AN ANALYSIS OF THE POWER OF XML-ENABLED AND NATIVE XML DATABASES

by

Harris Zafar

An honor's thesis submitted in partial fulfillment of the requirements for the degree of

B.S. Computer and Information Science with Honors

University of Oregon

March 23, 2001

University of Oregon

Abstract

AN ANALYSIS OF THE POWER OF XML-ENABLED AND NATIVE XML DATABASES

by Harris Zafar

Faculty Supervisor: Professor Chris Wilson
Department of Computer & Information Science

A thesis presented on the use of XML in the database industry, making a comparison between databases capable of handling XML and databases that store XML in its native form. This paper takes an in-depth look at the differences between native XML databases and XML-enabled databases in terms of their structures, how they store XML documents, and how XML is retrieved from these databases.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

Indeed, the Extensible Markup Language (XML) is emerging as the format of choice for a various types of data, especially documents. With its ability to tag different fields, XML makes searching simpler and more dynamic. Since XML content is free from presentation format - which independent style sheets specify - XML enables the extensive reuse of information. This allows enterprises to turn the same content into press releases, white papers, brochures, presentations, and Web pages. For enterprises trying to meld incompatible systems, XML can serve as a common transport technology for moving data around in a system-neutral format. In addition, XML is able manage all types of data, including text, images and sound, and it is user-extensible to handle anything special.

Until now, the problem has existed as to how the XML-tagged data can be managed. One solution that shows potential is the use databases to store, retrieve, and manipulate XML. The idea is to place the XML data in a framework where searching, analysis, updating, and output can proceed in a more manageable, orderly, and well-understood environment. Databases are advantageous since users are familiar with them and their behavior.

However, there are many types of databases. XML can be stored in three types of databases [30]:

1. **Native XML Database (NXD)** - A database fundamentally designed to store and manipulate XML data. Data access is through XML and related standards only (i.e. XPath, XSL-T, DOM or SAX). This includes all databases where the underlying data representation maintains the full XML structure and associated meta-data. Access to the data storage by means other than XML related technologies is not allowed. The

fundamental unit of storage in an NXD is an XML document. Tamino, dbXML and X-Hive are all database products that fall into this category.

2. **XML Enabled Database (XEDB) -** A database that has an added XML mapping layer (provided by the database vendor or a third party) that manages the storage and retrieval of XML data. Data that is mapped into the database is mapped into application specific formats, and the original XML metadata and structure may be lost. Data retrieved as XML is not guaranteed to have originated in XML form. Data manipulation may occur via either XML specific technologies or other database technologies (i.e. SQL). The fundamental unit of storage in an XEDB depends on its implementation. The XML solutions from Oracle and Microsoft, as well as many third party tools, fall into this category.

3. **Hybrid XML Database (HXD) -** A database that supports native XML storage as well as an XML mapping layer over non-XML data. Databases such as Excelon and Ozone fall into this category. The important thing to note is that there are different APIs to access the objects stored in Ozone. XPath, DOM, or SAX can be used directly on XML data or through a mapping layer on non-XML data. On the other hand, the API of the non-XML objects can also be used to access the data.

So which one is better to use? There really is no concrete answer as to what database is "better." There are XML-enabled databases that handle XML fine and that are based on tried-and-true relational or object-oriented models. These databases typically accept XML, parse it into chunks that fit the database schema, and then store it as usual. To retrieve XML, the chunks are pieced back together again and returned.

This document takes a look at the typical XML-enabled relational database paradigm and also at the new native XML database paradigm. The aim is to show how each handles XML data not only to show how each differ from one another, but also to display the powerful combination XML technology makes with database technology.

Chapter 1

XML BASICS

1.1 WHAT IS XML?

The eXtensible Markup Language (XML) is a meta-markup language that provides a format for describing structured data. Although called a language, XML is really a tool for defining markup languages, where a markup language is a set of tags and attributes with various constraints on them. One can use XML to design miniature markup languages of their own that are tailored to specific problem domains. For instance, XML has been used to define WML (Wireless Markup Language) which is the language used in WAP-communications.

On the surface, XML looks like HTML. Both are derived from the Standard Generalized Markup Language (SGML), but XML is different from HTML in syntax and semantics. It is incorrect to say that XML is another version of HTML. HTML is a fixed markup language based on ISO 8879:1986 SGML, whereas XML is based on the ISO standard language ISO 8879:1988 SGML. This is important to note not only because it shows that XML is not HTML, but also because it means that XML, like SQL, is not vendor-specific, and hence is more likely to remain in use for a long time.

Like HTML, XML makes use of elements and attributes. However, where HTML specifies what each tag and attribute means, XML uses the tags only to delimit pieces of data, and leaves the interpretation of the data completely to the application that reads it.

1.2 XML Elements and Attributes

An XML element is made up of a start tag, an end tag, and data in between. The start and end tags describe the data within the tags, which is considered the value of the element. For example, the following XML element is a <professor> element with the value "Joe Smith."

```
<professor>Joe Smith</professor>
```

The element name "professor" allows you to mark up the value "Joe Smith" semantically, so you can differentiate that particular bit of data from another, similar bit of data. For example, there might be another element with the value "Joe Smith."

```
<student>Joe Smith</student>
```

Because each element has a different tag name, you can easily tell that one element refers to Joe Smith, the professor, while the other refers to Joes Smith, the student. If there were no way to mark up the data semantically, having two elements with the same value might cause confusion.

In addition, XML tags are case-sensitive, so the following are each a different element.

```
<City> <CITY> <city>
```

An element can optionally contain one or more attributes. An attribute is a name-value pair separated by an equal sign (=).

```
<CITY zip="97403">Eugene</CITY>
```

In this example, **zip="97403"** is an attribute of the <CITY> element. Attributes are used to attach additional, secondary information to an element, usually meta information. Attributes can also accept default values, while elements cannot. Each attribute of an element can be specified only once, but in any order.

1.3 XML Rules

XML files are text files but are not meant to be read by humans. The reason they are text files is to allow programmers to more easily debug applications. The rules, however, of XML are much stricter than HTML. Errors in XML syntax halt document processing, and users or applications receive error messages instead of a best-guess interpretation of the document structure. For instance, a forgotten end-tag or an attribute without quotes makes the file unusable, while in HTML such practices are tolerated.

XML documents must follow rules for identifying document parts and creating nested element structures. Elements in XML documents cannot overlap. This means that if an XML element's start tag appears within another element, it must end within that same containing element as well. For example, the following HTML code combines bold and italic formatting by overlapping the structures:

```
<b>This is bold. <i>This is bold italic.</b> This is italic.</i>
```

In some HTML browsers, this text would appear as :

This is bold. ***This is bold italic.*** *This is italic.*

In an XML parser, however, all processing halts as soon as the parser comes across </b> since the parser is looking for </i> and will not accept </b>. To achieve the same formatting in XML, one would use the following:

```
<b>This is bold. </b> <i><b>This is bold italic.</b> This is italic.</i>
```

This extra work results in a leap forward for interoperability. Since this creates far less "guessing" code for the XML processor, it fits more easily in smaller-scale processing, like embedded systems. Structural ambiguities are eliminated from XML documents since all XML parsers will see the same nested element structure.

1.6 XML Schema vs. DTD

Although XML is intolerant about syntax, it offers developers more alternatives for defining meaning in XML documents. With XML, we can create our own markup vocabulary or choose from a wide assortment of markup vocabularies suitable to our industry or project type. Schemas and document type definitions (DTDs) give us the power to describe these vocabularies, but we can also create documents using vocabularies without formal definitions. Schemas and DTDs formally identify the relationships between the various elements that form the document. For example, let's take the following XML code segment:

```
<memo>
        <to>All faculty</to>
        <from>Jon Doe</from>
        <date>28th January</date>
        <subject>Faculty Meeting</subject>
        <para>Don't forget about the meeting tomorrow at noon.</para>
</memo>
```

For this simple memo, the XML DTD might look like this:

```
<!DOCTYPE memo [
<!ELEMENT memo    (to, from, date, subject?, para+) >
<!ELEMENT para    (#PCDATA) >
<!ELEMENT to      (#PCDATA) >
<!ELEMENT from    (#PCDATA) >
<!ELEMENT date    (#PCDATA) >
<!ELEMENT subject (#PCDATA) >
]>
```

This DTD tells the computer that a memo consists of a sequence of header elements, <to>, <from>, <date> and, optionally, <subject>, which must be followed by the contents of the memo. The content of the memo defined in this simple example is made up of a number of paragraphs, at least one of which must be present (this is indicated by the + immediately after para). If the + was replaced by a *, this would mean that we would have a set with zero or more elements. In this basic example, a paragraph has been defined as a leaf node that can contain parsed character data (#PCDATA), i.e. data that has been checked to ensure that it contains no unrecognized markup strings. Similarly, the <to>, <from>, <date> and <subject> elements have been declared to be leaf nodes in the document structure tree.

For this simple memo, the XML Schema might look like this:

```
<Schema name="memoSchema" xmlns="urn:schemas-microsoft-com:xml-data"
  xmlns:dt="urn:schemas-microsoft-com:datatypes">
        <ElementType name="date" content="textOnly" dt:type="string"/>
        <ElementType name="from" content="textOnly" dt:type="string"/>
        <ElementType name="memo" content="eltOnly" order="seq">
                <element type="to" minOccurs="1" maxOccurs="1"/>
                <element type="from" minOccurs="1" maxOccurs="1"/>
                <element type="date" minOccurs="1" maxOccurs="1"/>
                <element type="subject" minOccurs="0" maxOccurs="1"/>
                <element type="para" minOccurs="1" maxOccurs="*"/>
        </ElementType>
        <ElementType name="para" content="textOnly" dt:type="string"/>
        <ElementType name="subject" content="textOnly" dt:type="string"/>
        <ElementType name="to" content="textOnly" dt:type="string"/>
</Schema>
```

The schema begins with the <Schema> element containing the declaration of the schema namespace and, in this case, the declaration of the "datatypes" namespace as well. (An XML namespace is a collection of names that can be used as element or attribute names in an XML document. The namespace qualifies element names uniquely on the Web in order to avoid conflicts between elements with the same name.) The first, 'xmlns="urn:schemas-microsoft-com:xml-data",' indicates that this XML document is an XML Schema. The second, 'xmlns:dt="urn:schemas-microsoft-com:datatypes",' allows you to type element and attribute content by using the *dt* prefix on the **type** attribute within their ElementType and AttributeType declarations.

XML Schema is an XML-based syntax for defining how an XML document is marked up. The Schema language, which is itself represented in XML 1.0 and uses namespaces, substantially reconstructs and considerably extends the capabilities found in XML 1.0 document type definitions (DTDs). DTDs have many drawbacks, including the use of non-XML syntax, no support for datatyping, and non-extensibility. For example, DTDs do not allow you to define element content as anything other than another element or a string. For more information about DTDs, see the Worldwide Web Consortium (W3C) XML Recommendation. XML Schema improves upon DTDs in several ways, including the use of XML syntax, and support for datatyping and namespaces. For example, an XML Schema allows you to specify an element as an integer, a float, a Boolean, a URL, and so on. XML Schemas can become hard to read but they can be parsed within any XML tool. (Another draw back of a DTD is that it requires a special editor for this.)

Chapter 2

RELATED XML TECHNOLOGIES

2.1 XPATH

XPath is used by both XSLT and by XPointer. XPath is the XML Path Language for addressing (referring to) parts of an XML document [1]. The XPath Recommendation is a joint work representing the combined efforts of the XSL Working Group and the XML Linking Working Group. XPath was created in order to provide a common syntax and semantics for querying and addressing the contents of XML documents that could be used by XSLT (XSL Transformation Language), XLink, and XPointer.

XPath gets its name through its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document. It operates on the abstract, logical structure of an XML document as a tree of nodes. XPath uses "expressions" that are evaluated into one of four basic datatypes with respect to a certain "context". The way that a context is determined differs for XSLT and XPointer. Both XSLT and XPointer extend XPaths "core functions", which are defined by its Core Function Library.

As stated, the primary purpose of XPath is to address parts of an XML document. In support of this primary purpose, it also provides basic facilities for manipulation of strings, numbers and booleans. XPath uses a compact, non-XML syntax to help with the use of XPath within URIs and XML attribute values. XPath operates on the abstract, logical structure of an XML document, rather than its surface syntax. XPath gets its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document [36].

XPath models an XML document as a tree of nodes. There are different types of nodes, including element nodes, attribute nodes, and text nodes. XPath defines a way to compute a string-value for each type of node. The primary syntactic construct in XPath is the expression, which is evaluated to yield an object, which has one of the following four basic types:

- node-set (an unordered collection of nodes without duplicates)
- boolean (true or false)
- number (a floating-point number)
- string (a sequence of UCS characters)

The following is a simple example XPath expression. This example returns the fifth scene in the second act of a screenplay [1].

```
/screenplay/act[2]/scene[5]
```

2.2 XPointer

XPointer is the language used as a fragment identifier for any URI reference that locates a resource of Internet media type text/xml, application/xml, text/xml-external-parsed-entity, or application/xml-external-parsed-entity [37]. XPointer, which is based on the XML Path Language (XPath), supports addressing into the internal structures of XML documents. It allows for examination of a hierarchical document structure and choice of its internal parts based on various properties, such as element types, attribute values, character content, and relative position. In particular, it provides for specific reference to elements, character strings, and other parts of XML documents, whether or not they bear an explicit ID attribute.

XPointer's extensions to XPath allow it to:

- Address points and ranges as well as whole nodes
- Locate information by string matching
- Use addressing expressions in URI references as fragment identifiers (after suitable escaping)

The following is an example of an XPointer command [1]:

http://www.holoweb.net/~liam/xmldocs#xpointer(id(simon12))

Here, the *#xpointer(id(simon12))* part is a fragment using XPointer. The actual syntax of the fragment identifier (the # and everything following it) is defined by XPath. If multiple xpointer( ) expressions are given, they are evaluated from left to right. This example could have been written using an HTML-style XPointer:

http://www.holoweb.net/~liam/xmldocs#simon12

2.3 XLINK

XLink is an XML-based approach to link between documents. Currently, there is little software available to support XLink directly, but XLink is still a good way of transporting link information between databases[1]. XLink allows elements to be inserted into XML documents in order to create and describe links between resources. XLink uses XML syntax to create structures that can describe both HTML-style unidirectional hyperlinks and much more powerful multidirectional linking constructs. It allows XML documents to [38]:

1. Assert linking relationships among more than two resources
2. Associate metadata with a link
3. Express links that reside in a location separate from the linked resources

XLink works by proving you with global attributes you can use to mark your elements as linking elements. In order to use linking elements, the declaration of the XLink namespace is required. An example of this is as follows [1]:

```
<bibliography xmlns:xlink="http://www.w3.org/1999/xlink">
    --- Content that uses links ---
</bibliography>
```

This example allows the content of the *bibliography* element to contain XLink hypertext links. To enable XLink in the entire document, the *xmlns:xlink* attribute would be put on the outermost document element desired.

2.4 XSLT

XSLT is a language for transforming XML documents into other XML documents. XSLT is designed for use as part of XSL, which is a style sheet language for XML. In addition to XSLT, XSL includes an XML vocabulary for specifying formatting. XSL specifies the styling of an XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary [39].

XSLT is also designed to be used independently of XSL. However, XSLT is not intended as a completely general-purpose XML transformation language. Rather, it is designed primarily for the kinds of transformations that are needed when XSLT is used as part of XSL.

A transformation expressed in XSLT describes rules for transforming a source tree into a result tree. The transformation is achieved by associating patterns with templates. A pattern is matched against elements in the source tree. A template is instantiated to create part of the result tree. The result tree is separate from the source tree. The structure of the result tree can be completely different from the structure of the source tree. In constructing the result tree, elements from the source tree can be filtered and reordered, and arbitrary structure can be added[39].

XSLT meets the need to be able to transform information marked up in XML from one vocabulary to another. It is a powerful implementation of a tree-oriented transformation language for transmuting instances of XML using one vocabulary into either simple text, the legacy HTML vocabulary, or XML instances using any other vocabulary imaginable [40].

Chapter 3

RELATIONAL DATABASES

3.1 What is an RDBMS?

Relational databases consist of a set of *tables*, where each table is a set of *records*. A record, in turn, is a set of *fields*, and each field is a pair *field-name/field-value*. All records in a particular table have the same number of fields with the same field-names. A relational database contains both data and metadata, that is, both data and information about data. The information is of three kinds:

1. Information about the entire Relational Database Management System (RDBMS)
2. Information about a specific database
3. Information about the columns of a result set

Conceptually, the relational model is very simple: a set of tables that one operates into and gets another set of tables as the result. However, it is very inefficient to scan through the tables. During the early days of relational databases, many people said they were too slow to be of any practical use. At that time, people were used to navigating though ISAM (Indexed Sequential Access Method) files or a linked structure such as IMS (Information Management System).

A fundamental benefit of relational databases is the concept of *views* [28]. Database views allow developers to present data in any number of "logical" combinations, hiding any details of the underlying physical storage. In effect, views *transform* the structure of one or more underlying tables into a more useful or appropriate structure for the demands of a specific application. Since views can be based on other views, this mechanism offers a tremendously powerful, layered mechanism for presenting information in any way that is appropriate for the task at hand.

Today, relational databases employ a series of techniques to improve performance, and most are based on the use of indexes. The most common type of index is the B-Tree and variants, the same used by DBF files. Some databases may use hash tables, bitmaps and other data structures, but the common point is that the index can speed the search for a particular row or the sorting of a set of rows.

Yet, simply creating one or more indexes may not help performance, as indexes can actually degrade performance when not used the right way. An index means more writes to the disk and a bigger log for databases that implements transactions.

Most databases have some kind of query optimizer that chooses the best path to get the data that satisfies a given query. The query optimizer may not use any index at all, if it thinks it will be necessary to scan most of the table.

An index may considerably speed up joins, ORDER BY and GROUP BY clauses from a SELECT statement. It may also speed up many queries, if some conditions match the columns and sort order of the index.

3.2 XML AND RDBMS

XML is valuable in that it is a universal data format that is text-based, is easily parsed, and enables interoperability. Once converted into an XML data source, the data within the relational database can be easily accessed and manipulated by other applications and by HTML pages. Perhaps one of the most important things XML allows us to do is that it provides a simple mechanism for data exchange between organizations, applications, and databases. The technological elegance of being able to send a packet of data that is both human readable and self-describing will ensure it will make invasions in areas where EDI never could. Because of that, XML can accelerate the entire B2B revolution that is just starting now.

Kevin William, one of the authors of Professional XML and Professional ASP XML, had this to say in regards to how XML fits in with relational databases:

> *As a long-time relational database developer, one of the greatest challenges I have had to face is that of data import and export. XML allows information to be transferred in a standardized, human-readable format that incorporates structure. Rather than writing dozens of customized data transformations, each company participating in data transfer can write one - to and from the common XML format.*

But is XML a database? An important thing to note is how XML differs from the traditional relational database. The main ways in which an XML document differs from the sort of data one would find in a typical relational database include the fact that fields and data in an XML document are normally intermixed (mixed content). In an XML document, fields do not have length restrictions, they can nest arbitrarily, and they have a sequence. To understand the implications of arbitrary field nesting, take a look at some examples. A simple way to look at this is with an HTML-like example:

```
<ul>
   <li><p>First item</p></li>
   <li><p>Second item</p></li>
   <li>
      <ol>
         <li><p>This is an li within an ol within an li</p></li>
         <li><p>Here's another one</p></li>
      </ol>
   </li>
</ul>
```

There could be hundreds of nested *ul*, *ol*, or *li* elements in there, and it would all still be legal. Since this is the fundamental nature of XML, if one is storing XML documents in a database, they probably just have to live with it. Recursion like this is very common in XML. As a result, there is still a fundamental disconnect between the flat field in the relational world and the complex nesting field in XML.

XML fields also have a sequence. For example, a chapter may contain a title, followed by several paragraphs of text. If one stores the chapter in a database and then extracts it, they want to ensure that the paragraphs reappear in the correct order. This may seem quite obvious, but to implement it, one might end up giving each paragraph a sequence number and using an ORDER BY clause or an SQL cursor, which is difficult to do efficiently. Some would choose to offload some of this work onto the author by assigning each paragraph a required Sequence Number attribute. However, what if the author wants to reverse the order of two paragraphs, or copy part of a chapter and edit it? That author would probably forget to change the attributes. Sorting items in the database client can be one good way of splitting up the work so that the database server does not grind to a halt. This all may sound a bit drastic, but consider that most relational databases were not really designed for this sort of data and generally work best with small data items (such as integers) rather than paragraphs.

Conversely, there are times when one would rather use a relational database instead of a text XML document. As mentioned above, an XML document and a relational database are not the same. Both are much better choices than flat text files. There are five major problems with flat files, one of which is brilliantly solved by XML and four by an RDBMS. XML eliminates the need to invent a file format for each application and the need to write a parser for this format. As for an RDBMS, it addresses the following four problems [14]:

1. **Size/cache.** As the size of the document (or the number of related documents) increases, one must implement some sort of cache because large DOM trees cannot be kept in RAM and users will not wait for an XML parser to read the file from beginning to end each time. Fragmentation may be an alternative to caching, but here too, much custom code must be written to keep these fragments in order.

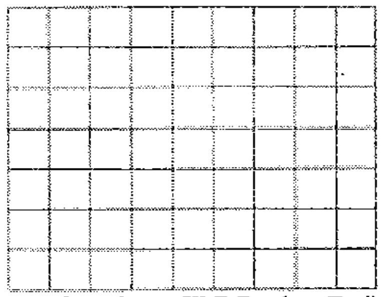
2. **Concurrent Access.** If more than one user is making changes to the document(s), some mechanism (software or convention) must exist to ensure that only one user makes changes to the document at any given moment.

3. **Transactional Integrity.** In a networked environment, it must be ensured that any request is either processed fully, or not at all. This means that one must be able to roll back a transaction.

4. **Security and administration.** An administrator's control over who can read, write and update what parts of an XML document are very limited.

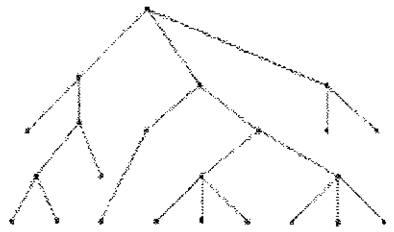
3.3 Mapping XML Data To Relational Data

XML information must be flattened to be in a relational model. To illustrate the difference between relational data and XML data, consider the following figures. Figure 1 shows a relational table as a rectangular grid. Figure 2 shows that an XML document is a tree, and Figure 3 shows how the tree is not a rectangle.



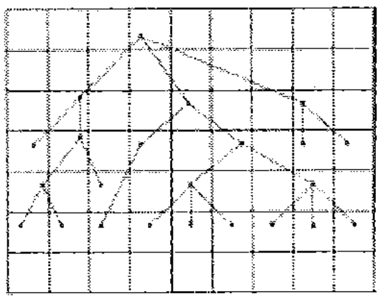
Source: Open Source XML Database Toolkit. (NY: John Wiley & Sons, 2000), p.166

Figure 1 Relational Data Model



Source: Open Source XML Database Toolkit. (NY: John Wiley & Sons, 2000), p.166

Figure 2 XML Document Structure



Source: Open Source XML Database Toolkit. (NY: John Wiley & Sons, 2000), p.166

Figure 3 XML Document Stored in a RDBMS

With data stored in XML documents, it should be possible to query the contents of these documents. Ideally, one should be able to issue queries over sets of XML documents to remove, combine, and analyze their contents.

An XML document is an example of a *semi-structured* data set. It is tree-structured, with each node in the tree described by a label. Hence, semi-structured query languages and query evaluation techniques can be used to issue these queries over the XML documents (which is covered in more depth in Chapter 3). While this approach is feasible, one could very possibly find other approaches that can even leverage relational technology to provide query capability over XML documents. It is, indeed, possible to use standard commercial relational database systems to evaluate powerful queries over XML documents, with the existence of a DTD or XML Schema.

How to store XML data/documents in a relational database really depends on how flexible one wants to be. Intra Extra Digital (http://www.iedigital.net) is a company located in London, which uses the following technique to store XML data into a relational database. They use a simple technique in which they have a table for attributes and a table for elements. The attribute table has a join on the element table to say what element the attribute belongs to, while the element has joins to itself to say whom the parent of an element is. This allows them to store an object-like tree structure and generate XML documents from any point in the tree.

Relational database systems are mature and scale very well, and they have the additional advantage that in a relational database, XML data and traditional (structured) data can co-exist, making it possible to build applications that involve both kinds of data with little extra effort. Relational databases, however, have been built to support traditional (structured) data and the requirements of processing XML data are vastly different from the requirements to process such traditional data. To optimize the use of relational database systems for XML, recent work has concentrated on models and algorithms to extract the schema from XML. The goal of that work is to analyze the semi-structured data and (possibly) the query workload of the target application in order to find the best-approximated schema. This way, the XML data can be stored in the relational database with little off cuts, and schema extraction makes it also easier to formulate queries

There are several ways to approach the problem of querying XML documents using relational databases. One approach is to process the DTD (or XML Schema) to generate relational schema. Then, the XML documents can be parsed (conforming to DTDs) and loaded into tuples of relational tables in a standard commercial DBMS. Next, the semi-structured queries (such as XML-QL or XQL) can be translated over XML documents into SQL queries over the corresponding relational data. Finally, the results can then be converted back to XML.

It is important to note that there are a number of limitations in current relational database systems that, in some instances, make using relational technology for XML queries either awkward or inefficient. Relational technology proves awkward for queries that require complex XML constructs in their results and may be inefficient when fragmentation, due to the handling of set-valued attributes and sharing, causes too many joins in the evaluation of simple queries.

Traditionally, relational schemas have been derived from a data model such as the Entity-Relational model. This translation is straightforward since there is a clear separation between entities and their attributes. Each entity and its attributes are mapped to a relation. When converting an XML Schema (or DTD) to relations, it may be temping to map each element in the Schema to a relation and map the attributes of the element to attributes of the relation. However, there is no correspondence between elements and attributes of the Schema and entities and attributes of the ER-model. What would be considered 'attributes' in an ER-

Model are often most naturally represented as elements in an XML Schema. Thus, directly mapping elements to relations is likely to lead to excessive fragmentation of the document.

3.3.1 Two Inlining Techniques

In their article ([17]), the people at the University of Wisconsin-Madison present two techniques for storing XML documents in a relational database system. They describe the Basic Inlining Technique, the Shared Inlining Technique, and then explain a hybrid of these two. The Basic Inlining Technique solves the fragmentation problem by inlining as many descendants of an element as possible into a single relation. However, this basic technique creates relations for *every* element because an XML document can be rooted at any element in the Schema or DTD. This technique was developed by examining DTDs instead of XML Schemas. Let us refer to the following extract from an example XML document and its related DTD presented in this article [17] to illustrate this technique:

```
<book>
    <title>My XML Book</title>
    <author id= "doe">
        <name>
            <firstname>John</firstname>
            <lastname>Doe</lastname>
        </name>
        <address>
            <city>Eugene</city>
            <state>OR</state>
            <zip>97403</zip>
        </address>
    </author>
</book>
```

```
<!ELEMENT book   (booktitle, author)
<!ELEMENT article   (title, author*, contactauthor) >
<!ELEMENT contactauthor  EMPTY>
<!ATTLIST contactauthor authorID IDREF IMPLIED >
<!ELEMENT author  (name, address) >
<!ELEMENT name  (firstname?, lastname) >
<!ELEMENT firstname (#PCDATA) >
<!ELEMENT lastname (#PCDATA) >
<!ELEMENT address ANY >
```

Using this Basic Inlining technique, the *author* element would be mapped to a relation with attributes *firstname*, *lastname*, and *address*. In addition, relations would be created for *firstname*, *lastname*, and *address*. The researchers at the University of Wisconsin-Madison admit that they must address two complications: set-valued attributes and recursion. Looking at this previous example, when creating a relation for *article*, the set of authors cannot be inlined because the traditional relational model does not support set-valued attributes. Instead, what must be used is the standard technique for storing sets in a relational database by creating a relation for *author* and linking *authors* to *articles* using a foreign key.

Using inlining only would necessarily limit the level of nesting in the recursion. For that reason, the recursive relationship is expressed using the notion of relational keys and by using relational recursive processing to retrieve the relationship.

As a result of the Basic Inlining Technique, the XML document above would be converted to the following tuple in the book relation:

```
(1, My XML Book, John, Doe, <city>Eugene</city><state>OR</state><zip>97403</zip>, doe)
```

The ANY field, address, is stored as an un-interpreted string. Hence, the nested structure is not visible to the database system without further support for XML. One important thing to note here is that if John Doe is the author for multiple books, then the author information will be repeated for each book since that information is repeated in the corresponding XML documents.

The Basic Inlining Technique is good for only particular types of queries, such as a query to list all book authors. For other queries, though, the Basic Inlining Technique will be quite inefficient, such as a query to list all authors having the last name Doe. Such a query will have to be executed as the union of five separate queries. As pointed out before, one other great disadvantage of the Basic Inlining Technique is the large number of relations it creates.

The Shared Inlining Technique aims to solve the problems of the Basic Inlining Technique by ensuring that an element is represented by exactly one relation only [17]. The main idea behind this technique is to identify the elements that are represented in multiple relations by the Basic Inlining Technique (such as *firstname*, *lastname*, and *address* elements in the above example) and to share them by creating separate relations for these elements. One prominent feature of the Shared Inlining Technique is the small number of relations compared to the Basic Inlining Technique.

The element sharing in this technique has query processing implications. For example, a selection query over all authors accesses only one relation using the Shared Inlining Technique, whereas it would access five relations using the Basic Inlining Technique. Even though the Shared Inlining Technique addresses some of the shortcomings of the Basic Inlining Technique, and even shares some of its strengths, the Basic Technique still performs better in one important aspect. The Basic Inlining Technique is better in reducing the number of joins starting at a particular element. Hence, the researchers from the University of Wisconsin-Madison briefly mention the idea of a hybrid approach that combines the sharing features of the Shared Inlining Technique with the join reduction properties of the Basic Inlining Technique.

When storing data in the database, it is often acceptable to throw away much of the information about a document (such as its name and DTD) as well as its physical structure (such as entity definition and usage, the order in which attribute values and sibling elements occur, the way in which binary data is stored, CDATA sections, and encoding information). Similarly, when retrieving data from the database, the resulting XML document is likely to contain no CDATA or entity usage (other than the predefined entities lt, gt, amp, apos, and quot) and the order in which sibling elements and attributes appear is likely to be the order in which the data was returned by the database.

One consequence of ignoring information about the document and its physical structure is that storing the data from a document into the database and then reconstructing the document from that data (a process called "round-tripping") often results in a different document, even in the orthodox sense of the term. Whether this is acceptable depends on each individual's needs and might influence one's choice of database and data transfer middleware.

3.4 Generating XML Data From Relational Data

When mapping information from traditional (i.e. relational) data models to XML, there is no single right answer. Instead, there are a variety of possible methods. At one end is the custom mapping of individual pieces of information from the database into a "predetermined" schema (e.g. an industry standard interchange format). This method implies considerable custom code to execute the mappings between the two loosely coupled schema (that of the database and that of the predetermined schema).

At the other end is a 'data-dump" of the entire contents of a database with the intention of re-creating the database with all of its relationships and data intact. This method tends to yield a mechanical mapping onto a generic metadata schema. XML Metadata Interchange (XMI), promoted by the Object Management Group, is one such generic metadata schema. In this method, the schema is not about the business; it is about the metadata concepts themselves.

Both methods are common and helpful ways of modeling relational data. Both, however, have their drawbacks: the need for considerable custom code, and the complicating of the business concepts. Lee Buck, Chief Technology Officer of Extensibility Inc., set forth an approach that lies between these two extremes and applies it to relational databases [21]. This approach provides mechanical mappings of key database structure concepts while preserving the concept that the schema should be about the business information. This approach has the advantage of generating easily understandable schemas which fit with existing data structures and which leverage XML concepts such as ID/IDREF. This approach is not guaranteed to provide 100% reliability when round-tripping data from an RDBMS to XML back to an RDBMS, but when possible, Lee states these limitations are identified, along with probable workarounds.

Let's take a sample relational database to explain the methods. The following is an employee database (partially referenced from the *Advanced XML* article on the website of Extensibility, Inc) consisting of four tables to monitor employee performance and the employee's sales.

```
TABLE EMPLOYEE
    NUM            LONGINT     PRIMARY KEY
    FNAME          STRING 32
    LNAME          STRING 32
    HIRE_DATE      DATE
    TERM_DATE      DATE        MAY BE NULL
TABLE PERFORMANCE_REVIEW
    EMP_NUM        LONGINT     PRIMARY KEY   FOREIGN KEY
    REVIEW_DATE    DATE        PRIMARY KEY
    REVIEW         TEXT
TABLE SALE
    SALE_ID        INT         PRIMARY KEY
    EMP_NUM        LONGINT     FOREIGN KEY
    SALE_DATE      DATE
TABLE ITEM
    ITEM_ID        INT         PRIMARY KEY
    SALE_ID        INT         FOREIGN KEY
    PRICE          INT
```

There are multiple ways of simply expressing this data in XML form. A simple way of doing so is to begin by having an element named after the table. This element will correspond to a row of data with attributes containing the values of each of the columns. This schema is often called *attribute normal form.* So, an example of an XML document with information drawn from the above example may look like this:

```
<EMPLOYEE
   NUM = '12345'
   FNAME = 'Joe'
   LNAME = 'Smith'
   HIRE_DATE = '3/15/01' >
</EMPLOYEE>
```

Another simple way of expressing this data in XML form is to begin, again, by having an element named after the table, corresponding to a row of data with sub-elements containing the values of each of the column data. This schema is often referred to as *element normal form.*

An example of an XML document with the information drawn from this sample example may then look like this:

```
<EMPLOYEE>
    <NUM>12345</NUM>
    <FNAME>Joe</FNAME>
    <LNAME>Smith</LNAME>
    <HIRE_DATE>3/15/01</HIRE_DATE>
</EMPLOYEE>
```

As noted by Lee Buck of Extensibility Inc [21], although this may be a good start at representing our relational data in XML form, there are several essential enhancements that can be made to this model:

1. Support for datatypes information
2. Retention of key relationships
3. Leveraging of XML's ID/IDREF facilities

Traditional data sources are inclined to be strongly typed. While an XML document, by definition, will represent its information in textual form, maintaining information about the underlying data type is critical. While a complete list of datatypes is an intangible goal, a practical set has been collected from the various schema submissions to the World Wide Consortium (W3C). This set includes string, boolean, int, float, number, date, time, and dateTime. Such datatyping ability is not included in a DTD. Therefore, to have support for datatypes information, one would use an XML Schema, as discussed in Chapter 1.

Much of the power and complexity of mapping data to XML comes from mapping the relationships between pieces of data. In the relational world, this equates to the modeling of primary-foreign key relationships.

A primary key offers a unique value by which a single row of a particular table may be accessed. XML has a comparable concept of an ID attribute, which provides unique access to an element. Many implementations of the Document Object Model (DOM) provide indexed access to such elements, which makes leveraging the similarities often desirable. The problem, however, is that XML IDs must be unique across the entire document, while a primary key is unique only within that column.

To provide the necessary global uniqueness, a supplemental *pkeyID* attribute can be used to hold a version of the primary key data that is made globally unique. Doing so takes advantage of two facts: 1) the primary key is unique within the context of the element type that contains each row (typically named after the table itself), and 2) the element type's name is unique within the document. Hence, in order to make the locally unique name globally unique, the key value is assigned the element name. For example:

```
<EMPLOYEE pkeyID = "EMPLOYEE.12345">
    <NUM>12345</NUM>
    <FNAME>Joe</FNAME>
    <LNAME>Smith</LNAME>
    <HIRE_DATE>3/15/01</HIRE_DATE>
</EMPLOYEE>
```

The existence of foreign keys within a table is what binds different tables together. Just as XML's ID concept presented a useful counterpart for primary keys, so too its IDREF concept provides a powerful counterpart to foreign keys. Of course, issues of uniqueness scope intervene here as well. A similar technique is used to accommodate them. A new IDREF attribute is created for the table element whose name is derived from the column name (with an _IDref appended). Values in a document will be constructed in a similar fashion as for *pkeyID* attributes. For example:

```
<EMPLOYEE pkeyID = "EMPLOYEE.12345">
   <NUM>12345</NUM>
...
<PERFORMACE_REVIEW  EMP_NUM_IDref = "EMPLOYEE.12345">
   <EMP_NUM>12345</EMP_NUM>
...
```

There is an alternative to the method described above in some cases. This technique takes advantage of the fact that, in XML, relationships may be deduced from context (specifically containment) as well as through id/idref relationships. So, in the above example, PERFORMANCE_REVIEW elements can be associated with their corresponding EMPLOYEE elements by placing them inside the latter. For example:

```
<EMPLOYEE  NUM_id = "EMPLOYEE.12345"
   NUM = '12345'
   FNAME = 'Joe'
   LNAME = 'Smith'
   HIRE_DATE = '3/15/01'>
   <PERFORMANCE_REVIEW
      REVIEW_DATE = '4/15/01'
      REVIEW = 'bad' />
   <PERFORMANCE_REVIEW
      REVIEW_DATE = '5/15/01'
      REVIEW = 'better' />
</EMPLOYEE>
```

It should be noted that this is not always desirable and is often not even possible. The following conditions must be met for this to be appropriate:

1. The foreign key must not be nullable (i.e. optional).
2. It must be the only foreign key so modeled in the table.
3. Every desired row must refer to a row that will be included.
4. The foreign key must not point to the same table.

If XML is to realize its potential, some mechanism is needed to publish relational data as XML documents. Towards that goal, one of the major challenges is discovering a way to efficiently structure and tag data from one or more tables as a hierarchical XML document. Different alternatives are achievable, depending on when this processing occurs and how much of it is done inside the relational engine.

To publish relational data as XML documents, an implementation to efficiently carry out the conversion is needed. Given a language specification for converting relational tables to XML documents, an implementation to carry out the conversion raises many challenges. Relational tables are flat, while XML documents are tagged, hierarchical and graph-structured. What is the best way to go from the former to the latter? In order to answer this question, people at the IBM Almaden Research Center wrote an article entitled "Efficiently Publishing Relational Data as XML Documents," in which they characterize the set of alternatives based on whether tagging and structuring are done early or late in query processing. They then refine this set based on how much processing is done inside the relational engine and explore various alternatives within this space.

Comparing the alternatives using a commercial database system, they concluded that an "unsorted outer union" approach (based on late tagging and late structuring) is attractive when the resulting XML document fits in main memory, while a "sorted outer union" approach (based on late tagging and early structuring) performs well otherwise. The results also illustrated that constructing an XML document inside a relational engine is considerably more efficient than doing so outside the engine. Hence, constructing an XML document inside the relational engine has a two-fold advantage. It allows existing SQL APIs to be reused for XML documents, and it is also much more efficient.

In order to understand the numerous alternatives for publishing relational data as XML documents, the solution set is characterized based on the key differences between relational tables and XML documents. Specifically, XML documents have tags and nested structure, whereas relational tables do not. Hence, in converting from relational tables to XML documents, tags and structure must be added somewhere along the way. One approach is to do tagging as the final step of query processing (late tagging), while another approach is to do it earlier in the process (early tagging). Similarly, structuring can be done as the final step of query processing (late structuring) or it can be done earlier (early structuring). Each alternative in this set has variants depending on how much work is done inside the relational engine.

3.4.1 Unsorted Outer Union Approach

In the class of alternatives that postpone tagging and structuring, both tagging and structuring are done as the final step of constructing an XML document. Hence, the construction of an XML document is understandably split into two phases: (a) content creation, where relational data is produced, and (b) tagging and structuring, where the relational data is structured and tagged to produce the XML document. We first deal with content creation. We consider only "inside the engine" approaches so that database functionality, such as joins, can be exploited.

One simple approach to producing the needed content is by joining all of the source tables. In our example, this would be done by joining the Employee, Performance_Review, Sale, and Item tables. Note that the joins relate parents to their children. This approach has the benefit of using regular, set-oriented relational processing, but it also has a serious pitfall in that it has both content and processing redundancy. To see this, consider what the result of these joins would look like. Each employee's information would be repeated at least SA * IT times, where SA is the number of sales associated with the employee and IT is the number of items per sale. The problem is that multi-valued data dependencies are created when a hierarchical structure is represented as a single table. This not only increases the size of the result, but also the amount of processing to produce it, both of which are likely to severely impact performance.

To remedy this, the people at the IBM Almaden Research Center identified the unsorted outer union approach. They saw that the number of tuples in the relational result grows as the product of the number of children per parent. They discovered that if they could limit the result's size to be the sum of the number of children per parent, redundancy could be reduced. To achieve this, the representation of a given child of a parent must be separated from the representation of the other children of the same parent. For example, one tuple of the relational result should represent either a performance review or a sales order associated with the employee, not both.

To execute this approach, each path from the root-level table to the leaf-level table is computed using joins. In our example, there are two paths: Employee–Performance_Review and Employee–Sales–Item. Hence, Employee is joined with Performance_Review (one path) and Employee is joined with Sale, which in turn is joined with Item (second path). The computation is shown in Figure 4.

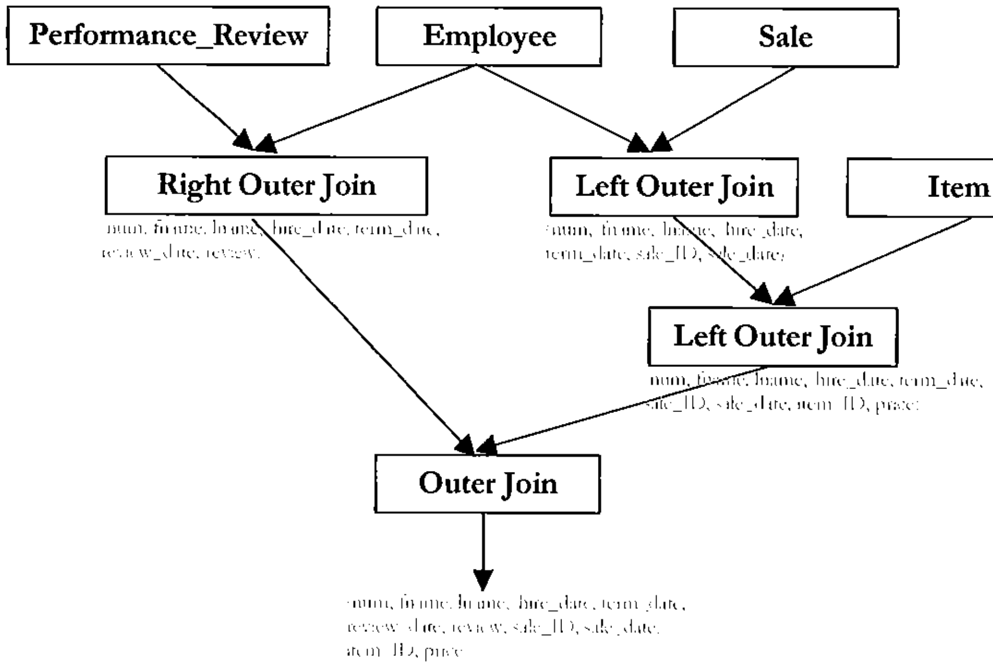

Performance_Review
Employee
Sale
Right Outer Join
num, fname, lname, hire_date, term_date, review_date, review
Left Outer Join
num, fname, lname, hire_date, term_date, sale_ID, sale_date
Item
Left Outer Join
num, fname, lname, hire_date, term_date, sale_ID, sale_date, item_ID, price
Outer Join
num, fname, lname, hire_date, term_date, review_date, review, sale_ID, sale_date, item_ID, price

Figure 4 Unsorted Outer Union Approach

Recall the use of outer joins. In an outer join between *table1* and *table2*, *table1* rows without a matching *table2* row appear exactly once in the result, with the result of columns inherited from *table2* assigned null values. In a left outer join, *table1* rows without a matching *table2* row appear in the result, but not vice versa. In a right outer join, *table2* rows without a matching *table1* row appear in the result, but not vice versa. In a full outer join (the last join in Figure 4), both *table1* and *table2* rows that do not have a match appear in the result.

The final step in the process of creating the relational content is to join together all the tuples representing leaf-level elements in the XML tree (by use of the outer union in Figure 4) into a single relation. The obvious way to do this is to union the content corresponding to each leaf level element. There is, however, some difficulty with this strategy since the tuples corresponding to different leaf-level elements are not required to have the same number or types of columns. In order to handle this variety, a separate column is allocated in the result of the union for each distinct column in the union's input. For each tuple representing a specific leaf-level element and its ancestors, only a subset of these columns will be used and the rest will be set to null.

To keep track of the source of each tuple (i.e. to differentiate a *performance_review* tuple from a *sale* tuple), a *type* column is added to the result of the outer union as well. This approach is called the Path Outer Union approach since it computes each path from the root-level table to a leaf-level table and outer unions them.

The Path Outer Union approach removes much of the data redundancy (and associated computation redundancy) of the Redundant Relation approach. This is because children of the same parent are represented in separate tuples. On the other hand, there still exists some data redundancy. In particular, parent information is repeated with every child of that parent (i.e. employee information is repeated with every sale). One way to get around this is to feed the parent information directly into the outer union operator and to carry only parent ids along with the children. This reduces data redundancy, but it increases the number of tuples in the result because each parent is now represented by a separate tuple. This approach is referred to as the Node Outer Union approach to distinguish it from the earlier Path Outer Union approach. One concern with the Outer Union approaches is that the number of columns in the result increases with the width and depth of the XML document.

Now that the relational content is constructed for the creation of the XML document, the final step is to tag and structure the results. If this is completed inside the relational engine, it can be executed as an aggregate function. Such a function would be invoked as the last processing step, after the relational content has been created, and this (single) aggregate function would logically put together all XML fragments.

In order to tag and structure the results, two things must be done: (1) all siblings in the desired XML document must be grouped under the same parent and (2) information from each tuple must be extracted and tagged to produce the XML result. An efficient way to group siblings is to use a main-memory hash table to look up the parent of a node, given the parent's type and id information (including the ids of the parent's ancestors). Thus, when a tuple containing information about an XML element is seen, it is hashed on the element's type and the ids of its ancestors in order to determine whether its parent is already present in the hash table. If the parent is present, a new XML element is created and added as a child of the parent. If the parent is not present, a hash is performed on the type and ids of all ancestors excluding the parent in order to determine if the grandparent exists. If the grandparent exists, the parent is created and then the child is created. If the grandparent is also not present, the procedure is repeated until an ancestor is present in the hash table or the root of the document is reached. After all the input tuples have been hashed, the entire tagged structured can be written out as an XML file.

The main drawback of using this hash-based tagging mechanism is that performance can degrade rapidly when there is inadequate memory to hold the hash table and the intermediate result.

3.4.2 Sorted Outer Union Approach

The main problem with the previous approach is that complex memory management must be performed in the hash-based tagger when memory is scarce. To eradicate this problem, the people at the IBM Almaden Research Center found that the relational engine could be used to produce "structured content", which can subsequently be tagged using a constant space tagger.

The key to structuring relational content is to order it the same way that it needs to appear in the result XML document. This can be achieved by ensuring that:

1) All information about a node in the XML tree occurs either before or along with the information about the children of that node in the XML tree. This essentially says that parent information occurs before, or with, child information.

2) All tuples representing information about a node and its descendants in an XML tree occur together. This ensures that information regarding a particular node and its descendants is not mixed in with information about non-descendant nodes.

3) The relative order of the tuples matches that of any user-specified order. This is to handle user defined ordering requests.

Performing one final relational sort of the unstructured relational content is enough to ensure these properties. To ensure conditions 1 and 2, all that is needed is to sort the result of the Node Outer Union on its ID fields, with the IDs of parent nodes occurring higher in the sort order than the IDs of children nodes. Hence, in Figure 4, sorting the result on the composite key (num, sale_ID, item_ID) will guarantee that the result is in document order. Tuples having null values in the sort fields must occur before tuples having non-null values, meaning that null values must sort low when in sorted order. Condition 1 is then satisfied since a tuple corresponding to a parent node (i.e. Employee) will have null values for the child ID columns (i.e. sale_ID). To ensure that tuples with null values in sorted columns occur first, parent tuples (employees) will always occur before child tuples (sales). Also, since the parent's ID occurs before a child's ID in the sort order, the children of a parent node are grouped together after the parent, thus satisfying condition 2.

The Sorted Outer Union approach has the advantage of scaling to large data volumes because relational database sorting is disk-friendly.

Once structured content is created, the next step is to tag and construct the resulting XML document. Since tuples arrive in document order, they can immediately be tagged and written out as they are seen. The tagger only needs memory to remember the parent IDs of the last tuple seen. These IDs are used to identify when all the children of a particular parent node have been seen so that the closing tag associated with the parent can be written out. For example, after all the items of a sales order have been seen, the closing tag for sales order (</sale>) must be written out. To identify this, the tagger stores the ID of the current sales order and compares it with that of the next tuple. Hence, the storage needed by the tagger is proportional only to the level of nesting and is independent of the size of the XML document.

Chapter 4

NATIVE XML DATABASES

4.1 What is a Native XML Database?

A native XML database is a database designed from the ground up for storing XML data, preserving all data structure with no mapping required for storage. All query operations, indexing, transactions, server logic and all other database operations are specifically tailored for the needs of working with XML data. Tamino and dbXML are native XML databases, Excelon, XHive and Ozone are object databases with an XML layer added, and Oracle and MS SQL server are, of course, relational databases with mapping layers.

Native XML databases and Object databases with XML layers will look quite similar to the user and administrator and can be considered roughly equivalent from that perspective. Relational databases, however, have the added burden of mapping, which complicates and slows down the process of working with XML data. Each approach has strengths and weaknesses. For many applications where legacy data is involved, the relational approach is probably better, but for applications where XML is a primary concern, a native XML database will generally provide an easier solution.

There are several reasons to use existing database types, and existing database products, to store XML even if it is not in its native form. First, relational and object-oriented databases are well known, while native XML databases are in their infancy. Second, as a result of the familiarity with relational and object-oriented databases, users understand their behavior, especially in regards to performance. There is an unwillingness to move to a native XML database whose characteristics – especially scalability – have not been tested relatively as much. Finally, relational and object-oriented databases are reliable choices in the corporate mind. It falls into the old "nobody ever got fired for buying X" rationale.

On the other hand, there are several criticisms of the use of relational databases to store XML. For example, one of the attractive features XML possesses is its hierarchical organization, which database tables eliminate. As shown in the previous chapter, relational databases must map XML to relational tables and, therefore, flatten XML structures into rows and columns each time data is needed. Uche Ogbuji, principal consultant at Fourthought Inc. in Boulder, Colo., says XML is a mismatch with relational databases. "You can do tricky joins associating XML type to a database row to make them work, but they're hard to maintain," he says.

Relational databases cannot handle data with dynamic structure, which is the key to XML's extensibility. An XML database must be able to store and retrieve any well-formed XML document, even if the DTD or XML Schema of the document is not available. An RDBMS, however, needs schema definitions for each table (so a document with an unknown tag would require a change request for a new schema definition) to be built and approved before it can be put into production.

Additionally, translating XML to and from the database requires substantial processing, especially for large or complex documents. This performance factor may be most inconvenient when dealing with one of XML's strong points: producing Web pages from format-independent content. The problem is that the resulting pages may not load quickly enough. Frequently, a client requires a certain relational database to be used, regardless of its appropriateness to the task. In such cases, Ogbuji says he prefers placing a wrapper around the relational database to handle the XML translation. Though a feasible option, there is a lot of overhead in such an approach.

One way around these difficulties is to store the data in a native XML database. This immediately eliminates the need for translation between XML and the database. A new breed of such native XML databases is now emerging. Databases tailored for the storage of XML data represent an exciting new opportunity for improvement in the storage and manipulation of data and metadata. For a large set of applications, an XML database will often far surpass traditional data storage mechanisms in convenience, ease of development, and performance.

A number of applications exist that are ideally more suited for XML databases. Examples of such applications include:

- Corporate information portals
- Membership databases
- Product Catalogs
- Parts Databases
- Patient Information tracking
- Business-to-Business document exchange

Since native XML databases represent a new technology, there has been (up to this point) no concerted effort to develop specifications specifically for the market. This lack of specifications inevitably increases the learning curve for employees, prevents product interoperability and eventually slows the adoption of the products in the market place. To tackle these issues, a decision was made to start the XML:DB initiative, which hopes to bring standards to the XML database industry and to make XML databases the standard toolset used by IT departments worldwide.

Formally defined, a native XML database is a database whose internal data structures map directly onto the hierarchical format of XML [22]. Users of a native XML database would not be encouraged to distinguish between some external "interchange" format and an internal "efficient" format, nor to design applications that distinguish "business data" from "document content." In a native XML database, such distinctions are meaningless. What is appealing to people is the way working with XML allows one to think, the tools it allows one to use, the flexibility that it gives over how to manage data, and the ease with which data can be exchanged between entities. It is also appealing because XML is a tool of the Internet age and therefore, by extension, so are XML databases. Native XML databases use schemas and Document Type Definitions (DTDs) to describe, store and locate data. As a result, some are claiming it runs faster than the relational databases can.

4.2 What Makes An XML Database Native?

An XML database is a collection of XML documents and their parts, maintained by a system having the means to administer and control the collection itself and the information represented by that collection. It is more than merely a repository of structured documents or semi-structured data. As is true for the administration of other forms of data, management of persistent XML data requires capabilities to deal with data independence, integration, access rights, versions, views, integrity, redundancy, consistency, recovery, and enforcement of standards. Even for many applications in which XML is used as a transient data exchange format, there remains the need for persistent storage in XML form to preserve the communications between different parties in the form understood and agreed to by the parties.

The XML:DB initiative (http://www.xmldb.org) has defined three classes of XML database system, supporting native XML databases (designed to store and manipulate XML documents), XML-enabled databases (providing XML interfaces to other forms of stored data), and hybrid XML databases (accessible through XML and other interfaces), and there are a variety of database system prototypes and products in each of these classes.

For an XML database, one essential semantic issue is document equivalence: when are two documents or document parts the same? This question is important in satisfying requirements for evidence and archiving, for version management, for metadata management, and (as is true of all forms of data) for query optimization.

The XML 1.0 specification defines the components of XML documents, partitioning them into logical structures ("declarations, elements, comments, character references, and processing instructions, all of which are indicated in the document by explicit markup") and physical structures (entities, which may include entity references). The text stored within these structures may correspond to character data, markup, white space, or end-of-line markers. The specification is clear in stating which of these components must be provided to an application by an XML processor and which text representations must be considered to be identical by all XML processors. It is expected that all XML applications will "view" documents in terms of these components.

It is very naive to assume that general-purpose XML database systems can be built using models that ignore the documents' structure, just as it is wrong to use models that ignore the enterprise data represented by those documents' content. In general, reconstruction of an original XML document from the data stored in an XML database must be possible.

4.3 Native XML DBMS vs. RDBMS and OODBMS

Relational DBMSs hold a small fraction of the world's data, for several good reasons. They tend to require professional administration. They require data to be tabular and to correspond to a previously stated schema, which promotes integrity but discourages rapid development and change for irregular data or data whose structure change rapidly. For document data, semi-structured data models offer a promise of addressing all but the first objection. But for now, they lack the features needed for robust systems.

As semi-structured data becomes more widely shared, and is processed more automatically, organizations will need data management abilities over this data, such as powerful queries, integrity, updates, and versioning [33]. XML data can be stored directly in relational systems (by encoding its graph), but relational operators are inadequate for the operations users want. Object-oriented database vendors have even begun addressing this need by extending their capabilities to support XML. Relational systems are also moving in this direction. For efficiency, these products will often use highly tuned indexed structures, rather than just store XML as text.

XML is a universal convention for describing information that not only structures data according to formal criteria, but also according to content features. Because XML is so flexible, it can be used for a great number of purposes. In addition, XML blurs the distinction between structured data and unstructured documents because XML can describe text documents, as well as fields in traditional relational database tables. Only database engines that make native use of XML can use its complete functional range.

The long-term goal for database support of XML and other semi-structured data may be best seen in the database research community. Researchers are addressing the challenges [16] of interfacing to semi-structured data, and of managing that data. Other projects are studying the use of graph-structured data models (such as that which underlies XML) as a common representation for heterogeneous information sources, including both structured and semi-structured sources.

Compared with an ordinary relational or object database, semi-structured databases offer several capabilities [33]:

1. **Irregular structure.** Document data is frequently quite variable in structure, especially if constructed from multiple sources. Relational systems can model some irregularity by having missing attributes as nulls. However, SQL is complex with null values, and current storage structures can have excessive overhead.

2. **Tag and path operations.** Conventional database languages allow manipulation of element values, but not element names. Semi-structured databases provide operators that test tag names (i.e., "find all authors that have a FirstName element"). They also include operators that manipulate paths. For example, one can have path expressions with wild cards, to ask for a FirstName element at any depth within a Book element.

3. **Hierarchical model.** Some data is most naturally modeled as a hierarchy. For this data, hierarchical languages simplify data manipulation.

4. **Sequence**. Unlike tables, document sections are ordered, so sequence must be represented. In query processing, especially joins and updates, sequence introduces significant complexity.

These features have been demonstrated in research samples and are likely to appear in commercial products in the next few years. However, it is still uncertain how the market will be split among the three approaches: (1) layered over an object database, (2) layered over a relational database, or (3) directly over some new data manager.

For applications involving regularly structured data, XML tools will not replace such databases. There is too much functionality to implement quickly and migration would be too traumatic. Still, XML is rapidly gaining a role for even highly structured data as an interface format.

Whenever required (i.e., to publish the information on the Web, or send it over the wire for e-commerce), XML versions of appropriate views of the data can be created. Sometimes, these will be created on the fly, in response to user queries, while for other applications (especially where the data is nonvolatile or users don't need completely current information) the XML may be created in advance. Already, some vendors (i.e. Oracle and IBM) have released tools for creating XML extracts of databases, and these tools can possibly become more powerful. Import utilities are also being customized to accept XML.

There are several benefits of publishing database contents as XML. First, the XML output includes its own schema information. So, for anyone who understands the tags used, the information is self-describing. Also, XML reduces the need for multi-site systems to migrate to a new interface all at once. With XML Schema, one can keep part of the format open. If new tags are inserted, those sites equipped to use the new information can do so, while parsers for other sites will ignore it.

4.4 TAMINO

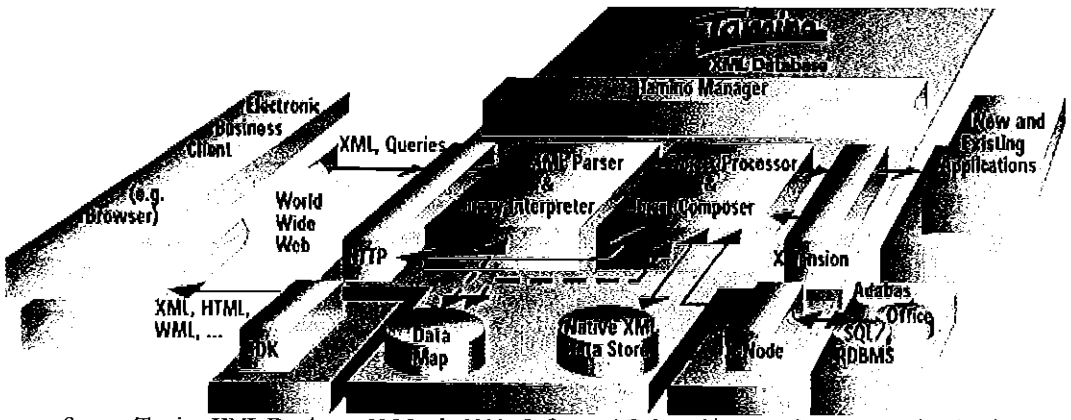
While relational databases can create a context for data through tables, columns, joins, etc., they work best with data that fits into this structure. Once the data has been extracted from the database, its meaning completely relies on the processing applications that process it further. In complex environments, this often leads to problems that are difficult to fix, such as unexpected application behavior, lack of scalability, and maintainability. If data is sent outside the enterprise, the context is entirely unpredictable.

That has led Software AG to design a virtual database management system called Tamino, which includes a high-performance native-XML database. Supporting all kinds of data types, this combination makes up a scalable architecture for the following: Storage Publishing and Exchange of Electronic Documents

Based on a small and fast XML engine that is able to natively process XML, Tamino XML Database is the first virtual DBMS that permits direct storage, integration, and exchange of XML data. This guarantees high performance and scalability since an extra layer for data conversion to and from XML is not needed. Moreover, unexpected changes in the format of a data stream, which is a key feature of XML, can be processed based on the embedded metadata. Tamino delivers XML information with great performance for transaction-oriented applications within enterprises or on the Web. Tamino can also integrate data from existing databases into XML structures.

Tamino (which is an acronym for "Transaction Architecture for the Management of INternet Objects") is the world's first native XML Information Server. It is a complete Web-enabled data management system for data exchange and application integration and is a technology that can turn enterprise data into Internet objects. Tamino establishes a highly reliable, scalable and open environment, extending enterprise transaction logic to the Internet. Tamino's XML technologies, which include *Data Map*, *X-Port*, *X-Node*, etc., allow data scattered over the enterprise (or even between business partners) to be connected.

Tamino incorporates an XML engine on top of an integrated native-XML data store. Furthermore, Tamino XML Database provides virtual DBMS capabilities. The product enables users to view all integrated external heterogeneous data sources by making their content available to the XML world in XML format and in real-time. This means that Tamino provides integrated access to existing legacy data residing in external data sources (i.e. Relational DBMS or data created by Office applications). The architecture for Tamino XML Database is displayed in Figure 5.



Source: Tamino XML Database. 20 March, 2001. Software AG (http://www.softwareag.com/tamino/)

Figure 5 Tamino XML Database Architecture

The main functional components of this architecture are as follows:

- **XML Engine** ➔ A high performance, robust server that allows Tamino to store XML objects natively in and retrieve them from a native-XML data store and various other data sources.
- **Tamino X-Node** ➔ An integration component that allows access to existing heterogeneous databases with traditional data structures, regardless of database type or location
- **Data Map** ➔ The knowledge base, which contains XML metadata, such as DTDs (document type definitions), style sheets, relational schemas, etc., defining the rules according to which XML objects are stored and composed.
- **Tamino Manager** ➔ An administration tool implemented as a client-server application that is integrated into the System Management Hub, which is Software AG's multi-platform environment for the unified management of Software AG products.
- **Tamino X-Tension** ➔ Server extensions that are user-defined function plug-ins of the Tamino XML Database server which handles data in some specific way that cannot be anticipated by a standard function provided by Tamino

Tamino uses XML as its primary means for structuring, organizing and storing information. The advantages of XML-based storage over traditional databases are compelling. Administration effort is considerably reduced as well because Tamino can easily structure any well-formed XML document and create a structural definition (DTD), even if none exists. Modification of information structure can be done on the fly. Sophisticated and powerful full-text queries can be executed with XQuery, Tamino's XPath-based query mechanism for document retrieval. Query parameters cause Tamino to apply the rules from the appropriate schema in the data map, thus extracting information from the source and returning it as XML.

4.5 XML Query Languages

To have a native XML database, this implies that there must be some way of querying the XML data. Several languages have been constructed in hopes of efficiently querying data from XML documents. For the scope of this paper, three such query technologies will be examined: XQuery, XML-QL, and XSLT.

4.5.1 XQuery

As already explained, XML is capable of labeling the information content of diverse data sources, including structured and semi-structured documents, relational databases, and object repositories. A query language that uses the structure of XML wisely can express queries across all these kinds of data, whether the data is physically stored in XML or viewed as XML via middleware. Because query languages have typically been designed for specific kinds of data, most existing proposals for XML query languages are robust for particular types of data sources but weak for other types. Among existent XML query languages is a new query language called XQuery, which is designed to be broadly applicable across all types of XML data sources. This is the newest current technology, as the W3C just released its Working Draft for XQuery on the 15th of February, 2001.

XQuery is designed to be a small, easy to implement language in which queries are concise and easily understood. It is also flexible enough to query a broad range of XML information sources, including both databases and documents. The Query Working Group has identified a requirement for both a human-readable query syntax and an XML-based query syntax. XQuery is designed to meet the first of these two requirements. An alternative, XML-based syntax for the XQuery semantics will be defined separately.

Important issues remain open in the design of XQuery. Some of these issues deal with relationships between XQuery and other XML activities, for example:

- The semantics of XQuery are defined in terms of the operators of the XML Query Algebra.
- The type system of XQuery is the type system of XML Schema. Work is in progress to ensure that the type systems of XQuery, the XML Query Algebra, and XML Schema are all completely aligned.
- XQuery relies on path expressions for navigating in hierarchic documents. It expects these expressions to conform to the semantics of XPath, mentioned in Chapter 2.

XQuery is a functional language in which a query is represented as an expression. XQuery supports several kinds of expressions. Hence, its queries may take numerous different forms. The various forms of XQuery expressions can be nested with full generality, so the notion of a "sub-query" is natural to XQuery. The input and output of a query are instances of a data model called the XML Query Data Model. This data model is a refinement of the data model described in the XPath specification, in which a document is modeled as a tree of nodes. . The following is an example query, which looks in the second chapter of the document named "zoo.xml" and finds the figure(s) with the caption "Tree Frogs."

```
document("zoo.xml")/chapter[2]//figure[caption = "Tree Frogs"]
```

XQuery provides a core library of built-in functions for use in queries. One of these core functions is used in the above example (*document*), which returns the root node of a named document. The XQuery core function library contains all the functions of the XPath core function library, all the aggregation functions of SQL (such as *avg*, *sum*, *count*, *max*, and *min*), and a number of other useful functions. For example, the *distinct* function eliminates duplicates from a list, and the empty function returns TRUE only if its argument is an empty list

In addition to the built-in functions, XQuery allows users to define their own functions. Each function definition must declare the datatypes of its parameters and result. It must also provide an expression (the "body" of the function) that defines how the result of the function is computed from its parameters. When a function is called upon, its arguments must be valid instances of the declared parameter types. The result of a function must also be a valid instance of its declared type. These rules are checked using the type-inference rules of the XML Query Algebra.

4.5.2 XML-QL

XML-QL (developed by researchers from the University of Pennsylvania, AT&T Labs, INRIA, and the University of Washington and submitted to the W3C in August 1998) uses path expressions and patterns to extract data from the input XML data. It has variables to which this data is bound, and it has templates, which show how the output XML data is to be constructed. Both patterns and templates use the XML syntax. When restricted to relational data, XML-QL is as expressive as relational calculus or relational algebra. XML-QL can extract data from existing XML documents and construct new XML documents. It can also support both ordered and unordered views on an XML document.

XML-QL assumes a semi-structured data model in which data is represented as an edge-labeled graph. Variables in XML-QL are bound to nodes in the semi-structured data model. In terms of XML, this means that variables are bound to element content, not to elements.

The capabilities described for XML-QL are often similar to those provided by the XSL transformation and pattern languages. Both approaches are block structured and template oriented. Both offer the ability to return trees or graphs, create new elements in the output, and query XML. The biggest differences are syntactic. XSL uses a URL-like syntax for specifying patterns, queries, and the inherent XML document structure to set the limits of query blocks. XML-QL uses an XML-like query-by-example pattern to select data and explicit keywords to delimit blocks. On the other hand, although XML-QL shares some functionality with XSL, XML-QL supports more data-intensive operations, such as joins and aggregates, and has better support for constructing new XML data, which is required by transformations.

XML-QL uses element patterns to match data in an XML document. The following example is taken from AT&T's research web page for XML-QL. It produces all authors of books whose publisher is "Addison-Wesley" in the XML document "bib.xml." Any URI (uniform resource identifier) representing an XML-data source may appear on the right-hand side of IN.

```
WHERE <bib><book>
      <publisher><name>"Addison-Wesley"</name></publisher>
      <title> $t </title>
      <author> $a </author>
    </book></bib> IN "bib.xml"
CONSTRUCT $a
```

Informally put, this query matches every <book> element in the XML document bib.xml that has at least one <title> element, one <author> element, and one <publisher> element whose <name> element is equal to Addison-Wesley. For each such match, it binds the variables $t and $a to every title and author pair respectively. Note that variable names are preceded by $ to distinguish them from string literals in the XML document (i.e. Addison-Wesley).

With many query languages in existence, this provides XML developers different options for how to query XML documents or XML databases. This may also be a disadvantage since this means there is no standard way of querying XML. Though W3C has already established a working draft recommendation for XQuery, there are still those who use XML-QL to query XML data. Regardless, this illustrates the emergence of native XML databases in which XML data can be stored in its native form and can also be retrieved via an XML query language.

CONCLUSION

With the IT market ever-changing, new technologies emerge. XML has been around for a few years now, but it has only been until recently that it was realized how powerful XML storage in a database can be. With a few firms accepting this idea, along came a new idea of storing XML in its native form, as opposed to mapping it to the relational schema of tables and columns.

Relational databases have the advantage of being around for such a long time. With this time, relational technology has matured, and people understand the power it possesses. Relational databases are a great choice for a firm because people already understand that technology. Native XML database technology is still very much in its infancy. Storage in a native XML database does eliminate the overhead of mapping XML data to the database's schema and can operate at an impressive pace. However, with so much of the world's data safely nestled in relational databases, not many are eager to make the leap over to XML databases. Yet, the gradual change and acceptance of XML technology can be seen. The power of the XML language has been understood, which is why major database vendors are adding that extra XML mapping layer into their products.

In addition, many expect that the major database vendors will soon offer their own native XML databases in response to demand for XML processing that Web-based e-commerce applications will require. This will let information technology departments that must buy from specific vendors all they need, getting native XML functionality from their approved vendors.

Demand for XML will expand; new uses will include Internet search engines that use XML tags, e-commerce systems that must produce output rapidly, electronic data interchange with XML tags, data reuse and content personalization. The move to XML databases to handle such applications will proceed in turn. As XML standards are being developed further, the future of XML and its related technologies is very bright. Undoubtedly, XML will play a central role in data communication. To the extent of native XML databases' role in the future of IT, that remains to be seen, but things are looking very promising.

GLOSSARY

Attribute

XML structural construct. A name-value pair, separated by an equals sign, included inside a tagged element that modifies certain features of the element. All attribute values, including things like size and width, are in fact text strings and not numbers. For XML, all values must be enclosed in quotation marks.

Cascading Style Sheets (CSS)

Formatting descriptions that provide augmented control over presentation and layout of HTML and XML elements. See also **Extensible Stylesheet Language**.

Character data

All the text content of an element or attribute that is not markup. XML differentiates this plain text from binary data.

Document element

The element in an XML document that contains all other elements. It is the top-level element of an XML document and must be the first element in the document. There is exactly one document element, no part of which appears in the content of any other element. The terms root element and document element are interchangeable.

Document entity

The starting-point for an **XML parser**. Unlike other entities, the document entity has no name and cannot be referenced. It is the entity in which the XML declaration and document type declaration can occur.

Document Object Model (DOM)

A platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents. The Document Object Model provides a standard set of objects for representing HTML and XML documents, a standard model of how these objects can be combined, and a standard interface for accessing and manipulating them.

Document Type Declaration

XML structural construct. Consists of markup code that indicates the grammar rules, or **Document Type Definition (DTD)**, for the particular class of document. The document type declaration can also point to an external file that contains all or part of the DTD. It must appear following the XML declaration and preceding the **document element**. The syntax of the document type declaration is <!DOCTYPE content>.

Document Type Definition (DTD)

Can accompany a document, essentially defining the rules of the document, such as which elements are present and the structural relationship between the elements. It defines what tags can go in your document, what tags can contain other tags, the number and sequence of the tags, the attributes your tags can have, and optionally, the values those attributes can have. See also schema.

Element

XML structural construct. An XML element consists of a start tag, an end tag, and the information between the tags, which is often referred to as the contents. Elements used in an XML file are described by a **DTD** or **schema**, either of which can provide a description of the structure of the data.

Entity

XML structural construct. A file, database record, or another item that contains data. The primary purpose of an entity is to hold content—not structure, rules, or grammar. Each entity is identified by a unique name and contains its own content, from a single character inside the document to a large file that exists outside the document.

Extensible Linking Language (XLL)

An XML vocabulary that provides links in XML similar to those in HTML but with more functionality. In addition to offering URL-based hyperlinks and anchors, XLL also supports linking to an arbitrary position in a document and multidirectional links. page level.

Extensible Markup Language (XML)

A subset of **SGML** that is optimized for delivery over the Web, XML provides a uniform method for describing and exchanging structured data that is independent of applications or vendors.

Extensible Stylesheet Language (XSL)

A language used to transform XML-based data into HTML or other presentation formats, for display in a Web browser. The transformation of XML into formats, such as HTML, is done in a declarative way, making it often easier and more accessible than through scripting.

Invalid document

Documents that don't follow the XML tag rules. If a document has a DTD or schema, and it doesn't follow the rules defined in its DTD or schema, that document is invalid as well.

Mixed content

Element types with mixed content are allowed to hold either character data alone or character data interspersed with child elements.

Namespace

A mechanism that allows developers to uniquely qualify the element names and relationships and to make these names recognizable. By doing so, they can avoid name collisions on elements that have the same name but are defined in different vocabularies. They allow tags from multiple namespaces to be mixed, which is essential if data is coming from multiple sources.

Reference node

The reference node for a search context is the node that is the immediate parent of all nodes in the search context. Every search context has an associated reference node.

SAX

See **Simple API for XML**.

Schema

A formal specification of element names that indicates which elements are allowed in an XML document, and in what combinations. It also defines the structure of the document: which elements are child elements of others, the sequence in which the child elements can appear, and the number of child elements. A schema is functionally equivalent to a **DTD**, but is written in XML. A schema also provides for extended functionality such as data typing, inheritance, and presentation rules. Consequently, the new schema languages are far more powerful than DTDs.

SGML

See **Standard Generalized Markup Language**.

Simple API for XML (SAX)

An XML API that allows developers to take advantage of event-driven XML parsing. Unlike the DOM specification, SAX doesn't require the entire XML file to be loaded into memory.

Simple Object Access Protocol (SOAP)

Provides an open, extensible way for applications to communicate using XML-based messages over the Web, regardless of what operating system, object model, or language they use. SOAP provides a way to use the existing Internet infrastructure to enable applications to communicate directly with each other without being unintentionally blocked by firewalls.

Standard Generalized Markup Language (SGML)

The international standard for defining descriptions of structure and content of electronic documents. Despite its name, SGML is not a language in itself, but a way of defining languages that are developed along its general principles.

Valid XML

XML that conforms to the rules defined in the XML specification, as well as the rules defined in the **DTD** or **Schema**. Because all of this parsing and checking can take time and because validation might not always be necessary, XML supports the notion of the well-formed document.

W3C

See **Worldwide Web Consortium**.

Well-formed XML

XML that follows the XML tag rules listed in the W3C Recommendation for XML 1.0, but does not have a DTD or schema. Well-formed XML documents are easy to create because they don't require the additional work of creating a DTD. Well-formed XML can save download time because the client does not need to download the DTD, and it can save processing time because the **XML parser** doesn't need to process the DTD.

Worldwide Web Consortium (W3C)

A standards body physically located at MIT and virtually at that sets standards for XML, HTML, XSL, and many other Web technologies.

XLL

See **Extensible Linking Language**.

XML

See **Extensible Markup Language**.

XML-Data

A language used to create a **schema**, which identifies the structure and constraints of a particular XML document. XML-Data carries out the same basic tasks as **DTD**, but with more power and flexibility. Unlike **DTD**, which requires its own language and syntax, XML-Data uses XML syntax for its language.

XML document

A document object that is well formed, according to the XML recommendation, and that might (or might not) be valid.

XML parser

A software module used to read XML documents and provide access to their content and structure. The XML parser generates a hierarchically structured tree, then hands off data to viewers and other applications for processing, and finally returns the results to the browser. A validating XML parser also checks the XML syntax and reports errors.

XPath

The result of an effort to provide a common syntax and semantics for functionality shared between **XSL Transformations** (XSLT) and **XPointer**. The primary purpose of XPath is to address parts of an XML document. It also provides basic facilities for manipulation of strings, numbers and booleans.

XML Pointer Language (XPointer)

A **W3C** initiative that specifies constructs for addressing the internal structures of XML documents. In particular, it provides for specific reference to elements, character strings, and other parts of XML documents, whether or not they bear an explicit ID attribute.

XML Query Language (XQL)

An extension to the capabilities of **XSL** that will provide for searching into, and data retrieval from, XML documents. It provides ways to manipulate XML in order to create new documents, to control the content of existing documents, and to manage the ordering and presentation of these documents along with XSL.

XML Schema

See **schema**.

XSL Transformations (XSLT)

Provides two "hooks" for extending the language, one hook for extending the set of instruction elements used in templates and one hook for extending the set of functions used in **XPath** expressions. These hooks are both based on XML namespaces.

BIBLIOGRAPHY

[1] Quin, Liam. *Open Source XML Database Toolkit.* New York: Wiley Computer Publishing, 2000

[2] Sturm, Jake. *Developing XML Solutions.* Redmond, Washington: Microsoft Press, 2000

[3] Nakhimovsky, Alexander & Myers, Tom. *Professional Java XML Programming.* UK:Wrox Press, 1999

[4] Abiteboul, Serge, Peter Buneman, and Dan Suciu. *Data On The Web: From Relations to Semistructured Data and XML.* San Francisco, California: Morgan Kaufmann Publishers, 2000

[5] Bourret, Ronald. *XML Database Products.* Nov. 2000. http://www.rpbourret.com/xml/XMLDatabaseProds.htm

[6] Bourret, Ronald. *XML And Databases.* Nov. 2000 http://www.rpbourret.com/xml/XMLAndDatabases.htm

[7] Martin, Bryan. *An Introduction to the Extensible Markup Language (XML).* Jan. 2001 http://www.personal.u-net.com/~sgml/xmlintro.htm

[8] SpiderPro. *KickStart XML Tutorial.* Jan. 2001 http://www.spiderpro.com/bu/buxmlm001.html

[9] MSDN Online Library. *Microsoft XML 3.0 – XML Developer's Guide – XML Glossary.* Nov. 2000. http://msdn.microsoft.com/library/default.asp?URL=/library/psdk/xmlsdk/xmls6g53.htm

[10] MSDN Online Library. *Microsoft XML 3.0 - XML Tutorial.* Nov. 2000 http://msdn.microsoft.com/library/psdk/xmlsdk/xmlt7k18.htm

[11] Williams, Keven, et. al. *XML Structures for Existing Databases.* Jan. 2001 http://www-106.ibm.com/developerworks/library/x-struct/?dwzone=xml

[12] Tash, Jeff. *Differences Between an RDBMS And An XML Document.* Oct. 1999 http://www.planetit.com/techcenters/docs/enterprise_apps_systems-data_management/opinion/PIT19991012S0004

[13] Heinemann, Charles. *Creating XML Data Sources from Relational Databases.* Mar. 1998 http://msdn.microsoft.com/xml/articles/xml030998.asp

[14] ITER. *FAQ DBDOM.* Feb. 2001. http://www.iter.co.il/projects/dbdom/faq.html

[15] Ambler, Scott. *Mapping Objects To Relational Databases.* Oct. 2000 http://www.ambysoft.com/mappingObjects.pdf

[16] Widom, Jennifer. *Data Management for XML: Research Directions.* April 1999. http://www-db.stanford.edu/~widom/xml-whitepaper.html

[17] Shanmugasundaram, Jayavel, Kristin Tufte, Gang He, Chun Zhang, David DeWitt, and Jeffrey Naughton. *Relational Databases for Querying XML Documents: Limitations and Opportunities.* VLDB Conference, Sept. 1999

[18] Shanmugasundaram, Jayavel, Eugene Shekita, Rimon Barr, Michael Carey, Bruce Lindsay, Hamid Pirahesh, and Berthold Reinwald. *Efficiently Publishing Relational Data as XML Documents.* VLDB Conference, Sept. 2000

[19] Beauchemin, Bob. *The XML Files.* Sept. 2000 http://www.sqlmag.com/Articles/Index.cfm?ArticleID=9161&pg=2

[20] W3C. *XML Representation of a Relational Database.* July 1997 http://www.w3.org/XML/RDB.html

[21] Buck, Lee. *Modeling Relational Data in XML.* March 2001 http://www.extensibility.com/tibco/resources/modeling.htm

[22] Champion, Michael. *Native XML vs. XML-enabled: The Difference Makes a Difference.* March 2001. http://www.softwareag.com/xml/library/champion_nativexml.htm

[23] DeJong, Jennifer. *Storing XML Data.* Dec. 2000. http://www.sdtimes.com/news/019/special1.htm

[24] Software AG. *Tamino XML Database Specification.* March 2001 http://www.softwareag.com/tamino/

[25] Klemz, Janet. *A Birdseye View of XML.* Jan. 2001. http://mcs.uww.edu/mcsms/950-780/KlemzJM28/Research1.htm

[26] OASIS. *The XML Cover Pages.* March 2001. http://www.oasis-open.org/cover/

[27] W3C. *W3C Recommendation XML 1.0 Specification.* Oct. 2000. http://www.w3.org/TR/REC-xml

[28] Muench, Steve. *Using XML and Relational Databases for Internet Applications.* Jan. 2001 http://technet.oracle.com/tech/xml/info/htdocs/relational/index.htm

[29] Dejesus, Edmund X. *XML Enters the DBMS Arena.* Oct. 2000. http://www.computerworld.com/cwi/story/0,1199,NAV47_STO53026,00.html

[30] *XML:DB Initiative for XML Databases.* Jan. 2001. http://www.xmldb.org/index.html

[31] Chinwala, Maria, Rakesh Malhotra, and John A. Miller. *W3C Progress Towards Standards For XML Databases.* March 2001.

[32] Rosenthal, Aaron, Len Seligman, and Roger Costello. *XML, Databases, and Interoperability.* Federal Database Colloquium, AFCEA, San Diego, 1999

[33] Seligman, Len and A. Rosenthal. *The Impact of XML on Databases and Data Sharing.* Dec. 2000

[34] Deutsch, Alin, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. *A Query Language for XML.* 1999. http://www.research.att.com/~mff/files/final.html

[35] W3C. *XQuery: A Query Langue for XML.* W3C Working Draft. Feb. 2001. http://www.w3.org/TR/xquery/

[36] W3C. *XML Path Language (XPath) Version 1.0.* W3C Recommendation. Nov. 1999. http://www.w3.org/TR/xpath

[37] W3C. *XML Pointer Language (XPointer) Version 1.0.* W3C Last Call Working Draft. Jan. 2001. http://www.w3.org/TR/xptr

[38] W3C. *XML Linking Language (XLink) Version 1.0.* W3C Proposed Recommendation. Dec. 2000. http://www.w3.org/TR/xlink/

[39] W3C. *XSL Transformations (XSLT) Version 1.1.* W3C Working Draft. Dec. 2000. http://www.w3.org/TR/xslt11/

[40] Holmon, G. Ken. *What is XSLT?* Aug. 2000. http://www.xml.com/pub/a/2000/08/holman/index.html