

Relational Databases, Object Databases, Key-Value Stores, Document Stores, and Extensible Record Stores: A Comparison

Rick Cattell December 2010

1. Overview

Traditionally, the obvious platform for most database applications has been a relational DBMS. You might use a specialized parallel relational DBMS if you required high throughput for “data warehousing”, or an object database system if your application had unusual functionality or performance requirements, e.g. for in-memory caching or fast relationship traversal. However, a general-purpose RDBMS like Oracle or MySQL has usually been the answer.

This has changed somewhat recently. There is now recognition in database research that “one size does not fit all”, for example in the widely-referenced paper by [Stonebraker and colleagues](#). And in the Web 2.0 industry, many companies have abandoned traditional RDBMSs for so-called “NoSQL” data stores that provide much higher scalability, or they have built a distributed caching layer on top of RDBMSs. Specialized, OLTP-scalable RDBMSs are also coming to market; I will comment on their prospects as well.

The NoSQL data stores are difficult to describe, because they evolved in an ad hoc fashion, with different data models and capabilities. The basic data elements may be called “objects”, “documents”, or “records”. The “NoSQL” term has been criticized, even by those in the NoSQL movement, because the essence of these systems is not the lack of SQL, but rather a number of things:

1. The NoSQL systems provide little or no pre-defined schema, in contrast to most RDBMSs. Objects (or records, or documents) can have any number of attributes of any type. The lack of a global schema makes it easy to evolve a highly available system: there is no downtime for “schema upgrade”.
2. These systems have a simple query interface, rather than a SQL processor. Some argue that this provides better performance; I will simply argue that SQL makes it easy to get *bad* performance, e.g. by encouraging expensive joins.
3. The most important NoSQL feature is scalability over dozens or hundreds of nodes, in exchange for giving up 100% ACID semantics over distributed data. Instead these systems provide “eventual consistency”, or guarantee consistency only within a single object (or record or document).
4. Finally, NoSQL systems provide high availability, which is necessary to make scalability across many machines useful. This is achieved through transparent failover and recovery using mirror copies. However, all mirrors may not be up-to-date, and mirroring cannot be done at the granularity of an entire database, else recovery and replacement times would be intolerably long.

I believe that several key events sparked the recent popularity of the NoSQL databases:

- In Web 1.0 (with millions of readers but few writers), traditional RDBMSs were fine for data storage. In Web 2.0 (with potentially millions of readers *and* writers), a more scalable solution is essential.
- The popular [memcached](#) demonstrated that in-memory indexes can be scalable, distributing and replicating objects over multiple nodes. The simple key+object data model works for a wide variety of applications and caches.
- Google's [BigTable](#) and Amazon's [Dynamo](#) demonstrated that simple record storage could be scaled to hundreds or thousands of nodes, a feat that RDBMSs have never achieved.

Note that the scalability of NoSQL systems does come with compromises: your application must be able to operate with a lower level of consistency guarantee, without transactions that span many records. I believe the NoSQL systems are popular with Web 2.0 applications because these applications generally have simple operations with few consistency requirements. If decision support queries are required, they can be done offline using older data. The price/performance for NoSQL throughput on scalable open source systems is then compelling.

2. NoSQL Data Stores

The NoSQL data stores can be categorized into three groups, according to their data model and functionality:

- *Key-value Stores* provide a distributed index for object storage, where the objects are typically not interpreted by the system: they are stored and handed back to the application as BLOBs, as in the popular memcached system just mentioned. However, these systems usually provide object replication for recovery, partitioning of the data over many machines, and rudimentary object persistence. Examples of key-value stores are memcached, Redis, Riak, Scalaris, and Project Voldemort.
- *Document Stores* provide more functionality: the system recognizes the structure of the objects stored. Objects (or documents) may have a variable number of named attributes of various types (integers, strings, and possibly nested objects), objects can be grouped into collections, and the system provides a simple query mechanism to search collections for objects with particular attribute values. Like the key-value stores, document stores can also partition the data over many machines, replicate data for automatic recovery, and persist the data. Examples of document stores are CouchDB, MongoDB, SimpleDB, and Dynamo.
- *Extensible Record Stores*, sometimes called wide column stores, provide a data model more like relational tables, but with a dynamic number of attributes, and like document stores, they provide higher scalability and availability made possible by database partitioning and by abandoning database-wide ACID semantics. Examples of extensible records stores are BigTable, HBase, HyperTable, and Cassandra.

The extensible records stores and some of the document stores come with (or are built on) a distributed computing infrastructure, e.g. for scattering / gathering work across nodes, and determining group membership. For example, Bigtable is used with a distributed Google File System (GFS), a distributed lock service (Chubby), and a MapReduce system.

Not all of the systems fall exactly into three well-defined categories. For example, Riak might be categorized as a document store, since it does provide some notion of fields; however it does not provide a field-based query mechanism. More importantly, the systems have major differences in the scalability they provide, and many of them have only hash indices, and thus cannot be used for “range” queries or sorting.

3. New RDBMSs and other options

Some progress has also been made on RDBMS scalability. For example, Oracle RAC and MySQL Cluster provide some partitioning of load over multiple nodes. More recently, ScaledB, NimbusDB, Clustrix, and VoltDB are aimed at higher scalability. VoltDB looks promising, providing scalability on top of a performant in-memory RDBMS with minimal overhead. Clustrix also looks promising, but is sold as an appliance. Other RDBMSs also provide scalability over multiple nodes, but only in limited use cases, e.g. for data warehousing where the data is read-only or read-mostly.

Benchmarks will soon show whether these new RDBMSs provide scalability to 100 nodes or more, but I believe this is possible, as long as you constrain your use of the RDBMS:

- You cannot scale well if your SQL operations span many nodes.
- You cannot scale well if your transactions span many nodes.

Note that the NoSQL systems avoid these problems by not supporting these distributed transaction or SQL capabilities. I have written a [paper with Stonebraker](#) outlining key points that make database system scalable.

It should also be noted that distributed caches have already been used for a decade or more to improve RDBMSs scalability, providing a front-end that can support many of the database operations. This approach has simply become more popular with the introduction of open-source solutions like memcached and the NoSQL systems, and there has been increasing emphasis on the caching layer *replacing* some or all of the RDBMS operations. Previous caching products include Coherence, TerraCotta, GigaSpaces, and object database layers like GemStone GemFire.

4. Choosing a solution

Given this confusing array of alternatives, which data storage system do you choose for your application? This is not an easy question to answer, because it depends on a lot of factors. Here are a few guidelines:

- A relational DBMS remains the best choice, when you can achieve the performance you want on a single server or with a caching front end. There are many more applications and tools available, and there are a number of strong

vendors to choose from, along with “free” solutions like MySQL, PostgreSQL, and JavaDB. If you want the object-oriented programming language integration of object-oriented DBMSs, you can use object/relational mapping solutions, albeit with some performance and convenience drawbacks.

- Again on a single server, an object-oriented DBMS remains the best choice when you need performance reference-following, object persistence, and/or cached objects. There are also cases where you may want additional functionality of an ODBMS, e.g. in supporting persistence for virtually any programming language data structure.
- Some of the more scalable RDBMSs like MySQL Cluster and VoltDB might allow you to get adequate performance for your scale. This is an attractive option if you are already using SQL. Of course, you can also “shard” over RDBMSs yourself, but you take on all of the replication, failover, recovery, distribution, and management problems yourself!
- If you only require hashed indexing of objects based on a single key, you may find that one of the key-value stores is the best solution, particularly if you have a fast LAN and you want to partition your data across RAM in many machines. These systems generally provide persistence as well.
- If you need (or will need) scalability to 100X the speed of a single server, and you cannot use an indexed caching front end (or you need more functionality than that provides, e.g. multi-attribute queries), then you should look at the document stores or extensible record stores.

Here are some guidelines in choosing among document stores and extensible record stores:

- If you can run your system “in the cloud” of Amazon’s or Google’s application environments, you could use their scalable data storage solutions. All of the other systems are open source, so you may prefer Amazon or Google because of the level of support and the maturity of the software.
- If you’re moving on to the open source solutions, you want to consider the maturity of the alternatives. Look at how many contributors are participating, and for how long. I expect some shake-out among the open source projects in 2011; the solutions with the largest company participation are likely to grow in functionality, performance, and robustness.
- Figure out your concurrency and consistency requirements. For example, CouchDB provides asynchronous replication but no partitioning of data. Thus, it only provides a well-proven solution if you have a few “writers” and many “readers” that can live with slightly-old data, or if you are willing to provide partitioning the data yourself (which can be a daunting task, since you must also distribute queries, recovery, and other functions you require).
- If you can’t live with GPL licensing, for example if you’re embedding the solution within a non-GPL product of your own, that eliminates some alternatives like MongoDB and HyperTable.

The NoSQL arena has moved quickly this year, with lots of energy from the open source community. Many people from different perspectives are “weighing in” with their opinions, on various websites. I predict consolidation for the NoSQL products: a small number of them will pull ahead in popularity, functionality, and maturity. Some people predict that these projects will all fall by the wayside as RDBMSs improve in scalability and efficient use of RAM. However, that seems unlikely to me on the current product trajectories. I think there are exciting and different ideas for data storage here. I am currently finishing a [paper comparing the scalable data stores](#) and a [web site](#) with pointers to more information about them.