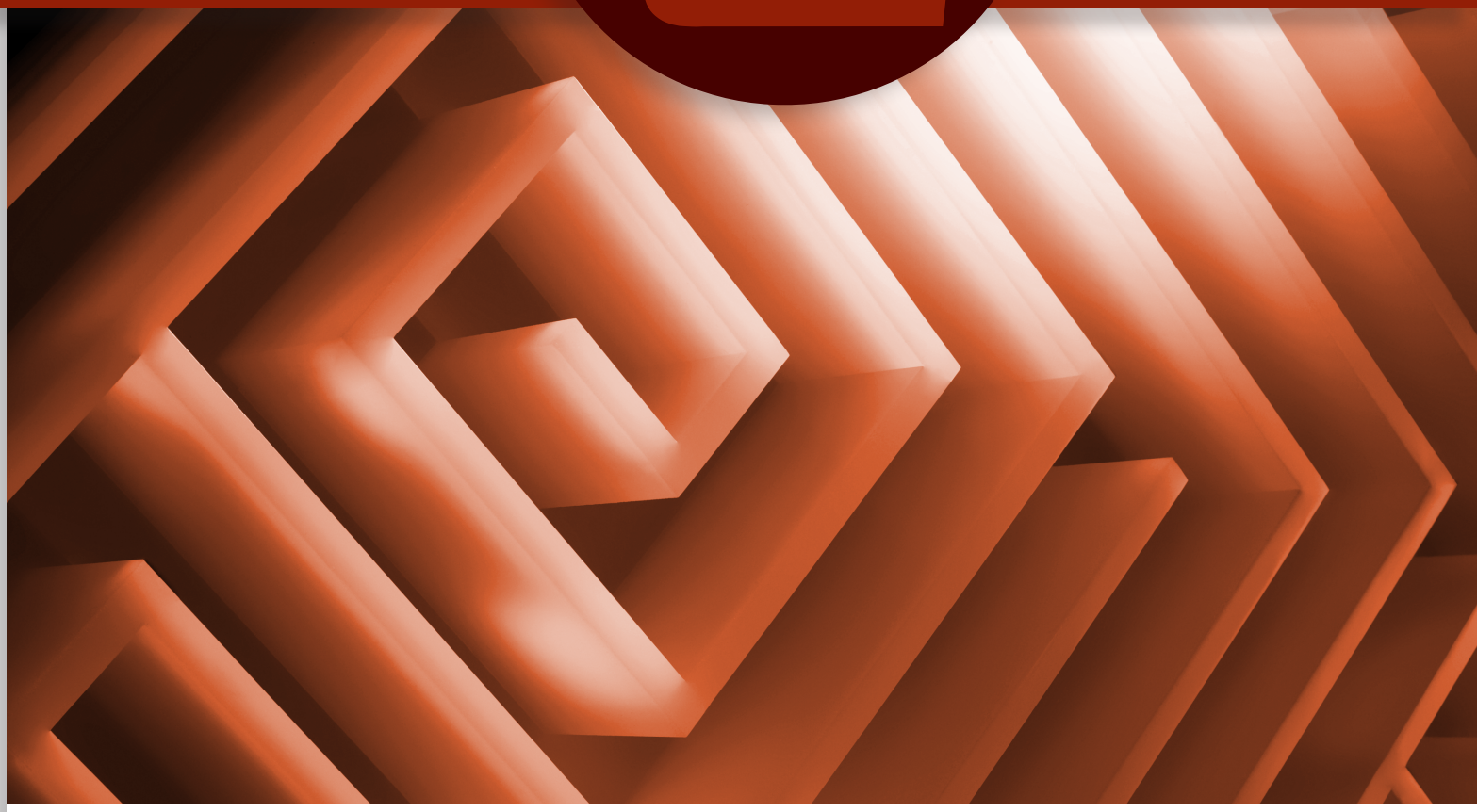
InfoWorld DeepDive



Which freaking database should I use?

IN THE ERA OF BIG DATA, GOOD OLD RDBMS IS NO LONGER THE RIGHT TOOL FOR MANY DATABASE JOBS. HERE'S A QUICK GUIDE TO CHOOSING AMONG NOSQL ALTERNATIVES

Copyright © 2013 InfoWorld Media Group. All rights reserved.

Deep Dive

STRATEGY

Which freaking database should I use?

In the era of big data, good old RDBMS is no longer the right tool for many database jobs. Here's a quick guide to choosing among NoSQL alternatives

BY ANDREW C. OLIVER

Right off the bat, much of your audience is in unfamiliar territory.

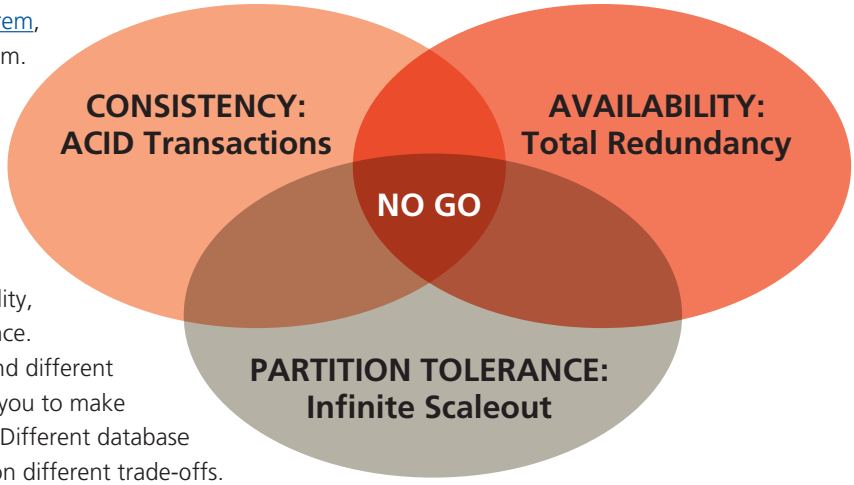
I've been in Chicago for the last few weeks setting up our first satellite office for my company. While Silicon Valley may be the home of big data vendors, Chicago is the home of the big data users and practitioners. So many people here "get it" that you could go to a packed meetup or big data event nearly every day of the week.

Big data events almost inevitably offer an introduction to NoSQL and why you can't just keep everything in an RDBMS anymore. Right off the bat, much of your audience is in unfamiliar territory. There are several types of NoSQL databases and rational reasons to use them in different situations for different datasets. It's much more complicated than tech industry marketing nonsense like "NoSQL = scale."

Part of the reason there are so many different types of NoSQL databases lies in the CAP theorem, aka Brewer's Theorem. The CAP theorem states you can provide only two out of the following three characteristics: consistency, availability, and partition tolerance. Different datasets and different runtime rules cause you to make different trade-offs. Different database technologies focus on different trade-offs. The complexity of the data and the scalability of the system also come into play.

Another reason for this divergence can be found in basic computer science or even more basic mathematics. Some datasets can be mapped easily to key-value pairs; in essence, flattening the data doesn't make it any less meaningful, and no reconstruction of its relationships is necessary. On the other hand, there are datasets where the relationship to other items of data is as important as the items of data themselves.

Relational databases are based on relational algebra, which is more or less an outgrowth of set theory. Relationships based on set theory are effective for many datasets, but where parent-child or distance of relationships are required, set theory isn't very effective. You may need graph theory to efficiently design a data solution.



CAP THEOREM

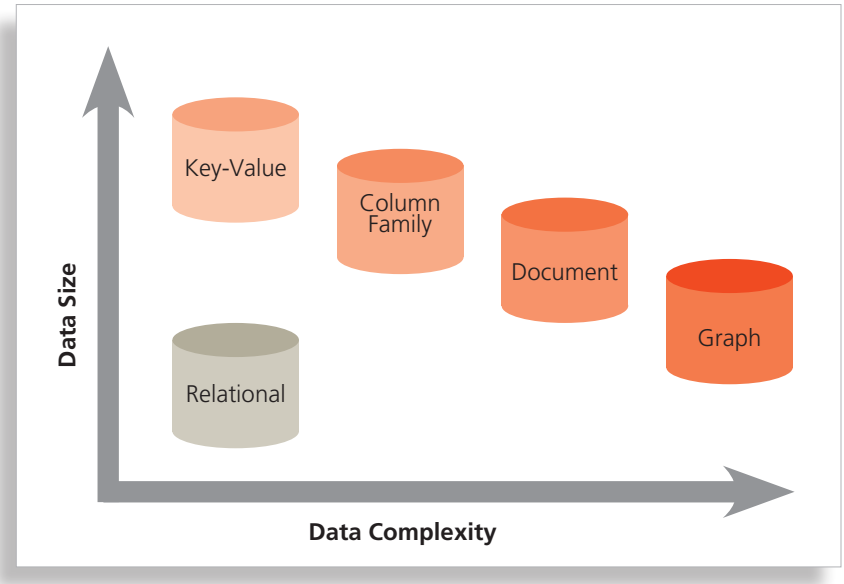
Deep Dive

Many developers think document databases are the Holy Grail since they fit neatly with object-oriented programming.

In other words, relational databases are overkill for data that can be effectively used as key-value pairs and underkill for data that needs more context. Overkill costs you scalability; underkill costs you performance.

Key-value pair databases

Key-value pair databases include the current 1.8 edition of Couchbase and Apache Cassandra. These are highly scalable, but offer no assistance to developers with complex datasets. If you essentially need a disk-backed, distributed hash table and can look everything up by identity, these will scale well and be lightning fast. However, if you find that you're looking up a key to get to another key to get to another key to get to your value, you probably have a more complicated case.



KEY-VALUE DATABASE

There are a number of different permutations of key-value pair databases. These are basically different trade-offs on the CAP theorem and different configurations of storage and memory use. Ultimately, you have some form of what is basically a hash table.

This is fine for flat parts lists so long as they don't composite. This is also fine for stock quotes, "right now," or other types of lists where that key has meaning and is the primary way you're going to look up the value. Usually these are combined with an index, and there is a way to query against the values or generate a list of keys, but if you need a lot of that, you probably should look elsewhere.

Column family/big table databases

Most key-value stores (including Cassandra) offer some form of grouping for columns and can be considered "column family" or "big table" as well. Some databases such as HBase were designed as column family stores from the beginning. This is a more advanced form of a key-value pair database. Essentially, the keys and values become composite. Think of this as a hash map crossed with a multidimensional array. Essentially each column contains a row of data.

According to Robin Schumacher, the vice president of products for DataStax, which sells a certified version of Cassandra, "A popular use case for Cassandra is time series data, which can come from devices, sensors, websites (e.g., Web logs), financial tick data, etc. The data typically comes in at a high rate of speed, can come from multiple locations at once, adds up quickly, and requires fast write capabilities as well as high-performance reads over time slices."

You can use also use MapReduce on these, so they can be good analytical stores for semi-structured data. These are highly scalable, but not usually transactional. If the relationships between the data are as important as the data itself (such as distance or path calculations), then don't use a column family/big table database.

Document databases

Many developers think document databases are the Holy Grail since they fit neatly with object-oriented programming. With high-flying vendors like 10gen (MongoDB), Couchbase, and Apache's CouchDB, this is where most of the vendor buzz is generated.

Frank Weigel from Couchbase pointed out to me that the company is moving from a key-value pair database in version 1.8 to a document database in 2.0. According to him, the "document database is a natural progression. From clustering to accessing data, document databases and key-value stores are exactly the same, except in a document database, the database understands the documents in the datastore." In other words, the values are JSON, and the elements inside the JSON document can be indexed for

Deep Dive

better querying and search.

The sweet spot for these is where you're probably already generating JSON documents. As Max Schireson, president of 10gen told me, you should consider a document database if your "data is too complex to model in a relational database. For example, a complex derivative security might be hard to store in a traditional format. Electronic health records provide another good example. If you were considering using an XML store, that's a strong sign to consider MongoDB and its use of JSON/BSON."

This is probably your operational store -- where data being collected from users, systems, social networks, or whatever is being collected. This is not likely where you are reporting from, though databases such as MongoDB often have some form of MapReduce available. While at least in MongoDB, you can query on anything, you will not generally achieve acceptable performance without an index.

Just as we shouldn't try to solve all of our problems with an RDBMS, we shouldn't try to solve all of our math problems with set theory.

Graph databases

Graph databases are really less about the volume of data or availability and more about how your data is related and what calculations you're attempting to perform. As Philip Rathle, senior director of product engineering at Neo Technologies (makers of Neo4j), told me, graph databases are especially useful when "the data set is fundamentally interconnected and non-tabular. The primary data access pattern is transactional, i.e., OLTP/system of record vs. batch... bearing in mind that graph databases allow relatedness operations to occur transactionally that, in an RDBMS world, would need to take place in batch."

This flies in the face of most NoSQL marketing: A specific reason for a graph database is that you need a transaction that is more correct for your data structure than what is offered by a relational database.

Common uses for graph databases include geospatial problems, recommendation engines, network/cloud analysis, and bioinformatics -- basically, anywhere that the relationship between the data is just as important as the data itself. This is also an important technology in various financial analysis functions. If you want to find out how vulnerable a company is to a bit of "bad news" for another company, the directness of the relationship can be a critical calculation. Querying this in several SQL statements takes a lot of code and won't be fast, but a graph database excels at this task.

You really don't need a graph database if your data is simple or tabular. A graph database is also a poor fit if you're doing OLAP or length analysis. Typically, graph databases are paired with an index to allow for better search and lookup, but the graph part has to be traversed; for that, you need a fix on some initial node.

Sorting it all out

Graph databases provide a great example of why it's so hard to name these new database types. "NewDB" is my preferred name -- except that, oops, some are as old as or older than the RDBMS. "NoSQL" isn't a great name because some of these support SQL and SQL is really orthogonal to the capabilities of these systems.

Finally, "big data" isn't exactly right because you don't need large data sets to take advantage of databases that fit your data more naturally than relational databases. "Nonrelational" doesn't quite apply, either, because graph databases are very relational; they just track different forms of relationships than traditional RDBMSes.

In truth, these are the rest of the databases that solve the rest of our problems. The marketing noise of past decades combined with hardware and bandwidth limitations, as well as lower expectations in terms of latency and volume, prevented some of the older kinds of databases from reaching as wide notoriety as RDBMSes.

Just as we shouldn't try to solve all of our problems with an RDBMS, we shouldn't try to solve all of our math problems with set theory. Today's data problems are getting complicated: The scalability, performance (low latency), and volume needs are greater. In order to solve these problems, we're going to have to use more than one database technology.

Andrew C. Oliver *is a professional cat herder who moonlights as a software consultant. He is most known for founding the POI project, which is now hosted at Apache. He also was one of the early developers at JBoss.*