Not the same thing as non-SQL DBMS

NoSQL Databases

Curt Monash

For cutting-edge applications – often but not only internet-centric – NoSQL technology can make sense today. In other use cases, its drawbacks are likely to outweigh its advantages.

Many of the world's largest and fastest-growing databases belong to website owners (e.g. Google, Facebook, Twitter) or other internet companies (e.g., Zynga, maker of the games Farmville and Mafia Wars). Those companies routinely reject Oracle and similar high-end database management systems (DBMS). So do smaller companies in similar industries, who can only dream of running databases that big. Their reasons commonly include:

- They don't want to pay Oracle's license fees, and indeed have a strong bias toward open source software;
- They don't need Oracle's high-end features;
- In fact, they don't need most SQL functionality;
- They aren't that excited about writing SQL anyway (or generating via, say, an object-relational mapping layer);
- License fees aside, they believe Oracle's architecture and features get in the way of scalability, and the same goes for any other SQL DBMS whose features are used more or less as intended.

In short, some of the largest and most innovative applications in the world are being built by people who don't see much value in DBMS in general, nor in high-end SQL DBMS in particular.

When one first hears this, it can sound like madness. Upon investigation, however, it turns out to make a certain sense. Databases are being built for single applications, optimizing performance, networking considerations, and/or software license fees at the expense of application extensibility. In such scenarios:

- DBMS lose their traditional roles as powerful DML (Data Manipulation Language) interpreters; rather, application programmers have to code every bit of data manipulation smarts themselves;
- Also falling by the wayside are most DBMS performance optimizations for different classes of queries;
- All that data management subsystems are used for is to read and write small amounts of data in very rapid manner, and then to back up, replicate, and otherwise manage the already-stored data.

Consider, for example, the following use cases:

- A large website exists primarily to accept and serve back photographs (or songs, etc.), small snippets of text, and the contents of simple database records. Almost everything is keyed on user IDs. Throughput is massive. (Social networking, photo sharing);
- Most of what happens in the database is that counters keep getting incremented, and not for anything where real money changes hands. So write locks are terrible bottlenecks, and transaction integrity – while ideally a nice-to-have – is not essential. (Social gaming, article sharing);
- A central server wants to coordinate application versions and some amount of data across a broad number of occasionally-connected instances, perhaps for a broad variety of applications. (Mobile computing, social gaming, etc.).

E-commerce aside, those use cases cover a large fraction of what's going on in internet innovation. And while they all can be satisfied with traditional relational DBMS*, they all fit the DBMS-unfriendly template that:

- Joins are inessential or secondary;
- Transaction semantics are inessential or secondary, and;
- Two-phase commit is an overly restrictive way of replicating data.

**After all, relational DBMS can be used to do pretty much anything.*

And so the challenges to traditional database management ideas are piling up. For years, world-class applications have been built using MySQL, hardly the most robust of DBMS.

What's worse, these applications haven't used more than a tiny fraction of MySQL's capabilities. Indeed, the biggest systems have relied on "sharding" MySQL – putting different rows of a MySQL table onto different machines, and *relying on application logic* to know which machine to access. And if those applications use any joins at all, they're only ones that will never cause data to move to from one node to another as part of the join resolution. Perhaps worse yet, the same applications often also rely on an in-memory key-value store called memcached. (More on the "key-value" data model below).
And since all that isn't already far enough the relational DBMS world for some developers' tastes, it's beginning to be superseded by a popular new movement called "NoSQL", which aspires to get SQL-based DBMS out of the stack entirely.

Before going further, let's clear up one point immediately: "NoSQL database" is *not* the same thing as "non-SQL DBMS". True, the term "NoSQL" technically stands for "Not Only SQL" – but taking that to an extreme (which some marketeers do) is misleading. After all, non-SQL DBMS have flourished literally since the invention of database management systems, over 40 years ago. Some leading pre-relational mainframe DBMS – notably IMS and Adabas – survive to this day. Small enterprise databases, built on Microsoft Access or Apple FileMaker, may have nothing to do with SQL. While medium-sized enterprises usually run on relational DBMS, Intersystems Caché and various "multi-value" systems also have had considerable success. Even large enterprises often use special purpose systems, such as multidimensional "OLAP" servers, or MarkLogic (for XML data). And none of those has much to do with the NoSQL market.*

**One exception: MarkLogic is seemingly making a bid for MarkLogic Server to be classified alongside NoSQL document/object stores such as MongoDB and CouchDB. But why exactly MarkLogic aspires to be compared to startup companies with 15 or so employees each is not entirely clear.*

Rather, NoSQL DBMS start from three design premises:
- Transaction semantics are unimportant, and locking is downright annoying;
- Joins are also unimportant, especially joins of any complexity;
- There are some benefits to having a DBMS even so.

NoSQL DBMS further incorporate one or more of three assumptions:
- The database will be big enough that it should be scaled across multiple servers;
- The application should run well if the database is replicated across multiple geographically distributed data centers, even if the connection between them is temporarily lost;
- The database should run well if the database is replicated across a host server and a bunch of occasionally-connected mobile devices.

In addition, NoSQL advocates commonly favor the idea that a database should have no fixed schema, other than whatever emerges as a byproduct of the application-writing process.

"Not Only SQL" is not the only terminological problem around NoSQL. Much of the innovation in the NoSQL area revolves around the area of "consistency", but that word does not mean the same thing as it does in ACID (Atomicity, Consistency, Isolation, Durability); if anything, consistency is closer to "durability", in that it refers to the desirable property of getting a correct answer back from the DBMS even in a condition of (partial) failure. In essence, there are three reasonable approaches to consistency in a replicated data scenario:*

DBMS based on graphical data models are also sometimes suggested to be part of NoSQL

- Traditional/near-perfect consistency, in which processing stops until the system is assured that an update has propagated to all replicas. (This is typically enforced via a two-phase commit protocol.) The downside to this model, of course, is that a single node failure can bring at least part of the system to a halt;
- Eventual consistency, in which inaccurate reads are permissible just so long as the data is synchronized "eventually". With eventual consistency, the network is rarely a bottleneck at all – but data accuracy may be less than ideal;
- Read-your-writes (RYW) consistency, in which data from any single write is guaranteed to be read accurately, even in the face of a small number of network outages or node failures. However, a sequence of errors can conceivably produce inaccurate reads in ways that perfect consistency would forbid.

Some systems allow tuning – e.g. configuration – as to which consistency model is supported; others are more locked in to a particular choice at this time.

** The theory behind all this is Eric Brewer's CAP Theorem, for Consistency, Availability, and Partition Tolerance, the point being that you can't have all three of those in the same system. However – you guessed it – "Availability" and "Partition" are used in unconventional word-senses too.*

If not SQL, then what? A number of possibilities have been tried, with the four main groups being:
- Simple key-value store;
- Quasi-tabular;
- Fully SQL/tabular;
- Document/object.

DBMS based on graphical data models are also sometimes suggested to be part of NoSQL, as are the file systems that underlie many MapReduce implementations. But as a general rule, those data models are most effective for analytic use cases somewhat apart from the NoSQL mainstream.

A *key-value store* is like a relational DBMS in which there only can be a single 3-column entity-attribute-value table, and in which you can't do self-joins. (In that analogy, the key part of the key-value pair may be thought of as an entity-attribute composite.) Thus, any conception of "object" has to live in the application logic; the data management software is little more than an intelligent storage system. Key-value stores may have modest performance advantages over the more efficient implementations of other models, but otherwise there's little advantage to using a key-value store. (One exception: You might want to use a persistent data store, such as Northscale's beta product Membase, as the target for porting an existing memcached-based application.) Most key-value store products, Membase included, have or soon are planned to have alternative interfaces with at least somewhat richer data models.

More powerful are the *quasi-tabular* systems such as Cassandra, HBase, or (the original one) Google BigTable. In these, you can store what amount to rows, without worrying about whether each row has values for the same set of columns. Thus, a quasi-tabular database is like a relational database – albeit one with lots of NULL values – but with its schema controlled by the application program rather than a DBA.

The most prominent NoSQL implementations at big name web companies are of Cassandra or HBase, with Facebook, Twitter, Digg, StumbleUpon, and many others having joined the bandwagon. Both Cassandra and HBase are open source projects; neither is deemed to yet have reached its 1.0 release. But they have significant production installations even so. The go-to vendors for Cassandra and HBase are Riptano and Hadoop specialist Cloudera respectively. (HBase is closely tied to the Hadoop MapReduce project.)
There's also a new generation of *SQL-based* systems that seem to overcome some of the NoSQL community's objections to conventional SQL DBMS, including Schooner, Clustrix, dbShards, VoltDB, and Akiban. Often, they come in key-value flavors as well, with a performance advantage of less than 2:1 versus the SQL implementations. (Schooner Information Systems, an appliance maker, offers at least benchmarks for a broad variety of NoSQL systems, but probably gets its sales largely from memcached and/or MySQL implementations.) Schooner somewhat aside, most of these vendors are still in early days in terms of getting actual customers.

Finally, there are the NoSQL *document/object stores*, most notably CouchDB (which boasts a Lotus Notes-like replication model) and MongoDB (which has a standard NoSQL laundry list of replication options). These store JSON (JavaScript Object Notation) objects – i.e., collections of name-value pairs – directly. CouchDB and MongoDB also both have ways of indexing, querying, and/or updating individual "fields" within the document schema.
CouchDB and MongoDB both have considerable numbers of known users, generally for applications that don't seem to demand large data volumes or high throughput. The go-to vendor for CouchDB is Couchio or, if you have a larger database, Cloudant. The company behind MongoDB is 10gen.

So should you adopt NoSQL technology? Key considerations include:

- *Immaturity*. The very term "NoSQL" has only been around since 2009. Most NoSQL "products" are open source projects backed by a company of fewer than 20 employees;
- *Open source*. Many NoSQL adopters are constrained, by money or ideology, to avoid closed-source products. Conversely, it is difficult to deal with NoSQL products' immaturity unless you're comfortable with the rough-and-tumble of open source software development;
- *Internet orientation*. A large fraction of initial NoSQL implementations are for web or other internet (e.g., mobile application) projects;
- *Schema mutability*. If you like the idea of being able to have different schemas for different parts of the same "table", NoSQL may be for you. If you like the database reusability guarantees of the relational model, NoSQL may be a poor fit;
- *Project size*. For a large (and suitable) project, the advantages of NoSQL technology may be large enough to outweigh its disadvantages. For a small, ultimately disposable project, the disadvantages of NoSQL may be minor. In between those extremes, you may be better off with SQL;
- *SQL DBMS diversity*. The choice of SQL DBMS goes far beyond the "Big 3-4" of Oracle, IBM DB2, Microsoft SQL Server, and SAP/Sybase Adaptive Server Anywhere. MySQL, PostgreSQL, and other mid-range SQL DBMS – open source or otherwise – might meet your needs. So might some of the scale-out-oriented startups cited above. Or if your needs are more analytic, there's a whole range of powerful and cost-effective specialized products, from vendors such as Netezza, Vertica, Aster Data, or EMC/Greenplum.

Curt Monash is a leading analyst, strategic advisor and writer of articles and blogs about the software industry (www.dbms2.com).

Dit artikel kwam tot stand in nauwe samenwerking met Techweb.com.