Lost in translation

Christian Merenda
OOMEGA GbR
Fingerhutstraße 6
80995 Munich, Germany
christian.merenda@oomega.net
www.oomega.net

November 8, 2007

Anticlimax of object-oriented databases?

Nearly every programmer already faced the problem of choosing the appropriate technology to structure the data of his specific business domain.

Actually the decision is very hard, as you have not only a tremendous amount of products to choose from, there are also plenty of standards which compete for your attention. In the context of Java object persistence you are confronted with different standards like ODMG, JDO, EJB, XML or the recently published Java Persistence API.

Though object-oriented databases were expected to gain substantial market share in the nineties of the last century, they have not established on a grand scale until today. Why?

- One major problem of object databases is the lack of standardised implementations. Nearly every object database has its own query language and differs in many aspects from the rest of the market. Actually the consortium of object database vendors, the Object Data Management Group (ODMG) disbanded in 2001.[1]

- The continuous market power of relational database vendors and the widespread adoption of corresponding products might be another deciding issue. In fact, the third version of the ODMG standard was broadened to include object-relational mapping systems. Obviously this shift for the benefit of relational databases and application servers come at the expense of object-oriented databases.

In my opinion the most urging challenge regarding object database technologies is to intensify market communication. A good way to explain the benefits of ODBMS is to differentiate the various concepts and to discuss them separately. I'm going to proceed with that and it will turn out that object and relational databases are on the way to converge to each other.

But let me clear one last point first: It's likely to be misunderstood when talking about terms which come in different flavour to the mind of readers with diverse backgrounds. Especially when talking about object databases as the products differ

[1] Fortunately the Object Management Group (OMG) has decided in 2006 to develop the "4th generation" standard for object databases.

quite a lot. Not only for that reason I will talk about the neologism object-structured databases (OSDB) to get in the way of unwanted connotations.

Object-Structured Data Modelling vs. Object Encapsulation

The concept of object encapsulation is by far one of the most exciting features of object-oriented programming languages. Therefore many object databases adopted that stunning concept. In my opinion this was a great mistake.

Obviously persistent data is not meant to be bound to one specific programming model or language. Long term persistent data outlives programms in heterogeneous environments and you cannot think of object databases restricting access to only a few programming languages which are all tied up with the object oriented programming model.

In my opinion object encapsulation perfectly matches developers needs in the context of an object-oriented API to a database, which comes in addition and not at the expense of a dynamic API. The former can be used to navigate through and update the database with the help of a published set of interface methods, the latter is a convenient way to analyse the data dynamically with the help of a query language.

"In fact there is an intrinsic tension between the notion of encapsulation, which hides data and makes it available only through a published set of interface methods, and the assumption underlying much database technology, which is that data should be accessible to queries based on data content rather than predefined access paths. Database-centric thinking tends to view the world through a declarative and attribute-driven viewpoint, while OOP tends to view the world through a behavioral viewpoint. This is one of the many impedance mismatch issues surrounding OOP and databases."[2]

But object encapsulation is not the only one feature brought to us by object-oriented programming and modelling languages. Think of classes, inheritance, polymorphism, extents, attributes and associations. These modelling concepts, which are widely adopted due to the Unified Modeling Language (UML) and the evolvement of object-oriented programming languages like Java, C++ or C#, are perfectly fine to express data models and valid database states. It's a superior and powerful model for databases and exactly that is the reason why relational database systems are currently upgrading to object-relational databases.

An object-structured data model has two main benefits.

- At first the expressiveness compared to the relational model is greater, so a bulk of integrity constraints are part of that model and therefore can be guaranteed by an object-structured database system. This leads to less code and superior correctitude of software applications.

- Secondly an object-structured data model is well suited for a comfortable object-oriented language binding. So a seamless integration into the leading object-oriented programming model is possible.

[2] Wikipedia (Object database, 2007), http://en.wikipedia.org/wiki/ODBMS (08.11.2007)

Hopefully it turned out that object-oriented concepts can be divided into two areas: object-structured modelling and object encapsulation. The former is an essential part of core data modelling and the latter comes to play in the context of object-oriented data access. A specialised object-oriented API would work with object encapsulation as soon as object-structured data arrives at the client application.

This clear separation of concepts leads to the following pretty result: Programming language independence is within reach, because language bindings to non object-oriented programming languages can be provided as well as a seamless integration with the object-oriented programming model.

Object-Oriented Database API vs. Relational Data Access

In the last passage I argued that object encapsulation should be treated as part of an object-oriented database API. Now let us have a closer look at object-oriented database access. Generally we can distinguish between two different access modes.

- Create or find object-structured data and use the surrounding object skin to navigate through the database and/or update the database contents. Delete objects if they are not needed any more.

- Query the database and calculate dynamically structured information thanks to the well-known object-structured data model.

To speak in the words of relational databases INSERT, UPDATE and DELETE statements are seamlessly integrated into the object-oriented programming language whereas the SELECT statement is partitioned into three classes:

- **Query for dynamically structured data**: Query the database to calculate dynamically structured informations.

- **Query for object-structured data**: Query the database to receive one or more objects to start from.

- **Navigational access**: Use the methods of encapsulated object-structured data to navigate through the object graph.

One important feature of object-oriented databases is exactly that object-oriented API. Object-oriented access to databases is probably one of the most urging topics of the software industry - debated for nearly two decades now.

Object-oriented database vendors compete with application servers and object-relational mapping solutions. You can take it for granted that O/R mapping solutions can never reach homogeneous database systems in terms of performance as an unnecessary and annoying layer is introduced into real-world systems which impossibly boosts performance (and developer productivity).

On the other hand the emergence of O/R mapping solutions can be easily understood: Here you have a clear separation between an object-oriented API and the (relational) data back-end. Therefore you can communicate to the database with

nearly every programming language and additionally you are satisfied with seamlessly integrated object-oriented access.

The impedance mismatch between object-structured data (how it is needed in the context of an object-oriented API) and the relational model is likely to disappear in the future as the major relational database vendors are migrating to object-relational concepts. This might improve performance of O/R mapping solutions in the first step. Afterwards, I guess, the object-relational database vendors will offer an object-oriented API as part of their products. And the end of the story is: O/R mapping solutions will disappear in favour of fast and homogeneous object-relational database systems. Or should I call them object-structured databases?

But how much time will this journey take? Probably it will last quite long as relational database vendors are satisfied with their respective market share today. As market rules apply also to the database business the journey might be abbreviated by competition, especially by object-oriented database vendors.

Object-Oriented Middleware Layer

As we now have a consistent view of an object-oriented data access API, let us discuss an object-oriented middleware layer - one of the major benefits of object-oriented databases. The middleware layer comes into play if the client and the database server reside on different machines.

If you query your database for object-structured data the result is transported to the client and encapsulated with object skins. If you use these objects to navigate through the corresponding object graph, linked objects must be transparently downloaded on demand. This leads to the following architecture:

- Methods are invoked at the client in order to save network costs.
- Persistent objects are physically available in the main memory of the client application thus they can be used as the application model.
- Object updates are transparently forwarded to the database server before query resolution or during a transaction's commit operation.

These client-side caching facilities interlocked with concurrency control mechanisms are really a great benefit for the software developer of object-oriented applications.

In the context of *Object-Structure Data Modelling* the actual data transmitted between the client and the server can be programming-language independent. Therefore the server does not need to know whether the client is written in Java, C++ or C#. It delivers object-structured data and the client surrounds it with a matching object skin.

Object-Structured Database Storage Back-end

Persistent data is typically stored on a hard-drive. When it comes to information processing the data of a database is not loaded to main memory as a whole. The data is read in parts from the hard-drive as needed and may be written back to the disk as changes and transaction commits occur. Pages are atomic units of work with respect to physical data access: a page can be loaded within a single I/O operation.

As a matter of course databases try to reduce the amount of I/O operations as they are a common bottleneck in terms of performance. Therefore data is organised into clusters which in turn are mapped to physical pages. The cluster organisation typically differs for object-oriented and relational databases.

- In object-oriented databases the cluster organisation is often optimised for navigational access: linked objects are put into the same cluster. If you navigate from one object to another (which is typically for object-oriented data access) a second I/O operation is often not necessary as the linked object was part of the same cluster recently loaded into main memory.

- In relational databases clusters are optimised for relational data access, i.e. query resolution in terms of the relational algebra. Tuples of relational tables are typically put into the same cluster as the contents of tables as a whole play a dominant role for querying a relational database.

Apart from the cluster organisation navigational pointers are part of physical object representations whereas in relational databases expensive join operations are necessary to navigate between linked tuples.

Therefore it is often stated that object-oriented databases outperform relational databases in terms of object-oriented or better to say navigational access. This is true for sure. Especially the physical representation of navigational pointers comes at nearly no cost, but for the great benefit of object navigation. On the other hand relational databases may be faster in terms of traditional data access as the object-oriented cluster organisation may come at the expense of efficient query resolution.

Obviously there is a performance trade-off between traditional data access methods and object-oriented navigational access. The former is optimised by relational and the latter by object-oriented database vendors whereas object-structured database vendors would try to balance this trade-off, because both access methods have their right to exist side by side.

Please be aware that an object-structured data model does not interfere traditional data access tuning and so does definitely not determinate the performance trade-off for the benefit of navigational access. The actual implementation of the database back-end may vary from one database vendor to another - strategic design decisions and the choice of algorithms influence the performance. As a conclusion standardised performance arguments pro or contra object-structured databases would be actually worthless.

Finally I should mention one last argument here. Think of object-relational mapping solutions and/or J2EE application servers and try to analyse the performance behaviour:

- The relational database organises tuples into clusters and stores them in pages. If data is accessed, pages are loaded into main memory. A physical representation of relational tuples can be found in main memory.

- The object-relational mapping engine transforms relational tuples into objects: another physical representation in main memory. These objects can be used by the application programmer.

It should be clear that managing the same kind of persistent information twice in main memory and transforming the representations back and forth is by far not the fastest way to process data.

Transparent Persistence

This very last chapter is devoted to my favourite topic: transparent persistence. This technique is propagated by the JDO standard to some extent and even more by a few database vendors or open source communities.

What is transparent persistence?

Take your favourite object-oriented programming language, program your application as you would do without a database and here you are.

You shouldn't be surprised, that I definitely do *not* share this mindset. Please remember.

- As stated earlier data should neither be bound to a specific programming model nor to a particular programming language.

- Object-oriented programming interferes with database queries as encapsulation is a main concept of object orientation.

- Object-oriented modelling with programming languages lacks superior modelling features like attributes (in the sense of true part-of relations), bidirectional associations or exact multiplicities apart from others.

Anyhow I'm an absolutely enthusiastic advocate of transparent persistence. Did I take you by surprise now? Let's clarify my contradiction.

In my opinion transparent persistence is not about changing the way of modelling and accessing object-structured databases. This should be done via a superior object-structured data model and a comfortable object-oriented data access API that supports queries and navigation side by side.

Rather you should think about changing the way of programming object-oriented applications in general. To separate the model from the view and controller is actually best practice irrespective of persistent or transient data, isn't it?

- Wouldn't it be quite nice to query your transient data in main memory in exactly the same fashion as you would interact with a database server?

- Wouldn't you get excited to use modelling features like bidirectional associations and exact multiplicity constraints without the need of additional and cumbersome coding?

- Wouldn't you love the ability to switch from a simple transient in-memory application to a client-server multi-user environment without the need of changing a single line of code?

The answers are quite clear: probably you would be in favour of such a technology. Hence my very personal view of transparent persistence is that an object-structured database product should offer language specific in-memory implementations and therefore could and should be used in the general case even if you are dealing only with transient data.

Conclusion

Hopefully I could give you a thorough understanding of the various competencies of object databases, the differences between the object-oriented and the relational approaches and their convergence to the golden mean.