Object Oriented Databases Are Worth a Closer Look

Object-oriented databases (OODBs) can offer considerable advantages over relational databases (RDBs), yet application developers often opt for RDBs. Don't shortchange yourself. Learn the pros and cons of the OODB option.

by Steve Franklin

espite the advantages that object-oriented databases (OODBs) can offer over relational databases (RDBs), OODBs have not been able to shake the RDB stronghold on data-driven systems. They are a newer technology that has proven popular with many DB architects, yet application developers often opt for RDBs.

Don't overlook OODBs for your next database project. To choose the right one for you, evaluate your risk tolerance, technical requirements, performance needs, and overall technical solution. Under the right circumstance, an OODB can be an excellent tool. Under the wrong circumstance, it can force significant design rework—if not architecture rework—down the road. Learn the strengths and weaknesses of the OODB option.

OODB Strengths

Design Elegance

An OODB can simplify your system's persistency challenges. For dynamic systems, significant effort is often invested in the data access layer. An OODB does not eliminate the need for this effort, but in many cases it can make deciding how to save and retrieve information simpler.

In an object-oriented project, integration between the application logic and the archived data can be fairly transparent. The ability to keep the persistence and application logic in the same object-oriented paradigm simplifies modeling, design tool requirements, and visualization of the system architecture and design. Some OODBs even have small footprints that allow them to integrate with your application on an embedded system.

With some OODBs, you may be persisting not only data but also the entire object, including its implemented behavior. Also, some OODBs allow you to call the persisted object's methods on a remote database server, thereby giving you some advantages in scalability and distribution. In a relational database, by contrast, you'd need to implement stored procedures or COM objects to accomplish this, forcing some duplication of coding efforts or a more awkward architecture.

Reduced Development

Training and ramp-up can represent a significant cost to projects. Furthermore, tool requirements, peer reviews, build instructions, and test requirements can be complicated because of multiple tool, language, and environment support. An OODB can reduce the number of required languages in your architecture. The development team would no longer need to have skills in technologies such as JDBC, Pro*C/C++, and PL/SQL. Instead, developers are able to work with more commonly known concepts such as Java objects, iteration through collections, etc.

OODBs also can reduce development time by allowing developers to focus on object persistence, not the decomposition of objects to rows and parent/child relationships in one or more tables. In most cases, a developer simply saves an object to the database. Compare this single-step procedure with the challenges of saving a complex object to an RDB. Not only does the developer have to reduce the object to a series of records, but transaction management, error handling, and subsequent reconstruction also will be required when retrieving the object. Furthermore, object attributes must be ported to ANSI-compliant data types.

Some OODBs provide fairly seamless integration between the application and persistence code, which simplifies debugging and testing of the final product. Whereas some database architectures require libraries and a protocol between your application code and your data store, an OODB can skip this constraint and allow your debugger to monitor what happens as the data is being saved.

Performance Perks

Depending on the product and implementation you pursue, OODBs can bring significant performance to the right type of application. If your application uses an RDB and you must reconstruct an object from data in the database, you frequently have to perform multiple queries. Each of these queries incurs overhead and, without careful planning of indices, can result in serious performance penalties. An OODB reduces this problem significantly, provided that you know the object identifier or OID —the database retrieves the object in its entirety or as a lazy fetch, a query technique by which your code loads portions as needed.

A number of OODB implementations also employ client-side caching in addition to server-side caching. Although this has its disadvantages, it certainly can improve application performance. You often will see "warm" and "cold" database benchmarks, where a warm benchmark implies one or more reruns of a test to evaluate caching and buffering advantages. OODBs often excel at warm benchmarks because of their client-side caching facilities.

OODB Weaknesses

Object-oriented databases have weaknesses that you must consider as well. One of the most obvious and significant drawbacks is the risk involved in an RDB-to-OODB migration. Relational databases have their own cons but they are proven and successfully used on data-driven systems. The architecture/design patterns for integrating RDBs into your system are well defined and demonstrated.

Tight Application Coupling

In many OODB implementations, the OODB is tightly coupled to the application. This simplifies both the design and code, but the data abstraction layer has some value also. By removing it, you lose a layer of insulation from the database. This can make it more difficult to migrate to a competing product without significant changes and re-tests. You could always implement a data abstraction layer around the OODB implementation. Although you would lose some of the OODB's benefits by doing this, development of this layer for an OODB certainly would be cheaper than creating an RDB equivalent.

Performance Flaws

I have already discussed an OODB's strength at retrieving entire objects efficiently. OODBs typically are weaker for ad hoc queries against the database. Loose

navigation through data can be challenging, too, and OODB query optimization and functionality often lags behind major relational database products.

Some OODB implementations do not provide sufficiently granular locking. Consequently, your code could lock huge groups of objects if you are not careful. Whereas most RDBs implement row-level locking, some OODBs still implement page locking. Furthermore, locks can span relationships within an object, potentially allowing one action to lock a significant amount of data.

Limited Platform Support

Even when an OODB is implemented in Java, you should still look for a guarantee of support on a variety of platforms. Although Java is portable, complex Java software can have quirks and nuances in a given environment, to which only the vendor can help you migrate. Some OODB vendors are unable to devote adequate resources to a broad support base across multiple platforms because they have smaller budgets than major RDB vendors. To gauge platform support, find out what the vendor provides and check newsgroups (e.g., comp.databases.object and comp.object) to monitor the frequency of discussions for a given platform and product combination.

Porting Difficulties

Saving objects to an OODB is vastly different from saving objects to most relational databases. Consequently, it is a chore to move your RDB-based system to an OODB and once you commit your design and development to an OODB, you may find it difficult to move back to an RDB.

For many projects, it is much safer to migrate from a relational database to an object relational database. Some vendors such as Oracle have made this migration a natural one by simply adding object relational features to their core product. As a result, your legacy RDB can often be extended to incorporate object relational functionality and ported in a piecewise fashion.

Multiple Skill Requirements

As with RDBs, each OODB tends to have its own proprietary quirks and extensions. Consequently, finding people with skills that are specific to a given OODB will be more challenging than finding people with mainstream RDB skills for a given database (i.e., DB2, Oracle, Microsoft SQL Server, etc.). Furthermore, finding individuals with proven, deep experience in OODB administration will be challenging. Granted, many OODB users feel that OODBs require less administration in the early stages, but like most growing systems OODBs still require tuning in preparation for deployment. Switching from a relational to an OO database paradigm also will require some training and mentoring because issues such as performance, locks, and joins often require different approaches.

Complex Queries

Query support varies quite a bit among the various OODBs. Your application will not always be able to retrieve an object by its Object ID. Rather, there will be times when you need to search by ranges, patterns, and fuzzy criteria spanning objects that do not have obvious relationships. Ad hoc query support seems to be an area where OODBs find it difficult to compete in both performance and features. It is likely that this functionality will continue to mature, and many applications do have well-defined retrieval requirements that will not suffer due to any loss of ad hoc query functionality.