



The Future of Interoperability: Emerging NoSQLs Save Time, Increase Efficiency, Optimize Business Processes, and Maximize Database Value

Author:

Tim Dunnington

Director of Interoperability, Informatics
Corporation of America



THE FUTURE OF INTEROPERABILITY: Emerging NoSQLs Save Time, Increase Efficiency, Optimize Business Processes, and Maximize Database Value

Interoperability brings patient care coordination into the 21st century by establishing rigorous standards such as Integrating the Healthcare Enterprise (IHE) Technical Framework and Nationwide Health Information Network (NHIN) Direct. But even with the high power-to-price ratio of today's multi-core, multi-terabyte systems, large state-wide or national IHE and NHIN deployments could tax computing resources attempting to track millions of patients and terabytes of documents per year. Relational databases can be utilized to handle this load, but maintaining throughput and providing scalability, while containing costs, is challenging.

A new class of database technologies, dubbed "NoSQL," has recently emerged with the goal of reducing the dissonance between programming languages and databases, and increasing performance in store-and-retrieve style applications. These technologies show promise for storing data naturally so that it can be worked with in the programming language much more easily, and provides higher performance, greater scalability, and a simplified programming interface for specific applications. Large IHE Registry and Repository infrastructure projects, and NHIN Direct HISP installations, show good potential for implementation on NoSQL databases.

Interoperability Background

Hospital integration is nothing new. By the mid-1990s, hospitals and integrated delivery network (IDNs) were well on their way to implementing organization-wide integration projects, a functionality that was enabled by the HL7 standards that define a set of hospital integration events. The result was a relatively small dataset: typical transactions tended to be 1K or 2K bytes. As IDN's grew and the number of HL7-capable systems grew, the number of transactions also grew in proportion. It is not uncommon for an IDN to relay hundreds of thousands of HL7 transactions per day. The total throughput for the system is measured in megabytes. Integration Engines have been developed to help move all these transactions, guarantee their delivery, and monitor all activity. They were designed largely to fill the particular need of event-based transaction processing, and as a result are designed to handle large numbers of small transactions very quickly.

In general, the early integration models offer no "patient care document" that displays the entire episode as a single entity. This ability must rely on a system capable of assembling all the discrete bits into a single whole record that becomes visible to the end-user. Health information exchange (HIE) systems evolved to fill that need, but even with these systems, integration is still largely a development effort: systems do not connect "out-of-the-box." This is why IHE and NHIN formed: to solve the end-to-end interoperability problem.

In addition to shifting the scope of integration from an IDN to a community, or even to a state or national level, there is also a shift from discrete event messaging to patient care Continuity of Care Documents (CCD). These come in XML format, which incurs high overhead cost in processing time. The reason for this is that each XML element name is fairly large in comparison to the data stored in the element, such that even a

Characteristics of NoSQL databases involve:

- Reducing the dissonance between the database and the programming language
 - Less dedicated ORM code
 - Shorter development cycles as a result
- Being schema-less -- so that any document can be stored without the code overhead of cleansing and normalizing the data
 - Shorter database design cycles
 - Support for Hierarchical datasets
- Providing high throughput (store fast, retrieve fast)
- Having very large storage capacity
- Simplified data replication
- Offering extremely large-scale expandability (through distributed processing and “sharding”)

small XML document can easily contain more bytes in element names, namespace declarations, and attribute names than in actual data bytes. They also require more computing power and memory to parse and process than a similarly-sized text document.

As the IHE-enabled communities grow in size to encompass many IDN's, or even an entire state, organizations will be processing a growing number of large XML documents. The trend is clear: much more processing power and storage space will be needed to process the XML data. At the same time, the number of these documents will grow, pushing our hardware and software to its limits. Toward that end, software developers will need a solution that increases the developer's efficiency in coding (performing less coding to bridge databases) and scaling in order to make it easier for users to simply dive in and work with the data.

RDBMS Technologies

RDBMS (Relational DataBase Management Systems) and their associated programming language, SQL (Structured Query Language), have been the mainstay database technology for decades. These products drive business applications around the world, and nearly all business applications and websites start with a relational database.

RDBMS design and SQL are slightly at odds with the programming languages developers use to build applications. System design typically (but not always) starts with a database design first, and then a mapping of the database design to the programming language. In general, this involves creating a set of programming language elements that “hide” the database from the rest of the program. A significant effort must be put into developing this software layer.

This problem is so common that developers have built generic, re-useable tools that will scan a database and automatically create the appropriate code, including all of the necessary CRUD (Create, Retrieve, Update, and Delete) operations. These ORM (Object Relational Mapping) tools are now ubiquitous and available for most platforms (such as Hibernate for Java, or EF and LINQ for .NET). But even these require some amount of work by the developer to: customize the behavior of the code; overcome shortcomings in the default behaviors; or ensure proper performance characteristics. In every case, developers spend significant time and resources mapping data from a database engine to the target programming language, resulting in a significant amount of time spent mapping.

Hierarchical Data

The other disconnect between RDBMS and programming languages comes in the form of hierarchical data structures. Filesystem directories (or folders) are one example of a hierarchical data structure. A directory can contain any number of files or directories. There is no limit to how many directories a directory can contain, or how “deep” the directory tree goes. This does not map well to RDBMS systems, which expect the hierarchy to be fixed. Tricks abound to handle this situation, but they are hacks of sorts: the RDBMS system does not inherently store and retrieve hierarchical data.

Incidentally, the XML data used in CCD documents in interoperability is also hierarchical in nature, and while the hierarchy can be limited by the different CCD subsets (HITSP, NIST and so on), the fact remains that the data in XML does not directly translate well to an RDBMS. In addition to XML, there are many real-world examples of tree structures: organizational charts, family trees, chat forums, business entity models, the hyper linking of the World Wide Web.

Store and Retrieve Applications

A whole class of applications exists that do not need the full power and complexity of RDBMS systems. The work of modeling the domain data to an RDBMS does not provide any functional gain. In the case of interoperability, the basic data operations of IT Infrastructure provide:

- Storage of data, usually a relatively-large XML document
- Retrieval of data, whole and on-demand, with returned data identical to what was stored
- Maintenance of associations between patients, documents, and folders of documents

Applications such as these are referred to as “store-and-retrieve” applications. Unlike the CRUD applications, these need only the “create” and “retrieve” part of CRUD. Data is not updated, and deletes are only performed very rarely (if at all).

Certainly RDBMS provide BLOB fields, and good interfaces into the BLOB data, that provides a storage mechanism for large binary data objects. But these were intended not as a primary data storage mechanism, but as another data type for a discrete element. For example, if we were storing a table of physician contact information, perhaps one of the BLOB fields is a JPG picture of the doctor. The RDBMS does not store the data in the actual data tables, but instead it files on the filesystem. These external files are associated with the table using pointers maintained by the RDBMS system. This means that accessing the data programmatically is not the same as accessing any other field in an RDBMS: another programming model has to be used to access the data, further increasing the complexity of the code.

NoSQL Solutions

Because of all these disconnects – hierarchical data, large datasets, and very large capacity — and other challenges inherent to RDBMS systems, some developers have been searching for a better way.

The result has been an emerging set of technologies referred to as “NoSQL.” These databases provide fast access to storing and retrieving very large sets of data, and in a means that performance does not suffer due to scalability. They include different classes of databases, two of which are:

- **Document-based databases** –These databases store BLOBs of data, with an implied structure such as JSON or XML. The database does not require, or enforce, a structure on the document stored. It simply stores any document whole, but also allows for indexing elements or querying elements inside the document. Some examples here include MongoDB (<http://mongodb.org>) and CouchDB (<http://couchdb.apache.org>)
- **Key-value stores** – These databases store sets of key-value pairs (or “associative arrays”). A single key, typically a string, is used to reference a BLOB of data (the value). The BLOB can be text in any format. These stores are convenient for storing lots of data records quickly, and in a way that is more native to the programming language. The key-value store will store any key-value pair without any pre-defined structure. Examples include Kyoto Cabinet (<http://fallabs.com/kyotocabinet/>) and Cassandra (<http://cassandra.apache.org/>)

As with any technology, NoSQL has its downsides (some of which are corollaries to the benefits):

- Lack of ODBC means that existing development, reporting, and monitoring tools won't work; this also leads to a lack of “built-in” support in programming languages.
- Loss of complete support of ACID principles, particularly around replication efforts; some NoSQL implementations provide “eventual consistency” of replications.
- Development methodologies; developers must re-think implementations.

Nevertheless, these drawbacks do need to be carefully considered before implementing a project in NoSQL. That said, interoperability could potentially benefit from NoSQL solutions in terms of:

- Quickly storing very large numbers of relatively large documents
- Inherently quick retrieval
- Higher capacity
- Replication for disaster recovery situations
- Distribution for vertical scalability and high availability
- Being schema-less
- Storage of hierarchical data

While RDBMS systems can provide these functions to varying degrees, they are not optimized for many of these functions and require compromises in code complexity, technology expenses, and design complexity.

NoSQL solutions, on the other hand, are built from their core to easily support these features, and are starting to add mature functions for querying and reporting.

Conclusion

IHE and NHIN functional designs cannot keep pace with the rapid adoption and interoperability of patient data in health care organizations without raising implementation and operational costs. While mature RDBMS database technologies excel at their ability to store discrete data elements and retrieve those elements in endless combinations, the costs threaten to derail the benefits. Developers must spend more time marshalling and un-marshalling data rather than utilizing it, and, as the size of datasets grows, RDBMS performance suffers – leading to more time and money spent to optimize platform performance. Ultimately, this could make the potential for interoperability negligible due to costs and complexity.

NoSQL seems poised to reduce the dissonance between programming languages and databases, thus filling the needs of the store-and-retrieve application space. Interoperability is more-or-less a store-and-retrieve application, and as such can take advantage of the potential and promise of NoSQL. In doing so, interoperability can benefit by keeping the cost of delivery low, ensuring that the costs of interoperability do not detract from its goal of improving patient care.



Author

Tim Dunnington

Director of Interoperability, Informatics Corporation of America

INFORMATICS
CORPORATION OF AMERICA

About Informatics Corporation of America (ICA)

Informatics Corporation of America (ICA) was established to take innovative technology developed by practicing physicians and informatics professionals at Vanderbilt Medical Center to the broader healthcare market. Its flexible architecture integrates legacy systems to provide a comprehensive clinical view of patient records within Integrated Delivery Systems and Health Information Exchanges/Regional Health Information Organizations. Today, ICA is unmatched in its ability to deliver a cost-effective, proven solution that not only leverages complete data across clinical settings to aid decision-making and improve patient outcomes, but also utilizes real-time clinical information when and where it is needed. Visit www.icainformatics.com



integrating care. **improving health.**