



<http://www.progressivebusinesstechnologytraining.com/>

XML Best Practices for Microsoft SQL Server 2005

Shankar Pal, Vishesh Parikh, Vasili Zolotov, Leo Giakoumakis, Michael Rys
Microsoft Corporation

April 2004
Revised April 2007

Applies to:
Microsoft SQL Server 2005

Summary: Learn about the guidelines for XML data modeling and usage in Microsoft SQL Server 2005, and see illustrative examples. To get the most from this article, you should have a basic understanding of XML features in SQL Server; for background material, see [XML Support in Microsoft SQL Server 2005](#) on the Microsoft Developer Network.

Contents

[Introduction](#)
[Data Modeling](#)
[Data Modeling Using XML Data Type](#)
[Usage](#)
[Data Binding in Queries](#)
[Catalog Views for Native XML Support](#)

Introduction

Microsoft SQL Server 2000 and SQLXML Web Releases provide powerful XML data management capabilities. These features focus on the mapping between relational and XML data. XML views of relational data can be defined using annotated XSD (AXSD) to provide an XML-centric approach that supports bulk load of XML data, and query and update capabilities on XML data. Transact-SQL extensions provide SQL-centric approach for mapping relational query results to XML (using FOR XML) and generating relational views from XML (using OpenXML).

Microsoft SQL Server 2005 provides extensive support for XML data processing. XML values can be stored natively in an XML data type column, which can be typed according to a collection of XML schemas, or left untyped. You can index the XML column. Furthermore, fine-grained data manipulation is supported using XQuery and XML DML, the latter being an extension for data modification.

In addition, the SQLXML, FOR XML and OpenXML features have been extended in SQL Server 2005. Together with the newly added native XML support, SQL Server 2005 provides a powerful platform for developing rich applications for semi-structured and unstructured data management.

With all the added functionality, the users have more design choices for their data storage and application development. This article provides guidelines for XML data modeling and usage in SQL Server 2005. It is divided into the following two topics:

- *Data modeling*

XML data can be stored in multiple ways in SQL Server 2005—for example, using native XML data type and XML shredded into tables. This topic provides guidelines for making the appropriate choices for modeling your XML data. It also covers indexing of XML data, property promotion, and typing of XML instances.

- *Usage*

This topic discusses usage-related topics, such as loading XML data into the server and type inference during query compilation, explains and differentiates closely-related features, and suggests appropriate use of these features. The ideas are illustrated with examples.

Data Modeling

This section outlines reasons for using XML in SQL Server 2005, provides guidelines for choosing between native XML storage and XML view technology, and gives data modeling suggestions.

Relational or XML Data Model

If your data is highly structured with known schema, relational model is likely to work the best for data storage. Microsoft SQL Server provides the necessary functionality and the tools you may need. On the other hand, if the structure is flexible (semi-structured or unstructured) or unknown, you have to give due considerations to modeling such data.

XML is a good choice if you want a platform-independent model to ensure portability of the data using structural and semantic markup. Furthermore, it is an appropriate option if some of the following properties are satisfied:

- Your data is sparse, you do not know the structure of the data, or the structure of your data may change significantly in the future.
- Your data represents containment hierarchy (as opposed to references among entities) and may be recursive.
- Order of markups and values is inherent in your data.
- You want to query into the data or update parts of it based on its structure.

If none of these conditions are met, you should use relational data model. For example, if your data is in XML format but your application merely uses the database to store and retrieve the data, an [n]varchar(max) column is all you need.

Storing the data in an XML column brings additional benefits; the engine checks that the data is well-formed or valid according to a prespecified XML schema, and supports fine-grained query and update of the XML data.

Reasons for Storing XML Data in SQL Server 2005

Here are some reasons for using native XML features in SQL Server 2005 as opposed to managing your XML data in the file system:

- You want to use administrative functionality of the database server for managing your XML data (for example, backup, recovery and replication).
- You want to share, query, and modify your XML data in an efficient and transacted way. Fine-grained data access is important to your application. For example, you may want to extract some of the sections within an XML document, or you may want to insert a new section without replacing your whole document.
- You have relational data and SQL applications, and you want interoperability between relational and XML data within your application. You need language support for query and data-modification for cross-domain applications.
- You want the server to guarantee well-formed data, and optionally validate your data according to XML schemas.
- You want indexing of XML data for efficient query processing and good scalability, and the use a first-rate query optimizer.
- You want SOAP, ADO.NET and OLE DB accesses to the XML data.

If none of these conditions are satisfied, you may be better off storing your data as a non-XML, large object type, such as [n]varchar(max) or varbinary(max).

XML Storage Options

The storage options for XML in SQL Server 2005 are as follows:

- Native storage as XML data type:

The data is stored in an internal representation that preserves the XML content of the data, such as containment hierarchy, document order, element and attribute values, and so on. Specifically, the InfoSet content of the XML data is preserved (for more information

on InfoSet, see <http://www.w3.org/TR/xml-infoset>). It may not be an exact copy of the text XML, since the following information is not retained: insignificant white spaces, order of attributes, namespace prefixes, and XML declaration.

For typed XML data type (that is, XML data type bound to XML schemas), the type-relevant information of the post-schema validation Infoset (PSVI), which adds type information to the Infoset, is encoded in the internal representation. This improves parsing speed significantly. (For more information, see the W3C XML Schema specifications at <http://www.w3.org/TR/xmlschema-1> and <http://www.w3.org/TR/xmlschema-2> and the XQuery 1.0 and XPath 2.0 Data Model working draft at <http://www.w3.org/TR/2005/WD-xpath-datamodel-20050211>.)

- Mapping between XML and relational storage:

Using annotated schema (AXSD), the XML is decomposed into columns in one or more tables, preserving fidelity of the data at the relational level—hierarchical structure is preserved, while order among elements is ignored. The schema cannot be recursive.

- Large object storage ([n]varchar(max) and varbinary(max)):

An exact copy of the data is stored. This is useful for special-purpose applications such as legal documents. Most applications do not require an exact copy, and are satisfied with the XML content (Infoset fidelity).

In general, you may need to use a combination of these approaches. For example, you may want to store your XML data in an XML data type column and promote properties from it into relational columns. Or, you may want to use mapping technology and store non-recursive parts in non-XML columns and only the recursive parts in XML data type columns.

Choice of XML Technology

The choice of XML technology (native XML versus XML view) generally depends upon the following factors:

- Storage options:

Your XML data may be more suitable for large object storage (for example, a product manual), or more amenable to storage in relational columns (for example, line items converted to XML). Each storage option preserves document fidelity to a different extent.

- Query capabilities:

You may find one storage option more suitable than others, based on the nature of your queries and on the extent to which you query your XML data. Fine-grained query of your XML data—for example, predicate evaluation on XML nodes—is supported to varying degrees in the two storage options.

- Indexing XML data:

You may want to index the XML data to speed up XML query performance. Indexing options vary with the storage options; you need to make the appropriate choices to optimize your workload.

- Data modification capabilities:

Some workloads involve fine-grained modification of XML data (for example, adding a new section within a document), while others do not (for example, Web content). Data modification language support may be important for your application.

- Schema support:

Your XML data may be described by a schema, which may or may not be an XML schema document. The support for schema-bound XML depends upon the XML technology.

Needless to say, different choices have different performance characteristics.

Native XML Storage

You can store your XML data in an XML data type column at the server. This is a suitable choice if:

- You want a straightforward way of storing your XML data at the server while preserving document order and document structure.
- You may or may not have a schema for your XML data.
- You want to query and modify your XML data.
- You want to index the XML data for faster query processing.
- Your application needs system catalog views to administer your XML data and XML schemas.

Native XML storage is useful when you have XML documents with a wide range of structures, or XML documents conforming to different or complex schemas that are too hard to map to relational structures.

Example: Modeling XML Data Using XML Data Type

Consider a product manual in XML format, consisting of a separate chapter for each topic, and multiple sections within each chapter. A section can contain sub-sections, so that <section> is a recursive element. Product manuals contain plenty of mixed content, diagrams and technical material; the data is semi-structured. Users may want to perform contextual search for topics of interest (for example, the section on "clustered index" within the chapter on "indexing"), and query technical quantities.

A suitable storage model for your XML documents is an XML data type column. This preserves the Infoset content of your XML data. Indexing the XML column benefits query performance.

Example: Retaining Exact Copies of XML Data

Suppose government regulations require you to retain exact textual copies of your XML documents (for example, signed documents, legal documents, or stock transaction orders). You may want to store your documents in an [n]varchar(max) column.

For querying, convert the data to XML data type at runtime and execute XQuery on it. The runtime conversion may be expensive especially when the document is large. If you query often, you can redundantly store the documents in an XML data type column and index it, while you return exact document copies from the [n]varchar(max) column.

The XML column may be a computed column based on the [n]varchar(max) column. However, you cannot create an XML index on a computed, XML column, nor can an XML index be built on [n]varchar(max) or varbinary(max) columns.

XML View Technology

By defining a mapping between your XML schemas and the tables in your database, you create an "XML view" of your persistent data. XML bulk load can be used to populate the underlying tables using the XML view. You can query the XML view using XPath 1.0; the query is translated into SQL queries on the tables. Similarly, updates are propagated to those tables as well.

This technology is useful when:

- You want to have an XML-centric programming model using XML views over your existing relational data.
- You have a schema (XSD, XDR) for your XML data, which an external partner may have provided.
- Order is not important in your data, your queryable data is not recursive, or the maximal recursion depth is known in advance.
- You want to query and modify the data through the XML view using XPath 1.0.
- You want to bulk-load XML data and decompose them into the underlying tables using the XML view.

Examples include relational data exposed as XML for data exchange and web services, and XML data with fixed schema. For more information, see <http://msdn.microsoft.com/SQLXML>.

You can also publish XML from relational and XML data stored at the server using FOR XML. For more information, refer to [Using FOR XML to Generate XML from Rowsets](#) in this article.

Example: Modeling Data Using Annotated XML Schema (AXSD)

Suppose you have existing relational data (for example, customers, orders, and line items) that you would like to manipulate as XML. Define an XML view using AXSD over the relational data. The XML view allows you to bulk-load XML data into your tables, and query and update the relational data using the XML view. This model is useful if you need to exchange data with XML markup with other applications while your SQL applications work uninterrupted.

Hybrid Model

Quite often, a combination of relational and XML data type columns is appropriate for data modeling. Some of the values from your XML data can be stored in relational columns, and the rest, or the entire XML value, stored in an XML column. This may yield better performance (for example, you have full control over the indexes created on the relational columns) and locking characteristics. However, you undertake greater responsibility for managing your data storage.

The values to store in relational columns depend on your workload. For example, if you retrieve entire XML values based on the path expression `/Customer/@CustId`, then promoting the value of the **CustId** attribute into a relational column and indexing it may yield faster query performance. On the other hand, if your XML data is extensively and non-redundantly decomposed into relational columns, the reassembly cost may be significant.

For highly structured XML data (for example, the content of a table has been converted into XML), you can map all values to relational columns, possibly using XML view technology.

Data Modeling Using XML Data Type

This section discusses data modeling topics for native XML storage. These include indexing XML data, property promotion, and typed XML data type.

Same or Different Table

An XML data type column can be created in a table containing other relational columns, or in a separate table with a foreign key relationship to a main table.

Create an XML data type column in the same table when one of the following conditions is true:

- Your application performs data retrieval on the XML column and does not require an XML index on the XML column, or.
- You want to build an XML index on the XML data type column, and the primary key of the main table is the same as its clustering key. See the section on [Indexing an XML Data Type Column](#) for more details.

Create the XML data type column in a separate table if the following conditions are true:

- You want to build an XML index on the XML data type column, but the primary key of the main table is not the same as its clustering key, or the main table does not have a primary key, or the main table is a heap (that is, no clustering key). This may be true if the main table already exists.

- You do not want table scans to slow down due to the presence of the XML column in the table, which takes up space whether it is stored in-row or out-of-row.

Granularity of XML Data

The granularity of the XML data stored in an XML column is critical for locking and update characteristic. SQL Server employs the same locking mechanism for both XML and non-XML data. Thus, row-level locking causes all XML instances in the row to be locked. When the granularity is large, locking large XML instances for updates causes throughput to decline in a multiuser scenario. To improve concurrency in a high update scenario, the XML data can be shredded into relational rows in one or more tables. Severe decomposition of this kind may lose object encapsulation and the structure of the XML data, and raises reassembly cost.

Updates to an XML instance are performed in-place and incrementally—in other words, without replacing the entire XML instance in most cases. Thus, updating the value of a single attribute is efficient and fairly independent of the size of the XML instance.

A balance between data modeling requirements and locking characteristics is important for good design.

Untyped, Typed, and Constrained XML Data Type

The SQL Server 2005 XML data type implements the ISO SQL-2003 standard XML data type. As such, it can store well-formed XML 1.0 documents as well as so-called XML content fragments with text nodes and an arbitrary number of top-level elements in an *untyped* XML column. The system checks for the well-formedness of the data, does not require the column to be bound to XML schemas, and rejects data that is not well-formed in the extended sense. This is true also of untyped XML variables and parameters.

If you have XML schemas describing your XML data, you can associate the schemas with the XML column to yield *typed* XML. The XML schemas are used to validate the data, perform more precise type checks during compilation of query and data modification statements than untyped XML, and optimize storage and query processing.

Use untyped XML data type under the following conditions:

- You do not have a schema for your XML data.
- You have schemas but you do not want the server to validate the data. This is sometimes the case when an application performs client-side validation before storing the data at the server, or temporarily stores XML data invalid according to the schema, or uses XML schema features not supported at the server (for example, `key/keyref`).

Use typed XML data type under the following conditions:

- You have schemas for your XML data and you want the server to validate your XML data according on the XML schemas.

- You want to take advantage of storage and query optimizations based on type information.
- You want to take better advantage of type information during compilation of your queries such as static type errors.

Typed XML columns, parameters and variables can store XML documents or content, which you have to specify as a flag (DOCUMENT or CONTENT, respectively) at the time of declaration. Furthermore, you have to provide one or more XML schemas. Specify DOCUMENT if each XML instance has exactly one top-level element; otherwise, use CONTENT. The query compiler uses DOCUMENT flag in type checks during query compilation to infer singleton top-level elements.

In addition to typing an XML column, you can use relational (column or row) constraints on typed or untyped XML data type columns. Use constraints under the following conditions:

- Your business rules cannot be expressed in XML schemas. For example, the delivery address of a flower shop must be within 50 miles of its business location, which can be written as a constraint on the XML column. The constraint may involve XML data type methods within scalar (as opposed to table-valued) user-defined functions.
- Your constraint involves other XML or non-XML columns in the table. An example is the enforcement of the ID of a Customer (/Customer/@CustId) found in an XML instance to match the value in a relational CustomerID column.

Document Type Definition (DTD)

XML data type columns, variables, and parameters can be typed using XML schema, but not using DTD. You can convert DTDs to XML schema documents using third-party tools, and load the XML schemas into the database.

Inline DTD can be used for both untyped and typed XML instances to supply default values and to replace entity references with their expanded form.

Internal Storage of XML Data

The XML data supplied by a user is stored internally in a binary format, which can be parsed faster than the textual representation of the XML data. This binary format yields some compression in the general case, and is limited to 2GB per instance. The amount of compression depends upon the length and number of repeating tags, and the types of the values occurring in the XML data. The following example shows how the size of the stored XML data can be computed.

Example: Computing Stored XML Size

We use table docs (pk INT PRIMARY KEY, xCol XML) with an untyped XML column in most of our examples; these can be extended to typed XML in a straightforward way (see SQL Server 2005 Books Online for information on the use of typed XML).

```
CREATE TABLE docs (pk INT PRIMARY KEY, xCol XML)
```

For ease of exposition, queries are described for XML data instances such as the following:

```
INSERT INTO docs VALUES (1,
'<book genre="security" publicationdate="2002" ISBN="0-7356-1588-2">
  <title>Writing Secure Code</title>
  <author>
    <first-name>Michael</first-name>
    <last-name>Howard</last-name>
  </author>
  <author>
    <first-name>David</first-name>
    <last-name>LeBlanc</last-name>
  </author>
  <price>39.99</price>
</book>' )
```

The stored size in bytes of the XML instances in the XML column can be found using the **DATALENGTH()** function:

```
SELECT DATALENGTH (xCol)
FROM docs
```

In-Row and Out-of-Row Storage

Small XML data type instances are stored within the rows of a table. Larger values that cannot be accommodated within a disk page are stored out of row with an in-row pointer of 16 bytes.

Storing XML values in-row reduces the record density and slows down table scans over the non-XML columns in the table. In such cases, the "large value types out of row" option can be specified in the system stored procedure **sp_tableoption** to store all large data types off-row.

Indexing an XML Data Type Column

XML indexes can be created on XML data type columns. It indexes all tags, values and paths over the XML instances in the column and benefits query performance. Your application may benefit from an XML index under the following conditions:

- Queries on XML columns are common in your workload. XML index maintenance cost during data modification must be taken into account.
- Your XML values are relatively large and the retrieved parts are relatively small. Building the index avoids parsing the whole data at runtime and benefits index lookups for efficient query processing.

The first index on an XML column is the "primary XML index". Using it, three types of secondary XML indexes can be created on the XML column to speed up common classes of queries, as described below.

Primary XML Index

This indexes all tags, values and paths within the XML instances in an XML column. The base table (that is, the table in which the XML column occurs) must have a clustered index on the primary key of the table; the primary key is used to correlate index rows with the rows in the base table. Full XML instances are retrieved from the XML columns (for example, `SELECT *`). Queries use the primary XML index, returning scalar values or XML subtrees using the index.

Example: Creating Primary XML Index

The following statement creates a primary XML index called `idx_xCol` on the XML column `xCol` of the table `docs`:

```
CREATE PRIMARY XML INDEX idx_xCol on docs (xCol)
```

Secondary XML Indexes

Once the primary XML index has been created, you may want to create secondary XML indexes to speed up different classes of queries within your workload. Three types of secondary XML indexes—`PATH`, `PROPERTY`, and `VALUE`—benefit path-based queries, custom property management scenarios, and value-based queries, respectively.

The `PATH` index builds a B+-tree on (path, value) pair of each XML node in document order over all XML instances in the column. The `PROPERTY` index creates a B+-tree clustered on the (PK, path, value) pair within each XML instance, where PK is the primary key of the base table. Finally, the `VALUE` index creates a B+-tree on (value, path) pair of each node in document order across all XML instances in the XML column.

Here are some guidelines for creating one or more of these indexes:

- If your workload uses path expressions heavily on XML columns, the `PATH` secondary XML index is likely to speed up your workload. The most common case is the use of `exist()` method on XML columns in `WHERE` clause of Transact-SQL.
- If your workload retrieves multiple values from individual XML instances using path expressions, clustering paths within each XML instance in the `PROPERTY` index may be helpful. This scenario typically occurs in a property bag scenario when properties of an object are fetched and its relational primary key value is known.
- If your workload involves querying for values within XML instances without knowing the element or attribute names that contain those values, you may want to create the `VALUE` index. This typically occurs with descendant axes lookups, such as `//author[last-name="Howard"]`, where `<author>` elements can occur at any level of the hierarchy and the search value ("Howard") is more selective than the path. It also occurs in "wildcard" queries, such as `/book [@* = "novel"]`, where the query looks for `<book>` elements with some attribute having the value "novel".

Example: Path-Based Lookup

Suppose the query below is common in your workload:

```
SELECT pk, xCol
FROM docs
WHERE xCol.exist ('/book[@genre = "security"]') = 1
```

The path expression `/book/@genre` and the value "security" correspond to the key fields of the PATH index. Consequently, secondary XML index of type PATH is helpful for this workload:

```
CREATE XML INDEX idx_xCol_Path on docs (xCol)
USING XML INDEX idx_xCol FOR PATH
```

Example: Fetching Properties of an Object

Consider the query below that retrieves the first names of authors of a book from each row in table docs:

```
SELECT ref.value ('first-name', 'nvarchar(64)'),
       ref.value ('last-name', 'nvarchar(64)')
FROM docs CROSS APPLY xCol.nodes ('/book/author') R(ref)
```

The property index is useful in this case and is created as follows:

```
CREATE XML INDEX idx_xCol_Property on docs (xCol)
USING XML INDEX idx_xCol FOR PROPERTY
```

Example: Value-Based Query

In the following query, a partial path is specified using `//`, so that the lookup based on the value of ISBN benefits from the use of the VALUE index:

```
SELECT xCol
FROM docs
WHERE xCol.exist ('//book/@ISBN[. = "0-7356-1588-2"]') = 1
```

The VALUE index is created as follows:

```
CREATE XML INDEX idx_xCol_Value on docs (xCol)
USING XML INDEX idx_xCol FOR VALUE
```

XML Index on Multiple File Groups

XML indexes are collocated with the base table; that is, XML index rows are stored in the same file groups and table partitions as the corresponding base table rows. This may sometimes require large file groups for XML blobs and their collocated XML indexes. The `TEXTIMAGE_ON <filegroup>` specification in the `CREATE TABLE` statement stores the XML blobs in the specified filegroup—the XML index rows are still collocated with the base table, while large XML node values are in the same file group as the XML blobs. This reduces the size of the individual file groups and provides more convenience for data management. For example,

when the non-XML data in the row is small relative to the size of the XML data, this technique can distribute the storage more evenly.

Full-Text Index on XML Column

You can create a full-text index on XML columns; this indexes the content of the XML values while ignoring the XML markup. Attribute values are not full-text indexed (since they are considered part of the markup) and element tags are used as token boundaries. You can combine full-text search with XML index usage in some scenarios:

- Filter the XML values of interest using SQL full-text search.
- Query those XML instances, which uses XML index on the XML column.

Example: Combining Full-Text Search with XML Querying

The steps for creating full-text index on an XML column are identical to those for other SQL type columns. The DDL statements are as follows, in which PK__docs__023D5A04 is the single-column primary key index of the table:

```
CREATE FULLTEXT CATALOG ft AS DEFAULT
CREATE FULLTEXT INDEX ON dbo.docs (xCol) KEY INDEX PK__docs__023D5A04
```

Once the full-text index has been created on the XML column, the following query checks that an XML instance contains the word "Secure" in the title of a book:

```
SELECT *
FROM docs
WHERE CONTAINS(xCol, 'Secure')
AND xCol.exist('/book/title/text()[contains(., "Secure")]') =1
```

The **CONTAINS()** method uses the full-text index to subset the XML instances that contain the word "Secure" anywhere in the document. The **exist()** clause ensures that the word "Secure" occurs in the title of a book.

Full-text search using **CONTAINS()** and XQuery **contains()** have different semantics. The latter is a substring match, while the former is a token match using stemming. Thus, if the search is for the string "run" in the title, then "run", "runs" and "running" all match, since both the full-text **CONTAINS()** and the XQuery **contains()** are satisfied. However, the query above does not match the word "UnSecured" in the title (the full-text **CONTAINS()** fails but the XQuery **contains()** is satisfied). Furthermore, full-text search employs word stemming, while XQuery **contains()** is a literal match. In general, for a pure substring match, the full-text **CONTAINS()** clause should be removed. This difference is illustrated in the next example.

Example: Full-Text Search on XML Values Using Stemming

The XQuery **contains()** check in [Example: Combining Full-Text Search with XML Querying](#) cannot be eliminated in general. Consider the query:

```
SELECT *
FROM docs
WHERE CONTAINS(xCol, 'run')
```

The word "ran" in the document matches the search condition owing to stemming. Furthermore, the search context is not checked using XQuery.

When XML is decomposed using AXSD into relational columns that are full-text indexed, XPath queries over the XML view do not perform full-text search on the underlying tables.

Support for Different Languages in Full-Text Index on XML Column

Unlike nvarchar or varchar columns that can have only one word breaker for the entire column, an XML data type column supports multiple language word breakers using the xml:lang attribute on XML elements. The word breaker for the specified language is used on the content of that element. A sub-element can specify a different language in an xml:lang attribute. Thus, not only can different XML instances but also a single XML instance can involve multiple word breakers.

This gives rise to interesting possibilities. For example, a Word 2003 document may contain sections in different languages. The document in WordML XML representation can be stored in an XML data type column, and the appropriate language word breakers are used for full-text indexing.

A full-text query can specify the language to use, as shown in the example below.

Example: Full-Text Search Specifying a Language

The query below specifies that the full-text search should be performed for the German language.

```
SELECT * FROM docs
WHERE contains (xCol, 'Visionen', LANGUAGE 'German')
```

Property Promotion

If queries are made principally on a small number of element and attribute values (for example, find customers based on customer ID—that is, the value of /Customer/@CustId is specified), you may want to promote those values into relational columns. This is helpful when queries are issued on a small part of the XML data while the entire XML instance is retrieved. Creating XML index on the XML column is overkill; instead, the promoted column can be indexed. Queries must be written to use the promoted column (that is, the query optimizer does not retarget queries on the XML column to the promoted column).

The promoted column can be a computed column in the same table or a separate, user-maintained column in a table. This is adequate when singleton values (that is, single-valued properties) are promoted from each XML instance. However, for multivalued properties, you have to create a separate table for the property, as described in the following section.

Computed Column Based on XML Data Type

A computed column can be created using a user-defined function (UDF) that invokes XML data type methods. The type of the computed column can be any SQL type, including XML. This is illustrated in the following example.

Example: Computed Column Based on XML Data Type Method

Create the user-defined function for ISBN of books:

```
CREATE FUNCTION udf_get_book_ISBN (@xData xml)
RETURNS varchar(20)
WITH SCHEMABINDING
BEGIN

    DECLARE @ISBN varchar(20)

    SELECT @ISBN = @xData.value('/book[1]/@ISBN', 'varchar(20)')

    RETURN @ISBN

END
```

Add a computed column to the table for ISBN:

```
ALTER TABLE docs
ADD ISBN AS dbo.udf_get_book_ISBN(xCol)
```

The computed column can be indexed in the usual way.

Example: Queries on Computed Column Based on XML Data Type Methods

To obtain the <book> whose ISBN is 0-7356-1588-2, the query:

```
SELECT pk, xCol
FROM docs
WHERE xCol.exist ('/book[@ISBN = "0-7356-1588-2"]') = 1
```

on the XML column can be rewritten to use the computed column as follows:

```
SELECT pk, xCol
FROM docs
WHERE ISBN = '0-7356-1588-2'
```

You can create a user-defined function to return XML data type and create a computed column using the UDF. However, you cannot create an XML index on the computed, XML column.

Creating Property Tables

You may want to promote some of the multivalued properties from your XML data into one or more tables, create indexes on those tables, and retarget your queries to use them. A typical scenario is one in which a small number of properties cover most of your query workload. You can do the following:

- Create one or more tables to hold the multivalued properties. You may find it convenient to store one property per table, and to duplicate the primary key of the base table in the property tables for back join with the base table.
- If you want to maintain the relative order of the properties, you need to introduce a separate column for the relative order.
- Create triggers on the XML column to maintain the property table(s). Within the triggers, do one of the following:
 - Use XML data type methods, such as **nodes()** and **value()**, to insert and delete rows of the property table(s). (See the section [value\(\), nodes\(\), and OpenXML\(\)](#) for more discussion of the **nodes()** method.)
 - Create streaming table-valued function(s) in CLR to insert and delete rows of the property table(s).
- Write queries for SQL access to the property tables and XML access to the XML column in the base table, with joins between the tables using their primary key.

Example: Create Property Table

Suppose you want to promote first name of authors. Books have one or more authors, so that first name is a multivalued property. Each first name is stored in a separate row of a property table. The primary key of the base table is duplicated in the property table for back join.

```
CREATE TABLE tblPropAuthor (propPK int, propAuthor varchar(max))
```

Example: Create User-Defined Function to Generate a Rowset from XML Instance

The table-valued user-defined function `udf_XML2Table` below accepts a primary key value and an XML instance. It retrieves the first name of all authors of `<book>` elements and returns a rowset of (primary key, first name) pairs. Indexing a computed column based on XML data type methods without using a wrapper user-defined function is not supported in SQL Server 2005.

```
CREATE FUNCTION udf_XML2Table (@pk int, @xCol xml)
RETURNS table WITH SCHEMABINDING
AS RETURN(
    select @pk as PropPK, nref.value('.', 'varchar(max)') as propAuthor
    from    @xCol.nodes('/book/author/first-name') R(nref)
)
```

Example: Create Triggers to Populate Property Table

Insert trigger—Inserts rows into the property table:

```

CREATE TRIGGER trg_docs_INS on docs FOR INSERT
AS
BEGIN
    insert into tblPropAuthor
        select p.*
        from inserted as I CROSS APPLY
            dbo.udf_XML2Table(I.pk, I.xCol) as P
END

```

Delete trigger—Deletes rows from the property table based on the primary key value of deleted rows:

```

create trigger trg_docs_DEL on docs for delete
as
begin
    delete tblPropAuthor where propPK IN
        (select p.PropPK
         from deleted as D CROSS APPLY
             dbo.udf_XML2Table(D.pk, D.xCol) as P
        )
end

```

Update trigger—Deletes existing rows in property table corresponding to the updated XML instance and inserts new rows into property table:

```

create trigger trg_docs_UPD
on docs
for update
as
if update(xCol) or update(pk)
begin
    delete tblPropAuthor where propPK IN
        (select p.PropPK
         from deleted as D CROSS APPLY
             dbo.udf_XML2Table(D.pk, D.xCol) as P
        )

    insert into tblPropAuthor
        select p.*
        from inserted as I CROSS APPLY
            dbo.udf_XML2Table(i.pk, i.xCol) as P
end

```

Example: Find XML Instances Whose Authors Have the First Name "David"

The query can be formulated on the XML column. Alternatively, it can search the property table for first name "David" and perform a back join with the base table to return the XML instance, as shown here:

```

SELECT xCol
FROM docs JOIN tblPropAuthor ON docs.pk = tblPropAuthor.propPK
WHERE tblPropAuthor.propAuthor = 'David'

```

Example: Solution Using CLR Streaming Table-Valued Function

This solution consists of the following steps:

1. Define a CLR class CXmlStreamingTVF that implements IEnumerator and contains a method InitMethod to generate a streaming table-valued output by applying a simple path expression on an XML instance.
2. Create an assembly and a Transact-SQL user-defined function (UDF) to invoke the CLR class.
3. Define insert, update and delete triggers using the UDF to maintain the property table(s).

First, create the streaming CLR function shown below. XML data type is exposed as a managed class SqlXml in ADO.NET; it supports the method **CreateReader()** that returns an XmlReader:

```
using System;
using System.Xml;
using System.IO;
using System.Data;
using System.Data.Sql;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.Collections;

public class CXmlStreamingTVF : IEnumerator {
    private XmlReader m_reader;
    private SqlXml m_doc;
    private string m_name;
    private string[] m_path;
    private int m_pathLoc;

    public CXmlStreamingTVF (SqlXml doc, string simplePath) {
        m_doc = doc;
        m_reader = m_doc.CreateReader();
        m_path = simplePath.Split(new char[]{'/'});
        m_pathLoc = m_path.Length-1;
    }

    //Three IEnumerator methods.
    //Custom code for Navigating the document for a simple path.
    public bool MoveNext () {
        bool new_row = false;
        while (!new_row && !m_reader.EOF) {
            m_reader.Read();
            if (m_reader.LocalName==m_path[m_pathLoc] &&
                m_pathLoc==m_path.Length-1 &&
                m_reader.NodeType==XmlNodeType.Element) {
                m_name = m_reader.ReadString();
                new_row = true;
            }
            else if (m_reader.LocalName==m_path[m_pathLoc] &&
                m_reader.NodeType==XmlNodeType.Element &&
                m_reader.IsEmptyElement==false) {
                if (m_pathLoc==1 && m_reader.Depth!=0)
                    continue;
            }
        }
    }
}
```

```

        m_pathLoc++;
    }
    else if (m_pathLoc!=1 &&
        m_reader.LocalName==m_path[m_pathLoc-1] &&
        m_reader.NodeType==XmlNodeType.EndElement) {
        m_pathLoc--;
    }
}
return new_row;
}

public object Current { get { return this; } }
public void Reset () {
    m_reader.Close();
    m_reader = m_doc.CreateReader();
}

[SqlFunctionAttribute (FillRowMethodName="CLROpenXml")]
public static IEnumerable InitMethod (SqlXml doc, string simplePath)
{
    return new CXmlStreamingTVF(doc, simplePath);
}

public static void CLROpenXml(Object obj, out string name) {
    CXmlStreamingTVF stream = (CXmlStreamingTVF) obj;
    name = stream.m_name;
}
}

```

Next, create an assembly and a Transact-SQL user-defined function **SQL_streaming_xml_tvf** corresponding to the CLR method **InitMethod()**.

```

CREATE ASSEMBLY CLRXML
FROM 'C:\temp\StreamingTVF.dll'
WITH PERMISSION_SET = SAFE

CREATE FUNCTION SQL_streaming_xml_tvf (
    @xData XML, @xPath nvarchar(max))
RETURNS table (FirstName nvarchar(max))
AS
EXTERNAL NAME [CLRXML].[CXmlStreamingTVF].[InitMethod]

```

The UDF is used to define the table-valued function **CLR_udf_XML2Table** for rowset generation:

```

create function CLR_udf_XML2Table (@pk int, @xCol xml)
returns @ret_Table table (FK int, FirstName varchar(max))
with schemabinding
as
begin
    insert into @ret_Table
    select @pk, FirstName
    FROM    SQL_streaming_xml_tvf (@xCol, '/book/author/first-name')
    return
end

```

Finally, define triggers as in "Example: Create Triggers to Populate Property Table" with the function CLR_udf_XML2Table replacing udf_XML2Table. Thus, the insert trigger is as follows:

```
create trigger CLR_trg_docs_INS on docs for insert
as
begin
    insert into tblPropAuthor
        select p.*
        from    inserted as I CROSS APPLY
            dbo.CLR_udf_XML2Table(I.pk, I.xCol) as P
end
```

The delete and the update triggers are similar to the non-CLR ones and are obtained by merely replacing the function **udf_XML2Table()** with **CLR_udf_XML2Table()**.

Pros and Cons of These Two Alternatives

When the function **udf_XML2Table()** used to generate, remove and modify the rows in the property table is CPU intensive, the CLR-based approach is generally faster. This includes XML data with very complex structure so that the XML parsing is computationally expensive. When the evaluation of the function **udf_XML2Table()** is cheap, the difference diminishes. For small XML sizes and simple path expressions, the cost of context switching may hurt the CLR-based solution more.

Unlike the Transact-SQL and XQuery-based solution, the path expression in the CLR-based solution is hard-coded. This works well as long as the path expressions are known ahead of time. In all other cases, the Transact-SQL and XQuery-based solutions are the only viable ones.

XML Schema Collections

An XML schema collection is a meta-data entity, scoped by a relational schema, which contains one or more XML schemas that may be related (for example, through <xs:import>) or unrelated. Individual XML schemas within an XML schema collection are identified using their target namespace.

An XML schema collection is created using CREATE XML SCHEMA COLLECTION syntax and providing one or more XML schemas. More XML schema components can be added to an existing XML schema, and more schemas can be added to an XML schema collection using ALTER XML SCHEMA COLLECTION syntax. XML schema collections can be secured like any SQL object using SQL Server 2005's security model.

Multityped Column

An XML schema collection C types an XML column xCol according to multiple XML schemas. Additionally, the flag DOCUMENT or CONTENT specifies whether XML trees or fragments, respectively, can be stored in column xCol.

For DOCUMENT, each XML instance specifies the target namespace of its top-level element in the instance, according to which it is validated and typed. For CONTENT, on the other hand, each top-level element can specify any one of the target namespaces in C. The XML instance is validated and typed according to all the target namespaces occurring in an instance.

Schema Evolution

XML schema collection is used to type XML columns, variables and parameters. It provides a mechanism for XML schema evolution. Suppose you add an XML schema with target namespace BOOK-V1 to an XML schema collection C. An XML column xCol typed using C can store XML data conforming to BOOK-V1 schema.

Suppose an application wants to extend the XML schema with new schema components, such as complex type definitions and top-level element declarations. These new schema components can be added to BOOK-V1 schema and do not require revalidation of the existing XML data in column xCol.

Suppose later the application wants to provide a new version of the XML schema, for which it chooses the target namespace BOOK-V2. This XML schema can be added to C. The XML column can store instances of both BOOK-V1 and BOOK-V2, and execute queries and data modification on XML instances conforming to these namespaces.

Lax Validation Disallowed in Wildcard Sections

The XML schema processor does not support lax validation in wildcard sections (xs:any and xs:anyAttribute) and xs:anyType. For wildcard sections, the XML schema can specify either processContents = "strict" or processContents = "skip". For xs:anyType, only strict validation is supported.

Strict validation ensures that more precise type information regarding the XML nodes instantiating these schema components is known during validation and used during query compilation. Skip semantics loses the typing information and the corresponding nodes are treated as untyped (xdt:untyped in the case of elements and xdt:untypedAtomic in the case of attributes).

If skip semantics for xs:anyType is desired, then introduce a new complex type that uses xs:any and xs:anyAttribute with processContents = "skip" as shown below:

```
<xs:complexType name="skipAnyType" mixed="true">
  <xs:sequence>
    <xs:any processContents="skip" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:anyAttribute processContents="skip"/>
</xs:complexType>
```

Using xs:datetime, xs:date, and xs:time

Values of type xs:datetime, xs:date, and xs:time must be specified in ISO 8601 format and include a time zone. Otherwise, the data validation for these values fails. Thus, 2005-05-27T14:11:00.943Z is valid as a value of type xs:datetime, but the following are not: 2005-05-27 14:11:00.943Z (missing date and time separator "T"), 2005-05-27T14:11:00.943 (missing time zone) and 2005-05-27 14:11:00.943 (missing time separator and time zone). Similarly, 2005-05-27Z is a valid xs:date value but 2005-05-27 is not since no time zone is specified.

Untyped XML data may contain date, time, and datetime values that an application may wish to convert to the SQL types dateTime or smallDateTime. These date, time and datetime values may not conform to ISO 8601 format or contain a time zone. Similarly, typed XML may contain such values as types other than xs:date, xs:time, and xs:dateTime (for example, xs:string). In both cases, the values should be converted first to [n]varchar and then to SQL datetime or smalldatetime, as the following example illustrates.

Example: Extracting datetime Value from Untyped XML

To obtain the value of the CreationTime attribute from the following data:

```
declare @var xml
select @var =
  '<QueryExecutionStats>
    <GeneralStats ExecutionCount="1"
      LastExecutionTime="2005-05-19 14:11:00.943"
      CreationTime="2005-05-19 14:11:00.913"/>
    <WorkerTime Total="3361" Last="3361" Min="3361" Max="3361"/>
    <PhysicalReads Total="0" Last="0" Min="0" Max="0"/>
    <PhysicalWrites Total="0" Last="0" Min="0" Max="0"/>
    <LogicalReads Total="0" Last="0" Min="0" Max="0"/>
  </QueryExecutionStats>'
```

a `value()` method is used to retrieve the value as nvarchar(64), which is then cast to SQL datetime type:

```
select cast (@var.value(
  '(/QueryExecutionStats /GeneralStats/@CreationTime)[1]',
  'nvarchar(max)') AS datetime) as creation_time
```

Usage

Loading XML Data

Transferring XML Data from SQL Server 2000 to SQL Server 2005

You can transfer XML data to SQL Server 2005 in multiple ways. We discuss a few options:

- If you have your data in an [n]text or image column in a SQL Server 2000 database, import the table using, say, DTS, into a SQL Server 2005 database. Change the column type to [n]varchar(max) or varbinary(max), respectively, and then to XML using ALTER TABLE statement.
- You can bulk-copy your data from SQL Server 2000 using bcp out, and bulk-insert into the SQL Server 2005 database using bcp in.
- If you have data in relational columns in a SQL Server 2000 database, create a new table with an [n]text column and optionally a primary key column for a row identifier. Use client side programming to retrieve XML generated at the server with FOR XML, and write it into the [n]text column. Then use the above-mentioned techniques to transfer data to a SQL Server 2005 database. You may choose to write the XML into an XML column in the SQL Server 2005 database directly.

Example: Changing Column Type to XML

Suppose you want to change the type of an [n]text, [n]varchar, varbinary, or XML column XYZ in table R to XML typed using the XML schema collection bookCollection. The following statement performs this type change:

```
ALTER TABLE R ALTER COLUMN XYZ XML (bookCollection)
```

The target is untyped XML if no XML schema collection is specified.

Text Encoding

SQL Server 2005 stores XML data in Unicode (UTF-16). XML data retrieved from the server comes out in UTF-16 encoding; if you want a different encoding for data retrieval, your application needs to perform the necessary conversion on the retrieved UTF-16 data.

When converting a string type to XML data type, SQL Server 2005 uses the code page of the collation of the source string to determine the encoding. If XML encoding information is specified using the "encoding" attribute in the XML declaration (for example, `<?xml version="1.0" encoding="windows-1256" ?>`), the encoding must be compatible with the string's code page. The string data can be parsed correctly by the XML parser as long as these two collations are compatible. Otherwise, an error may be raised or invalid data may be loaded. The same behavior also occurs when a client application sends a string value to the server for conversion to XML data type.

Sometimes, you may have XML data in different encodings, or have no advance knowledge about the encodings. The recommendation in such situations is to provide the XML data as a binary data type (for example, `varbinary(max)`). The server derives the encoding from the byte-order mark of the data stream (0xFFFE indicates UTF-16) or, if present, the XML declaration. Consequently, the easiest way of avoiding XML encoding mismatch in an XML parameter is to send the XML data from the client as native XML (using the `SqlXml` class in ADO.NET) or binary type, or to convert from [var]binary data type to XML at the server.

To summarize, the rules are:

- If your text XML is in Unicode (UCS-2, UTF-16), assigning it to an XML column, variable or parameter does not pose any problems.
- If the encoding is not Unicode and is implicit (due to the source code page), the string code page in the database should be the same as or compatible with the code points that you want to load (use COLLATE if necessary). If no such server code page exists, you have to add an explicit XML declaration to specify the proper encoding.
- To use an explicit encoding, either use varbinary type, which has no interaction with code pages, or use a string type of the appropriate code page. Then assign the data to XML column, variable or parameter.

Thus, if you want to pass UTF-8, it is safest to pass it in as varbinary(max). UTF-16 data can be passed in as nvarchar(max) where no byte order mark is required, or as varbinary(max) with the byte order mark 0xFFFE as the first two bytes to indicate UTF-16 encoding.

Bulk-Loading XML Data

You can bulk-load XML data into the server using SQL Server's bulk-loading capabilities, such as BCP, OPENROWSET, and BULK INSERT. OPENROWSET allows you to load data into an XML column from files. The following example illustrates this point.

Example: Loading XML from Files

This example shows how to insert a row in table docs. The value of the XML column is loaded from file C:\temp\xmlfile.xml as binary LOB (BLOB), and the pk column is supplied the value 10. The file is loaded as a BLOB (instead of a CLOB or NCLOB) to accept any encoding that the XML document may be encoded in.

```
INSERT INTO docs
SELECT 10, xCol
FROM (SELECT *
      FROM OPENROWSET (BULK 'C:\temp\xmlfile.xml', SINGLE_BLOB)
      AS xCol) AS R(xCol)
```

Non-Binary Collations

The XML collation used for XML data type is a binary collation and is case-sensitive (the so-called Unicode code point collation). Applications may have a different requirement, such as case insensitive searches. This can be achieved by promoting the appropriate string values into a computed column of type varchar with the appropriate collation. Query the computed column for collation-dependent operations. Furthermore, suppose the XML column contains German and Chinese data strings. You can use operations specific to each of these collations on two computed columns, one for each of these languages.

XQuery and Type Inference

XQuery (<http://www.w3.org/TR/xquery/>) embedded in Transact-SQL is the language supported for querying XML data type. The language is under development (currently in last call) by the

World Wide Web Consortium (W3C) with the participation of all major database vendors including Microsoft. It includes XPath 2.0 as navigation language. Language constructs for data modification are available on XML data type as well. See books online for information on the XQuery constructs, functions and operators supported in SQL Server 2005.

Error Model

Compilation errors are returned from syntactically incorrect XQuery expressions and XML DML statements. The compilation phase checks static type correctness of XQuery expressions and DML statements, and uses XML schemas for type inferences in case of typed XML. It raises static type errors if an expression could fail at runtime due to type safety violation. Examples of static error are addition of a string to an integer and querying for a non-existent node for typed data.

As a deviation from the W3C standard, XQuery runtime errors are converted into empty sequences, which may propagate as empty XML or NULL to the query result depending upon the invocation context.

Explicit casting to the proper type allows users to work around static errors although runtime cast errors will be transformed to empty sequences.

The following subsections discuss type checking in greater detail.

Singleton Checks

Location steps, function parameters, and operators (for example, eq) requiring singletons return an error if the compiler cannot determine whether a singleton is guaranteed at runtime. The problem arises often with untyped data and sometimes with typed data. For example, lookup of an attribute requires a singleton parent element; an ordinal selecting a single parent node is adequate. Evaluation of **nodes()-value()** combination (see the section [value\(\), nodes\(\), and OpenXML\(\)](#)) to extract attribute values may not require the ordinal specification, as shown in the next example, since the **nodes()** method emits singleton context items.

Example: Known Singleton

In this example, the **nodes()** method generates a separate row for each <book> element. (See the section [value\(\), nodes\(\), and OpenXML\(\)](#) for a more detailed description of the **nodes()** method.) The **value()** method evaluated on a <book> node extracts the value of @genre, which, being an attribute, is a singleton.

```
SELECT nref.value('@genre', 'varchar(max)') Genre
FROM docs CROSS APPLY xCol.nodes('//book') AS R(nref)
```

XML schema is used for type checking of typed XML. If a node is specified as singleton in the XML schema, the compiler uses that information and no error occurs. Otherwise, an ordinal selecting a single node is required. In particular, the use of descendant axis, such as in

`/book//title`, loses singleton cardinality inference for the `<title>` element even if the XML schema specifies it to be so. Rewrite it as `(/book//title)[1]`.

It is important to keep the distinction between `//first-name[1]` and `(//first-name)[1]` in mind for type checking. The former returns a sequence of `<first-name>` nodes in which each node is the leftmost `<first-name>` node amongst its siblings. The latter returns the first, singleton `<first-name>` node in document order in the XML instance.

Example: Use of `value()`

The query below on untyped XML column results in static, compilation error since `value()` expects a singleton node as the first argument and the compiler cannot determine whether only one `<last-name>` node will occur at runtime:

```
SELECT xCol.value('//author/last-name', 'nvarchar(50)') LastName
FROM docs
```

It is tempting to try the following fix:

```
SELECT xCol.value('//author/last-name[1]', 'nvarchar(50)') LastName
FROM docs
```

However, this does not rectify the error since multiple `<author>` nodes may occur in each XML instance. The following rewrite works:

```
SELECT xCol.value('(//author/last-name)[1]', 'nvarchar(50)') LastName
FROM docs
```

This query returns the value of the first `<last-name>` element in each XML instance.

Parent Axis

If the type of a node cannot be determined, it becomes `xs:anyType`, which is not implicitly cast to any other type. This occurs most notably during navigation using parent axis (for example, `xCol.query('/book/@genre/../../price')`); the parent node type is determined to be `xs:anyType`. An element may also be defined as `xs:anyType` in an XML schema. In both cases, the loss of more precise type information often leads to static type errors, and requires explicit cast of atomic values to their specific type.

data(), text(), and string() Accessors

XQuery has a function `fn:data()` to extract scalar, typed values from nodes, a node test `text()` to return text nodes, and the function `fn:string()` that returns the string value of a node. Their usages are sometimes confusing. Guidelines for their proper use in SQL Server 2005 are as follows. Consider the XML instance `<age>12</age>`.

- Untyped XML: The path expression `/age/text()` returns the text node "12". The function `fn:data(/age)` returns the string value "12" and so does `fn:string(/age)`.
- Typed XML: In SQL Server 2005, the expression `/age/text()` returns static error for any simple typed `<age>` element. On the other hand, `fn:data(/age)` returns integer 12, while `fn:string(/age)` yields the string "12".

Used within a `query()` method, the result of each of these functions and nodes tests are converted to text nodes and serialized as a single XML data type instance. Text nodes are visually represented by their string value, and the serialization of multiple text nodes appears as a concatenation of their string values. However, a search for the concatenated string or a part of it yields an empty result whenever it does not come from a single text node. This difference is illustrated in the example below.

Example: Serialization of Text Nodes

In the following query, the text nodes under all the `<author>` elements are retrieved within a `query()` method. There are four such text nodes in the example used in [Example: Creating Primary XML Index](#), and the serialized output appears as "MichaelHowardDavidLeBlanc" in SQL Server Management Studio.

```
SELECT xCol.query ('//author/*/text()') LastName
FROM docs
```

A search for the value "MichaelHowardDavidLeBlanc" in the query below returns an empty result since the search value does not equal that of any single text node under an `<author>` element:

```
SELECT xCol.query ('//author/*/text()[. = "MichaelHowardDavidLeBlanc" ]')
FROM docs
```

Functions and Operators over Union Types

Union types require careful handling owing to type checking. Two of the problems are illustrated in the following examples.

Example: Function over Union Type

Consider an element definition for `<r>` of a union type

```
<xs:element name="r">
  <xs:simpleType>
    <xs:union memberTypes="xs:int xs:float xs:double"/>
  </xs:simpleType>
</xs:element>
```

Within XQuery context, the "average" function `fn:avg (/r)` returns a static error since the XQuery compiler cannot sum values of different types (`xs:int`, `xs:float`, or `xs:double`) for the `<r>`

elements in the argument of **fn:avg()**. To get around this, rewrite the function invocation as `fn:avg(for $r in //r return xs:double ($r))`.

Example: Operator over Union Type

The addition operation '+' requires precise types of the operands, so that the expression `((//r)[1]) + 1` returns a static error with the above type definition for element <r>. One rewrite to fix the problem is `xs:int((//r)[1]) + 1`.

value(), nodes(), and OpenXML()

You can use multiple **value()** methods on XML data type in a **SELECT** clause to generate a rowset of extracted values. The **nodes()** method yields an internal reference for each selected node which can be used to query further. The **nodes()** method can operate over an XML column. The combination of **nodes()** and **value()** methods can be more efficient in generating the rowset when it has many columns and perhaps the path expressions used in its generation are complex.

The **nodes()** method yields instances of a special XML data type, each of which has its context set to a different selected node. Such an XML instance supports **query()**, **value()**, **nodes()** and **exist()** methods, and can be used in **count(*)** aggregations and IS NULL checks. All other uses result in error.

Example: Use of nodes()

Suppose you want to extract first and last names of authors, whose first name is not "David", as a rowset consisting of two columns, FirstName and LastName. Using **nodes()** and **value()** methods, you can achieve this as follows:

```
SELECT nref.value('first-name[1]', 'nvarchar(50)') FirstName,
       nref.value('last-name[1]', 'nvarchar(50)') LastName
FROM   docs CROSS APPLY xCol.nodes('//author') AS R(nref)
WHERE  nref.exist('.[first-name != "David"]') = 1
```

In this example, `nodes('//author')` yields a rowset of references to <author> elements for each XML instance. The first and last names of authors are obtained by evaluating **value()** methods relative to those references.

SQL Server 2000 provides a facility for generating a rowset from an XML instance using **OpenXml()**. You can specify the relational schema for the rowset and how values inside the XML instance map to columns in the rowset.

Example: Use of OpenXml() on XML Data Type

We can rewrite the query from the previous example using **OpenXml()** as shown below, by creating a cursor, reading each XML instance into an XML variable, and applying **OpenXML** to it:

```

DECLARE name_cursor CURSOR
FOR
    SELECT xCol
    FROM docs
OPEN name_cursor
DECLARE @xmlVal XML
DECLARE @idoc int
FETCH NEXT FROM name_cursor INTO @xmlVal

WHILE (@@FETCH_STATUS = 0)
BEGIN
    EXEC sp_xml_preparedocument @idoc OUTPUT, @xmlVal
    SELECT *
    FROM OPENXML (@idoc, '//author')
        WITH (FirstName varchar(50) 'first-name',
             LastName varchar(50) 'last-name') R
    WHERE R.FirstName != 'David'

    EXEC sp_xml_removedocument @idoc
    FETCH NEXT FROM name_cursor INTO @xmlVal
END
CLOSE name_cursor
DEALLOCATE name_cursor

```

OpenXml() creates an in-memory representation and uses work tables instead of the query processor. Its parsing procedure **sp_xml_preparedocument** requires a well-formed XML document and does not accept XML fragments. **OpenXML()** relies on the XPath 1.0 processor of MSXMLSQL, which is a private version of the MSXML 3.0 processor used by the database engine, instead of the XQuery engine. The work tables are not shared among multiple calls to **OpenXml()** even on the same XML instance. This limits its scalability. **OpenXml()** allows you to access an edge table format for the XML data when the **WITH** clause is not specified. Also, it allows you to use the remainder of the XML value in a separate, "overflow" column.

The combination of **nodes()** and **value()** functions use XML indexes effectively. Thus, this combination can exhibit greater scalability than **OpenXml**.

Example: Use of OpenXml on a Single XML Instance

OpenXml is often used to shred a single XML instance into a relational form, for example, when the XML data is received on the wire. In this case, no cursor is required. This example shows a stored procedure that accepts a single XML instance for shredding the XML the same way as that using the cursor example above.

```

CREATE PROCEDURE SHRED_SINGLE_XML @xmlVal nvarchar(max)
AS
BEGIN
    DECLARE @idoc INT
    EXEC sp_xml_preparedocument @idoc OUTPUT, @xmlVal
    SELECT *
    FROM OPENXML (@idoc, '//author')
        WITH (FirstName varchar(50) 'first-name',
             LastName varchar(50) 'last-name') R

```

```
WHERE R.FirstName != 'David'  
  
EXEC sp_xml_removedocument @idoc  
END
```

The stored procedure can be invoked as shown here:

```
DECLARE @xVal XML  
SET @xVal = (SELECT xCol FROM docs WHERE pk=1)  
EXEC SHRED_SINGLE_XML @xVal
```

Using FOR XML to Generate XML from Rowsets

You can generate an XML data type instance from a rowset using **FOR XML** with the new **TYPE** directive.

The result can be assigned to an XML data type column, variable or parameter. Furthermore, **FOR XML** can be nested to generate any hierarchical structure. This makes nested **FOR XML** much more convenient to write than **FOR XML EXPLICIT**, but it may not perform as well for deep hierarchies. **FOR XML** also introduces a new **PATH** mode that specifies the path in the XML tree where a column's value should appear.

The new **FOR XML TYPE** directive can be used to define read-only XML views over relational data with SQL syntax. The view can be queried with SQL statements and embedded XQuery, as the following example shows. For instance, you can refer to such SQL views in stored procedures. More information can be found in the MSDN article [What's New in FOR XML in Microsoft SQL Server 2005](#).

Example: SQL View Returning Generated XML Data Type

The following SQL view definition creates an XML view over a relational column (pk) and book authors retrieved from an XML column:

```
CREATE VIEW V (xmlVal) AS  
SELECT pk, xCol.query('/book/author')  
FROM docs  
FOR XML AUTO, TYPE
```

The view V contains a single row with a single column xmlVal of XML type. It can be queried like a regular XML data type instance. For example, the following query returns the author whose first name is "David":

```
SELECT xmlVal.query('//author[first-name = "David"]')  
FROM V
```

The query execution materializes the XML instance before executing the **query()** method on it. Hence, this approach does not perform or scale well except when the aggregated XML instance is small.

SQL view definitions are somewhat analogous to XML views created using annotated schemas. However, there are important differences. The SQL view definition is read-only and must be manipulated with embedded XQuery; not so for XML views using annotated schema. Furthermore, the SQL view materializes the XML result before applying the XQuery expression, while XPath queries on XML views evaluate SQL queries on the underlying tables.

Adding Business Logic

Your business logic can be added to XML data in several ways:

- You can write row or column constraints to enforce domain-specific constraints during insertion and modification of XML data. Constraints using XML data type methods are allowed only within a scalar user-defined function.
- You can write a trigger on the XML column that fires when you insert or update values in the column. The trigger can contain domain-specific validation rules or populate property tables.
- You can write SQLCLR functions in managed code to which you pass XML values, and use XML processing capabilities provided by System.Xml namespace. An example is to apply XSL transformation to XML data, as shown below. Alternatively, you can deserialize the XML into one or more managed classes and operate on them using managed code.
- You can write Transact-SQL stored procedures and functions that invoke processing on the XML column for your business needs.

Example: Applying XSL Transformation

Consider a CLR function TransformXml() that accepts an XML data type instance and an XSL transformation stored in a file, applies the transformation to the XML data and returns the transformed XML in the result. A skeleton function written in C# is as follows:

```
using System;
using System.Data.SqlTypes;
using System.Xml;
using System.Xml.XPath;
using System.Xml.Xsl;

public class TransformXml
{
    public static SqlXml ApplyXslTransform (SqlXml XmlData, string xslPath) {
        // Load XSL transformation
        XslCompiledTransform xform = new XslCompiledTransform();
        xform.Load (xslPath);

        // Load XML data
        XPathDocument xDoc = new XPathDocument (XmlData.CreateReader());
        XPathNavigator nav = xDoc.CreateNavigator ();

        // Apply the transformation
        // using makes sure that we flush the writer at the end
        using (XmlWriter writer = nav.AppendChild())
        {
            xform.Transform(XmlData.CreateReader(), writer);
        }
    }
}
```

```

// Return the transformed value
SqlXml retSqlXml = new SqlXml (nav.ReadSubtree());
return (retSqlXml);
}
}

```

Once the assembly is registered and a corresponding user-defined Transact-SQL function **SqlXslTransform()** corresponding to **ApplyXslTransform()** is created, the function can be invoked from Transact-SQL as in the following query:

```

SELECT SqlXslTransform (xCol, 'C:\temp\xsltransform.xml')
FROM docs
WHERE xCol.exist('/book/title/text()[contains(., "Secure")]') =1

```

The query result contains a rowset of the transformed XML.

SQLCLR opens up a whole new world that can be used for decomposing XML data into tables or property promotion, and querying XML data using managed classes in the System.Xml namespace. More information can be found in SQL Server 2005 and Visual Studio "Whidbey" books online.

Data Binding in Queries

When your data resides in a combination of relational and XML data type columns, you may want to write queries that combine relational and XML data processing. For example, you can convert the data in relational and XML columns into an XML data type instance using **FOR XML** and query it using XQuery. Conversely, you can generate a rowset from XML values (see [Usage](#)) and query it using Transact-SQL.

A more convenient and efficient way of writing cross-domain queries is to use the value of a SQL variable or column within XQuery or XML DML expressions:

- You can use **sql:variable()** to use the value of a SQL variable in your XQuery or XML DML expression.
- You can use **sql:column()** to use values from a relation column in your XQuery or XML DML expression.

This approach allows applications to parameterize queries, as shown in the example below. However, XML and user-defined type are not permitted in **sql:variable()** and **sql:column()**.

Example: Data Binding Using sql:variable()

The query below is a modified version of the one shown in [Example: Queries on Computed Column Based on XML Data Type Methods](#). In this version, the ISBN of interest is passed in using a SQL variable @isbn. By replacing the constant with **sql:variable()**, the query can be used to search for any ISBN, not just the one whose ISBN is 0-7356-1588-2.

```

DECLARE @isbn varchar(20)
SET      @isbn = '0-7356-1588-2'
SELECT  xCol
FROM    docs
WHERE   xCol.exist ('/book[@ISBN = sql:variable("@isbn")]') = 1

```

Sql:column() can be used in a similar way and provides additional benefits. Indexes over the column may be used for efficiency as decided by the cost-based query optimizer. Furthermore, the computed column may store a promoted property, as discussed in [Computed Column Based on XML Data Type](#).

Catalog Views for Native XML Support

Catalog views exist to provide meta-data information regarding XML usage. A few of these are discussed below.

XML Indexes

XML index entries appear in the catalog view `sys.indexes` with the index "type" 3. The "name" column contains the name of the XML index.

XML indexes are also recorded in the catalog view `sys.xml_indexes`, which contains all the columns of `sys.indexes` and a few special ones meaningful for XML indexes. The value NULL in the column "secondary_type" indicates a primary XML index; the values 'P', 'R' and 'V' stand for PATH, PROPERTY and VALUE secondary XML indexes, respectively.

Space usage of XML indexes can be found in the table-valued function **sys.dm_db_index_physical_stats()**. It provides information such as the number of disk pages occupied, average row size in bytes, number of records and other information for all index types, including XML indexes. This information is available for each database partition; XML indexes use the same partitioning scheme and partitioning function of the base table.

Example: Space Usage of XML Indexes

```

SELECT sum (page_count)
FROM    sys.dm_db_index_physical_stats (db_id(), object_id('docs'),
        DEFAULT, DEFAULT, 'DETAILED') SDPS
        JOIN sys.xml_indexes SXI ON (SXI.index_id = SDPS.index_id)
WHERE   SXI.name = 'idx_xCol_Path'

```

This yields the number of disk pages occupied by the XML index `idx_xCol_Path` in table `docs` across all partitions. Without the **sum()** function, the result would return the disk page usage per partition.

Retrieving XML Schema Collections

XML schema collections are enumerated in the catalog view `sys.xml_schema_collections`. The XML schema collection "sys" is defined by system and contains predefined namespaces that can be used in all user-defined XML schema collections without having to load them explicitly. This list contains the namespaces for xml, xs, xsi, fn, and xdt. Two other catalog views worth mentioning are: `sys.xml_schema_namespaces`, which enumerates all namespaces within each XML schema collection; and `sys.xml_schema_components`, which enumerates all XML schema components within each XML schema.

The built-in function `XML_SCHEMA_NAMESPACE(schemaName, XmlSchemacollectionName, namespace-uri)` yields an XML data type instance containing XML schema fragments for schemas contained in an XML schema collection, except for the predefined XML schemas.

You can enumerate the contents of an XML schema collection in the following ways:

- Write Transact-SQL queries on the appropriate catalog views for XML schema collections.
- Use the built-in function `XML_SCHEMA_NAMESPACE()`. You can apply XML data type methods on the output of this function. However, you cannot modify the underlying XML schemas.

These are illustrated in the examples that follow.

Example: Enumerate XML Namespaces in XML Schema Collection

Use the following query for XML schema collection "myCollection":

```
SELECT XSN.name
FROM sys.xml_schema_collections XSC JOIN sys.xml_schema_namespaces XSN
ON (XSC.xml_collection_id = XSN.xml_collection_id)
WHERE XSC.name = 'myCollection'
```

Example: Enumerate Contents of an XML Schema Collection

The following statement enumerates the contents of the XML schema collection "myCollection" within (relational) schema dbo.

```
SELECT XML_SCHEMA_NAMESPACE (N'dbo', N'myCollection')
```

Individual XML schemas within the collection can be obtained as XML data type instances by specifying the target namespace as the third argument to `XML_SCHEMA_NAMESPACE()`, as shown below.

Example: Output a Specified Schema from an XML Schema Collection

The following statement outputs the XML schema with target namespace "http://www.microsoft.com/books" from the XML schema collection "myCollection" within (relational) schema dbo.

```
SELECT XML_SCHEMA_NAMESPACE (N'dbo', N'myCollection',  
N'http://www.microsoft.com/books')
```

Querying XML Schemas

If you want to query XML schemas that you have loaded into XML schema collections, you may do so in the following ways:

- Write Transact-SQL queries on catalog views for XML schema namespaces.
- Create a table containing an XML data type column to store your XML schemas, in addition to loading them into the XML type system. You can query the XML column using the XML data type methods. Furthermore, you can build XML index on this column. However, maintaining consistency between the XML schemas stored in the XML column and the XML type system is left to the application. For example, if you drop the XML schema namespace from the XML type system, you have to drop it also from your table to preserve consistency.