Storage best practices

March 2005

DB2® Information Management Software

Comparing XML and relational storage: A best practices guide

Contents

When to choose XML storage or a relational model

While there have been years of research into physical and logical database design in purely relational systems, little definitive work has been done on the influence of XML on the logical and physical database design of unified XML/ relational systems.

The bulk of the influence of XML on logical and physical database design is based on fundamental properties of XML that make it different from the relational model:

- ***XML is self-describing.*** *A given document contains not only the data, but also the necessary metadata. As a result, an XML document can be searched or updated without requiring a static definition of the schema. Relational models, on the other hand, require more static schema definitions. All the rows of a table must have the same schema.*
- ***XML is hierarchical.*** *A given document represents not only base information, but also information about the relationship of data items to each other in the form of the hierarchy. Relational models require all relationship information to be expressed either by primary key or foreign key relationships or by representing that information in other relations.*
- ***XML is sequence-oriented–order is important.*** *Relational models are set-oriented–order is unimportant.*

None of these differences indicate that XML is better or worse than purely relational models. In fact, XML and relational models are complementary solutions. Some data is inherently hierarchical while other data is inherently tabular; some data has more rigid schema while other data has less rigid schema; some data needs to follow a specific order while other data does not.

It is important to answer the question of when to use XML and when to use a purely relational model. While an obvious answer to that question is "use XML when working with XML data," that approach is oversimplified and consequently can be wrong. In many cases, it is advantageous to use an XML data representation even for non-XML data; similarly, there are times when using an XML data representation for XML data is not the correct choice.

Situations in which using an XML representation is not correct, even if the data is in XML form, include the following:

- ***When downstream processing of the data is relational.*** *Even if data originates in XML, if subsequent processing of that data depends on the data being stored in a relational database—for example, when applying relational online analytical processing (OLAP) to the data in a data warehouse—then storing the data as relational data instead of in XML form may be the correct choice.*
- ***When the highest possible performance is required.*** *Despite the fact that XML database processing is relatively fast, some expense is associated with parsing and interpreting that XML. Applications that cannot tolerate even the small incremental expense associated with processing XML data should consider storing the data as relational data instead of XML form.*
- ***When any normalized data components have value outside the XML representation or the data need not be retained in XML form to have value.*** *In many ways, XML is hierarchical and the ability to model parent/child relationships through position in a hierarchy greatly simplifies many data modeling tasks. As a result, a natural inclination is to model hierarchical data using XML. However, XML may not be the optimal choice when the children do not need the parents to provide value. Consider the example of a data mart that needs to perform inventory control for a retail store. While it is useful to have product-line item information stored directly within purchase orders, that line item information contains value outside the purchase orders. Namely, it is not necessary to know who the customers are and*

what they purchased in order to control inventory. Conversely, if there is value in retaining that parent/child relationship—for example, when the inventory control system is expanded to support product placement and hence needs to know what products were purchased at a given time—then retaining the data in XML form may be the better choice.

- ***When the data is naturally tabular.*** *If the data being modeled is naturally tabular, it may be better to represent that information as a relation instead of retaining it in XML form. The storage data model should match, to the greatest extent possible, the highest value and most critical usage model for that data.*

Situations in which an XML representation is beneficial, even if data is not in XML form, include:

- ***When schema is volatile.*** *If the schema of the data changes often, then representing it in relational form may be onerous given the cost and difficulty of changing the associated relational schema. While some forms of schema modification are relatively painless in relational databases, such as adding a new column to a table, other forms are more involved, such as dropping a column or changing the type of a column. Still other forms of schema modification are downright difficult, such as normalizing one table into many or denormalizing many tables into one. Any portions of the schema that are highly volatile can be expressed as a single XML column. The self-describing nature of XML and its use in XQuery makes schema modification of that data much simpler.*
- ***When data is inherently hierarchical in nature.*** *Some data is inherently tabular, while other data is naturally hierarchical. A relational model makes the most sense for inherently tabular data, whereas XML is often the best representation for inherently hierarchical data. For example, a bill of materials explosion can be stored in a relational database and reconstructed using recursive SQL, similar to an organizational chart. However, it is much more natural to represent such inherently hierarchical data in a hierarchical structure such as XML.*

- ***When data represents business objects in which the component parts do not make sense when removed from the context of that business object.*** *If the data represents a business object and the content makes sense or provides value only in the context of that business object, it may be best to represent it in XML form. For example, consider a standard employee and phone number relationship in which one employee can have multiple phone numbers—one for the office, one for fax, one for home and one for mobile. If the most frequent usage pattern is to retrieve phone numbers in the context of the employee, it does not make sense to normalize the data and introduce a table solely to track the multiplicity of phone numbers associated with an employee and then further incur the cost of a join to reunite employees with their phone numbers. A better choice may be to keep those phone numbers in the employee relation and represent them using XML.*
- ***When applications have sparse attributes.*** *Similarly, some applications have a large number of possible attributes, most of which are sparse or absent for any given data value. A classic example is a merchant catalog—the number of different attributes to track for a given catalog item is huge, including size, color, weight, material, style, weave, power requirements, thermal requirements, fuel requirements and a nearly endless list of other considerations. For any given object, only a subset of these attributes is relevant—the weave of a sweater makes sense but the weave of a lawn mower is nonsensical. Conversely, the kind of fuel is important for a lawn mower but not for a rolltop desk.*

  The typical relational approach to solving this problem is a table of arbitrary attribute/value pairs associated with the parent to describe the object. The cost of joining with this table to re-create the characteristics of the object can be high, as can the complexity of applications that search for objects with a certain complex expression of properties (for example, a color of red, a weave of herringbone or a material that does not include wool) Representing such descriptive attribute information as XML data allows for more natural representation as well as less complex and expensive searching.
- ***When low-volume data is highly structured.*** *In many applications, structured information is critical to the application but it exists in very small quantities. While that information can be represented in a normal relation, this approach can lead to massive relational schemas. For example, a complex star-schema topology with one table per dimension can have hundreds or even thousands of dimensions.*

Many of these dimensions have relatively low cardinality (that is, they contain few rows). For example, a common dimension is the "region" in which an object was sold. Most companies have fewer than 100 regions, and many have fewer than 20. While a heavyweight table can be used for each dimension, this approach results in many small tables that need to be managed. An alternative is to represent many dimensions using a single table with an XML column, encoding data as XML and then creating views over that data. This approach enables data to be extracted for a dimension from within the XML. Instead of using many little tables, a single larger table with multiple views—one per dimension—is used. This can dramatically reduce the number of managed objects in a database, and thus can reduce the cost of ownership.

Design trade-offs

Several trade-offs are enabled by the fundamental differences between XML and relational schemas.

The principal trade-off is flexibility versus performance. As a self-describing data structure, XML allows diverse forms of data to be stored in a single document or row without sacrificing the ability to search or aggregate portions of that data. However, that flexibility results in greater CPU run time and increased I/O cost to interpret data.

The relational model, given its more rigid schema, requires significantly less interpretation and, as such, will generally perform better. However, in some cases XML will improve performance over relational models precisely because of its flexibility. Relational models often require data normalization to get data values to fit well into the model (for example, when splitting logically interconnected data between tables). Whenever data is normalized, there is a strong probability that joins will be required to reform that data, and those joins may cost more than storing the data as XML and retrieving it without requiring a join.

What is a unified XML/relational database?

Several use cases predicate the use of the XML data model; unfortunately, most businesses cannot sacrifice performance, so they tend to force XML data into a large object (LOB) or normalize (shred) and store the XML data within the relational database. The ideal solution is a database that can combine the flexibility of the XML model with the performance of the relational model; that is, support for an XML data type within a relational database. This XML data type would fully support the XML model, both logically and physically, yet allow the data to be physically managed by a single database. The following cases illustrate the use of unified XML/relational databases.

Case 1a: Data has inherent hierarchical relationships

If data values of the child are not logically independent of the parent, XML may best represent the information instead of normalized tables. For example, line items in a purchase order are good to represent in an XML document because they are not logically independent of the parent. A stand-alone line item has little inherent meaning or value—the value comes from understanding who has purchased the item or what other items were purchased with it. As a result, representing line items and purchase orders in different tables does not add value and may add cost—both performance costs to join the purchase order and line item tables and operational costs to manage and maintain two different tables instead of one.

On the other hand, part descriptions based on part numbers may not be best represented in that same XML purchase order document. It may be better to normalize part descriptions into another table, because the relationship between the part numbers and the part descriptions is logically independent of the purchase orders or line items in which the part numbers are used.

In addition, representations of the part descriptions in different languages may be better rendered as an XML document because the different descriptions have little meaning outside the scope of the part.

Consider a relational schema as shown in Figure 1:

```
Create table PO (PO_NUM int not null primary key, CUST_NUM int);

Create table LineItem (PO_NUM int, PART_NO int, QUANTITY int);

Create table Parts (PART_NO int not null primary key, PRICE
  decimal(10,2), TAXABLE char (1));

Create table PartsDesc (PART_NO int, LANGUAGE int, Part_Desc
  char(25));
```

Table PO

PO_NUM	CUST_NUM
987564331	A6789
….	…

Table Parts

PART_NO	PRICE	TAXABLE
A54	2.00	Y
985	115.21	Y
…	…	…

Table LineItem

PO_NUM	PART_NO	QUANTITY
987564331	A54	12
987564331	985	1
…	…	…

Table PartsDesc

PART_NO	LANGUAGE	PART_DESC
A54	1	Pencils
A54	2	lápiz
…	…	……

Figure 1: A four-way join of all tables is required to retrieve line item details

In this example, the table PO must be joined with the LineItem table to get all the parts, which must then be joined with the Parts and the PartsDesc tables to get the price and part description, respectively.

A better alternative using XML as a data type may be:

```
Create table PO (PO_NUM int not null primary key, CUST_NUM int,
  PARTS_ORDERED xml);

Create table Parts (PART_NO int not null primary key, PRICE
  decimal(10,2), TAXABLE char (1), PART_DESC xml);
```

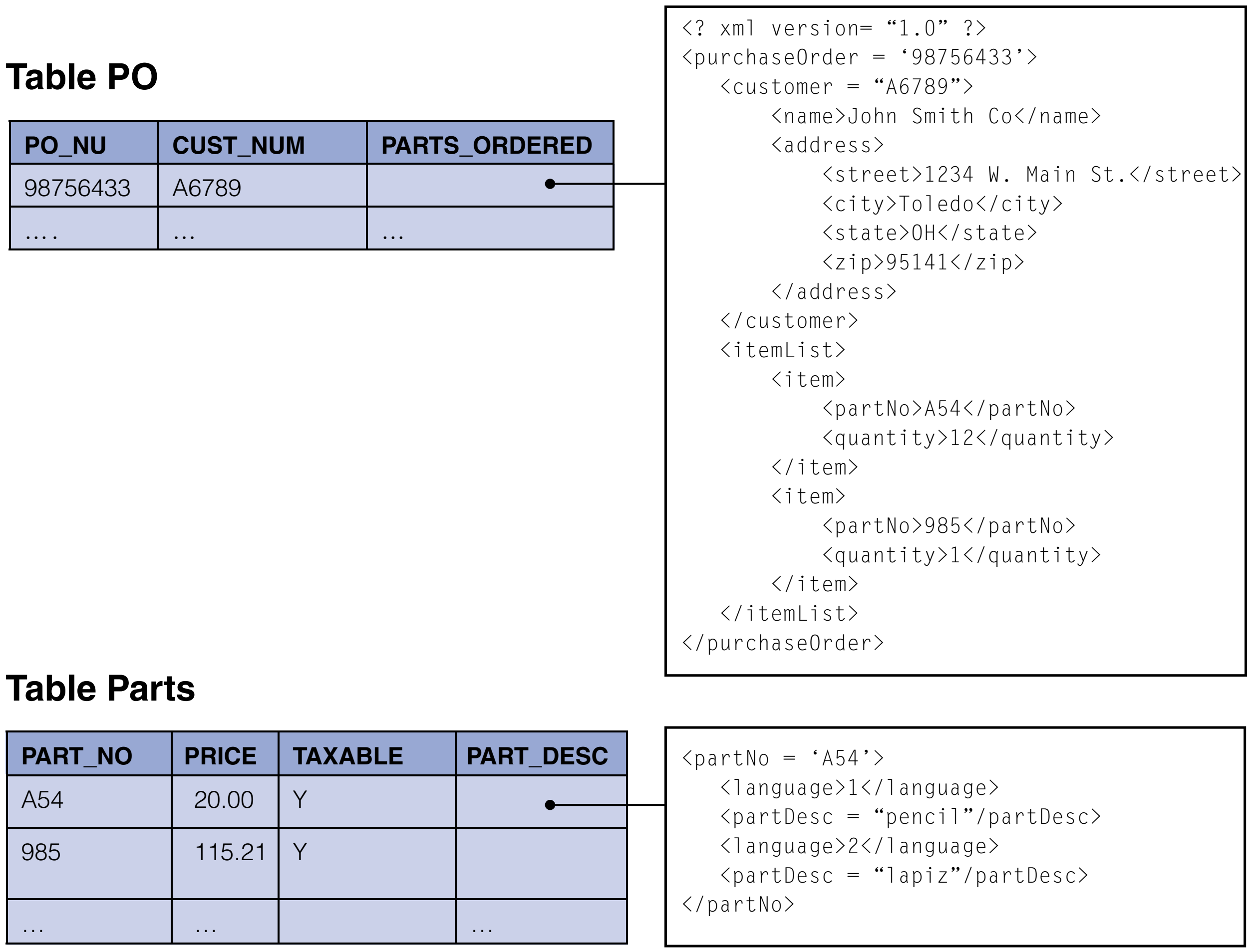

Table PO

PO_NU	CUST_NUM	PARTS_ORDERED
98756433	A6789	●
….	…	…

```
<? xml version= "1.0" ?>
<purchaseOrder = '98756433'>
   <customer = "A6789">
       <name>John Smith Co</name>
       <address>
           <street>1234 W. Main St.</street>
           <city>Toledo</city>
           <state>OH</state>
           <zip>95141</zip>
       </address>
   </customer>
   <itemList>
       <item>
           <partNo>A54</partNo>
           <quantity>12</quantity>
       </item>
       <item>
           <partNo>985</partNo>
           <quantity>1</quantity>
       </item>
   </itemList>
</purchaseOrder>
```

Table Parts

PART_NO	PRICE	TAXABLE	PART_DESC
A54	20.00	Y	●
985	115.21	Y	
…	…		…

```
<partNo = 'A54'>
   <language>1</language>
   <partDesc = "pencil"/partDesc>
   <language>2</language>
   <partDesc = "lapiz"/partDesc>
</partNo>
```

Figure 2: By storing the purchase order and parts description as XML, a four-way join is reduced to a two-way join to retrieve all information necessary to re-create the purchase order

In this case, when a PO_NUM is given, all the parts are directly available; they need to be joined only with Parts to get the price and description, reducing a four-table join to a two-table join with little loss of generality.

Using XML as a data type is most desirable if access to the children is usually through the parent. In Figure 2, the reference to the part description for a given language comes only after finding the part itself—a clear parent/child relationship is significant not only in terms of the logical data model, but also in terms of the access patterns.

The result is a clean logical model that is easier to understand. The XML model contains fewer artifacts than the relational model, so it is easier to manage, requires fewer join operations and often performs better.

Case 1b: Data has multiple inherent hierarchical relationships
Multiple containment relationships increase the probability for considering XML rather than normalization.

As the number of containment relationships increases, so does the cost of normalization. That is because the cost of performing the necessary joins to put the information into the correct context increases. A single join is enough for a single-level containment relationship, with one join added per level of containment. If that same containment is represented in an XML document, no joins are necessary. This can have significant performance impact as well as affect the understandability of both the data relationships and application logic.

For a relational schema like the following:

```
Create table Employee (EMP_NUM int not null primary key, EMP_NAME
  char(25));

Create table Dependent (EMP_NUM int not null, DEPENDENT_NUM int
  not null, DEPENDENT_NAME char(25));

Create table DependentSchool (DEPENDENT_NUM int not null,
  SCHOOL_NUM int not null, START_DATE date, STOP_DATE date);
```

A simpler alternative using XML as a data type may be:

```
Create table Employee (EMP_NUM int not null primary key, EMP_NAME
  char(25), DEPENDENT_INFO xml);
```

A better example is a bill-of-materials case in which the relational schema may be very simple:

```
Create table Part (PART_NO int not null primary key, PARENT_PART_
  NO int, PART_NAME char (25), DESC char(25));
```

However, the cost of re-creating the list of parts within a part is an expensive self-join. A better alternative using XML as a data type may be:

```
Create table PartAssembly (ASSEMBLY_NO int not null primary key,
  DESC xml);
```

The XML approach does not simplify the schema, but it turns a recursive self-join to get the full description of an assembly into a single primary key singleton fetch.

Case 2: Data has containment relationships
If high performance is required and data redundancy is tolerable, it may be best to use XML to avoid join processing. In other words, denormalization—but denormalization into XML data instead of relational data.

In a traditional database system, this technique is limited by the number of children in a given row, because there must be a column in the parent per row per column in the child. Also, the relational approach has practical limits. Those limits do not exist when using XML, because all the children can appear to be a single column to the relational engine.

Consider a relational schema such as:

```
Create table PO (PO_NUM int not null primary key, CUST_NAME
  varchar(25));

Create table LineItem (PO_NUM int, PART_NUM int, QUANTITY int);

Create table Item (PART_NUM int not null primary key, PART_NAME
  varchar(25));
```

Because a PO has no upper limit on the number of items, these relational schemas are typically hard to denormalize and represent in a relational model without massive data redundancy. As a result, any query to reconstruct the PO requires at least a two-way join, and often a three-way join or more.

If high amounts of data redundancy can be tolerated, relational modeling can duplicate the PO information in a single table:

```
Create table PO (
  PO_NUM        int not null,
  CUST_NAME     varchar(25),
  PART_NUM      int,
  QUANTITY      int,
  PART_NAME     varchar(25) );
```

Tables such as these have a large amount of redundant data, particularly as the number of columns in the PO table increases—for example, to represent customer address, shipping address, order arrival date and order fulfillment date. However, this approach reduces a three-way join to an index range scan on PO_NUM.

With XML, denormalization with reduced redundancy becomes a trivial task:

```
Create table PO (PO_NUM int not null primary key, CUST_NAME
  varchar(25), ITEMS xml);
```

Here, a unified approach is superior to the purely relational approach because it avoids data duplication, eliminates join processing and provides the entire PO using primary key–based selection.

Case 3: Data has sparse attributes or a large number of attributes

For data that includes a very large list of attributes, many that are typically null (that is, sparse attributes), XML may be a better alternative than a long list of null columns or a normalized table.

Two typical approaches are taken with non-XML relational systems in this case. The first approach is simply to represent each attribute as a column. For example, assume a table that will record employee wage information. Hourly workers have an hourly wage, salaried workers have a salary, executives have a salary and bonus and salesmen have a salary and commission.

```
Create table EmployeeWage (EMP_NO int not null primary key, BASE_
  SALARY decimal(10,2), HOURLY_WAGE decimal(5,2), COMMISSION
  decimal(5,2), BONUS decimal(10,2));
```

Most columns are null for any given employee in this scenario. There are also practical limits for the number of attributes that can be tracked. For large numbers of attributes, normalization is typically used—either with a specific set of tables or generic tables. Given the preceding employee example, multiple tables such as the following could be created:

```
Create table ExecWage (EMP_NO int not null primary key, SALARY
  decimal(10,2), BONUS decimal(10,2));

Create table EmpWage (EMP_NO int not null primary key, HOURLY_
  WAGE decimal(5,2));

Create table SalesWage (EMP_NO int not null primary key, SALARY
  decimal(10,2), COMMISION decimal(5,2));
```

Alternatively, a generic table that can assign a given employee multiple entries could be created as follows:

```
Create table WageInfo (EMP_NO int not null, WAGE_TYPE char(1) not
  null, WAGE_AMOUNT decimal(10,2) not null);
```

Executives in this case would have two entries: one with a WAGE_TYPE of "S" representing base salary and another with a WAGE_TYPE of "B" representing a bonus.

For such sparse-attribute (or numerous-attribute) cases, a corresponding table with an XML column that could represent all the data would appear as:

```
Create table EmployeeWage (EMP_NO int not null primary key, WAGE_
  INFO xml);
```

Case 4: Schema evolution

If the schema is evolving fairly quickly, consider XML for the volatile portions of that schema. Because XML is self-describing, a single column of XML type can hold vastly different forms of data or different versions of the same type of data. If the application is aware of the different versions, the physical database schema need not change at all and no data migration is necessary.

For example, consider a table originally created as:

```
Create table Person (SSN decimal(9,0) not null primary key, NAME
  varchar(25), PHONE decimal(10,0));
```

To record the person's birthday, adding an "alter table add column" command is relatively easy. However, some changes are considerably more difficult. For example, to record multiple phone numbers, the phone column can be normalized and moved to another table—perhaps leaving a null phone column in the person table or performing data migration to remove it. Alternatively, more columns can be added to hold the different phone numbers. An equally difficult change would be to represent the name as first name, middle initial and last name. The problem becomes even more severe when considering that different cultures do not necessarily place the surname last. The culture then needs to be recorded in addition to the name to allow correct interpretation of the name.

A single XML column (assuming applications can handle back-level versions of data) can solve the preceding cases:

```
Create table Person (SSN decimal(9,0) not null primary key,
  DETAILS xml);
```

Alternatively, multiple XML columns can be used:

```
Create table Person (SSN decimal(9,0) not null primary key, NAME
  xml, PHONE xml);
```

Case 5: Highly variable or multiple schema

If an application requires that data be held and processed from multiple schema, XML may be a good alternative, because it is self-describing and can hold many forms of data in a single column.

For example, consider the "to do" queue of a call center employee. To represent all the different forms of data needed by employees to return a call for a trouble ticket is a daunting task—including, for example, previous call records, scanned images of forms, a transactional history of payments and basic customer information. A purely relational schema to hold all that information can be very complex—with correspondingly complex applications necessary to interpret that data and translate it into something usable by the employee. Alternatively, representation of the necessary data can be simplified through the use of binary large object (BLOB) columns to hold the data. However, this approach either renders the data unsearchable or requires complex application logic in the form of user-defined functions to interpret the data.

The same schema in XML can be represented with a table such as:

```
Create table WorkQueue (EMP_NO int not null primary key, DATA_
  ABOUT_CALL xml);
```

One of the principal advantages of XML is the ability to store different artifact types in a single column, thus eliminating the need for numerous tables. A reduction in the number of tables results in a significant reduction in overall management cost because there are fewer objects to administer.

Customer usage scenarios

While the preceding guidelines are helpful, the question of real-world functionality must be addressed. Do cases in which XML is preferable to relational models exist in the marketplace? Following are brief descriptions of four actual customer usage scenarios.

Scenario 1: Persistence and search of XML messages

A large financial services firm replaced its custom electronic data interchange (EDI) approach for customer interactions with an architecture that uses digitally signed Web services. The firm's goal was both to reduce the cost that its new customers had to absorb to integrate with the existing EDI environment, and to make it easier, cheaper and faster to introduce or change services using the Web-based service approach. The firm's services include operations such as buying and selling securities, transferring money or securities between accounts, opening and closing accounts and inquiring about the state of any operation.

While data about the accounts and securities is maintained in a more traditional relational system, the financial services firm was interested in persisting, searching and analyzing the XML Web services interactions to monitor business processes, perform audits and enhance non-repudiation capabilities.

Scenario 2: Call center

A large insurance company wanted to re-architect its call center application. While reviewing system requirements, the company noted that it needed the capability to handle many different kinds of documents. In addition, it needed to handle many different versions of the same form because some claims take longer to resolve than others and forms may change after submission but before resolution. The company also needed to control the order of work in the call center queues.

The insurance company decided to represent the call center queue as an ordered collection of documents, with each document representing a discrete piece of work for the representative to execute that day. In addition, each document referenced one or more documents associated with a particular work item. Each document was in XML form and most documents contained associated XML schema.

Scenario 3: Sparse-attribute/highly volatile schema
A large travel firm tracked most of its data in a highly structured relational table while tracking customer profiles as relational BLOBs because the profiles are highly volatile. For example, each time a new diet becomes popular, customers request food of that type—more often than not, those fad diets fade away over time. These BLOBs also have very sparse attributes—that is, most customers do not have every variation of frequent flier number, frequent driver number or sleep number; most do not have special medication conditions to note; and many do not even have preferred travel times. Because the BLOB representation was fragile, the travel firm elected to use XML instances to contain the profiles instead.

Scenario 4: E-forms processing
A firm providing XML-based e-forms processing to large government organizations required a high-performance, high-fidelity form repository to preserve digital signatures and high scalability to store and search those forms.

Summary

A true native XML data store is more than merely a data store that exposes XML to its clients—it must represent the XML throughout the entire data engine stack from client to disk and back out again.

While XML storage may seem best for XML data, and relational storage best for relational data, in many cases this does not hold true. At times, relational storage proves best for XML data and XML storage proves best for tabular data.

A unified XML/relational database system offers compelling value to its users—namely, the ability to mix XML and relational forms in the same repository, allowing users to select the most effective data model for storing the data. Unified systems do not mandate that all data be represented as relational data, nor do they require that all data be in XML—unified systems offer a choice.

GC34-2497-00