XML Matters: Putting XML in context with hierarchical, relational, and object-oriented models

David Mertz, Ph.D.

April 01, 2001

On the way to making a point about how XML is best suited to work with databases, David Mertz discusses how XML fits with hierarchical, relational, and object-oriented data modeling paradigms.

View more content in this series

XML is an extremely versatile data transport format, but despite high hopes for it, XML is mediocre to poor as a data storage and access format. It is not nearly time to throw away your (SQL) relational databases that are tuned to quickly and reliably query complex data. So just what is the relationship between XML and the relational data model? And more specifically, what's a good design approach for projects that utilize both XML and relational databases -- with data transitions between the two? This column discusses how abstract theories of data models, as conceptualized by computer scientists, help us develop specific multirepresentational data flows. Future columns will look at specific code and tools to aid the transitions; this column addresses the design considerations.

Data models

Let me bracket XML for a few moments by talking about abstract data models. A lot of work goes into the details of a good database design at the level of specific project requirements. I'm interested in something more general than requirements: I'm interested in paradigms of data modeling.

In broad themes, database management systems (DBMSs) have historically been of three types: hierarchical, relational, and object-oriented. Generally speaking, hierarchical databases arose in the 1960s, on the mainframe data processing technology of the time. Network databases are similar to hierarchical ones, but they allow multiple parent/child relations. In the 1970s, the rigorous mathematical work of E. F. Codd and others created the now-ubiquitous relational database model. In the 1980s -- largely because of the growing popularity of object-oriented programming -- object databases gained a measure of popularity, especially to model so-called rich data, such as multimedia formats.

Currently, relational database management systems (RDMSs) continue to be the dominant data-storage technique for large-scale systems. Hierarchical and object databases fill niche
requirements. However, for many years, many popular DBMSs have been hybrid object-relational. So, in practice, some borders between data-model paradigms have blurred.

The hierarchical model

In a hierarchical database (HDBMS), you begin with a strictly defined tree of data nodes. Each node can contain some identifying data, plus a set of subnodes of a specific child type. The number of subnodes can vary between sibling nodes at the same level, but the type of all "cousins" is identical. Figure 1 illustrates the relationships.

Figure 1. A hypothetical hierarchical database model

![Hierarchical Database Model Diagram]

In hierarchical databases, data access is utterly predictable in structure; therefore both retrieval and updates can be highly optimized by a DBMS. For example, in the pictured model, you might determine a particular book's publisher, in a pseudo query syntax, with:

```
Programming/C.J.Date/An Introduction to Database Systems/Publisher?
```

Any query such as the one in Listing 1 has a precise path to the specified datum, which a DBMS can quickly determine in balanced trees and byte-offset codings. You'd form a query about the publisher of any book in exactly the same (unique) form, just with different category, author, and so on. To continue with the hypothetical DBMS, you could write a more general procedural query in something like the (Python-like) pseudo-code in Listing 2.

```
for book in get_children("Programming/C.J.Date"):  
    print book.field("Title"), book.field("Publisher")
```

By this point, XML programmers probably have noticed that the pseudo-query syntax looks a lot like XPath (the XML Path Language), and the procedural pseudo-code looks a fair amount like DOM (Document Object Model). Keep that in mind as I take you through some more models.
The relational model

A relational database consists of a set of tables, where each table consists a fixed collection of columns (also called fields). An indefinite number of rows (or records) occurs within each table. However, each row must have a unique primary key, which is a sort of name for that particular bundle of data. Figure 2 illustrates relational database structure (covering roughly the same data as the hierarchical example):

Figure 2. A hypothetical relational database model

As well as having primary keys, tables typically have some secondary keys. The secondary keys correspond with primary keys in other tables. For example, in Figure 2, the BOOKS table has secondary keys AuthorID and PubID. These, in turn, serve as primary keys for the AUTHORS and PUBLISHERS tables. The idea here is that every BOOKS row has a distinct ISBN value, each AUTHORS a unique AuthorID, and each PUBLISHERS a unique PubID.

As a constraint on the relation between tables, you can state, for example, that for a row to exist in BOOKS, there must exist a row in PUBLISHERS with the PubID you want to use in BOOKS. If one publisher can "have" multiple books in this way, it's called a one-to-many relation. On the other hand, if one author can have multiple books, and one book can also have multiple authors, it's called a many-to-many relation. To round things out, you can also define one-to-one relations, where one primary key must match exactly one secondary key. It is the job of RDBMSs to enforce just these types of rules.

In relational databases, the design of tables can become quite elaborate, involving subtle decisions. The main concern in design is for the proper normalization of tables. The goal in normalization -- to condense the first through fifth "form" of normalization -- is to remove all redundancy in the way data is stored. Each non-key datum should live in only one place. This goal is accomplished almost automatically in hierarchical databases, but takes work in relational ones. For example, the example pictured in Figure 2 probably has a normalization problem. If books can have multiple authors, where might the database store a second author? The only real option is to create an extra row in BOOKS. But if you do that, you need to repeat an identical PubID, Date,
and Title just to mention a second author. Not only does this require extra storage space, it risks introducing errors if the Titles do not quite match up between the rows. To address this, you would need to rethink the design, and it would probably involve creating some more tables and relations.

Compared with the hierarchical model, the relational model is quite complicated. But with the complication comes a huge increase in power. You can ask essentially any question you want of an RDBMS, but for an HDBMS you can only ask the questions designed into the system. For example, suppose you wondered what authors were born later than 1970. In an HDBMS like the one illustrated above, the only way to discover this would be an extremely costly search of every book leaf node. With an RDBMS you use the straightforward SQL query, as shown in Listing 3.

```
SELECT AuthorName FROM AUTHORS WHERE AuthorBDay > 1970
```

For a more complex question, you have to join multiple tables, but normalization allows you to do that in complex ways. For example, the above authors who publish with Random House might be queried as:

```
SELECT AuthorName
FROM AUTHORS a, BOOKS b, PUBLISHERS p
WHERE AuthorBDay > 1970
AND a.AuthorID = b.AuthorID
AND p.Publisher = "Random House"
```

The query in Listing 4 states several relations you'd want to hold. Think of this as a filter on the tables, each narrowing the search. The RDBMS can implement these internally however it wants, but imagine the following steps (in the reverse order of the specified query; but query conditions can occur in any order):

1. Narrow PUBLISHERS to only "Random House" (PubID 03-4472822)
2. Only consider BOOKS with matching PubID
3. Grab the list of AuthorID's from these considered BOOKS rows
4. Of the AUTHORS rows with considered AuthorID's, determine how many have the right AuthorBDay

The problem with a query like Listing 4 is that it requires a number of steps, some of which can be resource intensive. Even the things that are easy in an HDBMS are likely to be relatively hard in an RDBMS. However, the things that are extremely hard in HDBMS (all but a very limited few) are only moderately hard in an RDBMS.

**The object database model**

Object databases (ODBMSs) in some ways go back to the hierarchical model. Objects in an ODBMS -- much like objects in an object-oriented programming language -- are bundles of data and behaviors. In this sense, objects are similar to branch nodes of an HDBMS, which likewise contain a bundle of child nodes.

There are two unique features of object databases:
• Objects can be heterogeneous, and each contain a different collection of "owned" data
• Objects can contain some inherent "intelligence"

Each of those features merits brief elaboration. As with the other models, look at a diagram (Figure 3) to start with.

**Figure 3. A model of a hypothetical object database**

Heterogeneous objects in an ODBMS allow each bundle of data to contain just what it needs. To help imagine this, extend the example database: now it's not simply library book records. Figure 3 depicts rather an actual online library whose content is delivered out of the database. Different media -- sound recordings, e-texts, movies, and more -- require different descriptive information (and contain different content bitstreams). A known ObjectID points to each object, but the object does not have a rigidly uniform set of child nodes as in an HDBMS.

Because objects in an ODBMS can contain a variety of attributes and data, querying objects is often performed through a set of methods. Each object implements these methods in a way that is appropriate for itself. In the example given in Figure 3, two methods might be "summarize" and "transport." A summary of a book might be its abstract, while the summary of a movie might be a trailer. Each object has the necessary intelligence to know which is a relevant way of summarizing itself. Another way of thinking of an object's "intelligence" is in terms of the metadata that it carries.

The Object Database Management Group (ODMG) has proposed a standard query language for ODBMSs called OQL (see Related topics).

**So where does XML fit?**

As readers already familiar with XML will have pieced together, XML is something of a hybrid. XML is probably most similar to object databases in data modeling, inasmuch as it also consists of nodes, and nodes can contain heterogeneous data. On the other hand, the degree of heterogeneity of nodes depends a lot on the particular DTDs or schemas used to define the structure of an XML document. In something like XHTML or DocBook, it's only a slight exaggeration to say that almost any element can appear almost anywhere. But in DTDs oriented more to data records, an XML document could be as rigid as a hierarchical database. As a transport format XML is rich enough to represent either objects or hierarchies fully, given only the right DTD/schema.
XML is less natural in representing relational databases. But I should be precise about what is "less natural" here. XML is certainly able to adequately represent anything that comes out of an RDBMS. You can represent each table directly, albeit far more verbosely than do actual RDBMSs. For example, you might propose the DTD in Listing 5 to represent the BOOKS table in the example database.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT BOOKS (BOOK*)>
<!ELEMENT BOOK (ISBN,AuthorID,PubID,Date,Title)>
<!ELEMENT ISBN (#PCDATA)>
<!ELEMENT AuthorID (#PCDATA)>
<!ELEMENT PubID (#PCDATA)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT Title (#PCDATA)>
```

You could use schemas for richer typing, but the point is that there is no difficulty in representing a particular RDBMS table as XML.

Similarly, you can represent any specific join you might perform (as in the SQL examples in Listing 3 and Listing 4) in XML equally easily. In practice, representing a query result is the greatest and most common use of XML for RDBMSs. A particular contact, or requestor, usually does not need an entire data set but simply some particular filtered and structured part of it. The GROUP BY and SORT clauses in SQL allow for more structuring than the examples in this column have demonstrated, but XML node hierarchies can represent their results as well.

The problem for many XML-everywhere (and XML-only) aspirations is that at the core of an RDBMS are its relations -- in particular, the set of constraints that exists between tables. Enforcing the constraints is what makes RDBMSs so useful and powerful. While it would surely be possible to represent a constraint set in XML for purposes of communicating it, XML has no inherent mechanism for enforcing constraints of this sort (DTDs and schemas are constraints of a different, more limited sort). Without constraints, you just have data, not a data model (to slightly oversimplify matters).

Some XML proponents advocate adding RDBMS-type constraints into XML; others suggest building XML into RDBMSs in some deep way. I believe that these are extremely bad ideas that arise mostly out of a "buzzword compliance" style of thinking. Major RDBMS vendors have spent many years of effort in getting relational matters right, and especially right in a way that maximizes performance. You cannot just quickly tack on a set of robust and reliable relational constraints to the representation in XML that, really, is closer to a different modeling paradigm. Moreover, the verbosity and formatting looseness of XML are, at heart, quite opposite to the strategies RDBMSs use to maximize performance (and, to a lesser extent, reliability), such as fixed record lengths and compact storage formats. In other words, go ahead and be excited by XML's promise of a universal data transport mechanism, but keep your backend data on something designed for it, like DB2 or Oracle (or on Postgres or MySQL for smaller-scale systems).
Related topics

- The Object Database Management Group (ODMG) home page is another source of information on use of XML as a data-transport format.
- Find other articles in David Mertz’s XML Matters column.