NoSQL

GREG BURD

Greg Burd is a Developer Advocate for Basho Technologies, makers of Riak. Before Basho, Greg spent nearly ten years as the product manager for Berkeley DB at Sleepycat Software and then at Oracle. Previously, Greg worked for NeXT Computer, Sun Microsystems, and KnowNow. Greg has long been an avid supporter of open source software.
greg@basho.com

Choosing between databases used to boil down to examining the differences between the available commercial and open source relational databases. The term "database" had become synonymous with SQL, and for a while not much else came close to being a viable solution for data storage. But recently there has been a shift in the database landscape. When considering options for data storage, there is a new game in town: NoSQL databases. In this article I'll introduce this new category of databases, examine where they came from and what they are good for, and help you understand whether you, too, should be considering a NoSQL solution in place of, or in addition to, your RDBMS database.

What Is NoSQL?

The only thing that all NoSQL solutions providers generally agree on is that the term "NoSQL" isn't perfect, but it is catchy. Most agree that the "no" stands for "not only"—an admission that the goal is not to reject SQL but, rather, to compensate for the technical limitations shared by the majority of relational database implementations. In fact, NoSQL is more a rejection of a particular software and hardware architecture for databases than of any single technology, language, or product. Relational databases evolved in a different era with different technological constraints, leading to a design that was optimal for the typical deployment prevalent at that time. But times have changed, and that once successful design is now a limitation. You might hear conversations suggesting that a better term for this category is NoRDBMS or half a dozen other labels, but the critical thing to remember is that NoSQL solutions started off with a different set of goals and evolved in a different environment, and so they are operationally different and, arguably, provide better-suited solutions for many of today's data storage problems.

Why NoSQL?

NoSQL databases first started out as in-house solutions to real problems in companies such as Amazon Dynamo [1], Google BigTable [2], LinkedIn Voldemort [3], Twitter FlockDB [4], Facebook Cassandra [5], Yahoo! PNUTS [6], and others. These companies didn't start off by rejecting SQL and relational technologies; they tried them and found that they didn't meet their requirements. In particular, these companies faced three primary issues: unprecedented transaction volumes, expectations of low-latency access to massive datasets, and nearly perfect service availability while operating in an unreliable environment. Initially, companies tried the traditional approach: they added more hardware or upgraded to faster hardware as it became available. When that didn't work, they tried to scale existing relational solutions by simplifying their database schema, de-normalizing the schema, relaxing durability and referential integrity, introducing various query caching layers, separating read-only from write-dedicated replicas, and, finally, data partitioning in an attempt to address these new requirements. Although each of these techniques extended the functionality of existing relational technologies, none fundamentally addressed the core limitations, and they all introduced additional overhead and technical tradeoffs. In other words, these were good band-aids but not cures.

A major influence on the eventual design of NoSQL databases came from a dramatic shift in IT operations. When the majority of relational database technology was designed, the predominant model for hardware deployments involved buying large servers attached to dedicated storage area networks (SANs). Databases were designed with this model in mind: They expected there to be a single machine with the responsibility of managing the consistent state of the database on that system's connected storage. In other words, databases managed local data in files and provided as much concurrent access as possible given the machine's hardware limitations. Replication of data to scale concurrent access across multiple systems was generally unnecessary, as most systems met design goals with a single server and reliability goals with a hot stand-by ready to take over query processing in the event of master failure. Beyond simple failover replication, there were only a few options, and they were all predicated on this same notion of completely consistent centralized data management. Technologies such as two-phase commit and products such as Oracle's RAC were available, but they were hard to manage, very expensive, and scaled to only a handful of machines. Other solutions available included logical SQL statement-level replication, single-master multi-replica log-based replication, and other home-grown approaches, all of which have serious limitations and generally introduce a lot of administrative and technical overhead. In the end, it was the common architecture and design assumptions underlying most relational databases that failed to address the scalability, latency, and availability requirements of many of the largest sites during the massive growth of the Internet.

Given that databases were centralized and generally running on an organization's most expensive hardware containing its most precious information, it made sense to create an organizational structure that required at least a 1:1 ratio of database administrators to database systems to protect and nurture that investment. This, too, was not easy to scale, was costly, and could slow innovation.

A growing number of companies were still hitting the scalability and performance wall even when using the best practices and the most advanced technologies of the time. Database architects had sacrificed many of the most central aspects of a relational database, such as joins and fully consistent data, while introducing many complex and fragile pieces into the operations puzzle. Schema devolved from many interrelated fully expressed tables to something much more like a simple key/value look-up. Deployments of expensive servers were not able to keep up with demand. At this point these companies had taken relational databases so far outside their intended use cases that it was no wonder that they were unable to meet performance requirements. It quickly became clear to them that they could do much better by building something in-house that was tailored to their particular workloads. These in-house custom solutions are the inspiration behind the many NoSQL products we now see on the market.

NoSQL's Foundations

Companies needed a solution that would scale, be resilient, and be operationally efficient. They had been able to scale the Web (HTTP) and dynamic content generation and business logic layers (Application Servers), but the database continued to be the system's bottleneck. Engineers wanted databases to scale like Web servers—simply add more commodity systems and expect things to speed up at a nearly linear rate—but to do that they would have to make a number of tradeoffs. Luckily, due to the large number of compromises made when attempting to scale their existing relational databases, these tradeoffs were not so foreign or distasteful as they might have been.

Consistency, Availability, Partition Tolerance (CAP)

When evaluating NoSQL or other distributed systems, you'll inevitably hear about the "CAP theorem." In 2000 Eric Brewer proposed the idea that in a distributed system you can't continually maintain perfect consistency, availability, and partition tolerance simultaneously. CAP is defined by Wikipedia [7] as:

Consistency: all nodes see the same data at the same time
Availability: a guarantee that every request receives a response about whether it was successful or failed
Partition tolerance: the system continues to operate despite arbitrary message loss

The theorem states that you cannot simultaneously have all three; you must make tradeoffs among them. The CAP theorem is sometimes incorrectly described as a simple design-time decision—"pick any two [when designing a distributed system]"—when in fact the theorem allows for systems to make tradeoffs at run-time to accommodate different requirements. Too often you will hear something like, "We trade consistency (C) for AP," which can be true but is often too broad and exposes a misunderstanding of the constraints imposed by the CAP theorem. Look for systems that talk about CAP tradeoffs relative to operations the product provides rather than relative to the product as a whole.

Relaxing ACID

Anyone familiar with databases will know the acronym ACID, which outlines the fundamental elements of transactions: atomicity, consistency, isolation, and durability. Together, these qualities define the basics of any transaction. As NoSQL solutions developed it became clear that in order to deliver scalability it might be necessary to relax or redefine some of these qualities, in particular consistency and durability. Complete consistency in a distributed environment requires a great deal of communication involving locks, which force systems to wait on each other before proceeding to mutate shared data. Even in cases where multiple systems are generally not operating on the same piece of data, there is a great deal of overhead that prevents systems from scaling.

To address this, most NoSQL solutions choose to relax the notion of complete consistency to something called "eventual consistency." This allows each system to make updates to data and learn of other updates made by other systems within a short period of time, without being totally consistent at all times. As changes are made, tools such as vector clocks are used to provide enough information to reason about the ordering of those changes based on an understanding of the causality of the updates. For the majority of systems, knowing that the latest consistent information will eventually arrive at all nodes is likely to be enough to satisfy design requirements.

Another approach is optimistic concurrency control, using techniques such as multi-version concurrency control (MVCC). Such techniques allow for consistent reading of data in one transaction with concurrent writing in another transaction but do not address write conflicts and can introduce more transaction retries when transactions overlap or are long-running.

Both eventual consistency and MVCC require that programmers think differently about the data they are managing in the application layer. Both introduce the potential that data read in a transaction may not be entirely up-to-date even though it may be consistent.

Achieving durability has long been the bottleneck for database systems. It's easy to understand why writing to disk slows down the database; disk access times are orders of magnitude slower than writes to memory. Most database solutions recognize the potential performance tradeoffs related to durability and offer ways to tune writes to match application requirements, balancing durability against speed. Be careful, as this can mean leaving a small window of opportunity where seemingly committed transactions can be lost under certain failure conditions. For example, some will consider an update durable when it has been written into the memory of some number of systems. If those systems were to lose power, that commit would be lost. Network latencies are much faster than disk latencies, and by having data stored on more than one system the risk of losing information due to system failure is much lower. This definition of durability is very different from what you've come to expect from a relational database, so be cautious. For instance, Redis, MongoDB, HBase, and Riak range from minimally durable to highly durable, in that order. As with any other database, when evaluating NoSQL solutions, be sure to know exactly what constitutes durability for that product and how that impacts your operational requirements.

Data and Access Model

The relational data model with its tables, views, rows, and columns has been very successful and can be used to model most data problems. By using constraints, triggers, fine-grained access control, and other features, developers can create systems that enforce structure and referential integrity and that secure data. These are all good things, but they come at a price. First, there is no overlap in the data representation in SQL databases and in programming languages; each access requires translation to and from the database. Object to relational mapping (ORM) solutions exist to transparently transform, store, and retrieve object graphs into relational databases, and although they work well, they introduce overhead into a process and slow it down further. This is an impedance mismatch that introduces overhead where it is least needed. Second, managing global constraints in a distributed environment is tricky and involves creating barriers (locks) to coordinate changes so that these constraints are met. This introduces network overhead and sometimes can stall progress in the system.

NoSQL solutions have taken a different approach. In fact, NoSQL solutions diverge quite a bit from one another as well as from the RDBMS norm. There are three main data representation camps within NoSQL: document, key/value, and graph. There is still a fairly diverse set of solutions within each of these categories. For instance, Riak, Redis, and Cassandra are all key/value databases, but with Cassandra you'll find a slightly more complex concept, based on Google's BigTable, called "column families," which is very different from the more SimpleDB-like "buckets containing key/value pairs" approach of the other two.

Document-oriented databases store data in formats that are more native to the languages and systems that interact with them. JavaScript Object Notation (JSON) and its binary encoded equivalent, BSON, are used as a simple dictionary/array representation of data. MongoDB stores BSON documents and provides a JSON-encoded query syntax for document retrieval. For all intents and purposes, JSON has replaced most of the expected use cases for XML, and although XML has other advantages (XQuery, typed and verifiable schema), JSON is the de facto Web-native format for data on the wire and now can be stored, indexed, and queried in some NoSQL databases.

Neo4j is a graph-based NoSQL database that stores information about nodes and edges and provides simple, highly optimized interfaces to examine the connectedness of any part of the graph. With the massive growth of social networking sites and the value of understanding relationships between people, ideas, places, etc., this highly specialized subset of data management has received a great deal of attention lately. Look for more competitors in this space as the use cases for social networks continue to grow.

Global constraints are generally not available or very rudimentary. The reasoning for not introducing more than basic constraints is simple: anything that could potentially slow or halt the system violates availability, responsiveness, and latency requirements of these systems. This means that applications using NoSQL solutions will have to build logic above the database that monitors and maintains any additional consistency requirements.

A crucial part for the success of the relational market as a whole has been the relative uniformity of SQL solutions. Certainly there are differences between vendors and a non-trivial penalty when migrating from one SQL solution to another, but at a high level, they all start with the same APIs (ODBC, JDBC, and SQL), and that allows a universe of tools and a population of experts to flourish. NoSQL solutions also diverge when it comes to access, encoding, and interaction with the server. While some NoSQL products provide a HTTP/REST API, others provide simple client libraries or network protocols that use widely available data encoding libraries such as Thrift and Protobufs. Some provide multiple methods of access, and the more successful NoSQL solutions will generally have pre-built integrations with most of the popular languages, frameworks, and tools, so that this is not as big an issue as it may seem.

This diversity is representative of the fact that NoSQL is a broad category where there are no standards such as SQL to unify vendors. Customers should therefore choose carefully—vendor lock-in is a given at this point in the NoSQL market. That said, most leading NoSQL solutions are open source, which does defray some of the risk related to the sponsoring company changing hands or going out of business. In the end, though, moving from one NoSQL solution to another will be a time-consuming mistake that you should try at all costs to avoid.

Distributed Data, Distributed Processing

NoSQL solutions are generally designed to manage large amounts of data, more than you would store on any single system, and so all generally have some notion of partitioning (or sharding) data across the storage found on multiple servers rather than expecting a centrally connected SAN or networked file system. The benefits of doing this transparently are scalability and reliability. The additional reliability comes when partitions overlap, keeping redundant copies of the same data at multiple nodes in the system. Not all NoSQL systems do this. The drawback will be some amount of duplicated data and costs associated with managing consistency across these partitions. In addition it is critical to understand the product's approach to distributing data. Is it a master/replica, master/master, or distributed peers? Where are the single points of failure? Where are there potential bottlenecks? When reviewing the NoSQL solutions it's important not to gloss over the details, and be sure to run extensive in-house tests where you introduce all manner of failure conditions into your testbed. The good NoSQL solutions will prove resilient even when operating in punishing environments, whereas the less mature may lose data or stop operating when too many unforeseen conditions arise.

In addition to distributing data, many NoSQL solutions offer some form of distributed processing, generally based on MapReduce. This can be a powerful tool when used correctly, but again the key when evaluating NoSQL solutions is to dig into the details and understand exactly what a vendor means by "we support MapReduce."

NoSQL in Practice

There are many products that now claim to be part of the NoSQL database market, far too many to mention here or describe in any detail. That said, there are a few with which you should become familiar, as they are likely to become long-term tools everyone uses. Let's examine the leading products broken down by category.

	MongoDB	**CouchDB**	**Riak**	**Redis**	**Voldemort**	**Cassandra**	**HBase**
Language	C++	Erlang	Erlang	C++	Java	Java	Java
License	AGPL	Apache	Apache	BSD	Apache	Apache	Apache
Model	Document	Document	Key/value	Key/value	Key/value	Wide Column	Wide Column
Protocol	BSON	HTTP/REST	HTTP/REST or TCP/ Protobufs	TCP		TCP/Thrift	HTTP/REST or TPC/Thrift
Storage	Memory mapped b-trees	COW-BTree	Pluggable: InnoDB, LevelDB, Bitcask	In memory, snapshot to disk	Pluggable: BDB, MySQL, in-memory	Memtable/ SSTable	HDFS
Inspiration		Dynamo	Dynamo		Dynamo	BigTable, Dynamo	BigTable
Search	Yes	No	Yes	No	No	Yes	Yes
MapReduce	Yes	No	Yes	No	No	Yes	Yes

Here are a few company use cases where NoSQL is a critical component of the data architecture.

Facebook: HBase for Messages

When Facebook decided to expand its messaging services infrastructure [8] to encompass email, text, instant messages, and more, it knew it had to have a distributed fault-tolerant database able to manage petabytes of messages and a write-dominated workload. It needed something to handle their existing service of "over 350 million users sending over 15 billion person-to-person messages per month" and a chat service that "supports over 300 million users who send over 120 billion messages per month," and to allow plenty of room for growth. Although they considered MySQL, a solution with which they have extensive experience [9], they created Cassandra [10]—an open source project combining elements of Google's BigTable [2] and Amazon's Dynamo [1] designs—and Apache HBase, a distributed database that closely mimics Google's BigTable implementation and is tightly integrated with Apache Hadoop and ZooKeeper. Facebook favored the strong consistency model, automatic failover, load-balancing, compression, and MapReduce support of HBase for their new production messaging service.

Craigslist: MongoDB for Archived Postings

Craigslist recently migrated over 2 billion archived postings from its MySQL clusters into a set of replicated MongoDB servers [11]. They still use MySQL for all active postings on their site, but these days when you log in and review old posts that are now expired, you are accessing their new MongoDB-based service. For Craigslist, scalability and reliability were central requirements, but one of the more interesting features they gained by going to MongoDB was schema flexibility. Their MySQL databases were so large that any schema change (ALTER TABLE) would take around two months to complete across the replicated database set. Contrast that with MongoDB, where data is stored as JSON documents with no schema enforcement at all. By separating archived postings from live postings, Craigslist simplified their architecture and made it easier to change their production schema as requirements changed.

Conclusions and Advice

The SQL vs. NoSQL debate will continue, and both sides will benefit from this competition in a market that was stagnant for far too long. I think that as the dust settles it will become evident that NoSQL solutions will work alongside SQL solutions, each doing what they do best. Facebook, Twitter, and many other companies are integrating NoSQL databases into their infrastructure right alongside SQL databases. Each has its strengths and weaknesses; neither will entirely displace the other. Some future SQL databases may start to take on features only found in NoSQL, such as elasticity and an ability to scale out to large amounts of commodity hardware. The demand for SQL will not go away anytime soon, nor will the reality of today's more distributed, virtualized, and commodity-based IT infrastructure. As more companies begin capturing and analyzing an increasing amount of data about their customers, the need for databases that can efficiently manage and analyze that kind of data will only grow.

The key to making a good decision in this market is to remain as objective and open-minded as possible while evaluating these products and talking to their vendors. Recognize that this is a new market, and that right now the market leaders are working hard to cement their positions. Use that to your advantage by asking hard questions and expecting detailed answers. Never decide on a solution without having first put that vendor's product through rigorous testing using your data on your systems. Be sure to interact with and learn more about the vendor; they are all interested in building relationships with you and will generally go out of their way to help you with their product. Spend some time understanding their particular views on eventual consistency, durability, CAP, and replication, and be certain you understand how those tradeoffs will impact your design. Make sure that they support enough tools to meet your needs today and tomorrow. Look for a solid community to bolster your development. In the end, be sure that the solution you pick is going to support you as your product matures. Make sure the vendor has shared their road map and explained how they will help you move from one version to the next, because this area of technology is far from static.

References

[1] Amazon Dynamo: http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html.

[2] Google BigTable: http://labs.google.com/papers/bigtable.html.

[3] LinkedIn Voldemort: http://project-voldemort.com/.

[4] Twitter FlockDB: http://engineering.twitter.com/2010/05/introducing-flockdb.html.

[5] Facebook Cassandra: http://cassandra.apache.org/.

[6] Yahoo! PNUTS: http://www.brianfrankcooper.net/pubs/pnuts.pdf.

[7] CAP theorem: http://en.wikipedia.org/wiki/CAP_theorem.

[8] The Underlying Technology of Messages: https://www.facebook.com/notes/facebook-engineering/the-underlying-technology-of-messages/454991608919.

[9] MySQL at Facebook: http://www.facebook.com/MySQLatFacebook.

[10] Note on creating Cassandra: https://www.facebook.com/note.php?note_id=24413138919.

[11] Jeremy Zawodny, "Lessons Learned from Migrating 2+ Billion Documents at Craigslist": http://www.10gen.com/presentation/mongosf2011/craigslist.