Five Database Technology Trends

with Guy Harrison

Guy Harrison

I first started working with database technology in the mid-1980s. A lot has changed in database technology over those 30 years, but I would have to say that there have been more fundamental changes in databases over the past five years than in the 25 years preceding. I think these changes can be boiled down to a few key trends that are revolutionizing the databases of today and the future. Those trends include:

- The end of the era in which a single type of database technology—the relational database—served as a one-size-fits-all-solution.
- The emergence of big data technologies, most significantly Hadoop.
- The explosion of non-relational operational databases (aka NoSQL) such as Cassandra and MongoDB.
- The increasing capacity to store data in columnar rather than row-oriented format.
- The diminishment of the spinning magnetic disk as the only viable technology for persistently storing data.

Trend #1: The End of Relational Hegemony

Database technology arguably predates digital computing: Many of IBMs first "business machines" were tabulators—which were able to summarize and analyze data held on punched cards. These punched cards arguably represented a paper database. However, what we think of as computer databases emerged from the combination of digital computing and persistent storage media such as magnetic disks and tape.

The first digital databases—circa 1950—were based on sequential files that would be scanned into memory on every access. However, during the 1960s, true database management systems emerged that combined indexed access methods and structured schemas. By the end of the 1960s, the hierarchical and network database models were established and pervasive.

Edgar Codd first defined *A Relational Model of Data for Large Shared Data Banks* in 1970. The relational model had several advantages over the existing models: most significantly it supported the concept of ad-hoc queries in which the data model was not required to anticipate all of the possible queries that might be supported. The relational model also required an interactive query language—eventually standardized on SQL—that allowed non-programmers to execute query operations. This interactive query language broke the logjam of report writing that plagued IT departments of the time and was a major factor in the rapid uptake of Relational Database Management Systems (RDBMS).

From the first release of Oracle in 1978 to the middle of the 2000s, virtually every significant database released was either relational or claimed to be relational. The database timeline in Figure 1 shows that virtually every database released from 1980 to 2000 was—or at least claimed to be—a relational database.

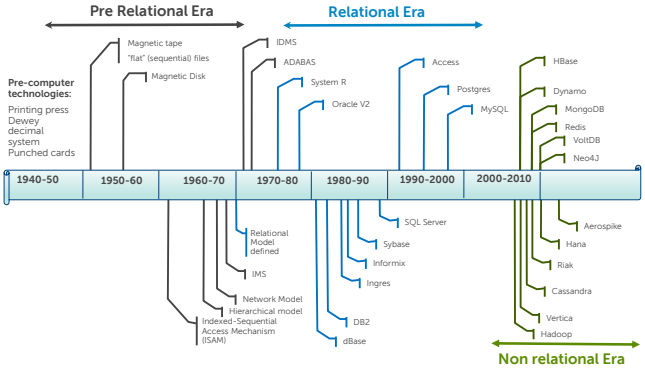

Figure 1. Database timeline

While some non-relational systems—Object-Oriented Database Management Systems (OODBMSs) and XML databases—were developed, none gained significant traction during this period. For an entire generation of software professional, the RDBMS reigned supreme.

The First Cracks Appear

The end of the relational hegemony had been foretold since the middle of the last decade, most significantly by relational database pioneer Mike Stonebraker, whose 2005 paper *"One Size Fits All": An Idea Whose Time Has Come and Gone* argued that modern applications would be better served by a mix of database systems optimized for specific workloads. At the time, the proposed "new models" of DBMS were largely theoretical. However, Stonebraker correctly identified the key trend: the complete dominance (though not primary role) of the relational database was about to end.

As we can see from the timeline in Figure 1, between 2005 and 2014 literally dozens of new databases were released—almost none of which were fully relational.

The Third Platform

There are many factors behind the end of the relational supremacy, but at a high level, it is a result of the paradigm shifts in

software application architecture. We've seen any number of paradigm shifts within the last three decades, including microcomputer to minicomputer and the introduction of the Internet. However, arguably all of the things we've seen fall into three major waves of application architecture.

IDC coined the term "the third platform" to describe the current application landscape and contrast it to what has come before. In IDC's widely accepted model, software applications have undergone three fundamental changes:

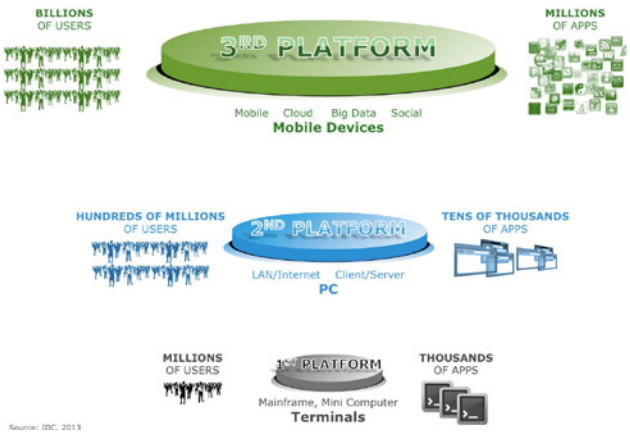

Figure 2. The third platform (source: IDC)

- The first platform was based on mainframe systems accessed through "dumb" terminals. All processing logic was centralized. These systems typically supported only hundreds of users, although millions of users in aggregate accessed this category of applications.
- The second platform emerged with the client server model and early web applications. These applications divided application logic into two or more tiers (one of which was typically the database). As the Internet became a universal WAN, these applications entered the realm of the mainstream consumer and attracted hundreds of millions of users.
- The third platform emerged from the nexus of megatrends that we've all experienced over the past 10 years: smartphones, social networking applications, and cloud computing have combined to revolutionize not just application architecture but digital lifestyles and our very societies. Big Data was arguably born out of the third platform but also strongly influenced its growth. The increasing data generated by our mobile and social interactions creates new avenues for user experience that in turn generate more data. The increasing prevalence of Internet-connected sensors—the Internet of Things (IoT)—further accelerates the paradigm shift.

A New Database World Order

Databases supporting the third platform need to deal with availability, throughput, and performance requirements that were simply not in existence when the RDBMS was conceived. The demands on databases have included some of the following pressures:

- Global availability
- Unstructured data and rapidly changing schemas
- Massive volumes—data volumes had been increasing steadily since the genesis of the database, but new significant pressure arose to store large amounts of unstructured data at prices far lower than were possible with a relational schema.
- Diverging storage technologies that for the first time created an increasing divergence between the price of *storing* data and the price of *accessing* data. Solid-state disk technologies and in-memory database options became available, which reduced the price of delivering an I/O operation, but these technologies were far more expensive when it came to storage. It became impossible to provide the best value dollar/GB and best value dollar/IOPS in the same database technology and storage architecture.
- The economics of various components of the technology stack also increasingly demanded changes in core database architectures. CPU and memory capacity increased exponentially in accordance with Moore's law, while network and disk capacity increased geometrically at best. As a result, it became increasingly more economical to move processing to the data, rather than to move data to the processing.

Each of these will be discussed in upcoming sections, but in overview we are currently in an era in which several different database technologies are employed for most moderately sized businesses:

- The relational system remains dominant as the underlying storage system for traditional business systems—in-house ERP and CRM. For instance, no non-relational system is certified to support a SAP implementation and there is little sign that this will change. However, as ERP and CRM systems migrate to cloud-based SaaS alternatives, many will be re-engineered and some may potentially adopt a non-relational data store.
- The relational database remains entrenched as the basis for virtually all in-house data warehouses, and BI tools in general only deliver their full functionality when working against relational systems that support the full range of ANSI SQL, including windowing functions, star schema optimizations, bitmap indexes, and so on. Again, there is some threat here from SQL-on-Hadoop, but for the most part the real-time data warehouse remains the province of the relational system.
- Hadoop is increasingly deployed alongside the relational data warehouse as a "data lake." Hadoop serves as the staging area for data that has yet to find its way into a fixed-schema DW and as a repository for fine-grained raw data that may be the subject of data science projects.
- Web applications—particularly those that in the past would have been based on the LAMP (Linux, Apache, MySQL & Perl/Python/PHP) stack—are increasing based not on a relational system such as MySQL but on a non-relational alternative such as MongoDB. These decisions are largely being driven from the ground up as developers decide on the NoSQL database in the absence of any corporate mandate and in projects that adopt the DevOps style of application development.

- In addition to the widespread use cases outlined above, there are increasingly viable niches for "NewSQL" or otherwise not-quite-relational database systems. These include columnar variations on the data warehouse such as Vertica, in-memory systems such as Hana and VoltDB, and graph-based databases such as Neo4J.

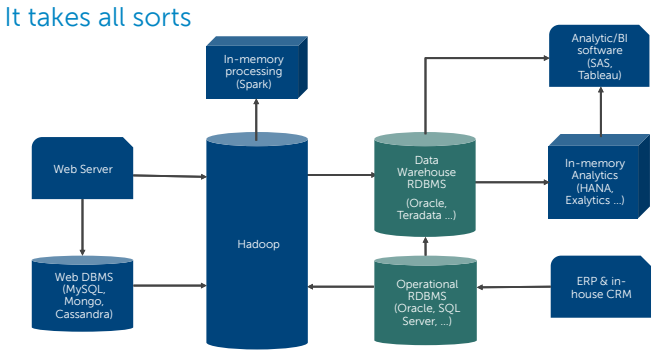

Figure 3. Modern enterprise data architectures combine multiple technologies

Trend #2: Big Data and Hadoop

Every few years we seem to be subjected to a new buzzword that so rapidly dominates all conversation as to become meaningless. Many believe the term "big data"—seemingly added to any conversation that in any way relates to data—has become such a buzzword. But in reality, big data encompasses two very real trends in modern data management:

- We are required to store more, and more varied, data than ever before.
- The use of data has shifted in importance: analysis of data used to be primarily about *operational efficiency*; in the modern era it has become about *competitive advantage*.

The Data Revolution

While there are many arguably valid definitions of big data, my personal favorite is the *Industrial Revolution metaphor* coined by Joe Hellerstein at O'Reilly. Before the Industrial Revolution in the 1800s, all goods were hand produced. Likewise, before the industrial revolution of data, all data was produced by hand: entered by individuals paid to do so by the organization. Nowadays, of course, data comes in from everywhere: customers, social networks and, increasingly, Internet-enabled devices.

From a database point of view, this involves more than just storing qualitatively more data. We are being called upon to manage the storage of fast amounts of unstructured, unprocessed data. In the past data was cleansed, schematized and aggregated prior to being loaded into a data warehouse. This still occurs, but we are also required to store the original, unadulterated raw data. Why? Because often it's the analysis of this original data that can be used to unlock competitive advantage through advanced techniques of data science.

Google and Hadoop

Undeniably, Google is the pre-eminent pioneer of big data. From the very beginning, Google was faced with the challenge of storing and analyzing exponentially increasing volumes of data, and managing data that was inherently flexible and evolving in format.

No existing database management system had either the capacity or capability of storing the volumes and varieties of data that Google needed to master. So Google was forced to adopt its own hardware and software stack for the purpose of building a scalable commercial solution.

This stack involved many components but most notably included the following building block technologies:

- Google File System (GFS) is a distributed file system that allows the directly attached storage in very large numbers of commodity servers to be exposed as a single logical file system.
- Map Reduce is a programming model that allows complex problems to be broken up into simple parallelized map functions together with reduce functions that combine the outputs from each parallel stream. Multiple Map Reduce pipelines can be constructed to solve a very wide variety of problems. While somewhat simplistic (at least compared to modern alternatives) Map Reduce is an algorithm that is applicable to a wide variety of problems.
- Bigtable is a non-relational database system built on GFS. It uses a normalized data model in which rows are associated with a variable, sparse, and potentially very large number of columns, avoiding in many cases the need for separate storage of detail records and for joins.

Google published details of the above three technologies during the latter part of the last decade. Consequently, an open-source project arose that implemented these and other key components of the Google stack. This project is, of course, Hadoop, which is arguably the most significant new database storage system to be released in the last decade.

Hadoop was nurtured at Yahoo!, which allowed it to achieve fairly rapidly the massive scalability for which it is well known. It provides compelling advantages over relational systems for big data storage:

- It is significantly cheaper than all alternatives for storing large amounts of data online. This is what Forrester calls "*Hadooponomics*": the ability to linearly scale data storage and data-processing costs.
- It supports the storage and processing of unstructured and semi-structured data. There is no requirement to define the schema of a data item for storing it in Hadoop. While this undoubtedly leads to some degree of "Hadumping"—the thoughtless dumping of all data without analysis—it obviates the delay and cost involved with ETL into traditional stores.
- Libraries of data science and advanced analysis exist for Hadoop, providing data scientists with at least a starting point for the analysis of data held in Hadoop.

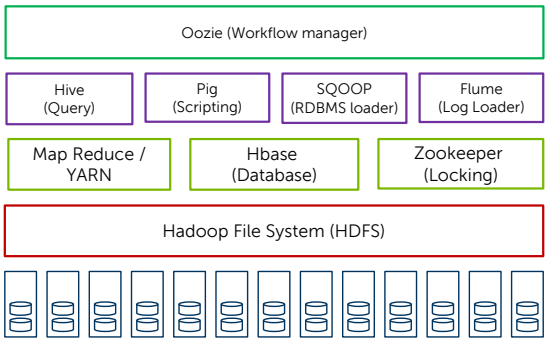

Figure 4. Simplified Hadoop stack

Today, Hadoop forms the background of most big data projects.

Trend #3: Distributed Non-Relational Databases (a.k.a. NoSQL)

Hadoop is an example of a non-relational database system. However, while Hadoop arose because of pressures on analysis of massive amounts of data, other non-relational systems were developed to meet other needs.

Traditionally, start-ups relied heavily on open-source software stacks: Linux, Apache, and MySQL were the key components of web start-ups such as Facebook and Twitter. However, as these sites experienced rapid growth, it became increasingly difficult to scale systems built on these foundations.

Means of scaling the web server are well established, but the database presented different challenges. Some degree of relatively easy scalability can be achieved by read offloading: using read only replicas or distributed object cases such as Memcached. However, at some point the master database becomes the bottleneck.

The most common solution to this dilemma is "sharding." In a sharded database, the largest tables are partitioned across multiple nodes. This partitioning is based on some key value, such as a user ID. The application must be modified to know how to read from the appropriate shard as well as typically integrating code to handle memory-cached copies of data and the read-only replicas.

Although sharding succeeded as a solution and is still in use in sites such as Facebook and Twitter, it is widely regarded as a poor solution. A sharded database has lost almost all the characteristics of a relational system: joins are not possibleand transactional logic can no longer be expressed simply in SQL.

It has been recognized for some time that the requirements of massive scalability and availability common to the biggest web applications are inconsistent with the transactional model expressed in relational systems. This inconsistency is expressed in the CAP (aka Brewer's) theorem.

The CAP theorem states that you can only have two of these three properties in a system:

- Consistency: Everyone always sees the same version of all data.
- Availability: The system can remain available when nodes fail.
- Partition tolerance: The system can survive and remain available when split in two by a network partition.

While a system such as Oracle RAC could provide high availability and strong consistency, a new model was needed for systems that would prefer to sacrifice strong consistency in order to maintain availability across multiple data centers and geographies.

Amazon published their model for highly available data storage in 2007, coining the term "Eventually Consistent" to describe this model. Together with the Google Bigtable model discussed earlier, these ideas inspired a variety of NoSQL databases to emerge, including Hbase, Riak, DynamoDB, and Cassandra.

Non-relational systems were also influenced by object-oriented and XML databases. Programmers have been frustrated for a long time over the need to map program objects into relational form. The so-called "impedance mismatch" between the two models creates programmer overhead and hard-to-maintain code. As a consequence of the emergence of JSON (JavaScript Object Notation) as a widely used object document model in web

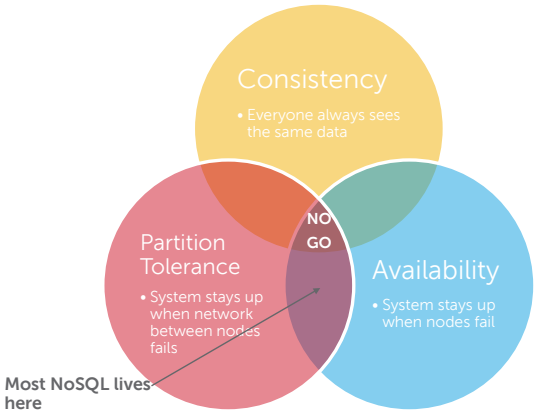

Figure 5. CAP (Brewer's) theorem

applications and the validation of non-relational systems, a number of JSON-based non-relational databases emerged, such as MongoDB and Couchbase.

Today these non-relational systems—usually referred to as NoSQL databases—are an increasingly common choice for back-ending web applications.

Trend #4: Columnar Databases

The natural way that most of us think about data is to imagine the data organized into horizontal rows and vertical columns—the same way that we might enter the data on index cards, a paper ledger, or Excel. This organization suited early database systems as well, since most operations were on individual rows/records: the well-known CRUD (Create, Read, Update, Delete) pattern.

However, in data analysis it is rare to perform an analysis on all the columns in a single row; rather, we tend to do operations on all the rows in a single column. This shift toward column-based processing created the opportunity for a new organizational model for data: the columnar storage model.

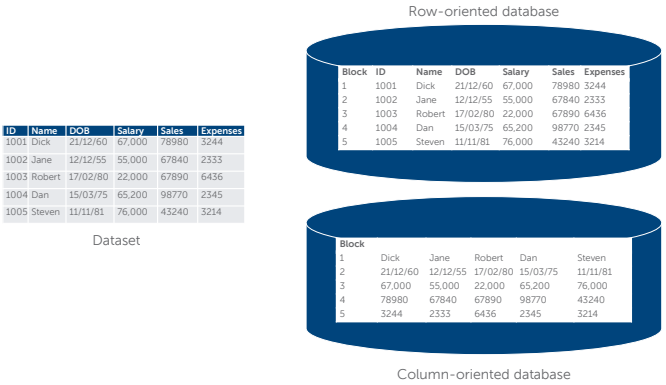

Figure 6. Row-oriented and column-oriented databases

In a column store, columns tend to be organized together on disk rather than by clustering rows together on disk (Figure 6). This has several notable advantages:

- Aggregate operations such as SUM(), AVG(), etc., typically need to do less I/O, since all the data they need is grouped together on disk.
- Compression is improved, since there tends to be more repetition across columns than across rows.

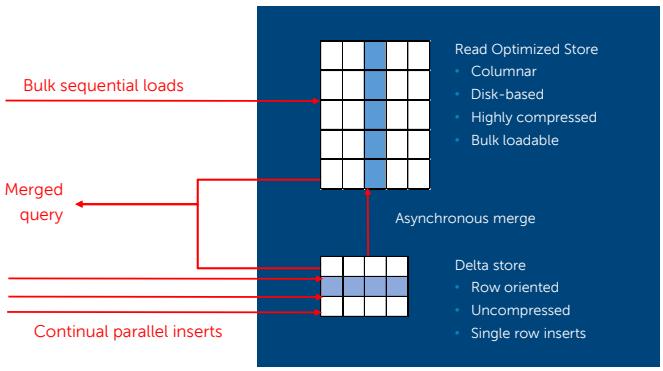

Figure 7. Delta store within a column store

Weighed against these advantages is, primarily, a reduction in efficiency for typical OLTP operations. An insert, for instance, must perform as many block operations as there are columns in the row. Continual inserts effectively require continual reorganization of the column store, which is rarely practical. For this reason, most column stores support the idea of a "delta store": a staging area that is row organized. Inserts build up in this delta store and are periodically merged with the column store. In this interim period, queries must access both the delta store and the column store in order to retrieve up-to-date data.

Sybase IQ was one of the first commercial column-store databases. Mike Stonebraker and his colleagues defined a theoretical model for column stores called "C-store" that heavily influenced the design of HP Vertica and other column-oriented systems.

Oracle uses column-oriented architecture within both Exadata Hybrid Columnar Compression (EHCC) and Oracle Database 12*c* In-Memory Option.

In EHCC, each block contains data for specific columns, but all columns for a specific row are held within a single compression unit—typically a 1 MB structure. EHCC allows Oracle to get the higher compression rates allowed for by columnar storage while limiting impact on single-row operations, which can still be satisfied by a single 1 MB I/O.

We'll consider Oracle Database 12*c* In-Memory Option in the next section.

Trend #5: The Demise of the Spinning Disk

The magnetic disk device has been a ubiquitous presence within digital computing since the 1950s. The essential architecture has changed very little over that time: one or more platters contain magnetic charges that represent bits of information. These magnetic charges are read and written by an actuator arm, which moves across the disk to a specific position on the radius of the platter and then waits for the platter to rotate to the appropriate location.

Figure 8. Disk devices over the years (Photo courtesy of Paul R. Potts)

The time taken to read an item of information is the sum of the time taken to move the head into position (seek time), the time taken to rotate the item into place (rotational latency), and the time taken to transmit the item through the disk controller (transfer time).

Figure 8 illustrates that while the size and density of these devices have changed over the years, the architecture remains virtually identical. While Moore's law drives exponential growth in CPU, memory, and disk density, it does not apply to the mechanical aspects of this performance; consequently, magnetic disks have become an increasing drag on database performance over the years.

The promise of solid-state disks has led some to anticipate a day when all magnetic disks are replaced by solid-state disks. While this might someday come to pass, in the short term the economics of storage and the economics of I/O are at odds: magnetic disk technology provides a more economical medium per unit of storage, while flash technology provides a more economical medium for delivering high I/O rates and low latencies.

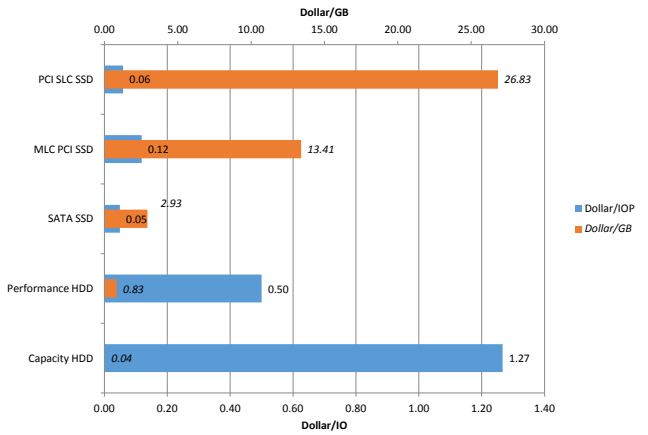

Figure 9. Economics of storage for solid-state and magnetic disk technologies

Figure 9 illustrates the two competing trends: while the cost of I/O is reduced with solid-state technology, the cost per TB increases. Various flavors of SSD (PCI/SATA and MLC/SLC) offer different price and performance characteristics in magnetic disks (15 K vs. 7 K RPM, for instance). The SSD devices that offer good economies of I/O offer poorer economies for mass storage. Of course, the cost per gigabyte for SSD is dropping rapidly, but not faster than the falling cost of magnetic disks or the growth in database storage demand—especially in the era of big data.

Since most databases include both hot and cold data—small amounts of frequently accessed data as well as large amounts of idle data—most databases will experience the best economic benefit by combining both solid-state and traditional magnetic disk technologies. This is why Exadata combines both magnetic disks and flash disks to achieve the ability to provide the optimal balance between storage economics and performance. If Exadata contained only magnetic disks, it could not provide superior

OLTP performance; if it contained only SSD, it could not offer compelling economical storage for large databases.

Moore's law also continues to drive an increase in memory capacity and decrease in memory cost. Many smaller databases can now fit entirely within the memory capacity of a single server, and certainly within the memory capacity of a cluster. For these databases an in-memory solution may be even more attractive than a full SSD architecture.

However, these advantages can only be realized fully by a non-traditional database architecture. Simply creating a massive buffer case on top of an Oracle database, for instance, will not provide benefits for all operations. Full table scans will still use direct path reads to disk and checkpoints and redo operations still need to run to a persistent storage layer.

Therefore databases that seek to be truly in memory typically use different architectural patterns. For instance in Times 10, data is guaranteed to always be in memory, and no user operation ever waits for disk I/O (although disks are still used to store snapshots and transaction logs). By default, a commit writes asynchronously to the transaction log, so that the user does not wait on the IO. Strictly speaking this violates the ACID transaction model, since a committed transaction could be lost if the database failed before the data made it to disk.

HANA employs a mix of columnar and row-based storage, and also requires persistent disks (often SSDs) for transaction log and checkpoint writes.

One of the purest in-memory databases is VoltDB. VoltDB uses a cluster of in-memory partitions that replicate data across machines. During a commit, data is written to multiple machines, ensuring that data is not lost should there be a power failure. This approach is called "K-safety."

VoltDB also eliminates latch waits by allowing only a single thread to access a partition at any moment. This is an interesting aspect of in-memory systems, since when you eliminate I/O waits you are likely to find that latch waits, which prevent simultaneous memory accesses, become the next logical bottleneck.

Oracle Database 12*c* In-Memory Option uses a combination of columnar and in-memory approaches. Data is still held in the traditional buffer cache and data files, but it can also be held in an in-memory column format. As with other column stores, there is a row store for deltas (the SMU) that buffers modifications.

Cloud: The Missing Trend?

It's been true for several years that you can add the phrase "in the cloud" to almost any technology marketing phrase to increase its value. Since cloud is such a big trend, should "databases in the cloud" or Database as a Service (DBaaS) be a key database technology trend?

Possibly—in that I tend to believe that the database architectures in the cloud are essentially driven by the database architectures that we see on premise, especially those being pioneered in the largest Web 2.0 companies. Furthermore, DBaaS uptake will be slower than other cloud categories, due to some of the unique characteristics of databases:

When the database is remote from the application it serves, network latencies become a limiting performance factor that cannot be overcome. A lot of work has gone into HTTP to make it very responsive on web pages through things like staged rendering and batching of request/reply packets; less work has gone into shaping database server–application server network traffic because it's assumed that they are both on the same local area network or on a dedicated interconnect.

For a similar reason—assumption that the application server and database server are on a private network—the database communication and authentication protocols are not as hardened as in HTTPS. Data items may be transmitted in plain text, and authentication is often restricted to one "super" account that can do anything. Consequently, opening the database listener port to the Internet potentially exposes the entire database to attack or to successful data sniffing.

Data sovereignty and other considerations apply most strongly to the database, so it's probably the last element of your on-premise servers to be moved to cloud.

None of this is to say that databases won't move into the cloud over time, but in my opinion they will follow applications into the cloud rather than lead the way. ▲

Guy Harrison is an Oracle ACE and executive director of research and development at Dell. He is the author of Oracle Performance Survival Guide (Prentice Hall, 2009) and MySQL Stored Procedure Programming (O'Reilly, with Steven Feuerstein) as well as other books, articles, and presentations on database technology. He also writes a monthly column for Database Trends and Applications *(***www.dbta.com***). He is co-author of the upcoming* Oracle Exadata Expert Handbook *(Pearson, 2015). Guy can be found on the Internet at* **www.guyharrison.net** *and on e-mail at* **guy.harrison@software.dell.com***, and he is* **@guyharrison** *on Twitter.*