Couchbase

NoSQL Database Technology

Post-relational data management for interactive software systems

Table of Contents

Summary

Interactive software (software with which a person iteratively interacts in real time) has changed in fundamental ways over the last 35 years. The "online" systems of the 1970s have, through a series of intermediate transformations, evolved into today's Web and mobile applications. These systems solve new problems for potentially vastly larger user populations, and they execute atop a computing infrastructure that has changed even more radically over the years.

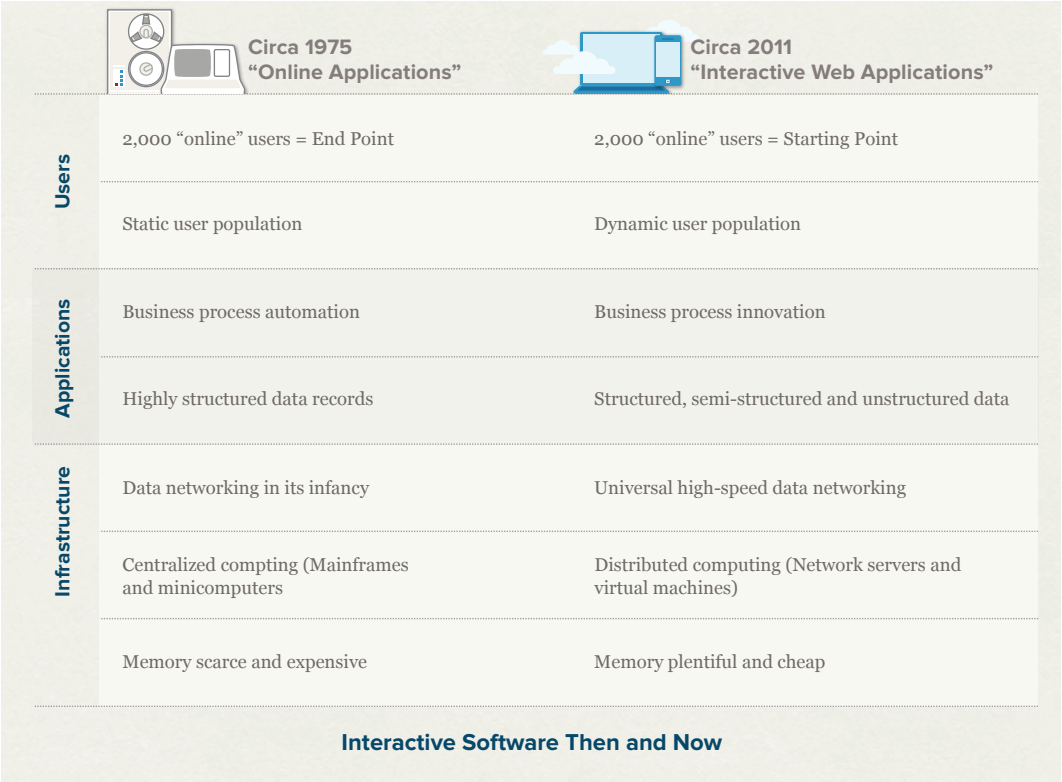
The architecture of these software systems has likewise transformed. A modern Web application can support millions of concurrent users by spreading load across a collection of application servers behind a load balancer. Changes in application behavior can be rolled out incrementally without requiring application downtime by gradually replacing the software on individual servers. Adjustments to application capacity are easily made by changing the number of application servers.

But database technology has not kept pace. Relational database technology, invented in the 1970s and still in widespread use today, was optimized for the applications, users and infrastructure of that era. In some regards, it is the last domino to fall in the inevitable march toward a fully-distributed software architecture. While a number of bandaids have extended the useful life of the technology (horizontal and vertical sharding, distributed caching and data denormalization), these tactics nullify key benefits of the relational model while increasing total system cost and complexity.

In response to the lack of commercially available alternatives, organizations such as Google and Amazon were, out of necessity, forced to invent new approaches to data management. These "NoSQL" or non-relational database technologies are a better match for the needs of modern interactive software systems. But not every company can or should develop, maintain and support its own database technology. Building upon the pioneering research at these and other leading-edge organizations, commercial suppliers of NoSQL database technology have emerged to offer database technology purpose-built to enable the cost-effective management of data behind modern Web and mobile applications.

Interactive software has changed

As Table 1 below shows, there are fundamental differences in the users, applications and underlying infrastructure between interactive software systems of the 1970s and those being built today.



	Circa 1975 "Online Applications"	Circa 2011 "Interactive Web Applications"
Users	2,000 "online" users = End Point	2,000 "online" users = Starting Point
	Static user population	Dynamic user population
Applications	Business process automation	Business process innovation
	Highly structured data records	Structured, semi-structured and unstructured data
Infrastructure	Data networking in its infancy	Universal high-speed data networking
	Centralized compting (Mainframes and minicomputers	Distributed computing (Network servers and virtual machines)
	Memory scarce and expensive	Memory plentiful and cheap

Interactive Software Then and Now

Users

In 1975, an interactive software system with 2,000 users represented the pinnacle of scale. Few organizations built, deployed and supported such systems. American Airlines Sabre® System (first installed in a travel agency in 1976) and Bank of America's branch banking automation system represent two notable interactive software systems that scaled to these heights. But these were exceptions.

Today, applications accessed via the public Web have a potential user base of over two billion users. Whether an online banking system, a social networking or gaming application, or an e-commerce application selling goods and services to the public, there are innumerable examples of software systems that routinely support a population of users many orders of magnitude beyond the largest of the 1970s. A system with only 2,000 users is the exception now, assuming the application is not an abject failure.

There is also user growth and churn today not seen in systems of the 1970s. Once rolled out, the number of travel agents or tellers added to, or removed from, these systems was highly predictable and relatively easy to manage (albeit somewhat manually and at measured

pace). Users worked during well-defined office hours, providing windows of opportunity for scheduled system downtime and maintenance.

Today, Web applications can serve a global population of users 24 hours a day, 365 days per year. A newly launched software system can grow from no users to over a million users almost literally overnight. Not all users are active on these systems at any given time, and some users may use an application only a few times, never to return, and without providing notice of their intent to leave.

Applications

In 1975, interactive software systems were primarily designed to automate what were previously tedious, paper-based business processes – teller transactions, flight reservations, stock trades. These "transactions" typically mirrored what clerical employees had been doing "by hand" for decades – filling in fields on a structured business form, then filing or sending forms to other employees who would tally them, update impacted ledgers and notate files to effect "transactions." Online transaction processing systems accelerated these tasks and reduced the probability of error, but in most cases they were automating versus innovating.

Versus simply automating long-standing manual business processes, today's Web applications are breaking new ground in every direction. They are changing the nature of communication, shopping, advertising, entertainment and relationship management. But they are works in progress. There are no old business forms to simply mimic, or processes to study and automate. It may be trite, but change is truly the only constant in these systems. And a database has to be flexible enough to change with them.

Infrastructure

Perhaps the most obvious difference between interactive software then and now is the infrastructure atop which they execute.

Centralization characterized the computing environment in the 1970s – mainframes and minicomputers with shared CPU, memory and disk subsystems were the norm. Computer networking was in its infancy. Memory was an expensive, scarce resource. Today, distributed computing is the norm. Within a datacenter, servers and virtual machines are interconnected via high-speed data networks. Users of software systems access them from even more widely distributed desktop, laptop and mobile computing devices.

The IBM System/360 Model 195 was "the most powerful computer in IBM's product line" from August 1969 through the mid-1970s. The most powerful configuration of this system shipped with 4MB of main (core) memory. Today, a single high-end microprocessor can have more L1 cache memory on the processor die itself, with support for many orders of magnitude more main memory.

Application architecture has changed

Directly addressing the aforementioned changes, and in contrast to the scale-up, centralized approach of circa 1975 interactive software architecture, modern Web applications are built to scale out – simply add more commodity Web servers behind a load balancer to support more users. Scaling out is also a core tenet of the increasingly important cloud computing model, in which virtual machine instances can be easily added or removed to match demand.

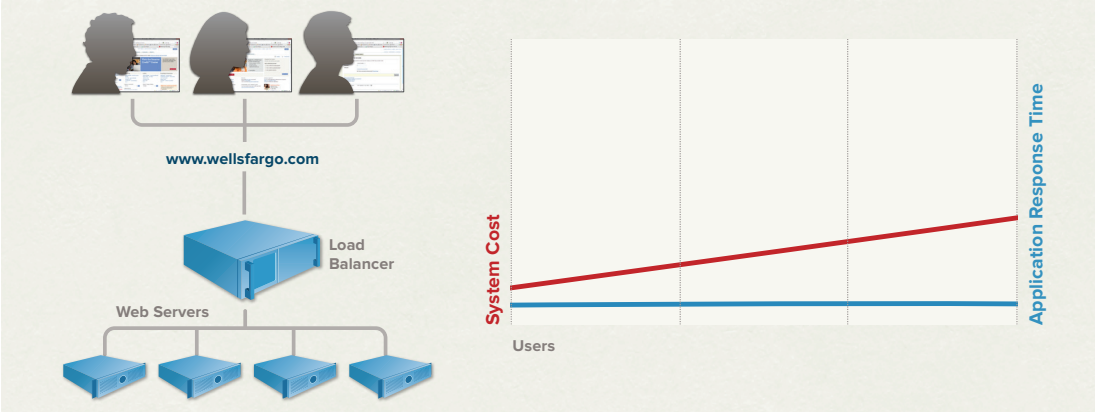


Figure 1: Web Application – Logic Scales Out. To support more users for a Web application, you simply add more commodity Web servers. As a result, system cost expands linearly with linear increases in users, and performance remains constant. This model scales out indefinitely for all practical purposes.

The cost and performance curves are obviously attractive, but ultimately, flexibility is the big win in this approach.

As users come and go, commodity servers (or virtual machines) can be quickly added or removed from the server pool, matching capital and operating costs to the difficult-to-predict size and activity level of the user population. And by distributing the load across many servers, even across geographies, the system is inherently fault-tolerant, supporting continuous operations.

As application needs change, new software can be gradually rolled out across subsets of the overall server pool. Facebook, as an example, slowly dials up new functionality by rolling out new software to a subset of their entire application server tier (and user population) in a stepwise manner. If issues crop up, servers can be quickly reverted to the previous known good build. All this can be done without ever taking the application "offline."

Database architecture has not kept pace

In contrast to the sweeping changes in application architecture, relational database (RDBMS) technology, a "scale-up" technology that has not fundamentally changed in over 40 years, continues to be the default choice for holding data behind Web applications. Not surprisingly, RDBMS technology reflects the realities (users, applications, and infrastructure) of the environment that spawned it.

Because it is a technology designed for the centralized computing model, to handle more users one must get a bigger server (increasing CPU, memory and I/O capacity) (see Figure 2). Big servers tend to be highly complex, proprietary, and disproportionately expensive pieces of engineered machinery, unlike the low-cost, commodity hardware typically deployed in Web- and cloud-based architectures. And, ultimately, there is a limit to how big a server one can purchase, even given an unlimited willingness and ability to pay.

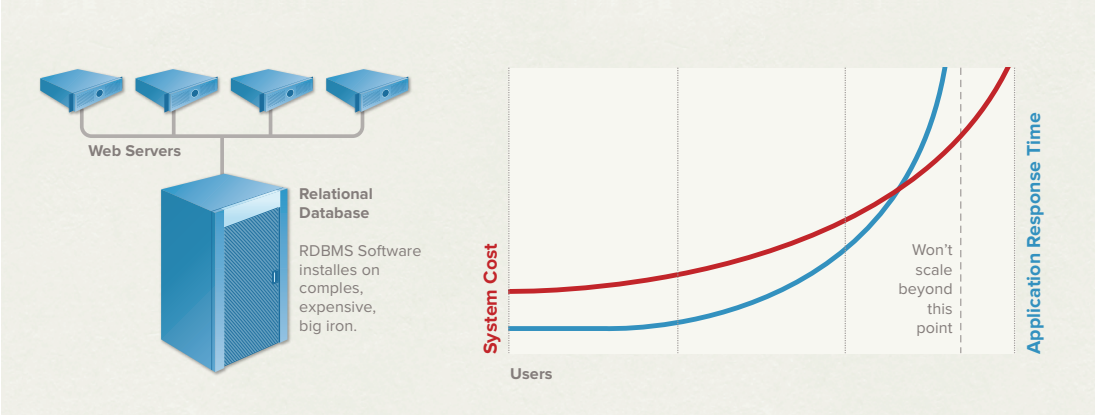


Figure 2: Web Application – RDBMS Scales Up. To support more users, you must get a bigger database server for your RDBMS. As a result, system cost grows exponentially with linear increases in users, and application response time degrades asymptotically.

While the scaling economics are certainly inferior to the model now employed at the application logic tier, it is once again flexibility (or lack thereof) that is the "high-order bit" to consider.

Upgrading a server is an exercise that requires planning, acquisition and application downtime to complete. Given the relatively unpredictable user growth rate of modern software systems, inevitably there is either over- or under-provisioning of resources. Too much and you've overspent, too little and users can have a bad application experience or the application can outright fail. And with all the eggs in a single basket, fault tolerance and high-availability strategies are critically important to get right.

Perhaps the least obvious, but arguably the most damaging downside of using RDBMS technology behind modern interactive software systems is the rigidity of the database schema. As noted previously, we are no longer simply automating long-standing and well-understood paper-based processes, where database record formats are pre-defined and largely static. But RDBMS technology requires the strict definition of a "schema" prior to storing any data into the database. Changing the schema once data is inserted is A Big Deal. Want to start capturing new information you didn't previously consider? Want to make rapid changes to application behavior requiring changes to data formats and content? With RDBMS technology, changes like these are extremely disruptive and therefore are frequently avoided – the opposite behavior desired in a rapidly evolving business and market environment.

Tactics to extend the useful scope of RDBMS technology

In an effort to address the shortcomings of RDBMS technology when used behind modern interactive software systems, developers have adopted a number of "bandaid" tactics.

Sharding

The RDBMS data model and transaction mechanics fundamentally assume a centralized computing model – shared CPU, memory and disk. If the data for an application will not fit on a single server or, more likely, if a single server is incapable of maintaining the I/O throughput required to serve many users simultaneously, then a tactic known as sharding is frequently employed. In this approach an application will implement some form of data partitioning to manually spread data across servers. For example, users that live west of the Mississippi River may have their data stored in one server, while those who live east of the river will be stored in another.

While this does work to spread the load, there are undesirable consequences to the approach.

1. When you fill a shard, it is highly disruptive to re-shard. When you fill a shard, you have to change the sharding strategy in the application itself. For example, if you had partitioned your database by placing all accounts east of the Mississippi on one server and all accounts west in another and then reach the limits of their capacity, you must change the sharding approach which means changing your application. Where previously the application had to know "this is an east of the Mississippi customer and thus I need to look in this database server," now it must know "if it is east of the Mississippi and below the Mason-Dixon Line, I need to look in that server now."

2. You lose some of the most important benefits of the relational model. You can't do "joins" across shards – if you want to find all customers that have purchased a pair of wool socks but haven't purchased anything in over 6 months, you must run a query on every server and piece the results together in application software. In addition, you can't do cross-node locking when making updates. So one must ensure all data that could need to be atomically operated on is resident on a single server, unless using an external TP monitor system or complex logic in the application itself.

3. You have to create and maintain a schema on every server. If you have new information you want to collect, you must modify the database schema on every server, then normalize, retune and rebuild the tables. What was hard with one server is a nightmare across many. For this reason, the default behavior is to minimize the collection of new information.

Denormalizing

Before storing data in an RDBMS, a schema must be created defining precisely what data can be stored in the database and the relationships between data elements. Data is decomposed into a "normal form" and a record is typically spread across many interlinked tables. In order to update a record, all these tables must be locked down and updated atomically, lest the database become corrupted. This approach substantially limits the latency and throughput of concurrent updates and is, for most practical purposes, impossible to implement across server boundaries.

To support concurrency and sharding, data is frequently stored in a denormalized form when an RDBMS is used behind Web applications. This approach potentially duplicates data in the database, requiring updates to multiple tables when a duplicated data item is changed, but it reduces the amount of locking required and thus improves concurrency.

At the limit the relational schema is more or less abandoned entirely, with data simply stored in key-value form, where a primary key is paired with a data "blob" that can hold any data. This approach allows the type of information being stored in the database to change without requiring an update to the schema. It makes sharding much easier and allows for rapid changes in the data model. Of course, just about all relational database functionality is lost in the process (though if the database is sharded, much of the functionality was already lost). Notwithstanding all these problems, many organizations are using relational technology in precisely this manner given the familiarity of specific RDBMS technologies to developers and operations teams, and, until recently, the lack of good alternatives.

Distributed caching

Another tactic used to extend the useful scope of RDBMS technology has been to employ distributed caching technologies, such as Memcached. Today, Memcached is a key ingredient in the data architecture behind 18 of the top 20 largest (by user count) Web applications, including Google, Wikipedia, Twitter, YouTube, Facebook, Craigslist, and tens of thousands of other corporate and consumer Web applications. Most new Web applications now build Memcached into their data architecture from day one.

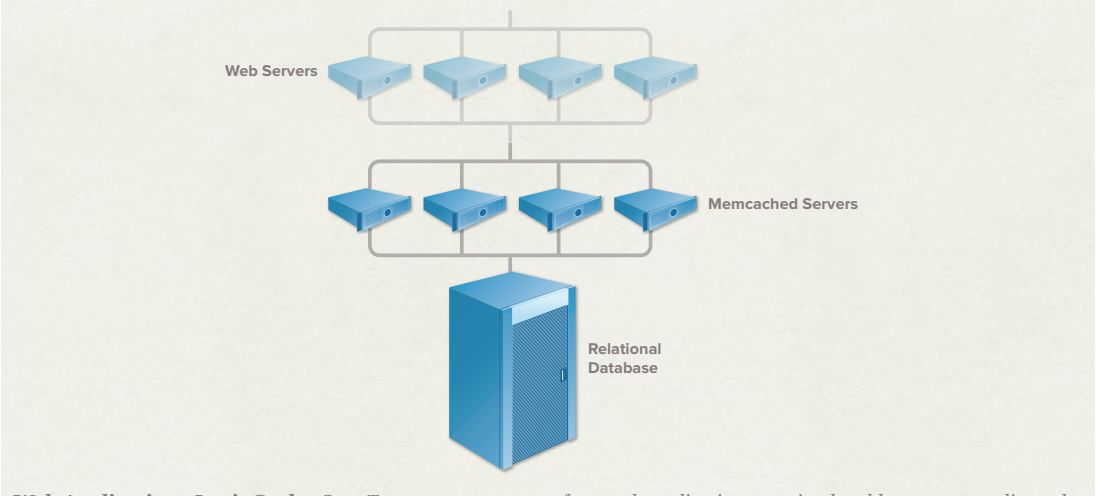


Figure 3: Memcached distributed caching technology extends the useful life of RDBMS technology behind interactive Web applications, spreading data across servers and leveraging the availability and performance of main memory.

Memcached builds on two of the most important infrastructure transitions over the last 40 years: the shift to distributed computing atop high-speed data networks, and advances in main memory (RAM) price/performance.

Memcached "sits in front" of an RDBMS system, caching recently accessed data in memory and storing that data across any number of servers or virtual machines. When an application needs access to data, rather than going directly to the RDBMS, it first checks Memcached to see if the data is available there; if it is not, then the database is read by the application and stored in Memcached for quick access next time it is needed.

While useful and effective to a point, Memcached and similar distributed caching technologies used for this purpose are no panacea and can even create problems of their own:

1. Accelerates only data reads. Memcached was designed to accelerate the reading of data by storing it in main memory, but it was not designed to permanently store data. Memcached stores data in memory. If a server is powered off or otherwise fails, or if memory is filled up, data is lost. For

this reason, all data writes must be done on the RDBMS. Because all data is written to the RDBMS, the application will simply read it from the database if it is not present in the cache. While offloading database reads helps many applications, there is still a write bottleneck that can be significant.

2. Cold cache thrash. Many applications become so dependent on Memcached performance that the loss of even a single server in a Memcached cluster can result in serious consequences for the users of an application. As the application seeks but doesn't find data in the caching tier, it is forced to read the data from the RDBMS. With enough cache misses the RDBMS can be flooded with read requests. The result can be a delay in both reads and writes of data which can lead to application time-outs, unacceptably slow application response times and user dissatisfaction.

3. Another tier to manage. It should be obvious that inserting another tier of infrastructure into the architecture to address some (but not all) of the failings of RDBMS technology in the modern interactive software use case can create its own set of problems: more capital costs, more operational expense, more points of failure, more complexity.

"NoSQL" database technologies

The techniques used to extend the useful scope of RDBMS technology fight symptoms but not the disease itself. Sharding, denormalizing, distributed caching and other tactics all attempt to paper over one simple fact: RDBMS technology is a forced fit for modern interactive software systems.

Because vendors of RDBMS technology have little incentive to disrupt a technology generating billions of dollars for them annually, application developers were forced to take matters into their own hands. Google (Big Table) and Amazon (Dynamo) are two leading web application developers who invented, developed and depend on their own database technologies. These "NoSQL" databases, each eschewing the relational data model, are a far better match for the needs modern interactive software systems.

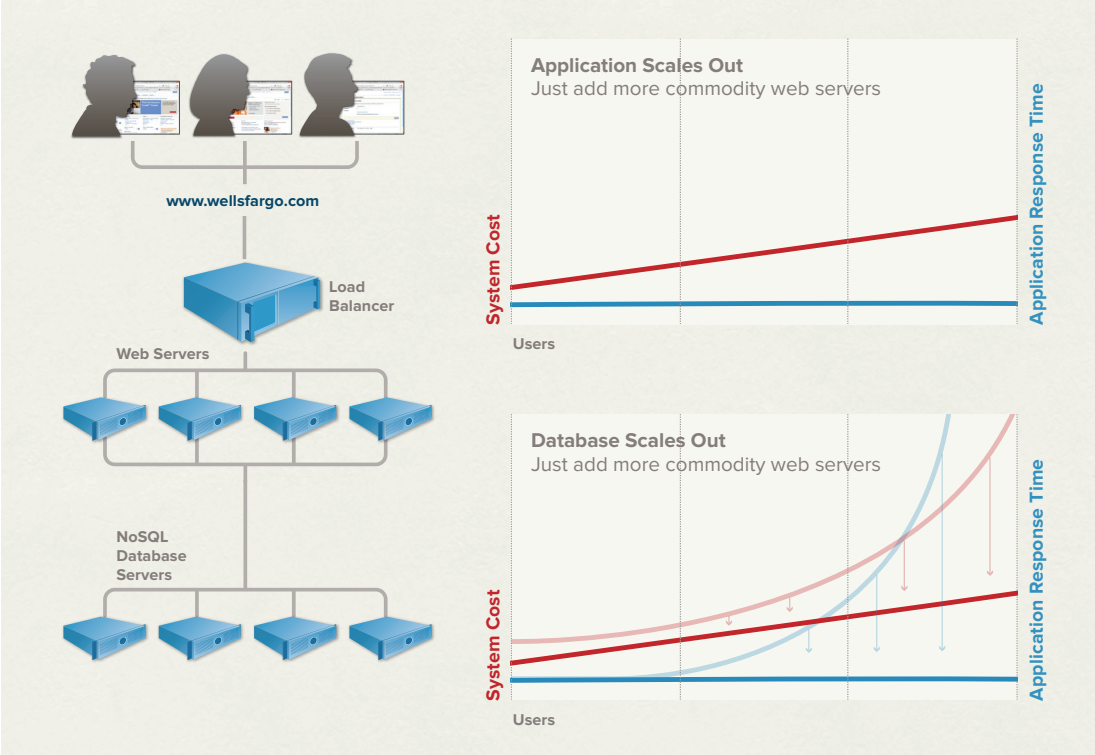


Figure 4: In contrast to the non-linear increase in total system cost and asymptotic degradation of performance previously seen with RDBMS technology, NoSQL database technology flattens both curves.

While implementations differ, NoSQL database management systems share a common set of characteristics:

1. No schema required. Data can be inserted in a NoSQL database without first defining a rigid database schema. As a corollary, the format of the data being inserted can be changed at any time, without application disruption. This provides immense application flexibility, which ultimately delivers substantial business flexibility.

2. Auto-sharding (sometimes called "elasticity"). A NoSQL database automatically spreads data across servers, without requiring applications to participate. Servers can be added or removed from the data layer without application downtime, with data (and I/O) automatically spread across the servers. Most NoSQL databases also support data replication, storing multiple copies of data across the cluster, and even across data centers, to ensure high availability and support disaster recovery. A properly managed NoSQL database system should never need to be taken offline, for any reason, supporting 24x7x365 continuous operation of applications.

3. Distributed query support. “Sharding” an RDBMS can reduce, or eliminate in certain cases, the ability to perform complex data queries. NoSQL database systems retain their full query expressive power even when distributed across hundreds or thousands of servers.

4. Integrated caching. To reduce latency and increase sustained data throughput, advanced NoSQL database technologies transparently cache data in system memory. This behavior is transparent to the application developer and the operations team, in contrast to RDBMS technology where a caching tier is usually a separate infrastructure tier that must be developed to, deployed on separate servers, and explicitly managed by the ops team.

Mobile application data synchronization

While Web applications have been struggling with data management challenges for more then ten years, mobile applications have exploded in popularity over the last couple of years, bringing data management challenges of their own.

Apple iOS applications best represent this new genre of interactive software system. Launched on July 10, 2008, the Apple App Store allows iPhone, iPod Touch and iPad users to browse and download applications built with the Apple iOS SDK. As of March 2011 the App Store offers over 300,000 unique applications, over 10 billion applications have been downloaded, and over $2 billion in payments have been made by Apple to the developers of these applications.

These “native” mobile applications (and similar applications available for Android, Blackberry and Windows Mobile devices) execute on the device and typically store their data on the device itself, allowing operation whether or not connected to the Internet. Unlike the data management needs of a Web application, these systems obviously do not need to support potentially millions of concurrent users and thus don’t require auto-sharding, distributed query support or caching of data (data is usually stored in high-speed non-volatile memory versus on disk media).

But these systems present a data synchronization challenge. Because a mobile device can be easily lost or damaged, a reliable mechanism for data backup is required. Many mobile applications also have sister Web applications. A project management system, for example, may have both a native iPhone application and a Web application interface. When using the native iPhone application, a user may make changes to a local copy of the data while disconnected from the Internet. When a connection is re-established, the data should be synchronized with the Web application to ensure data consistency across views.

Some NoSQL database management systems are beginning to support synchronization of data between mobile devices and database clusters deployed in a data center (or “in the cloud”). Because many Web application developers are also delivering native mobile versions of their applications, this functionality is increasingly attractive to developers evaluating database alternatives.

Open source and commercial NoSQL database technologies

Unlike Google and Amazon, few companies can or should build and maintain their own database technology. But the need for a new approach is nearly universal. The vast majority of new interactive software systems are Web applications with the characteristics and needs described in this document. These systems are being built by organizations of all sizes and across all industries. Interactive software is fundamentally changing, and the database technology used to support these systems is changing too.

A number of commercial and open source database technologies such as Couchbase (a database combining the leading NoSQL data management technologies CouchDB, Membase and Memcached), MongoDB, Cassandra, Riak and others are now available and increasingly represent the “go to” data management technology behind new interactive Web applications.

For more information on NoSQL use cases, visit www.couchbase.com/why-nosql/use-cases