Published on *InfoWorld* (http://www.infoworld.com)

Slacker databases break all the old rules

By Peter Wayner
Created *2009-03-24 03:00AM*

> So you've got some data to store. In the past, the answer was simple: Hook up an official database, pour the data into it, and let the machine sort everything out for you while you spend your time writing big checks to the database manufacturer. Now things aren't so cut and dry. A fresh round of exciting new tools is tacking the two letters "db" onto a pile of code that breaks with the traditional relational model. Old database administrators call them "toys" and hint at terrible dangers to come from the follies of these young whippersnappers. The whippersnappers just tune out the warnings because the new tools are good enough and fast enough for what they need.

The non-relational upstarts are grabbing attention because they're willfully ignoring many of the rules that codify the hard lessons learned by the old database masters. The problem is that these belts-and-suspenders strictures often make it hard to create really, really big databases that suck up all of the cycles of a room full of machines. Because all Web application designers dream of building a startup that needs a really big room filled with machines to hold all of the data of all of the users, the rules need to be bent or even broken.

[For a brief look at more alternative databases, see Open source and SaaS offerings rethink the database [1]. Catch InfoWorld's cloud computing reviews and analysis: Cloud versus cloud: Amazon, Google, AppNexus, and GoGrid [2] | Inside Amazon Web Services [3] | App builders in the sky [4] | Windows Azure Services Platform gives wings to .Net [5] | What cloud computing really means [6].]

The first thing to go is the venerable old JOIN. College students used to dutifully work through exercises that taught them how to normalize the data by breaking the tables up into as many parts as practical. Disk space was expensive then, and a good normalization expert could really pack in the data. The problem is that JOINs are really, really slow when the data is spread out over several machines. Now that disk space is so cheap and many of the data models don't benefit as much from normalization, JOINs are easy to leave behind.

The next trick is to start using phrases like "eventual consistency." Amazon's documentation for SimpleDB includes this inexact promise: "Consistency is usually reached within seconds, but a high system load or network partition might increase this time." The new twerps really get those codgers steamed when they talk about how all of the computers in the cluster will get around to replicating the data and giving consistent answers when the machines are good and ready. For the kids, consistency is akin to cod liver oil or making your bed in the morning.

This distinction between immediate and eventual consistency is deeply philosophical and depends on how important the data happens to be. The old guard who start reaching for their heart medication at the news of these new databases are usually bank programmers who want to make sure that the accounts balance at the end of the day. After all, the bank's brilliant leaders can't turn around and "invest" the cash in subprime mortgages if there's one penny missing after a failed database transaction.

At least they're not hauling the DBAs before Congress to explain where the cash went.

But many modern Web sites will sail on without a hiccup if some transaction fails. I see glitches on Facebook regularly. The world won't end if some snarky, anonymous comment on Slashdot disappears. None of these sites cares if the accounting is as good as a bank's, and they don't really need all of the power of a traditional database. (Some wags suggest that banks put the money from an Oracle license into a fund to compensate the people who actually lose money on a failed transaction from one of these newfangled data stores.)

To get an understanding of this expanding tier of non-traditional databases, I took a few out for a ride and built up some test applications with them. The field was surprisingly diverse despite the fact that the offerings are so stripped down that they really don't have more than three major commands: Insert, Update, and Delete. Some offer clustering. Some are available only as a service. Some have grand pretensions to take over the entire server stack. Some play better with AJAX tools than others. None of them is right for everyone, and all of them are completely wrong for the bankers out there. (See the sidebar, "Open source and SaaS offerings rethink the database [1].")

I also excluded a few interesting tools because of space or just because they were slightly different. Sun, for instance, is now bundling a version of a relational database called Derby with its Java VM. Oracle has its own embedded tool once known as Sleepycat's Berkeley DB but now called the Oracle Embedded Database. Some programmers are even creating very low-rent libraries that write the objects directly to the disk. One project, Prevayler [7], brags that all of the code from one version could fit legibly on a T-shirt. These products are also stretching the meaning of the two letters "db," but they didn't fit in this comparison.

Amazon SimpleDB

SimpleDB is one of the most advanced and most cloud-like components of Amazon's great push [8] into cloud computing services. Once you sign up and get your secret password, you can ship off some Web service XML filled with pairs of keys and values to SimpleDB and it will store the data for you -- well, as long as you keep paying the bills shown on the meter. You don't need to think about installing anything or backing it up. Amazon hides all of that work for you behind its Web service wall.

SimpleDBcomes with two levels of hierarchy on top of the piles of data pairs. The top level is the "domain" and the second level is the "item." After you choose the domain and item names, you pour in the pairs. SimpleDB's comparatively feature-rich API includes the ability to sort the data and even count the number of items that match the query. You can even write queries that exclude values that don't start with a certain string. This may not sound like much to someone who uses SQL Server or Oracle, but some of these low-rent databases can't even sort the data in the result set.

SimpleDBis meant to be used with Amazon's Simple Storage Service (S3), because each of the values in the pairs is limited to 1,024 bytes. That's enough for many strings, but it's not enough for many content engines. So you store a pointer to the data in S3. There are a few libraries like an extension of the Java Persistence Architecture that straddle the two clouds and handle this pointer juggling for you.

There are other limitations that can lead you to start doing JOIN-like things with multiple calls. Each query can only run 5 seconds. The answer can only hold 250 items. Each item can have only 250 pairs. Some people half joke about concatenating multiple values with keys like "description1," "description2," and "description3." There are many simple work-arounds for the limitations, but they start to make you wonder whether SimpleDB is supposed to make your life easier or harder.

Amazon is beginning to rewrite the APIs to push for more and better authentication. Come September 2009, calls to the SimpleDB (and a few other services) will run through SSL, providing both security and authentication. Amazon is also enhancing the signature mechanism to use more sophisticated

hashing algorithms that pack together more of the request. This is just one of the ways that Amazon is slowly rolling out small improvements.

The company is also creating more libraries that make it simpler to use the service. There are dozens of packages that work with all of the major languages and some of the minor ones. The documentation is extensive. It's usually possible to start up and begin storing your data in little time.

The price is now easier to handle. There's a "free tier" of service that lets you burn up to 25 hours of computation time per month -- enough, Amazon estimates, to run a basic logging tool that processes less than 2 million requests a month. Plus, Amazon recently slashed the price for storage from $1.50 to 25 cents per gigabyte. The company appears committed to keeping the charges transparent so users will have the right incentives to structure their consumption.

Amazon has one of the more advanced terms of service. There are plenty of clauses that work through some of the problems you might encounter, and several caught my untrained eye. For instance, Amazon claims, "We may delete, without liability of any kind, any of your Amazon SimpleDB Content that has not been accessed in the previous 6 months." This may be perfectly acceptable for the people who are taking the system out for a spin with test data and not paying for it, but the phrasing suggests a bit of the omnipotence that Amazon probably feels it needs to keep its datacenter running.

There are other squishy issues. For instance, the terms of service include a long list of forbidden data, such as promoting illegal activities and discriminating on the basis of "race, sex, religion, nationality, disability, sexual orientation, or age." Imagine you're running a Web site for some church campaigning against gay marriage. That sounds like it might be dinged for discriminating against sexual orientation. But let's say you're campaigning for gay marriage by protesting these churches. Are you discriminating on the basis of religion?

I feel sorry for the lawyers who are going to parse the complaints, but at least they can rest easy knowing they can pretty much delete your data "for any reason or for no reason." Whew. If you're just using the free service, Amazon doesn't have to give you any notice, but it promises a 60-day notice if you're a paying customer. You can get your data back -- if you pay the storage charges that keep accruing.

Google App Engine

Google App Engine [9] isn't a database per se. It's a cloud for distributing Python applications, and it comes with its own database hidden away inside. It's not really possible to access the database without going through the application layer first. But it's not hard to wrap up a database call and format the data for the request, so it might be proper to think of App Engine as a database with a layer of embedded procedures that are written in Python.

This extra layer of customizability is often quite useful. Many of the complaints about the other toy databases revolve around how a missing feature makes it impossible to find the right data. If you want to add a bit more functionality to the database here, you can whip up many of the features locally in Python. If you want a JOIN, you can synthesize one in Python and probably customize the memory cache at the same time. This is especially useful for Web applications that let users store their data in the service. If you need to add security to restrict each user to the right data, you can code that in Python too.

The App Engine data store is much more structured than Amazon's SimpleDB, and it gets much of this structure from Python's object model. You don't store key-value pairs, but Python objects, and those are defined with something that's pretty similar to an SQL schema. You can set the type of each column, make some of them required, and then ask for indexing across the columns that you'll need. The transaction mechanism is also deeply entwined with Python because each transaction is really just a

Python function. This is a bit of a simplistic statement because there is a list of restrictions on what can happen inside this function (including rules such as each item can be updated only once). The good news is that the Google team is building special transaction methods that abstract away some of the common behavior (such as "Create or Update" a row).

Searching is deliberately set up to be SQL-like; in fact, Google offers its own SQL-like language, GQL, that's parsed into queries. There's also a Python-based set of methods that can be chained together to handle the data selection and querying. You don't need to waste the cycles parsing the query.

It's worth pointing out that the Python stack includes a number of features that aren't found in the best of databases. There's a library for manipulating image files by cropping and even a Google-esque "I feel lucky" function that will fix up the picture with some magic formula. If you want to e-mail someone, you can. You can also store data as Google documents, spreadsheets, and calendar items. It may seem like just a database at first, but it's easy to get sucked into the Google stack.

Until a few weeks ago, App Engine was beta and using it was free. It's still free as long as you stay within some basic quotas. After that, Google is charging with a mechanism that's pretty similar to Amazon's. The price for storage is cheaper (12 cents per gigabyte per month), but the charge for bandwidth is about the same (10 cents per gigabyte coming in.)

Google's terms of service carry a different set of responsibilities than Amazon's. You're required to formulate a privacy policy and guard the data of your users. If your users violate copyright rules, you must respond to DMCA (Digital Millennium Copyright Act) takedown notices or Google will do it for you. Google retains the right to delete any content at any time for any reason: "You agree that Google has no responsibility or liability for the deletion or failure to store any Content and other communications maintained or transmitted through use of the Service."

These terms have become more focused over the years. Google now promises to give you 90 days to get your data off of the servers if it decides to cancel your account -- something it can do "for any reason." Many of the changes I've noticed over that time seem to be focused on DMCA issues, which tie everyone in knots up and down the chain.

It's an interesting question what would happen if you decide to leave Google or Google asks you to leave. Google distributes a nice development tool that makes it easy for you to test your applications on your local machine. There's no technical reason why you couldn't host your service on your own server with these tools, except you would lose some of the cloud-like features. The data store included for testing wouldn't replicate itself automatically, but it seems to do everything else on my local machine. As always, there are some legal questions because "license is for the sole purpose of enabling you to use and enjoy the benefit of the Service."

Apache CouchDB

There's no reason why you need to work with a cloud to enjoy these new services. CouchDB is one of the many open source projects that build a simple database for storing key-value pairs. The project, written in Erlang, is supported under the aegis of the Apache Software Foundation. You can install it on any server by downloading the source files and compiling them. Then there are no more charges except paying for the server.

The CouchDB is similar to Amazon's tool, but it has some crucial differences. You still store key-value pairs as rows, but these pairs can be any of the standard JSON (JavaScript Object Notation) data types like Booleans and numbers. These values aren't limited to being 1,024-byte strings, something that makes it possible to store long values and even things like images. All of the requests and responses are formatted as JavaScript. There are no XML-based Web services, just JSON.

The biggest differences come when you're writing queries. CouchDB lets you write separate map

functions and reduce functions using JavaScript. A simple query might just be a map function with a single "if" clause that tests to see whether the data is greater or less than some number. The reduce functions are only required if you're trying to compute some function across all of the data found by the map function. Counting the number of rows that are found is easy to do, and it's possible to carry off arbitrarily cool things as well, because the map function is limited only by what you can specify in JavaScript. This can be very powerful, although try as I might, I couldn't figure out any non-academic uses beyond counting the number of matches. The documentation includes one impressive reduction function that computes statistics, but I don't know if CouchDB is really the right tool for that kind of thing. If you need complex statistics, it may be better to stick with a traditional database with a traditional package for building reports with statistics.

There are still some limitations to this project. While the front page of the project calls it "a distributed, fault-tolerant, and schema-free document-oriented database," you won't get the distribution and fault-tolerance without some manual intervention. The nice AJAX interface to CouchDB includes a form that you can fill out to replicate the database. It's not automatic yet.

There are plans for an access control and security model, but these are not well-documented or even apparent in the APIs. They are designed to use pure JavaScript instead of SQL or some other language, which is a nice idea. You don't give or take away permissions to read documents; you just write a JavaScript function that returns true or false.

This approach isn't as limiting as it might seem. As I was working with these databases, I soon began to see how anyone could layer on a security model at the client with the judicious use of some encryption. Empowering the client reduces the need for much security work at the server, something I wrote about in *Translucent Databases [10]*.

Observations like this are driving some of the more extreme users to push toward using CouchDB as the entire server stack. J. Chris Anderson, one of the committers on the project, wrote a fascinating piece [11] arguing that CouchDB is all you need for an application server. The business logic for displaying and interacting with the data is written in JavaScript and downloaded from CouchDB as just another packet of JSON data.

In Anderson's eyes, there's no big reason to use Ruby, Python, Java, or PHP on the server when it can all be packaged in JavaScript. This may be a bit extreme because there will always be some business cases when the client machine can't be trusted to do the right thing, but they may be fewer than we know. Lightweight tools like CouchDB are encouraging people to rethink how much code we really need to get the job done.

Persevere

At first glance, the Persevere database looks like most of the others. You push pairs of keys and values into it, and it stores them away. But that's just the beginning. Persevere provides a well-established hierarchy for objects that makes it possible to add much more structure to your database, giving it much of the form that we traditionally associate with the last generation of databases. Persevere is more of a back-end storage facility for JavaScript objects created by AJAX toolkits like Dojo, a detail that makes sense given that some of the principal developers work for SitePen, a consulting group with a core group of Dojo devotees.

Persevere is not like some of the other databases in this space that seem proud of the fact that they're "schema-free." It lets you add as much schema as you want to bring structure to your pairs. Instead of calling the top level of the hierarchy a domain (SimpleDB) or a document (CouchDB), Persevere calls them objects and even lets you create subclasses of the objects. If you want to enforce rules, you can insist that certain fields be filled with certain types, but there's no recommendation. The schema rules are optional.

The roots in the Dojo team are apparent because Dojo comes with a class, JsonRestStore, that can connect with Persevere and a number of other databases. including CouchDB. (Dojo 1.2 will also connect with Amazon's S3 but not SimpleDB and Google's Feed and Search APIs but not App Engine, at least out of the box.) The "Store" is sophisticated and has some surprising facilities. When I was playing with it initially, I hadn't given the clients the permissions to store data directly. The tool stored the data locally as if I were offline and had no connection with the database. When I granted the correct permissions later, the changes streamed through as if I had reconnected.

Persevere provides a great deal of connectivity through this tight connection with Dojo. You can create grid and tree widgets, then link them directly to the JsonRestStore; the widgets will let you edit the data. Voila! You've got remote access to a database in about 20 lines of JavaScript.

I encountered a number of small glitches that were probably due more to my lack of experience than to underlying bugs. Some things just started working correctly when I figured out exactly what to do. It's not so much Persevere itself you need to master, but the AJAX frameworks you're using in front of it. The documentation from Dojo is better than most AJAX frameworks, but it will take some time for Dojo to catch up with the underlying complexity that's hidden by the smooth surface of Persevere.

Cloud or cluster

After playing with these databases, I can understand why some people will keep using the word "toys" to describe them. They do very little, and their newness limits your options. There were a number of times when I realized that a fairly standard feature from the SQL world would make life simpler. Many of the standard SQL-based tools, like the reporting engines, can't connect with these oddities. There are a great many things that can be done with MySQL or Oracle out of the box.

But that doesn't mean that I'm not thinking of using them for one of my upcoming projects. They are solid data stores and so tightly integrated with AJAX that they make development very easy. Most Web sites don't need all of the functions of a MySQL or Oracle, and JOIN-free schemas are still pretty useful for many common data structures, including one-to-many and one-to-one relationships. Even many-to-one relationships are feasible until something needs to be changed. Given that database administrators are often denormalizing the tables to speed them up, you might say that these non-relational tools just save them a step.

One of the trickier questions is whether to use a cloud or build your own cluster of machines. Both Google and Amazon offer multimachine promises that CouchDB and Persevere can't match. You've got to push the buttons yourself with CouchDB. The Persevere team talks about scaling in the future. But it can be hard to guess how good the promises of Amazon and Google might be. What happens if Amazon or Google loses a disk? What if they lose a rack? They still don't make explicit promises and their terms of service explicitly disclaim any real responsibility.

Amazon's terms, for instance, repeat this sentiment a number of times: "We are not responsible for any unauthorized access to, alteration of, or the deletion, destruction, damage, loss or failure to store any of, Your Content (as defined in Section 10.2), your Applications, or other data which you submit or use in connection with your account or the Services."

I can't say I blame Amazon or Google because who knows who is ultimately responsible for a lost transaction? It could be any programmer in the stack, and it would be practically impossible to decide who trashed something. But it would be nice to have more information. Is the data in a SimpleDB stored in a RAID disk? Is a copy kept in another geographic area unlikely to be hit by the same earthquake, hurricane, or wildfire? The online backup community is starting to offer these kinds of details, but the clouds have not been so forthcoming.

All of these considerations make it clear to me that these are still toy databases that are best suited for

applications that can survive a total loss of data. They're noble experiments that do a good job of making the limitations of scale apparent to programmers by forcing them to work with a data model that does a better job of matching the hardware. They are fun, fast, and so reasonable in price that you can forget about writing big checks and concentrate on figuring out how to work around the lack of JOINs.

- Cloud Computing
- Data Management
- Amazon Web Services
- Amazon.com
- Apache
- Google
- Applications
- Data management
- Database management systems
- Infrastructure services

Links:
[1] https://www.infoworld.com/article/09/03/24/12TC-databases-alts_1.html?source=fssr
[2] https://www.infoworld.com/article/08/07/21/30TC-cloud-reviews_1.html?source=fssr
[3] https://www.infoworld.com/article/08/08/13/33TC-amazon-web-services_1.html?source=fssr
[4] https://www.infoworld.com/article/08/09/22/39TC-dev-clouds_1.html?source=fssr
[5] https://www.infoworld.com/article/08/12/15/51TC-azure-preview_1.html?source=fssr
[6] https://www.infoworld.com/article/08/04/07/15FE-cloud-computing-reality_1.html?source=fssr
[7] http://www.prevayler.org/
[8] https://www.infoworld.com/article/08/08/13/33TC-amazon-web-services_1.html
[9] https://www.infoworld.com/article/08/05/12/20TC-google-app-engine_1.html
[10] http://www.wayner.org/node/52
[11] http://jchris.mfdz.com/code/2008/10/standalone_applications_with_co

-