Developing Object-Relational Database Applications, Part 2: application implementation

Paul Brown

June 06, 2002

Introduction

This is the second article in a two-part series on developing object-relational database applications. It is derived from a book published by Informix Press entitled Developing Object-Relational Database Applications. The first article, Part 1, covers database analysis and design methodology. This article, Part 2, highlights application implementation.

Initial implementation

At the end of part one, we saw how to build a conceptual model of the problem domain your object-relational database application is being built to support. This conceptual model consists of a semantic data model-and Extended ER or UML diagram-and a set of definitions describing the structure and behavior of each of the domains (or object classes) used within that schema. In this installment, we describe the set of steps to follow that will turn the results of this analysis into working software.

Put simply, the problem is to turn pictures into efficient code. This requires a two-step process. First, you need to use the results of previous analysis to design a correct and complete body of code. Second, you need to evaluate this code to ensure that your technical design and hardware meet the system's functionality, performance, and reliability objectives. Where it falls short, adjust your design or your architecture to meet your goals.

A note on analysis paralysis

Before you go any further, a word of caution. A good rule for software developers is that, while a little analysis goes a long way, a lot of analysis leads nowhere. Trying to build a comprehensive conceptual model is usually an exercise in futility.
In the first place, if you study any subject for long enough, you will discover that there is almost always an infinite amount to learn about it. Knowledge may not be a burden, but most information systems are built under time and material pressures, and this denies you the luxury of leisurely scholarship. Waiting until you know everything before you start coding will result in project over-runs.

In the second place, problem domains change. What was true at one time can be false a week later. Users will vacillate, change their minds, argue amongst themselves, and even lie outright. Measuring your pictures against their reality is hard. This leads to visions and revisions as you struggle to cope with changing requirements.

"Analysis paralysis"--endless study of a problem that is changing far faster than analysts can keep up with-describes the condition some projects fall into.

As Don Knuth famously observed, creating a computer program is like building a model of the world inside a computer. Models are abstractions; they simplify a complex phenomenon by focusing on the important or relevant aspects of it, while ignoring the rest. The trick is ensuring that the parts of the problem you are concentrating on are the right ones, and that your model has adequate explanatory power.

Therefore, it is a good idea to do a minimum of analysis and build the initial version of your system straight away. Then compare the software to reality. Differences between how a computer program and the world behave are easy to detect, and imply that the model needs refinement. Besides, the ultimate measure of successful software is utility. Working systems with flaws can be modified and improved.

Polished pictures are necessary and useful. But ultimately they aren't what help your users.

**Overview of Design Methodology**

The steps to follow to turn your conceptual model into database implementation are as follows:

1. Implement the type system based on the domain/object analysis. Use the user-defined type and user-defined function mechanisms for this.
2. Create an initial, relational design that uses the type system to record facts describing the state of the problem domain. This involves writing the "create table" statements.
3. Normalize the schema to ensure that it is free of ambiguity. This might lead you to adjust your table designs.
4. Populate your schema with production scale data, and implement a set of workload queries against that database to measure its performance. This involves writing DML queries that operate over the database schema.
5. Revisit any of the design decisions made in steps one through four, based on what you learn along the way.

Over the next few pages we cover each of these steps in more detail.
Implementing the Type System

The ORDBMS data model provides a variety of techniques for implementing new object classes. We provide a brief overview of your choices in the figure below:

**Figure 1. Example of UML class diagram with ordinal support**

<table>
<thead>
<tr>
<th>Mechanism Name</th>
<th>Example</th>
<th>Strengths and Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Built-in Types</strong></td>
<td>INTEGER, VARCHAR, DATE etc. These are standardized in the SQL-92 language specification.</td>
<td>Mature and high performance because they are compiled into the ORDBMS, but they are very simple. Good building blocks for other types.</td>
</tr>
<tr>
<td><strong>DISTINCT Types</strong></td>
<td>CREATE DISTINCT TYPE String AS 'VARCHANCE(127)'</td>
<td>Simple to create, and useful when what you want is something very close to another type.</td>
</tr>
<tr>
<td><strong>ROW TYPES</strong></td>
<td>CREATE ROW TYPE Address { Addr_Line_1 String NOT NULL, Addr_Line_2 String NOT NULL, City String NOT NULL, State String, ZipCode PostCode, Country String NOT NULL }</td>
<td>Relatively easy to use means of combining pre-existing types into more complex objects, and enforcing rules about contents. ROW TYPES have several drawbacks that make them a poor choice for types to define columns.</td>
</tr>
<tr>
<td><strong>Java Classes</strong></td>
<td>Combination of Java UDRs with opaque data storage.</td>
<td>More complex to develop, but an excellent choice when you want code that runs both the outside, and inside, the DBMS.</td>
</tr>
<tr>
<td><strong>OPAQUE TYPES</strong></td>
<td>CREATE OPAQUE TYPE GeoPoint { IntegerLength = 16 };</td>
<td>Most complex to develop, but these are the most powerful in terms of performance, scalability, and the range of object sizes that can be supported.</td>
</tr>
</tbody>
</table>

In this section we explore the pros and cons of the various UDT mechanisms in more detail.

**Built-in Datatypes**

Even with ORDBMS products, built-in types are sometimes the best alternative. This is particularly true of the more sophisticated SQL-92 types. For recording DATE and DATETIME values, you can't beat them.

However, the way SQL-92 supports these types has several shortcomings. For example, SQL's approach to managing INTERVAL data is complex and arguably incomplete. And some developers porting applications fairly frequently express the wish that Informix products implemented some aspects of other products' functionality.

Consequently, even when you choose to strictly limit the datatypes you use to the standard set from SQL-92, it can be useful to create user-defined routines that embed new, desirable functionality into the SQL query language.

**DISTINCT Types**

Often, you will find that the structure and behavior of a new type almost exactly mimics an existing type. For example, in our schema we identified a domain we called Quantity. Instances of Quantity are very like the SQL-92 INTEGER type, except that they cannot be negative. It would be ideal to take advantage of the built-in INTEGER's maturity and performance, while at the same time using the database to ensure that no one accidentally enters a negative Quantity value. This can be done using the DISTINCT type, and modifying the CAST function to enforce the correctness rule.

For example, see Figure 2, below:
Figure 2: DISTINCT Type example using CAST to enforce data integrity

```sql
-- This script creates a new DISTINCT type called
-- Quantity using the built-in SQL-92 INTEGER type
-- In order to check that the Quantity instance is
-- a non-negative value, I overload the CAST with my
-- own function.
--
-- CREATE DISTINCT TYPE Quantity AS INTEGER;
--
-- CREATE FUNCTION INTEGER2Quantity ( Arg1 INTEGER )
-- RETURNS Quantity
-- IF ( Arg1 < 0 ) THEN
-- RAISE EXCEPTION -746,0,
-- "Error: Quantity must be non-negative";
-- END IF;
-- RETURN Arg1::LVARCHAR::Quantity;
-- END FUNCTION;
--
-- DROP CAST ( INTEGER AS Quantity );
-- CREATE IMPLICIT CAST
-- ( INTEGER AS Quantity WITH INTEGER2Quantity );
```

There are several important details in this code. First, note how the function's return casts the argument value to the Quantity type, but does so by first turning it into an LVARCHAR. This is necessary in an SPL function because casting directly to the Quantity from the INTEGER would cause the ORDBMS to invoke this same function again, and again, recursively, until memory is exhausted. Implementing this cast function as an external, C UDR would eliminate the need to perform this extra processing step.

Also, note the use of the IMPLICIT casting in this example. By default, DISTINCT types are created with EXPLICIT casts between themselves and their parent. EXPLICIT casting can cause you to clutter your queries with double colons. IMPLICIT casting eliminates the need for these.

Of the domains identified in our 'Boxes-R-Us' conceptual model, the Quantity, the various identifiers, and domains that can hold a small, fixed range of values-like the set of United States used in an Address object-are all well handled using this kind of technique.

**ROW TYPES**

For certain complex types-domains/objects with internal structure—the ROW TYPE mechanisms are ideal. Taking a class diagram, you can implement a corresponding ROW TYPE by creating an element for each of the class's attributes. UML class diagrams cannot represent it directly, but with the ROW TYPE mechanism you can specify which elements of the type are compulsory and which are optional using the NOT NULL constraint.

In the following figure, Figure 3, we present the implementation of a ROW TYPE that models the BirthDay domain described in the previous installment. To make this paper self-contained, we repeat the UML class diagram that we developed earlier. Included in this figure is the code
implementing the behavior for this type: a constructor, and logic to compare two birthdays for equality. We discuss this code in detail in a later section.

Figure 3: ROW TYPE Implementation of Birthday Object Class

```
--
-- This script illustrates how the Birthday Object
-- Class in Figure 11. might be implemented using the ROW
-- TYPE.
--
CREATE ROW TYPE BirthDay (
    MMonth INTEGER NOT NULL,
    DDay INTEGER NOT NULL,
    FromLeapYear BOOLEAN NOT NULL
);
GRANT USAGE ON TYPE BirthDay TO PUBLIC;
--
-- Constructor
--
-- Note that this example uses a combination of built-in
-- SQL-92 expressions { MONTH(), DAY() } and another
-- function whose implementation we do not show here.
--
CREATE FUNCTION BirthDay ( Arg1 date )
RETURNING BirthDay
DEFINE nYear INTEGER;
LET nYear = YEAR(Arg1);
RETURN ROW (MONTH(Arg1),
    DAY(Arg1),
    IsLeapYear(nYear))::BirthDay;
END FUNCTION;
GRANT EXECUTE ON FUNCTION BirthDay ( date ) TO PUBLIC;
--
-- Compare() support function, used as the basis for a
-- set of other functions, like Equal(). Note that the
-- ORDBMS will not use this function for sorting or
-- indexing a column of this datatype.
--
CREATE FUNCTION Compare ( Arg1 BirthDay, Arg2 BirthDay )
RETURNS INTEGER
DEFINE nRetVal INTEGER;
LET nRetVal = -10;
IF ((NOT(Arg1.MMonth = 2 AND Arg2.MMonth = 2)) OR
    ((Arg1.FromLeapYear AND Arg2.FromLeapYear) OR
    (NOT Arg1.FromLeapYear ) AND
    ( NOT Arg2.FromLeapYear ))) THEN
    IF (( Arg1.MMonth < Arg2.MMonth ) OR
    ( Arg1.DDay < Arg2.DDay))) THEN
        LET nRetVal = -1;
    ELIF (( Arg1.MMonth > Arg2.MMonth ) OR
    ( Arg1.DDay > Arg2.DDay))) THEN
        LET nRetVal = 1;
    ELSE
        LET nRetVal = 0;
    END IF;
ELSE
    LET nRetVal = 0;
END IF;
ELIF ( Arg1.FromLeapYear ) THEN
    IF (( Arg1.DDay = 29) AND ( Arg2.DDay = 28 )) THEN
        LET nRetVal = 0;
    ELIF ( Arg1.DDay < Arg2.DDay ) THEN
        LET nRetVal = 1;
    ELSE
        LET nRetVal = -1;
```
ROW TYPEs have significant drawbacks. At this time, you cannot build an index using a ROW TYPE, sort them, or use ROW TYPEs in a union query. This limits ROW TYPE as a UDT mechanism, but they are still useful for implementing application-level datatypes, like the Birthday type we introduce above.

Of the domains/objects identified in our conceptual model, the different kinds of Address are ideal candidates for implementation as ROW TYPEs. The fact that these domains/objects are associated in an inheritance hierarchy is a further indication that using a ROW TYPE is appropriate.

Java Classes
Java®-in-the-server is a relatively recent addition to the Informix Dynamic Server.2000 product. But it is a highly effective mechanism for implementing certain kinds of domains/objects. Java's greatest virtue is its mobility; the same Java class can be deployed within the ORDBMS server, or within a middleware application server, or even within an applet running in a browser. For this reason, Java is the ideal approach to implement anything that needs to be deployed outside the ORDBMS.

For example, consider the PersonName and Phone_Number domain/object. In addition to storing and querying instances of this type in the database, it would be useful if the same objects could present themselves for end-user modification in a client program. The Java language's object-oriented character makes creating such an integrated component possible. And Java's dynamic linking properties allow it to be deployed.

OPAQUE TYPE
OPAQUE TYPEs are the most difficult UDTs to implement. But OPAQUE TYPEs are the most flexible mechanism, and they offer the best performance. For these reasons, most commercial DataBlade™ products make extensive use of OPAQUE TYPEs. As far as the ORDBMS is concerned, the contents of OPAQUE TYPE data are simply arrays of bytes. You use the query language to invoke the type's behaviors by binding function names with column and table names in queries.
In our 'Boxes-R-Us' example, it would be a good idea to implement types like Geo_Point, Mass, and Size as OPAQUE TYPEs. Each of these involves relatively complex logic involving a structure composed out of simple types. Also, the logic implementing support operations for sorting and indexing instances of these objects would be called very frequently, making it necessary for the types to perform well.

**Implementing Behaviors:**

Although you can use any of the procedural languages to implement functions for any kind of user-defined datatype (built-ins, DISTINCT, ROW, and OPAQUE) certain languages are more appropriate for certain type mechanisms. For example, the stored procedure language (SPL) is ideally suited for implementing behaviors for ROW TYPES, because it shares the simplicity of the ROW TYPE mechanism. On the other hand, C is the preferred means of applying logic to OPAQUE types.

Deciding which language to use is a complex problem. The basic rules are as follows:

- If a user-defined function contains a SQL callback (embeds a query within its logic), then the cost associated with running the query is almost always the most computationally expensive aspect of executing the function, regardless of the language used to implement it. This means functions containing SQL callbacks should almost always be created using SPL, because that is by far the simplest technique.
- SPL is the easiest language to use, and represents the fastest way to develop a new function. Java is a more complex language than SPL, so developing logic in Java takes more time. At the same time, the presence of a large library of utilities for Java makes it easier to develop complex extensions. C is the lowest-level programming language, and it is therefore more difficult to use than either Java or SPL.
- Queries that invoke C and Java functions can be parallelized. Queries invoking SPL logic cannot.
- Java code is mobile. A properly implemented Java class can be deployed within the ORDBMS, within a middleware application server, or within an applet running remotely. C and SPL, on the other hand, cannot.

An important objective in developing ORDBMS applications is to minimize the effort involved. This indicates that it is a good idea to re-use components wherever possible, and to use the simplest technique that is practical. Initially, use SPL for new application-specific extensions. As part of your testing effort, you will be able to identify any under-performing types and, if necessary, re-implement them using a mechanism that is more efficient, but also more time-consuming.

**Performance of Alternative UDR Mechanisms**

Assessing the runtime performance of the different mechanisms is a complex problem. In the following figure, Figure 4, we present the result of a simple-minded experiment conducted to evaluate the overhead of invoking a UDR, and the runtime performance of various languages.

We measured the calling overhead of SPL, Java, and C by implementing a small UDR that simply returns its argument value immediately. And to estimate the runtime performance of the various languages, we implemented an algorithm that determines whether or not an integer argument is a
prime number or not. This algorithm simply loops through the integers in ascending order, seeking an integer value that divides the argument value evenly, and stopping when it reaches the square root of the argument value. For extra realism, each UDR includes a redundant memory allocation and string manipulation step.

The idea is that this algorithm will take longer to run--on average--for larger numbers. This is complicated by the fact that the larger the number, the less the likelihood that it is prime. Nevertheless, the experimental results are revealing.

To obtain the measurements we present in the figure below, we exercised these functions over several blocks of numbers where the count of numbers in each block was identical (10,000), but the size of the numbers varied between 1 and 10,000,010,000. The idea is that it takes more computational resources to determine whether or not larger numbers are prime.

Figure 4. Comparing the runtime performance of three UDR mechanisms

The conclusion is that the calling overhead of both C and SPL is significantly lower than Java, and C has by far the best runtime performance. But the superior runtime performance of Java means that once the UDR reaches a certain degree of complexity, Java outperforms SPL. Note that the final value of the "SPL Prime Performance" was 2401 seconds, which we do not show to clarify the information on this chart.

Domains, Objects and UDR/UDT Development

When developing a new object/datatype that is to be embedded into the ORDBMS, it is helpful to keep in mind that you cannot anticipate all of the ways in which it will be used. Perhaps the greatest strength of the original relational DBMS products was the way they allowed users to answer ad hoc questions asked at runtime that were not anticipated at design time. In a similar way, it is usually a good idea to implement as much of the behavior of your new type as you think users might need. That is, focus on implementing each database type one at a time as an autonomous, complete, and self-contained object definition.

For example, consider the Mass object identified as part of the conceptual analysis. In the following figure, Figure 4a, we illustrate how this type might be used.
A Canadian customer wants a box with a capacity of '3 KG'

```
SELECT P.Id, P.Capacity 
FROM Products P 
WHERE P.Capacity > '3 KG' 
ORDER BY P.Capacity ASC;
```

**Figure 5. Sample data and query illustrating use of Mass in SQL**

When implementing the Mass OPAQUE TYPE, it would be a good idea to implement all of the functionality you would associate with a more conventional number, even if your analysis does not indicate that you need them. One way to think about this is to say that Mass is an instance of a *pattern*: which you might call the "number" pattern. Numbers can use all of the mathematical operators and indexing support in addition to the operators you see in this query. By contrast, mathematical operators make no sense for the Dimensions type, because it follows another pattern.

Consider how the new type might be used in other applications, and in situations where your database is deployed in a distributed mode.

**SQL Queries and UDRs**

One way to implement the Mass type would be to use a table to hold each of the various units of mass (kilograms, grams, ounces, pounds) and the conversion ratios to other units. Then the user-defined functions that compare instances of a Mass datatype could use an SQL query over this table. But this approach has multiple weaknesses.

In the first place, queries are very expensive operations, and are ultimately unnecessary in this example. Conversion rates do not change, and new Units are rare. Hard-coding conversion rates as a static data structure within the conversion logic improves the performance and scalability of the extension. Second, using tables leads to the situation where there are tables in the database that have no apparent relationship to anything else.

Therefore, it is a good idea to avoid putting SQL queries into UDRs. And if you do, avoid using 'C' or Java. Run-time costs of a UDR containing SQL are almost all incurred in the query; it does not matter which language you use.
But, on the other hand, a complete ban on using SQL in behavior functions is unreasonable. Some conversion rates do change with time; as is the case with Currency. On a day-to-day basis, the exchange rate of Canadian dollars to U.S. dollars varies. The most efficient way to do this is to create a table to store exchange rates, and to include SELECT queries in the logic that handles the conversions. Although this slows the database down, it is necessary to ensure that query results are correct.

Using DataBlades
Frequently, you will find that some of the datatypes you need to use in your schema are available in a DataBlade. In our example, the Geographic_Point and Document objects are obvious examples of functionality you can buy rather than build. In general, this is a very good idea. When the code in your application is the same code running in thousands of other people's, and when the code was written in the first place by an expert in some narrow technical field, that code will probably have fewer bugs and will outperform anything you could write yourself.

Also, there are many places on the Web to find useful "bladelet" extensions. Typically, these come complete with full source code, making them easy to modify. And they are valuable examples, illustrating more complex aspects of the Server Application Programming Interface (SAPI) and UDT/UDR design. In the figure below, Figure 6, we include a partial list of useful Web sites.

**Figure 6. Web resources for pre-packaged extensions and example code**

<table>
<thead>
<tr>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.inforix.com/">http://www.inforix.com/</a></td>
<td>The Inforix Developer Network contains multiple examples and many instructional references on DataBlade development. It is run by the Inforix DataBlade Engineering group.</td>
</tr>
<tr>
<td><a href="http://www.iug.org/">http://www.iug.org/</a></td>
<td>Informix's International User Group hosts a collection of open-source bladelets, each of which implements something small but rather useful.</td>
</tr>
</tbody>
</table>

Eventually, you will arrive at the point where each of the objects/domains identified in the conceptual analysis phase has a corresponding user-defined type and a set of user-defined functions represented in the database. In addition to the domains in the data model, it is a good idea to take a look at the set of workload queries, and any application-level business processes you have identified. The types you create need never be used in a database table. But they may still be useful in a query, or in a scripted business process.

**Designing the Schema**

Having implemented at least a prototype for each of the low-level domains in the schema, the next step is to use these types to construct a set of tables based on the high-level E-ER model.

Below, we describe the procedure to follow. In Figure 7, we present most of the schema creation script that results from the application of these steps against the conceptual model defined in the previous section.
1. For each entity that does not participate in an inheritance hierarchy, create a corresponding table with the same name, and a set of columns matching the entity's attributes. Label any keys and column constraints. The Customers table illustrates this.

2. For entities in an inheritance hierarchy, create a ROW TYPE matching each entity's structure (you might have done this already), organized appropriately according to the hierarchy. Then, create the corresponding set of tables, using the ROW TYPEs to define their structures.

3. Extend table definitions by adding columns for each foreign key reference. For example, each branch manufactures one product, resulting in a foreign key. To model this, you add a column to the Branches table to hold values of the Products table's primary key columns.

4. For each of the multivalent relationships, create a table consisting of columns for each of the primary keys of participating tables, adding columns for each attribute of the relationship. Relationships like the one between the various kinds of Employees and Branches are handled this way.

Why not create ROW TYPEs for all tables? Once the database schema is fairly stable, this can be a good idea. Entities often correspond to application-level objects which have business process behavior associated with them. But the relationship between ROW TYPEs and tables is a complex one, and the inability to easily modify types makes evolving typed tables difficult. ALTER TABLE allows you to modify columns only if the table was not created with a type. However, for inheritance hierarchies, the use of ROW TYPEs is mandatory.

The result of this transformation is a data definition language script that creates a set of tables and constraints. Readers familiar with RDBMS development will be aware that in practice, creating tables involves additional decisions: how to organize data over disk resources, how to set up permissions and roles on tables, and how to make bulk loading as convenient and efficient as possible by disabling logging and integrity constraints. In this paper, we ignore these important questions to focus on design issues. In general, ORDBMS extensibility does not change how you should approach these questions, and in general, lessons learned with RDBMS databases apply equally to ORDBMS databases.

In the following figure, Figure 7, we present a sub-set of the DDL script produced by the steps outlined above.
Readers should note several of the details in this schema. First, observe how rigidly we adhere to the precepts of **strong typing**. The semantics of the schema in Figure 7 are reasonably self-evident from the types used in the columns. And strong typing serves to highlight potential relationships between tables. For example, because we know that the column Customers.Location is a GeoPoint, and that the Sales.Region column is a Geo_Polygon, you can tell that it is possible to write a query between these tables to infer new information. Or that the Product.Price and the Sales.Quota are related semantically because they share the same datatype.

Second, note the way this schema uses the SERIAL type for a surrogate key. In practice, although the SERIAL and INTEGER types are identical with respect to their physical structure, and although the ORDBMS provides built-in functionality to support queries comparing SERIAL and INTEGER instances, creating identity types requires some gymnastics. Product_Id is a DISTINCT TYPE of INTEGER.
**Keys and Constraints**

First, a note on keys. Speaking strictly, a key is a column (or set of columns) where the key value(s) in any given row will not be found in the same columns in another row. Another way to say this is that no two values in the key column(s) are **equal**. With SQL-92 databases, any of the built-in datatypes can be used to define a key column, because they all possess a built-in equality expression. And for efficiency, a DBMS can build an index for all of the built-in types.

By implication, any object class/datatype that has an Equal() user-defined function can be used to define a key column in a table. For example, suppose the definition of a Product_Num requires something more complicated than an INTEGER. Many businesses use "intelligent" part numbers, where information about the part or product is encoded within the sequence of numbers or letters making up the identifier. Enforcing an integrity constraint over a product number, particularly in circumstances where simple string matches are inadequate, can be extremely useful.

Extensibility impacts the topic of keys in other ways. In our Boxes-R-Us schema, examine the Works_At table in more detail. Of all the tables in this schema, it is the only one without a primary key. This is because its uniqueness is more complex than other tables. Strictly speaking, in order to maintain the integrity of the table, two rows R1 and R2 are in conflict when:

Figure 8: Temporally constrained primary key

```plaintext
{ ( R1.Employee = R2.Employee ) AND
  ( R1.Branch = R2.Branch ) AND
  ( Overlaps ( R1.Duration, R2.Duration ) )
}
```

This kind of constraint cannot be expressed as a conventional PRIMARY KEY, and must be enforced using a TRIGGER.

**Handling COLLECTIONs**

The question of whether to use a COLLECTION, or to create a secondary table instead, can be a difficult one to answer. On the one hand, COLLECTIONs are frequently appealing from a data modeling point of view because they can simplify the schema design and come closer to representing a user-level perspective. On the other hand, COLLECTIONs can cause performance and data integrity problems. Current indexing techniques do not let you quickly find all COLLECTION instances containing a particular value, for example. And ensuring that the value in a COLLECTION complies with some kind of integrity rule is difficult.

In general, use COLLECTIONS when the non-first normal form attribute contains instances of a domain that has a very low cardinality (only a few possible values, enforced as part of the type's definition). For example, in our Boxes-R-Us database, the Products table includes a column that lists the colors in which the product is available. It happens that the range of colors is quite limited, so we can model this domain using an **enumeration** pattern, and enforce its correctness that way.

On the other hand, one might conceivably model the relationship between Sales, Products, and Customers as a non-first normal form attribute of any one of these entities. But that would be a huge mistake. Nested relational schema sounds like a good idea, but experience has shown
that it is fraught with peril. The principle danger is that your underlying schema, which should be
designed with the need to support a variety of end-user perspectives uppermost in mind, can come
to reflect a single perspective. Amending it to accommodate another user's requirements can
become very difficult.

**Handling Inheritance**
Inheritance is another seductive concept because human beings are, by nature, classifiers. We
like to arrange things in terms of their structure and their relationship to other things. For these
reasons, inheritance is a useful modeling tool under many circumstances. But it is not the right way
to model everything.

Large inheritance hierarchies--and just what is "large" varies depending on the kinds of data
involved--can cause a drag on performance, particularly when you write join queries involving
the hierarchy. Query execution time is not the problem. Rather, the complexity of parsing and
optimizing queries involving large inheritance hierarchies requires lots of CPU and memory. Also,
the relative immaturity of the ORDBMS's table maintenance facilities hinders developers from
making extensive use of inheritance.

It is sound software engineering practice to limit your use of any data modeling feature to just
those circumstances where it makes sense. For example, in the Boxes-R-Us schema modeling the
various categories of employees using an inheritance hierarchy is a good use of the technique. By
contrast, modeling each of the various kinds of boxes as a sub-table beneath product would not
be.

**Normalization and ORDBMS Tables**
The word "normalization" chills the heart of many database developers. The concept is typically
explained as an apparently arbitrary set of complex rules and technical jargon. But at its heart,
normalization has a very simple objective: ensure each fact is stored exactly once. To apply
normalization algorithms a database schema is to anneal the initial, naive design in order to
eliminate the potential for certain anomalies to occur.

Normalization is a set of formal techniques that distinguishes database development from
other computer programming disciplines. There are no good technical reasons-like improving
performance, scalability, or efficiency-to normalize a database schema. It is justified purely on the
grounds that information systems must first and foremost be correct. Running twice as fast in the
wrong direction is worse than standing still.

In this section we give a brief overview of normalization concepts. The ORDBMS data model has
little to add to the story many readers will have heard before. This re-capitulation is included to
make the paper self-contained. Because normalization is a large topic, we cover only the basics
here.

**Anomalies**
The purpose of normalization is to try to free the database from the possibility of certain anomalies.
A database schema has an anomaly when a write operation over the schema has unexpected or
unintended consequence. Because an object-relational database is intended to record facts about
objects in the problem domain, these unintended consequences relate to other facts, not directly affected by the original action.

In this section we introduce and illustrate three kinds of anomalies we would like to avoid.

Consider the following table, Figure 9, which is one way to store facts about products and branches, and the relationship between them. In this hypothetical table, the primary key would be the combination of the Product_Id and Branch_Id.

**Figure 9. Hypothetical table to illustrate anomalies**

<table>
<thead>
<tr>
<th>Product_Id</th>
<th>Product_Name</th>
<th>Product_type</th>
<th>Branch_Id</th>
<th>Address</th>
<th>Branch_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>&quot;Shoe&quot;</td>
<td>. .</td>
<td>Sacramento</td>
<td>1 O F.</td>
<td>. .</td>
</tr>
<tr>
<td>101</td>
<td>&quot;Shoe&quot;</td>
<td>. .</td>
<td>Houston</td>
<td>2 Grand.</td>
<td>. .</td>
</tr>
<tr>
<td>102</td>
<td>&quot;Gip&quot;</td>
<td>. .</td>
<td>Miami</td>
<td>8 Del.</td>
<td>. .</td>
</tr>
<tr>
<td>111</td>
<td>&quot;Large Pizza&quot;</td>
<td>. .</td>
<td>San Diego</td>
<td>1 B C.</td>
<td>. .</td>
</tr>
<tr>
<td>211</td>
<td>&quot;Computer&quot;</td>
<td>. .</td>
<td>San Jose</td>
<td>Indus.</td>
<td>. .</td>
</tr>
</tbody>
</table>

1. **INSERT Anomaly**
   Consider what happens in this table if we were to try to record a fact about a new Branch. This would require that we also record a fact about a new Product. But this information might not be known, so the INSERT could not be completed. We call this an INSERT anomaly.

2. **DELETE Anomaly**
   Now, consider what happens if Boxes-R-Us closes the "St Louis" branch. This requires that we delete the corresponding row from this table. But this has the undesirable side effect of removing the fact about the existence of a "Small Pizza" product. The "Small Pizza" is also referenced elsewhere in the database, so removing this information is a serious problem. We call this a DELETE anomaly.

3. **UPDATE Anomaly**
   And finally, consider what might happen if Boxes-R-Us were to alter the design of their "Shoe" box. If the manager in "Sacramento" made the modification, but the manager in "Houston" did not, we would have a problem. Changing one row but not the other leads to an inconsistency in the database. We call this an UPDATE anomaly.

Of course, the whole point of this example is that facts about branches, facts about products, and facts about the relationships between them should not be modeled in this way. The purpose of normalization is to detect these situations when they arise, and then modify the design to eliminate them. Applying the rules of normalization produce a schema like the one we present in Figure 7.

**Normalization Basics**

A central concept in normalization is *functional dependency*. This idea derives from relational theory, and describes the relationship between attributes in a relation/table. For example, given a list of products, if you have a valid Product_Id value you can unambiguously determine other information about the product: name, physical_size, mass, and so on. Note that the reverse is not true. Two different products can have the same mass, although they have distinct identities.
The Boxes-R-Us schema includes other situations that we might choose to label as functional dependencies. For example, the Customers table has DeliveryAddress and Geo_Location columns/attributes. In a sense, addresses determine locations. Given a street address and zip code it is possible to figure out the corresponding latitude and longitude pair. And in fact, there is software available that does exactly this.

Where a functional dependency can be computed with a user-defined function it can be a good idea not to store the computed value. Techniques like functional indices can address performance issues, and using functions in this way saves space. Alternatively, if the function is very expensive to run then storing the value can also make sense.

Normalization algorithms examine the functional dependencies in the set of relations/tables. They use this information to determine when an anomaly exists. And they can even indicate what changes--breaking a table up, moving an attribute from one table to another--would eliminate the problem.

**Objective of Normalization**

Rather than a complete description of normalization algorithms, we instead describe a reasonable goal for normalization. All of the database tables in Figure 7 (with the exception of the Works_At table) are in what is known as Boyce-Codd normal form (BCNF). Formally, a table is in BCNF if, and only if, for every functional dependency, the functional dependency's left-hand value (Product_Id, Customer_Id, or possibly MailAddress) is a key. A more pithy version of the same principle goes, "attribute values can be accessed using the key, the whole key, and nothing but the key."

Many computer-aided software design tools have normalization algorithms built into them. Starting with a set of diagrams that reflect the conceptual model, the tools can generate a DDL at whatever level of normalization is desired. But these algorithms are only as good as the information captured in the model concerning keys and functional dependencies.

**Loading and Testing**

Test early, and test often, is the best possible advice to follow in any kind of software development. Testing helps to identify conceptual errors, design flaws, code bugs, and anything you have overlooked. What is surprising is how often development schedules place testing and validation as the very last task, instead of integrating them into other tasks.

Fortunately, unlike other programming systems where you need to build elaborate harness code in order to test a module, the environment into which a user-defined function is to be deployed--the object-relational DBMS--can also function as a test harness.

**Unit Testing of Extensions**

Thoroughly tested extensions are essential to the smooth operation of the database. And although the engine's debugging facilities are immature, the nature of the ORDBMS makes both correctness and performance testing relatively easy. Following certain design principles for your user-defined types also helps to ensure that the set of behaviors provided for a type are symmetric—for example, that each constructor function has a corresponding print functions.
For scale testing—running the extension many times to check for memory leaks and so on—using the DBMS query language's join capacity can be useful. For example, in the following figure, **Figure 10**, we present the outline of a query to test the GreaterThan(Mass, Mass) user-defined function. This query will execute the UDR about 2000 times.

**Figure 10: Query testing one aspect of the GreaterThan user-defined function**

```sql
SELECT Mass(Q1.Num,U1.Val) AS Less,
      Mass(Q2.Num,U2.Val) AS More
FROM TABLE(SET{1.0,10.0,100.0,1000.0,10000.0,100000.0
  }::SET(DECIMAL(10,2) NOT NULL)) Q1 ( Num ),
TABLE(SET{'G','gram','OUNCE','OZ','Ounce','POUND','LB'
  }::SET(LVARCHAR NOT NULL)) U1 ( Val ),
TABLE(SET{1.0,10.0,100.0,1000.0,10000.0,100000.0
  }::SET(DECIMAL(10,2) NOT NULL)) Q2 ( Num ),
TABLE(SET{'G','gram','OUNCE','OZ','Ounce','POUND','LB'
  }::SET(LVARCHAR NOT NULL)) U2 ( Val )
WHERE Mass(Q1.Num,U1.Val) > Mass(Q2.Num,U2.Val);
```

The idea is that you can use the combinatorial potential of the query language to generate a huge number of individual tests in a single expression. Tests of several user-defined functions can be combined into a single expression, and if you keep an eye on the messages the engine prints to its log file, problems like memory leaks and certain code bugs in the function will be reported.

Negative tests—where you pass arguments into the user-defined functions that make them generate an exception—require a different approach. These require that you use the EXECUTE FUNCTION syntax, and pass into the function argument values that either deliberately provoke an error, or exercise the extremes of the range of values each argument can accept.

**Testing of Workload Queries**

Fairly soon, after you have finalized the schema design, it is an excellent idea to load the database with a set of data that is close to anticipated production volumes. Then, you can exercise the database using queries that characterize the anticipated workload. You can use the same technique introduced in **Figure 11** to create large amounts of data values. For example, the following figure populates the Products table with a million rows of data.

**Figure 11: INSERT as SELECT to populate products table**

```sql
INSERT INTO Products
  ( Name, Dimensions, Capacity, Available_In, Price )
SELECT MakeString( Random(1000), 24 ),
      Random_Dimensions(),
      Random_Mass(),
      Random_Color_Set(),
      Random_Currency()
FROM TABLE(SET{0,1,2,3,4,5,6,7,8,9}) N1 ( Num ),
TABLE(SET{0,1,2,3,4,5,6,7,8,9}) N2 ( Num ),
TABLE(SET{0,1,2,3,4,5,6,7,8,9}) N3 ( Num ),
TABLE(SET{0,1,2,3,4,5,6,7,8,9}) N4 ( Num ),
TABLE(SET{0,1,2,3,4,5,6,7,8,9}) N5 ( Num ),
TABLE(SET{0,1,2,3,4,5,6,7,8,9}) N6 ( Num );
```
MakeString(INTEGER, INTEGER), the first user-defined function in this query, creates a randomly distributed string the length of the second argument. Each of the other functions performs a similar task.

Based on the results of this testing, you can determine what your next task should be. An under-performing workload query might indicate that you should change the implementation of a user-defined function, or create an index, or re-visit decisions about how you have organized data storage. Synthetic data is rarely perfect. But it can provide you with valuable, early indications of problems.

**Note on the organic information system**

In the first installment of this series, we noted that developers make the best use of their object-relational DBMS product by taking a more holistic view of the information system. Object-relational, or extensible, database management systems permit developers to embed considerable portions of the logic of the information system within the data management framework, and they allow developers to combine logic and data using the declarative query language.

But the ORDBMS has additional virtues. Developers can add new functionality, and existing logic can be changed or upgraded without halting the running system. This changes how information systems are managed. Usually, software upgrades are released periodically. This is due to the way conventional programming language technology resolves references between compiled modules as they are linked into an executable. But with the ORDBMS, linking is incremental, and therefore far more flexible.

The effect of this is that information systems built using an ORDBMS as a platform are more organic in their development. Errors and extensions can be accommodated by changing or adding individual modules within a running system. In fact, it can be constructive to view the ORDBMS as more than a data repository relegated to the bottom of the software stack.

Rather, an ORDBMS's abilities to flexibly combine logic and data within a single framework distinguish it from either conventional DBMS or middleware software. They place it into a new category. And what is useful about deploying information systems using these organic or evolutionary principles is the way it helps you cope better with the constantly changing, day-to-day demands of end users.

**Summary and conclusions**

The first installment of this series described an approach to conceptual modeling suitable for use with an object-relational DBMS, like Informix Dynamic Server.2000. The functionality of Informix Dynamic Server.2000 can be thought of as an implementation of the relational model that is closer to the spirit of the original ideas than relational DBMS products built to support SQL-92. Essentially, the Informix Dynamic Server.2000 engine allows you to build a data model to store relevant facts about the important objects in a problem domain.

All of the methodologies and techniques used with relational DBMSs--like entity-relationship diagrams, normalization theory, and even the principles of physical tuning--apply equally in object-
relational DBMSs. But unlike RDBMS products, an ORDBMS provides better data modeling because it lets developers model objects using user-defined types, and user-defined functions. The result of the conceptual analysis is a set of diagrams that represent the conceptual view of the problem domain, and the structure and behavior of the objects within it.

In this second and final installment we saw a set of techniques for turning these diagrams into working code. First, you develop a set of user-defined types and functions that correspond to the set of domains or objects identified in the conceptual analysis phase. Then, you combine these objects into a database schema—a set of tables and constraints where the new types are used in the table’s columns. Normalization theory, which seeks to ensure that a database schema does not contain the possibility of error, can be applied in object-relational databases.

Finally, we noted that object-relational DBMS technology represents something quite new. Perhaps the greatest challenge in software development is responding quickly to end users and, today, customers or consumers. Making these adjustments at Web speed calls for a new approach to server software: one that combines performance, scalability, and reliability with flexibility and an improved ability to model the world. Supporting this kind of organic information system is what object-relational technology is all about.

© Copyright IBM Corporation 2002
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)