When to Use an ODBMS

By Rick Grehan

Before we begin, we should acknowledge reality. There are situations in which an RDBMS works just fine. There are plenty of remarkably good applications out there that have been running on relational databases for years. Similarly, there are remarkably good RDBMS packages available. In fact, two noteworthy open-source RDBMS products -- MySQL and PostgreSQL -- are by anyone's measure wonderful products.

However, there are times when the superior database for an application is an ODBMS. Below is a list (not exhaustive) of those sorts of applications in which ODBMSes tend to provide a better solution than RDBMSes.

Embedded DBMS Applications

Embedded DBMS Applications might involve a database on partially connected mobile devices or as a OO data cache in an application that demands a super-fast response time. If you are using Java or .NET, this requires a self-contained, non-intrusive, and easy-to-deploy persistence solution for the client-side or in the middleware.

Why? Storing Java or .NET objects 'just as they are in memory' is always the leanest and least intrusive way to implement a persistence solution. Using an RDBMS requires the overhead of object-relational mapping, resulting in an increased demand on resources. In addition, an RDBMS approach requires greater administration involvement, especially when you must deploy updated class schemes to your installed base.

Complex Data Relationships

Or, more precisely, *complex object relationships*. In such applications, classes define multiple cross-references among themselves. Applications that include networked data structures fall into this category.

Why? Complex cross referencing among objects can be difficult and error-prone to model in a relational database system. Relationships among objects are often dealt with (in an RDBMS) using foreign keys. So, fetching an object -- and then fetching objects it references, and then the objects *they* reference -- can result in complicated and difficult-to-maintain code.

Meanwhile, most ODBMSes implement *reachability persistence*. That means that any object referenced by a persistent object is also persistent. (If `Object A` references `Object B`, and `Object A` is persistent ... then `Object B` becomes persistent automatically.) Typically, the depth to which reachability persistence extends in an object tree can be specified by the programmer. As a result, whole "gobs" of objects can be stored or fetched with a single call; the ODBMS engine handles the details of maintaining the references when objects are stored, and satisfying them when objects are fetched.

'Deep' Object Structures

This is related to the preceding item. Not all data is easily organized into the tabular, rows-and-column form that one associates with RDBMSes. Some data is best organized in 'unusual' graph structures, or various sorts of tree structures. A very "deep" tree structure, for example, presents to the database programmer a lengthy parent, child, grandchild, great-grandchild, etc. string of references that can be tricky to support in an RDBMS.

Why? As above, highly-connected object structures are not easily translated to "fit" into a relational database. The conversion code can be confusing and difficult to maintain (in one sense, the original structure is *lost* in the translation). It might also be prone to integrity corruption (e.g. only a 'piece' of the tree gets stored), and requires large sections of code to be wrapped in transactions to safeguard the data relations ... thus impeding performance in multiuser applications.

As above, an ODBMS requires no translation of the original structure into a model for the database. If the ODBMS provides programmer control over the depth of reachability persistence, the developer can control whether a whole tree is fetched or stored, a branch is fetched or stored, or individual twigs are fetched and stored. And, again, the integrity of the structure is preserved by the database engine itself.

Changing Data (Object) Structures

Suppose you anticipate that the class structures of your application will change over time. Perhaps you recognize that there's a good probability that new data members will be added, or new object relationships will have to be added. (Most applications evolve as they age; and the data structures they support must evolve as well.)

Why? An ODBMS will typically weather data structure changes more easily than an RDBMS. If you use an RDBMS, you'll likely have to change the schema (to fit the new object structure), then alter the query code to handle the changes. You may even have to write a one-time conversion application to update the tables to the new format (the sort of *throw-away* application you only write when you must, and wish you didn't have to waste the time doing).

Some ODBMSes allow you to change the structure of objects "on the fly". You can mingle "old" and "new" objects in the same database. If the new object structure has additional fields, reading an old object into the new application simply loads the additional fields with default (e.g., null or zero) values. If the new object structure has fewer fields, reading an old object into the new application skips the now non-existent fields. (The ODBMS db4o, for example, even provides a mechanism whereby "old" objects can be "upgraded" to new objects invisibly ... as they are accessed from the database.)

Your Development Team is Using Agile Techniques

Agile programming techniques are rapidly gaining in popularity as they demonstrate their benefit in reducing development errors. An ODBMS will fit more smoothly into Agile development than an RDBMS.

Why? We could not say this better than agile guru Scott Ambler in his whitepaper for the ODBMS.ORG portal:

> *Modern software development processes are evolutionary in nature, but more often than not agile. Agile techniques include refactoring, agile modeling, continual regression*

testing, configuration management of all development assets, and separate sandboxes for developers to work in. The use of relational database (RDBMS) technology complicates the adoption of these techniques due to the technical impedance mismatch, the cultural impedance mismatch, and the current lack of tool support. Object databases (ODBMSs) make it easier to be agile.

You're Programming in an OO Language

This might seem so obvious that it's almost ludicrous to bring up ... but it should not be discounted.

Why? Think about it. Using an ODBMs instead of an RDBMs means that you don't have to write translation code to pass data back and forth between row objects fetched from the database and actual objects in your application. Nor do you have to write object/schema mapping code (if you're using an ORDBMS). This could be an important consideration if, for example, you have to maintain multiple applications that access the same database, but in slightly different ways. In such a situation, you have to make sure that all of the translation code in all of the different applications are in synchronization. And, if something changes in the structure of the database, you have to search through all your applications and make sure that the change is properly accounted for.

With an ODBMS, the access is the same from all applications ... because the objects being fetched and stored are being manipulated in the same way. And, if you've properly factored your applications, a change in the class structure is a change to a single library. No searching through applications to fix up the translation code.

Your Objects Include Collections

Your application includes one or more classes that define members that are collections (a `List`, a `Set`, etc.)

Why? A collection within an object often represents a one-to-many relationship. Such relationships, modeled by an RDBMS, require an intermediate table, that serves as the link between the "parent" object (kept in one table) and the objects in the collection (kept in another table). The whole matter becomes even trickier if the collection is allowed to store objects of different classes.

Meanwhile, most ODBMSes would have no trouble with such an arrangement. The collection is treated as just another object (albeit a potentially 'deep' object), and most ODBMSes will allow you to fetch and store the parent object -- along with its member collection and all the contents -- with a single call.

Data is Accessed by Navigation Rather Than Query

This is actually related to the first two items listed on this page. It is often a natural consequence of such object structures.

Why? If data is stored in a highly-networked structure, and data access if primarily via navigation through the object structure, rather than a query on data values, an ODBMS is almost certainly superior. Navigation through the tree -- using an RDBMS -- resolves into a series of query-and-fetch operations (typically, SQL `SELECT` statements). In an ODBMS, navigation through the tree is expressed naturally using the native language's constructs. The resulting code is easier to understand

and maintain.

Conclusion

As we stated at the outset, there are some instances in which using an RDBMS makes practical sense. We have described various circumstances in which the use of an ODBMS is a more natural choice.

Rick Grehan is a QA Engineer at Compuware/Numega labs, where he has worked on Java and .NET projects.

He is also a contributing editor for InfoWorld Magazine. His work has appeared in Embedded Systems Programming, EDN, The Microprocessor Report, and Computer Design. Before coming to Compuware, Rick was on the Discover DSP Project at Metrowerks, Inc.

Earlier, Rick was a Senior Editor at BYTE Magazine, where he was the Lab Director, and authored BYTE's JavaTalk column.