Fighting Impedance Mismatch at the Database Level

A Technical White Paper by:

Mary A. Finn

Product Marketing Manager

InterSystems Corporation

Introduction

With the maturation and wide acceptance of Java, object-oriented programming has moved to the foreground of the application development landscape. Because of their rich data models and support for productivity-enhancing concepts such as encapsulation, inheritance, and polymorphism, object technologies like Java, C++, and COM, are favored by today's application developers.

However, much of the world's data still resides in relational databases. Developers of database applications (that is, any application that accesses stored data) often find themselves fighting impedance mismatch: the inherent disconnect between the object and relational data models. Efforts to "map" relational data into a usable object format are often detrimental to both programmer productivity and application performance.

However, impedance mismatch can be mitigated by the proper choice of database technology. This paper defines impedance mismatch and gives two simple examples of how it affects application development. It then discusses, with regards to impedance mismatch, the pros and cons of three kinds of database: relational, object, and Caché, the multidimensional database from InterSystems.

Understanding impedance mismatch

Impedance mismatch is a term borrowed from electrical engineering, but in the software world it refers to the inherent difference between the relational and object data models.

Very simply put, the relational model organizes all data into rows and columns. Every row represents a record, and the columns represent the various data items in a record. If the data is too complex to be described by a two-dimensional grid, additional tables are created to hold "related" information. Thus, every table in a relational schema will hold some, but not all, of the data items for a great many records.

The object data model is not constrained to keeping data in rows and columns. Instead, the developer creates a definition – a template – that completely describes a certain class of information. Every record (object) is a specific instance of that class. Thus each object contains all the data items for one, and only one, record. But that's not all. Class definitions may also include pieces of code, called methods, which act upon the data described by the class. There is no analogous construct in the relational model.

A simple example

To illustrate the difference between the two data models, assume that you are developing an accounts receivable application. Your application will undoubtedly need to keep track of a number of invoices, each of which will have some header information (such as the invoice date), an invoice number, and one or more line items. Every line item will include, among other things, information about the product ordered, and the quantity of product ordered.

One way of modeling the invoice in a relational database is to create two tables. One table – called Invoice – includes the header information that only appears once on each invoice. Another table – LineItems – contains columns for the Invoice_Parent, Line_Item_Product_Code, and Line_Item_Quantity. The first of these is especially important because it is this value that "relates" the line items to information in the Invoice table.

Note that neither table contains all the information about any given invoice. Instead, each contains some of the information about many invoices. If your application is designed, for example, to print an invoice, it must access both the Invoice and LineItems tables for the header and detail information respectively. Also note that the tables do not contain any instructions about how to format the data for printing. Those instructions exist outside the database itself.

In the object model, data need not fit into rows and columns, so the Invoice class definition will look like a list of all the data items that make up an invoice. There will be properties containing the header information such as InvoiceDate, InvoiceNumber, etc., and a collection of one or more instances of the LineItem class. The LineItem class includes the properties ProductCode and LineItemQuantity.

Class definitions are merely blueprints of the data format. Each individual invoice is one specific instance of the invoice class, and contains the specific instances of the LineItem class which belong to it. Thus, every Invoice object contains all the information for a given invoice, and only information for that invoice.

But class definitions may also contain methods that act upon the data described by the class. For example, your Invoice class may include a Print( ) method that dictates how to format invoice information for printing. Persistent objects will include some sort of Save( ) method that specifies how objects are stored in the database. The default implementation of the Save( ) method will be determined by the structure of the database engine, and provided by the database vendor.

Impedance mismatch when manipulating the database

Consider the case of creating a new invoice with one line item in your accounts receivable application. If you were programming against a relational database, your code would look something like that shown in Example #1. It would include two Insert statements: one to add the header information to the Invoice table, and another to add the detail information to the LineItems table. Insert is a standard SQL command, and the relational database vendor will provide for its implementation.

Example #1 :
Creating a new Invoice using the relational model

```
If (flag="New") {
Insert Into Invoice
(Invoice_Date, Invoice_Number)
Values(Today,:NewInvoiceNumber)
Insert Into LineItems
(Invoice_Parent,Line_Item_Quantity,
Line_Item_Product Code)
Values(:NewInvoiceNumber,:Quantity,:ProdOrdered)
}
```

The code for saving an invoice with one line item using an object model is shown in Example #2. Except for syntax details, it looks quite similar to the relational example. The main difference is that the Save( ) method is only called once.

Example #2:
Creating a new invoice using the object model

```
If (flag="New") {
objInv=new Invoice()
objInv.InvoiceDate=Today
objInv.InvoiceNumber=NewInvoiceNumber
objLI=new ObjInv.LineItem()
objLI.LineItemQuantity=Quantity
objLI.ProductCode=ProdOrdered
objInv.Save()
}
```

Now imagine that you want to write the business logic for your application in an object-oriented language such as Java or C++, but you need to store your data in a relational database. To accomplish this for your invoice, the SQL Insert statements must be programmed within the Save( ) method of your Invoice class definition. Here is one manifestation of impedance mismatch – an object class with a collection having to be translated to the disparate tables of a relational database engine.

Impedance Mismatch in Design

Another form of impedance mismatch can crop up during the application design process. In addition to enabling a richer, more intuitive way of modeling data, object technology encompasses several concepts that significantly enhance programmer productivity. In particular, object technology supports the concepts of inheritance and polymorphism.

Inheritance refers to the fact that one class definition can be derived from another. For example, in your accounts receivable application you might create a generic InvoiceTemplate class, and have the more specific SoftwareInvoice and HardwareInvoice classes inherit properties and methods from InvoiceTemplate. (They may also include non-inherited properties and methods that are specific to each class.) As the application evolves, if changes are made to InvoiceTemplate, inheritance dictates that those changes are automatically reflected in the SoftwareInvoice and HardwareInvoice class definitions.

Polymorphism refers to the fact that different implementations of a method can share a common interface. For example, the Print( ) method in SoftwareInvoice and HardwareInvoice may include different instructions for formatting, etc. However, to print an invoice, your application only needs to load an object into memory, and call its Print( ) method. Thanks to polymorphism, the object will "know" how to format itself for printing, depending on which class it belongs to.

Neither inheritance nor polymorphism exist in the relational model. Some large relational database vendors such as Oracle, MicroSoft, and IBM have attempted to implement object-oriented design concepts, but the results generally fall short of the capabilities expected by object programmers.

Approaches to Mitigating Impedance Mismatch

The two examples of impedance mismatch given above are very simplistic, but they serve to demonstrate the problem. The work required to "normalize" impedance mismatch can be significant, and grows dramatically as application complexity increases. However, the effects of impedance mismatch can be substantially reduced by the proper choice of database technology. Let's consider three options for data storage: a relational database, a "pure" object database, and the Caché multidimensional database.

Using a relational database

This paper has already discussed how trying to use a relational database with an application grounded in object technology poses serious impedance mismatch problems. But sometimes developers don't have a choice. They may need to access existing data that resides in a relational database. In that case, one option is to use an "object-relational mapping" tool, whether it be a stand-alone tool, or the mapping capabilities built in to some so-called "object-relational" databases.

In essence, mapping tools create a file – a map – that contains the code for translating between objects and relational tables. Developers must specify exactly how that translation is to be done, that is, which object properties correspond to which data columns in which tables, and vice versa. Once created, the map is saved, and invoked every time the application moves data to or from the database. Some object-relational mapping tools provide a runtime caching component to help counteract the performance penalty introduced by translating data between objects and relational forms.

Aside from any runtime performance problems, object-relational mapping can significantly slow down application development. Most mapping tools do not implement, or only partially implement, object modeling concepts such as inheritance, polymorphism, etc. Therefore, as an application is adapted and modified, new, updated object-relational maps must be created.

Developers battling the impedance mismatch between object-oriented applications and relational databases might want to consider migrating the data into a more object-friendly data store. They must weigh the one-time effort required to reformat and transfer the data against the ongoing work and performance losses of using an object-relational map.

Using an object database

At first glance, it would appear that impedance mismatch can be totally eliminated by storing data in a "pure" object database. That is – partly – true. In general, it is easy for an object-oriented application to interact with an object database. However, in this scenario, impedance mismatch occurs when you want to run an SQL query against the database. SQL is by far the world's most widely used query language, and it assumes that data is stored in relational tables. Some object database vendors provide data access via an object query language (OQL), but these languages do not enjoy widespread acceptance. In order to be compatible with common data analysis and reporting applications, an object database must support ODBC and JDBC, and must therefore provide some mechanism for projecting data as relational tables.

The typical solution is, once again, mapping. The drawbacks of mapping – performance losses and the lack of support for data model evolution – still apply. The upside is that the map only need be invoked when an SQL query is run against the database.

Using a multidimensional Caché with Unified Data Architecture

There is a third option for data storage – Caché, the multidimensional database from InterSystems. Although multidimensional databases are often thought of as playing in the data warehousing arena, Caché is designed to be part of transaction processing applications. And it implements a unique approach to reducing impedance mismatch – the Unified Data Architecture.

Thanks to the Unified Data Architecture, the object and relational data models "share" Caché's multidimensional data. Multidimensional arrays are easily projected as tables because tables are nothing more than two-dimensional arrays. Similarly, there is easy correlation between objects and multidimensional arrays because neither is constrained to the rows-and-columns format of relational technology. The translation between data forms is automated, and becomes part of the compiled data definition. To the developer, every table is effectively an object, and every object is one or more tables.

Some other attributes of Caché's Unified Data Architecture:

- **Full concurrency**
  Updates to the data made through the relational interface are instantly accessible via the object interface and vice versa.

- **Support for data model evolution**
  Changes to the data structure definition are automatically reflected in both the object and relational representations.

- **Full SQL support**
  SQL DDL, DML, and DCL commands are all supported.

- **Full object support**
  Object modeling concepts such as simple and multiple inheritance, polymorphism, advanced data types, and method generators are all supported.

- **Object serving**
  Objects defined in the unified data architecture can be served up as Java, C++, or COM objects, providing compatibility with a variety of object-oriented technologies.

The Unified Data Architecture can dramatically reduce impedance mismatch, but cannot entirely eliminate it. There are some concepts, for example, object methods and relational triggers that cannot be shared automatically. Nevertheless, Caché is a good choice for developers looking to combine object and relational data access.

Conclusions

In recent years, object-oriented programming languages such as Java, C++, and COM have become the dominant technologies for application development. Therefore, developers of database applications need to be able to project data as objects.

However, SQL is by far the dominant technology for data analysis and reporting. Thus, to be useful, any database must be capable of projecting data as relational tables that can be accessed via ODBC and JDBC.

Impedance mismatch – the inherent disconnect between object and relational data models – cannot be avoided, but it can be significantly mitigated by the proper choice of database technology. For new application development, it makes sense to store data in a multidimensional database such as Caché, which, through its Unified Data Architecture, allows data to be concurrently projected as both objects and tables.

For ongoing application development, when existing data is already stored in, say, a relational database, developers should consider converting that data into a multidimensional form. By making a one-time effort to convert their data, they avoid the performance penalties and data model evolution headaches that are endemic to object-relational mapping.

InterSystems
World Headquarters

One Memorial Drive
Cambridge, MA 02142
USA
Phone: +1.617.621.0600
Fax: +1.617.494.1631

InterSystems.com

InterSystems
CACHÉ
Make Applications Faster

Part No: A°B-06-01