XML and Databases

Ronald Bourret
Independent consultant**,** Felton, CA

18 Woodwardia Ave.
Felton, CA 95018
USA

Spring 2005

1. Introduction

This paper gives a high-level overview of how to use XML with databases. It describes how the differences between data-centric and document-centric documents affect their usage with databases, how XML is commonly used with relational databases, and what native XML databases are and when to use them.

NOTE: Although the information discussed in this paper is (mostly) up-to-date, the idea that the world of XML and databases can be seen through the data-centric/document-centric divide is somewhat dated. At the time this paper was originally written (1999), it was a convenient metaphor for introducing native XML databases, which were then not widely understood, even in the database community. However, it was always somewhat unrealistic, as many XML documents are not strictly data-centric or document-centric, but somewhere in between. So while the data-centric/document-centric divide is a convenient starting point, it is better to understand the differences between XML-enabled databases and native XML databases and to choose the appropriate database based on your processing needs. For a more modern look at the difference between XML-enabled and native XML databases, see chapter 1 of XML for DB2 Information Integration.

2. Is XML a Database?

Before we start talking about XML and databases, we need to answer a question that occurs to many people: "Is XML a database?"

An XML document is a database only in the strictest sense of the term. That is, it is a collection of data. In many ways, this makes it no different from any other file -- after all, all files contain data of some sort. As a "database" format, XML has some advantages. For example, it is self-describing (the markup describes the structure and type names of the data, although not the semantics), it is portable (Unicode), and it can describe data in tree or graph structures. It also has some disadvantages. For example, it is verbose and access to the data is slow due to parsing and text conversion.

A more useful question to ask is whether XML and its surrounding technologies constitute a "database" in the looser sense of the term -- that is, a database management system (DBMS). The answer to this question is, "Sort of." On the plus side, XML provides many of the things found in databases: storage (XML documents), schemas (DTDs, XML Schemas, RELAX NG, and so on), query languages (XQuery, XPath, XQL, XML-QL, QUILT, etc.), programming interfaces (SAX, DOM, JDOM), and so on. On the minus side, it lacks many of the things found in real databases: efficient storage, indexes, security, transactions and data integrity, multi-user access, triggers, queries across multiple documents, and so on.

Thus, while it may be possible to use an XML document or documents as a database in environments with small amounts of data, few users, and modest performance requirements, this will fail in most production environments, which have many users, strict data integrity requirements, and the need for good performance.

A good example of the type of "database" for which an XML document is suitable is an .ini file -- that is, a file that contains application configuration information. It is much easier to invent a small XML language and write a SAX application for interpreting that language than it is to write a parser for comma-delimited files. In addition, XML allows you to have nested entries, something that is harder to do in comma-delimited files. However, this is hardly a database, since it is read and written linearly, and then only when the application is started and ended.

Examples of more sophisticated data sets for which an XML document might be suitable as a database are personal contact lists (names, phone numbers, addresses, etc.), browser bookmarks, and descriptions of the MP3s you've stolen with the help of Napster. However, given the low price and ease of use of databases like dBASE and Access, there seems little reason to use an XML document as a database even in these cases. The only real advantage of XML is that the data is portable, and this is less of an advantage than it seems due to the widespread availability of tools for serializing databases as XML.

3. Why Use a Database?

The first question you need to ask yourself when you start thinking about XML and databases is why you want to use a database in the first place. Do you have legacy data you want to expose? Are you looking for a place to store your Web pages? Is the database used by an e-commerce application in which XML is used as a data transport? The answers to these questions will strongly influence your choice of database and middleware (if any), as well as how you use that database.

For example, suppose you have an e-commerce application that uses XML as a data transport. It is a good bet that your data has a highly regular structure and is used by non-XML applications. Furthermore, things like entities and the encodings used by XML documents probably aren't important to you -- after all, you are interested in the data, not how it is stored in an XML document. In this case, you'll probably need a relational database and software to transfer the data between XML documents and the database. If your applications are object-oriented, you might even want a system that can store those objects in the database or serialize them as XML.

On the other hand, suppose you have a Web site built from a number of prose-oriented XML documents. Not only do you want to manage the site, you would like to provide a way for users to search its contents. Your documents are likely to have a less regular structure and things such as entity usage are probably important to you because they are a fundamental part of how your documents are structured. In this case, you might want a product like a native XML database or a content management system. This will allow you to preserve physical document structure, support document-level transactions, and execute queries in an XML query language.

4. Data versus Documents

Perhaps the most important factor in choosing a database is whether you are using the database to store *data* or *documents*. For example, is XML used simply as a data transport between the database and a (possibly non-XML) application? Or is its use integral, as in the case of XHTML and DocBook documents? This is usually a matter of intent, but it is important because all *data-centric documents* share a number of characteristics, as do all *document-centric documents*, and these influence how XML is stored in the database. The next two sections examine these characteristics.

(Historical footnote: I first heard the terms *data-centric* and *document-centric* on the xml-dev mailing list. I don't know who coined them, but I've found messages from 1997 using the term document-centric and messages from 1998 using both terms.)

4.1 Data-Centric Documents

Data-centric documents are documents that use XML as a data transport. They are designed for machine consumption and the fact that XML is used at all is usually superfluous. That is, it is not important to the application or the database that the data is, for some length of time, stored in an XML document. Examples of data-centric documents are sales orders, flight schedules, scientific data, and stock quotes.

Data-centric documents are characterized by fairly regular structure, fine-grained data (that is, the smallest independent unit of data is at the level of a PCDATA-only element or an attribute), and little or no mixed content. The order in which sibling elements and PCDATA occurs is generally not significant, except when validating the document.

Data of the kind that is found in data-centric documents can originate both in the database (in which case you want to expose it as XML) and outside the database (in which case you want to store it in a database). An example of the former is the vast amount of legacy data stored in relational databases; an example of the latter is scientific data gathered by a measurement system and converted to XML.

For example, the following sales order document is data-centric:

```
   <SalesOrder SONumber="12345">
      <Customer CustNumber="543">
         <CustName>ABC Industries</CustName>
         <Street>123 Main St.</Street>
         <City>Chicago</City>
         <State>IL</State>
         <PostCode>60609</PostCode>
      </Customer>
      <OrderDate>981215</OrderDate>
      <Item ItemNumber="1">
         <Part PartNumber="123">
            <Description>
               <p><b>Turkey wrench:</b><br />
               Stainless steel, one-piece construction,
               lifetime guarantee.</p>
            </Description>
            <Price>9.95</Price>
         </Part>
         <Quantity>10</Quantity>
      </Item>
      <Item ItemNumber="2">
         <Part PartNumber="456">
            <Description>
               <p><b>Stuffing separator:<b><br />
               Aluminum, one-year guarantee.</p>
            </Description>
            <Price>13.27</Price>
         </Part>
         <Quantity>5</Quantity>
      </Item>
   </SalesOrder>
```

In addition to such obviously data-centric documents as the sales order shown above, many prose-rich documents are also data-centric. For example, consider a page on Amazon.com that displays information about a book. Although the page is largely text, the structure of that text is highly regular, much of it is common to all pages describing books, and each piece of page-specific text is limited in size. Thus, the page could be built from a simple, data-centric XML document that contains the information about a single book and is retrieved from the database, and an XSL stylesheet that adds the boilerplate text. In general, any Web site that dynamically constructs HTML documents today by filling a template with database data can probably be replaced by a series of data-centric XML documents and one or more XSL stylesheets.

For example, consider the following document describing a flight:

```
   <FlightInfo>
      <Airline>ABC Airways</Airline> provides <Count>three</Count>
      non-stop flights daily from <Origin>Dallas</Origin> to
      <Destination>Fort Worth</Destination>. Departure times are
      <Departure>09:15</Departure>, <Departure>11:15</Departure>,
      and  <Departure>13:15</Departure>.  Arrival  times  are  minutes
later.
   </FlightInfo>
```

This could be built from the following XML document and a simple stylesheet:

```
<Flights>
   <Airline>ABC Airways</Airline>
   <Origin>Dallas</Origin>
   <Destination>Fort Worth</Destination>
   <Flight>
      <Departure>09:15</Departure>
      <Arrival>09:16</Arrival>
   </Flight>
   <Flight>
      <Departure>11:15</Departure>
      <Arrival>11:16</Arrival>
   </Flight>
   <Flight>
      <Departure>13:15</Departure>
      <Arrival>13:16</Arrival>
   </Flight>
</Flights>
```

4.2 Document-Centric Documents

Document-centric documents are (usually) documents that are designed for human consumption. Examples are books, email, advertisements, and almost any hand-written XHTML document. They are characterized by less regular or irregular structure, larger grained data (that is, the smallest independent unit of data might be at the level of an element with mixed content or the entire document itself), and lots of mixed content. The order in which sibling elements and PCDATA occurs is almost always significant.

Document-centric documents are usually written by hand in XML or some other format, such as RTF, PDF, or SGML, which is then converted to XML. Unlike data-centric documents, they usually do not originate in the database. (Documents built from data inserted into a template are data-centric; for more information, see the end of section 4.1.) For information on software you can use to convert various formats to XML, see the links to various lists of XML software.

For example, the following product description is document-centric:

```
<Product>

<Intro>
The <ProductName>Turkey Wrench</ProductName> from <Developer>Full
Fabrication Labs, Inc.</Developer> is <Summary>like a monkey wrench,
but not as big.</Summary>
</Intro>

<Description>

<Para>The turkey wrench, which comes in <i>both right- and left-
handed versions (skyhook optional)</i>, is made of the <b>finest
stainless steel</b>. The Readi-grip rubberized handle quickly adapts
to your hands, even in the greasiest situations. Adjustment is
```

```
   possible through a variety of custom dials.</Para>

   <Para>You can:</Para>

   <List>
   <Item><Link      URL="Order.html">Order      your      own      turkey
wrench</Link></Item>
   <Item><Link        URL="Wrenches.htm">Read        more        about
wrenches</Link></Item>
   <Item><Link URL="Catalog.zip">Download the catalog</Link></Item>
   </List>

   <Para>The turkey wrench costs <b>just $19.99</b> and, if you
   order now, comes with a <b>hand-crafted shrimp hammer</b> as a
   bonus gift.</Para>

   </Description>

   </Product>
```

4.3 Data, Documents, and Databases

In practice, the distinction between data-centric and document-centric documents is not always clear. For example, an otherwise data-centric document, such as an invoice, might contain large-grained, irregularly structured data, such as a part description. And an otherwise document-centric document, such as a user's manual, might contain fine-grained, regularly structured data (often metadata), such as an author's name and a revision date. Other examples include legal and medical documents, which are written as prose but contain discrete pieces of data, such as dates, names, and procedures, and often must be stored as complete documents for legal reasons.

In spite of this, characterizing your documents as data-centric or document-centric will help you decide what kind of database to use. As a general rule, data is stored in a traditional database, such as a relational, object-oriented, or hierarchical database. This can be done by third-party *middleware* or by capabilities built in to the database itself. In the latter case, the database is said to be *XML-enabled*. Documents are stored in a *native XML database* (a database designed especially for storing XML) or a *content management system* (an application designed to manage documents and built on top of a native XML database).

These rules are not absolute. Data -- especially semi-structured data -- can be stored in native XML databases and documents can be stored in traditional databases when few XML-specific features are needed. Furthermore, the boundaries between traditional databases and native XML databases are beginning to blur, as traditional databases add native XML capabilities and native XML databases support the storage of document fragments in external (usually relational) databases.

The remainder of this paper discusses the strategies and issues surrounding the storage and retrieval of data (section 5) and documents (section 6). For an up-to-date list of XML database products, see XML Database Products.

5. Storing and Retrieving Data

In order to transfer data between XML documents and a database, it is necessary to map the XML document schema (DTD, XML Schemas, RELAX NG, etc.) to the database schema. The data transfer software is then built on top of this mapping. The software may use an XML query language (such as XPath, XQuery, or a proprietary language) or simply transfer data according to the mapping (the XML equivalent of SELECT * FROM Table).

In the latter case, the structure of the document must exactly match the structure expected by the mapping. Since this is often not the case, products that use this strategy are often used with XSLT. That is, before transferring data to the database, the document is first transformed to the structure expected by the mapping; the data is then transferred. Similarly, after transferring data from the database, the resulting document is transformed to the structure needed by the application.

5.1 Mapping Document Schemas to Database Schemas

Mappings between document schemas and database schemas are performed on element types, attributes, and text. They almost always omit physical structure (such as entities, CDATA sections, and encoding information) and some logical structure (such as processing instructions, comments, and the order in which elements and PCDATA appear in their parent). This is more reasonable than it may sound, as the database and application are concerned only with the data in the XML document. For example, in the sales order shown above, it doesn't matter if the customer number is stored in a CDATA section, an external entity, or directly as PCDATA, nor does it matter if the customer number is stored before or after the order date.

One consequence of this is that "round-tripping" a document -- that is, storing the data from a document in the database and then reconstructing the document from that data -- often results in a different document, even in the canonical sense of the term. Whether this is acceptable depends on your needs and might influence your choice of software.

Two mappings are commonly used to map an XML document schema to the database schema: the *table-based mapping* and the *object-relational mapping*.

5.1.1 Table-Based Mapping

The table-based mapping is used by many of the middleware products that transfer data between an XML document and a relational database. It models XML documents as a single table or set of tables. That is, the structure of an XML document must be as follows, where the <database> element and additional <table> elements do not exist in the single-table case:

```
<database>
   <table>
```

```
         <row>
            <column1>...</column1>
            <column2>...</column2>
            ...
         </row>
         <row>
            ...
         </row>
         ...
      </table>
      <table>
         ...
      </table>
      ...
   </database>
```

Depending on the software, it may be possible to specify whether column data is stored as child elements or attributes, as well as what names to use for each element or attribute. In addition, products that use table-based mappings often optionally include table and column metadata either at the start of the document or as attributes of each table or column element. Note that the term "table" is usually interpreted loosely. That is, when transferring data from the database to XML, a "table" can be any result set; when transferring data from XML to the database, a "table" can be a table or an updateable view.

The table-based mapping is useful for serializing relational data, such as when transferring data between two relational databases. Its obvious drawback is that it cannot be used for any XML documents that do not match the above format.

5.1.2 Object-Relational Mapping

The object-relational mapping is used by all XML-enabled relational databases and some middleware products. It models the data in the XML document as a tree of objects that are specific to the data in the document. In this model, element types with attributes, element content, or mixed content (*complex element types*) are generally modeled as classes. Element types with PCDATA-only content (*simple element types*), attributes, and PCDATA are modeled as scalar properties. The model is then mapped to relational databases using traditional object-relational mapping techniques or SQL 3 object views. That is, classes are mapped to tables, scalar properties are mapped to columns, and object-valued properties are mapped to primary key / foreign key pairs.

(The name "object-relational mapping" is actually a misnomer, as the object tree can be mapped directly to object-oriented and hierarchical databases. However, it is used because the overwhelming majority of products that use this mapping use relational databases and the term "object-relational mapping" is well known.)

It is important to understand that the object model used in this mapping is **not** the Document Object Model (DOM). The DOM models the document itself and is the same for all XML documents, while the model described above models the data in the document and is different for each set of XML documents that conforms to a given XML schema. (By convention, the term *XML schema* with a lower case "s" refers to any XML schema -- such as a DTD, an XML Schema document, or a RELAX NG schema. *XML Schema* with a capital "S" refers to the W3C's XML Schema language.) For example, the sales order document shown above could be modeled as a tree of objects from four classes -- SalesOrder, Customer, Item, and Part -- as shown in the following diagram:

```
        SalesOrder
       /    |    \
Customer   Item   Item
            |      |
           Part   Part
```

Were a DOM tree built from the same document, it would be composed of objects such as Element, Attr, and Text:

```
                 Element --- Attr
               (SalesOrder) (SONumber)
          ____/   /   \   \_____
         /       /     \        \
  Element     Text     Element  Element
(Customer) (OrderDate)  (Item)    (Item)
     |                    |         |
    etc.                 etc.      etc.
```

Whether the objects in the model are actually instantiated depends on the product. Some products allow you to generate the classes in the model, then use these objects from these classes in your application. With such products, data is transferred between the XML document and these objects and between these objects and the database. Other products use the objects only as a tool to help visualize the mapping and transfer data directly between the XML document and the database. Whether it is useful to instantiate the intermediate objects depends entirely on your application.

(The binding of XML documents to objects is commonly known as *XML data binding*, after Sun's Java Architecture for XML Binding. A number of products implement XML data binding; many of these can transfer data between the objects and the database as well. For more information, see XML Data Binding Resources.)

The way in which the object-relational mapping is supported varies from product to product. For example:

- All products support the mapping of complex element types to classes and simple element types and attributes to properties.
- All(?) products allow you to designate a root element that is not mapped to the object model or database. This wrapper element is useful when you want to store multiple top-level objects in the same XML document. For example, if you want to store multiple sales orders in the same XML document, you need to wrap these in a single root element.

The wrapper element is equivalent to the <database> element when an XML document using the table-based mapping contains multiple tables and is present only to meet XML's requirement that a document has a single root element. A few products allow you to designate elements at lower levels as wrapper elements as well.

- Most products allow you to specify whether properties are mapped to attributes or child elements in the XML document. Some products allow you to mix attributes and child elements; others require you to use only one or the other.
- Most products allow you to use different names in the XML document and the database; a few require you to use the same names in both.
- Most products allow you to specify the order in which child elements appear in their parent, although in such a way that it is impossible to build many content models. Fortunately, the supported content models are sufficient for most data-centric documents. (Child order is important if you want to validate the document.)
- Some products allow you to map complex element types to scalar properties. This is useful when the element type contains mixed content, such as the <Description> element in the sales order example. Although the <Description> element has child elements and text in the form of XHTML, it is more useful to view the description as a single property than to break it down into its component pieces.
- Few products support the mapping of PCDATA in mixed content.

For a complete description of the object-relational mapping, see section 3 of "Mapping DTDs to Databases".

5.2 Query Languages

Many products transfer data directly according to the model on which they are built. Because the structure of the XML document is often different from the structure of the database, these products often include or are used with XSLT. This allows users to transform documents to the structure dictated by the model before transferring data to the database, as well as the reverse. Because XSLT processing can be expensive, some products also integrate a limited number of transformations into their mappings.

The long term solution to this problem is the implementation of query languages that return XML. Currently (November, 2002), most of these languages rely on SELECT statements embedded in templates. This situation is expected to change when XQuery and SQL/XML are finalized, as major database vendors are already working on implementations. Unfortunately, almost all of XML query languages (including XQuery 1.0 and the initial release of SQL/XML) are read-only, so different means will be needed to insert, update, and delete data in the near term. (In the long term, XQuery and SQL/XML will add these capabilities.)

5.2.1 Template-Based Query Languages

The most common query languages that return XML from relational databases are template-based. In these languages, there is no predefined mapping between the document and the database. Instead, SELECT statements are embedded in a template and the results are processed by the data transfer software. For example, the following template (not used by any real product) uses <SelectStmt> elements to include SELECT statements and $column-name values to determine where the results should be placed:

```
<?xml version="1.0"?>
<FlightInfo>
   <Introduction>The following flights are available:</Introduction>
   <SelectStmt>SELECT Airline, FltNumber,
               Depart, Arrive FROM Flights</SelectStmt>
      <Flight>
         <Airline>$Airline</Airline>
         <FltNumber>$FltNumber</FltNumber>
         <Depart>$Depart</Depart>
         <Arrive>$Arrive</Arrive>
      </Flight>
   <Conclusion>We hope one of these meets your needs</Conclusion>
</FlightInfo>
```

The result of processing such a template might be:

```
<?xml version="1.0"?>
<FlightInfo>
   <Introduction>The following flights are available:</Introduction>
   <Flights>
      <Flight>
         <Airline>ACME</Airline>
         <FltNumber>123</FltNumber>
         <Depart>Dec 12, 1998 13:43</Depart>
         <Arrive>Dec 13, 1998 01:21</Arrive>
      </Flight>
      ...
   </Flights>
   <Conclusion>We hope one of these meets your needs.</Conclusion>
</FlightInfo>
```

Template-based query languages can be tremendously flexible. Although the feature set varies from product to product, some commonly occurring features are:

- The ability to place result set values anywhere in the output document, including as parameters in subsequent SELECT statements.
- Programming constructs such as for loops and if statements.
- Variables and function definitions.
- Parameterization of SELECT statements through HTTP parameters.

Template-based query languages are used almost exclusively to transfer data from relational databases to XML documents. Although some products that use template-based query languages can transfer data from XML documents to relational databases, they do not use their full template language for this purpose. Instead, they use a table-based mapping, as described above.

5.2.2 SQL-Based Query Languages

SQL-based query languages use modified SELECT statements, the results of which are transformed to XML. A number of proprietary SQL-based languages are currently available. The simplest of these uses nested SELECT statements, which are transformed directly to nested XML according to the object-relational mapping. The next uses SQL 3 object views, also transformed directly to XML. The last uses OUTER UNION statements and special markup to determine how results are transformed to XML.

In addition to the proprietary languages, a number of companies joined together in 2000 to standardize XML extensions to SQL. Their work is now part of the ISO SQL specification and it known as SQL/XML; final approval is expected in late 2003. SQL/XML introduces an XML data type and adds a number of functions to SQL so it is possible to construct XML elements and attributes from relational data.

For example, the following query:

```
   SELECT Orders.SONumber,
          XMLELEMENT(NAME "Order",
                     XMLATTRIBUTES(Orders.SONumber AS SONumber),
                     XMLELEMENT(NAME "Date", Orders.Date),
                     XMLELEMENT(NAME  "Customer",  Orders.Customer))  AS
xmldocument
   FROM Orders
```

constructs a result set with two columns. The first column contains the sales order number and the second column contains an XML document. There is one XML document per row, constructed from the data from the corresponding row in the Orders table. For example, the document for the row for sales order 123 might be:

```
   <Order SONumber="123">
      <Date>10/29/02</Date>
      <Customer>Gallagher Industries</Customer>
   </Order>
```

Information about SQL/XML is difficult to find on the Web. For older information and some introductory articles, see the SQLX Group Web site, from which the final committee draft of the SQL/XML specification (PDF) is available.

5.2.3 XML Query Languages

Unlike template-based query languages and SQL-based query languages, which can only be used with relational databases, XML query languages can be used over any XML document. To use these with relational databases, the data in the database must be modeled as XML, thereby allowing queries over virtual XML documents.

With XQuery, either a table-based mapping or an object-relational mapping can be used. If a table-based mapping is used, each table is treated as a separate document and joins between tables (documents) are specified in the query itself, as in SQL. If an object-relational mapping is used, hierarchies of tables are treated as a single document and joins are specified in the mapping. It appears likely that table-based mappings will be used in most implementations over relational databases, as these appear to be simpler to implement and more familiar to users of SQL.

With XPath, an object-relational mapping must be used to do queries across more than one table. This is because XPath does not support joins across documents. Thus, if the table-based mapping was used, it would be possible to query only one table at a time.

5.3 Storing Data in a Native XML Database

It is also possible to store data in XML documents in a native XML database. There are several reasons to do this. The first of these is when your data is semi-structured. That is, it has a regular structure, but that structure varies enough that mapping it to a relational database results in either a large number of columns with null values (which wastes space) or a large number of tables (which is inefficient). Although semi-structured data can be stored in object-oriented and hierarchical databases, you can also choose to store it in a native XML database in the form of an XML document.

A second reason to store data in a native XML database is retrieval speed. Depending on how the native XML database physically stores data, it might be able to retrieve data much faster than a relational database. The reason for this is that some storage strategies used by native XML databases store entire documents together physically or use physical (rather than logical) pointers between the parts of the document. This allows the documents to be retrieved either without joins or with physical joins, both of which are faster than the logical joins used by relational databases.

For example, consider the sales order document shown above. In a relational database, this would be stored in four tables -- SalesOrders, Items, Customers, and Parts -- and retrieving the document would require joins across these tables. In a native XML database, the entire document might be stored in a single place on the disk, so retrieving it or a fragment of it requires a single index lookup and a single read to retrieve the data. A relational database requires four index lookups and at least four reads to retrieve the data.

The obvious drawback in this case is that the increased speed applies only when retrieving data in the order it is stored on disk. If you want to retrieve a different view of the data, such as a list of customers and the sales orders that apply to them, performance will probably be worse than in a relational database. Thus, storing data in a native XML database for performance reasons applies only when one view of the data predominates in your application.

A third reason to store data in a native XML database is that you want to exploit XML-specific capabilities, such as executing XML queries. Given that few data-centric applications need this today and that relational databases are implementing XML query languages, this reason is less compelling.

One problem with storing data in a native XML database is that most native XML databases can only return the data as XML. (A few support the binding of elements or attributes to application variables.) If your application needs the data in another format (which is likely), it must parse the XML before it can use the data. This is clearly a disadvantage for local applications that use a native XML database instead of a relational database, as it incurs overhead not found in (for example) an ODBC application. It is not a problem with distributed applications that use XML as a data transport, since they must incur this overhead regardless of what type of database is used.

5.4 Data Types, Null Values, Character Sets, and All That Stuff

This section discusses a number of issues related to storing data from XML documents in traditional databases. (The issues about data types and binary data apply to storing data in native XML databases as well.) Generally, you will have no choice about how your data transfer software resolves these issues. However, you should be aware that these issues exist, as they might help you to choose software.

5.4.1 Data Types

XML does not support data types in any meaningful sense of the word. Except for unparsed entities, all data in an XML document is text, even if it represents another data type, such as a date or an integer. Generally, the data transfer software will convert data from text (in the XML document) to other types (in the database) and vice versa.

How the software determines what conversion to perform is product-specific, but two methods are common. The first method is that the software determines the data type from the database schema, since this is always available at run time. (The XML schema is not necessarily available at run time and might not even exist.) The second method is that the user explicitly supplies the data type, such as in mapping information. This can be written by the user or generated automatically from a database schema or XML schema. When generated automatically, data types can be retrieved from database schemas and from some types of XML schemas (XML Schemas, RELAX NG).

One additional problem with conversions is what text formats are recognized (when transferring data from XML) or can be created (when transferring data to XML). In most cases, the number of text formats that are supported for a particular data type is likely to be limited, such as to a single, specific format or to those supported by a given JDBC driver. Dates are most likely to cause problems, as the range of possible formats is enormous. Numbers, with their various international formats, can cause problems as well.

5.4.2 Binary Data

There are three common ways to store binary data in XML documents: Base64 encoding (a MIME encoding that maps binary data to a subset of US-ASCII [0-9a-zA-Z+/] -- see RFC 2045), hex encoding (where each binary octet is encoded using two characters representing hexadecimal digits [0-9a-fA-F]), and unparsed entities (where the binary data is stored in a separate physical entity from the rest of the XML document).

The biggest problem with binary data is that many (most?) data transfer products don't support it, so be sure to check if yours does. A secondary problem occurs if unparsed entities are used. The problem here is that most (all?) data transfer products don't store notation and entity declarations. Thus, the notation associated with a particular entity will be lost when data is transferred from an XML document to the database. (For more information about notations, see section 4.7 of the XML 1.0 recommendation.)

5.4.3 Null Data

In the database world, *null data* means data that simply isn't there. This is very different from a value of 0 (for numbers) or zero length (for a string). For example, suppose you have data collected from a weather station. If the thermometer isn't working, a null value is stored in the database rather than a 0, which would mean something different altogether.

XML also supports the concept of null data through optional element types and attributes. If the value of an optional element type or attribute is null, it simply isn't included in the document. As with databases, attributes containing zero length strings and empty elements are not null: their value is a zero-length string.

When mapping the structure of an XML document to the database and vice versa, you should be careful that optional element types and attributes are mapped to nullable columns and vice versa. The result of not doing so is likely to be an insertion error (when transferring data to the database) or an invalid document (when transferring data from the database).

Because the XML community is likely to have a more flexible notion of what is meant by null than the database community -- in particular, many XML users are likely to consider attributes containing zero-length strings and empty elements to be "null" -- you should check how your software handles this situation. Some software offers the user the choice of defining what constitutes "null" in an XML document, including supporting the xsi:null attribute from XML Schemas.

5.4.4 Character Sets

By definition, an XML document can contain any Unicode character except some of the control characters. Unfortunately, many databases offer limited or no support for Unicode and require special configuration to handle non-ASCII characters. If your data contains non-ASCII characters, be sure to check how and if both your database and data transfer software handle these characters.

5.4.5 Processing Instructions and Comments

Processing instructions and comments are not part of the "data" of an XML document and most (all?) data transfer software cannot handle them. The problem is that they can occur virtually anywhere in an XML document and therefore do not easily fit into the table-based and object-relational mappings. (One clearly unworkable solution in a relational database is to add tables for processing instructions and comments, add foreign keys to these tables for all other tables, and to check these tables whenever processing another table.) Thus, most data transfer software simply discards them. If processing instructions and comments are important to you, check how and if your software handles them. You might also consider using a native XML database.

5.4.6 Storing Markup

As was mentioned in section 5.1.2, it is sometimes useful to store elements with mixed content in the database without further parsing. When this is done, markup is stored with markup characters. However, a problem arises with how to store markup characters that are not used for markup. In an XML document, these are stored using the lt, gt, amp, quot, and apos entities. This can also be done in the database. For example, the following description:

```
<description>
   <b>Confusing example:</b> &lt;foo/&gt;
</description>
```

can be stored in the database as:

```
   <b>Confusing example:</b> &lt;foo/&gt;
```

The problem with this is that non-XML query languages, such as SQL, do not scan column values for markup and entity usage and interpret them accordingly. Thus, if you wanted to search for the string "<foo/>" with SQL, you would need to know that you actually needed to search for the string "&lt;foo/&gt;".

On the other hand, if entity references are expanded, the data transfer software cannot distinguish markup from entity usage. For example, if the above description is stored in the database as follows, the software cannot tell whether <b>, </b>, and <foo/> are markup or text.

```
<b>Confusing example:</b> <foo/>
```

The long term solution to this problem is XML-aware databases in which actual markup is treated differently from things that only look like markup. Still, there will probably always be problems when non-XML applications handle XML.

5.5 Generating XML Schemas from Relational Schemas and Vice Versa

A common question when transferring data between XML documents and a database is how to generate XML schemas from database schema and vice versa. Before explaining how to do this, it is worth noting that this is a design-time operation. The reason for this is that most data-centric applications, and virtually all vertical applications, work with a known set of XML schemas and database schemas. Thus, they don't need to generate schemas at run-time. Furthermore, as will be seen below, the procedures for generating schemas are less than perfect. Applications that need to store random XML documents in a database should probably use a native XML database instead of generating schemas at run time.

The easiest way to generate relational schemas from XML schemas and vice versa is to simply hardcode a path through the object-relational mapping, which has a number of optional features. Similar procedures exist for use with object-oriented databases. (For step-by-step examples of each procedure, see section 4 of Mapping DTDs to Databases.)

To generate a relational schema from an XML schema:

1. For each complex element type, create a table and a primary key column.
2. For each element type with mixed content, create a separate table in which to store the PCDATA, linked to the parent table through the parent table's primary key.
3. For each single-valued attribute of that element type, and for each singly-occurring simple child element, create a column in that table. If the XML schema has data type information, then set the data type of the column to the corresponding type. Otherwise, set the data type to a pre-determined type, such as CLOB or VARCHAR(255). If the child element type or attribute is optional, make the column nullable.
4. For each multi-valued attribute and for each multiply-occurring simple child element, create a separate table to store values, linked to the parent table through the parent table's primary key.
5. For each complex child element, link the parent element type's table to the child element type's table with the parent table's primary key.

To generate an XML schema from a relational schema:

1. For each table, create an element type.
2. For each data (non-key) column in that table, as well as for the primary key column(s), add an attribute to the element type or a PCDATA-only child element to its content model.
3. For each table to which the primary key is exported, add a child element to the content model and process the table recursively.
4. For each foreign key, add a child element to the content model and process the foreign key table recursively.

There are a number of drawbacks to these procedures. Many of these are easy to fix by hand, such as name collisions and specifying column data types and lengths. (DTDs do not contain data type information, so it is impossible to predict what data types should be used in the database. Note that data types and lengths can be predicted from an XML Schema document.)

A more serious problem is that the data "model" used by the XML document is often different (and usually more complex) than the most efficient model for storing data in the database. For example, consider the following fragment of XML:

```
<Customer>
   <Name>ABC Industries</Name>
   <Address>
      <Street>123 Main St.</Street>
      <City>Fooville</City>
      <State>CA</State>
      <Country>USA</Country>
      <PostCode>95041</PostCode>
   </Address>
</Customer>
```

The procedure for generating a relational schema from an XML schema would create two tables here: one for customers and one for addresses. However, in most cases it would make more sense to store the address in the customer table, not a separate table.

The <Address> element is a good example of a *wrapper element*. Wrapper elements are generally used for two reasons. First, they help to provide additional structure that makes the document easier to understand. Second, they are commonly used as a form of data typing. For example, the <Address> element could be passed to a routine that converts all addresses to Address objects, regardless of where they occur.

While wrapper elements are useful in XML, they generally cause problems in the form of extra structure when they are mapped to the database. For this reason, they should generally be eliminated from the XML schema before generating a relational schema. Since it is unlikely that the XML schema can or should be changed on a permanent basis, this results in a mismatch between the actual documents and the documents expected by the data transfer software, since the wrapper elements are not included in the mapping. This can be resolved by transforming the documents at run time, such as with XSLT: wrapper elements are eliminated before transferring data to the database and inserted after transferring data from the database.

In spite of these drawbacks, the above procedures still provide a useful starting point for generating XML schemas from relational schema and vice versa, especially in large systems.

6. Storing and Retrieving Documents

There are two basic strategies to storing XML documents: store them in the file system or as a BLOB in a relational database and accept limited XML functionality, or store them in a native XML database.

6.1 Storing Documents in the File System

If you have a simple set of documents, such as a small documentation set, the easiest way to store them is in the file system. You can then use tools like grep to query them and sed to modify them. (Full text searches of XML documents are obviously inaccurate, as they cannot easily distinguish markup from text and cannot understand entity usage. However, in a small system, such inaccuracies may be acceptable.) If you want simple transaction control, you can place your documents in a version control system such as CVS or RCS.

6.2 Storing Documents in BLOBs

A slightly more sophisticated option is to store documents as BLOBs in a relational database. This provides a number of the advantages of databases: transactional control, security, multi-user access, and so on. In addition, many relational databases have tools for searching text and can do such things as full-text searches, proximity searches, synonym searches, and fuzzy searches. Some of these tools are being made XML-aware, which will eliminate the problems involved with searching XML documents as pure text.

When you store XML documents as BLOBs, you can also easily implement your own simple XML-aware indexing, even if the database cannot index XML. One way to do this is to create two tables, an index table (known as a *side table* in DB2, where the idea originated) and a document table. The document table contains a primary key and a BLOB column in which the document is stored. The index table contains a column containing the value to be indexed and a foreign key pointing to the primary key of the document table.

When the document is stored in the database, it is searched for all instances of the element or attribute being indexed. Each instance is stored in the index table, along with the primary key of the document. The value column is then indexed, which allows an application to quickly search for a particular element or attribute value and retrieve the corresponding document.

For example, suppose you had a set of documents that matched the following DTD and want to build an index of authors:

```
<!ELEMENT Brochure (Title, Author, Content)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Author (#PCDATA)>
<!ELEMENT Content (%Inline;)> <!-- Inline entity from XHTML -->
```

You could store these in the following tables:

```
Authors                    Brochures
----------------------     ---------
Author     VARCHAR(50)     BrochureID INTEGER
BrochureID INTEGER         Brochure   LONGVARCHAR
```

When you insert a brochure into the database, the application inserts the brochure into the Brochures table, then scans it for <Author> elements, storing their values and the brochure ID in the Authors table. The application can then retrieve brochures by author with a simple SELECT statement. For example, to retrieve all of the brochures written by the author Chen, the application executes the statement:

```
   SELECT Brochure
   FROM Brochures
   WHERE  BrochureID  IN  (SELECT  BrochureID  FROM  Authors  WHERE
Author='Chen')
```

A more sophisticated index table would contain four columns: element type or attribute name, type (element or attribute), value, and document ID. This could store the values of multiple markup constructs in a single table and would need to be indexed on name, type, and value. Writing a generic SAX application to populate such a table would be relatively easy.

6.3 Native XML Databases

If you need more features than are offered by one of the simple systems described above, you should consider a native XML database. *Native XML databases* are databases designed especially to store XML documents. Like other databases, they support features like transactions, security, multi-user access, programmatic APIs, query languages, and so on. The only difference from other databases is that their internal model is based on XML and not something else, such as the relational model.

Native XML databases are most commonly used to store document-centric documents. The main reason for this is their support of XML query languages, which allow you to ask questions like, "Get me all documents in which the third paragraph after the start of the section contains a bold word," or even just to limit full-text searches to certain portions of a document. Such queries are clearly difficult to ask in a language like SQL. Another reason is that native XML databases preserve things like document order, processing instructions, and comments, and often preserve CDATA sections, entity usage, and so on, while XML-enabled databases do not.

Native XML databases are also commonly used to integrate data. While data integration has historically been performed through federated relational databases, these require all data sources to be mapped to the relational model. This is clearly unworkable for many types of data and the XML data model provides much greater flexibility. Native XML databases also handle schema changes more easily than relational databases and can handle schemaless data as well. Both are important considerations when integrating data from sources not under your direct control.

The third major use case for native XML databases is semi-structured data, such as is found in the fields of finance and biology, which change so frequently that definitive schemas are often not possible. Because native XML databases do not require schemas (as do relational databases), they can handle this kind of data, although applications often require humans to process it.

The final major use of native XML database is in handling schema evolution. While native XML databases do not provide complete solutions by any means, they do provide more flexibility than relational databases. For example, native XML databases do not require existing data to be migrated to a new schema, can handle schema changes for which there is no data migration path, and can store data even if it conforms to an unknown version of a schema.

Other uses of native XML databases include providing data and metadata caches for long-running transactions, handling large documents, handling hierarchical data, and acting as a mid-tier data cache. For a complete discussion of use cases for native XML databases, see Going Native: Use Cases for Native XML Databases.

6.3.1 What is a Native XML Database?

The term "native XML database" first gained prominence in the marketing campaign for Tamino, a native XML database from Software AG. Perhaps due to the success of this campaign, the term came into common usage among companies developing similar products. The drawback of this is that, being a marketing term, it has never had a formal technical definition.

One possible definition (developed by members of the XML:DB mailing list) is that a native XML database is one that:

Defines a (logical) model for an XML document -- as opposed to the data in that document -- and stores and retrieves documents according to that model. At a minimum, the model must include elements, attributes, PCDATA, and document order. Examples of such models are the XPath data model, the XML Infoset, and the models implied by the DOM and the events in SAX 1.0.
Has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage.
Is not required to have any particular underlying physical storage model. For example, it can be built on a relational, hierarchical, or object-oriented database, or use a proprietary storage format such as indexed, compressed files.

The first part of this definition is similar to the definitions of other types of databases, concentrating on the model used by the database. It is worth noting that a given native XML database might store more information than is contained in the model it uses. For example, it might support queries based on the XPath data model but store documents as text. In this case, things like CDATA sections and entity usage are stored in the database but not included in the model.

The second part of the definition states that the fundamental unit of storage in a native XML database is an XML document. While it seems possible that a native XML database could assign this role to document fragments, it is filled by documents in all(?) native XML databases today.

(The fundamental unit of storage is the lowest level of context into which a given piece of data fits, and is equivalent to a row in a relational database. Its existence does not preclude retrieving smaller units of data, such as document fragments or individual elements, nor does it preclude combining fragments from one document with fragments for another document. In relational terms, this is equivalent to saying that the existence of rows does not preclude retrieving individual column values or creating new rows from existing rows.)

The third part of the definition states that the underlying data storage format is not important. This is true, and is analogous to stating that the physical storage format used by a relational database is unrelated to whether that database is relational.

6.3.2 Native XML Database Architectures

The architectures of native XML databases fall into two broad categories: text-based and model-based.

6.3.2.1 Text-Based Native XML Databases

A *text-based native XML database* is one that stores XML as text. This might be a file in a file system, a BLOB in a relational database, or a proprietary text format. (It is worth noting that a relational database that has added XML-aware processing of CLOB (Character Large OBject) columns is, in fact, a native XML database with respect to these abilities.)

Common to all text-based native XML databases are indexes, which allow the query engine to easily jump to any point in any XML document. This gives such databases a tremendous speed advantage when retrieving entire documents or document fragments. This is because the database can perform a single index lookup, position the disk head once, and, assuming that the necessary fragment is stored in contiguous bytes on the disk, retrieve the entire document or fragment in a single read. In contrast, reassembling a document from pieces, as is done in a relational database and some model-based native XML databases, requires multiple index lookups and multiple disk reads.

In this sense, a text-based native XML database is similar to a hierarchical database, in that both can outperform a relational database when retrieving and returning data according to a predefined hierarchy. Also like a hierarchical database, text-based native XML databases are likely to encounter performance problems when retrieving and returning data in any other form, such as inverting the hierarchy or portions of it. Whether this will prove to be true is as yet unknown, but the predominance of relational databases, whose use of logical pointers allows all queries of the same complexity to be performed with the same speed, seems to indicate it will be the case.

6.3.2.2 Model-Based Native XML Databases

The second category of native XML databases is *model-based native XML databases*. Rather than storing the XML document as text, they build an internal object model from the document and store this model. How the model is stored depends on the database. Some databases store the model in a relational or object-oriented database. For example, storing the DOM in a relational database might result in tables such as Elements, Attributes, PCDATA, Entities, and EntityReferences. Other databases use a proprietary storage format optimized for their model.

(For an example of a simple, model-based native XML database built on a relational database, see the system described by Mark Birbeck on the XML-L mailing list for December, 1998. The system uses five tables -- attribute definitions, element/attribute association, content model definition, attribute values, and element values (PCDATA or pointers to other elements) -- and a model that includes only elements, attributes, text, and document order. Look for the topics entitled "Record ends, Mixed content, and storing XML documents on relational database" and "storing XML documents on relational database".)

Model-based native XML databases built on other databases are likely to have performance similar to those databases when retrieving documents for the obvious reason that they rely on those systems to retrieve data. However, the design of the database, especially for native XML databases built on top of relational databases, has significant room for variation. For example, a database that used a straight object-relational mapping of the DOM could result in a system that required executing separate SELECT statements to retrieve the children of each node. On the other hand, most such databases optimize their storage models and retrieval software. For example, Richard Edwards has described a system for storing the DOM in a relational database that can retrieve any document fragment (including the entire document) with a single SELECT statement.

Model-based native XML databases that use a proprietary storage format are likely to have performance similar to text-based native XML databases when retrieving data in the order in which it is stored. This is because most(?) such databases use physical pointers between nodes, which should provide performance similar to retrieving text. (Which is faster also depends on the output format. Text-based systems are obviously faster at returning documents as text, while model-based systems are obviously faster at returning documents as DOM trees, assuming their model maps easily to the DOM.)

Like text-based native XML databases, model-based native XML databases are likely to encounter performance problems when retrieving and returning data in any form other than that in which it is stored, such as when inverting the hierarchy or portions of it. Whether they will be faster or slower than text-based systems is not clear.

6.3.3 Features of Native XML Databases

This section briefly discusses a number of the features found in native XML databases. It should help give you an idea of what features are available today and what features to expect in the future.

6.3.3.1 Document Collections

Many native XML databases support the notion of a collection. This plays a role similar to a table in a relational database or a directory in a file system. For example, suppose you are using a native XML database to store sales orders. In this case, you might want to define a sales order collection so that queries over sales orders could be limited to documents in that collection.

As another example, suppose you are storing the manuals for all of a company's products in a native XML database. In this case, you might want to define a hierarchy of collections. For example, you might have a collection for each product and, within this collection, collections for all of the chapters in each manual.

Whether collections can be nested depends on the database.

6.3.3.2 Query Languages

Almost all native XML databases support one or more query languages. The most popular of these are XPath (with extensions for queries over multiple documents) and XQuery, although numerous proprietary query languages are supported as well. When considering a native XML database, you should probably check that the query language supports your needs, as these might range from full-text-style searches to the need to recombine fragments from multiple documents.

In the future, most native XML databases will probably support XQuery from the W3C.

6.3.3.3 Updates and Deletes

Native XML databases have a variety of strategies for updating and deleting documents, from simply replacing or deleting the existing document to modifications through a live DOM tree to languages that specify how to modify fragments of a document. Most of these methods are proprietary. However, two somewhat standard languages for updating XML documents have emerged:

- XUpdate, from the XML:DB Initiative, is an XML-based language. It uses XPath to identify a set of nodes, then specifies whether to insert or delete these nodes, or insert new nodes before or after them. XUpdate has been implemented in a number of native XML databases.
- A set of extensions to XQuery has been proposed by members of the W3C XQuery working group and Patrick Lehti. Variations on these extensions have been implemented in a number of native XML databases and it seems likely that these will form the basis of the update syntax in XQuery.

In spite of these languages, update abilities are likely to remain fragmented until an update syntax is formally added to XQuery.

6.3.3.4 Transactions, Locking, and Concurrency

Virtually all native XML databases support transactions (and presumably support rollbacks). However, locking is often at the level of entire documents, rather than at the level of individual nodes, so multi-user concurrency can be relatively low. Whether this is an issue depends on the application and what constitutes a "document". For example:

- A document is a chapter of a user's guide and writers edit chapters. Document-level locking is unlikely to cause concurrency problems, as two writers updating the same chapter at the same time is unlikely.
- A document stores all of the data about a company's sales leads and salespeople enter new sales lead information. Document-level locking is likely to cause concurrency problems, as the chances of two salespeople updating lead information at the same time is fairly high. Fortunately, this can be at least partially solved by creating one sales lead document per prospective customer.
- A document contains the data used in a workflow, such as a financial contract. Each step of the workflow reads data from the document and adds its own data. For example, one step might perform a credit check and add a credit score to the document. Another step might check for outstanding balances on other contracts with the same customer and add the total outstanding balance. If node-level locking is used, some of these steps may be executed in parallel. If document-level locking is used, they must be executed serially to avoid write conflicts. This may cause unacceptable delays in high-volume applications.

The problem with node-level locking is implementing it. Locking a node usually requires locking its parent, which in turn requires locking its parent, and so on back to the root, effectively locking the entire document. To see why this is true, consider a transaction that reads a leaf node. If the transaction does not acquire locks on the ancestors of the leaf node, another transaction can delete an ancestor of the leaf node, in turn deleting the leaf node. However, it is also clear that another transaction should be able to update those parts of the document not on the direct path from the root to the leaf node.

A partial solution for this problem is proposed by Stijn Dekeyser, et al. While they do not entirely avoid the problem of locking the ancestors of a target node, they do make these locks more flexible by annotating them with the query defining the path from the locked node to the target node. This allows other transactions to determine whether they conflict with transactions already holding locks. (Because evaluating queries in order to acquire locks is prohibitively expensive, the actual scheme is somewhat more limited than what is described here, but this is the general idea.) For links to papers describing other locking schemes, see the Academic Papers section of XML / Database Links. Note that a few native XML databases do support node-level locking, but I don't know how they implement it.

In the future, most native XML databases will probably offer node-level locking.

6.3.3.5 Application Programming Interfaces (APIs)

Almost all native XML databases offer programmatic APIs. These are usually in the form of an ODBC-like interface, with methods for connecting to the database, exploring metadata, executing queries, and retrieving results. Results are usually returned as an XML string, a DOM tree, or a SAX Parser or XMLReader over the returned document. If queries can return multiple documents, then methods for iterating through the result set are available as well. Although most native XML databases offer proprietary APIs, two vendor-neutral XML database APIs have been (are being) developed [July, 2004]:

- The XML:DB API from XML:DB.org is programming language-neutral, uses XPath as its query language, and is being extended to support XQuery. It has been implemented by a number of native XML databases and may have been implemented over non-native databases as well.
- JSR 225: XQuery API for Java (XQJ) is based on JDBC and uses XQuery as its query language. It is being developed through Sun's Java Community Process (JCP) and a draft version is available. Because this initiative is being led by IBM and Oracle, its eventual wide-spread adoption seems likely.

Most native XML databases also offer the ability to execute queries and return results over HTTP.

6.3.3.6 Round-Tripping

One important feature of native XML databases is that they can *round-trip* XML documents. That is, you can store an XML document in a native XML database and get the "same" document back again. This is important to document-centric applications, for which things like CDATA sections, entity usage, comments, and processing instructions form an integral part of the document. It is also vital to many legal and medical applications, which are required by law to keep exact copies of documents.

(Round-tripping is less important to data-centric applications, which generally care only about elements, attributes, text, and hierarchical order. All software that transfers data between XML documents and databases can round-trip these. It can also round-trip sibling order (the order of elements and PCDATA in their parent) in a limited number of cases that are important to data-centric applications. However, because it cannot round-trip sibling order in general, and also cannot round-trip processing instructions, comments, and physical structure (entity references, CDATA sections and so on), it is unsuitable for document-centric applications.)

All native XML databases can round-trip documents at the level of elements, attributes, PCDATA, and document order. How much more they can round-trip depends on the database. As a general rule, text-based native XML databases round-trip XML documents exactly, while model-based native XML databases round-trip XML documents at the level of their document model. In the case of particularly minimal document models, this means round-tripping at a level less than canonical XML.

Since the level of round-tripping you need depends entirely on your application, you may have a choice of many native XML databases or be restricted to only a few.

6.3.3.7 Remote Data

Some native XML databases can include remote data in documents stored in the database. Usually, this is data retrieved from a relational database with ODBC, OLE DB, or JDBC and modeled using the table-based mapping or an object-relational mapping. Whether the data is live -- that is, whether updates to the document in the native XML database are reflected in the remote database -- depends on the native XML database. Eventually, most native XML databases will probably support live remote data.

6.3.3.8 Indexes

All native XML databases support indexes as a way to increase query speed. These are three types of indexes. *Value indexes* index text and attribute values and are used to resolve queries such as, "Find all elements or attributes whose value is 'Santa Cruz'." *Structural indexes* index the location of elements and attributes and are used to resolve queries such as, "Find all Address elements." Value and structural indexes are combined to resolve queries such as, "Find all City elements whose value is 'Santa Cruz'." Finally, *full-text indexes* index the individual tokens in text and attribute values and are used to resolve queries such as, "Find all documents that contain the words 'Santa Cruz'," or, in conjunction with structural indexes, "Find all documents that contain the words 'Santa Cruz' inside an Address element."

Most native XML databases support both value and structural indexes. Some native XML databases support full-text indexes.

6.3.3.9 External Entity Storage

A difficult question when storing XML documents is how to handle external entities. That is, should they be expanded and their value stored with the rest of the document, or should the entity reference be left in place? There is no single answer to this question.

For example, suppose a document includes an external general entity that calls a CGI program for the current weather report. If the document is used as a Web page to give current weather reports, it would be a mistake to expand the entity reference, as the Web page would no longer return live data. On the other hand, if the document were part of a collection of historical weather data, it would be a mistake *not* to expand the entity reference, as the document would always retrieve the current data rather than containing the historic data.

As another example, consider a product manual that consisted of nothing but references to external entities that point to the chapters of the manual. If some of these chapters were used in other documents, such as manuals for differents model of the same product, it would be a mistake to expand these references, as this would result in multiple copies of the same chapters.

I'm not sure how native XML databases handle this problem. Ideally, they should allow you to specify whether to expand external entity references on a case-by-case basis.

6.3.4 Normalization, Referential Integrity, and Scalability

For many people, especially those with relational database backgrounds, native XML databases raise a number of controversial issues, particularly with respect to issues surrounding the storage of data (as opposed to documents). These are discussed in the following sections.

6.3.4.1 Normalization

Normalization refers to the process of designing a database schema in which a given piece of data is represented only once. Normalization has several obvious advantages, such as reducing disk space and eliminating the possibility of inconsistent data, which can occur when a given piece of data is stored in more than one place. It is one of the cornerstones of relational technology and is a flashpoint in discussions about storing data in native XML databases.

Before discussing normalization further, it is useful to note that it is a non-issue for many document-centric documents. For example, consider a collection of documents describing a company's products. In many such collections, there is only a small amount of data that is common to all documents -- copyright notices, corporate addresses and phone numbers, product logos, and so on -- and this amount is so small in relation to the total that almost nobody considers normalizing it. (On the other hand, other documentation sets have significant overlap between documents and are worth normalizing.)

Like relational databases, there is nothing in native XML databases that forces you to normalize your data. That is, you can design bad data storage with a native XML database just as easily as you can with a relational database. Thus, it is important to consider the structure of your documents before you store them in a native XML database. (Native XML databases have one advantage over relational databases here. Since native XML databases do not have database schemas, you can store similar documents with more than one schema in the database at the same time. While you may still need to redesign queries and convert your existing documents -- a non-trivial process -- this may ease the transition process.)

Normalizing data for a native XML database is largely the same as normalizing it for a relational database: you need to design your documents so that no data is repeated. One difference between native XML databases and relational databases is that XML supports multi-valued properties while (most) relational databases do not. This makes it possible to "normalize" data in a native XML database in a way that is not possible in a relational database.

For example, consider a sales order. This consists of header information, such as the sales order number, date, and customer number, and one or more line items, which contain a part number, quantity, and total price. In a relational database, the header information must be stored in a separate table from the line items, since there are multiple line items per header. In a native XML database, this information can be stored in a single document without redundancy, since the hierarchical nature of XML allows a given parent element to have multiple children.

Unfortunately, real-world normalization isn't this simple. For example, what happens when you want the sales order to contain customer information, such as contact names and shipping and billing addresses? There are two choices here. First, you can duplicate the customer information in each sales order for that customer, leading to redundant data and all the problems it entails. Second, you can store the customer information separately and either provide an XLink in the sales order document to the customer document or join the two documents when the data is queried. This assumes that either XLinks are supported (in most cases, they aren't, although support is planned) or the query language supports joins (also not always true).

In practice, there are no clear answers. Real-world relational data is often non-normal for performance reasons, so having non-normal XML data might not be as bad as it sounds, but this is a decision you need to make. If you are storing document-centric documents and can normalize these to a reasonable degree -- such as storing chapters or procedures as separate documents and joining them to create end-user documents -- then a native XML database is probably a good solution for you, especially since it will provide you with features such as XML query languages not found in other databases. If you are storing data-centric documents and a native XML database improves your application's performance or provides semi-structured data storage that you can't find in another database, then you should use it. But if you find yourself essentially building a relational database inside your native XML database, you might ask yourself why you aren't using a relational database in the first place.

6.3.4.2 Referential Integrity

Closely related to normalization is *referential integrity*. Referential integrity refers to the validity of pointers to related data and is a necessary part of maintaining a consistent database state: It does you no good if a sales order contains a customer number and there is no corresponding customer record. The shipping department won't know where to send the items purchased and the accounting department won't know where to send the invoice.

In a relational database, referential integrity means ensuring that foreign keys point to valid primary keys -- that is, checking that the primary key row corresponding to any foreign key exists. In a native XML database, referential integrity means ensuring that "pointers" in XML documents point to valid documents or document fragments.

Pointers in XML documents come in several forms: ID/IDREF attributes, key/keyref fields (as defined in XML Schemas), XLinks, and various proprietary mechanisms. The latter includes language-specific "referencing" elements and attributes, such as the ref attribute of the <element> element in XML Schemas, and database-specific linking mechanisms. While language-specific "referencing" elements and attributes are quite common, database-specific linking mechanisms are not.

Referential integrity in native XML databases can be broken into two categories: integrity of internal pointers (pointers within a document) and integrity of external pointers (pointers between documents). The referential integrity of internal pointers that use nonstandard mechanisms is never enforced, since native XML databases have no way of identifying such pointers. The referential integrity of internal pointers that use a standard mechanism, such as ID/IDREF or key/keyref, is at least partially supported through validation.

The reason that such support is partial is that most native XML databases perform validation only when a document is inserted into the database. Thus, if updates are performed at the document level -- that is, by deleting and then replacing a document -- validation is sufficient to ensure the integrity of internal pointers. But if updates are performed at the node level -- that is, inserting, modifying, and deleting individual nodes -- then the database needs to perform additional work (such as validating all changes) to guarantee the referential integrity of internal pointers. Only a few native XML databases do this, if any.

The referential integrity of external pointers is (as far as I know) not supported, even by the few databases that support external pointers. If an external pointer points to a resource stored elsewhere in the database, there is no reason not to enforce the integrity of the pointer. However, if the pointer points to a resource outside the database, lack of enforcement is reasonable. For example, suppose a document contains an XLink that points to a document on an external Web site. The database obviously has no control over whether the latter document exists and thus cannot reasonably enforce the integrity of the XLink.

In the future, most native XML databases will probably support the referential integrity of internal pointers that use standard mechanisms. Many native XML databases will probably support external pointers of some sort, as well as the integrity of external pointers that point to resources stored in the database. For the moment, though, it is up to applications to enforce the integrity of pointers -- both internal and external -- in most cases.

6.3.4.3 Scalability

Scalability is well outside my area of expertise, so most of what follows is speculation. In general, my guess is that native XML databases will scale very well in some environments and very poorly in others.

Like hierarchical and relational databases, native XML databases use indexes as a way to initially find data. This means that locating documents and document fragments is related solely to index size, not to document size or the number of documents, and that native XML databases can locate the start of a document or fragment as fast as other databases using the same indexing technology. Thus, native XML databases will scale as well as other databases in this respect.

Like hierarchical databases and unlike relational databases, many native XML databases physically link related data. In particular, text-based native XML databases physically group related data together and model-based native XML databases that use proprietary storage systems often use physical pointers to group related data. (Model-based native

XML databases built on relational databases, like relational databases themselves, use logical links.)

Because physical links are faster to navigate than logical links, native XML databases, like hierarchical databases, can retrieve data more quickly than relational databases. Thus, they should scale well with respect to retrieving data. In fact, they should scale even better than relational databases in this respect, since scalability is related to a single, initial index lookup rather than the multiple lookups required by a relational database. (It should be noted that relational databases also offer physical linking of data, in the form of clustered indexes. However, such linking applies to individual tables rather than an entire hierarchy.)

Unfortunately, this scalability is limited. Like hierarchical databases, the physical wiring in native XML databases applies only to a particular hierarchy. That is, retrieving data in the hierarchy in which it is stored is very quick, but retrieving the same data in a different hierarchy is not. For example, suppose customer information is stored inside each sales order document. Retrieving sales order documents will be very fast, since this is the order in which the data is stored. Retrieving a different view of the data, such as a list of the sales orders for a given customer, will be much slower, since the physical links no longer apply.

To alleviate this problem, native XML databases make heavy use of indexes, often indexing all elements and attributes. While this helps decrease retrieval time, it increases update time, since maintaining such indexes can be expensive. This is not a problem in read-only environments, but might cause problems in heavily transactional environments.

Finally, native XML databases scale much more poorly than relational databases in searching for unindexed data. Both native XML databases and relational databases must search the data linearly in this case, but the situation is far worse in native XML databases due to less complete normalization. For example, suppose you want to search for all sales orders with a given date and that dates are not indexed. In a relational database, this means reading all the values in the OrderDate column. In a native XML database, this means reading each sales order document in its entirety and looking for <OrderDate> elements. Not only are the contents of each <OrderDate> element read, the contents of all of the other elements are read as well. Worse yet, in text-based native XML databases, the text must be parsed and converted to date format before it can be compared to the target date.

So, will scalability be a problem for you? It really depends on your application. If your application usually needs the data in the view in which it is stored, then a native XML database should scale well. This is the case for many document-centric documents. For example, the documents comprising a product manual will almost always be retrieved in their entirety. On the other hand, if your application has no preferred view of the data, then scalability may be a problem.

This concludes our description of native XML databases. In the next two sections, we will look at two specialized types of native XML databases: persistent DOMs and content management systems.

6.4 Persistent DOMs (PDOMs)

A *persistent DOM* or *PDOM* is a specialized type of native XML database which implements the DOM over some sort of persistent storage. Unlike most native XML databases, which can return documents as DOM trees, the DOM tree returned by a PDOM is live. That is, changes made to the DOM tree are reflected directly in the database. (Whether changes are actually made immediately or as a result of a call to a commit method depends on the database.) In most native XML databases, the DOM tree returned to the application is a copy, and changes are made to the database either through an XML update language or by replacing the entire document.

Because PDOM trees are live, the database is usually local. That is, it is in the same process space as the application, or at least on the same machine, although this is not necessarily required. This is done for efficiency, as a PDOM over a remote database would require frequent calls to the remote server.

PDOMs serve the same role for DOM applications that object-oriented databases serve for object-oriented applications: they provide persistent storage for the application data, as well as serving as virtual memory for the application. The latter role is particularly important for DOM applications that operate on large documents, as DOM trees can easily exceed XML documents in size by a factor of 10. (The actual factor depends on the average size of the text in the document, with documents containing small average text sizes having much higher factors.)

6.5 Content Management Systems

Content management systems are another specialized type of native XML database. They are designed for managing human-written documents, such as user's manuals and white papers, and are built on top of native XML databases. The database is generally hidden from the user behind a front end that provides features such as:

- Version and access control
- Search engines
- XML/SGML editors
- Publishing engines, such as to paper, CD, or the Web
- Separation of content and style
- Extensibility through scripting or programming
- Integration of database data

The term content management system, as opposed to *document management system*, reflects the fact that such systems generally allow you to break your documents into discrete content fragments, such as examples, procedures, chapters, or sidebars, as well as metadata, such as author names, revision dates, and document numbers, rather than having to manage each document as a whole. Not only does this simplify such things as how to coordinate the work of multiple writers working on the same document, it allows you to assemble entirely new documents from existing components.

7. Conclusion

NXDs aren't a panacea and they're definitely not intended to replace existing database systems. They're simply another tool for the XML developers' tool chest, and when applied in the right circumstances they can yield significant benefits. If you have lots of XML data to store, then an NXD is worth a look, and might just prove to be the right tool for the job.

.