The NoSQL Alternative

By: Ameer Gaafar

For decades, relational databases have been the de facto choice and standard for persisting enterprise data. However, RDBS positioning as the undisputed king of persistence is being threatened by a new family of persistence choices collectively, and bluntly, named NoSQL databases!

The power beneath SQL databases stems from the relational model. A model that allows consumers to query small and dynamic subsets of huge data masses in blazing speed. Ironically, perhaps, this same model is also the weakness that allows NoSQL offerings to become a viable, and sometimes superior, alternative to persistence needs.

Today's enterprise applications attempt to model complex business needs that continue to evolve in pursuit of competitiveness and efficiency. These applications are predominantly written in Object-oriented programming languages, and interface extensively with persistence back-ends to carry out complex logic and computation. But programming languages like C# or Java are *Object-oriented* while SQL databases are *Relation-oriented*, which creates a deep conflict between these two, very different, paradigms! At the surface, these two aspects, logic and persistence, appear separated enough to warrant any concern. However, the software industry quickly learned that this *impedance mismatch* is a primary source of complexity and instability!

Historically, the initial attempts to have Object-oriented programming languages and SQL databases interface with each other was through shallow interface layers. ODBC, JDBC, and their incarnations fall in this category. They represent very simplistic interfaces that allow software developers to compose and pass SQL statements and operate on returned tabular results. While this approach allows developers to use the full power of RDBMS, it does not exploit Object-orientation modeling powers, and therefore limit developers' ability to tackle complexity. The solution that prevailed in the software industry for a long time was to build abstraction layers that hide some, or all, of the relational nature of the database and have these abstraction layers create objects from tabular results for computation purpose, and decompose them into tabular forms for persistence. There are currently different types of these abstraction layers varying in depth and complexity with Object Relational Mappers (ORM) representing the peek of abstraction and depth. ORM layers attempt to completely hide the relational nature of the database, and provide mechanisms to represent Object-oriented concepts that do not have relational counterparts such as inheritance and composition. Application layers that consume ORM services are not aware of the nature of the database, and said to have the *illusion* of having objects in memory, while these objects are in fact rows in database tables!

As elegant as it seems, this solution comes at a steep price! Learning to tame an ORM layer can be an exercise that requires good knowledge of both paradigms, and generalized abstractions mean that application level developers have little or no control over optimizing generated SQL. Inadvertent or reckless use of ORM capabilities can easily lead to catastrophic performance and stability consequences. I do not know of any ORM user who did not witness these catastrophes more often than they wished for!

The *impedance mismatch* between the two paradigms is not the only weakness there is! Today's applications exist on a massive scale and many are being used by hundreds or even thousands of users. While SQL excels at querying small subsets dynamically, it does not offer an equally good solution for massive, extensive, or parallel updates. In fact, SQL was never intended for this use case. Remember what the Q in SQL stands for? A solution that is widely accepted in the industry for this situation is to have two separate databases, one shallow database with minimal relations to accept updates, aka an OLTP database, and another de-normalized database for Business Intelligence (BI) and reporting needs. The solution dictates that the OLTP database will periodically update its BI counterpart through what is widely known as an Extract-Transform-Load (ETL) process. Notice how this solution sacrifices concurrency and compromises the relational nature of the database in search for an optimum solution.

In this context, NoSQL databases offer viable alternatives that tackle the issues discussed above! The *impedance mismatch* is not nearly as painful because many flavors of NoSQL databases do not follow a rigid paradigm. For instance, both document store and key-value pair NoSQL databases can easily persist objects without needless complications. They can even store deep object compositions without deflating them into rigid tabular structures. Another variant named Map-Reduce databases excels at concurrency and scalability. It is actually possible to use these databases in what is called lock-nothing, share-all situations.

Another, less evident, strength of NoSQL databases lies in their ability to handle schema-less data! While relational databases require a pre-defined schema for each table, NoSQL databases do not have this requirement. This simply means that as application persistence needs change over time, perhaps with newer software versions, it is possible to have different generations or versions of persisted data all coexisting in the same data store.

This was a brief introduction to NoSQL capabilities and how they render NoSQL a viable persistence choice for many applications. Relational databases, on the other hand, are likely to stay with us for long years to come, but they are no longer the single undisputed persistence choice, and that's great news for the software industry.