, even though no template changes have been made. This is done to catch any ad hoc changes that may have crept in.

Following this we have sections discussing patterns and practices for continuous deployment, for immutable servers, and for managing configuration definitions

I have questions about continuous synchronization. Suppose someone changes the server configuration, $100 million exits the bank, and configuration goes back to normal. Do we become complacent that configuration is auto-magically maintained? Is there no need to review all changes that are made?

Chapter 9—Patterns for Defining Infrastructure

As you generate more and more servers, new issues come up for infrastructure as code. Size, complexity, and number of users all conspire to make it more difficult. As more servers are affected, any change has more potential to cause problems—so you are tempted to make changes less often. You spend more time on being careful with changes. You are less likely to allow users to provision their own resources because they might break more things. This often leads to centralized control, which really means more meetings. The challenge is to set up your infrastructure so that the impact of any given change is minimized.

For environments, we again see that the antipattern is the handcrafted infrastructure, which, again, is all I have ever seen. Specific examples are given as well as all the reasons this is a bad way to go. You do not want to have any snowflake environments that are unique and special, and to which you can't make changes. One of the recommended ways to handle this is reusable definition files. Code examples are shown for this.

Other sections cover organizing infrastructure and running definition tools. I like the section on the pitfalls of sharing infrastructure elements. Do you know who else is sharing the disks that your mission-critical application runs on? Do you trust your hosting service? Do they know what the technicians are actually doing with those resources?

We are told not to share a database to keep applications separate, so one database change won't affect multiple applications. Sounds great, unless you're paying Oracle license fees. But then, perhaps we are being told to move away from Oracle? We are advised to rebuild the existing infrastructure, one piece at a time. Who has time for this? My customers don't want any downtime as it is. Overall, this is a huge shift in mindset for the organization.

Chapter 10—Software Engineering Practices for Infrastructure

The central theme of this book has been that systems and resources used to run software are viewed as software. You can use software tools and practices from the software development world to manage these systems. The reason to do all this is to build quality into these systems. The argument is that building infrastructure—i.e., servers, storage, and networking—needs to be handled the same way as software development. Infrastructure developers need to operate the same way as software developers, following these principles: deliver working code (infrastructure) early, continually deliver small and useful increments, build only what is needed at the moment, build as simply as possible, make sure each change is well built, get feedback for every change, and assume that requirements will change and everything you deliver will need to be modified as users employ the code/infrastructure.

The section covering system quality makes a major assertion about high-quality software: while usually viewed as functional correctness, it really is an enabler of change. The test of quality infrastructure is how quickly and safely changes are made to it. Note the wording *are made* to it, not *can be made*. A high-quality infrastructure means that it is being changed continuously. In the "VCS for Infrastructure Management" section, there is a good discussion of what to manage in VCS for infrastructure. A topic that was new to me, continuous integration (CI), is the practice of frequently integrating and testing all changes to a system as the changes are being developed. You can have separate branches in your VCS to allow a large change to be implemented before integrating and testing with other changes in other branches. There is a detailed discussion of why this isn't CI and why you don't want to do it. Other sections cover continuous delivery (CD), code quality, and managing major infrastructure changes.

Chapter 11—Testing Infrastructure Changes

The previous chapter focused on quality; here we look at testing, specifically automated testing. We are told that testing is essential to be able to continually monitor changes made to a system, but maintaining such an automated test suite is not easy. Note that automation doesn't mean testing goes faster or that testing is too slow to include in said automation. As always, trying to save time by not testing only results in spending more time later looking for bugs. When done well, automated testing results in fewer errors in production, faster resolution for the errors that are found, and the ability to support frequent changes to the system.

The agile approach to testing means that testing is integrated with implementation. Changes are made and testing is done, and both happen continuously. If the automated testing is going to be successful, the entire team must be involved, not just a few people.

The Test Pyramid shows how different types of testing fit together, from low-level tests run more often to medium-level tests to high-level tests run less frequently. The complexity of the testing increases as you move up the pyramid, and you don't spend the time running the medium- and high-level tests until the low-level tests have been passed.

There are tools available to help you set up automated testing. Issues to consider include how to securely connect to a server to run tests and how monitoring is a form of testing.

The section covering managing test code has a very good discussion of wasted time. For instance, an error may be caused by your cloud vendor's API and not your code. Wait—the cloud *doesn't* cure all of our problems?

In the section on roles and workflow for testing, we learn of the three amigos conversation, wherein three people are involved before any work is done: the person doing the work, the person requesting the work, and the person who will test the result. Worth a read.

Chapter 12—Change Management Pipelines for Infrastructure

Continuous delivery means that all the parts of a system work after a change has been made, and we now look at how to set up a pipeline to deliver such changes. The pipeline provides immediate and complete testing after each change, testing system elements progressively (the test pyramid), supporting manual validation steps (user acceptance test), and applying changes into production quickly with low risk. Having delineated how it works, the benefits of said pipeline are explained: the infrastructure components are always production ready, because any change has gone through the pipeline before getting to production. This makes changes easier and less painful. This also supports any governance requirements.

Coverage of guidelines for designing pipelines talks about the need to have consistency across stages of the pipeline. In a perfect world, all environments would be identical, but the world isn't perfect. When it's too expensive to replicate all aspects of the production environment in the test environment, you need to get production-like sooner rather than later. "Sooner" in this context means as far down the test pyramid as possible. How easy it is to mess all this up is shown in the section titled "Devoops." A simple error in one file caused all the servers in the infrastructure to become inaccessible. Automation is great, but it also means errors are automatically distributed throughout the infrastructure. Note that there was no automation to correct this error, someone had to manually access each server and correct the file, one server at a time.

Other sections cover basic pipeline designs practices for using a pipeline and scaling pipelines to more complex systems. I had never heard of Conway's Law, which, the author tells us, states that any group that builds a system will, in the end, design the system to reflect the group's communication structure. I found this interesting and reminiscent of the idea that dog owners begin to resemble their dogs. This chapter ends with a discussion of techniques for handling dependencies between components and practices for managing interfaces between components.

Chapter 13—Workflow for the Infrastructure Team

All of this talk of automation is a radically different way to work than what I have experienced in my many years of IT. After working directly on servers, we are told we must move to working on servers indirectly. This chapter looks at how to get IT operations teams to move from the old ways to the new.

The first topic tells you all you need to know. We are told to automate anything that moves. More specifically, we are told to automate tasks whenever possible. We are told that laziness is the first of three great virtues of a programmer, who will go to great effort to reduce overall work, i.e., the time taken to automate tasks will reduce the time needed, overall, for those tasks. But we aren't told the other two great virtues! Do programmers have virtue? Is that a plugin? Further, we should review all tasks to see

if they are even necessary. That's radical! My job is almost entirely time-wasting paperwork and time keeping. Also, create documentation so that users can execute tasks on their own. Again, that is pretty out there; most of what we do is prevent users from having any access to anything that can be broken.

In a section titled "Using a Local Sandbox," the need for test systems is explained, and these need to represent the real infrastructure. I have never worked anywhere where the money was spent to make test look like production. In my world, each customer system is different. Imagine the cost to reproduce all these different systems for test.

Next, we learn about codebase organization patterns: how to organize all the definitions, scripts, and configuration files. Turns out that the members of the team who take care of routine tasks that don't add value have the monkey role. There are build monkeys and merge monkeys. If the task doesn't add value, why is it being done? I know: there isn't any point in asking.

Assuming you have a workflow, you should think about assessing your workflow's effectiveness. In this section there is a discussion of the best process for handling emergency fixes. The traditional answer is to connect to servers directly and be a hero—saving the day and creating one-off, unique, fragile snowflake configurations. The modern approach is to use the normal process for emergencies. Any change, emergency or not, goes through the normal automated process. I wonder how many shops actually follow this advice.

Chapter 14—Continuity with Dynamic Infrastructure

This chapter covers continuity, which the author uses to cover operational quality. This means services are always available. Traditionally, to provide the level of operational quality where services don't go down, you limit change. You only allow changes to be made through a very long, careful process, and you don't make changes very often. All of this makes sense, but it's in conflict with dynamic infrastructure, where systems are constantly being changed. To deal with this, we need to change the goal. Instead of limiting change because it might break the system, we need to make the system more reliable even as changes are continuously made. The discussion covers service continuity, data continuity, disaster recovery, and security.

In the section on service continuity we learn about the hidden impact of out-of-hours maintenance, where long-running batch jobs grow to overtake the SLA targets. Also interesting is the description of 12-factor applications, which are the guidelines to be followed to make an application work well in the context of dynamic infrastructure. Another example of the change in thinking required by infrastructure as code is that you really are building on unreliable infrastructure. At the scale of the cloud, hardware will most assuredly fail, no matter how many "nines" of reliability you have on any one component. You can pay more for even more reliable hardware, but at some point it doesn't make sense. You need to stop the reliability arms race and lean into a simple truth, you will need to replace hardware when it fails, and you might as well save some money. Reasonably reliable hardware for resilient systems and software makes the most sense.

Other topics in this chapter that I found interesting include zero-downtime changes, dark launching, and zero-downtime changes with data, disaster recovery, prevention versus recovery, and Netflix and the Simian Army.

Security had to come up somewhere and here it is. The automation may create new ways for attackers to gain access to your systems. One example is that frequently rebuilding your servers may lead you to assume that attackers can't make changes and gain access. But this also means that you may not notice changes made by an attacker after each rebuild, and you may not notice attackers gaining access over and over in the interval between automated rebuilds. In fact, the automatic rebuilds automatically cover the tracks left by the attackers.

Chapter 15—Organizing for Infrastructure as Code

This final chapter looks at how to make all this happen in your organization. Your group will have to move from the old ways of working, such as manually configuring servers, to a new era of automated, dynamic infrastructure. Also, note that you need to avoid just bolting automation on top of the old manual processes.

The section on evolutionary architecture describes the road to the cloud, and we are told that this road to the new world has many obstacles. You will be challenged to find the time to make this change while dealing with commercial pressures to keep existing services running.

You will need to know what your team is trying to accomplish. The measuring effectiveness section tells us how to use Kanban to make work visible and encourages us to set up blameless post-mortems. Once I was in a post-mortem, and I stood up and stated that the mistakes were my fault. Management openly expressed disappointment with me, not for my mistakes but for admitting them. They said they enjoyed the "hunt for the guilty," and my honesty had taken this simple, job-related joy from them. Lesson learned: lie to protect what little joy your manager gets from their job!

The chapter ends with a conclusion: with regard to infrastructure as code, it's never finished. I'm not surprised by this, and I'd add that the resistance to change won't suddenly go away either. There will always be some good reason to go back to manual changes on servers.

Conclusion

If you are wondering about infrastructure as code or how your job in the IT world will change over the next few years, this book is worth your time. Since it was published in 2015, I'm sure many of the details have changed. Just this week I was in a meeting for planning the next required round of infrastructure upgrades. Customers will be required to schedule 12 hours of downtime for each of their systems. You won't be surprised to learn that the hardest part of this upgrade project is getting each customer to schedule this downtime. I guess infrastructure as code just isn't going to work for me. ▲

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

Comparing Oracle with PostgreSQL

By Timothy Steward

The foundation of relational databases began with an experimental system. The core of that system was the basis of the System R research. Today this research remains the key functionality of modern database technology. The need for concurrency control and scalability built and maintained by the power of the SQL language exists among the most popular databases of choice, such as MySQL, Oracle, SQL Server, and the popular attention-seeker PostgreSQL. It's not often that you combine open-source and closed proprietary source databases in the same sentence, especially when you think about the giant or the champion in the database realm known as Oracle. But the contender, PostgreSQL, has a lot of similarities to the champion. In both cases they are ACID compliant with full transactional logging capabilities. When you add the EnterpriseDB Advanced Server database, the gap between the champion and the contender starts to close even more. Exploring the quadrants, Gartner appears to recognize that this gap may be closing in the relational database arena.

Comparing the two technologies, we look at Oracle Enterprise Edition, including tools, and the EnterpriseDB Advanced Server, including tools. The comparison allows an Oracle DBA to make an easier transition to the PostgreSQL environment. At first glance you can immediately notice some overall similarities, especially when it comes to SQL capabilities and application development. It's intriguing to think of being able to execute familiar SQL or PL/SQL syntax directly in a PostgreSQL database.

Understanding the terminology can often be confusing if you have been accustomed to Oracle for a number of years.

What	Oracle	PostgreSQL
Table or index	Table or index	Relation
Row	Row	Tuple
Column	Column	Attribute
Data block	Data block	Page (on disk)
Page	Page	Buffer (in memory)

The perfect database may not really exist, both databases have some unique options that will cause a DBA to have nightmares:

- Unlimited database size
- Unlimited rows per table
- Unlimited indexes per table

Unlimited nightmares are true options, showing the flexibility and control you can have within the database versus the restrictions within Oracle. The restrictions are for a good cause, so the unlimited options are possible with PostgreSQL but definitely not advised.

Capacity is almost equal between the technologies when you consider options for creating a stable database that you can maintain. The variation of the columns has a range for PostgreSQL due to the different data types being used.

Maximum	Oracle	PostgreSQL
Table size	4GB x db block size (default 32 TB)	32 TB
Row size	4 TB	1.6 TB
Field size	4 GB-1x db block size	1 GB
Columns per table	1000	250-1600

Tables and partitions are quite similar, and both are feature-rich. There have been some great improvements in recent releases of PostgreSQL with the addition of declarative partitioning and features allowing partitions to be created on multiple columns. Temporary tables are always useful, but the concept of global temporary tables does not exist in PostgreSQL.

Entities	Oracle	PostgreSQL
Temporary tables, (materialized) views, constraints	Same	
Partitioning: range, hash, list, sub-partitioning, IOT	Similar	
Interval partitioning, partitioned indexes	Yes	No

Data types can be the heart and soul of your database. The flexibility within a data model is key to supporting an application. It's known that an Oracle database can lend its flexibility to support everything from OLTP to data warehouses. This is also true for PostgreSQL, which also offers an extra layer of flexibility, allowing you to combine the likes of a NoSQL database with the flair of a relational database. The addition of the JSON and JSONB data types makes this possible.

Spatial data in conjunction with blob capabilities can be a necessity in the world of storing maps and navigational system–related databases. PostGIS is an easily configurable extension built to handle spatial data.

MAX	Oracle	PostgreSQL
Day/time	Yes	
Row id	Yes	
XML Type	Yes	
JSON	Is json check constraint	Native JSON & JSONB with 58 operators, functions, relational converters
Spatial		Yes

There is a common misconception that PostgreSQL can't handle blobs or clob data. The underlying community PostgreSQL has the data type of bytea, which indeed handles binary data. Advanced Server adds the additional blob/clob data types, which will allow data to exist in the same format as Oracle.

MAX	Oracle	PostgreSQL
Integer	Number	DEC, NUMERIC, SMALLINT, INT, BINARY_INTEGER, PLS_INTEGER, INTEGER, BIGINT
Floating point	BINARY_FLOAT, BINARY_DOUBLE	FLOAT, REAL, DOUBLE_PRECISION
Decimal	Number	DEC, DECIMAL, NUMERIC
String	CHAR, VARCHAR2, CLOB, NCLOB, NVARCHAR2, NCHAR	CHARACTER, TEXT, CHAR VARYING, CHARACTER VARYING, VARCHAR, Clob,NClob
Binary	BLOB, RAW, BFILE	BYTEA, BFILE, Blob

Indexing options are quite similar between the two technologies for standard database options. When thinking of a relational database, you do not typically think about the ability to store documents or complete full-text searches. This could be one advantage where PostgreSQL offers the options of GIST and GIN, allowing you to speed up full-text searches built with the same technology of inverted indices as Elasticsearch. GIN can also be smaller than a B-tree index after creation, offering GIN as a possible substitute for B-tree indexes.

Entitles	Oracle	PostgreSQL
B-tree, hash, expresssions, partial, full-text, search, spatial	Same	
Reverse, bitmap, block range	Similar *Block range = Smart Scan*	
K-nearest-neighbor	With options	Native
GIST, GIN *Speed up full text searches*	No	Yes

In the Oracle world we are used to transactions being implicit. For example, a new table creation will do a commit internally. The concept of rollback is not available in this scenario. Within PostgreSQL you have the ability to create transactional DDL, which can include DDL and DML. Transactional DDL will allow everything in your script to roll back if there is a failure. Rerunning a clean process can often be beneficial.

Entities	Oracle	PostgreSQL
Union, Intersect, Except, Inner joins, Outer joins, Merge joins, Common table expresssions, Windowing functions, Parallel query, Query hints, Alter session, Dynamic SQL	Same	
Transactional DDL	Yes	Yes

If we make a guess, we may say that 50% of all SQL scripts written to support application code will use the SYSDATE or ROWNUM. Common SQL extensions and DBA favorites such as the mysterious DUAL table are also present.

Entities	Oracle	PostgreSQL
DUAL, DECODE, Rownum, Sysdate, Systimestamp, NVL, NVL2	Same	

Comparing the two technologies often confuses most techies that have been working in one particular industry. Aside from the terminology, some overall concepts can be confusing. In the simplest form, we know that PostgreSQL is considered open and Oracle is considered closed. Oracle conceptually has an isolated operating system environment, whereas PostgreSQL will adapt and integrate into its surroundings.

For example, it has been said that Oracle is a resource hog, using what's available—mainly because it functions like an operating system. PostgreSQL, on the hand, believes in allowing the operating system to carry the load and not trying to reinvent the wheel: "Why do the work when the O/S can do it for me?"

Conceptually there are some users, roles, and schemas with different meanings but with the same purpose in mind. In Oracle you have users and roles, where PostgreSQL only has roles. But with these roles you can actually log into the database.

If confusion settles in, Advanced Server can make things more relatable. A short navigation through the database brings things to life with the capabilities to utilize the all_ and user_ views or some of the most common DBA views.

DBMS_ALERT	DBMS_AQ	DBMS_AQADM	DBMS_JOB
DBMS_LOB	DBMS_LOCK	DBMS_MVIEW	DBMS_PIPE
DBMS_PROFILER	DBMS_RANDOM	DBMS_RLS	DBMS_SCHEDULER
DBMS_SQL	DBMS_UTILITY	DBMS_CRYPTO	UTL_HTTP
UTL_MAIL	UTL_SMTP	UTL_URL	UTL_FILE

With these common views, the ability to use standard scripts to tune and monitor the database is available. Some key dictionary tables, such as pg_stat_statements, pg_stat_activity, and pg_locks, can produce standard session details.

If scripting isn't your tool of choice there are GUI options such as the standard pgAdmin that ships with PostgreSQL, allowing good options to monitor a single node. For a more complete enterprise solution, EDB offers Postgres Enterprise Manager

"PostgreSQL and Oracle can now be mentioned in the same conversation, as they both share a solid place in the database ecosystem. The key to a successful migration will be to determine the proper use case, evaluate the application, and perform a thorough analysis."

(PEM). For a complete solution, the functionality will bring things closer in relation to OEM offerings. In most environments the developers' favorite is Toad for Oracle. Having used Toad for many years, there is no fear: with the release of Toad Edge for Postgres, things are really shaping up.

To share or not to share, that is the question. RAC appears to have found a place within the infrastructure of every large corporation. Sometimes the true purpose of a good technology gets lost in the hype. The actual comparison is a matter of shared disk vs. shared nothing.

Understanding your use case is the key. The fundamental purpose of RAC is to provide a high availability cluster with load balancing. For PostgreSQL if the use case arises for shared disk, the Red Hat Cluster Suite can be implemented. To complete the solution for HA with load balancing, you could make use of the streaming replication that's native to PostgreSQL, with Pgpool for load balancing, and implement EDB Postgres Failover Manager (EFM) to give you full control over the HA environment.

Entities	Oracle	PostgreSQL
Point-in-time recovery (PITR)	Similar	
Backup and recovery	RMAN	BART
Standby database	Active Data Guard	Streaming Rep/EFM
Flashback	Yes	No

With a proper HA solution in place, you tend to consider your disaster recovery needs and a possible means to have your data geographically disbursed. This can leave you in search of a proper multi-master replication option. Oracle has Golden Gate, which can assist you in this area, but with PostgreSQL, the EnterpriseDB tool replication server will also give you the power of active-active replication with change data capture and features to handle the conflict resolution.

Entities	Oracle	PostgreSQL
Wait events/timed statistics	Similar	
Connection pooling: CPU and I/O resource limits	Similar	
Columnar store	In-memory option	Cstore FDW
In-memory database	Yes	No
Multi-master replication	Golden Gate	XDB Replication Server

With both databases the deployment options are almost endless. You truly can run the same Postgres everywhere.

- Bare metal (Windows, RHEL, CentOS, Linux on Power, SLES, Debian)
- Virtualized deployments (VMware)
- Container deployments (OpenShift, Kubernetes)
- Public/private cloud deployments (AWS, Azure, Alibaba, Google)

An open-source initiative can truly be achieved with a PostgreSQL solution. The price point can be affordable without causing procurement nightmares. Independent of virtualization, a per-core subscription model with no vendor lock-in can sound appealing without the fear of a daunting audit lingering in the shadows.

Sample performance stats show that it's possible to achieve high TPS, billions of writes, and scaling of concurrent users with some flexibility in database size.

Global mobile ad network

- Largest database is 14 TB
- 1.2 billion transactions a day, 55 K transaction per second
- 400 concurrent users
- Analyzes 240 TB of data per day

Online brokerage firm

- 1 billion writes a day
- 3,000 transactions per second
- 800 concurrent users

Global stock trade underwriter

- Largest database is 8 TB
- 6 to 10 million transactions per day
- Global consumer financial services provider
- Example application database is 2 TB
- 200 K SELECT statements per second

PostgreSQL and Oracle can now be mentioned in the same conversation, as they both share a solid place in the database ecosystem. The key to a successful migration will be to determine the proper use case, evaluate the application, and perform a thorough analysis. EDB has performed a multitude of successful migrations to date, building a deep knowledge base of the comparison and challenges. Comparing the technologies may excite you—or you'll appreciate the quick installation and setup—but proper planning will result in successful project. ▲

The Half-Life of Data

By Bud Walker

Data is the new oil of the digital economy. Effectively used, data can help organizations to better understand customer needs and provide winning strategies to meet them. But the data you depend on to make these important decisions is always in flux. This concept is particularly applicable to contact details, as ***customers are literally a moving target***. People change addresses, names, phone numbers, emails, jobs, and status on purpose all the time, and seemingly unpredictably. As time elapses, so does the validity and value of contact details. As customers retire, die, or get married or divorced, stored data becomes stale and out of date, affecting the accuracy and usefulness of data used for communication, analytics, and compliance.

In *The Half Life of Facts: Why Everything We Know Has an Expiration Date,* Samuel Arbesman illustrates how the data you depend on to make important decisions can change with alarming frequency:

> *"Facts change all the time. Smoking has gone from doctor recommended to deadly. We used to think the Earth was the center of the universe and that Pluto was a planet. For decades, we were convinced that the brontosaurus was a real dinosaur. In short, what we know about the world is constantly changing."*

It's worth noting that the brontosaurus is back now, which is exactly Arbesman's point about information: it all goes bad eventually. And while we can't stop facts from changing, we can recognize that what we know "changes in understandable and systematic ways." In science, the term "half-life" is descriptive of the time it takes for half of a body of entities to decay.

Arbesman postulates that by fusing together science and mathematics, we can measure how long it will take for knowledge in any field to change. If equating contact information with radioactivity proves sound, knowing the half-life of contact data would be instrumental in predicting its quality and understanding the importance of maintaining it.

A single atom of uranium actually has an unpredictable rate of decay. It may decay before you finish this article, or it may take millions of years to break down. That sounds about right with respect to contact data—it's unpredictable. However, because each uranium chuck represents trillions of atoms, we can use the probability law of large numbers to derive a constant half-life from a reliable average of 704 million years. Similarly, an element of datum has a wide variance, but as an aggregate, contact data has a trackable record of change.

Viewing individual contacts and contact data elements as a greater entity—like the U.S. Census Bureau database—gives us the opportunity to apply the probability law of large numbers. By tracking the annual changes in contact information through statistics from the U.S. Census, we can then establish the actual half-life of this contact element and determine a conservative expiration date for contact data. Plugging census data into the half-life formula reveals that the average half-life of contact data is 45.4 months, or 3 years and 9 months. The formula and our conclusions are below:

According to U.S. Census Bureau statistics, we have a population of 316 million in the U.S.

Marriages = 2.3 million · Divorces = 1.2 million
Births = 4.3 million · Deaths = 2.5 million
Moves = 47 million

Total Changes = 57.3 million per year
Total Changes = 4.8 million per month

If we bring in the half-life formula:

$$t1/2 = (t \ln 1/2)/(\ln mf / mi)$$
$$t1/2 = \ln (1/2)/\ln((316\text{-}4.8)/316) = 45.4 \text{ months}$$

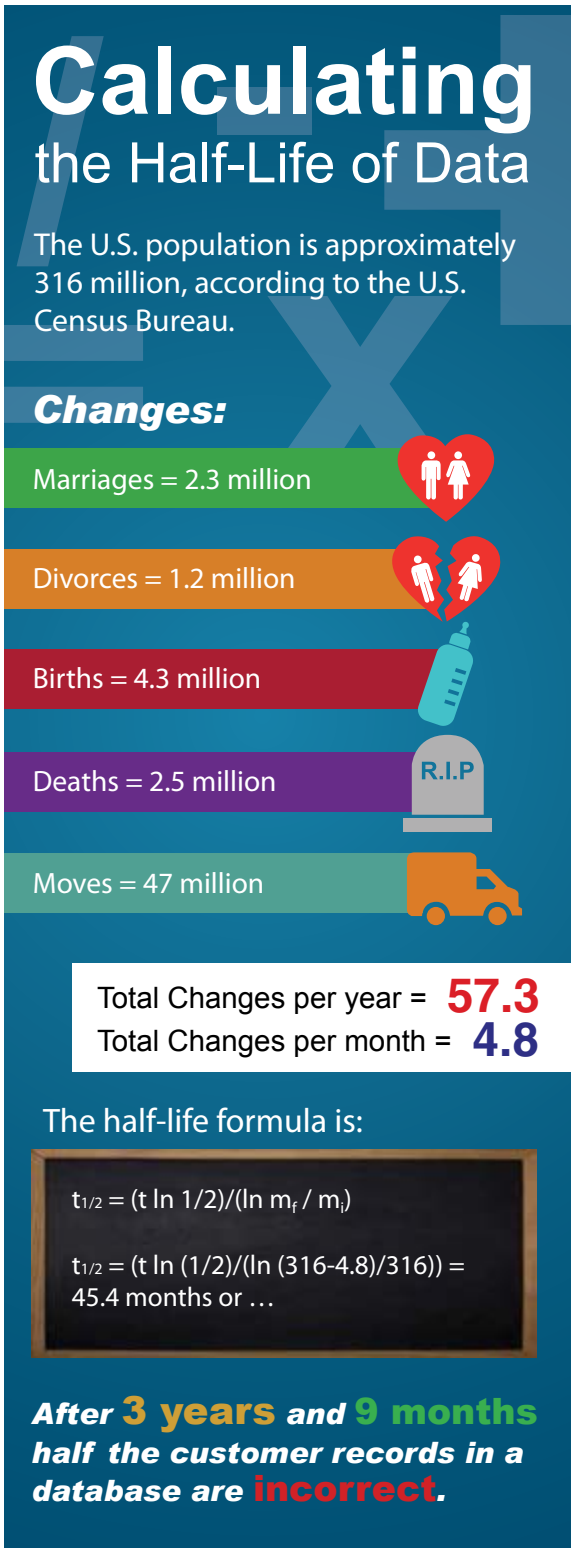
Therefore, after 3 years and 9 months, one-half of the records in a customer's database are incorrect.

This simple exercise doesn't include duplicates, errors, email addresses, or phone numbers—all variables that directly affect your ability to nurture leads and maintain contact. According to Gartner, endangering customer relationships and retention with bad customer data actually costs U.S. businesses in excess of $600 billion per year. Gartner also cites bad data as a fundamental driver of failure—holding it responsible for 40% of business initiatives that went nowhere.

Poor data quality can put up to 12% of your revenue in jeopardy and stall labor productivity by as much as 20% if left untreated. Quality data can help position your organization for growth; at the very least, it addresses these looming problems that we've proven will get worse over time. Awareness of your data's half-life should come with this essential take away: utilizing regularly updated, authoritative reference data is critical to effectively mitigating contact decay.

The foremost goal for you and your customers should be to achieve the highest-quality data at the most affordable price. The 1-10-100 rule postulates that it takes $1 to verify the accuracy of a customer record at point of entry, $10 to clean it in batch form, and $100 per record if nothing is done at all. This includes the costs associated with undeliverable shipments, low customer retention, and unsuccessful sales and marketing initiatives. The bottom line? It costs organizations more *not* to have a "verify and cleanse" solution in place for validating contact data. With that in mind, here are a couple of approaches to consider.

One of the best and most cost-effective ways to ensure sound data in your systems is by verifying data as it enters the database. At some point, most of us have felt uncomfortable about sharing real details on some domains in cyberspace, and we have entered intentionally faulty ones. But whether by accident or in-



tention, erroneous data entering the system brings in the bad right from the get-go. That is why tools that match records in real time can perform a number of functions to facilitate verification.

This prevents corrupt data from entering the system and delays the cost of cleansing that entry. Just like checking IDs at the door, web forms can be equipped with autocomplete for near-instant verification of address, telephone, and email entries.

These contact variables can be pinged immediately to ensure that a number is live and callable, or that an email address is active and receiving email. Referencing the 1-10-100 rule, spending $1 per customer is basically database insurance. It's a cheap expenditure to stop garbage, errors, duplicates, or fraudulent attempts before they ever become an issue.

However, while this is the best form of front-line defense, it does not relieve the duty to run regular matching, deduping, and standardized cleansing routines because it does not alleviate the problems outlined by changes in census data. Global address, phone, email, and name verification solutions go a step further, consolidating or eliminating duplicate records through name-to-address matching and maintaining up-to-date records inside your system.

There are numerous ways to handle this through APIs that fit into your existing data pipelines, plus CRM plugins or one-off batch submissions via the web. On-premise cleansing with open-source batch programs offer added security for firms concerned with finance and healthcare regulations, and only record-matching subscriptions are needed to keep things compliant around the clock. Profiling your data will offer an added understanding of where your weaknesses in data collection might lie, and monitoring your routines with reporting offers easy assessment of how your data is improving over time.

Once you have your regimen in place, you can build on that strong foundation of data quality to enrich records for greater value and utility. Adding demographic and geographic data can provide for better business intelligence, analytics, and sales and marketing initiatives. Powerful, flexible, and scalable solutions can integrate into your existing pipelines and established data quality regimen to augment what you know about who you do business with.

While seemingly tangential, ancillary information could offer that extra bit of insight that may change your course of action, support a key aspect in identity verification, or offer the icing on the cake when interfacing with customers. Comprehensive intelligence plays a major part in how technology companies at the forefront of innovation forecast their business goals and find opportunity.

To gain a single, accurate, and trusted view of critical information assets, a multi-faceted approach to data collection and filtering is optimal. The verification of facts is much like writing a history term paper. The more primary and secondary sources you have to draw from, the firmer your supported conclusions will be. To this end, independent data brokers who are resource agnostic may originate more accurate and trusted stores of what you truly need. For all intents and purposes, the more quality sources the better.

You will find that what sets data quality competitors apart is not just the quality of reference data sources at each one's disposal but also the tools they offer to access those sources. To facilitate an effective data quality campaign, you'll need a partner who can not only advise you on a sound data quality regimen for your business needs but also one that is equipped with data quality tools to establish consistent reliability. A commitment to data quality excellence will position your company for growth—and may prove to be the hallmark of your success. ▲

Melissa (www.melissa.com) transforms stale dated, incomplete, customer data into accurate, rich, valuable information that drives enhanced analytics, improves multichannel marketing and enriches the customer experience. Melissa also specializes in modern technologies including digital identity verification, demographic enrichment, and location intelligence.

How Romeo Won the Heart of Juliet

Fourth International NoCOUG SQL Challenge

Once upon a time, Romeo, the son of Montague, told his cousin Benvolio that he was in love with Rosaline but she was not returning his affections. Benvolio sang a song by the great American songwriter Stephen Foster:

"There are plenty of fish in the sea
As good as ever were caught."

Meanwhile, Count Paris, a relative of Prince Escalus, asked for the hand of Juliet, daughter of Capulet, in marriage. Capulet organized a grand feast and invited Count Paris. Juliet agreed to talk to Count Paris at the feast. Benvolio suggested that Romeo gatecrash the feast so that Romeo could meet other women. Romeo agreed, but only because Rosaline would also be at the feast. At the feast, Romeo instantly fell in love with Juliet and completely forgot about Rosaline. Romeo then sang another song by maestro Stephen Foster.

"I dream of Juliet with the light brown hair,
Borne, like a vapor, on the summer air;
I see her tripping where the bright streams play,
Happy as the daisies that dance on her way."

Count Paris and Romeo both wanted to know when Juliet's birthday was, so that they could send her an edible arrangement. Juliet connected to an Oracle In-Memory pluggable RAC database and typed the following SQL statements:

```
CREATE TABLE Birthdays (birthday DATE NOT NULL PRIMARY KEY)
GRANT SELECT ON Birthdays TO Paris;
GRANT SELECT ON Birthdays TO Romeo;
```

Juliet then inserted a number of values into the Birthdays table, only one of which was her real birthday. She then took Romeo aside and whispered the *month* of her birthday into his ear. She then took Count Paris aside and whispered the *day* of her birthday into his ear.

Romeo declared dejectedly: *"I don't know when your birthday is. But Count Paris doesn't know either."*

Count Paris exclaimed excitedly: *"At first I didn't know when your birthday was. But now I do!"*

Not to be outdone, Romeo exclaimed: *"Now I do too!"*

Count Paris whipped out his Samsung Galaxy S6 and ordered a dozen gourmet dipped swizzled strawberries from **ediblearrangements.com**. It was too late for same-day delivery, so he ordered next-day delivery along with a card saying "Better Late Than Never!" Juliet wasn't thrilled but politely thanked Count Paris.

Romeo did not have a smartphone but he had a bike and, having worked as a bike messenger, he knew the streets of Verona like the back of his hand. He pedaled furiously to the Edible Arrangements store on the next block where he bought a dozen gourmet dipped swizzled strawberries and brought them back to Juliet. Juliet was thrilled because, more than anything, she loved to receive gourmet dipped swizzled strawberries on her birthday.

And that's how Romeo won the heart of Juliet. Our story ends with Count Paris singing another verse from Benvolio's song:

"There are plenty of fish in the sea
But, oh, they're hard to be caught."

The obvious question remains: what was the day of the feast? ▲

Fourth International NoCOUG SQL Challenge

by Iggy Fernandez

In an interview for the *NoCOUG Journal* (http://www.nocoug.org/Journal/NoCOUG_Journal_200608.pdf#page=4), Steven Feuerstein was asked: *"SQL is a set-oriented non-procedural language; i.e., it works on sets and does not specify access paths. PL/SQL on the other hand is a record-oriented procedural language, as is very clear from the name. What is the place of a record-oriented procedural language in the relational world?"*

Steven replied: *"Its place is proven: SQL is not a complete language. Some people can perform seeming miracles with straight SQL, but the statements can end up looking like pretzels created by someone who is experimenting with hallucinogens. We need more than SQL to build our applications, whether it is the implementation of business rules or application logic. PL/SQL remains the fastest and easiest way to access and manipulate data in an Oracle RDBMS, and I am certain it is going to stay that way for decades."*

To prove Steven correct, NoCOUG has held four international SQL challenges.

First International NoCOUG SQL challenge (2009)

An ancient 20-sided die was discovered in the secret chamber of mystery at Hogwash School of Es-Cue-El. A mysterious symbol was inscribed on each face of the die. The great Wizard of Odds discovered that each symbol represents a number. He also discovered that the die was biased: that is, it was more probable that certain numbers would be displayed than others if the die was used in a game of chance. The great wizard recorded this information in tabular fashion as described below.

Name	Null?	Type
FACE_ID	NOT NULL	INT
FACE_VALUE	NOT NULL	INT
PROBABILITY	NOT NULL	REAL

The great wizard then invited all practitioners of the ancient arts of Es-Cue-El to create an Es-Cue-El spell to display the probabilities of obtaining various sums when the die was thrown "N" times in succession in a game of chance.

The contest was a great success; nine solutions were found by participants in seven countries and three continents but the winner, Alberto Dell'Era from Italy, rose above the competition by implementing Discrete Fourier Transforms and becoming the first knight of the August Order of the Wooden Pretzel. You can read an explanation of his wonderful solution at http://www.nocoug.org/Journal/NoCOUG_Journal_200908.pdf#page=14. Alberto also implemented Fast Fourier Transforms but we won't even go there.

Second International NoCOUG SQL Challenge (2011)

An ancient manuscript titled "Love Your Data" was discovered in the secret chamber of mystery at Hogwash School of Es-Cue-El. The manuscript was covered with mysterious words and the great Wizard of Odds implored contestants to create an Es-Cue-El spell that revealed the secret message. Here is a short excerpt from the ancient manuscript.

	A	
COMPREHENSION	ABILITY	OLD
	ABOUT	
	ALWAYS	
SCIENCE	AND	PHYSICS
	ANY	
	AS	
SO	ASK	ABILITY

Andre Araujo (Australia), Rob van Wijk (Netherlands), and Ilya Chuhnakov (Russia) submitted solutions and became the second, third, and fourth knights of the August Order of the Wooden Pretzel. Ilya submitted two solutions: one using the MODEL clause and one using recursive common table expressions. You can read their wonderful solutions in the 100th issue of the *NoCOUG Journal* (http://www.nocoug.org/Journal/NoCOUG_Journal_201111.pdf#page=20).

Third International NoCOUG SQL Challenge (2012)

The Wicked Witch of the West had invited six friends to the Third Annual Witching & Wizarding Ball at Pythian Academy of Es-Cue-El & No-Es-Cue-El. Burdock Muldoon and Carlotta Pinkstone both said they would come if Albus Dumbledore came. Daisy Dodderidge said she would come if Albus Dumbledore and Burdock Muldoon both came. And so on and so forth. The Wicked Witch of the West needed an Es-Cue-El or No-Es-Cue-El spell to determine whom she needed to persuade to attend the wizarding ball in order to ensure that all her invitees attended.

Master sorcerer Lukasz Plata of Poland not only solved the problem with a single SQL statement but provided a proof that his solution was correct. He became the fifth knight of the August Order of the Wooden Pretzel. You can read his wonderful solution at http://www.nocoug.org/Journal/NoCOUG_Journal_201211.pdf#page=12.

Fourth International NoCOUG SQL Challenge (2015)

The fourth challenge was published in the May 2015 issue of the *NoCOUG Journal* (http://www.nocoug.org/Journal/NoCOUG_Journal_201505.pdf#page=22) and was quite unlike the first three challenges. The first three challenges required the

contestants to devise an efficient mathematical algorithm and then implement it using SQL. For example, the winning solution to the first challenge used advanced mathematical techniques called "convolutions" and "Fourier transforms." This is not a typical use of SQL in the real world. Also, contestants in the first three challenges sent their entries directly to NoCOUG (though they were free to publish their entries on their own websites or blogs).

The fourth challenge on the other hand did not require contestants to be expert mathematicians. Instead the challenge was an exercise in logic and in interpreting a "functional specification." This is closer to the uses of SQL in the real world. Also, contestants posted their solutions on the NoCOUG blog (**https://nocoug.wordpress.com/2015/05/11/fourth-international-nocoug-sql-challenge/**). This meant that all the contestants benefited from the work of others.

The challenge was a disguised variant of the problem called "Cheryl's Birthday" that was featured in the 2015 Singapore and Asian Schools Math Olympiad for 14-year-old students.

Albert and Bernard just became friends with Cheryl, and they want to know when her birthday is. Cheryl gives them a list of 10 possible dates:

May		15	16			19
June				17	18	
July	14		16			
August	14	15		17		

Cheryl then tells Albert and Bernard separately the month and the day of her birthday respectively.

- **Albert:** I don't know when Cheryl's birthday is, but I know that Bernard doesn't know too.
- **Bernard:** At first I don't [sic] know when Cheryl's birthday is, but I know now.
- **Albert:** Then I also know when Cheryl's birthday is.

So when is Cheryl's birthday? (**https://en.wikipedia.org/wiki/Cheryl%27s_Birthday**)

NoCOUG tried to disguise the problem by using characters from Shakespeare's play Romeo and Juliet. Albert became Romeo and Cheryl became Juliet.

Once upon a time, Romeo, the son of Montague, told his cousin Benvolio that he was in love with Rosaline but she was not returning his affections. Benvolio sang a song by the great American songwriter Stephen Foster:

"There are plenty of fish in the sea
As good as ever were caught."

Meanwhile, Count Paris, a relative of Prince Escalus, asked for the hand of Juliet, daughter of Capulet, in marriage. Capulet organized a grand feast and invited Count Paris. Juliet agreed to talk to Count Paris at the feast. Benvolio suggested that Romeo gatecrash the feast so that Romeo could meet other women. Romeo agreed, but only because Rosaline would also be at the feast. At the feast, Romeo instantly fell in love with Juliet and completely forgot about Rosaline. Romeo then sang another song by maestro Stephen Foster.

"I dream of Juliet with the light brown hair,
Borne, like a vapor, on the summer air;
I see her tripping where the bright streams play,
Happy as the daisies that dance on her way."

Very melodramatic. The solution is iteratively obtained by applying each clue in turn:

- The first clue has two parts: Albert (who has been told the month by Cheryl) cannot uniquely determine the day at this stage in the game (Clue 1a) and knows that Bernard (who has been told the day by Cheryl) cannot uniquely determine the month at this stage in the game (Clue 1b). By applying Clue 1a, we (the public) can eliminate all months which only contain a single candidate day (there are no such months in the sample data above) and, by applying Clue 1b, we can eliminate all months which contain a candidate day that is unique (because if Cheryl's birthday occurred in such a month, then there remains a possibility that Bernard could determine the month at this stage in the game if the day given to him by Cheryl was unique). Clue 1a does not help us here but Clue 1b allows us to eliminate all days in May and June from contention.
- The second clue (Clue 2) is that Bernard (who has only been told the day by Cheryl) is able to use Clue 1a and Clue 1b to uniquely determine the month. We (the public) still don't know Cheryl's birthday, but, by applying Clue 2, we can eliminate July 14 and August 14 from contention because, if either one of them was Cheryl's birthday, then Bernard would not have been able to uniquely determine the month. Only July 16, August 15, and August 17 are left in contention.
- The third clue (Clue 3) is that Albert (who has only been told the month by Cheryl) is able to use the previous clues (and hence knows that only July 16, August 15, and August 17 are in contention) to uniquely determine the day. We (the public) can therefore eliminate August 15 and August 17 from contention because Albert would not have been able to uniquely determine the day if Cheryl's birthday had been in August (since August contains two candidate days that are still in contention).

This leaves only July 16 in contention. Cheryl's birthday must be on July 16.

Assuming that the data is stored in a table called Dates with a single column called DateOfBirth, the above exercise in logic can be expressed in SQL as follows:

```
select
  m, d
from (
  select
    m, d,
    count(*) over (partition by m) as m_count
  from (
    select
      m, d,
      count(*) over (partition by d) as d_count
    from (
      select
        m, d, m_count,
        min(d_count) over (partition by m) as min_d_count
      from (
        select
          m, d,
          count(*) over (partition by m) as m_count,
          count(*) over (partition by d) as d_count
        from (
          select distinct
            extract(month from dateofbirth) as m,
            extract(day from dateofbirth) as d
          from dates
```

```
      )
    )
  )
  -- Clue 1a and Clue 1b
  where m_count > 1 and min_d_count > 1
 )
  -- Clue 2
 where d_count = 1
)
-- Clue 3
where m_count = 1
```

The original goal specified in the challenge announcement was an ANSI-standard SQL query of minimum length. However, NoCOUG did not list the candidate dates and required that the solution be able to process all data sets that fit the rest of the story. Perhaps because Cheryl's Birthday was a well-known problem, perhaps because contestants were influenced by the work of others, or perhaps because of sheer subtlety, all the contestants initially missed Clue 1a. The initial set of solutions were therefore incorrect. Since the logic of the solution was now an open secret, the original goal of minimum length was discarded. Chris Goerg from Germany then submitted the following correct solution using the MODEL clause.

```
with d as (
  select unique
    extract(month from dateofbirth) m,
    extract(day from dateofbirth) d
  from dates
)
select m, d from (
  select * from d
  model
  dimension by (m, d)
  measures(0 s, 0 t)
  rules (

    -- Count the number of times each day is duplicated
    -- Store the count in s

    s[m,d] = sum(1)[m, cv()],

    -- Apply Clue 1a and Clue 1b
    -- Set t to 1 if a date is still in contention

    t[m,d] = case
             when min(s)[cv(), d] > 1 and sum(1)[cv(), d] > 1
             then 1
             end,

    -- Apply Clue 2
    -- Count the number of times each day is duplicated
    -- Store the result in s
    -- Only dates with s = 1 remain in contention after this point

    s[m,d] = case
             when t[cv(), cv()] = 1
             then sum(t)[m, cv()]
             end,

    -- Apply Clue 3
    -- Count the number of dates still in contention in each month
    -- Store the result in t
    -- Only dates with s = 1 and t = 1 remain in contention after this point

    t[m,d] = sum(case when s = 1 then 1 end)[cv(), d]
  )
)
where s = 1
and t = 1
```

The secret of Chris's solution is his use of *two* measures (s and t), not just one. Both measures are initialized with the value 0. We can visualize the progress of his solution with a little PIVOT magic:

```
select * from (
with d as (
  select unique
    extract(month from dateofbirth) m,
    extract(day from dateofbirth) d
  from dates
)
select * from (
  select * from d
  model
  dimension by (m, d)
  measures(0 s, 0 t)
  rules (
    -- Add rules here
  )
)
)
pivot (min(nvl(to_char(s),'-'))||'|'||nvl(to_char(t),'-'))
for d in (14,15,16,17,18,19))
order by m;
```

The following tableaus show the progression of his solution. In the initial tableau, both s and t are set to 0.

M	14	15	16	17	18	19
5		0|0	0|0			0|0
6				0|0	0|0	
7	0|0		0|0			
8	0|0	0|0		0|0		

Next, count the number of times each day is duplicated and store the count in s.

M	14	15	16	17	18	19
5		2|0	2|0			1|0
6				2|0	1|0	
7	2|0		2|0			
8	2|0	2|0		2|0		

Next, apply Clue 1a and Clue 1b. Set t to 1 if a date is still in contention.

M	14	15	16	17	18	19
5		2|-	2|-			1|-
6				2|-	1|-	
7	2|1		2|1			
8	2|1	2|1		2|1		

Next, apply Clue 2. Count the number of times each day is duplicated and store the result in s. Only dates with s = 1 remain in contention after this point.

M	14	15	16	17	18	19
5		-|-	-|-			-|-
6				-|-	-|-	
7	2|1		1|1			
8	2|1	1|1		1|1		

Next, apply Clue 3. Count the number of dates still in contention in each month. Store the result in t. Only dates with s = 1 and t = 1 remain in contention after this point.

M	14	15	16	17	18	19
5		-|-	-|-			-|-
6				-|-	-|-	
7	2|1		1|1			
8	2|2	1|2		1|2		

Chris was judged the winner on the grounds of novelty and originality. He wins an Apple Watch Sport and becomes the sixth knight of the August Order of the Wooden Pretzel. ▲

NoCOUG Conference #128

Post-Conference Reception Hosted by Axxana

Penny Avril told us what's new and coming next in Oracle Database

The raffle winner is blank!

NoCOUG Conference #128

Post-Conference Reception Hosted by Axxana

Networking reception

Mandatory Kamran picture

NoCOUG Conference #128

Post-Conference Reception Hosted by Axxana

NoCOUG webmaster reunion: Vadim Barilko (right) and Eric Hutchinson (left).

NoCOUG
P.O. Box 3282
Danville, CA 94526

RETURN SERVICE REQUESTED