regardless of the paradigm used to model the application domain, most enterprise applications use the relational model for data storage. Relational database technology is mature, widely understood, and successfully deployed in countless applications. However, its dominance has also had some undesirable consequences for application development. For an application that models the business logic in an object-oriented way, the developer faces an impedance mismatch between the application’s object model and the data’s relational model. Object-relational mapping (ORM) frameworks exist to bridge this divide, but ORMs aren’t trivial to use and often introduce more complexity than the problem they solve.

Relational databases use a generic storage model based on tables and columns, so it’s no wonder they don’t scale well for all data types. Google designed BigTable and Amazon came up with Dynamo specifically to address these database issues. Both were implemented as distributed data stores designed to scale to a very large size. Recent experiences in data modeling with social networking applications such as Facebook, Twitter, and Digg also demonstrate this deficiency.

One way to address this problem is to model data so that it remains closer to the way the application uses it. On the basis of practical experiences, I describe a multiparadigm data-storage model that permits applications to work with data in a way that’s semantically closer to its usage patterns.

Toward a Semantically Richer Data Storage

Consider an enterprise application module that must handle document sets that can have many optional attributes—for example, an Address Book that can have fields like Name, House Number, Street Name, City, Zip Code, and Country Code. It can optionally include a list of telephone numbers and email addresses as well. With a relational model, you design it as a table with nullable attributes for optional items. For repeating entries, you use normalization techniques and store data in multiple tables to avoid data redundancy and inconsistent updates and deletes.

Distributing data across multiple tables has the effect of destructuring the semantics of the way your application looks at the data. The application would prefer to store the Address Book in a single structure that keeps the document whole and consistent with the way the domain model would use it.

Today, we have data stores that let us model...
The idea is to use each data store's strengths to meet your data model's requirements.

our data layer exactly this way. CouchDB (http://couchdb.org) and MongoDB (http://mongodb.org) store data in JavaScript Object Notation (JSON, http://json.org) “documents” and let applications manipulate document structures directly through their query engines.

Consider another example where your application must store various routes across cities to find optimal shipping strategies for your clients. Typically, you think of this as a graph, with the cities being the nodes and the connecting routes being the edges. You could store the structure in a relational database and use SQL queries that employ complicated joins across multiple tables. Or you could store the data in a graph database like Neo4J (http://neo4j.org) that offers nodes and relationships as first-class abstractions and various graph-manipulation APIs for use directly within the application layer.

Both of these examples have one thing in common. They express a need for a data model much richer than the universal table/column-based representation that a relational database offers—a need for a specialized representation of each individual data type used in your application. Specialized representation also implies specific query languages for each data store. This is a benefit in that you can use the most expressive language to query your data structures, but it also means learning a multitude of languages and their best practices.

SQL is no longer the universal query interface, so these new data stores are popularly referred to as NoSQL stores. “NoSQL” has many connotations, but the most popular one today is Not Only SQL.

Mult paradigm Programming with Data
When you’re using multiple data-representation techniques, you need the right tool for the right job. When you’re working on a large-scale application, use data stores that meet your application’s access-pattern requirements and offer the desired level of scalability and performance guarantees.

A relational database management system (RDBMS) engine is the right tool for handling relational data used in transactions requiring atomicity, consistency, isolation, and durability (ACID). However, an RDBMS isn’t an ideal platform for modeling complicated social data networks that involve huge volumes, network partitioning, and replication. Graph databases like Neo4J model such relationships much better. CouchDB offers offline data-processing capabilities through replication techniques and allows synchronization with other copies at a later time. MongoDB has blazing-fast in-memory operations.

Cassandra (http://cassandra.org) supports decentralized data storage for efficient columnar access from your application. It has the fault tolerance of Dynamo while offering a more advanced data model. For applications that need huge write scalability, Cassandra has proved to be a very good option.

If you need write scalability, Riak (http://riak.basho.com) is another option that models a key-value data store and offers decentralized access, availability, and network-partition tolerance.

The underlying idea is to use each data store’s strengths to meet your data model’s requirements. This brings the data’s storage model closer to the application-domain model that uses it. Plus, it gives you the scalability benefits these engines offer. The result is a multiparadigm strategy for data management.

With all these NoSQL stores acting as the interface to your application’s domain model, you might still need an underlying relational database to serve as the system of record for generating reports and audit trails, running of other batch processes, and so on. NoSQL stores don’t scale well for such jobs, and the RDBMS world offers lots of tool support in these areas.

Asynchronous Messaging for Eventual Consistency
However, one question still remains: How do you keep the underlying relational store in sync with the other data stores?

One option is to use asynchronous messaging as the backbone for an integration layer. By combining asynchronous messaging with the actor communication model,11 we can establish an architecture for achieving eventual consistency between the underlying database and the online domain-specific data stores.

Figure 1 gives the overall architecture for such a multiparadigm database infrastructure. In this approach, asynchronous messaging replicates necessary changes in individual data stores in the main relational database. The application determines which changes must be propagated to the underlying store as the system of record. For example, if you’re using MongoDB as a data store for online processing, the business components will store all your transaction data in MongoDB collections. When the application logic updates a collection online, the messaging system will trigger downstream updates to schedule jobs on the queue. These jobs are
processed asynchronously and keep the underlying relational data store consistent with the frontal online stores.

The overall application architecture benefits from asynchronous updates in a couple of ways. First, it scales well, because asynchronous processing doesn’t block—and can even be scheduled as offline threads of execution. Second, it guarantees delivery of updates to the underlying store within a specified time interval. The two stores might be inconsistent for a short time, but many applications can tolerate this delay to meet other scalability objectives. In other words, the consistency isn’t instant, so it’s called eventual consistency.

In a sense, the frontal data store is like a cache between the application and the database of record—that is, the SQL store. However, unlike a traditional database cache, the frontal store can provide a persistence abstraction that best fits the application. This multilevel model is also useful when the application requires an SQL store of record for nontechnical business reasons.

**NoSQL Stores: A User’s Point of View**

You must consider two important aspects of your application and infrastructure requirements before selecting a NoSQL storage engine.

First, every data store needs a user-friendly query interface. Unlike the relational world where SQL provides the universal query language, the NoSQL world has no such unifying query language. Every data store provides query interfaces in multiple host languages—Java, Ruby, Python, and so on. The languages vary significantly across storage engines, and the query mechanisms differ across the various stores.

Stores like MongoDB offer user-friendly query APIs as part of the client library implementation. In CouchDB, you must write views, using a map/reduce paradigm, to get data from the database. CouchDB comes with a default view-engine implementation in JavaScript. However, its view architecture is decoupled from the core server, so you can write your own view server using your preferred language.

So, every data store has its own query model and language that might be optimal for its underlying database engine, yet, the absence of a unified query model like SQL in the relational world could deter early adoption of NoSQL data stores.

The second important consideration when selecting a data store is scalability requirements. Each candidate data store has different strengths and weaknesses, which you will need to align with your application requirements. Some features for determining your selection are:

- throughput requirements of reads and writes for your application,
- whether your application needs to handle data distributed across nodes and serve query requests from users even when some nodes fail,
- whether your application needs offline data-processing capabilities,
- your application’s availability requirements, and
- your application’s data consistency requirements.

A complete analysis of all the stores with respect to distribution and scalability is beyond this article’s scope. For more details, see the documentation for each product.

**Advantages of Asynchronous Messaging**

Architectural blueprints like the one in Figure 1 are becoming more common. In one project at Anshin Software, we use MongoDB for document storage in collaboration with Oracle as the underlying system of record. The implementation archi-
When eventual consistency is sufficient for your application, asynchronous messaging provides a robust, scalable way to synchronize data between different data stores. It lets you decentralize data processing and store data in structures that are more closely aligned with the application logic. This leads to prospects for a much simpler programming model without the incidental complexities that additional glue frameworks bring to an application.

Challenges of Asynchronous Messaging

As with any architectural paradigm, there are a host of pitfalls to be aware of.

First, not all systems are suitable for enforcing an eventually consistent model. If your application requires all operations to be immediately available from the underlying relational database, this model isn’t for you. A typical example of such a use case is handling a banking system’s debit and credit transactions, which must be ACID consistent.

Second, the NoSQL database systems discussed here are relatively immature compared to the well-known SQL systems. Many of them are still evolving rapidly. All of them are being used in production systems, but few have reached a version 1.0 level. You can expect these systems to make some incompatible changes in the near term. Furthermore, if the application or data architect doesn’t use the architecture pattern carefully, the result could turn into a database cacophony.

Finally, traditional database architects, accustomed to using a single RDBMS for an application, will have to be convinced of the wisdom of a multiparadigm strategy.

References


NEXT ISSUE:

Software Architecture: Framing Stakeholders’ Concerns
This article was featured in computingnow. For access to more content from the IEEE Computer Society, see computingnow.computer.org.

Top articles, podcasts, and more.

computingnow.computer.org